e-maxx :: algo

Вас приветствует книга, собранная по материалам сайта e-maxx.ru/algo (по состоянию на 26 Apr 2012 1:48).

В этой книге Вы найдёте описание, реализации и доказательства множества алгоритмов, от известных всем до тех, которые являются уделом лучших олимпиадников и специалистов по Computer Science. В конце приведены ссылки на тематическую литературу, которую можно скачать с моего сайта, а также немного информации обо мне.

Приятного чтения!

Оглавление

Алгебра

элементарные алгоритмы

- Функция Эйлера и её вычисление
- Бинарное возведение в степень за O (log N)
- Алгоритм Евклида нахождения НОД (наибольшего общего делителя)
- Решето Эратосфена
- Расширенный алгоритм Евклида
- Числа Фибоначчи и их быстрое вычисление
- Обратный элемент в кольце по модулю
- Код Грея
- Длинная арифметика
- Дискретное логарифмирование по модулю M алгоритмом baby-step-giant-step Шэнкса за O (sqrt(M) log M)
- Диофантовы уравнения с двумя неизвестными: AX+BY=C
- Модульное линейное уравнение первого порядка: AX=B
- Китайская теорема об остатках. Алгоритм Гарнера
- Нахождение степени делителя факториала
- Троичная сбалансированная система счисления
- Вычисление факториала N! по модулю P за O (P log N)
- Перебор всех подмасок данной маски. Оценка 3^N для суммарного количества подмасок всех масок
- Первообразный корень. Алгоритм нахождения
- Дискретное извлечение корня
- Решето Эратосфена с линейным временем работы

сложные алгоритмы

- Тест BPSW на простоту чисел за O (log N)
- Эффективные алгоритмы факторизации: Полларда p-1, Полларда p, Бента, Полларда Монте-Карло, Ферма
- Быстрое преобразование Фурье за O (N log N). Применение к умножению двух полиномов или длинных чисел

Графы

элементарные алгоритмы

- Поиск в ширину
- Поиск в глубину
- Топологическая сортировка
- Поиск компонент связности

компоненты сильной связности, мосты и т.д.

- Поиск компонент сильной связности, построение конденсации графа за O (N + M)
- Поиск мостов за O (N + M)
- Поиск точек сочленения за O (N + M)

- Поиск мостов в режиме онлайн за O(1) в среднем

кратчайшие пути из одной вершины

- Алгоритм Дейкстры нахождения кратчайших путей от заданной вершины до всех остальных вершин за O (N^2 + M)
- Алгоритм Дейкстры для разреженного графа нахождения кратчайших путей от заданной вершины до всех остальных вершин за O (M log N)
- Алгоритм Форда-Беллмана нахождения кратчайших путей от заданной вершины до всех остальных вершин за O (N M)
- Алгоритм Левита нахождения кратчайших путей от заданной вершины до всех остальных вершин за O (N M)

кратчайшие пути между всеми парами вершин

- Нахождение кратчайших путей между всеми парами вершин графа методом Флойда-Уоршелла за O (n^3)
- Подсчёт количества путей фиксированной длины между всеми парами вершин, нахождение кратчайших путей фиксированной длины за O (n^3 log k)

минимальный остов

- Минимальное остовное дерево. Алгоритм Прима за O (n^2) и за O (m log n)
- Минимальное остовное дерево. Алгоритм Крускала за O (M log N + N^2)
- Минимальное остовное дерево. Алгоритм Крускала со структурой данных 'система непересекающихся множеств' за O (M log N)
- Матричная теорема Кирхгофа. Нахождение количества остовных деревьев за O (N^3)
- Код Прюфера. Формула Кэли. Количество способов сделать граф связным

циклы

- Нахождение отрицательного цикла в графе за O (N M)
- Нахождение Эйлерова пути или Эйлерова цикла за O (M)
- Проверка графа на ацикличность и нахождение цикла за O (M)

наименьший общий предок (LCA)

- Наименьший общий предок. Нахождение за O (sqrt (N)) и O (log N) с препроцессингом O (N)
- Наименьший общий предок. Нахождение за O (log N) с препроцессингом O (N log N) (метод двоичного подъёма)
- Наименьший общий предок. Нахождение за O (1) с препроцессингом O (N) (алгоритм Фарах-Колтона и Бендера)
- Задача RMQ (Range Minimum Query - минимум на отрезке). Решение за O (1) с препроцессингом O (N)
- Наименьший общий предок. Нахождение за O (1) в режиме оффлайн (алгоритм Тарьяна)

потоки и связанные с ними задачи

- Алгоритм Эдмондса-Карпа нахождения максимального потока за O (N M^2)
- Метод Проталкивания предпотока нахождения максимального потока за O (N^4)
- Модификация метода Проталкивания предпотока за O (N^3)
- Поток с ограничениями
- Поток минимальной стоимости (min-cost-flow). Алгоритм увеличивающих путей за O (N^3 M)
- Задача о назначениях. Решение с помощью min-cost-flow за O (N^5)
- Задача о назначениях. Венгерский алгоритм (алгоритм Куна) за O (N^4)
- Нахождение минимального разреза алгоритмом Штор-Вагнера за O(N^3)
- Поток минимальной стоимости, циркуляция минимальной стоимости. Алгоритм удаления циклов отрицательного веса
- Алгоритм Диница нахождения максимального потока

паросочетания и связанные с ними задачи

- Алгоритм Куна нахождения наибольшего паросочетания за O (N M)
- Проверка графа на двудольность и разбиение на две доли за O (M)
- Нахождение наибольшего по весу вершинно-взвешенного паросочетания за O (N^3)

- Алгоритм Эдмондса нахождения наибольшего паросочетания в произвольных графах за O (N^3)
- Покрытие путями ориентированного ациклического графа
- Матрица Татта. Рандомизированный алгоритм для поиска максимального паросочетания в произвольном графе

связность

- Рёберная связность. Свойства и нахождение
- Вершинная связность. Свойства и нахождение
- Построение графа с указанными величинами вершинной и рёберной связностей и наименьшей из степеней вершин

К-ые пути

обратные задачи

- Обратная задача SSSP (inverse-SSSP - обратная задача кратчайших путей из одной вершины) за O (M)
- Обратная задача MST (inverse-MST - обратная задача минимального остова) за O (N M^2)

разное

- Покраска рёбер дерева (структуры данных) - решение за O (log N)
- Задача 2-SAT (2-CNF). Решение за O (N + M)
- Heavy-light декомпозиция

Геометрия

элементарные алгоритмы

- Длина объединения отрезков на прямой за O (N log N)
- Знаковая площадь треугольника и предикат 'По часовой стрелке'
- Проверка двух отрезков на пересечение
- Нахождение уравнения прямой для отрезка
- Нахождение точки пересечения двух прямых
- Нахождение точки пересечения двух отрезков
- Нахождение площади простого многоугольника за O (N)
- Теорема Пика. Нахождение площади решётчатого многоугольника за O (1)
- Задача о покрытии отрезков точками
- Центры тяжести многоугольников и многогранников

более сложные алгоритмы

- Пересечение окружности и прямой
- Пересечение двух окружностей
- Построение выпуклой оболочки алгоритмом Грэхэма-Эндрю за O (N log N)
- Нахождение площади объединения треугольников. Метод вертикальной декомпозиции
- Проверка точки на принадлежность выпуклому многоугольнику за O (log N)
- Нахождение вписанной окружности в выпуклом многоугольнике с помощью тернарного поиска за O (N $\log^2$ C)
- Нахождение вписанной окружности в выпуклом многоугольнике методом сжатия сторон за O (N log N)
- Диаграмма Вороного в двумерном случае, её свойства, применение. Простейший алгоритм построения за O(N^4)
- Нахождение всех граней, внешней грани планарного графа за O (N log N)
- Нахождение пары ближайших точек алгоритмом разделяй-и-властвуй за O (N log N)
- Преобразование геометрической инверсии
- Поиск общих касательных к двум окружностям
- Поиск пары пересекающихся отрезков алгоритмом заметающей прямой за O (N log N)

Строки

- Z-фунция строки и её вычисление за O (N)
- Префикс-функция, её вычисление и применения. Алгоритм Кнута-Морриса-Пратта
- Алгоритмы хэширования в задачах на строки
- Алгоритм Рабина-Карпа поиска подстроки в строке за O (N)
- Разбор выражений за O (N). Обратная польская нотация
- Суффиксный массив. Построение за O (N log N) и применения
- Суффиксный автомат. Построение за O (N) и применения
- Нахождение всех подпалиндромов за O (N)
- Декомпозиция Линдона. Алгоритм Дюваля. Нахождение наименьшего циклического сдвига за O (N) времени и O (1) памяти
- Алгоритм Ахо-Корасик
- Суффиксное дерево. Алгоритм Укконена
- Поиск всех тандемных повторов в строке алгоритмом Мейна-Лоренца (разделяй-и-властвуй) за O (N log N)
- Поиск подстроки в строке с помощью Z- или Префикс-функции. Алгоритм Кнута-Морриса-Пратта
- Решение задачи "сжатие строки"
- Определение количества различных подстрок за O (N^2 log N)

Структуры данных

- Sqrt-декомпозиция
- Дерево Фенвика
- Система непересекающихся множеств
- Дерево отрезков
- Декартово дерево (treap, дерамида)
- Модификация стека и очереди для извлечения минимума за O (1)
- Рандомизированная куча

Алгоритмы на последовательностях

- Задача RMQ (Range Minimum Query - минимум на отрезке)
- Нахождение наидлиннейшей возрастающей подпоследовательности за O (N^2) и O (N log N)
- K-ая порядковая статистика за O (N)
- Нахождение наидлиннейшей возрастающей подпоследовательности за O (N^2) и O (N log N)

Динамика

- Динамика по профилю. Задача "паркет"
- Нахождение наибольшей нулевой подматрицы за O (N M)

Линейная алгебра

- Метод Гаусса решения системы линейных уравнений за O (N^3)

- Нахождение ранга матрицы за O (N^3)
- Вычисление определителя матрицы методом Гаусса за O (N^3)
- Вычисление определителя методом Краута за O (N^3)

Численные методы

- Интегрирование по формуле Симпсона
- Поиск корней методом Ньютона (касательных)
- Тернарный поиск

Комбинаторика

- Биномиальные коэффициенты
- Числа Каталана
- Ожерелья
- Расстановка слонов на шахматной доске
- Правильные скобочные последовательности. Нахождение лексикографически следующей, K-ой, определение номера
- Количество помеченных графов, связных помеченных графов, помеченных графов с K компонентами связности
- Генерация сочетаний из N элементов
- Лемма Бернсайда. Теорема Пойа
- Принцип включений-исключений

Теория игр

- Игры на произвольных графах. Метод ретроспективного анализа за O (M)
- Теория Шпрага-Гранди. Ним

Расписания

- Задача Джонсона с одним станком
- Задача Джонсона с двумя станками
- Оптимальный выбор заданий при известных временах завершения и длительностях выполнения

Разное

- Задача Иосифа
- Игра Пятнашки: существование решения
- Дерево Штерна-Броко. Ряд Фарея
- Поиск подотрезка массива с максимальной/минимальной суммой за O(N)

Приложение

- Литература
- Об авторе

Функция Эйлера

Определение

Функция Эйлера $\phi(n)$ (иногда обозначаемая $\varphi(n)$ или $phi(n)$) — это количество чисел от 1 до n, взаимно простых с n. Иными словами, это количество таких чисел в отрезке $[1; n]$, наибольший общий делитель которых с n равен единице.

Несколько первых значений этой функции (A000010 в энциклопедии OEIS):

$$\begin{aligned}
&\phi(1) = 1,\\
&\phi(2) = 1,\\
&\phi(3) = 2,\\
&\phi(4) = 2,\\
&\phi(5) = 4.
\end{aligned}$$

Свойства

Три следующих простых свойства функции Эйлера — достаточны, чтобы научиться вычислять её для любых чисел:

- Если p — простое число, то $\phi(p) = p - 1$.

  (Это очевидно, т.к. любое число, кроме самого p, взаимно просто с ним.)

- Если p — простое, a — натуральное число, то $\phi(p^a) = p^a - p^{a-1}$.

  (Поскольку с числом p^a не взаимно просты только числа вида $pk\,(k \in \mathcal{N})$, которых $p^a/p = p^{a-1}$ штук.)

- Если a и b взаимно простые, то $\phi(ab) = \phi(a)\phi(b)$ ("мультипликативность" функции Эйлера).

  (Этот факт следует из китайской теоремы об остатках. Рассмотрим произвольное число $z \le ab$. Обозначим через x и y остатки от деления z на a и b соответственно. Тогда z взаимно просто с ab тогда и только тогда, когда z взаимно просто с a и с b по отдельности, или, что то же самое, x взаимно просто с a и y взаимно просто с b. Применяя китайскую теорему об остатках, получаем, что любой паре чисел x и $y\,(x \le a,\ y \le b)$ взаимно однозначно соответствует число $z\,(z \le ab)$, что и завершает доказательство.)

Отсюда можно получить функцию Эйлера для любого n через его **факторизацию** (разложение n на простые сомножители):

если

$$n = p_1^{a_1} \cdot p_2^{a_2} \cdot \ldots \cdot p_k^{a_k}$$

(где все p_i — простые), то

$$\begin{aligned}
&\phi(n) = \phi(p_1^{a_1}) \cdot \phi(p_2^{a_2}) \cdot \ldots \cdot \phi(p_k^{a_k}) =\\
&= (p_1^{a_1} - p_1^{a_1-1}) \cdot (p_2^{a_2} - p_2^{a_2-1}) \cdot \ldots \cdot (p_k^{a_k} - p_k^{a_k-1}) =\\
&= n \cdot \left(1 - \frac{1}{p_1}\right) \cdot \left(1 - \frac{1}{p_2}\right) \cdot \ldots \cdot \left(1 - \frac{1}{p_k}\right).
\end{aligned}$$

Реализация

Простейший код, вычисляющий функцию Эйлера, факторизуя число элементарным методом за $O(\sqrt{n})$:

```
int phi (int n) {
        int result = n;
        for (int i=2; i*i<=n; ++i)
                if (n % i == 0) {
                        while (n % i == 0)
                                n /= i;
                        result -= result / i;
```

```
                }
        if (n > 1)
                result -= result / n;
        return result;
}
```

Ключевое место для вычисление функции Эйлера — это нахождение **факторизации** числа n. Его можно осуществить за время, значительно меньшее $O(\sqrt{n})$: см. Эффективные алгоритмы факторизации.

Приложения функции Эйлера

Самое известное и важное свойство функции Эйлера выражается в **теореме Эйлера**:

$$a^{\phi(m)} \equiv 1 \pmod m,$$

где a и m взаимно просты.

В частном случае, когда m простое, теорема Эйлера превращается в так называемую **малую теорему Ферма**:

$$a^{m-1} \equiv 1 \pmod m$$

Теорема Эйлера достаточно часто встречается в практических приложениях, например, см. Обратный элемент в поле по модулю.

Задачи в online judges

Список задач, в которых требуется посчитать функцию Эйлера,либо воспользоваться теоремой Эйлера, либо по значению функции Эйлера восстанавливать исходное число:

- UVA #10179 "Irreducible Basic Fractions" [сложность: низкая]
- UVA #10299 "Relatives" [сложность: низкая]
- UVA #11327 "Enumerating Rational Numbers" [сложность: средняя]
- TIMUS #1673 "**Допуск к экзамену**" [сложность: высокая]

Бинарное возведение в степень

Бинарное (двоичное) возведение в степень — это приём, позволяющий возводить любое число в n-ую степень за $O(\log n)$ умножений (вместо n умножений при обычном подходе).

Более того, описываемый здесь приём применим к любой **ассоциативной** операции, а не только к умножению чисел. Напомним, операция называется ассоциативной, если для любых a, b, c выполняется:

$$(a \cdot b) \cdot c = a \cdot (b \cdot c)$$

Наиболее очевидное обобщение — на остатки по некоторому модулю (очевидно, ассоциативность сохраняется). Следующим по "популярности" является обобщение на произведение матриц (его ассоциативность общеизвестна).

Алгоритм

Заметим, что для любого числа a и **чётного** числа n выполнимо очевидное тождество (следующее из ассоциативности операции умножения):

$$a^n = (a^{n/2})^2 = a^{n/2} \cdot a^{n/2}$$

Оно и является основным в методе бинарного возведения в степень. Действительно, для чётного n мы показали, как, потратив всего одну операцию умножения, можно свести задачу к вдвое меньшей степени.

Осталось понять, что делать, если степень n **нечётна**. Здесь мы поступаем очень просто: перейдём к степени $n - 1$, которая будет уже чётной:

$$a^n = a^{n-1} \cdot a$$

Итак, мы фактически нашли рекуррентную формулу: от степени n мы переходим, если она чётна, к $n/2$, а иначе — к $n - 1$. Понятно, что всего будет не более $2 \log n$ переходов, прежде чем мы придём к $n = 0$ (базе рекуррентной формулы). Таким образом, мы получили алгоритм, работающий за $O(\log n)$ умножений.

Реализация

Простейшая рекурсивная реализация:

```
int binpow (int a, int n) {
        if (n == 0)
                return 1;
        if (n % 2 == 1)
                return binpow (a, n-1) * a;
        else {
                int b = binpow (a, n/2);
                return b * b;
        }
}
```

Нерекурсивная реализация, оптимизированная (деления на 2 заменены битовыми операциями):

```
int binpow (int a, int n) {
        int res = 1;
        while (n)
                if (n & 1) {
                        res *= a;
                        --n;
                }
                else {
                        a *= a;
                        n >>= 1;
                }
        return res;
```

```
}
```

Эту реализацию можно ещё несколько упростить, заметив, что возведение a в квадрат осуществляется всегда, независимо от того, сработало условие нечётности n или нет:

```
int binpow (int a, int n) {
        int res = 1;
        while (n) {
                if (n & 1)
                        res *= a;
                a *= a;
                n >>= 1;
        }
        return res;
}
```

Наконец, стоит отметить, что бинарное возведение в степень уже реализовано в языке Java, но только для класса длинной арифметики BigInteger (функция pow этого класса работает именно по алгоритму бинарного возведения).

Алгоритм Евклида нахождения НОД (наибольшего общего делителя)

Даны два целых неотрицательных числа a и b. Требуется найти их наибольший общий делитель, т.е. наибольшее число, которое является делителем одновременно и a, и b. На английском языке "наибольший общий делитель" пишется "greatest common divisor", и распространённым его обозначением является $\gcd$:

$$\gcd(a, b) = \max_{k=1\ldots\infty\ :\ k|a\ \&\ k|b} k$$

(здесь символом "$|$" обозначена делимость, т.е. "$k|a$" обозначает "k делит a")

Когда оно из чисел равно нулю, а другое отлично от нуля, их наибольшим общим делителем, согласно определению, будет это второе число. Когда оба числа равны нулю, результат не определён (подойдёт любое бесконечно большое число), мы положим в этом случае наибольший общий делитель равным нулю. Поэтому можно говорить о таком правиле: если одно из чисел равно нулю, то их наибольший общий делитель равен второму числу.

Алгоритм Евклида, рассмотренный ниже, решает задачу нахождения наибольшего общего делителя двух чисел a и b за $O(\log \min(a, b))$.

Данный алгоритм был впервые описан в книге Евклида "Начала" (около 300 г. до н.э.), хотя, вполне возможно, этот алгоритм имеет более раннее происхождение.

Алгоритм

Сам алгоритм чрезвычайно прост и описывается следующей формулой:

$$\gcd(a, b) = \begin{cases} a, & \text{if b=0} \\ \gcd(b, a \bmod b), & \text{otherwise} \end{cases}$$

Реализация

```
int gcd (int a, int b) {
        if (b == 0)
                return a;
        else
                return gcd (b, a % b);
}
```

Используя тернарный условный оператор C++, алгоритм можно записать ещё короче:

```
int gcd (int a, int b) {
        return b ? gcd (b, a % b) : a;
}
```

Наконец, приведём и нерекурсивную форму алгоритма:

```
int gcd (int a, int b) {
        while (b) {
                a %= b;
                swap (a, b);
        }
        return a;
}
```

Доказательство корректности

Сначала заметим, что при каждой итерации алгоритма Евклида его второй аргумент строго убывает, следовательно, поскольку он неотрицательный, то алгоритм Евклида **всегда завершается**.

Для **доказательства корректности** нам необходимо показать, что $\gcd(a, b) = \gcd(b, a \bmod b)$ для любых $a \geq 0, b > 0$.

Покажем, что величина, стоящая в левой части равенства, делится на настоящую в правой, а стоящая в правой — делится на стоящую в левой. Очевидно, это будет означать, что левая и правая части совпадают, что и докажет корректность алгоритма Евклида.

Обозначим $d = \gcd(a, b)$. Тогда, по определению, $d|a$ и $d|b$.

Далее, разложим остаток от деления a на b через их частное:

$$a \bmod b = a - b \left\lfloor \frac{a}{b} \right\rfloor$$

Но тогда отсюда следует:

$$d \mid (a \bmod b)$$

Итак, вспоминая утверждение $d|b$, получаем систему:

$$\begin{cases} d \mid b, \\ d \mid (a \bmod b) \end{cases}$$

Воспользуемся теперь следующим простым фактом: если для каких-то трёх чисел p, q, r выполнено: $p|q$ и $p|r$, то выполняется и: $p \mid \gcd(q, r)$. В нашей ситуации получаем:

$$d \mid \gcd(b, a \bmod b)$$

Или, подставляя вместо d его определение как $\gcd(a, b)$, получаем:

$$\gcd(a, b) \mid \gcd(b, a \bmod b)$$

Итак, мы провели половину доказательства: показали, что левая часть делит правую. Вторая половина доказательства производится аналогично.

Время работы

Время работы алгоритма оценивается **теоремой Ламе**, которая устанавливает удивительную связь алгоритма Евклида и последовательности Фибоначчи:

Если $a > b \geq 1$ и $b < F_n$ для некоторого n, то алгоритм Евклида выполнит не более $n - 2$ рекурсивных вызовов.

Более того, можно показать, что верхняя граница этой теоремы — оптимальная. При $a = F_n, b = F_{n-1}$ будет выполнено именно $n - 2$ рекурсивных вызова. Иными словами, **последовательные числа Фибоначчи — наихудшие входные данные** для алгоритма Евклида.

Учитывая, что числа Фибоначчи растут экспоненциально (как константа в степени n), получаем, что алгоритм Евклида выполняется за $O(\log \min(a, b))$ операций умножения.

НОК (наименьшее общее кратное)

Вычисление наименьшего общего кратного (least common multiplier, lcm) сводится к вычислению $\gcd$ следующим простым утверждением:

$$\operatorname{lcm}(a, b) = \frac{a \cdot b}{\gcd(a, b)}$$

Таким образом, вычисление НОК также можно сделать с помощью алгоритма Евклида, с той же асимптотикой:

```
int lcm (int a, int b) {
        return a / gcd (a, b) * b;
}
```

(здесь выгодно сначала поделить на $\gcd$, а только потом домножать на b, поскольку это поможет избежать переполнений в некоторых случаях)

Литература

- Томас Кормен, Чарльз Лейзерсон, Рональд Ривест, Клиффорд Штайн. **Алгоритмы**: **Построение и анализ** [2005]

Решето Эратосфена

Решето Эратосфена — это алгоритм, позволяющий найти все простые числа в отрезке $[1; n]$ за $O(n \log \log n)$ операций.

Идея проста — запишем ряд чисел $1 \ldots n$, и будем вычеркивать сначала все числа, делящиеся на 2, кроме самого числа 2, затем деляющиеся на 3, кроме самого числа 3, затем на 5, затем на $7, 11$, и все остальные простые до n.

Реализация

Сразу приведём реализацию алгоритма:

```
int n;
vector<char> prime (n+1, true);
prime[0] = prime[1] = false;
for (int i=2; i<=n; ++i)
        if (prime[i])
                if (i * 1ll * i <= n)
                        for (int j=i*i; j<=n; j+=i)
                                prime[j] = false;
```

Этот код сначала помечает все числа, кроме нуля и единицы, как простые, а затем начинает процесс отсеивания составных чисел. Для этого мы перебираем в цикле все числа от 2 до n, и, если текущее число i простое, то помечаем все числа, кратные ему, как составные.

При этом мы начинаем идти от i^2, поскольку все меньшие числа, кратные i, обязательно имеют простой делитель меньше i, а значит, все они уже были отсеяны раньше. (Но поскольку i^2 легко может переполнить тип int, в коде перед вторым вложенным циклом делается дополнительная проверка с использованием типа $long\ long$.)

При такой реализации алгоритм потребляет $O(n)$ памяти (что очевидно) и выполняет $O(n \log \log n)$ действий (это доказывается в следующем разделе).

Асимптотика

Докажем, что асимптотика алгоритма равна $O(n \log \log n)$.

Итак, для каждого простого $p \le n$ будет выполняться внутренний цикл, который совершит $\frac{n}{p}$ действий. Следовательно, нам нужно оценить следующую величину:

$$\sum_{\substack{p \le n, \\ p\ is\ prime}} \frac{n}{p} = n \cdot \sum_{\substack{p \le n, \\ p\ is\ prime}} \frac{1}{p}.$$

Вспомним здесь два известных факта: что число простых, меньше либо равных n, приблизительно равно $\frac{n}{\ln n}$, и что k-ое простое число приблизительно равно $k \ln k$ (это следует из первого утверждения). Тогда сумму можно записать таким образом:

$$\sum_{\substack{p \le n, \\ p\ is\ prime}} \frac{1}{p} \approx \frac{1}{2} + \sum_{k=2}^{\frac{n}{\ln n}} \frac{1}{k \ln k}.$$

Здесь мы выделили первое простое из суммы, поскольку при $k = 1$ согласно приближению $k \ln k$ получится 0, что приведёт к делению на нуль.

Теперь оценим такую сумму с помощью интеграла от той же функции по k от 2 до $\frac{n}{\ln n}$ (мы можем производить такое приближение, поскольку, фактически, сумма относится к интегралу как его приближение по формуле прямоугольников):

$$\sum_{k=2}^{\frac{n}{\ln n}} \frac{1}{k \ln k} \approx \int_{2}^{\frac{n}{\ln n}} \frac{1}{k \ln k}\, dk.$$

Первообразная для подынтегральной функции есть $\ln \ln k$. Выполняя подстановку и убирая члены меньшего порядка, получаем:

$$\int\limits_{2}^{\frac{n}{\ln n}} \frac{1}{k \ln k}\, dk = \ln \ln \frac{n}{\ln n} - \ln \ln 2 = \ln(\ln n - \ln \ln n) - \ln \ln 2 \approx \ln \ln n.$$

Теперь, возвращаясь к первоначальной сумме, получаем её приближённую оценку:

$$\sum_{\substack{p \le n, \\ p\ is\ prime}} \frac{n}{p} \approx n \ln \ln n + o(n),$$

что и требовалось доказать.

Более строгое доказательство (и дающее более точную оценку с точностью до константных множителей) можно найти в книге Hardy и Wright "An Introduction to the Theory of Numbers" (стр. 349).

Различные оптимизации решета Эратосфена

Самый большой недостаток алгоритма — то, что он "гуляет" по памяти, постоянно выходя за пределы кэш-памяти, из-за чего константа, скрытая в $O(n \log \log n)$, сравнительно велика.

Кроме того, для достаточно больших n узким местом становится объём потребляемой памяти.

Ниже рассмотрены методы, позволяющие как уменьшить число выполняемых операций, так и значительно сократить потребление памяти.

Просеивание простыми до корня

Самый очевидный момент — что для того, чтобы найти все простые до n, достаточно выполнить просеивание только простыми, не превосходящими корня из n.

Таким образом, изменится внешний цикл алгоритма:

```
for (int i=2; i*i<=n; ++i)
```

На асимптотику такая оптимизация не влияет (действительно, повторив приведённое выше доказательство, мы получим оценку $n \ln \ln \sqrt{n} + o(n)$, что, по свойствам логарифма, асимптотически есть то же самое), хотя число операций заметно уменьшится.

Решето только по нечётным числам

Поскольку все чётные числа, кроме 2, — составные, то можно вообще не обрабатывать никак чётные числа, а оперировать только нечётными числами.

Во-первых, это позволит вдвое сократить объём требуемой памяти. Во-вторых, это уменьшит число делаемых алгоритмом операций примерно вдвое.

Уменьшение объёма потребляемой памяти

Заметим, что алгоритм Эратосфена фактически оперирует с n битами памяти. Следовательно, можно существенно сэкономить потребление памяти, храня не n байт — переменных булевского типа, а n бит, т.е. $n/8$ байт памяти.

Однако такой подход — "**битовое сжатие**" — существенно усложнит оперирование этими битами. Любое чтение или запись бита будут представлять из себя несколько арифметических операций, что в итоге приведёт к замедлению алгоритма.

Таким образом, этот подход оправдан, только если n настолько большое, что n байт памяти выделить уже нельзя. Сэкономив память (в 8 раз), мы заплатим за это существенным замедлением алгоритма.

В завершение стоит отметить, что в языке C++ уже реализованы контейнеры, автоматически осуществляющие битовое сжатие: vector<bool> и bitset<>. Впрочем, если скорость работы очень важна, то лучше реализовать битовое сжатие вручную, с помощью битовых операций — на сегодняшний день компиляторы всё же не в состоянии генерировать достаточно быстрый код.

Блочное решето

Из оптимизации "просеивание простыми до корня" следует, что нет необходимости хранить всё время весь массив $prime[1 \ldots n]$. Для выполнения просеивания достаточно хранить только простые до корня из n, т. е. $prime[1 \ldots \sqrt{n}]$, а остальную часть массива $prime$ строить поблочно, храня в текущий момент времени только один блок.

Пусть s — константа, определяющая размер блока, тогда всего будет $\left\lceil \frac{n}{s} \right\rceil$ блоков, k-ый блок ($k = 0 \ldots \left\lfloor \frac{n}{s} \right\rfloor$) содержит числа в отрезке $[ks; ks + s - 1]$. Будем обрабатывать блоки по очереди, т.е. для каждого k-го блока будем перебирать все простые (от 1 до $\sqrt{n}$) и выполнять ими просеивание только внутри текущего блока. Аккуратно стоит обрабатывать первый блок — во-первых, простые из $[1; \sqrt{n}]$ не должны удалить сами себя, а во-вторых, числа 0 и 1 должны особо помечаться как не простые. При обработке последнего блока также следует не забывать о том, что последнее нужное число n не обязательно находится в конце блока.

Приведём реализацию блочного решета. Программа считывает число n и находит количество простых от 1 до n:

```
const int SQRT_MAXN = 100000; // корень из максимального значения N
const int S = 10000;
bool nprime[SQRT_MAXN], bl[S];
int primes[SQRT_MAXN], cnt;

int main() {

        int n;
        cin >> n;
        int nsqrt = (int) sqrt (n + .0);
        for (int i=2; i<=nsqrt; ++i)
                if (!nprime[i]) {
                        primes[cnt++] = i;
                        if (i * 1ll * i <= nsqrt)
                                for (int j=i*i; j<=nsqrt; j+=i)
                                        nprime[j] = true;
                }

        int result = 0;
        for (int k=0, maxk=n/S; k<=maxk; ++k) {
                memset (bl, 0, sizeof bl);
                int start = k * S;
                for (int i=0; i<cnt; ++i) {
                        int start_idx = (start + primes[i] - 1) / primes[i];
                        int j = max(start_idx,2) * primes[i] - start;
                        for (; j<S; j+=primes[i])
                                bl[j] = true;
                }
                if (k == 0)
                        bl[0] = bl[1] = true;
                for (int i=0; i<S && start+i<=n; ++i)
                        if (!bl[i])
                                ++result;
        }
        cout << result;

}
```

Асимптотика блочного решета такая же, как и обычного решета Эратосфена (если, конечно, размер s блоков не будет совсем маленьким), зато объём используемой памяти сократится до $O(\sqrt{n} + s)$ и уменьшится "блуждание" по памяти. Но, с другой стороны, для каждого блока для каждого простого из $[1; \sqrt{n}]$ будет выполняться деление, что будет сильно сказываться при меньших размерах блока. Следовательно, при выборе константы s необходимо соблюсти баланс.

Как показывают эксперименты, наилучшая скорость работы достигается, когда s имеет значение приблизительно от 10^4 до 10^5.

Улучшение до линейного времени работы

Алгоритм Эратосфена можно преобразовать в другой алгоритм, который уже будет работать за линейное время — см. статью "Решето Эратосфена с линейным временем работы". (Впрочем, этот алгоритм имеет и недостатки.)

Расширенный алгоритм Евклида

В то время как "обычный" алгоритм Евклида просто находит наибольший общий делитель двух чисел a и b, расширенный алгоритм Евклида находит помимо НОД также коэффициенты x и y такие, что:

$$a \cdot x + b \cdot y = \gcd(a, b).$$

Т.е. он находит коэффициенты, с помощью которых НОД двух чисел выражается через сами эти числа.

Алгоритм

Внести вычисление этих коэффициентов в алгоритм Евклида несложно, достаточно вывести формулы, по которым они меняются при переходе от пары (a, b) к паре $(b\%a, a)$ (знаком процента мы обозначаем взятие остатка от деления).

Итак, пусть мы нашли решение (x_1, y_1) задачи для новой пары $(b\%a, a)$:

$$(b\%a) \cdot x_1 + a \cdot y_1 = g,$$

и хотим получить решение (x, y) для нашей пары (a, b):

$$a \cdot x + b \cdot y = g.$$

Для этого преобразуем величину $b\%a$:

$$b\%a = b - \left\lfloor \frac{b}{a} \right\rfloor \cdot a.$$

Подставим это в приведённое выше выражение с x_1 и y_1 и получим:

$$g = (b\%a) \cdot x_1 + a \cdot y_1 = \left(b - \left\lfloor \frac{b}{a} \right\rfloor \cdot a\right) \cdot x_1 + a \cdot y_1,$$

и, выполняя перегруппировку слагаемых, получаем:

$$g = b \cdot x_1 + a \cdot \left(y_1 - \left\lfloor \frac{b}{a} \right\rfloor \cdot x_1\right).$$

Сравнивая это с исходным выражением над неизвестными x и y, получаем требуемые выражения:

$$\begin{cases} x = y_1 - \left\lfloor \frac{b}{a} \right\rfloor \cdot x_1, \\ y = x_1. \end{cases}$$

Реализация

```
int gcd (int a, int b, int & x, int & y) {
        if (a == 0) {
                x = 0; y = 1;
                return b;
        }
        int x1, y1;
        int d = gcd (b%a, a, x1, y1);
        x = y1 - (b / a) * x1;
        y = x1;
        return d;
}
```

Это рекурсивная функция, которая по-прежнему возвращает значение НОД от чисел a и b, но помимо этого — также искомые коэффициенты x и y в виде параметров функции, передающихся по ссылкам.

База рекурсии — случай a. Тогда НОД равен b, и, очевидно, требуемые коэффициенты x и y равны 0 и 1 соответственно. В остальных случаях работает обычное решение, а коэффициенты пересчитываются по вышеописанным формулам.

Расширенный алгоритм Евклида в такой реализации работает корректно даже для отрицательных чисел.

Литература

- Томас Кормен, Чарльз Лейзерсон, Рональд Ривест, Клиффорд Штайн. **Алгоритмы**: **Построение и анализ** [2005]

Числа Фибоначчи

Определение

Последовательность Фибоначчи определяется следующим образом:

$$
\begin{aligned}
&F_0 = 0, \\
&F_1 = 1, \\
&F_n = F_{n-1} + F_{n-2}.
\end{aligned}
$$

Несколько первых её членов:

$$0,\ 1,\ 1,\ 2,\ 3,\ 5,\ 8,\ 13,\ 21,\ 34,\ ,\ 55,\ 89,\ \ldots$$

История

Эти числа ввёл в 1202 г. Леонардо Фибоначчи (Leonardo Fibonacci) (также известный как Леонардо Пизанский (Leonardo Pisano)). Однако именно благодаря математику 19 века Люка (Lucas) название "числа Фибоначчи" стало общеупотребительным.

Впрочем, индийские математики упоминали числа этой последовательности ещё раньше: Гопала (Gopala) до 1135 г., Хемачандра (Hemachandra) — в 1150 г.

Числа Фибоначчи в природе

Сам Фибоначчи упоминал эти числа в связи с такой задачей: "Человек посадил пару кроликов в загон, окруженный со всех сторон стеной. Сколько пар кроликов за год может произвести на свет эта пара, если известно, что каждый месяц, начиная со второго, каждая пара кроликов производит на свет одну пару?". Решением этой задачи и будут числа последовательности, называемой теперь в его честь. Впрочем, описанная Фибоначчи ситуация — больше игра разума, чем реальная природа.

Индийские математики Гопала и Хемачандра упоминали числа этой последовательности в связи с количеством ритмических рисунков, образующихся в результате чередования долгих и кратких слогов в стихах или сильных и слабых долей в музыке. Число таких рисунков, имеющих в целом n долей, равно F_n.

Числа Фибоначчи появляются и в работе Кеплера 1611 года, который размышлял о числах, встречающихся в природе (работа "О шестиугольных снежинках").

Интересен пример растения — тысячелистника, у которого число стеблей (а значит и цветков) всегда есть число Фибоначчи. Причина этого проста: будучи изначально с единственным стеблем, этот стебель затем делится на два, затем от главного стебля ответвляется ещё один, затем первые два стебля снова разветвляются, затем все стебли, кроме двух последних, разветвляются, и так далее. Таким образом, каждый стебель после своего появления "пропускает" одно разветвление, а затем начинает делиться на каждом уровне разветвлений, что и даёт в результате числа Фибоначчи.

Вообще говоря, у многих цветов (например, лилий) число лепестков является тем или иным числом Фибоначчи.

Также в ботанике известно явление "филлотаксиса". В качестве примера можно привести расположение семечек подсолнуха: если посмотреть сверху на их расположение, то можно увидеть одновременно две серии спиралей (как бы наложенных друг на друга): одни закручены по часовой стрелке, другие — против. Оказывается, что число этих спиралей примерно совпадает с двумя последовательными числами Фибоначчи: 34 и 55 или 89 и 144. Аналогичные факты верны и для некоторых других цветов, а также для сосновых шишек, брокколи, ананасов, и т.д.

Для многих растений (по некоторым данным, для 90% из них) верен и такой интересный факт. Рассмотрим какой-нибудь лист, и будем спускаться от него вниз до тех пор, пока не достигнем листа, расположенного на стебле точно так же (т.е. направленного точно в ту же сторону). Попутно будем считать все листья, попадавшиеся нам (т.е. расположенные по высоте между стартовым листом и конечным), но расположенными по-другому. Нумеруя их, мы будем постепенно совершать витки вокруг стебля (поскольку листья расположены на стебле по спирали). В зависимости от того, совершать витки по часовой стрелке или против, будет получаться разное число витков. Но оказывается, что число витков, совершённых нами по часовой стрелке, число витков, совершённых против часовой стрелки, и число встреченных листьев образуют 3 последовательных числа Фибоначчи.

Впрочем, следует отметить, что есть и растения, для которых приведённые выше подсчёты дадут числа из совсем других последовательностей, поэтому нельзя сказать, что явление филлотаксиса является законом, — это

скорее занимательная тенденция.

Свойства

Числа Фибоначчи обладают множеством интересных математических свойств.

Вот лишь некоторые из них:

- Соотношение Кассини:

$$F_{n+1}F_{n-1} - F_n^2 = (-1)^n.$$

- Правило "сложения":

$$F_{n+k} = F_k F_{n+1} + F_{k-1} F_n.$$

- Из предыдущего равенства при $k = n$ вытекает:

$$F_{2n} = F_n(F_{n+1} + F_{n-1}).$$

- Из предыдущего равенста по индукции можно получить, что

  F_{nk} всегда кратно F_n.

- Верно и обратное к предыдущему утверждение:

  если F_m кратно F_n, то m кратно n.

- НОД-равенство:

$$\gcd(F_m, F_n) = F_{\gcd(m,n)}.$$

- По отношению к алгоритму Евклида числа Фибоначчи обладают тем замечательным свойством, что они являются наихудшими входными данными для этого алгоритма (см. "Теорема Ламе" в Алгоритме Евклида).

Фибоначчиева система счисления

Теорема Цекендорфа утверждает, что любое натуральное число n можно представить единственным образом в виде суммы чисел Фибоначчи:

$$N = F_{k_1} + F_{k_2} + \ldots + F_{k_r}$$

где $k_1 \geq k_2 + 2, k_2 \geq k_3 + 2, \cdots, k_r \geq 2$ (т.е. в записи нельзя использовать два соседних числа Фибоначчи).

Отсюда следует, что любое число можно однозначно записать в **фибоначчиевой системе счисления**, например:

$$\begin{aligned} 9 &= 8 + 1 = F_6 + F_1 = (10001)_F, \\ 6 &= 5 + 1 = F_5 + F_1 = (1001)_F, \\ 19 &= 13 + 5 + 1 = F_7 + F_5 + F_1 = (101001)_F, \end{aligned}$$

причём ни в каком числе не могут идти две единицы подряд.

Нетрудно получить и правило прибавления единицы к числу в фибоначчиевой системе счисления: если младшая цифра равна 0, то её заменяем на 1, а если равна 1 (т.е. в конце стоит 01), то 01 заменяем на 10. Затем "исправляем" запись, последовательно исправляя везде 011 на 100. В результате за линейное время будет получена запись нового числа.

Перевод числа в фибоначчиеву систему счисления осуществляется простым "жадным" алгоритмом: просто перебираем числа Фибоначчи от больших к меньшим и, если некоторое $F_k \leq n$, то F_k входит в запись числа n, и мы отнимаем F_k от n и продолжаем поиск.

Формула для n-го числа Фибоначчи

Формула через радикалы

Существует замечательная формула, называемая по имени французского математика Бине (Binet), хотя она была известна до него Муавру (Moivre):

$$F_n = \frac{\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n}{\sqrt{5}}.$$

Эту формулу легко доказать по индукции, однако вывести её можно с помощью понятия образующих функций или с помощью решения функционального уравнения.

Сразу можно заметить, что второе слагаемое всегда по модулю меньше 1, и более того, очень быстро убывает (экспоненциально). Отсюда следует, что значение первого слагаемого даёт "почти" значение F_n. Это можно записать в строгом виде:

$$F_n = \left[\frac{\left(\frac{1+\sqrt{5}}{2}\right)^n}{\sqrt{5}}\right],$$

где квадратные скобки обозначают округление до ближайшего целого.

Впрочем, для практического применения в вычислениях эти формулы мало подходят, потому что требуют очень высокой точности работы с дробными числами.

Матричная формула для чисел Фибоначчи

Нетрудно доказать матричное следующее равенство:

$$\begin{pmatrix} F_{n-2} & F_{n-1} \end{pmatrix} \cdot \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} = \begin{pmatrix} F_{n-1} & F_n \end{pmatrix}.$$

Но тогда, обозначая

$$P \equiv \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix},$$

получаем:

$$\begin{pmatrix} F_0 & F_1 \end{pmatrix} \cdot P^n = \begin{pmatrix} F_n & F_{n+1} \end{pmatrix}.$$

Таким образом, для нахождения n-го числа Фибоначчи надо возвести матрицу P в степень n.

Вспоминая, что возведение матрицы в n-ую степень можно осуществить за $O(\log n)$ (см. Бинарное возведение в степень), получается, что n-ое число Фибоначчи можно легко вычислить за $O(\log n)$ с использованием только целочисленной арифметики.

Периодичность последовательности Фибоначчи по модулю

Рассмотрим последовательность Фибоначчи F_i по некоторому модулю p. Докажем, что она является периодичной, и причём период начинается с $F_1 = 1$ (т.е. предпериод содержит только F_0).

Докажем это от противного. Рассмотрим $p^2 + 1$ пар чисел Фибоначчи, взятых по модулю p:

$$(F_1, F_2),\ (F_2, F_3),\ \ldots,\ (F_{p^2+1}, F_{p^2+2}).$$

Поскольку по модулю p может быть только p^2 различных пар, то среди этой последовательности найдётся как минимум две одинаковые пары. Это уже означает, что последовательность периодична.

Выберем теперь среди всех таких одинаковых пар две одинаковые пары с наименьшими номерами. Пусть это пары с некоторыми номерами (F_a, F_{a+1}) и (F_b, F_{b+1}). Докажем, что $a = 1$. Действительно, в противном случае для них найдутся предыдущие пары (F_{a-1}, F_a) и (F_{b-1}, F_b), которые, по свойству чисел Фибоначчи, также будут равны друг другу. Однако это противоречит тому, что мы выбрали совпадающие пары с наименьшими номерами, что и требовалось доказать.

Литература

- Роналд Грэхэм, Дональд Кнут, Орен Паташник. **Конкретная математика** [1998]

Обратный элемент в кольце по модулю

Определение

Пусть задан некоторый натуральный модуль m, и рассмотрим кольцо, образуемое этим модулем (т.е. состоящее из чисел от 0 до $m-1$). Тогда для некоторых элементов этого кольца можно найти **обратный элемент**.

Обратным к числу a по модулю m называется такое число b, что:

$$a \cdot b \equiv 1 \pmod m,$$

и его нередко обозначают через a^{-1}.

Понятно, что для нуля обратного элемента не существует никогда; для остальных же элементов обратный может как существовать, так и нет. Утверждается, что обратный существует только для тех элементов a, которые **взаимно просты** с модулем m.

Рассмотрим ниже два способа нахождения обратного элемента, работающих при условии, что он существует.

В завершение, рассмотрим алгоритм, который позволяет найти обратные ко всех числам по некоторому модулю за линейное время.

Нахождение с помощью Расширенного алгоритма Евклида

Рассмотрим вспомогательное уравнение (относительно неизвестных x и y):

$$a \cdot x + m \cdot y = 1.$$

Это линейное диофантово уравнение второго порядка. Как показано в соответствующей статье, из условия $\gcd(a, m) = 1$ следует, что это уравнение имеет решение, которое можно найти с помощью Расширенного алгоритма Евклида (отсюда же, кстати говоря, следует, что когда $\gcd(a, m) \neq 1$, решения, а потому и обратного элемента, не существует).

С другой стороны, если мы возьмём от обеих частей уравнения остаток по модулю m, то получим:

$$a \cdot x = 1 \pmod m.$$

Таким образом, найденное x и будет являться обратным к a.

Реализация (с учётом того, что найденное x надо взять по модулю m, и x могло быть отрицательным):

```
int x, y;
int g = gcdex (a, m, x, y);
if (g != 1)
        cout << "no solution";
else {
        x = (x % m + m) % m;
        cout << x;
}
```

Асимптотика этого решения получается $O(\log m)$.

Нахождение с помощью Бинарного возведения в степень

Воспользуемся теоремой Эйлера:

$$a^{\phi(m)} \equiv 1 \pmod m,$$

которая верна как раз для случая взаимно простых a и m.

Кстати говоря, в случае простого модуля m мы получаем ещё более простое утверждение — малую теорему Ферма:

$$a^{m-1} \equiv 1 \pmod m.$$

Умножим обе части каждого из уравнений на a^{-1}, получим:

- для любого модуля m:

$$a^{\phi(m)-1} \equiv a^{-1} \pmod m,$$

- для простого модуля m:

$$a^{m-2} \equiv a^{-1} \pmod m.$$

Таким образом, мы получили формулы для непосредственного вычисления обратного. Для практического применения обычно используют эффективный алгоритм бинарного возведения в степень, который в нашем случае позволит произвести возведение в степень за $O(\log m)$.

Этот метод представляется несколько проще описанного в предыдущем пункте, однако он требует знания значения функции Эйлера, что фактически требует факторизации модуля m, что иногда может оказаться весьма сложной задачей.

Если же факторизация числа известна, то тогда и этот метод также работает за асимптотику $O(\log m)$.

Нахождение всех простых по заданному модулю за линейное время

Пусть дан простой модуль m. Требуется для каждого числа в отрезке $[1; m-1]$ найти обратное к нему.

Применяя описанные выше алгоритмы, мы получим лишь решения с асимптотикой $O(m \log m)$. Здесь же мы приведём простое решение с асимптотикой $O(m)$.

Решение это выглядит следующим образом. Обозначим через $r[i]$ искомое обратное к числу i по модулю m. Тогда для $i > 1$ верно тождество:

$$r[i] = -\left\lfloor \frac{m}{i} \right\rfloor \cdot r[m \bmod i]. \pmod m$$

Реализация этого удивительно лаконичного решения:

```
r[1] = 1;
for (int i=2; i<m; ++i)
        r[i] = (m - (m/i) * r[m%i] % m) % m;
```

Доказательство этого решения представляет из себя цепочку простых преобразований:

Распишем значение $m \bmod i$:

$$m \bmod i = m - \left\lfloor \frac{m}{i} \right\rfloor \cdot i,$$

откуда, беря обе части по модулю m, получаем:

$$m \bmod i = -\left\lfloor \frac{m}{i} \right\rfloor \cdot i. \pmod m$$

Умножая обе части на обратное к i и обратное к $(m \bmod i)$, получаем искомую формулу:

$$r[i] = -\left\lfloor \frac{m}{i} \right\rfloor \cdot r[m \bmod i], \pmod m$$

что и требовалось доказать.

Код Грея

Определение

Кодом Грея называется такая система нумерования неотрицательных чисел, когда коды двух соседних чисел отличаются ровно в одном бите.

Например, для чисел длины 3 бита имеем такую последовательность кодов Грея: $000, 001, 011, 010, 110, 111, 101, 100$. Например, $G(4) = 6$.

Этот код был изобретен Фрэнком Грэем (Frank Gray) в 1953 году.

Нахождение кода Грея

Рассмотрим биты числа n и биты числа $G(n)$. Заметим, что i-ый бит $G(n)$ равен единице только в том случае, когда i-ый бит n равен единице, а $i+1$-ый бит равен нулю, или наоборот (i-ый бит равен нулю, а $i+1$-ый равен единице). Таким образом, имеем: $G(n) = n \oplus (n >> 1)$:

```
int g (int n) {
        return n ^ (n >> 1);
}
```

Нахождение обратного кода Грея

Требуется по коду Грея g восстановить исходное число n.

Будем идти от старших битов к младшим (пусть самый младший бит имеет номер 1, а самый старший — k).
Получаем такие соотношения между битами n_i числа n и битами g_i числа g:

$$
\begin{aligned}
&n_k = g_k, \\
&n_{k-1} = g_{k-1} \oplus n_k = g_k \oplus g_{k-1}, \\
&n_{k-2} = g_{k-2} \oplus n_{k-1} = g_k \oplus g_{k-1} \oplus g_{k-2}, \\
&n_{k-3} = g_{k-3} \oplus n_{k-2} = g_k \oplus g_{k-1} \oplus g_{k-2} \oplus g_{k-3}, \\
&\ldots
\end{aligned}
$$

В виде программного кода это проще всего записать так:

```
int rev_g (int g) {
        int n = 0;
        for (; g; g>>=1)
                n ^= g;
        return n;
}
```

Применения

Коды Грея имеют несколько применений в различных областях, иногда достаточно неожиданных:

- n-битный код Грея соответствует гамильтонову циклу по n-мерному кубу.
- В технике, коды Грея используются для **минимизации ошибок** при преобразовании аналоговых сигналов в цифровые (например, в датчиках). В частности, коды Грея и были открыты в связи с этим применением.
- Коды Грея применяются в решении задачи о **Ханойских башнях**.

  Пусть n — количество дисков. Начнём с кода Грея длины n, состоящего из одних нулей (т.е. $G(0)$), и будем двигаться по кодам Грея (от $G(i)$ переходить к $G(i+1)$). Поставим в соответствие каждому i-ому биту текущего кода Грея i-ый диск (причём самому младшему биту соответствует наименьший по размеру диск, а самому старшему биту — наибольший). Поскольку на каждом шаге изменяется ровно один бит, то мы можем понимать изменение бита i как перемещение i-го диска. Заметим, что для всех дисков, кроме наименьшего, на каждом шаге имеется ровно

один вариант хода (за исключением стартовой и финальной позиций). Для наименьшего диска всегда имеется два варианта хода, однако имеется стратегия выбора хода, всегда приводящая к ответу: если n нечётно, то последовательность перемещений наименьшего диска имеет вид $f \to t \to r \to f \to t \to r \to \ldots$ (где f — стартовый стержень, t — финальный стержень, r — оставшийся стержень), а если n чётно, то $f \to r \to t \to f \to r \to t \to \ldots$.

- Коды Грея также находят применение в теории **генетических алгоритмов**.

Задачи в online judges

Список задач, которые можно сдать, используя коды Грея:

- SGU #249 "Matrix" [сложность: средняя]

Длинная арифметика

Длинная арифметика — это набор программных средств (структуры данных и алгоритмы), которые позволяют работать с числами гораздо больших величин, чем это позволяют стандартные типы данных.

Виды целочисленной длинной арифметики

Вообще говоря, даже только в олимпиадных задачах набор средств достаточно велик, поэтому произведём классификацию различных видов длинной арифметики.

Классическая длинная арифметика

Основная идея заключается в том, что число хранится в виде массива его цифр.

Цифры могут использоваться из той или иной системы счисления, обычно применяются десятичная система счисления и её степени (десять тысяч, миллиард), либо двоичная система счисления.

Операции над числами в этом виде длинной арифметики производятся с помощью "школьных" алгоритмов сложения, вычитания, умножения, деления столбиком. Впрочем, к ним также применимы алгоритмы быстрого умножения: Быстрое преобразование Фурье и Алгоритм Карацубы.

Здесь описана работа только с неотрицательными длинными числами. Для поддержки отрицательных чисел необходимо ввести и поддерживать дополнительный флаг "отрицательности" числа, либо же работать в дополняющих кодах.

Структура данных

Хранить длинные числа будем в виде вектора чисел int, где каждый элемент — это одна цифра числа.

```
typedef vector<int> lnum;
```

Для повышения эффективности будем работать в системе по основанию миллиард, т.е. каждый элемент вектора $lnum$ содержит не одну, а сразу 9 цифр:

```
const int base = 1000*1000*1000;
```

Цифры будут храниться в векторе в таком порядке, что сначала идут наименее значимые цифры (т.е. единицы, десятки, сотни, и т.д.).

Кроме того, все операции будут реализованы таким образом, что после выполнения любой из них лидирующие нули (т. е. лишние нули в начале числа) отсутствуют (разумеется, в предположении, что перед каждой операцией лидирующие нули также отсутствуют). Следует отметить, что в представленной реализации для числа ноль корректно поддерживаются сразу два представления: пустой вектор цифр, и вектор цифр, содержащий единственный элемент — ноль.

Вывод

Самое простое — это вывод длинного числа.

Сначала мы просто выводим самый последний элемент вектора (или 0, если вектор пустой), а затем выводим все оставшиеся элементы вектора, дополняя их нулями до 9 символов:

```
printf ("%d", a.empty() ? 0 : a.back());
for (int i=(int)a.size()-2; i>=0; --i)
        printf ("%09d", a[i]);
```

(здесь небольшой тонкий момент: нужно не забыть записать приведение типа (int), поскольку в противном случае число $a.size()$ будут беззнаковым, и если $a.size() \le 1$, то при вычитании произойдёт переполнение)

Чтение

Считываем строку в $string$, и затем преобразовываем её в вектор:

```
for (int i=(int)s.length(); i>0; i-=9)
```

```
        if (i < 9)
                a.push_back (atoi (s.substr (0, i).c_str()));
        else
                a.push_back (atoi (s.substr (i-9, 9).c_str()));
```

Если использовать вместо $string$ массив $char$'ов, то код получится ещё компактнее:

```
for (int i=(int)strlen(s); i>0; i-=9) {
        s[i] = 0;
        a.push_back (atoi (i>=9 ? s+i-9 : s));
}
```

Если во входном числе уже могут быть лидирующие нули, то их после чтения можно удалить таким образом:

```
while (a.size() > 1 && a.back() == 0)
        a.pop_back();
```

Сложение

Прибавляет к числу a число b и сохраняет результат в a:

```
int carry = 0;
for (size_t i=0; i<max(a.size(),b.size()) || carry; ++i) {
        if (i == a.size())
                a.push_back (0);
        a[i] += carry + (i < b.size() ? b[i] : 0);
        carry = a[i] >= base;
        if (carry)  a[i] -= base;
}
```

Вычитание

Отнимает от числа a число b ($a \geq b$) и сохраняет результат в a:

```
int carry = 0;
for (size_t i=0; i<b.size() || carry; ++i) {
        a[i] -= carry + (i < b.size() ? b[i] : 0);
        carry = a[i] < 0;
        if (carry)  a[i] += base;
}
while (a.size() > 1 && a.back() == 0)
        a.pop_back();
```

Здесь мы после выполнения вычитания удаляем лидирующие нули, чтобы поддерживать предикат о том, что таковые отсутствуют.

Умножение длинного на короткое

Умножает длинное a на короткое b ($b < \mathrm{base}$) и сохраняет результат в a:

```
int carry = 0;
for (size_t i=0; i<a.size() || carry; ++i) {
        if (i == a.size())
                a.push_back (0);
        long long cur = carry + a[i] * 1ll * b;
        a[i] = int (cur % base);
        carry = int (cur / base);
}
while (a.size() > 1 && a.back() == 0)
        a.pop_back();
```

Здесь мы после выполнения деления удаляем лидирующие нули, чтобы поддерживать предикат о том, что таковые отсутствуют.

(Примечание: способ **дополнительной оптимизации**. Если скорость работы чрезвычайно важна, то можно попробовать заменить два деления одним: посчитать только целую часть от деления (в коде это переменная $carry$), а затем уже посчитать по ней остаток от деления (с помощью одной операции умножения).

Как правило, этот приём позволяет ускорить код, хотя и не очень значительно.)

Умножение двух длинных чисел

Умножает a на b и результат сохраняет в c:

```
lnum c (a.size()+b.size());
for (size_t i=0; i<a.size(); ++i)
        for (int j=0, carry=0; j<(int)b.size() || carry; ++j) {
                long long cur = c[i+j] + a[i] * 1ll * (j < (int)b.size() ?
b[j] : 0) + carry;
                c[i+j] = int (cur % base);
                carry = int (cur / base);
        }
while (c.size() > 1 && c.back() == 0)
        c.pop_back();
```

Деление длинного на короткое

Делит длинное a на короткое b ($b < \text{base}$), частное сохраняет в a, остаток в $carry$:

```
int carry = 0;
for (int i=(int)a.size()-1; i>=0; --i) {
        long long cur = a[i] + carry * 1ll * base;
        a[i] = int (cur / b);
        carry = int (cur % b);
}
while (a.size() > 1 && a.back() == 0)
        a.pop_back();
```

Длинная арифметика в факторизованном виде

Здесь идея заключается в том, чтобы хранить не само число, а его факторизацию, т.е. степени каждого входящего в него простого.

Этот метод также весьма прост для реализации, и в нём очень легко производить операции умножения и деления, однако невозможно произвести сложение или вычитание. С другой стороны, этот метод значительно экономит память в сравнении с "классическим" подходом, и позволяет производить умножение и деление значительно (асимптотически) быстрее.

Этот метод часто применяется, когда необходимо производить деление по непростому модулю: тогда достаточно хранить число в виде степеней по простым делителям этого модуля, и ещё одного числа — остатка по этому же модулю.

Длинная арифметика по системе простых модулей (Китайская теорема или схема Гарнера)

Суть в том, что выбирается некоторая система модулей (обычно небольших, помещающихся в стандартные типы данных), и число хранится в виде вектора из остатков от его деления на каждый из этих модулей.

Как утверждает Китайская теорема об остатках, этого достаточно, чтобы однозначно хранить любое число в диапазоне от 0 до произведения этих модулей минус один. При этом имеется Алгоритм Гарнера, который позволяет произвести это восстановление из модульного вида в обычную, "классическую", форму числа.

Таким образом, этот метод позволяет экономить память по сравнению с "классической" длинной арифметикой (хотя в некоторых случаях не столь радикально, как метод факторизации). Кроме того, в модульном виде можно очень быстро производить сложения, вычитания и умножения, — все за асимптотически однаковое время, пропорциональное количеству модулей системы.

Однако всё это даётся ценой весьма трудоёмкого перевода числа из этого модульного вида в обычный вид, для чего, помимо немалых временных затрат, потребуется также реализация "классической" длинной арифметики с умножением.

Помимо этого, производить **деление** чисел в таком представлении по системе простых модулей не представляется возможным.

Виды дробной длинной арифметики

Операции над дробными числами встречаются в олимпиадных задачах гораздо реже, и работать с огромными

дробными числами значительно сложнее, поэтому в олимпиадах встречается только специфическое подмножество дробной длинной арифметики.

Длинная арифметика в несократимых дробях

Число представляется в виде несократимой дроби $\frac{a}{b}$, где a и b — целые числа. Тогда все операции над дробными числами нетрудно свести к операциям над числителями и знаменателями этих дробей.

Обычно при этом для хранения числителя и знаменателя приходится также использовать длинную арифметику, но, впрочем, самый простой её вид — "классическая" длинная арифметика, хотя иногда оказывается достаточно встроенного 64-битного числового типа.

Выделение позиции плавающей точки в отдельный тип

Иногда в задаче требуется производить расчёты с очень большими либо очень маленькими числами, но при этом не допускать их переполнения. Встроенный $8-10$-байтовый тип $double$, как известно, допускает значения экспоненты в диапазоне $[-308; 308]$, чего иногда может оказаться недостаточно.

Приём, собственно, очень простой — вводится ещё одна целочисленная переменная, отвечающая за экспоненту, а после выполнения каждой операции дробное число "нормализуется", т.е. возвращается в отрезок $[0.1; 1)$, путём увеличения или уменьшения экспоненты.

При перемножении или делении двух таких чисел надо соответственно сложить либо вычесть их экспоненты. При сложении или вычитании перед выполнением этой операции числа следует привести к одной экспоненте, для чего одно из них домножается на 10 в степени разности экспонент.

Наконец, понятно, что не обязательно выбирать 10 в качестве основания экспоненты. Исходя из устройства встроенных типов с плавающей точкой, самым выгодным представляется класть основание равным 2.

Дискретное логарифмирование

Задача дискретного логарифмирования заключается в том, чтобы по данным целым a, b, m решить уравнение:

$$a^x = b \pmod m,$$

где a и m — **взаимно просты** (примечание: если они не взаимно просты, то описанный ниже алгоритм является некорректным; хотя, предположительно, его можно модифицировать, чтобы он по-прежнему работал).

Здесь описан алгоритм, известный как "baby-step-giant-step algorithm", предложенный **Шэнксом** (Shanks) в 1971 г., работающий за время за $O(\sqrt{m} \log m)$. Часто этот алгоритм просто называют алгоритмом "meet-in-the-middle" (потому что это одно из классических применений техники "meet-in-the-middle": "разделение задачи пополам").

Алгоритм

Итак, мы имеем уравнение:

$$a^x = b \pmod m,$$

где a и m взаимно просты.

Преобразуем уравнение. Положим

$$x = np - q,$$

где n — это заранее выбранная константа (как её выбирать в зависимости от m, мы поймём чуть позже). Иногда p называют "giant step" (поскольку увеличение его на единицу увеличивает x сразу на n), а в противоположность ему q — "baby step".

Очевидно, что любое x (из промежутка $[0; m)$ — понятно, что такого диапазона значений будет достаточно) можно представить в такой форме, причём для этого будет достаточно значений:

$$p \in \left[1; \left\lceil \frac{m}{n} \right\rceil\right], \qquad q \in [0; n].$$

Тогда уравнение принимает вид:

$$a^{np-q} = b \pmod m,$$

откуда, пользуясь тем, что a и m взаимно просты, получаем:

$$a^{np} = ba^q \pmod m.$$

Чтобы решить исходное уравнение, нужно найти соответствующие значения p и q, чтобы значения левой и правой частей совпали. Иначе говоря, надо решить уравнение:

$$f_1(p) = f_2(q).$$

Эта задача решается с помощью метода meet-in-the-middle следующим образом. Первая фаза алгоритма: посчитаем значения функции f_1 для всех значений аргумента p, и отсортируем эти значения. Вторая фаза алгоритма: будем перебирать значение второй переменной q, вычислять вторую функцию f_2, и искать это значение среди предвычисленных значений первой функции с помощью бинарного поиска.

Асимптотика

Сначала оценим время вычисления каждой из функций $f_1(p)$ и $f_2(q)$. И та, и другая содержит возведение в степень, которое можно выполнять с помощью алгоритма бинарного возведения в степень. Тогда обе этих функции мы можем вычислять за время $O(\log m)$.

Сам алгоритм в первой фазе содержит вычисление функции $f_1(p)$ для каждого возможного значения p и дальнейшую сортировку значений, что даёт нам асимптотику:

$$O\left(\left\lceil \frac{m}{n} \right\rceil \left(\log m + \log \left\lceil \frac{m}{n} \right\rceil\right)\right) = O\left(\left\lceil \frac{m}{n} \right\rceil \log m\right).$$

Во второй фазе алгоритма происходит вычисление функции $f_2(q)$ для каждого возможного значения q и бинарный поиск по массиву значений f_1, что даёт нам асимптотику:

$$O\left(n\left(\log m + \log\left\lceil\frac{m}{n}\right\rceil\right)\right) = O\left(n \log m\right).$$

Теперь, когда мы сложим эти две асимптотики, у нас получится $\log m$, умноженный на сумму n и m/n, и практически очевидно, что минимум достигается, когда $n \approx m/n$, т.е. для оптимальной работы алгоритма константу n следует выбирать так:

$$n \approx \sqrt{m}.$$

Тогда асимптотика алгоритма принимает вид:

$$O\left(\sqrt{m}\ \log m\right).$$

Примечание. Мы могли бы обменять ролями f_1 и f_2 (т.е. на первой фазе вычислять значения функции f_2, а а второй — f_1), однако легко понять, что результат от этого не изменится, и асимптотику этим мы никак не улучшим.

Реализация

Простейшая реализация

Функция powmod выполняет бинарное возведение числа a в степень b по модулю m, см. Бинарное возведение в степень.

Функция solve производит собственно решение задачи. Эта функция возвращает ответ (число в промежутке $[0; m)$), точнее говоря, один из ответов. Функция вернёт -1, если решения не существует.

```
int powmod (int a, int b, int m) {
        int res = 1;
        while (b > 0)
                if (b & 1) {
                        res = (res * a) % m;
                        --b;
                }
                else {
                        a = (a * a) % m;
                        b >>= 1;
                }
        return res % m;
}

int solve (int a, int b, int m) {
        int n = (int) sqrt (m + .0) + 1;
        map<int,int> vals;
        for (int i=n; i>=1; --i)
                vals[ powmod (a, i * n, m) ] = i;
        for (int i=0; i<=n; ++i) {
                int cur = (powmod (a, i, m) * b) % m;
                if (vals.count(cur)) {
                        int ans = vals[cur] * n - i;
                        if (ans < m)
                                return ans;
                }
        }
        return -1;
}
```

Здесь мы для удобства при реализации первой фазы алгоритма воспользовались структурой данных "map" (красно-чёрным деревом), которая для каждого значения функции $f_1(i)$ хранит аргумент i, при котором это значение достигалось. При этом если одно и то же значение достигалось несколько раз, записывается наименьший из всех аргументов. Это сделано для того, чтобы впоследствии, на второй фазе алгоритма, нашёлся ответ в промежутке $[0; m)$.

Учитывая, что аргумент функции $f_1()$ на первой фазе у нас перебирался от единицы и до n, а аргумент функции $f_2()$ на второй фазе перебирается от нуля до n, то в итоге мы покрываем всё множество возможных ответов, т.к.

отрезок $[0; n^2]$ содержит в себе промежуток $[0; m)$. При этом отрицательным ответ получиться не мог, а ответы, большие либо равные m мы можем игнорировать — всё равно должны находиться соответствующие им ответы из промежутка $[0; m)$.

Эту функцию можно изменить на тот случай, если требуется находить **все решения** задачи дискретного логарифма. Для этого надо заменить "map" на какую-либо другую структуру данных, позволяющую хранить для одного аргумента сразу несколько значений (например, "multimap"), и соответствующим образом изменить код второй фазы.

Улучшенная реализация

При **оптимизации по скорости** можно поступить следующим образом.

Во-первых, сразу бросается в глаза ненужность бинарного возведения в степень на второй фазе алгоритма. Вместо этого можно просто завести переменную и домножать её каждый раз на a.

Во-вторых, таким же образом можно избавиться от бинарного возведения в степень и на первой фазе: в самом деле, достаточно один раз посчитать величину a^n, и потом просто домножать на неё.

Таким образом, логарифм в асимптотике по-прежнему останется, но это будет только логарифм, связанный со структурой данных $map <>$ (т.е., в терминах алгоритма, с сортировкой и бинарным поиском значений) — т.е. это будет логарифм от $\sqrt{m}$, что на практике даёт заметное ускорение.

```
int solve (int a, int b, int m) {
        int n = (int) sqrt (m + .0) + 1;

        int an = 1;
        for (int i=0; i<n; ++i)
                an = (an * a) % m;

        map<int,int> vals;
        for (int i=1, cur=an; i<=n; ++i) {
                if (!vals.count(cur))
                        vals[cur] = i;
                cur = (cur * an) % m;
        }

        for (int i=0, cur=b; i<=n; ++i) {
                if (vals.count(cur)) {
                        int ans = vals[cur] * n - i;
                        if (ans < m)
                                return ans;
                }
                cur = (cur * a) % m;
        }
        return -1;
}
```

Наконец, если модуль m достаточно мал, то можно и вовсе избавиться от логарифма в асимптотике — просто заведя вместо $map <>$ обычный массив.

Также можно вспомнить про хеш-таблицы: в среднем они работают также за $O(1)$, что в целом даёт асимптотику $O(\sqrt{m})$.

Линейные диофантовы уравнения с двумя переменными

Диофантово уравнение с двумя неизвестными имеет вид:

$$a \cdot x + b \cdot y = c,$$

где a, b, c — заданные целые числа, x и y — неизвестные целые числа.

Ниже рассматриваются несколько классических задач на эти уравнения: нахождение любого решения, получение всех решений, нахождение количества решений и сами решения в определённом отрезке, нахождение решения с наименьшей суммой неизвестных.

Вырожденный случай

Один вырожденный случай мы сразу исключим из рассмотрения: когда $a = b = 0$. В этом случае, понятно, уравнение имеет либо бесконечно много произвольных решений, либо же не имеет решений вовсе (в зависимости от того, $c = 0$ или нет).

Нахождение одного решения

Найти одно из решений диофантова уравнения с двумя неизвестными можно с помощью Расширенного алгоритма Евклида. Предположим сначала, что числа a и b неотрицательны.

Расширенный алгоритм Евклида по заданным неотрицательным числам a и b находит их наибольший общий делитель g, а также такие коэффициенты x_g и y_g, что:

$$a \cdot x_g + b \cdot y_g = g.$$

Утверждается, что если c делится на $g = \gcd(a, b)$, то диофантово уравнение $a \cdot x + b \cdot y = c$ имеет решение; в противном случае диофантово уравнение решений не имеет. Доказательство следует из очевидного факта, что линейная комбинация двух чисел по-прежнему должна делиться на их общий делитель.

Предположим, что c делится на g, тогда, очевидно, выполняется:

$$a \cdot x_g \cdot (c/g) + b \cdot y_g \cdot (c/g) = c,$$

т.е. одним из решений диофантова уравнения являются числа:

$$\begin{cases} x_0 = x_g \cdot (c/g), \\ y_0 = y_g \cdot (c/g). \end{cases}$$

Мы описали решение в случае, когда числа a и b неотрицательны. Если же одно из них или они оба отрицательны, то можно поступить таким образом: взять их по модулю и применить к ним алгоритм Евклида, как было описано выше, а затем изменить знак найденных x_0 и y_0 в соответствии с настоящим знаком чисел a и b соответственно.

Реализация (напомним, здесь мы считаем, что входные данные $a = b = 0$ недопустимы):

```
int gcd (int a, int b, int & x, int & y) {
        if (a == 0) {
                x = 0; y = 1;
                return b;
        }
        int x1, y1;
        int d = gcd (b%a, a, x1, y1);
        x = y1 - (b / a) * x1;
        y = x1;
        return d;
}

bool find_any_solution (int a, int b, int c, int & x0, int & y0, int & g) {
        g = gcd (abs(a), abs(b), x0, y0);
        if (c % g != 0)
```

```
                return false;
        x0 *= c / g;
        y0 *= c / g;
        if (a < 0)   x0 *= -1;
        if (b < 0)   y0 *= -1;
        return true;
}
```

Получение всех решений

Покажем, как получить все остальные решения (а их бесконечное множество) диофантова уравнения, зная одно из решений (x_0, y_0).

Итак, пусть $g = \gcd(a, b)$, а числа x_0, y_0 удовлетворяют условию:

$$a \cdot x_0 + b \cdot y_0 = c.$$

Тогда заметим, что, прибавив к x_0 число b/g и одновременно отняв a/g от y_0, мы не нарушим равенства:

$$a \cdot (x_0 + b/g) + b \cdot (y_0 - a/g) = a \cdot x_0 + b \cdot y_0 + a \cdot b/g - b \cdot a/g = c.$$

Очевидно, что этот процесс можно повторять сколько угодно, т.е. все числа вида:

$$\begin{cases} x = x_0 + k \cdot b/g, \\ y = y_0 - k \cdot a/g, \end{cases} \quad k \in Z$$

являются решениями диофантова уравнения.

Более того, только числа такого вида и являются решениями, т.е. мы описали множество всех решений диофантова уравнения (оно получилось бесконечным, если не наложено дополнительных условий).

Нахождение количества решений и сами решения в заданном отрезке

Пусть даны два отрезка $[min_x; max_x]$ и $[min_y; max_y]$, и требуется найти количество решений (x, y) диофантова уравнения, лежащих в данных отрезках соответственно.

Заметим, что если одно из чисел a, b равно нулю, то задача имеет не больше одного решения, поэтому эти случаи мы в данном разделе исключаем из рассмотрения.

Сначала найдём решение с минимальным подходящим x, т.е. $x \geq min_x$. Для этого сначала найдём любое решение диофантова уравнения (см. пункт 1). Затем получим из него решение с наименьшим $x \geq min_x$ — для этого воспользуемся процедурой, описанной в предыдущем пункте, и будем уменьшать/увеличивать x, пока оно не окажется $\geq min_x$, и при этом минимальным. Это можно сделать за $O(1)$, посчитав, с каким коэффициентом нужно применить это преобразование, чтобы получить минимальное число, большее либо равное min_x. Обозначим найденный x через $lx1$.

Аналогичным образом можно найти и решение с максимальным подходящим $x = rx1$, т.е. $x \leq max_x$.

Далее перейдём к удовлетворению ограничений на y, т.е. к рассмотрению отрезка $[min_y; max_y]$. Способом, описанным выше, найдём решение с минимальным $y \geq min_y$, а также решение с максимальным $y \leq max_y$. Обозначим x-коэффициенты этих решений через $lx2$ и $rx2$ соответственно.

Пересечём отрезки $[lx1; rx1]$ и $[lx2; rx2]$; обозначим получившийся отрезок через $[lx; rx]$. Утверждается, что любое решение, у которого x-коэффициент лежит в $[lx; rx]$ — любое такое решение является подходящим. (Это верно в силу построения этого отрезка: сначала мы отдельно удовлетворили ограничения на x и y, получив два отрезка, а затем пересекли их, получив область, в которой удовлетворяются оба условия.)

Таким образом, количество решений будет равняться длине этого отрезка, делённой на $|b|$ (поскольку x-коэффициент может изменяться только на $\pm b$), и плюс один.

Приведём реализацию (она получилась достаточно сложной, поскольку требуется аккуратно рассматривать случаи положительных и отрицательных коэффициентов a и b):

```
void shift_solution (int & x, int & y, int a, int b, int cnt) {
        x += cnt * b;
        y -= cnt * a;
}
```

```
int find_all_solutions (int a, int b, int c, int minx, int maxx, int miny,
int maxy) {
        int x, y, g;
        if (! find_any_solution (a, b, c, x, y, g))
                return 0;
        a /= g;  b /= g;

        int sign_a = a>0 ? +1 : -1;
        int sign_b = b>0 ? +1 : -1;

        shift_solution (x, y, a, b, (minx - x) / b);
        if (x < minx)
                shift_solution (x, y, a, b, sign_b);
        if (x > maxx)
                return 0;
        int lx1 = x;

        shift_solution (x, y, a, b, (maxx - x) / b);
        if (x > maxx)
                shift_solution (x, y, a, b, -sign_b);
        int rx1 = x;

        shift_solution (x, y, a, b, - (miny - y) / a);
        if (y < miny)
                shift_solution (x, y, a, b, -sign_a);
        if (y > maxy)
                return 0;
        int lx2 = x;

        shift_solution (x, y, a, b, - (maxy - y) / a);
        if (y > maxy)
                shift_solution (x, y, a, b, sign_a);
        int rx2 = x;

        if (lx2 > rx2)
                swap (lx2, rx2);
        int lx = max (lx1, lx2);
        int rx = min (rx1, rx2);

        return (rx - lx) / abs(b) + 1;
}
```

Также нетрудно добавить к этой реализации вывод всех найденных решений: для этого достаточно перебрать x в отрезке $[lx; rx]$ с шагом $|b|$, найдя для каждого из них соответствующий y непосредственно из уравнения $ax + by = c$.

Нахождение решения в заданном отрезке с наименьшей суммой x+y

Здесь на x и на y также должны быть наложены какие-либо ограничения, иначе ответом практически всегда будет минус бесконечность.

Идея решения такая же, как и в предыдущем пункте: сначала находим любое решение диофантова уравнения, а затем, применяя описанную в предыдущем пункте процедуру, придём к наилучшему решению.

Действительно, мы имеем право выполнить следующее преобразование (см. предыдущий пункт):

$$\begin{cases} x' = x + k \cdot (b/g), \\ y' = y - k \cdot (a/g), \end{cases} \qquad k \in Z.$$

Заметим, что при этом сумма $x + y$ меняется следующим образом:

$$x' + y' = x + y + k \cdot (b/g - a/g) = x + y + k \cdot (b - a)/g.$$

Т.е. если $a < b$, то нужно выбрать как можно меньшее значение k, если $a > b$, то нужно выбрать как можно большее значение k.

Если $a = b$, то мы никак не сможем улучшить решение, — все решения будут обладать одной и той же суммой.

Задачи в online judges

Список задач, которые можно сдать на тему диофантовых уравнений с двумя неизвестными:

- SGU #106 "The Equation" [сложность: средняя]

Модульное линейное уравнение первого порядка

Постановка задачи

Это уравнение вида:

$$a \cdot x = b \pmod n,$$

где a, b, n — заданные целые числа, x — неизвестное целое число.

Требуется найти искомое значение x, лежащее в отрезке $[0; n-1]$ (поскольку на всей числовой прямой, ясно, может существовать бесконечно много решений, которые будут отличаться друг друга на $n \cdot k$, где k — любое целое число). Если решение не единственно, то мы рассмотрим, как получить все решения.

Решение с помощью нахождения Обратного элемента

Рассмотрим сначала более простой случай — когда a и n **взаимно просты**. Тогда можно найти обратный элемент к числу a, и, домножив на него обе части уравнения, получить решение (и оно будет **единственным**):

$$x = b \cdot a^{-1} \pmod n$$

Теперь рассмотрим случай, когда a и n **не взаимно просты**. Тогда, очевидно, решение будет существовать не всегда (например, $2 \cdot x = 1 \pmod 4$).

Пусть $g = \gcd(\mathrm{a}, \mathrm{n})$, т.е. их наибольший общий делитель (который в данном случае больше единицы).

Тогда, если b не делится на g, то решения не существует. В самом деле, при любом x левая часть уравнения, т. е. $(a \cdot x) \pmod n$, всегда делится на g, в то время как правая часть на него не делится, откуда и следует, что решений нет.

Если же b делится на g, то, разделив обе части уравнения на это g (т.е. разделив a, b и n на g), мы придём к новому уравнению:

$$a' \cdot x = b' \pmod{n'}$$

в котором a' и n' уже будут взаимно просты, а такое уравнение мы уже научились решать. Обозначим его решение через x'.

Понятно, что это x' будет также являться и решением исходного уравнения. Однако если $g > 1$, то оно будет **не единственным** решением. Можно показать, что исходное уравнение будет иметь ровно g решений, и они будут иметь вид:

$$x_i = (x' + i \cdot n') \pmod n,$$
$$i = 0 \ldots (g-1).$$

Подводя итог, можно сказать, что **количество решений** линейного модульного уравнения равно либо $g = \gcd(\mathrm{a}, \mathrm{n})$, либо нулю.

Решение с помощью Расширенного алгоритма Евклида

Приведём наше модулярное уравнение к диофантову уравнению следующим образом:

$$a \cdot x + n \cdot k = b,$$

где x и k — неизвестные целые числа.

Способ решения этого уравнения описан в соответствующей статье Линейные диофантовы уравнения второго порядка, и заключается он в применении Расширенного алгоритма Евклида.

Там же описан и способ получения всех решений этого уравнения по одному найденному решению, и, кстати говоря, этот способ при внимательном рассмотрении абсолютно эквивалентен способу, описанному в предыдущем пункте.

Китайская теорема об остатках

Формулировка

В своей современной формулировке теорема звучит так:

Пусть $p = p_1 \cdot p_2 \cdot \ldots \cdot p_k$, где p_i — попарно взаимно простые числа.

Поставим в соответствие произвольному числу $a\,(0 \le a < p)$ кортеж $(a_1, \ldots, a_k)$, где $a_i \equiv a \pmod{p_i}$:

$$a \Longleftrightarrow (a_1, \ldots, a_k).$$

Тогда это соответствие (между числами и кортежами) будет являться **взаимно однозначным**. И, более того, операции, выполняемые над числом a, можно эквивалентно выполнять над соответствующими элементами кортежами — путём независимого выполнения операций над каждым компонентом.

Т.е., если

$$a \Longleftrightarrow \Big(a_1, \ldots, a_k\Big),$$
$$b \Longleftrightarrow \Big(b_1, \ldots, b_k\Big),$$

то справедливо:

$$(a+b) \pmod{p} \Longleftrightarrow \Big((a_1+b_1) \pmod{p_1}, \ldots, (a_k+b_k) \pmod{p_k}\Big),$$
$$(a-b) \pmod{p} \Longleftrightarrow \Big((a_1-b_1) \pmod{p_1}, \ldots, (a_k-b_k) \pmod{p_k}\Big),$$
$$(a \cdot b) \pmod{p} \Longleftrightarrow \Big((a_1 \cdot b_1) \pmod{p_1}, \ldots, (a_k \cdot b_k) \pmod{p_k}\Big).$$

В своей первоначальной формулировке эта теорема была доказана китайским математиком Сунь-Цзы приблизительно в 100 г. н.э. А именно, он показал в частном случае эквивалентность решения системы модулярных уравнений и решения одного модулярного уравнения (см. следствие 2 ниже).

Следствие 1

Система модулярных уравнений:

$$\begin{cases} x \equiv a_1 \pmod{p_1}, \\ \ldots, \\ x \equiv a_k \pmod{p_k} \end{cases}$$

имеет единственное решение по модулю p.

(как и выше, $p = p_1 \cdot \ldots \cdot p_k$, числа p_i попарно взаимно просты, а набор $a_1, \ldots, a_k$ — произвольный набор целых чисел)

Следствие 2

Следствием является связь между системой модулярных уравнений и одним соответствующим модулярным уравнением:

Уравнение:

$$x \equiv a \pmod{p}$$

эквивалентно системе уравнений:

$$\begin{cases} x \equiv a \pmod{p_1}, \\ \ldots, \\ x \equiv a \pmod{p_k} \end{cases}$$

(как и выше, предполагается, что $p = p_1 \cdot \ldots \cdot p_k$, числа p_i попарно взаимно просты, а a — произвольное целое число)

Алгоритм Гарнера

Из китайской теоремы об остатках следует, что можно заменять операции над числами операциями над кортежами. Напомним, каждому числу a ставится в соответствие кортеж $(a_1, \ldots, a_k)$, где:

$$a_i \equiv a \pmod{p_i}.$$

Это может найти широкое применение на практике (помимо непосредственного применения для восстановления числа по его остаткам по различным модулям), поскольку мы таким образом можем заменять операции в длинной арифметике операциями с массивом "коротких" чисел. Скажем, массива из 1000 элементов "хватит" на числа примерно с 3000 знаками (если выбрать в качестве p_i-ых первые 1000 простых); а если выбирать в качестве p_i-ых простые около миллиарда, то тогда хватит уже на число с примерно 9000 знаками. Но, разумеется, тогда нужно научиться **восстанавливать** число a по этому кортежу. Из следствия 1 видно, что такое восстановление возможно, и притом единственно (при условии $0 \le a < p_1 \cdot p_2 \cdot \ldots \cdot p_k$).
Алгоритм Гарнера и является алгоритмом, позволяющим выполнить это восстановление, причём достаточно эффективно.

Будем искать решение в виде:

$$a = x_1 + x_2 \cdot p_1 + x_3 \cdot p_1 \cdot p_2 + \ldots + x_k \cdot p_1 \cdot \ldots \cdot p_{k-1},$$

т.е. в смешанной системе счисления с весами разрядов $p_1, p_2, \ldots, p_k$.

Обозначим через r_{ij} ($i = 1 \ldots k-1$, $j = i+1 \ldots k$) число, являющееся обратным для p_i по модулю p_j (нахождение обратных элементов в кольце по модулю описано здесь:

$$r_{ij} = (p_i)^{-1} \pmod{p_j}.$$

Подставим выражение a в смешанной системе счисления в первое уравнение системы, получим:

$$a_1 \equiv x_1.$$

Подставим теперь выражение во второе уравнение:

$$a_2 \equiv x_1 + x_2 \cdot p_1 \pmod{p_2}.$$

Преобразуем это выражение, отняв от обеих частей x_1 и разделив на n_1:

$$\begin{aligned}
&a_2 - x_1 \equiv x_2 \cdot p_1 \pmod{p_2};\\
&(a_2 - x_1) \cdot r_{12} \equiv x_2 \pmod{p_2};\\
&x_2 \equiv (a_2 - x_1) \cdot r_{12} \pmod{p_2}.
\end{aligned}$$

Подставляя в третье уравнение, аналогичным образом получаем:

$$\begin{aligned}
&a_3 \equiv x_1 + x_2 \cdot p_1 + x_3 \cdot p_1 \cdot p_2 \pmod{p_3};\\
&(a_3 - x_1) \cdot r_{13} \equiv x_2 + x_3 \cdot p_2 \pmod{p_3};\\
&((a_3 - x_1) \cdot r_{13} - x_2) \cdot r_{23} \equiv x_3 \pmod{p_3};\\
&x_3 \equiv ((a_3 - x_1) \cdot r_{13} - x_2) \cdot r_{23} \pmod{p_3}.
\end{aligned}$$

Уже достаточно ясно видна закономерность, которую проще всего выразить кодом:

```
for (int i=0; i<k; ++i) {
        x[i] = a[i];
        for (int j=0; j<i; ++j) {
                x[i] = r[j][i] * (x[i] - x[j]);

                x[i] = x[i] % p[i];
                if (x[i] < 0)  x[i] += p[i];
        }
}
```

Итак, мы научились вычислять коэффициенты x_i за время $O(k^2)$, сам же ответ — число a — можно восстановить по формуле:

$$a = x_1 + x_2 \cdot p_1 + x_3 \cdot p_1 \cdot p_2 + \ldots + x_k \cdot p_1 \cdot \ldots \cdot p_{k-1}.$$

Стоит заметить, что на практике почти всегда вычислять ответ нужно с помощью Длинной арифметики, но при этом сами коэффициенты x_i по-прежнему вычисляются на встроенных типах, а потому весь алгоритм Гарнера является весьма эффективным.

Реализация алгоритма Гарнера

Удобнее всего реализовывать этот алгоритм на языке Java, поскольку она содержит стандартную длинную арифметику, а потому не возникает никаких проблем с переводом числа из модульной системы в обычное число (используется стандартный класс BigInteger).

Приведённая ниже реализация алгоритма Гарнера поддерживает сложение, вычитание и умножение, причём поддерживает работу с отрицательными числами (об этом см. пояснения после кода). Реализован перевод числа обычного десятичкого представления в модулярную систему и наоборот.

В данном примере берутся 100 простых после 10^9, что позволяет работать с числами до примерно 10^{900}.

```
final int SZ = 100;
int pr[] = new int[SZ];
int r[][] = new int[SZ][SZ];

void init() {
        for (int x=1000*1000*1000, i=0; i<SZ; ++x)
                if (BigInteger.valueOf(x).isProbablePrime(100))
                        pr[i++] = x;

        for (int i=0; i<SZ; ++i)
                for (int j=i+1; j<SZ; ++j)
                        r[i][j] = BigInteger.valueOf( pr[i] ).modInverse(
                                        BigInteger.valueOf( pr
[j] ) ).intValue();
}


class Number {

        int a[] = new int[SZ];

        public Number() {
        }

        public Number (int n) {
                for (int i=0; i<SZ; ++i)
                        a[i] = n % pr[i];
        }

        public Number (BigInteger n) {
                for (int i=0; i<SZ; ++i)
                        a[i] = n.mod( BigInteger.valueOf( pr
[i] ) ).intValue();
        }

        public Number add (Number n) {
                Number result = new Number();
                for (int i=0; i<SZ; ++i)
                        result.a[i] = (a[i] + n.a[i]) % pr[i];
                return result;
        }

        public Number subtract (Number n) {
                Number result = new Number();
                for (int i=0; i<SZ; ++i)
                        result.a[i] = (a[i] - n.a[i] + pr[i]) % pr[i];
                return result;
        }

        public Number multiply (Number n) {
                Number result = new Number();
                for (int i=0; i<SZ; ++i)
                        result.a[i] = (int)( (a[i] * 1l * n.a[i]) %
pr[i] );
                return result;
        }
```

```java
        public BigInteger bigIntegerValue (boolean can_be_negative) {
                BigInteger result = BigInteger.ZERO,
                        mult = BigInteger.ONE;
                int x[] = new int[SZ];
                for (int i=0; i<SZ; ++i) {
                        x[i] = a[i];
                        for (int j=0; j<i; ++j) {
                                long cur = (x[i] - x[j]) * 1l * r[j][i];
                                x[i] = (int)( (cur % pr[i] + pr[i]) %
pr[i] );
                        }
                        result = result.add( mult.multiply
( BigInteger.valueOf( x[i] ) ) );
                        mult = mult.multiply( BigInteger.valueOf
( pr[i] ) );
                }

                if (can_be_negative)
                        if (result.compareTo( mult.shiftRight(1) ) >= 0)
                                result = result.subtract( mult );

                return result;
        }
}
```

О поддержке **отрицательных** чисел следует сказать особо (флаг $\mathrm{can_be_negative}$ функции $\mathrm{bigIntegerValue}()$). Сама модулярная схема не предполагает различий между положительными и отрицательными числами. Однако можно заметить, что, если в конкретной задаче ответ по модулю не превосходит половины от произведения всех простых, то положительные числа будут отличаться от отрицательных тем, что положительные числа получатся меньше этой середины, а отрицательные — больше. Поэтому мы после классического алгоритма Гарнера сравниваем результат с серединой, и если он больше, то выводим минус, и инвертируем результат (т.е. отнимаем его от произведения всех простых, и выводим уже его).

Нахождение степени делителя факториала

Даны два числа: n и k. Требуется посчитать, с какой степенью делитель k входит в число $n!$, т.е. найти наибольшее x такое, что $n!$ делится на k^x.

Решение для случая простого k

Рассмотрим сначала случай, когда k простое.

Выпишем выражение для факториала в явном виде:

$$n! = 1\ 2\ 3\ \ldots\ (n-1)\ n$$

Заметим, что каждый k-ый член этого произведения делится на k, т.е. даёт +1 к ответу; количество таких членов равно $\lfloor n/k \rfloor$.

Далее, заметим, что каждый k^2-ый член этого ряда делится на k^2, т.е. даёт ещё +1 к ответу (учитывая, что k в первой степени уже было учтено до этого); количество таких членов равно $\lfloor n/k^2 \rfloor$.

И так далее, каждый k^i-ый член ряда даёт +1 к ответу, а количество таких членов равно $\lfloor n/k^i \rfloor$.

Таким образом, ответ равен величине:

$$\frac{n}{k} + \frac{n}{k^2} + \ldots + \frac{n}{k^i} + \ldots$$

Эта сумма, разумеется, не бесконечная, т.к. только первые примерно $\log_k n$ членов отличны от нуля. Следовательно, асимптотика такого алгоритма равна $O(\log_k n)$.

Реализация:

```
int fact_pow (int n, int k) {
        int res = 0;
        while (n) {
                n /= k;
                res += n;
        }
        return res;
}
```

Решение для случая составного k

Ту же идею применить здесь непосредственно уже нельзя.

Но мы можем факторизовать k, решить задачу для каждого его простого делителя, а потом выбрать минимум из ответов.

Более формально, пусть k_i — это i-ый делитель числа k, входящий в него в степени p_i. Решим задачу для k_i с помощью вышеописанной формулы за $O(\log n)$; пусть мы получили ответ Ans_i. Тогда ответом для составного k будет минимум из величин Ans_i / p_i.

Учитывая, что факторизация простейшим образом выполняется за $O(\sqrt{n})$, получаем итоговую асимптотику $O(\sqrt{n})$.

Троичная сбалансированная система счисления

Троичная сбалансированная система счисления — это нестандартная позиционная система счисления. Основание системы равно 3, однако она отличается от обычной троичной системы тем, что цифрами являются $-1, 0, 1$. Поскольку использовать -1 для одной цифры очень неудобно, то обычно принимают какое-то специальное обозначение. Условимся здесь обозначать минус единицу буквой z.

Например, число 5 в троичной сбалансированной системе записывается как $1zz$, а число -5 — как $z11$. Троичная сбалансированная система счисления позволяет записывать отрицательные числа без записи отдельного знака "минус". Троичная сбалансированная система позволяет дробные числа (например, $1/3$ записывается как 0.1).

Алгоритм перевода

Научимся переводить числа в троичную сбалансированную систему.

Для этого надо сначала перевести число в троичную систему.

Ясно, что теперь нам надо избавиться от цифр 2, для чего заметим, что $2 = 3 - 1$, т.е. мы можем заменить двойку в текущем разряде на -1, при этом увеличив следующий (т.е. слева от него в естественной записи) разряд на 1. Если мы будем двигаться по записи справа налево и выполнять вышеописанную операцию (при этом в каких-то разрядах может происходить переполнение больше 3, в таком случае, естественно, "сбрасываем" лишние тройки в старший разряд), то придём к троичной сбалансированной записи. Как нетрудно убедиться, то же самое правило верно и для дробных чисел.

Более изящно вышеописанную процедуру можно описать так. Мы берём число в троичной системе счисления, прибавляем к нему бесконечное число $\ldots 11111.11111\ldots$, а затем от каждого разряда результата отнимаем единицу (уже безо всяких переносов).

Зная теперь алгоритм перевода из обычной троичной системы в сбалансированную, легко можно реализовать операции сложения, вычитания и деления — просто сводя их к соответствующим операциям над троичными несбалансированными числами.

Вычисление факториала по модулю

В некоторых случаях необходимо считать по некоторому простому модулю p сложные формулы, которые в том числе могут содержать факториалы. Здесь мы рассмотрим случай, когда модуль p сравнительно мал. Понятно, что эта задача имеет смысл только в том случае, когда факториалы входят и в числитель, и в знаменатель дробей. Действительно, факториал $p!$ и все последующие обращаются в ноль по модулю p, однако в дробях все множители, содержащие p, могут сократиться, и полученное выражение уже будет отлично от нуля по модулю p.

Таким образом, формально **задача** такая. Требуется вычислить $n!$ по простому модулю p, при этом не учитывая все кратные p множители, входящие в факториал. Научившись эффективно вычислять такой факториал, мы сможем быстро вычислять значение различных комбинаторных формул (например, Биномиальные коэффициенты).

Алгоритм

Выпишем этот "модифицированный" факториал в явном виде:

$$
\begin{aligned}
&n!_{\%p} = \\
&= 1\cdot 2\cdot 3\cdot \ldots \cdot (p-2)\cdot (p-1)\cdot \underbrace{1}_{p}\cdot (p+1)\cdot (p+2)\cdot \ldots \cdot (2p-1)\cdot \underbrace{2}_{2p}\cdot (2p+1)\cdot \ldots \cdot \\
&\cdot (p^2-1)\cdot \underbrace{1}_{p^2}\cdot (p^2+1)\cdot \ldots \cdot n = \\
&= 1\cdot 2\cdot 3\cdot \ldots \cdot (p-2)\cdot (p-1)\cdot \cdot \underbrace{1}_{p}\cdot 1\cdot 2\cdot \ldots \cdot (p-1)\cdot \underbrace{2}_{2p}\cdot 1\cdot 2\cdot \ldots \cdot (p-1)\cdot \underbrace{1}_{p^2}\cdot \\
&\cdot 1\cdot 2\cdot \ldots \cdot (n\%p) \pmod p.
\end{aligned}
$$

При такой записи видно, что "модифицированный" факториал распадается на несколько блоков длины p (последний блок, возможно, короче), которые все одинаковы, за исключением последнего элемента:

$$
\begin{aligned}
&n!_{\%p} = \underbrace{1\cdot 2\cdot \ldots \cdot (p-2)\cdot (p-1)\cdot 1}_{1st}\cdot \underbrace{1\cdot 2\cdot \ldots \cdot (p-1)\cdot 2}_{2nd}\cdot \ldots \cdot \underbrace{1\cdot 2\cdot \ldots \cdot (p-1)\cdot 1}_{p-th}\cdot \ldots \cdot \\
&\cdot \underbrace{1\cdot 2\cdot \ldots \cdot (n\%p)}_{tail} \pmod p.
\end{aligned}
$$

Общую часть блоков посчитать легко — это просто $(p-1)! \bmod p$, которую можно посчитать программно или по теореме Вильсона (Wilson) сразу найти $(p-1)! \bmod p = p-1$. Чтобы перемножить эти общие части всех блоков, надо найденную величину возвести в степень по модулю p, что можно сделать за $O(\log n)$ операций (см. Бинарное возведение в степень; впрочем, можно заметить, что мы фактически возводим минус единицу в какую-то степень, а потому результатом всегда будет либо 1, либо $p-1$, в зависимости от чётности показателя. Значение в последнем, неполном блоке тоже можно посчитать отдельно за $O(p)$. Остались только последние элементы блоков, рассмотрим их внимательнее:

$$
n!_{\%p} = \underbrace{\ldots \cdot 1}\cdot \underbrace{\ldots \cdot 2}\cdot \underbrace{\ldots \cdot 3}\cdot \ldots \cdot \underbrace{\ldots \cdot (p-1)}\cdot \underbrace{\ldots \cdot 1}\cdot \underbrace{\ldots \cdot 1}\cdot \underbrace{\ldots \cdot 2}\cdot \ldots
$$

И мы снова пришли к "модифицированному" факториалу, но уже меньшей размерности (столько, сколько было полных блоков, а их было $\lfloor n/p \rfloor$). Таким образом, вычисление "модифицированного" факториала $n!_{\%p}$ мы свели за $O(p)$ операций к вычислению уже $(n/p)!_{\%p}$. Раскрывая эту рекуррентную зависимость, мы получаем, что глубина рекурсии будет $O(\log_p n)$, итого **асимптотика** алгоритма получается $O(p \log_p n)$.

Реализация

Понятно, что при реализации не обязательно использовать рекурсию в явном виде: поскольку рекурсия хвостовая, её легко развернуть в цикл.

```
int factmod (int n, int p) {
        int res = 1;
        while (n > 1) {
                res = (res * ((n/p) % 2 ? p-1 : 1)) % p;
                for (int i=2; i<=n%p; ++i)
```

```
                        res = (res * i) % p;
                n /= p;
        }
        return res % p;
}
```

Эта реализация работает за $O(p \log_p n)$.

Перебор всех подмасок данной маски

Перебор подмасок фиксированной маски

Дана битовая маска m. Требуется эффективно перебрать все её подмаски, т.е. такие маски s, в которых могут быть включены только те биты, которые были включены в маске m.

Сразу рассмотрим реализацию этого алгоритма, основанную на трюках с битовыми операциями:

```
int s = m;
while (s > 0) {
        ... можно использовать s ...
        s = (s-1) & m;
}
```

или, используя более компактный оператор for:

```
for (int s=m; s; s=(s-1)&m)
        ... можно использовать s ...
```

Единственное исключение для обоих вариантов кода — подмаска, равная нулю, обработана не будет. Её обработку придётся выносить из цикла, или использовать менее изящную конструкцию, например:

```
for (int s=m; ; s=(s-1)&m) {
        ... можно использовать s ...
        if (s==0)  break;
}
```

Разберём, почему приведённый выше код действительно находит все подмаски данной маски, причём без повторений, за O (их количества), и в порядке убывания.

Пусть у нас есть текущая подмаска s, и мы хотим перейти к следующей подмаске. Отнимем от маски s единицу, тем самым мы снимем самый правый единичный бит, а все биты правее него поставятся в 1. Затем удалим все "лишние" единичные биты, которые не входят в маску m и потому не могут входить в подмаску. Удаление осуществляется битовой операцией $\&m$. В результате мы "обрежем" маску $s-1$ до того наибольшего значения, которое она может принять, т.е. до следующей подмаски после s в порядке убывания.

Таким образом, этот алгоритм генерирует все подмаски данной маски в порядке строгого убывания, затрачивая на каждый переход по две элементарные операции.

Особо рассмотрим момент, когда $s=0$. После выполнения $s-1$ мы получим маску, в которой все биты включены (битовое представление числа -1), и после удаления лишних битов операцией $(s-1)\&m$ получится не что иное, как маска m. Поэтому с маской $s=0$ следует быть осторожным — если вовремя не остановиться на нулевой маске, то алгоритм может войти в бесконечный цикл.

Перебор всех масок с их подмасками. Оценка 3^n

Во многих задачах, особенно на динамическое программирование по маскам, требуется перебирать все маски, и для каждой маски - все подмаски:

```
for (int m=0; m<(1<<n); ++m)
        for (int s=m; s; s=(s-1)&m)
                ... использование s и m ...
```

Докажем, что внутренний цикл суммарно выполнит $O(3^n)$ итераций.

Доказательство: 1 **способ**. Рассмотрим i-ый бит. Для него, вообще говоря, есть ровно три варианта: он не входит в маску m (и потому в подмаску s); он входит в m, но не входит в s; он входит в m и в s. Всего битов n, поэтому всего различных комбинаций будет 3^n, что и требовалось доказать.

Доказательство: 2 **способ**. Заметим, что если маска m имеет k включённых битов, то она будет иметь

2^k подмасок. Поскольку масок длины n с k включёнными битами есть C_n^k (см. "биномиальные коэффициенты"), то всего комбинаций будет:

$$\sum_{k=0}^{n} C_n^k 2^k.$$

Посчитаем эту сумму. Для этого заметим, что она есть не что иное, как разложение в бином Ньютона выражения $(1+2)^n$, т.е. 3^n, что и требовалось доказать.

Первообразные корни

Определение

Первообразным корнем по модулю n (primitive root modulo n) называется такое число g, что все его степени по модулю n пробегают по всем числам, взаимно простым с n. Математически это формулируется таким образом: если g является первообразным корнем по модулю n, то для любого целого a такого, что $\gcd(a, n) = 1$, найдётся такое целое k, что $g^k \equiv a \pmod n$.

В частности, для случая простого n степени первообразного корня пробегают по всем числам от 1 до $n - 1$.

Существование

Первообразный корень по модулю n существует тогда и только тогда, когда n является либо степенью нечётного простого, либо удвоенной степенью простого, а также в случаях $n = 1, n = 2, n = 4$.

Эта теорема (которая была полностью доказана Гауссом в 1801 г.) приводится здесь без доказательства.

Связь с функцией Эйлера

Пусть g - первообразный корень по модулю n. Тогда можно показать, что наименьшее число k, для которого $g^k \equiv 1 \pmod n$ (т.е. k — показатель g (multiplicative order)), равно $\phi(n)$. Более того, верно и обратное, и этот факт будет использован нами ниже в алгоритме нахождения первообразного корня.

Кроме того, если по модулю n есть хотя бы один первообразный корень, то всего их $\phi(\phi(n))$ (т.к. циклическая группа с k элементами имеет $\phi(k)$ генераторов).

Алгоритм нахождения первообразного корня

Наивный алгоритм потребует для каждого тестируемого значения g $O(n)$ времени, чтобы вычислить все его степени и проверить, что они все различны. Это слишком медленный алгоритм, ниже мы с помощью нескольких известных теорем из теории чисел получим более быстрый алгоритм.

Выше была приведена теорема о том, что если наименьшее число k, для которого $g^k \equiv 1 \pmod n$ (т.е. k — показатель g), равно $\phi(n)$, то g — первообразный корень. Так как для любого числа a выполняется теорема Эйлера ($a^{\phi(n)} \equiv 1 \pmod n$), то чтобы проверить, что g первообразный корень, достаточно проверить, что для всех чисел d, меньших $\phi(n)$, выполнялось $g^d \not\equiv 1 \pmod n$. Однако пока это слишком медленный алгоритм.

Из теоремы Лагранжа следует, что показатель любого числа по модулю n является делителем $\phi(n)$. Таким образом, достаточно проверить, что для всех собственных делителей $d \mid \phi(n)$ выполняется $g^d \not\equiv 1 \pmod n$. Это уже значительно более быстрый алгоритм, однако можно пойти ещё дальше.

Факторизуем число $\phi(n) = p_1^{a_1} \dots p_s^{a_s}$. Докажем, что в предыдущем алгоритме достаточно рассматривать в качестве d лишь числа вида $\frac{\phi(n)}{p_i}$. Действительно, пусть d — произвольный собственный делитель $\phi(n)$. Тогда, очевидно, найдётся такое j, что $d \mid \frac{\phi(n)}{p_j}$, т.е. $d \cdot k = \frac{\phi(n)}{p_j}$. Однако, если бы $g^d \equiv 1 \pmod n$, то мы получили бы:

$$g^{\frac{\phi(n)}{p_j}} \equiv g^{d \cdot k} \equiv \left(g^d\right)^k \equiv 1^k \equiv 1 \pmod n,$$

т.е. всё равно среди чисел вида $\frac{\phi(n)}{p_i}$ нашлось бы то, для которого условие не выполнилось, что и требовалось доказать.

Таким образом, алгоритм нахождения первообразного корня такой. Находим $\phi(n)$, факторизуем его. Теперь перебираем все числа $g = 1 \dots n$, и для каждого считаем все величины $g^{\frac{\phi(n)}{p_i}} \pmod n$. Если для текущего g все эти числа оказались отличными от 1, то это g и является искомым первообразным корнем.

Время работы алгоритма (считая, что у числа $\phi(n)$ имеется $O(\log \phi(n))$ делителей, а возведение в

степень выполняется алгоритмом Бинарного возведения в степень, т.е. за $O(\log n)$) равно $O\left(\mathrm{Ans} \cdot \log \phi(n) \cdot \log n\right)$ плюс время факторизации числа $\phi(n)$, где Ans — результат, т.е. значение искомого первообразного корня.

Про скорость роста первообразных корней с ростом n известны лишь приблизительные оценки. Известно, что первообразные корни — сравнительно небольшие величины. Одна из известных оценок — оценка Шупа (Shoup), что, в предположении истинности гипотезы Римана, первообразный корень есть $O(\log^6 n)$.

Реализация

Функция powmod() выполняет бинарное возведение в степень по модулю, а функция generator (int p) - находит первообразный корень по простому модулю p (факторизация числа $\phi(n)$ здесь осуществлена простейшим алгоритмом за $O(\sqrt{\phi(n)})$). Чтобы адаптировать эту функцию для произвольных p, достаточно добавить вычисление функции Эйлера в переменной phi.

```
int powmod (int a, int b, int p) {
        int res = 1;
        while (b)
                if (b & 1)
                        res = int (res * 1ll * a % p),  --b;
                else
                        a = int (a * 1ll * a % p),  b >>= 1;
        return res;
}

int generator (int p) {
        vector<int> fact;
        int phi = p-1,  n = phi;
        for (int i=2; i*i<=n; ++i)
                if (n % i == 0) {
                        fact.push_back (i);
                        while (n % i == 0)
                                n /= i;
                }
        if (n > 1)
                fact.push_back (n);

        for (int res=2; res<=p; ++res) {
                bool ok = true;
                for (size_t i=0; i<fact.size() && ok; ++i)
                        ok &= powmod (res, phi / fact[i], p) != 1;
                if (ok)  return res;
        }
        return -1;
}
```

Дискретное извлечение корня

Задача дискретного извлечения корня (по аналогии с задачей дискретного логарифма) звучит следующим образом.
По данным n (n — простое), a, k требуется найти все x, удовлетворяющие условию:

$$x^k \equiv a \pmod n$$

Алгоритм решения

Решать задачу будем сведением её к задаче дискретного логарифма.

Для этого применим понятие Первообразного корня по модулю n. Пусть g — первообразный корень по модулю n (т.к. n — простое, то он существует). Найти его мы можем, как описано в соответствующей статье, за $O(\text{Ans} \cdot \log \phi(n) \cdot \log n) = O(\text{Ans} \cdot \log^2 n)$ плюс время факторизации числа $\phi(n)$.

Отбросим сразу случай, когда $a = 0$ — в этом случае сразу находим ответ $x = 0$.

Поскольку в данном случае (n — простое) любое число от 1 до $n - 1$ представимо в виде степени первообразного корня, то задачу дискретного корня мы можем представить в виде:

$$(g^y)^k \equiv a \pmod n$$

где

$$x \equiv g^y \pmod n$$

Тривиальным преобразованием получаем:

$$\left(g^k\right)^y \equiv a \pmod n$$

Здесь искомой величиной является y, таким образом, мы пришли к задаче дискретного логарифмирования в чистом виде. Эту задачу можно решить алгоритмом baby-step-giant-step Шэнкса за $O(\sqrt{n} \log n)$, т.е. найти одно из решений y_0 этого уравнения (или обнаружить, что это уравнение решений не имеет).

Пусть мы нашли некоторое решение y_0 этого уравнения, тогда одним из решений задачи дискретного корня будет $x_0 = g^{y_0} \pmod n$.

Нахождение всех решений, зная одно из них

Чтобы полностью решить поставленную задачу, надо научиться по одному найденному $x_0 = g^{y_0} \pmod n$ находить все остальные решения.

Для этого вспомним такой факт, что первообразный корень всегда имеет порядок $\phi(n)$ (см. статью о первообразном корне), т.е. наименьшей степенью g, дающей единицу, является $\phi(n)$. Поэтому добавление в показатель степени слагаемого с $\phi(n)$ ничего не меняет:

$$x^k \equiv g^{y_0 \cdot k + l \cdot \phi(n)} \equiv a \pmod n \quad \forall\, l \in \mathcal{Z}$$

Отсюда все решения имеют вид:

$$x = g^{y_0 + \frac{l \cdot \phi(n)}{k}} \pmod n \quad \forall\, l \in \mathcal{Z}$$

где l выбирается таким образом, чтобы дробь $\frac{l \cdot \phi(n)}{k}$ была целой. Чтобы эта дробь была целой, числитель должен быть кратен наименьшему общему кратному $\phi(n)$ и k, откуда (вспоминая, что наименьшее общее кратное двух чисел $\operatorname{lcm}(a, b) = \frac{a \cdot b}{\gcd(a,b)}$), получаем:

$$x = g^{y_0 + i \frac{\phi(n)}{\gcd(k, \phi(n))}} \pmod n \quad \forall\, i \in \mathcal{Z}$$

Это окончательная удобная формула, которая даёт общий вид всех решений задачи дискретного корня.

Реализация

Приведём полную реализацию, включающую нахождение первообразного корня, дискретное логарифмирование и нахождение и вывод всех решений.

```
int gcd (int a, int b) {
        return a ? gcd (b%a, a) : b;
}

int powmod (int a, int b, int p) {
        int res = 1;
        while (b)
                if (b & 1)
                        res = int (res * 1ll * a % p),  --b;
                else
                        a = int (a * 1ll * a % p),  b >>= 1;
        return res;
}

int generator (int p) {
        vector<int> fact;
        int phi = p-1,  n = phi;
        for (int i=2; i*i<=n; ++i)
                if (n % i == 0) {
                        fact.push_back (i);
                        while (n % i == 0)
                                n /= i;
                }
        if (n > 1)
                fact.push_back (n);

        for (int res=2; res<=p; ++res) {
                bool ok = true;
                for (size_t i=0; i<fact.size() && ok; ++i)
                        ok &= powmod (res, phi / fact[i], p) != 1;
                if (ok)  return res;
        }
        return -1;
}

int main() {

        int n, k, a;
        cin >> n >> k >> a;
        if (a == 0) {
                puts ("1\n0");
                return 0;
        }

        int g = generator (n);

        int sq = (int) sqrt (n + .0) + 1;
        vector < pair<int,int> > dec (sq);
        for (int i=1; i<=sq; ++i)
                dec[i-1] = make_pair (powmod (g, int (i * sq * 1ll * k % (n
- 1)), n), i);
        sort (dec.begin(), dec.end());
        int any_ans = -1;
        for (int i=0; i<sq; ++i) {
                int my = int (powmod (g, int (i * 1ll * k % (n - 1)), n) *
1ll * a % n);
                vector < pair<int,int> >::iterator it =
                        lower_bound (dec.begin(), dec.end(), make_pair (my, 0));
                if (it != dec.end() && it->first == my) {
                        any_ans = it->second * sq - i;
                        break;
                }
        }
        if (any_ans == -1) {
                puts ("0");
```

```
                return 0;
        }

        int delta = (n-1) / gcd (k, n-1);
        vector<int> ans;
        for (int cur=any_ans%delta; cur<n-1; cur+=delta)
                ans.push_back (powmod (g, cur, n));
        sort (ans.begin(), ans.end());
        printf ("%d\n", ans.size());
        for (size_t i=0; i<ans.size(); ++i)
                printf ("%d ", ans[i]);

}
```

Решето Эратосфена с линейным временем работы

Дано число n. Требуется найти **все простые** в отрезке $[2; n]$.

Классический способ решения этой задачи — **решето Эратосфена**. Этот алгоритм очень прост, но работает за время $O(n \log \log n)$.

Хотя в настоящий момент известно достаточно много алгоритмов, работающих за сублинейное время (т.е. за $o(n)$), описываемый ниже алгоритм интересен своей **простотой** — он практически не сложнее классического решета Эратосфена.

Кроме того, приводимый здесь алгоритм в качестве "побочного эффекта" фактически вычисляет **факторизацию всех чисел** в отрезке $[2; n]$, что может быть полезно во многих практических применениях.

Недостатком приводимого алгоритма является то, что он использует **больше памяти**, чем классическое решето Эратосфена: требуется заводить массив из n чисел, в то время как классическому решету Эратосфена достаточно лишь n бит памяти (что получается в 32 раза меньше).

Таким образом, описываемый алгоритм имеет смысл применять только до чисел порядка 10^7, не более.

Авторство алгоритма, по всей видимости, принадлежит Грайсу и Мисра (Gries, Misra, 1978 г. — см. список литературы в конце). (И, собственно говоря, называть данный алгоритм "решетом Эратосфена" некорректно: слишком отличаются эти два алгоритма.)

Описание алгоритма

Наша цель — посчитать для каждого числа i от в отрезке $[2; n]$ его **минимальный простой делитель** $lp[i]$.

Кроме того, нам потребуется хранить список всех найденных простых чисел — назовём его массивом $pr[]$.

Изначально все величины $lp[i]$ заполним нулями, что означает, что мы пока предполагаем все числа простыми. В ходе работы алгоритма этот массив будет постепенно заполняться.

Будем теперь перебирать текущее число i от 2 до n. У нас может быть два случая:

- $lp[i] = 0$ — это означает, что число i — простое, т.к. для него так и не обнаружилось других делителей.

  Следовательно, надо присвоить $lp[i] = i$ и добавить i в конец списка $pr[]$.
- $lp[i] \neq 0$ — это означает, что текущее число i — составное, и его минимальным простым делителем является $lp[i]$.

В обоих случаях дальше начинается процесс **расстановки значений** в массиве $lp[]$: мы будем брать числа, **кратные** i, и обновлять у них значение $lp[]$. Однако наша цель — научиться делать это таким образом, чтобы в итоге у каждого числа значение $lp[]$ было бы установлено не более одного раза.

Утверждается, что для этого можно поступить таким образом. Рассмотрим числа вида:

$$x_j = i \cdot p_j,$$

где последовательность p_j — это все простые, не превосходящие $lp[i]$ (как раз для этого нам понадобилось хранить список всех простых чисел).

У всех чисел такого вида проставим новое значение $lp[x_j]$ — очевидно, оно будет равно p_j.

Почему такой алгоритм корректен, и почему он работает за линейное время — см. ниже, пока же приведём его реализацию.

Реализация

Решето выполняется до указанного в константе числа N.

```
const int N = 10000000;
int lp[N+1];
vector<int> pr;

for (int i=2; i<=N; ++i) {
        if (lp[i] == 0) {
                lp[i] = i;
                pr.push_back (i);
```

```
        }
        for (int j=0; j<(int)pr.size() && pr[j]<=lp[i] && i*pr[j]<=N; ++j)
                lp[i * pr[j]] = pr[j];
}
```

Эту реализацию можно немного ускорить, избавившись от вектора pr (заменив его на обычный массив со счётчиком), а также избавившись от дублирующегося умножения во вложенном цикле for (для чего результат произведения надо просто запомнить в какой-либо переменной).

Доказательство корректности

Докажем **корректность** алгоритма, т.е. что он корректно расставляет все значения $lp[]$, причём каждое из них будет установлено ровно один раз. Отсюда будет следовать, что алгоритм работает за линейное время — поскольку все остальные действия алгоритма, очевидно, работают за $O(n)$.

Для этого заметим, что у любого числа i **единственно представление** такого вида:

$$i = lp[i] \cdot x,$$

где $lp[i]$ — (как и раньше) минимальный простой делитель числа i, а число x не имеет делителей, меньших $lp[i]$, т.е.:

$$lp[i] \le lp[x].$$

Теперь сравним это с тем, что делает наш алгоритм — он фактически для каждого x перебирает все простые, на которые его можно домножить, т.е. простые до $lp[x]$ включительно, чтобы получить числа в указанном выше представлении.

Следовательно, алгоритм действительно пройдёт по каждому составному числу ровно один раз, поставив у него правильное значение $lp[]$.

Это означает корректность алгоритма и то, что он работает за линейное время.

Время работы и требуемая память

Хотя асимптотика $O(n)$ лучше асимптотики $O(n \log \log n)$ классического решета Эратосфена, разница между ними невелика. На практике это означает лишь двукратную разницу в скорости, а оптимизированные варианты решета Эратосфена и вовсе не проигрывают приведённому здесь алгоритму.

Учитывая затраты памяти, которые требует этот алгоритм — массив чисел $lp[]$ длины n и массив всех простых $pr[]$ длины примерно $n / \ln n$ — этот алгоритм кажется уступающим классическому решету по всем статьям.

Однако спасает его то, что массив $lp[]$, вычисляемый этим алгоритмом, позволяет искать факторизацию любого числа в отрезке $[2; n]$ за время порядка размера этой факторизации. Более того, ценой ещё одного дополнительного массива можно сделать, чтобы в этой факторизации не требовались операции деления.

Знание факторизации всех чисел — очень полезная информация для некоторых задач, и этот алгоритм является одним из немногих, которые позволяют искать её за линейное время.

Литература

- David Gries, Jayadev Misra. A Linear Sieve Algorithm for Finding Prime Numbers [1978]

тест BPSW на простоту чисел

Введение

Алгоритм BPSW - это тест числа на простоту. Этот алгоритм назван по фамилиям его изобретателей: Роберт Бэйли (Ballie), Карл Померанс (Pomerance), Джон Селфридж (Selfridge), Сэмюэль Вагстафф (Wagstaff). Алгоритм был предложен в 1980 году. На сегодняшний день к алгоритму не было найдено ни одного контрпримера, равно как и не было найдено доказательство.

Алгоритм BPSW был проверен на всех числах до 10^{15}. Кроме того, контрпример пытались найти с помощью программы PRIMO (см. [6]), основанной на тесте на простоту с помощью эллиптических кривых. Программа, проработав три года, не нашла ни одного контрпримера, на основании чего Мартин предположил, что не существует ни одного BPSW-псевдопростого, меньшего 10^{10000} (псевдопростое число - составное число, на котором алгоритм даёт результат "простое"). В то же время, Карл Померанс в 1984 году представил эвристическое доказательство того, что существует бесконечное множество BPSW-псевдопростых чисел.

Сложность алгоритма BPSW есть O ($\log^3(N)$) битовых операций. Если же сравнивать алгоритм BPSW с другими тестами, например, тестом Миллера-Рабина, то алгоритм BPSW обычно оказывается в 3-7 раз медленнее.

Алгоритм нередко применяется на практике. По-видимому, многие коммерческие математические пакеты, полностью или частично, полагаются на алгоритм BPSW для проверки чисел на простоту.

Краткое описание

Алгоритм имеет несколько различных реализаций, отличающихся друг от друга только деталями. В нашем случае алгоритм имеет вид:

1. Выполнить тест Миллера-Рабина по основанию 2.

2. Выполнить сильный тест Лукаса-Селфриджа, используя последовательности Лукаса с параметрами Селфриджа.

3. Вернуть "простое" только в том случае, когда оба теста вернули "простое".

+0. Кроме того, в начало алгоритма можно добавить проверку на тривиальные делители, скажем, до 1000. Это позволит увеличить скорость работы на составных числах, правда, несколько замедлив алгоритм на простых.

Итак, алгоритм BPSW основывается на следующем:

1. (факт) тест Миллера-Рабина и тест Лукаса-Селфриджа если и ошибаются, то только в одну сторону: некоторые составные числа этими алгоритмами опознаются как простые. В обратную сторону эти алгоритмы не ошибаются никогда.

2. (предположение) тест Миллера-Рабина и тест Лукаса-Селфриджа если и ошибаются, то никогда не ошибаются на одном числе одновременно.

На самом деле, второе предположение вроде бы как и неверно - эвристическое доказательство-опровержение Померанса приведено ниже. Тем не менее, на практике ни одного псевдопростого до сих пор не нашли, поэтому условно можно считать второе предположение верным.

Реализация алгоритмов в данной статье

Все алгоритмы в данной статье будут реализованы на C++. Все программы тестировались только на компиляторе Microsoft C++ 8.0 SP1 (2005), также должны компилироваться на g++.

Алгоритмы реализованы с использованием шаблонов (templates), что позволяет применять их как к встроенным числовым типам, так и собственным классам, реализующим длинную арифметику. [пока длинная арифметика в статью не входит - TODO]

В самой статье будут приведены только самые существенные функции, тексты же вспомогательных функций можно скачать в приложении к статье. Здесь будут приведены только заголовки этих функций вместе с комментариями:

```
//! Модуль 64-битного числа
long long abs (long long n);
```

```
unsigned long long abs (unsigned long long n);

//! Возвращает true, если n четное
template <class T>
bool even (const T & n);

//! Делит число на 2
template <class T>
void bisect (T & n);

//! Умножает число на 2
template <class T>
void redouble (T & n);

//! Возвращает true, если n - точный квадрат простого числа
template <class T>
bool perfect_square (const T & n);

//! Вычисляет корень из числа, округляя его вниз
template <class T>
T sq_root (const T & n);

//! Возвращает количество бит в числе
template <class T>
unsigned bits_in_number (T n);

//! Возвращает значение k-го бита числа (биты нумеруются с нуля)
template <class T>
bool test_bit (const T & n, unsigned k);

//! Умножает a *= b (mod n)
template <class T>
void mulmod (T & a, T b, const T & n);

//! Вычисляет a^k (mod n)
template <class T, class T2>
T powmod (T a, T2 k, const T & n);

//! Переводит число n в форму q*2^p
template <class T>
void transform_num (T n, T & p, T & q);

//! Алгоритм Евклида
template <class T, class T2>
T gcd (const T & a, const T2 & b);

//! Вычисляет jacobi(a,b) - символ Якоби
template <class T>
T jacobi (T a, T b)

//! Вычисляет pi(b) первых простых чисел. Возвращает вектор с простыми и в pi
- pi(b)
template <class T, class T2>
const std::vector & get_primes (const T & b, T2 & pi);

//! Тривиальная проверка n на простоту, перебираются все делители до m.
//! Результат: 1 - если n точно простое, p - его найденный делитель, 0 -
если неизвестно
template <class T, class T2>
T2 prime_div_trivial (const T & n, T2 m);
```

Тест Миллера-Рабина

Я не буду заострять внимание на тесте Миллера-Рабина, поскольку он описывается во многих источниках, в том числе и на русском языке (например. см. [5]).

Замечу лишь, что скорость его работы есть O ($\log^3(N)$) битовых операций и приведу готовую реализацию этого алгоритма:

```
template <class T, class T2>
bool miller_rabin (T n, T2 b)
{

        // сначала проверяем тривиальные случаи
        if (n == 2)
                return true;
        if (n < 2 || even (n))
                return false;

        // проверяем, что n и b взаимно просты (иначе это приведет к ошибке)
        // если они не взаимно просты, то либо n не просто, либо
нужно увеличить b
        if (b < 2)
                b = 2;
        for (T g; (g = gcd (n, b)) != 1; ++b)
                if (n > g)
                        return false;

        // разлагаем n-1 = q*2^p
        T n_1 = n;
        --n_1;
        T p, q;
        transform_num (n_1, p, q);

        // вычисляем b^q mod n, если оно равно 1 или n-1, то n простое
(или псевдопростое)
        T rem = powmod (T(b), q, n);
        if (rem == 1 || rem == n_1)
                return true;

        // теперь вычисляем b^2q, b^4q, ... , b^((n-1)/2)
        // если какое-либо из них равно n-1, то n простое (или псевдопростое)
        for (T i=1; i<p; i++)
        {
                mulmod (rem, rem, n);
                if (rem == n_1)
                        return true;
        }

        return false;

}
```

Сильный тест Лукаса-Селфриджа

Сильный тест Лукаса-Селфриджа состоит из двух частей: алгоритма Селфриджа для вычисления некоторого параметра, и сильного алгоритма Лукаса, выполняемого с этим параметром.

Алгоритм Селфриджа

Среди последовательности 5, -7, 9, -11, 13, ... найти первое число D, для которого J (D, N) = -1 и gcd (D, N) = 1, где J(x,y) - символ Якоби.

Параметрами Селфриджа будут P = 1 и Q = (1 - D) / 4.

Следует заметить, что параметр Селфриджа не существует для чисел, которые являются точными квадратами. Действительно, если число является точным квадратом, то перебор D дойдёт до sqrt(N), на котором окажется, что gcd (D, N) > 1, т.е. обнаружится, что число N составное.

Кроме того, параметры Селфриджа будут вычислены неправильно для чётных чисел и для единицы; впрочем, проверка этих случаев не составит труда.

Таким образом, **перед началом алгоритма** следует проверить, что число N является нечётным, большим 2, и не является точным квадратом, иначе (при невыполнении хотя бы одного условия) нужно сразу выйти из алгоритма с результатом "составное".

Наконец, заметим, что если D для некоторого числа N окажется слишком большим, то алгоритм с вычислительной точки зрения окажется неприменимым. Хотя на практике такого замечено не было (оказывалось вполне достаточно 4-байтного числа), тем не менее вероятность этого события не следует исключать. Впрочем, например, на отрезке [1;

10^6] max(D) = 47, а на отрезке [10^{19}; $10^{19}+10^6$] max(D) = 67. Кроме того, Бэйли и Вагстаф в 1980 году аналитически доказали это наблюдение (см. Ribenboim, 1995/96, стр. 142).

Сильный алгоритм Лукаса

Параметрами алгоритма Лукаса являются числа D, P **и** Q такие, что D = P^2 - 4*Q ? 0, и P > 0.

(нетрудно заметить, что параметры, вычисленные по алгоритму Селфриджа, удовлетворяют этим условиям)

Последовательности Лукаса - это последовательности U_k и V_k, определяемые следующим образом:

$$U_0 = 0$$
$$U_1 = 1$$
$$\mathbf{U_k = P\ U_{k-1} - Q\ U_{k-2}}$$
$$V_0 = 2$$
$$V_1 = P$$
$$\mathbf{V_k = P\ V_{k-1} - Q\ V_{k-2}}$$

Далее, пусть M = N - J (D, N).

Если N простое, и gcd (N, Q) = 1, то имеем:

$$\mathbf{U_M = 0\ (mod\ N)}$$

В частности, когда параметры D, P, Q вычислены алгоритмом Селфриджа, имеем:

$$U_{N+1} = 0\ (mod\ N)$$

Обратное, вообще говоря, неверно. Тем не менее, псевдопростых чисел при данном алгоритме оказывается не очень много, на чём, собственно, и основывается алгоритм Лукаса.

Итак, **алгоритм Лукаса заключается в вычислении** U_M **и сравнении его с нулём**.

Далее, необходимо найти какой-то способ ускорения вычисления U_K, иначе, понятно, никакого практического смысла в этом алгоритма не было бы.

Имеем:

$$U_k = (a^k - b^k) / (a - b),$$
$$V_k = a^k + b^k,$$

где a и b - различные корни квадратного уравнения x^2 - P x + Q = 0.

Теперь следующие равенства можно доказать элементарно:

$$U_{2k} = U_k\ V_k\ (mod\ N)$$
$$V_{2k} = V_k^2 - 2\ Q^k\ (mod\ N)$$

Теперь, если представить M = E 2^T, где E - нечётное число, то легко получить:

$$\mathbf{U_M = U_E\ V_E\ V_{2E}\ V_{4E}\ ...\ V_{2^{T-2}E}\ V_{2^{T-1}E} = 0\ (mod\ N),}$$

и хотя бы один из множителей равен нулю по модулю N.

Понятно, что **достаточно вычислить** U_E **и** V_E, а все последующие множители V_{2E} V_{4E} ... $V_{2^{T-2}E}$ $V_{2^{T-1}E}$ можно **получить уже из них**.

Таким образом, осталось научиться быстро вычислять U_E и V_E для нечётного E.

Сначала рассмотрим следующие формулы для сложения членов последовательностей Лукаса:

$$U_{i+j} = (U_i\ V_j + U_j\ V_i) / 2\ (mod\ N)$$
$$V_{i+j} = (V_i\ V_j + D\ U_i\ U_j) / 2\ (mod\ N)$$

Следует обратить внимание, что деление выполняется в поле (mod N).

Формулы эти доказываются очень просто, и здесь их доказательство опущено.

Теперь, обладая формулами для сложения и для удвоения членов последовательностей Лукаса, понятен и способ ускорения вычисления U_E и V_E.

Действительно, рассмотрим двоичную запись числа E. Положим сначала результат - U_E и V_E - равными, соответственно, U_1 и V_1. Пройдёмся по всем битам числа E от более младших к более старшим, пропустив только самый первый бит (начальный член последовательности). Для каждого i-го бита будем вычислять U_{2i} и V_{2i} из предыдущих членов с помощью формул удвоения. Кроме того, если текущий i-ый бит равен единице, то к ответу будем прибавлять текущие U_{2i} и V_{2i} с помощью формул сложения. По окончании алгоритма, выполняющегося за O (log(E)), мы **получим искомые** U_E **и** V_E.

Если U_E или V_E оказались равными нулю (mod N), то число N простое (или псевдопростое). Если они оба отличны от нуля, то вычисляем V_{2E}, V_{4E}, ... $V_{2^{T-2}E}$, $V_{2^{T-1}E}$. Если хотя бы один из них сравним с нулём по модулю N, то число N простое (или псевдопростое). Иначе число N составное.

Обсуждение алгоритма Селфриджа

Теперь, когда мы рассмотрели алгоритм Лукаса, можно более подробно остановиться на его параметрах D,P,Q, одним из способов получения которых и является алгоритм Селфриджа.

Напомним базовые требования к параметрам:

```
P > 0,
D = P² - 4*Q ? 0.
```

Теперь продолжим изучение этих параметров.

D **не должно быть точным квадратом** (mod N).

Действительно, иначе получим:

$D = b^2$, отсюда J(D,N) = 1, P = b + 2, Q = b + 1, отсюда $U_{n-1} = (Q^{n-1} - 1) / (Q - 1)$.

Т.е. если D - точный квадрат, то алгоритм Лукаса становится практически обычным вероятностным тестом.

Один из лучших способов избежать подобного - **потребовать**, **чтобы** J(D,N) = -1.

Например, можно выбрать первое число D из последовательности 5, -7, 9, -11, 13, ..., для которого J(D,N) = -1. Также пусть P = 1. Тогда Q = (1 - D) / 4. Этот способ был предложен Селфриджем.

Впрочем, имеются и другие способы выбора D. Можно выбирать его из последовательности 5, 9, 13, 17, 21, ... Также пусть P - наименьшее нечётное, привосходящее sqrt(D). Тогда $Q = (P^2 - D) / 4$.

Понятно, что от выбора конкретного способа вычисления параметров Лукаса зависит и его результат - псевдопростые могут отличаться при различных способах выбора параметра. Как показала практика, алгоритм, предложенный Селфриджем, оказался очень удачным: все псевдопростые Лукаса-Селфриджа не являются псевдопростыми Миллера-Рабина, по крайней мере, ни одного контрпримера найдено не было.

Реализация сильного алгоритма Лукаса-Селфриджа

Теперь осталось только реализовать алгоритм:

```
template <class T, class T2>
bool lucas_selfridge (const T & n, T2 unused)
{

        // сначала проверяем тривиальные случаи
        if (n == 2)
                return true;
        if (n < 2 || even (n))
                return false;

        // проверяем, что n не является точным квадратом, иначе алгоритм
даст ошибку
        if (perfect_square (n))
                return false;

        // алгоритм Селфриджа: находим первое число d такое, что:
```

```
	// jacobi(d,n)=-1 и оно принадлежит ряду { 5,-7,9,-11,13,... }
	T2 dd;
	for (T2 d_abs = 5, d_sign = 1; ; d_sign = -d_sign, ++++d_abs)
	{
		dd = d_abs * d_sign;
		T g = gcd (n, d_abs);
		if (1 < g && g < n)
			// нашли делитель - d_abs
			return false;
		if (jacobi (T(dd), n) == -1)
			break;
	}

	// параметры Селфриджа
	T2
		p = 1,
		q = (p*p - dd) / 4;

	// разлагаем n+1 = d*2^s
	T n_1 = n;
	++n_1;
	T s, d;
	transform_num (n_1, s, d);

	// алгоритм Лукаса
	T
		u = 1,
		v = p,
		u2m = 1,
		v2m = p,
		qm = q,
		qm2 = q*2,
		qkd = q;
	for (unsigned bit = 1, bits = bits_in_number(d); bit < bits; bit++)
	{
		mulmod (u2m, v2m, n);
		mulmod (v2m, v2m, n);
		while (v2m < qm2)
			v2m += n;
		v2m -= qm2;
		mulmod (qm, qm, n);
		qm2 = qm;
		redouble (qm2);
		if (test_bit (d, bit))
		{
			T t1, t2;
			t1 = u2m;
			mulmod (t1, v, n);
			t2 = v2m;
			mulmod (t2, u, n);

			T t3, t4;
			t3 = v2m;
			mulmod (t3, v, n);
			t4 = u2m;
			mulmod (t4, u, n);
			mulmod (t4, (T)dd, n);

			u = t1 + t2;
			if (!even (u))
				u += n;
			bisect (u);
			u %= n;

			v = t3 + t4;
			if (!even (v))
				v += n;
			bisect (v);
			v %= n;
			mulmod (qkd, qm, n);
		}
```

```
        }

        // точно простое (или псевдо-простое)
        if (u == 0 || v == 0)
                return true;

        // довычисляем оставшиеся члены
        T qkd2 = qkd;
        redouble (qkd2);
        for (T2 r = 1; r < s; ++r)
        {
                mulmod (v, v, n);
                v -= qkd2;
                if (v < 0) v += n;
                if (v < 0) v += n;
                if (v >= n) v -= n;
                if (v >= n) v -= n;
                if (v == 0)
                        return true;
                if (r < s-1)
                {
                        mulmod (qkd, qkd, n);
                        qkd2 = qkd;
                        redouble (qkd2);
                }
        }

        return false;

}
```

Код BPSW

Теперь осталось просто скомбинировать результаты всех 3 тестов: проверка на небольшие тривиальные делители, тест Миллера-Рабина, сильный тест Лукаса-Селфриджа.

```
template <class T>
bool baillie_pomerance_selfridge_wagstaff (T n)
{

        // сначала проверяем на тривиальные делители - например, до 29
        int div = prime_div_trivial (n, 29);
        if (div == 1)
                return true;
        if (div > 1)
                return false;

        // тест Миллера-Рабина по основанию 2
        if (!miller_rabin (n, 2))
                return false;

        // сильный тест Лукаса-Селфриджа
        return lucas_selfridge (n, 0);

}
```

Отсюда можно скачать программу (исходник + exe), содержащую полную реализацию теста BPSW. [77 КБ]

Краткая реализация

Длину кода можно значительно уменьшить в ущерб универсальности, отказавшись от шаблонов и различных вспомогательных функций.

```
const int trivial_limit = 50;
```

```
int p[1000];

int gcd (int a, int b) {
	return a ? gcd (b%a, a) : b;
}

int powmod (int a, int b, int m) {
	int res = 1;
	while (b)
		if (b & 1)
			res = (res * 1ll * a) % m,  --b;
		else
			a = (a * 1ll * a) % m,  b >>= 1;
	return res;
}

bool miller_rabin (int n) {
	int b = 2;
	for (int g; (g = gcd (n, b)) != 1; ++b)
		if (n > g)
			return false;
	int p=0, q=n-1;
	while ((q & 1) == 0)
		++p,  q >>= 1;
	int rem = powmod (b, q, n);
	if (rem == 1 || rem == n-1)
		return true;
	for (int i=1; i<p; ++i) {
		rem = (rem * 1ll * rem) % n;
		if (rem == n-1)  return true;
	}
	return false;
}

int jacobi (int a, int b)
{
	if (a == 0)  return 0;
	if (a == 1)  return 1;
	if (a < 0)
		if ((b & 2) == 0)
			return jacobi (-a, b);
		else
			return - jacobi (-a, b);
	int a1=a,  e=0;
	while ((a1 & 1) == 0)
		a1 >>= 1,  ++e;
	int s;
	if ((e & 1) == 0 || (b & 7) == 1 || (b & 7) == 7)
		s = 1;
	else
		s = -1;
	if ((b & 3) == 3 && (a1 & 3) == 3)
		s = -s;
	if (a1 == 1)
		return s;
	return s * jacobi (b % a1, a1);
}

bool bpsw (int n) {
	if ((int)sqrt(n+0.0) * (int)sqrt(n+0.0) == n)  return false;
	int dd=5;
	for (;;) {
		int g = gcd (n, abs(dd));
		if (1<g && g<n)  return false;
		if (jacobi (dd, n) == -1)  break;
		dd = dd<0 ? -dd+2 : -dd-2;
	}
	int p=1,  q=(p*p-dd)/4;
	int d=n+1,  s=0;
	while ((d & 1) == 0)
		++s,  d>>=1;
```

```
        long long u=1, v=p, u2m=1, v2m=p, qm=q, qm2=q*2, qkd=q;
        for (int mask=2; mask<=d; mask<<=1) {
                u2m = (u2m * v2m) % n;
                v2m = (v2m * v2m) % n;
                while (v2m < qm2)   v2m += n;
                v2m -= qm2;
                qm = (qm * qm) % n;
                qm2 = qm * 2;
                if (d & mask) {
                        long long t1 = (u2m * v) % n,  t2 = (v2m * u) % n,
                                t3 = (v2m * v) % n,  t4 = (((u2m * u) % n)
* dd) % n;
                        u = t1 + t2;
                        if (u & 1)  u += n;
                        u = (u >> 1) % n;
                        v = t3 + t4;
                        if (v & 1)  v += n;
                        v = (v >> 1) % n;
                        qkd = (qkd * qm) % n;
                }
        }
        if (u==0 || v==0)  return true;
        long long qkd2 = qkd*2;
        for (int r=1; r<s; ++r) {
                v = (v * v) % n - qkd2;
                if (v < 0)  v += n;
                if (v < 0)  v += n;
                if (v >= n)  v -= n;
                if (v >= n)  v -= n;
                if (v == 0)  return true;
                if (r < s-1) {
                        qkd = (qkd * 1ll * qkd) % n;
                        qkd2 = qkd * 2;
                }
        }
        return false;
}

bool prime (int n) { // эту функцию нужно вызывать для проверки на простоту
        for (int i=0; i<trivial_limit && p[i]<n; ++i)
                if (n % p[i] == 0)
                        return false;
        if (p[trivial_limit-1]*p[trivial_limit-1] >= n)
                return true;
        if (!miller_rabin (n))
                return false;
        return bpsw (n);
}

void prime_init() { // вызвать до первого вызова prime() !
        for (int i=2, j=0; j<trivial_limit; ++i) {
                bool pr = true;
                for (int k=2; k*k<=i; ++k)
                        if (i % k == 0)
                                pr = false;
                if (pr)
                        p[j++] = i;
        }
}
```

Эвристическое доказательство-опровержение Померанса

Померанс в 1984 году предложил следующее эвристическое доказательство.

Утверждение: **Количество** BPSW-**псевдопростых от** 1 **до** X **больше** X^{1-a} **для любого** $a > 0$.

Доказательство.

Пусть k > 4 - произвольное, но фиксированное число. Пусть T - некоторое большое число.

Пусть $P_k(T)$ - множество таких простых p в интервале $[T; T^k]$, для которых:

(1) p = 3 (mod 8), J(5,p) = -1

(2) число (p-1)/2 не является точным квадратом

(3) число (p-1)/2 составлено исключительно из простых q < T

(4) число (p-1)/2 составлено исключительно из таких простых q, что q = 1 (mod 4)

(5) число (p+1)/4 не является точным квадратом

(6) число (p+1)/4 составлено исключительно из простых d < T

(7) число (p+1)/4 составлено исключительно из таких простых d, что q = 3 (mod 4)

Понятно, что приблизительно 1/8 всех простых в отрезке $[T; T^k]$ удовлетворяет условию (1). Также можно показать, что условия (2) и (5) сохраняют некоторую часть чисел. Эвристически, условия (3) и (6) также позволяют нам оставить некоторую часть чисел из отрезка $(T; T^k)$. Наконец, событие (4) обладает вероятностью $(c (\log T)^{-1/2})$, так же как и событие (7). Таким образом, мощность множества $P_k(T)$ прблизительно равна при T -> oo

$$\frac{cT^k}{\log^2 T}$$

где c - некоторая положительная константа, зависящая от выбора k.

Теперь мы **можем построить число** n, не являющееся точным квадратом, составленное из l простых из P_k (T), где l нечётно и меньше $T^2 / \log(T^k)$. Количество способов выбрать такое число n есть примерно

$$\binom{[cT^k/\log^2 T]}{\ell} > e^{T^2(1-3/k)}$$

для большого T и фиксированного k. Кроме того, каждое такое число n меньше e^{T^2}.

Обозначим через Q_1 произведение простых q < T, для которых q = 1 (mod 4), а через Q_3 - произведение простых q < T, для которых q = 3 (mod 4). Тогда gcd (Q_1, Q_3) = 1 и $Q_1 Q_3$? e^T. Таким образом, количество способов выбрать n **с дополнительными условиями**

```
n = 1 (mod Q1), n = -1 (mod Q3)
```

должно быть, эвристически, как минимум

$$e^{T^2 (1 - 3 / k)} / e^{2T} > e^{T^2 (1 - 4 / k)}$$

для большого T.

Но **каждое такое** n - **это контрпример к тесту** BPSW. Действительно, n будет числом Кармайкла (т. е. числом, на котором тест Миллера-Рабина будет ошибаться при любом основании), поэтому оно автоматически будет псевдопростым по основанию 2. Поскольку n = 3 (mod 8) и каждое p | n равно 3 (mod 8), очевидно, что n также будет сильным псевдопростым по основанию 2. Поскольку J(5,n) = -1, то каждое простое p | n удовлетворяет J(5,p) = -1, и так как p+1 | n+1 для любого простого p | n, отсюда следует, что n - псевдопростое Лукаса для любого теста Лукаса с дискриминантом 5.

Таким образом, мы показали, что для любого фиксированного k и всех больших T, будет как минимум $e^{T^2 (1 - 4 / k)}$ контрпримеров к тесту BPSW среди чисел, меньших e^{T^2}. Теперь, если мы положим $x = e^{T^2}$, будет как минимум $x^{1 - 4 / k}$ контрпримеров, меньших x. Поскольку k - случайное число, то наше доказательство означает, что **количество контрпримеров**, **меньших** x, **есть число**, **большее** x^{1-a} **для любого** a > 0.

Приложение. Все программы

Скачать все программы из данной статьи. [242 КБ]

Литература

Использованная мной литература, полностью доступная в Интернете:

1. Robert Baillie; Samuel S. Wagstaff

Lucas pseudoprimes
Math. Comp. 35 (1980) 1391-1417
mpqs.free.fr/LucasPseudoprimes.pdf

2. Daniel J. Bernstein
Distinguishing prime numbers from composite numbers: the state of the art in 2004
Math. Comp. (2004)
cr.yp.to/primetests/prime2004-20041223.pdf

3. Richard P. Brent
Primality Testing and Integer Factorisation
The Role of Mathematics in Science (1990)
wwwmaths.anu.edu.au/~brent/pd/rpb120.pdf

4. H. Cohen; H. W. Lenstra
Primality Testing and Jacobi Sums
Amsterdam (1984)
www.openaccess.leidenuniv.nl/bitstream/1887/2136/1/346_065.pdf

5. Thomas H. Cormen; Charles E. Leiserson; Ronald L. Rivest
Introduction to Algorithms
[без ссылки]
The MIT Press (2001)

6. M. Martin
PRIMO - Primality Proving
www.ellipsa.net

7. F. Morain
Elliptic curves and primality proving
Math. Comp. 61(203) (1993)
citeseer.ist.psu.edu/rd/43190198%2C72628%2C1%2C0.25%2CDownload/ftp%3AqSqqSqftp.
inria.frqSqINRIAqSqpublicationqSqpubli-ps-gzqSqRRqSqRR-1256.ps.gz

8. Carl Pomerance
Are there counter-examples to the Baillie-PSW primality test?
Math. Comp. (1984)
www.pseudoprime.com/dopo.pdf

9. Eric W. Weisstein
Baillie-PSW primality test
MathWorld (2005)
mathworld.wolfram.com/Baillie-PSWPrimalityTest.html

10. Eric W. Weisstein
Strong Lucas pseudoprime
MathWorld (2005)
mathworld.wolfram.com/StrongLucasPseudoprime.html

11. Paulo Ribenboim
The Book of Prime Number Records
Springer-Verlag (1989)
[без ссылки]

Список других рекомендуемых книг, которых мне не удалось найти в Интернете:

12. Zhaiyu Mo; James P. Jones
A new primality test using Lucas sequences
Preprint (1997)

13. Hans Riesel
Prime numbers and computer methods for factorization
Boston: Birkhauser (1994)

Эффективные алгоритмы факторизации

Здесь приведены реализации нескольких алгоритмов факторизации, каждый из которых по отдельности может работать как быстро, так и очень медленно, но в сумме они дают весьма быстрый метод.

Описания этих методов не приводятся, тем более что они достаточно хорошо описаны в Интернете.

Метод Полларда p-1

Вероятностный тест, быстро даёт ответ далеко не для всех чисел.

Возвращает либо найденный делитель, либо 1, если делитель не был найден.

```
template <class T>
T pollard_p_1 (T n)
{
        // параметры алгоритма, существенно влияют на производительность
и качество поиска
        const T b = 13;
        const T q[] = { 2, 3, 5, 7, 11, 13 };

        // несколько попыток алгоритма
        T a = 5 % n;
        for (int j=0; j<10; j++)
        {

                // ищем такое a, которое взаимно просто с n
                while (gcd (a, n) != 1)
                {
                        mulmod (a, a, n);
                        a += 3;
                        a %= n;
                }

                // вычисляем a^M
                for (size_t i = 0; i < sizeof q / sizeof q[0]; i++)
                {
                        T qq = q[i];
                        T e = (T) floor (log ((double)b) / log ((double)qq));
                        T aa = powmod (a, powmod (qq, e, n), n);
                        if (aa == 0)
                                continue;

                        // проверяем, не найден ли ответ
                        T g = gcd (aa-1, n);
                        if (1 < g && g < n)
                                return g;
                }

        }

        // если ничего не нашли
        return 1;

}
```

Метод Полларда "Po"

Вероятностный тест, быстро даёт ответ далеко не для всех чисел.

Возвращает либо найденный делитель, либо 1, если делитель не был найден.

```
template <class T>
T pollard_rho (T n, unsigned iterations_count = 100000)
```

```
{
        T
                b0 = rand() % n,
                b1 = b0,
                g;
        mulmod (b1, b1, n);
        if (++b1 == n)
                b1 = 0;
        g = gcd (abs (b1 - b0), n);
        for (unsigned count=0; count<iterations_count && (g == 1 || g ==
n); count++)
        {
                mulmod (b0, b0, n);
                if (++b0 == n)
                        b0 = 0;
                mulmod (b1, b1, n);
                ++b1;
                mulmod (b1, b1, n);
                if (++b1 == n)
                        b1 = 0;
                g = gcd (abs (b1 - b0), n);
        }
        return g;
}
```

Метод Бента (модификация метода Полларда "Ро")

Вероятностный тест, быстро даёт ответ далеко не для всех чисел.

Возвращает либо найденный делитель, либо 1, если делитель не был найден.

```
template <class T>
T pollard_bent (T n, unsigned iterations_count = 19)
{
        T
                b0 = rand() % n,
                b1 = (b0*b0 + 2) % n,
                a = b1;
        for (unsigned iteration=0, series_len=1;
iteration<iterations_count; iteration++, series_len*=2)
        {
                T g = gcd (b1-b0, n);
                for (unsigned len=0; len<series_len && (g==1 && g==n); len++)
                {
                        b1 = (b1*b1 + 2) % n;
                        g = gcd (abs(b1-b0), n);
                }
                b0 = a;
                a = b1;
                if (g != 1 && g != n)
                        return g;
        }
        return 1;
}
```

Метод Полларда Монте-Карло

Вероятностный тест, быстро даёт ответ далеко не для всех чисел.

Возвращает либо найденный делитель, либо 1, если делитель не был найден.

```
template <class T>
T pollard_monte_carlo (T n, unsigned m = 100)
{
        T b = rand() % (m-2) + 2;

        static std::vector<T> primes;
        static T m_max;
        if (primes.empty())
```

```
                primes.push_back (3);
        if (m_max < m)
        {
                m_max = m;
                for (T prime=5; prime<=m; ++++prime)
                {
                        bool is_prime = true;
                        for (std::vector<T>::const_iterator iter=primes.
begin(), end=primes.end();
                                iter!=end; ++iter)
                        {
                                T div = *iter;
                                if (div*div > prime)
                                        break;
                                if (prime % div == 0)
                                {
                                        is_prime = false;
                                        break;
                                }
                        }
                        if (is_prime)
                                primes.push_back (prime);
                }
        }

        T g = 1;
        for (size_t i=0; i<primes.size() && g==1; i++)
        {
                T cur = primes[i];
                while (cur <= n)
                        cur *= primes[i];
                cur /= primes[i];
                b = powmod (b, cur, n);
                g = gcd (abs(b-1), n);
                if (g == n)
                        g = 1;
        }

        return g;
}
```

Метод Ферма

Это стопроцентный метод, но он может работать очень медленно, если у числа есть маленькие делители.

Поэтому запускать его стоит только после всех остальных методов.

```
template <class T, class T2>
T ferma (const T & n, T2 unused)
{
        T2
                x = sq_root (n),
                y = 0,
                r = x*x - y*y - n;
        for (;;)
                if (r == 0)
                        return x!=y ? x-y : x+y;
                else
                        if (r > 0)
                        {
                                r -= y+y+1;
                                ++y;
                        }
                        else
                        {
                                r += x+x+1;
                                ++x;
                        }
}
```

Тривиальное деление

Этот элементарный метод пригодится, чтобы сразу обрабатывать числа с очень маленькими делителями.

```
template <class T, class T2>
T2 prime_div_trivial (const T & n, T2 m)
{

        // сначала проверяем тривиальные случаи
        if (n == 2 || n == 3)
                return 1;
        if (n < 2)
                return 0;
        if (even (n))
                return 2;

        // генерируем простые от 3 до m
        T2 pi;
        const vector<T2> & primes = get_primes (m, pi);

        // делим на все простые
        for (std::vector<T2>::const_iterator iter=primes.begin(), end=primes.
end();
                iter!=end; ++iter)
        {
                const T2 & div = *iter;
                if (div * div > n)
                        break;
                else
                        if (n % div == 0)
                                return div;
        }

        if (n < m*m)
                return 1;
        return 0;

}
```

Собираем всё вместе

Объединяем все методы в одной функции.

Также функция использует тест на простоту, иначе алгоритмы факторизации могут работать очень долго.
Например, можно выбрать тест BPSW (читать статью по BPSW).

```
template <class T, class T2>
void factorize (const T & n, std::map<T,unsigned> & result, T2 unused)
{
        if (n == 1)
                ;
        else
                // проверяем, не простое ли число
                if (isprime (n))
                        ++result[n];
                else
                        // если число достаточно маленькое, то его
разлагаем простым перебором
                        if (n < 1000*1000)
                        {
                                T div = prime_div_trivial (n, 1000);
                                ++result[div];
                                factorize (n / div, result, unused);
                        }
                        else
                        {
                                // число большое, запускаем на нем
алгоритмы факторизации
                                T div;
```

```
                                // сначала идут быстрые алгоритмы Полларда
                                div = pollard_monte_carlo (n);
                                if (div == 1)
                                        div = pollard_rho (n);
                                if (div == 1)
                                        div = pollard_p_1 (n);
                                if (div == 1)
                                        div = pollard_bent (n);
                                // придётся запускать 100%-ый алгоритм Ферма
                                if (div == 1)
                                        div = ferma (n, unused);
                                // рекурсивно обрабатываем найденные множители
                                factorize (div, result, unused);
                                factorize (n / div, result, unused);
                        }
}
```

Приложение

Скачать [5 КБ] исходник программы, которая использует все указанные методы факторизации и тест BPSW на простоту.

Быстрое преобразование Фурье за O (N log N). Применение к умножению двух полиномов или длинных чисел

Здесь мы рассмотрим алгоритм, который позволяет перемножить два полинома длиной n за время $O(n \log n)$, что значительно лучше времени $O(n^2)$, достигаемого тривиальным алгоритмом умножения. Очевидно, что умножение двух длинных чисел можно свести к умножению полиномов, поэтому два длинных числа также можно перемножить за время $O(n \log n)$.

Изобретение Быстрого преобразования Фурье приписывается Кули (Coolet) и Таки (Tukey) — 1965 г. На самом деле БПФ неоднократно изобреталось до этого, но важность его в полной мере не осознавалась до появления современных компьютеров. Некоторые исследователи приписывают открытие БПФ Рунге (Runge) и Кёнигу (Konig) в 1924 г. Наконец, открытие этого метода приписывается ещё Гауссу (Gauss) в 1805 г.

Дискретное преобразование Фурье (ДПФ)

Пусть имеется многочлен n-ой степени:

$$A(x) = a_0 x^0 + a_1 x^1 + \ldots + a_{n-1} x^{n-1}.$$

Не теряя общности, можно считать, что n является степенью 2. Если в действительности n не является степенью 2, то мы просто добавим недостающие коэффициенты, положив их равными нулю.

Из теории функций комплексного переменного известно, что комплексных корней n-ой степени из единицы существует ровно n. Обозначим эти корни через $w_{n,k}, k = 0 \ldots n-1$, тогда известно, что $w_{n,k} = e^{i\frac{2\pi k}{n}}$. Кроме того, один из этих корней $w_n = w_{n,1} = e^{i\frac{2\pi}{n}}$ (называемый главным значением корня n-ой степени из единицы) таков, что все остальные корни являются его степенями: $w_{n,k} = (w_n)^k$.

Тогда **дискретным преобразованием Фурье** (**ДПФ**) (discrete Fourier transform, DFT) многочлена $A(x)$ (или, что то же самое, ДПФ вектора его коэффициентов $(a_0, a_1, \ldots, a_{n-1})$) называются значения этого многочлена в точках $x = w_{n,k}$, т.е. это вектор:

$$\begin{aligned}\mathrm{DFT}(a_0, a_1, \ldots, a_{n-1}) &= (y_0, y_1, \ldots, y_{n-1}) = (A(w_{n,0}), A(w_{n,1}), \ldots, A(w_{n,n-1})) = \\ &= (A(w_n^0), A(w_n^1), \ldots, A(w_n^{n-1})).\end{aligned}$$

Аналогично определяется и **обратное дискретное преобразование Фурье** (InverseDFT). Обратное ДПФ для вектора значений многочлена $(y_0, y_1, \ldots y_{n-1})$ — это вектор коэффициентов многочлена $(a_0, a_1, \ldots, a_{n-1})$:

$$\mathrm{InverseDFT}(y_0, y_1, \ldots, y_{n-1}) = (a_0, a_1, \ldots, a_{n-1}).$$

Таким образом, если прямое ДПФ переходит от коэффициентов многочлена к его значениям в комплексных корнях n-ой степени из единицы, то обратное ДПФ — наоборот, по значениям многочлена восстанавливает коэффициенты многочлена.

Применение ДПФ для быстрого умножения полиномов

Пусть даны два многочлена A и B. Посчитаем ДПФ для каждого из них: $\mathrm{DFT}(A)$ и $\mathrm{DFT}(B)$ — это два вектора-значения многочленов.

Теперь, что происходит при умножении многочленов? Очевидно, в каждой точке их значения просто перемножаются, т.е.

$$(A \times B)(x) = A(x) \times B(x).$$

Но это означает, что если мы перемножим вектора $\mathrm{DFT}(A)$ и $\mathrm{DFT}(B)$, просто умножив каждый элемент одного вектора на соответствующий ему элемент другого вектора, то мы получим не что иное, как ДПФ от многочлена $A \times B$:

$$\mathrm{DFT}(A \times B) = \mathrm{DFT}(A) \times \mathrm{DFT}(B).$$

Наконец, применяя обратное ДПФ, получаем:

$$A \times B = \mathrm{InverseDFT}(\mathrm{DFT}(A) \times \mathrm{DFT}(B)),$$

где, повторимся, справа под произведением двух ДПФ понимается попарные произведения элементов векторов. Такое произведение, очевидно, требует для вычисления только $O(n)$ операций. Таким образом, если мы научимся вычислять ДПФ и обратное ДПФ за время $O(n \log n)$, то и произведение двух полиномов (а, следовательно, и двух длинных чисел) мы сможем найти за ту же асимптотику.

Следует заметить, что, во-первых, два многочлена следует привести к одной степени (просто дополнив коэффициенты одного из них нулями). Во-вторых, в результате произведения двух многочленов степени n получается многочлен степени $2n - 1$, поэтому, чтобы результат получился корректным, предварительно нужно удвоить степени каждого многочлена (опять же, дополнив их нулевыми коэффициентами).

Быстрое преобразование Фурье

Быстрое преобразование Фурье (fast Fourier transform) — это метод, позволяющий вычислять ДПФ за время $O(n \log n)$. Этот метод основывается на свойствах комплексных корней из единицы (а именно, на том, что степени одних корней дают другие корни).

Основная идея БПФ заключается в разделении вектора коэффициентов на два вектора, рекурсивном вычислении ДПФ для них, и объединении результатов в одно БПФ.

Итак, пусть имеется многочлен $A(x)$ степени n, где n — степень двойки, и $n > 1$:

$$A(x) = a_0 x^0 + a_1 x^1 + \ldots + a_{n-1} x^{n-1}.$$

Разделим его на два многочлена, один — с чётными, а другой — с нечётными коэффициентами:

$$A_0(x) = a_0 x^0 + a_2 x^1 + \ldots + a_{n-2} x^{n/2-1},$$
$$A_1(x) = a_1 x^0 + a_3 x^1 + \ldots + a_{n-1} x^{n/2-1}.$$

Нетрудно убедиться, что:

$$A(x) = A_0(x^2) + x A_1(x^2). \qquad (1)$$

Многочлены A_0 и A_1 имеют вдвое меньшую степень, чем многочлен A. Если мы сможем за линейное время по вычисленным $\mathrm{DFT}(A_0)$ и $\mathrm{DFT}(A_1)$ вычислить $\mathrm{DFT}(A)$, то мы и получим искомый алгоритм быстрого преобразования Фурье (т.к. это стандартная схема алгоритма "разделяй и властвуй", и для неё известна асимптотическая оценка $O(n \log n)$).

Итак, пусть мы имеем вычисленные вектора $\{y_k^0\}_{k=0}^{n/2-1} = \mathrm{DFT}(A_0)$ и $\{y_k^1\}_{k=0}^{n/2-1} = \mathrm{DFT}(A_1)$. Найдём выражения для $\{y_k\}_{k=0}^{n-1} = \mathrm{DFT}(A)$.

Во-первых, вспоминая (1), мы сразу получаем значения для первой половины коэффициентов:

$$y_k = y_k^0 + w_n^k y_k^1, \qquad k = 0 \ldots n/2 - 1.$$

Для второй половины коэффициентов после преобразований также получаем простую формулу:

$$y_{k+n/2} = A(w_n^{k+n/2}) = A_0(w_n^{2k+n}) + w_n^{k+n/2} A_1(w_n^{2k+n}) = A_0(w_n^{2k} w_n^n) + w_n^k w_n^{n/2} A_1(w_n^{2k} w_n^n) =$$
$$= A_0(w_n^{2k}) - w_n^k A_1(w_n^{2k}) = y_k^0 - w_n^k y_k^1.$$

(Здесь мы воспользовались (1), а также тождествами $w_n^n = 1$, $w_n^{n/2} = -1$.)

Итак, в результате мы получили формулы для вычисления всего вектора $\{y_k\}$:

$$y_k = y_k^0 + w_n^k y_k^1, \qquad k = 0 \ldots n/2 - 1,$$
$$y_{k+n/2} = y_k^0 - w_n^k y_k^1, \qquad k = 0 \ldots n/2 - 1.$$

(эти формулы, т.е. две формулы вида $a + bc$ и $a - bc$, иногда называют "преобразование бабочки" ("butterfly operation"))

Тем самым, мы окончательно построили алгоритм БПФ.

Обратное БПФ

Итак, пусть дан вектор $(y_0, y_1, \ldots, y_{n-1})$ — значения многочлена A степени n в точках $x = w_n^k$.

Требуется восстановить коэффициенты $(a_0, a_1, \ldots, a_{n-1})$ многочлена. Эта известная задача называется **интерполяцией**, для этой задачи есть и общие алгоритмы решения, однако в данном случае будет получен очень простой алгоритм (простой тем, что он практически не отличается от прямого БПФ).

ДПФ мы можем записать, согласно его определению, в матричном виде:

$$\begin{pmatrix} w_n^0 & w_n^0 & w_n^0 & w_n^0 & \cdots & w_n^0 \\ w_n^0 & w_n^1 & w_n^2 & w_n^3 & \cdots & w_n^{n-1} \\ w_n^0 & w_n^2 & w_n^4 & w_n^6 & \cdots & w_n^{2(n-1)} \\ w_n^0 & w_n^3 & w_n^6 & w_n^9 & \cdots & w_n^{3(n-1)} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ w_n^0 & w_n^{n-1} & w_n^{2(n-1)} & w_n^{3(n-1)} & \cdots & w_n^{(n-1)(n-1)} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_{n-1} \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_{n-1} \end{pmatrix}.$$

Тогда вектор $(a_0, a_1, \ldots, a_{n-1})$ можно найти, умножив вектор $(y_0, y_1, \ldots, y_{n-1})$ на обратную матрицу к матрице, стоящей слева (которая, кстати, называется матрицей Вандермонда):

$$\begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_{n-1} \end{pmatrix} = \begin{pmatrix} w_n^0 & w_n^0 & w_n^0 & w_n^0 & \cdots & w_n^0 \\ w_n^0 & w_n^1 & w_n^2 & w_n^3 & \cdots & w_n^{n-1} \\ w_n^0 & w_n^2 & w_n^4 & w_n^6 & \cdots & w_n^{2(n-1)} \\ w_n^0 & w_n^3 & w_n^6 & w_n^9 & \cdots & w_n^{3(n-1)} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ w_n^0 & w_n^{n-1} & w_n^{2(n-1)} & w_n^{3(n-1)} & \cdots & w_n^{(n-1)(n-1)} \end{pmatrix}^{-1} \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_{n-1} \end{pmatrix}.$$

Непосредственной проверкой можно убедиться в том, что эта обратная матрица такова:

$$\frac{1}{n} \begin{pmatrix} w_n^0 & w_n^0 & w_n^0 & w_n^0 & \cdots & w_n^0 \\ w_n^0 & w_n^{-1} & w_n^{-2} & w_n^{-3} & \cdots & w_n^{-(n-1)} \\ w_n^0 & w_n^{-2} & w_n^{-4} & w_n^{-6} & \cdots & w_n^{-2(n-1)} \\ w_n^0 & w_n^{-3} & w_n^{-6} & w_n^{-9} & \cdots & w_n^{-3(n-1)} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ w_n^0 & w_n^{-(n-1)} & w_n^{-2(n-1)} & w_n^{-3(n-1)} & \cdots & w_n^{-(n-1)(n-1)} \end{pmatrix}.$$

Таким образом, получаем формулу:

$$a_k = \frac{1}{n} \sum_{j=0}^{n-1} y_j w_n^{-kj}.$$

Сравнивая её с формулой для y_k:

$$y_k = \sum_{j=0}^{n-1} a_j w_n^{kj},$$

мы замечаем, что эти две задачи почти ничем не отличаются, поэтому коэффициенты a_k можно находить таким же алгоритмом "разделяй и властвуй", как и прямое БПФ, только вместо w_n^k везде надо использовать w_n^{-k}, а каждый элемент результата надо разделить на n.

Таким образом, вычисление обратного ДПФ почти не отличается от вычисления прямого ДПФ, и его также можно выполнять за время $O(n \log n)$.

Реализация

Рассмотрим простую рекурсивную **реализацию БПФ** и обратного БПФ, реализуем их в виде одной функции, поскольку различия между прямым и обратным БПФ минимальны. Для хранения комплексных чисел воспользуемся стандартным в C++ STL типом complex (определённым в заголовочном файле <complex>).

```
typedef complex<double> base;

void fft (vector<base> & a, bool invert) {
        int n = (int) a.size();
```

```
        if (n == 1)  return;

        vector<base> a0 (n/2),  a1 (n/2);
        for (int i=0, j=0; i<n; i+=2, ++j) {
                a0[j] = a[i];
                a1[j] = a[i+1];
        }
        fft (a0, invert);
        fft (a1, invert);

        double ang = 2*PI/n * (invert ? -1 : 1);
        base w (1),  wn (cos(ang), sin(ang));
        for (int i=0; i<n/2; ++i) {
                a[i] = a0[i] + w * a1[i];
                a[i+n/2] = a0[i] - w * a1[i];
                if (invert)
                        a[i] /= 2,  a[i+n/2] /= 2;
                w *= wn;
        }
}
```

В аргумент a функции передаётся входной вектор коэффициентов, в нём же и будет содержаться результат. Аргумент invert показывает, прямое или обратное ДПФ следует вычислить. Внутри функции сначала проверяется, что если длина вектора a равна единице, то ничего делать не надо - он сам и является ответом. Иначе вектор a разделяется на два вектора $\mathrm{a0}$ и $\mathrm{a1}$, для которых рекурсивно вычисляется ДПФ. Затем вычисляется величина w_n, и заводится переменная w, содержащая текущую степень w_n. Затем вычисляются элементы результирующего ДПФ по вышеописанным формулам.

Если указан флаг $\mathrm{invert} = \mathrm{true}$, то w_n заменяется на w_n^{-1}, а каждый элемент результата делится на 2 (учитывая, что эти деления на 2 произойдут в каждом уровне рекурсии, то в итоге как раз получится, что все элементы поделятся на n).

Тогда функция для **перемножения двух многочленов** будет выглядеть следующим образом:

```
void multiply (const vector<int> & a, const vector<int> & b, vector<int> &
res) {
        vector<base> fa (a.begin(), a.end()),  fb (b.begin(), b.end());
        size_t n = 1;
        while (n < max (a.size(), b.size()))  n <<= 1;
        n <<= 1;
        fa.resize (n),  fb.resize (n);

        fft (fa, false),  fft (fb, false);
        for (size_t i=0; i<n; ++i)
                fa[i] *= fb[i];
        fft (fa, true);

        res.resize (n);
        for (size_t i=0; i<n; ++i)
                res[i] = int (fa[i].real() + 0.5);
}
```

Эта функция работает с многочленами с целочисленными коэффициентами (хотя, понятно, теоретически ничто не мешает ей работать и с дробными коэффициентами). Однако здесь проявляется проблема большой погрешности при вычислении ДПФ: погрешность может оказаться значительной, поэтому округлять числа лучше самым надёжным способом — прибавлением 0.5 и последующим округлением вниз.

Наконец, функция для **перемножения двух длинных чисел** практически ничем не отличается от функции для перемножения многочленов. Единственная особенность — что после выполнения умножения чисел как многочлены их следует нормализовать, т.е. выполнить все переносы разрядов:

```
        int carry = 0;
        for (size_t i=0; i<n; ++i) {
                res[i] += carry;
                carry = res[i] / 10;
                res[i] %= 10;
        }
```

(Поскольку длина произведения двух чисел никогда не превзойдёт суммарной длины чисел, то размера вектора res хватит, чтобы выполнить все переносы.)

Улучшенная реализация: вычисления "на месте" без дополнительной памяти

Для увеличения эффективности откажемся от рекурсии в явном виде. В приведённой выше рекурсивной реализации мы явно разделяли вектор a на два вектора — элементы на чётных позициях отнесли к одному временно созданному вектору, а на нечётных — к другому. Однако, если бы мы переупорядочили элементы определённым образом, то необходимость в создании временных векторов тогда бы отпала (т.е. все вычисления мы могли бы производить "на месте", прямо в самом векторе a).

Заметим, что на первом уровне рекурсии элементы, младшие (первые) биты позиций которых равны нулю, относятся к вектору a_0, а младшие биты позиций которых равны единице — к вектору a_1. На втором уровне рекурсии выполняется то же самое, но уже для вторых битов, и т.д. Поэтому если мы в позиции i каждого элемента $a[i]$ инвертируем порядок битов, и переупорядочим элементы массива a в соответствии с новыми индексами, то мы и получим искомый порядок (он называется **поразрядно обратной перестановкой** (bit-reversal permutation)).

Например, для $n = 8$ этот порядок имеет вид:

$$a = \Big\{ \big[(a_0, a_4), (a_2, a_6)\big], \big[(a_1, a_5), (a_3, a_7)\big] \Big\}.$$

Действительно, на первом уровне рекурсии (окружено фигурными скобками) обычного рекурсивного алгоритма происходит разделение вектора на две части: $[a_0, a_2, a_4, a_6]$ и $[a_1, a_3, a_5, a_7]$. Как мы видим, в поразрядно обратной перестановке этому соответствует просто разделение вектора на две половинки: первые $n/2$ элементов, и последние $n/2$ элементов. Затем происходит рекурсивный вызов от каждой половинки; пусть результирующее ДПФ от каждой из них было возвращено на месте самих элементов (т.е. в первой и второй половинах вектора a соответственно):

$$a = \Big\{ \big[y_0^0,\ y_1^0,\ y_2^0,\ y_3^0\big], \big[y_0^1,\ y_1^1,\ y_2^1,\ y_3^1\big] \Big\}.$$

Теперь нам надо выполнить объединение двух ДПФ в одно для всего вектора. Но элементы встали так удачно, что и объединение можно выполнить прямо в этом массиве. Действительно, возьмём элементы y_0^0 и y_0^1, применим к ним преобразование бабочки, и результат поставим на их месте — и это место и окажется тем самым, которое и должно было получиться:

$$a = \Big\{ \big[y_0^0 + w_n^0 y_0^1,\ y_1^0,\ y_2^0,\ y_3^0\big], \big[y_0^0 - w_n^0 y_0^1,\ y_1^1,\ y_2^1,\ y_3^1\big] \Big\}.$$

Аналогично, применяем преобразование бабочки к y_1^0 и y_1^1 и результат ставим на их место, и т.д. В итоге получаем:

$$a = \Big\{ \big[y_0^0 + w_n^0 y_0^1,\ y_1^0 + w_n^1 y_1^1,\ y_2^0 + w_n^2 y_2^1,\ y_3^0 + w_n^3 y_3^1\big],$$
$$\big[y_0^0 - w_n^0 y_0^1,\ y_1^0 - w_n^1 y_1^1,\ y_2^0 - w_n^2 y_2^1,\ y_3^0 - w_n^3 y_3^1\big] \Big\}.$$

Т.е. мы получили именно искомое ДПФ от вектора a.

Мы описали процесс вычисления ДПФ на первом уровне рекурсии, но понятно, что те же самые рассуждения верны и для всех остальных уровней рекурсии. Таким образом, **после применения поразрядно обратной перестановки вычислять ДПФ можно на месте**, без привлечения дополнительных массивов.

Но теперь можно **избавиться и от рекурсии** в явном виде. Итак, мы применили поразрядно обратную перестановку элементов. Теперь выполним всю работу, выполняемую нижним уровнем рекурсии, т.е. вектор a разделим на пары элементов, для каждого применим преобразование бабочки, в результате в векторе a будут находиться результаты работы нижнего уровня рекурсии. На следующем шаге разделим вектор a на четвёрки элементов, к каждой применим преобразование бабочки, в результате получим ДПФ для каждой четвёрки. И так далее, наконец, на последнем шаге мы, получив результаты ДПФ для двух половинок вектора a, применим к ним преобразование бабочки и получим ДПФ для всего вектора a.

Итак, реализация:

```
typedef complex<double> base;

int rev (int num, int lg_n) {
        int res = 0;
```

```
        for (int i=0; i<lg_n; ++i)
                if (num & (1<<i))
                        res |= 1<<(lg_n-1-i);
        return res;
}

void fft (vector<base> & a, bool invert) {
        int n = (int) a.size();
        int lg_n = 0;
        while ((1 << lg_n) < n)  ++lg_n;

        for (int i=0; i<n; ++i)
                if (i < rev(i,lg_n))
                        swap (a[i], a[rev(i,lg_n)]);

        for (int len=2; len<=n; len<<=1) {
                double ang = 2*PI/len * (invert ? -1 : 1);
                base wlen (cos(ang), sin(ang));
                for (int i=0; i<n; i+=len) {
                        base w (1);
                        for (int j=0; j<len/2; ++j) {
                                base u = a[i+j],  v = a[i+j+len/2] * w;
                                a[i+j] = u + v;
                                a[i+j+len/2] = u - v;
                                w *= wlen;
                        }
                }
        }
        if (invert)
                for (int i=0; i<n; ++i)
                        a[i] /= n;
}
```

Вначале к вектору a применяется поразрядно обратная перестановка, для чего вычисляется количество значащих бит ($\lg_n$) в числе n, и для каждой позиции i находится соответствующая ей позиция, битовая запись которой есть битовая запись числа i, записанная в обратном порядке. Если получившаяся в результате позиция оказалась больше i, то элементы в этих двух позициях надо обменять (если не это условие, то каждая пара обменяется дважды, и в итоге ничего не произойдёт).

Затем выполняется $\lg n - 1$ стадий алгоритма, на k-ой из которых ($k = 2 \ldots \lg n$) вычисляются ДПФ для блоков длины 2^k. Для всех этих блоков будет одно и то же значение первообразного корня w_{2^k}, которое и запоминается в переменной wlen. Цикл по i итерируется по блокам, а вложенный в него цикл по j применяет преобразование бабочки ко всем элементам блока.

Можно выполнить дальнейшую **оптимизацию реверса битов**. В предыдущей реализации мы явно проходили по всем битам числа, попутно строя поразрядно инвертированное число. Однако реверс битов можно выполнять и по-другому.

Например, пусть j — уже подсчитанное число, равное обратной перестановке битов числа i. Тогда, при переходе к следующему числу $i + 1$ мы должны и к числу j прибавить единицу, но прибавить её в такой "инвертированной" системе счисления. В обычной двоичной системе счисления прибавить единицу — значит удалить все единицы, стоящие на конце числа (т.е. группу младших единиц), а перед ними поставить единицу. Соответственно, в "инвертированной" системе мы должны идти по битам числа, начиная со старших, и пока там стоят единицы, удалять их и переходить к следующему биту; когда же встретится первый нулевой бит, поставить в него единицу и остановиться.

Итак, получаем такую реализацию:

```
typedef complex<double> base;

void fft (vector<base> & a, bool invert) {
        int n = (int) a.size();

        for (int i=1, j=0; i<n; ++i) {
                int bit = n >> 1;
                for (; j>=bit; bit>>=1)
                        j -= bit;
                j += bit;
                if (i < j)
                        swap (a[i], a[j]);
        }
```

```
        for (int len=2; len<=n; len<<=1) {
                double ang = 2*PI/len * (invert ? -1 : 1);
                base wlen (cos(ang), sin(ang));
                for (int i=0; i<n; i+=len) {
                        base w (1);
                        for (int j=0; j<len/2; ++j) {
                                base u = a[i+j],  v = a[i+j+len/2] * w;
                                a[i+j] = u + v;
                                a[i+j+len/2] = u - v;
                                w *= wlen;
                        }
                }
        }
        if (invert)
                for (int i=0; i<n; ++i)
                        a[i] /= n;
}
```

Дополнительные оптимизации

Приведём также список других оптимизаций, которые в совокупности позволяют заметно ускорить приведённую выше "улучшенную" реализацию:

- **Предпосчитать реверс битов** для всех чисел в некоторой глобальной таблице. Особенно легко это, когда размер n при всех вызовах одинаков.

  Эта оптимизация становится заметной при большом количестве вызовов $fft()$. Впрочем, эффект от неё можно заметить даже при трёх вызовах (три вызова — наиболее распространённая ситуация, т.е. когда требуется один раз перемножить два многочлена).

- Отказаться от использования vector (**перейти на обычные массивы**).

  Эффект от этого зависит от конкретного компилятора, однако обычно он присутствует и составляет примерно 10%-20%.

- Предпосчитать **все степени** числа $wlen$. В самом деле, в этом цикле алгоритма раз за разом производится проход по всем степеням числа $wlen$ от 0 до $len/2-1$:

  ```
  for (int i=0; i<n; i+=len) {
          base w (1);
          for (int j=0; j<len/2; ++j) {
                  [...]
                  w *= wlen;
          }
  }
  ```

  Соответственно, перед этим циклом мы можем предпосчитать в некотором массиве все требуемые степени, и избавиться тем самым от лишних умножений во вложенном цикле.

  Ориентировочное ускорение — 5-10%.

- Избавиться от **обращений к массивам по индексам**, использовать вместо этого указатели на текущие элементы массивов, продвигая их на 1 вправо на каждой итерации.

  На первый взгляд, оптимизирующие компиляторы должны быть способны самостоятельно справиться с этим, однако на практике оказывается, что замена обращений к массивам $a[i+j]$ и $a[i+j+len/2]$ на указатели ускоряет программу в распространённых компиляторах. Выигрыш составляет 5-10%.

- **Отказаться от стандартного типа комплексных чисел** $\mathrm{complex}$, переписав его на собственную реализацию.

  Опять же, это может показаться удивительным, но даже в современных компиляторах выигрыш от такого переписывания может составлять до нескольких десятков процентов! Это косвенно подтверждает расхожее утверждение, что компиляторы хуже справляются с шаблонными типами данных, оптимизируя работу с ними гораздо хуже, чем с не-шаблонными типами.

- Другой полезной оптимизацией является **отсечение по длине**: когда длина рабочего блока становится маленькой (скажем, 4), вычислять ДПФ для него "вручную". Если расписать эти случаи в виде явных формул при длине, равной $4/2$, то значения синусов-косинусов примут целочисленные значения, за счёт чего можно получить прирост скорости ещё на несколько десятков процентов.

Приведём здесь реализацию с описанными улучшениями (за исключением двух последних пунктов, которые приводят к чрезмерному разрастанию кода):

```
int rev[MAXN];
base wlen_pw[MAXN];

void fft (base a[], int n, bool invert) {
        for (int i=0; i<n; ++i)
                if (i < rev[i])
                        swap (a[i], a[rev[i]]);

        for (int len=2; len<=n; len<<=1) {
                double ang = 2*PI/len * (invert?-1:+1);
                int len2 = len>>1;

                base wlen (cos(ang), sin(ang));
                wlen_pw[0] = base (1, 0);
                for (int i=1; i<len2; ++i)
                        wlen_pw[i] = wlen_pw[i-1] * wlen;

                for (int i=0; i<n; i+=len) {
                        base t,
                                *pu = a+i,
                                *pv = a+i+len2,
                                *pu_end = a+i+len2,
                                *pw = wlen_pw;
                        for (; pu!=pu_end; ++pu, ++pv, ++pw) {
                                t = *pv * *pw;
                                *pv = *pu - t;
                                *pu += t;
                        }
                }
        }

        if (invert)
                for (int i=0; i<n; ++i)
                        a[i] /= n;
}

void calc_rev (int n, int log_n) {
        for (int i=0; i<n; ++i) {
                rev[i] = 0;
                for (int j=0; j<log_n; ++j)
                        if (i & (1<<j))
                                rev[i] |= 1<<(log_n-1-j);
        }
}
```

На распространённых компиляторах данная реализация быстрее предыдущего "улучшенного" варианта в 2-3 раза.

Дискретное преобразование Фурье в модульной арифметике

В основе дискретного преобразования Фурье лежат комплексные числа, корни n-ой степени из единицы. Для эффективного его вычисления использовались такие особенности корней, как существование n различных корней, образующих группу (т.е. степень одного корня — всегда другой корень; среди них есть один элемент — генератор группы, называемый примитивным корнем).

Но то же самое верно и в отношении корней n-ой степени из единицы в модульной арифметике. Точнее, не для любого модуля p найдётся n различных корней из единицы, однако такие модули всё же существуют. По-прежнему нам важно найти среди них примитивный корень, т.е.:

$$\begin{aligned} &(w_n)^n = 1 \pmod p, \\ &(w_n)^k \neq 1 \pmod p, \qquad 1 \le k < n. \end{aligned}$$

Все остальные $n-1$ корней n-ой степени из единицы по модулю p можно получить как степени примитивного корня w_n (как и в комплексном случае).

Для применения в алгоритме Быстрого преобразования Фурье нам было нужно, чтобы примивный корень существовал для некоторого n, являвшегося степенью двойки, а также всех меньших степеней. И если в комплексном случае примитивный корень существовал для любого n, то в случае модульной арифметики это, вообще говоря, не

так. Однако, заметим, что если $n = 2^k$, т.е. k-ая степень двойки, то по модулю $m = 2^{k-1}$ имеем:

$$(w_n^2)^m = (w_n)^n = 1 \pmod p,$$
$$(w_n^2)^k = w_n^{2k} \neq 1 \pmod p, \qquad 1 \le k < m.$$

Таким образом, если w_n — примитивный корень $n = 2^k$-ой степени из единицы, то w_n^2 — примитивный корень 2^{k-1}-ой степени из единицы. Следовательно, для всех степеней двойки, меньших n, примитивные корни нужной степени также существуют, и могут быть вычислены как соответствующие степени w_n.

Последний штрих — для обратного ДПФ мы использовали вместо w_n обратный ему элемент: w_n^{-1}. Но по простому модулю p обратный элемент также всегда найдётся.

Таким образом, все нужные нам свойства соблюдаются и в случае модульной арифметики, при условии, что мы выбрали некоторый достаточно большой модуль p и нашли в нём примитивный корень n-ой степени из единицы.

Например, можно взять такие значения: модуль $p = 7340033$, $w_{2^{20}} = 5$. Если этого модуля будет недостаточно, для нахождения другой пары можно воспользоваться фактом, что для модулей вида $c2^k + 1$ (но по-прежнему обязательно простых) всегда найдётся примитивный корень степени 2^k из единицы.

```
const int mod = 7340033;
const int root = 5;
const int root_1 = 4404020;
const int root_pw = 1<<20;

void fft (vector<int> & a, bool invert) {
        int n = (int) a.size();

        for (int i=1, j=0; i<n; ++i) {
                int bit = n >> 1;
                for (; j>=bit; bit>>=1)
                        j -= bit;
                j += bit;
                if (i < j)
                        swap (a[i], a[j]);
        }

        for (int len=2; len<=n; len<<=1) {
                int wlen = invert ? root_1 : root;
                for (int i=len; i<root_pw; i<<=1)
                        wlen = int (wlen * 1ll * wlen % mod);
                for (int i=0; i<n; i+=len) {
                        int w = 1;
                        for (int j=0; j<len/2; ++j) {
                                int u = a[i+j],  v = int (a[i+j+len/2] * 1ll
* w % mod);
                                a[i+j] = u+v < mod ? u+v : u+v-mod;
                                a[i+j+len/2] = u-v >= 0 ? u-v : u-v+mod;
                                w = int (w * 1ll * wlen % mod);
                        }
                }
        }
        if (invert) {
                int nrev = reverse (n, mod);
                for (int i=0; i<n; ++i)
                        a[i] = int (a[i] * 1ll * nrev % mod);
        }
}
```

Здесь функция $\mathrm{reverse}$ находит обратный к n элемент по модулю mod (см. Обратный элемент в поле по модулю). Константы mod, root $\mathrm{root_pw}$ определяют модуль и примитивный корень, а $\mathrm{root_1}$ — обратный к root элемент по модулю mod.

Как показывает практика, реализация целочисленного ДПФ работает даже медленней реализации с комплексными числами (из-за огромного количества операций взятия по модулю), однако она имеет такие преимущества, как меньшее использование памяти и отсутствие погрешностей округления.

Некоторые применения

Помимо непосредственного применения для перемножения многочленов или длинных чисел, опишем здесь

некоторые другие приложения дискретного преобразования Фурье.

Всевозможные суммы

Задача: даны два массива $a[]$ и $b[]$. Требуется найти всевозможные числа вида $a[i] + b[j]$, и для каждого такого числа вывести количество способов получить его.

Например, для $a = (1, 2, 3)$ и $b = (2, 4)$ получаем: число 3 можно получить 1 способом, 4 — также одним, 5 — 2, 6 — 1, 7 — 1.

Построим по массивам a и b два многочлена A и B. В качестве степеней в многочлене будут выступать сами числа, т. е. значения $a[i]$ ($b[i]$), а в качестве коэффициентов при них — сколько раз это число встречается в массиве a (b).

Тогда, перемножив эти два многочлена за $O(n \log n)$, мы получим многочлен C, где в качестве степеней будут всевозможные числа вида $a[i] + b[i]$, а коэффициенты при них будут как раз искомыми количествами

Всевозможные скалярные произведения

Даны два массива $a[]$ и $b[]$ одной длины n. Требуется вывести значения каждого скалярного произведения вектора a на очередной циклический сдвиг вектора b.

Инвертируем массив a и припишем к нему в конец n нулей, а к массиву b — просто припишем самого себя. Затем перемножим их как многочлены. Теперь рассмотрим коэффициенты произведения $c[n \dots 2n-1]$ (как всегда, все индексы в 0-индексации). Имеем:

$$c[k] = \sum_{i+j=k} a[i]b[j].$$

Поскольку все элементы $a[i] = 0,\ i = n \dots 2n-1$, то мы получаем:

$$c[k] = \sum_{i=0}^{n-1} a[i]b[k-i].$$

Нетрудно увидеть в этой сумме, что это именно скалярное произведение вектора a на $k-n-1$-ый циклический сдвиг. Таким образом, эти коэффициенты (начиная с $n-1$-го и закачивая $2n-2$-ым) — и есть ответ на задачу.

Решение получилось с асимптотикой $O(n \log n)$.

Две полоски

Даны две полоски, заданные как два булевских (т.е. числовых со значениями 0 или 1) массива $a[]$ и $b[]$. Требуется найти все такие позиции на первой полоске, что если приложить, начиная с этой позиции, вторую полоску, ни в каком месте не получится true сразу на обеих полосках. Эту задачу можно переформулировать таким образом: дана карта полоски, в виде 0/1 — можно вставать в эту клетку или нет, и дана некоторая фигурка в виде шаблона (в виде массива, в котором 0 — нет клетки, 1 — есть), требуется найти все позиции в полоске, к которым можно приложить фигурку.

Эта задача фактически ничем не отличается от предыдущей задачи — задачи о скалярном произведении. Действительно, скалярное произведение двух 0/1 массивов — это количество элементов, в которых одновременно оказались единицы. Наша задача в том, чтобы найти все циклические сдвиги второй полоски так, чтобы не нашлось ни одного элемента, в котором бы в обеих полосках оказались единицы. Т.е. мы должны найти все циклические сдвиги второго массива, при которых скалярное произведение равно нулю.

Таким образом, эту задачу мы решили за $O(n \log n)$.

Поиск в ширину

Поиск в ширину (обход в ширину, breadth-first search) — это один из основных алгоритмов на графах.

В результате поиска в ширину находится путь кратчайшей длины в невзвешененном графе, т.е. путь, содержащий наименьшее число рёбер.

Алгоритм работает за $O(n + m)$, где n — число вершин, m — число рёбер.

Описание алгоритма

На вход алгоритма подаётся заданный граф (невзвешенный), и номер стартовой вершины s. Граф может быть как ориентированным, так и неориентированным, для алгоритма это не важно.

Сам алгоритм можно понимать как процесс "поджигания" графа: на нулевом шаге поджигаем только вершину s. На каждом следующем шаге огонь с каждой уже горящей вершины перекидывается на всех её соседей; т.е. за одну итерацию алгоритма происходит расширение "кольца огня" в ширину на единицу (отсюда и название алгоритма).

Более строго это можно представить следующим образом. Создадим очередь q, в которую будут помещаться горящие вершины, а также заведём булевский массив $\text{used}[]$, в котором для каждой вершины будем отмечать, горит она уже или нет (или иными словами, была ли она посещена).

Изначально в очередь помещается только вершина s, и $\text{used}[\text{s}] = \text{true}$, а для всех остальных вершин $\text{used}[] = \text{false}$. Затем алгоритм представляет собой цикл: пока очередь не пуста, достать из её головы одну вершину, просмотреть все рёбра, исходящие из этой вершины, и если какие-то из просмотренных вершин ещё не горят, то поджечь их и поместить в конец очереди.

В итоге, когда очередь опустеет, обход в ширину обойдёт все достижимые из s вершины, причём до каждой дойдёт кратчайшим путём. Также можно посчитать длины кратчайших путей (для чего просто надо завести массив длин путей $d[]$), и компактно сохранить информацию, достаточную для восстановления всех этих кратчайших путей (для этого надо завести массив "предков" $p[]$, в котором для каждой вершины хранить номер вершины, по которой мы попали в эту вершину).

Реализация

Реализуем вышеописанный алгоритм на языке C++.

Входные данные:

```
vector < vector<int> > g; // граф
int n; // число вершин
int s; // стартовая вершина (вершины везде нумеруются с нуля)

// чтение графа
...
```

Сам обход:

```
queue<int> q;
q.push (s);
vector<bool> used (n);
vector<int> d (n), p (n);
used[s] = true;
p[s] = -1;
while (!q.empty()) {
        int v = q.front();
        q.pop();
        for (size_t i=0; i<g[v].size(); ++i) {
                int to = g[v][i];
                if (!used[to]) {
                        used[to] = true;
                        q.push (to);
```

```
                        d[to] = d[v] + 1;
                        p[to] = v;
                }
        }
}
```

Если теперь надо восстановить и вывести кратчайший путь до какой-то вершины to, это можно сделать следующим образом:

```
if (!used[to])
        cout << "No path!";
else {
        vector<int> path;
        for (int v=to; v!=-1; v=p[v])
                path.push_back (v);
        reverse (path.begin(), path.end());
        cout << "Path: ";
        for (size_t i=0; i<path.size(); ++i)
                cout << path[i] + 1 << " ";
}
```

Приложения алгоритма

- Поиск **кратчайшего пути** в невзвешенном графе.
- Поиск **компонент связности** в графе за $O(n+m)$.

  Для этого мы просто запускаем обход в ширину от каждой вершины, за исключением вершин, оставшихся посещёнными ($\mathrm{used} = \mathrm{true}$) после предыдущих запусков. Таким образом, мы выполняем обычный запуск в ширину от каждой вершины, но не обнуляем каждый раз массив $\mathrm{used}[]$, за счёт чего мы каждый раз будем обходить новую компоненту связности, а суммарное время работы алгоритма составит по-прежнему $O(n+m)$ (такие несколько запусков обхода на графе без обнуления массива used называются серией обходов в ширину).
- Нахождения решения какой-либо задачи (игры) **с наименьшим числом ходов**, если каждое состояние системы можно представить вершиной графа, а переходы из одного состояния в другое — рёбрами графа.

  Классический пример — игра, где робот двигается по полю, при этом он может передвигать ящики, находящиеся на этом же поле, и требуется за наименьшее число ходов передвинуть ящики в требуемые позиции. Решается это обходом в ширину по графу, где состоянием (вершиной) является набор координат: координаты робота, и координаты всех коробок.
- Нахождение кратчайшего пути в 0-1-**графе** (т.е. графе взвешенном, но с весами равными только 0 либо 1): достаточно немного модифицировать поиск в ширину: если текущее ребро нулевого веса, и происходит улучшение расстояния до какой-то вершины, то эту вершину добавляем не в конец, а в начало очереди.
- Нахождение **кратчайшего цикла** в неориентированном невзвешенном графе: производим поиск в ширину из каждой вершины; как только в процессе обхода мы пытаемся пойти из текущей вершины по какому-то ребру в уже посещённую вершину, то это означает, что мы нашли кратчайший цикл, и останавливаем обход в ширину; среди всех таких найденных циклов (по одному от каждого запуска обхода) выбираем кратчайший.
- Найти все рёбра, лежащие **на каком-либо кратчайшем пути** между заданной парой вершин (a, b). Для этого надо запустить 2 поиска в ширину: из a, и из b. Обозначим через $d_a[]$ массив кратчайших расстояний, полученный в результате первого обхода, а через $d_b[]$ — в результате второго обхода. Теперь для любого ребра (u, v) легко проверить, лежит ли он на каком-либо кратчайшем пути: критерием будет условие $d_a[u] + 1 + d_b[v] = d_a[b]$.
- Найти все вершины, лежащие **на каком-либо кратчайшем пути** между заданной парой вершин (a, b). Для этого надо запустить 2 поиска в ширину: из a, и из b. Обозначим через $d_a[]$ массив кратчайших расстояний, полученный в результате первого обхода, а через $d_b[]$ — в результате второго обхода. Теперь для любой вершины v легко проверить, лежит ли он на каком-либо кратчайшем пути: критерием будет условие $d_a[v] + d_b[v] = d_a[b]$.
- Найти **кратчайший чётный путь** в графе (т.е. путь чётной длины). Для этого надо построить вспомогательный граф, вершинами которого будут состояния (v, c), где v — номер текущей вершины, $c = 0 \ldots 1$ — текущая чётность. Любое ребро (a, b) исходного графа в этом новом графе превратится в два ребра $((u, 0), (v, 1))$ и $((u, 1), (v, 0))$. После этого на этом графе надо обходом в ширину найти кратчайший путь из стартовой вершины в конечную, с чётностью, равной 0.

Задачи в online judges

Список задач, которые можно сдать, используя обход в ширину:

- SGU #213 "Strong Defence" [сложность: средняя]

Поиск в глубину

Это один из основных алгоритмов на графах.

В результате поиска в глубину находится лексикографически первый путь в графе.

Алгоритм работает за O (N+M).

Применения алгоритма

- Поиск любого пути в графе.
- Поиск лексикографически первого пути в графе.
- Проверка, является ли одна вершина дерева предком другой:
  В начале и конце итерации поиска в глубину будет запоминать "время" захода и выхода в каждой вершине. Теперь за O (1) можно найти ответ: вершина i является предком вершины j тогда и только тогда, когда $start_i < start_j$ и $end_i > end_j$.
- Задача LCA (наименьший общий предок).
- Топологическая сортировка:
  Запускаем серию поисков в глубину, чтобы обойти все вершины графа. Отсортируем вершины по времени выхода по убыванию - это и будет ответом.
- Проверка графа на ацикличность и нахождение цикла
- Поиск компонент сильной связности:
  Сначала делаем топологическую сортировку, потом транспонируем граф и проводим снова серию поисков в глубину в порядке, определяемом топологической сортировкой. Каждое дерево поиска - сильносвязная компонента.
- Поиск мостов:
  Сначала превращаем граф в ориентированный, делая серию поисков в глубину, и ориентируя каждое ребро так, как мы пытались по нему пройти. Затем находим сильносвязные компоненты. Мостами являются те рёбра, концы которых принадлежат разным сильносвязным компонентам.

Реализация

```
vector < vector<int> > g; // граф
int n; // число вершин

vector<int> color; // цвет вершины (0, 1, или 2)

vector<int> time_in, time_out; // "времена" захода и выхода из вершины
int dfs_timer = 0; // "таймер" для определения времён

void dfs (int v) {
        time_in[v] = dfs_timer++;
        color[v] = 1;
        for (vector<int>::iterator i=g[v].begin(); i!=g[v].end(); ++i)
                if (color[*i] == 0)
                        dfs (*i);
        color[v] = 2;
        time_out[v] = dfs_timer++;
}
```

Это наиболее общий код. Во многих случаях времена захода и выхода из вершины не важны, так же как и не важны цвета вершин (но тогда надо будет ввести аналогичный по смыслу булевский массив used). Вот наиболее простая реализация:

```
vector < vector<int> > g; // граф
int n; // число вершин

vector<char> used;

void dfs (int v) {
        used[v] = true;
        for (vector<int>::iterator i=g[v].begin(); i!=g[v].end(); ++i)
                if (!used[*i])
                        dfs (*i);
}
```

Топологическая сортировка

Дан ориентированный граф с n вершинами и m рёбрами. Требуется **перенумеровать** его вершины таким образом, чтобы каждое рёбро вело из вершины с меньшим номером в вершину с большим.

Иными словами, требуется найти перестановку вершин (**топологический порядок**), соответствующую порядку, задаваемому всеми рёбрами графа.

Топологическая сортировка может быть **не единственной** (например, если граф — пустой; или если есть три такие вершины a, b, c, что из a есть пути в b и в c, но ни из b в c, ни из c в b добраться нельзя).

Топологической сортировки может **не существовать** вовсе — если граф содержит циклы (поскольку при этом возникает противоречие: есть путь и из одной вершины в другую, и наоборот).

Распространённая задача на топологическую сортировку — следующая. Есть n переменных, значения которых нам неизвестны. Известно лишь про некоторые пары переменных, что одна переменная меньше другой. Требуется проверить, не противоречивы ли эти неравенства, и если нет, выдать переменные в порядке их возрастания (если решений несколько — выдать любое). Легко заметить, что это в точности и есть задача о поиске топологической сортировки в графе из n вершин.

Алгоритм

Для решения воспользуемся обходом в глубину.

Предположим, что граф ацикличен, т.е. решение существует. Что делает обход в глубину? При запуске из какой-то вершины v он пытается запуститься вдоль всех рёбер, исходящих из v. Вдоль тех рёбер, концы которых уже были посещены ранее, он не проходит, а вдоль всех остальных — проходит и вызывает себя от их концов.

Таким образом, к моменту выхода из вызова $\mathrm{dfs}(v)$ все вершины, достижимые из v как непосредственно (по одному ребру), так и косвенно (по пути) — все такие вершины уже посещены обходом. Следовательно, если мы будем в момент выхода из $\mathrm{dfs}(v)$ добавлять нашу вершину в начало некоего списка, то в конце концов в этом списке получится **топологическая сортировка**.

Эти объяснения можно представить и в несколько ином свете, с помощью понятия "**времени выхода**" обхода в глубину. Время выхода для каждой вершины v — это момент времени, в который закончил работать вызов $\mathrm{dfs}(v)$ обхода в глубину от неё (времена выхода можно занумеровать от 1 до n). Легко понять, что при обходе в глубину время выхода из какой-либо вершины v всегда больше, чем время выхода из всех вершин, достижимых из неё (т.к. они были посещены либо до вызова $\mathrm{dfs}(v)$, либо во время него). Таким образом, искомая топологическая сортировка — это сортировка в порядке убывания времён выхода.

Реализация

Приведём реализацию, предполагающую, что граф ацикличен, т.е. искомая топологическая сортировка существует. При необходимости проверку графа на ацикличность легко вставить в обход в глубину, как описано в статье по обходу в глубину.

```
int n; // число вершин
vector<int> g[MAXN]; // граф
bool used[MAXN];
vector<int> ans;

void dfs (int v) {
        used[v] = true;
        for (size_t i=0; i<g[v].size(); ++i) {
                int to = g[v][i];
                if (!used[to])
                        dfs (to);
        }
        ans.push_back (v);
}

void topological_sort() {
        for (int i=0; i<n; ++i)
                used[i] = false;
        ans.clear();
```

```
        for (int i=0; i<n; ++i)
                if (!used[i])
                        dfs (i);
        reverse (ans.begin(), ans.end());
}
```

Здесь константе $MAXN$ следует задать значение, равное максимально возможному числу вершин в графе.

Основная функция решения — это topological_sort, она инициализирует пометки обхода в глубину, запускает его, и ответ в итоге получается в векторе ans.

Задачи в online judges

Список задач, в которых требуется искать топологическую сортировку:

- UVA #10305 "Ordering Tasks" [сложность: низкая]
- UVA #124 "Following Orders" [сложность: низкая]
- UVA #200 "Rare Order" [сложность: низкая]

Алгоритм поиска компонент связности в графе

Дан неориентированный граф G с n вершинами и m рёбрами. Требуется найти в нём все компоненты связности, т. е. разбить вершины графа на несколько групп так, что внутри одной группы можно дойти от одной вершины до любой другой, а между разными группами — пути не существует.

Алгоритм решения

Для решения можно воспользоваться как обходом в глубину, так и обходом в ширину.

Фактически, мы будем производить **серию обходов**: сначала запустим обход из первой вершины, и все вершины, которые он при этом обошёл — образуют первую компоненту связности. Затем найдём первую из оставшихся вершин, которые ещё не были посещены, и запустим обход из неё, найдя тем самым вторую компоненту связности. И так далее, пока все вершины не станут помеченными.

Итоговая **асимптотика** составит $O(n+m)$: в самом деле, такой алгоритм не будет запускаться от одной и той же вершины дважды, а, значит, каждое ребро будет просмотрено ровно два раза (с одного конца и с другого конца).

Реализация

Для реализации чуть более удобным является обход в глубину:

```
int n;
vector<int> g[MAXN];
bool used[MAXN];
vector<int> comp;

void dfs (int v) {
        used[v] = true;
        comp.push_back (v);
        for (size_t i=0; i<g[v].size(); ++i) {
                int to = g[v][i];
                if (! used[to])
                        dfs (to);
        }
}

void find_comps() {
        for (int i=0; i<n; ++i)
                used[i] = false;
        for (int i=0; i<n; ++i)
                if (! used[i]) {
                        comp.clear();
                        dfs (i);

                        cout << "Component:";
                        for (size_t j=0; j<comp.size(); ++j)
                                cout << ' ' << comp[j];
                        cout << endl;
                }
}
```

Основная функция для вызова — $\mathrm{find_comps}()$, она находит и выводит компоненты связности графа.

Мы считаем, что граф задан списками смежности, т.е. $g[i]$ содержит список вершин, в которые есть рёбра из вершины i. Константе MAXN следует задать значение, равное максимально возможному количеству вершин в графе.

Вектор comp содержит список вершин в текущей компоненте связности.

Поиск компонент сильной связности, построение конденсации графа

Определения, постановка задачи

Дан ориентированный граф G, множество вершин которого V и множество рёбер — E. Петли и кратные рёбра допускаются. Обозначим через n количество вершин графа, через m — количество рёбер.

Компонентой сильной связности (strongly connected component) называется такое (максимальное по включению) подмножество вершин C, что любые две вершины этого подмножества достижимы друг из друга, т.е. для $\forall u, v \in C$:

$$u \mapsto v, v \mapsto u$$

где символом $\mapsto$ здесь и далее мы будем обозначать достижимость, т.е. существование пути из первой вершины во вторую.

Понятно, что компоненты сильной связности для данного графа не пересекаются, т.е. фактически это разбиение всех вершин графа. Отсюда логично определение **конденсации** G^{SCC} как графа, получаемого из данного графа сжатием каждой компоненты сильной связности в одну вершину. Каждой вершине графа конденсации соответствует компонента сильной связности графа G, а ориентированное ребро между двумя вершинами C_i и C_j графа конденсации проводится, если найдётся пара вершин $u \in C_i, v \in C_j$, между которыми существовало ребро в исходном графе, т.е. $(u, v) \in E$.

Важнейшим свойством графа конденсации является то, что он **ацикличен**. Действительно, предположим, что $C \mapsto C'$, докажем, что $C' \not\mapsto C$. Из определения конденсации получаем, что найдутся две вершины $u \in C$ и $v \in C'$, что $u \mapsto v$. Доказывать будем от противного, т.е. предположим, что $C' \mapsto C$, тогда найдутся две вершины $u' \in C$ и $v' \in C'$, что $v' \mapsto u'$. Но т.к. u и u' находятся в одной компоненте сильной связности, то между ними есть путь; аналогично для v и v'. В итоге, объединяя пути, получаем, что $v \mapsto u$, и одновременно $u \mapsto v$. Следовательно, u и v должны принадлежать одной компоненте сильной связности, т.е. получили противоречие, что и требовалось доказать.

Описываемый ниже алгоритм выделяет в данном графе все компоненты сильной связности. Построить по ним граф конденсации не составит труда.

Алгоритм

Описываемый здесь алгоритм был предложен независимо Косараю (Kosaraju) и Шариром (Sharir) в 1979 г. Это очень простой в реализации алгоритм, основанный на двух сериях поисков в глубину, и потому работающий за время $O(n + m)$.

На первом шаге алгоритма выполняется серия обходов в глубину, посещающая весь граф. Для этого мы проходимся по всем вершинам графа и из каждой ещё не посещённой вершины вызываем обход в глубину. При этом для каждой вершины v запомним **время выхода** $\text{tout}[v]$. Эти времена выхода играют ключевую роль в алгоритме, и эта роль выражена в приведённой ниже теореме.

Сначала введём обозначение: время выхода $\text{tout}[C]$ из компоненты C сильной связности определим как максимум из значений $\text{tout}[v]$ для всех $v \in C$. Кроме того, в доказательстве теоремы будут упоминаться и времена входа в каждую вершину $\text{tin}[v]$, и аналогично определим времена входа $\text{tin}[C]$ для каждой компоненты сильной связности как минимум из величин $\text{tin}[v]$ для всех $v \in C$.

Теорема. Пусть C и C' — две различные компоненты сильной связности, и пусть в графе конденсации между ними есть ребро (C, C'). Тогда $\text{tout}[C] > \text{tout}[C']$.

При доказательстве возникает два принципиально различных случая в зависимости от того, в какую из компонент первой зайдёт обход в глубину, т.е. в зависимости от соотношения между $\text{tin}[C]$ и $\text{tin}[C']$:

- Первой была достигнута компонента C. Это означает, что в какой-то момент времени обход в глубину заходит в некоторую вершину v компоненты C, при этом все остальные вершины компонент C и C' ещё не посещены. Но, т.к. по условию в графе конденсаций есть ребро (C, C'), то из вершины v будет достижима не только вся компонента C, но и вся компонента C'. Это означает, что при запуске из вершины v обход в глубину пройдёт по всем вершинам компонент C и C', а, значит, они станут потомками по отношению к v в дереве обхода в глубину, т.е. для любой вершины $u \in C \cup C', u \neq v$ будет выполнено $\text{tout}[v] > \text{tout}[u]$, ч.т.д.
- Первой была достигнута компонента C'. Опять же, в какой-то момент времени обход в глубину заходит в

некоторую вершину $v \in C'$, причём все остальные вершины компонент C и C' не посещены. Поскольку по условию в графе конденсаций существовало ребро (C, C'), то, вследствие ацикличности графа конденсаций, не существует обратного пути $C' \not\to C$, т.е. обход в глубину из вершины v не достигнет вершин C. Это означает, что они будут посещены обходом в глубину позже, откуда и следует $\text{tout}[C] > \text{tout}[C']$, ч.т.д.

Доказанная теорема является **основой алгоритма** поиска компонент сильной связности. Из неё следует, что любое ребро (C, C') в графе конденсаций идёт из компоненты с большей величиной tout в компоненту с меньшей величиной.

Если мы отсортируем все вершины $v \in V$ в порядке убывания времени выхода $\text{tout}[v]$, то первой окажется некоторая вершина u, принадлежащая "корневой" компоненте сильной связности, т.е. в которую не входит ни одно ребро в графе конденсаций. Теперь нам хотелось бы запустить такой обход из этой вершины u, который бы посетил только эту компоненту сильной связности и не зашёл ни в какую другую; научившись это делать, мы сможем постепенно выделить все компоненты сильной связности: удалив из графа вершины первой выделенной компоненты, мы снова найдём среди оставшихся вершину с наибольшей величиной tout, снова запустим из неё этот обход, и т.д.

Чтобы научиться делать такой обход, рассмотрим **транспонированный граф** G^T, т.е. граф, полученный из G изменением направления каждого ребра на противоположное. Нетрудно понять, что в этом графе будут те же компоненты сильной связности, что и в исходном графе. Более того, граф конденсации $(G^T)^{\text{SCC}}$ для него будет равен транспонированному графу конденсации исходного графа G^{SCC}. Это означает, что теперь из рассматриваемой нами "корневой" компоненты уже не будут выходить рёбра в другие компоненты.

Таким образом, чтобы обойти всю "корневую" компоненту сильной связности, содержащую некоторую вершину v, достаточно запустить обход из вершины v в графе G^T. Этот обход посетит все вершины этой компоненты сильной связности и только их. Как уже говорилось, дальше мы можем мысленно удалить эти вершины из графа, находить очередную вершину с максимальным значением $\text{tout}[v]$ и запускать обход на транспонированном графе из неё, и т.д.

Итак, мы построили следующий **алгоритм** выделения компонент сильной связности:

1 шаг. Запустить серию обходов в глубину графа G, которая возвращает вершины в порядке увеличения времени выхода tout, т.е. некоторый список order.

2 шаг. Построить транспонированный граф G^T. Запустить серию обходов в глубину/ширину этого графа в порядке, определяемом списком order (а именно, в обратном порядке, т.е. в порядке уменьшения времени выхода). Каждое множество вершин, достигнутое в результате очередного запуска обхода, и будет очередной компонентой сильной связности.

Асимптотика алгоритма, очевидно, равна $O(n + m)$, поскольку он представляет собой всего лишь два обхода в глубину/ширину.

Наконец, уместно отметить связь с понятием **топологической сортировки**. Во-первых, шаг 1 алгоритма представляет собой не что иное, как топологическую сортировку графа G (фактически именно это и означает сортировка вершин по времени выхода). Во-вторых, сама схема алгоритма такова, что и компоненты сильной связности он генерирует в порядке уменьшения их времён выхода, таким образом, он генерирует компоненты - вершины графа конденсации в порядке топологической сортировки.

Реализация

```
vector < vector<int> > g, gr;
vector<char> used;
vector<int> order, component;

void dfs1 (int v) {
        used[v] = true;
        for (size_t i=0; i<g[v].size(); ++i)
                if (!used[ g[v][i] ])
                        dfs1 (g[v][i]);
        order.push_back (v);
}

void dfs2 (int v) {
        used[v] = true;
        component.push_back (v);
        for (size_t i=0; i<gr[v].size(); ++i)
                if (!used[ gr[v][i] ])
                        dfs2 (gr[v][i]);
}

int main() {
```

```
        int n;
        ... чтение n ...
        for (;;) {
                int a, b;
                ... чтение очередного ребра (a,b) ...
                g[a].push_back (b);
                gr[b].push_back (a);
        }

        used.assign (n, false);
        for (int i=0; i<n; ++i)
                if (!used[i])
                        dfs1 (i);
        used.assign (n, false);
        for (int i=0; i<n; ++i) {
                int v = order[n-1-i];
                if (!used[v]) {
                        dfs2 (v);
                        ... вывод очередной component ...
                        component.clear();
                }
        }
}
```

Здесь в g хранится сам граф, а gr — транспонированный граф. Функция $dfs1$ выполняет обход в глубину на графе G, функция $dfs2$ — на транспонированном G^T. Функция $dfs1$ заполняет список $order$ вершинами в порядке увеличения времени выхода (фактически, делает топологическую сортировку). Функция $dfs2$ сохраняет все достигнутые вершины в списке $component$, который после каждого запуска будет содержать очередную компоненту сильной связности.

Литература

- Томас Кормен, Чарльз Лейзерсон, Рональд Ривест, Клиффорд Штайн. **Алгоритмы: Построение и анализ** [2005]
- M. Sharir. A strong-connectivity algorithm and its applications in data-flow analysis [1979]

Поиск мостов

Пусть дан неориентированный граф. Мостом называется такое ребро, удаление которого делает граф несвязным (или, точнее, увеличивает число компонент связности). Требуется найти все мосты в заданном графе.

Неформально эта задача ставится следующим образом: требуется найти на заданной карте дорог все "важные" дороги, т.е. такие дороги, что удаление любой из них приведёт к исчезновению пути между какой-то парой городов.

Ниже мы опишем алгоритм, основанный на поиске в глубину, и работающий за время $O(n+m)$, где n — количество вершин, m — рёбер в графе.

Заметим, что на сайте также описан онлайновый алгоритм поиска мостов — в отличие от описанного здесь алгоритма, онлайновый алгоритм умеет поддерживать все мосты графа в изменяющемся графе (имеются в виду добавления новых рёбер).

Алгоритм

Запустим обход в глубину из произвольной вершины графа; обозначим её через root. Заметим следующий **факт** (который несложно доказать):

- Пусть мы находимся в обходе в глубину, просматривая сейчас все рёбра из вершины v. Тогда, если текущее ребро (v, to) таково, что из вершины to и из любого её потомка в дереве обхода в глубину нет обратного ребра в вершину v или какого-либо её предка, то это ребро является мостом. В противном случае оно мостом не является. (В самом деле, мы этим условием проверяем, нет ли другого пути из v в to, кроме как спуск по ребру (v, to) дерева обхода в глубину.)

Теперь осталось научиться проверять этот факт для каждой вершины эффективно. Для этого воспользуемся "временами входа в вершину", вычисляемыми алгоритмом поиска в глубину.

Итак, пусть $tin[v]$ — это время захода поиска в глубину в вершину v. Теперь введём массив $fup[v]$, который и позволит нам отвечать на вышеописанные запросы. Время $fup[v]$ равно минимуму из времени захода в саму вершину $tin[v]$, времён захода в каждую вершину p, являющуюся концом некоторого обратного ребра (v, p), а также из всех значений $fup[to]$ для каждой вершины to, являющейся непосредственным сыном v в дереве поиска:

$$fup[v] = \min \begin{cases} tin[v], & \\ tin[p], & \text{for all (v,p) — back edge} \\ fup[to], & \text{for all (v,to) — tree edge} \end{cases}$$

(здесь "back edge" — обратное ребро, "tree edge" — ребро дерева)

Тогда, из вершины v или её потомка есть обратное ребро в её предка тогда и только тогда, когда найдётся такой сын to, что $fup[to] \le tin[v]$. (Если $fup[to] = tin[v]$, то это означает, что найдётся обратное ребро, приходящее точно в v; если же $fup[to] < tin[v]$, то это означает наличие обратного ребра в какого-либо предка вершины v.)

Таким образом, если для текущего ребра (v, to) (принадлежащего дереву поиска) выполняется $fup[to] > tin[v]$, то это ребро является мостом; в противном случае оно мостом не является.

Реализация

Если говорить о самой реализации, то здесь нам нужно уметь различать три случая: когда мы идём по ребру дерева поиска в глубину, когда идём по обратному ребру, и когда пытаемся пойти по ребру дерева в обратную сторону. Это, соответственно, случаи:

- $used[to] = false$ — критерий ребра дерева поиска;
- $used[to] = true \ \&\& \ to \ne parent$ — критерий обратного ребра;
- $to = parent$ — критерий прохода по ребру дерева поиска в обратную сторону.

Таким образом, для реализации этих критериев нам надо передавать в функцию поиска в глубину вершину-предка текущей вершины.

```
const int MAXN = ...;
vector<int> g[MAXN];
bool used[MAXN];
```

```
int timer, tin[MAXN], fup[MAXN];

void dfs (int v, int p = -1) {
        used[v] = true;
        tin[v] = fup[v] = timer++;
        for (size_t i=0; i<g[v].size(); ++i) {
                int to = g[v][i];
                if (to == p)  continue;
                if (used[to])
                        fup[v] = min (fup[v], tin[to]);
                else {
                        dfs (to, v);
                        fup[v] = min (fup[v], fup[to]);
                        if (fup[to] > tin[v])
                                IS_BRIDGE(v,to);
                }
        }
}

void find_bridges() {
        timer = 0;
        for (int i=0; i<n; ++i)
                used[i] = false;
        for (int i=0; i<n; ++i)
                if (!used[i])
                        dfs (i);
}
```

Здесь основная функция для вызова — это find_bridges — она производит необходимую инициализацию и запуск обхода в глубину для каждой компоненты связности графа.

При этом $\text{IS_BRIDGE}(a, b)$ — это некая функция, которая будет реагировать на то, что ребро (a, b) является мостом, например, выводить это ребро на экран.

Константе MAXN в самом начале кода следует задать значение, равное максимально возможному числу вершин во входном графе.

Стоит заметить, что эта реализация некорректно работает при наличии в графе **кратных рёбер**: она фактически не обращает внимания, кратное ли ребро или оно единственно. Разумеется, кратные рёбра не должны входить в ответ, поэтому при вызове IS_BRIDGE можно проверять дополнительно, не кратное ли ребро мы хотим добавить в ответ. Другой способ — более аккуратная работа с предками, т.е. передавать в dfs не вершину-предка, а номер ребра, по которому мы вошли в вершину (для этого надо будет дополнительно хранить номера всех рёбер).

Задачи в online judges

Список задач, в которых требуется искать мосты:

- UVA #796 "Critical Links" [сложность: низкая]
- UVA #610 "Street Directions" [сложность: средняя]

Поиск точек сочленения

Пусть дан связный неориентированный граф. **Точкой сочленения** (или точкой артикуляции, англ. "cut vertex" или "articulation point") называется такая вершина, удаление которой делает граф несвязным.

Опишем алгоритм, основанный на поиске в глубину, работающий за $O(n+m)$, где n — количество вершин, m — рёбер.

Алгоритм

Запустим обход в глубину из произвольной вершины графа; обозначим её через root. Заметим следующий **факт** (который несложно доказать):

- Пусть мы находимся в обходе в глубину, просматривая сейчас все рёбра из вершины $v \neq \text{root}$. Тогда, если текущее ребро (v, to) таково, что из вершины to и из любого её потомка в дереве обхода в глубину нет обратного ребра в или какого-либо предка вершины v, то вершина v является точкой сочленения. В противном случае, т.е. если обход в глубину просмотрел все рёбра из вершины v, и не нашёл удовлетворяющего вышеописанным условиям ребра, то вершина v не является точкой сочленения. (В самом деле, мы этим условием проверяем, нет ли другого пути из v в to)
- Рассмотрим теперь оставшийся случай: $v = \text{root}$. Тогда эта вершина является точкой сочленения тогда и только тогда, когда эта вершина имеет более одного сына в дереве обхода в глубину. (В самом деле, это означает, что, пройдя из root по произвольному ребру, мы не смогли обойти весь граф, откуда сразу следует, что root — точка сочленения).

(Ср. формулировку этого критерия с формулировкой критерия для алгоритма поиска мостов.)

Теперь осталось научиться проверять этот факт для каждой вершины эффективно. Для этого воспользуемся "временами входа в вершину", вычисляемыми алгоритмом поиска в глубину.

Итак, пусть $tin[v]$ — это время захода поиска в глубину в вершину v. Теперь введём массив $fup[v]$, который и позволит нам отвечать на вышеописанные запросы. Время $fup[v]$ равно минимуму из времени захода в саму вершину $tin[v]$, времён захода в каждую вершину p, являющуюся концом некоторого обратного ребра (v, p), а также из всех значений $fup[to]$ для каждой вершины to, являющейся непосредственным сыном v в дереве поиска:

$$fup[v] = \min \begin{cases} tin[v], & \\ tin[p], & \text{for all (v,p) — back edge} \\ fup[to], & \text{for all (v,to) — tree edge} \end{cases}$$

(здесь "back edge" — обратное ребро, "tree edge" — ребро дерева)

Тогда, из вершины v или её потомка есть обратное ребро в её предка тогда и только тогда, когда найдётся такой сын to, что $fup[to] < tin[v]$.

Таким образом, если для текущего ребра (v, to) (принадлежащего дереву поиска) выполняется $fup[to] \geq tin[v]$, то вершина v является точкой сочленения. Для начальной вершины $v = \text{root}$ критерий другой: для этой вершины надо посчитать число непосредственных сыновей в дереве обхода в глубину.

Реализация

Если говорить о самой реализации, то здесь нам нужно уметь различать три случая: когда мы идём по ребру дерева поиска в глубину, когда идём по обратному ребру, и когда пытаемся пойти по ребру дерева в обратную сторону. Это, соответственно, случаи $used[to] = false$, $used[to] = true \ \&\& \ to \neq parent$, и $to = parent$. Таким образом, нам надо передавать в функцию поиска в глубину вершину-предка текущей вершины.

```
vector<int> g[MAXN];
bool used[MAXN];
int timer, tin[MAXN], fup[MAXN];

void dfs (int v, int p = -1) {
        used[v] = true;
        tin[v] = fup[v] = timer++;
        int children = 0;
        for (size_t i=0; i<g[v].size(); ++i) {
                int to = g[v][i];
```

```
                if (to == p)  continue;
                if (used[to])
                        fup[v] = min (fup[v], tin[to]);
                else {
                        dfs (to, v);
                        fup[v] = min (fup[v], fup[to]);
                        if (fup[to] >= tin[v] && p != -1)
                                IS_CUTPOINT(v);
                        ++children;
                }
        }
        if (p == -1 && children > 1)
                IS_CUTPOINT(v);
}

int main() {
        int n;
        ... чтение n и g ...

        timer = 0;
        for (int i=0; i<n; ++i)
                used[i] = false;
        dfs (0);
}
```

Здесь константе MAXN должно быть задано значение, равное максимально возможному числу вершин во входном графе.

Функция $\mathrm{IS_CUTPOINT}(v)$ в коде — это некая функция, которая будет реагировать на то, что вершина v является точкой сочленения, например, выводить эту вершины на экран (надо учитывать, что для одной и той же вершины эта функция может быть вызвана несколько раз).

Задачи в online judges

Список задач, в которых требуется искать точки сочленения:

- UVA #10199 "Tourist Guide" [сложность: низкая]
- UVA #315 "Network" [сложность: низкая]

Поиск мостов в режиме онлайн

Пусть дан неориентированный граф. Мостом называется такое ребро, удаление которого делает граф несвязным (или, точнее, увеличивает число компонент связности). Требуется найти все мосты в заданном графе.

Неформально эта задача ставится следующим образом: требуется найти на заданной карте дорог все "важные" дороги, т.е. такие дороги, что удаление любой из них приведёт к исчезновению пути между какой-то парой городов.

Описываемый здесь алгоритм является **онлайновым**, что означает, что входной граф не является известным заранее, а рёбра в него добавляются по одному, и после каждого такого добавления алгоритм пересчитывает все мосты в текущем графе. Иными словами, алгоритм предназначен для эффективной работы на динамическом, изменяющемся графе.

Более строго, **постановка задачи** следующая. Изначально граф пустой и состоит из n вершин. Затем поступают запросы, каждый из которых — это пара вершин (a, b), которые обозначают ребро, добавляемое в граф. Требуется после каждого запроса, т.е. после добавления каждого ребра, выводить текущее количество мостов в графе. (При желании можно поддерживать и список всех рёбер-мостов, а также явно поддерживать компоненты рёберной двусвязности.)

Описываемый ниже алгоритм работает за время $O(n \log n + m)$, где m — число запросов. Алгоритм основан на структуре данных "система непересекающихся множеств".

Приведённая реализация алгоритма, впрочем, работает за время $O(n \log n + m \log n)$, поскольку использует в одном месте упрощённую версию системы непересекающихся множеств без ранговой эвристики.

Алгоритм

Известно, что рёбра-мосты разбивают вершины графа на компоненты, называемые компонентами рёберной двусвязности. Если каждую компоненту рёберной двусвязности сжать в одну вершину, и оставить только рёбра-мосты между этими компонентами, то получится ациклический граф, т.е. лес.

Описываемый ниже алгоритм поддерживает в явном виде этот **лес компонент рёберной двусвязности**.

Понятно, что изначально, когда граф пустой, он содержит n компонент рёберной двусвязности, не связанных никак между собой.

При добавлении очередного ребра (a, b) может возникнуть три ситуации:

- Оба конца a и b находятся в одной и той же компоненте рёберной двусвязности — тогда это ребро не является мостом, и ничего не меняет в структуре леса, поэтому просто пропускаем это ребро.

  Таким образом, в этом случае число мостов не меняется.

- Вершины a и b находятся в разных компонентах связности, т.е. соединяют два дерева. В этом случае ребро (a, b) становится новым мостом, а эти два дерева объединяются в одно (а все старые мосты остаются).

  Таким образом, в этом случае число мостов увеличивается на единицу.

- Вершины a и b находятся в одной компоненте связности, но в разных компонентах рёберной двусвязности. В этом случае это ребро образует цикл вместе с некоторыми из старых мостов. Все эти мосты перестают быть мостами, а образовавшийся цикл надо объединить в новую компоненту рёберной двусвязности.

  Таким образом, в этом случае число мостов уменьшается на два или более.

Следовательно, вся задача сводится к эффективной реализации всех этих операций над лесом компонент.

Структуры данных для хранения леса

Всё, что нам понадобится из структур данных, — это система непересекающихся множеств. На самом деле, нам понадобится делать два экземпляра этой структуры: одна будет для поддержания **компонент связности**, другая — для поддержания **компонент рёберной двусвязности**.

Кроме того, для хранения структуры деревьев в лесу компонент двусвязности для каждой вершины будем хранить указатель $\mathrm{par}[]$ на её предка в дереве.

Будем теперь последовательно разбирать каждую операцию, которую нам надо научиться реализовывать:

- **Проверка**, **лежат ли две указанные вершины в одной компоненте связности/двусвязности**. Делается обычным запросом к структуре "система непересекающихся множеств".
- **Соединение двух деревьев в одно** по некоторому ребру (a, b). Поскольку могло получиться, что ни вершина a, ни вершина b не являются корнями своих деревьев, то единственный способ соединить эти два дерева

— **переподвесить** одно из них. Например, можно переподвесить одно дерево за вершину a, и затем присоединить это к другому дереву, сделав вершину a дочерней к b.

Однако встаёт вопрос об эффективности операции переподвешивания: чтобы переподвесить дерево с корнем в r за вершину v, надо пройти по пути из v в r, перенаправляя указатели $\mathrm{par}[]$ в обратную сторону, а также меняя ссылки на предка в системе непересекающихся множеств, отвечающей за компоненты связности.

Таким образом, стоимость операции переподвешивания есть $O(h)$, где h — высота дерева. Можно оценить её ещё выше, сказав, что это есть величина $O(\mathrm{size})$, где size — число вершин в дереве.

Применим теперь такой стандартный приём: скажем, что из двух деревьев **переподвешивать будем то, в котором меньше вершин**. Тогда интуитивно понятно, что худший случай — когда объединяются два дерева примерно равного размера, но тогда в результате получается дерево вдвое большего размера, что не позволяет такой ситуации происходить много раз. Формально это можно записать в виде рекуррентного соотношения:

$$T(n) = \max_{k=1\ldots n-1} \{\, T(k) + T(n-k) + O(n) \,\},$$

где через $T(n)$ мы обозначили число операций, необходимое для получения дерева из n вершин с помощью операций переподвешивания и объединения деревьев. Это известное рекуррентное соотношение, и оно имеет решение $T(n) = O(n \log n)$.

Таким образом, суммарное время, затрачиваемое на всех переподвешивания, составит $O(n \log n)$, если мы всегда будем переподвешивать меньшее из двух дерево.

Нам придётся поддерживать размеры каждой компоненты связности, но структура данных "система непересекающихся множеств" позволяет делать это без труда.

- **Поиск цикла**, образуемого добавлением нового ребра (a, b) в какое-то дерево. Фактически это означает, что нам надо найти наименьшего общего предка (LCA) вершин a и b.

  Заметим, что потом мы сожмём все вершины обнаруженного цикла в одну вершину, поэтому нас устроит любой алгоритма поиска LCA, работающий за время порядка его длины.

  Поскольку вся информация о структуре дерева, которая у нас есть, — это ссылки $par[]$ на предков, то единственно возможным представляется следующий алгоритм поиска LCA: помечаем вершины a и b как посещённые, затем переходим к их предкам $\mathrm{par}[a]$ и $\mathrm{par}[b]$ и помечаем их, потом к их предкам, и так далее, пока не случится, что хотя бы одна из двух текущих вершин уже помечена. Это будет означать, что текущая вершина — и есть искомый LCA, и надо будет заново повторить путь до неё от вершины a и от вершины b — тем самым мы найдём искомый цикл.

  Очевидно, что этот алгоритм работает за время порядка длины искомого цикла, поскольку каждый из двух указателей не мог пройти расстояние, большее этой длины.

- **Сжатие цикла**, образуемого добавлением нового ребра (a, b) в какое-то дерево.

  Нам требуется создать новую компоненту рёберной двусвязности, которая будет состоять из всех вершин обнаруженного цикла (понятно, что обнаруженный цикл сам мог состоять из каких-то компонент двусвязности, но это ничего не меняет). Кроме того, надо произвести сжатие таким образом, чтобы не нарушилась структура дерева, и все указатели $\mathrm{par}[]$ и две системы непересекающихся множеств были корректными.

  Самый простой способ добиться этого — **сжать все вершины найденного цикла в их** LCA. В самом деле, вершина-LCA — это самая высокая из сжимаемых вершин, т.е. её par остаётся без изменений. Для всех остальных сжимаемых вершин обновлять тоже ничего не надо, поскольку эти вершины просто перестают существовать — в системе непересекающихся множеств для компонент двусвязности все эти вершины будут просто указывать на вершину-LCA.

  Но тогда получится, что система непересекающихся множеств для компонент двусвязности работает без эвристики объединения по рангу: если мы всегда присоединяем вершины цикла к их LCA, то этой эвристике нет места. В этом случае в асимптотике возникнет $O(\log n)$, поскольку без эвристики по рангу любая операция с системой непересекающихся множеств работает именно за такое время.

  Для достижения асимптотики $O(1)$ на один запрос необходимо объединять вершины цикла согласно ранговой эвристике, а затем присвоить par нового лидера в $\mathrm{par}[\mathrm{LCA}]$.

Реализация

Приведём здесь итоговую реализацию всего алгоритма.

В целях простоты система непересекающихся множеств для компонент двусвязности написана **без ранговой эвристики**, поэтому итоговая асимптотика составит $O(\log n)$ на запрос в среднем. (О том, как достичь асимптотики $O(1)$, написано выше в пункте "Сжатие цикла".)

Также в данной реализации не хранятся сами рёбра-мосты, а хранится только их количество — см. переменная

$\mathrm{bridges}$. Впрочем, при желании не составит никакого труда завести set из всех мостов.

Изначально следует вызвать функцию $\mathrm{init}()$, которая инициализирует две системы непересекающихся множеств (выделяя каждую вершину в отдельное множество, и проставляя размер, равный единице), проставляет предков par.

Основная функция — это $\mathrm{add_edge}(a, b)$, которая обрабатывает запрос на добавление нового ребра.

Константе MAXN следует задать значение, равное максимально возможному количеству вершин во входном графе.

Более подробные пояснения к данной реализации см. ниже.

```
const int MAXN = ...;

int n, bridges, par[MAXN], bl[MAXN], comp[MAXN], size[MAXN];


void init() {
        for (int i=0; i<n; ++i) {
                bl[i] = comp[i] = i;
                size[i] = 1;
                par[i] = -1;
        }
        bridges = 0;
}


int get (int v) {
        if (v==-1)  return -1;
        return bl[v]==v ? v : bl[v]=get(bl[v]);
}

int get_comp (int v) {
        v = get(v);
        return comp[v]==v ? v : comp[v]=get_comp(comp[v]);
}

void make_root (int v) {
        v = get(v);
        int root = v,
                child = -1;
        while (v != -1) {
                int p = get(par[v]);
                par[v] = child;
                comp[v] = root;
                child=v;  v=p;
        }
        size[root] = size[child];
}


int cu, u[MAXN];

void merge_path (int a, int b) {
        ++cu;

        vector<int> va, vb;
        int lca = -1;
        for(;;) {
                if (a != -1) {
                        a = get(a);
                        va.pb (a);

                        if (u[a] == cu) {
                                lca = a;
                                break;
                        }
                        u[a] = cu;

                        a = par[a];
                }
```

```
                if (b != -1) {
                        b = get(b);
                        vb.pb (b);

                        if (u[b] == cu) {
                                lca = b;
                                break;
                        }
                        u[b] = cu;

                        b = par[b];
                }
        }

        for (size_t i=0; i<va.size(); ++i) {
                bl[va[i]] = lca;
                if (va[i] == lca)  break;
                --bridges;
        }
        for (size_t i=0; i<vb.size(); ++i) {
                bl[vb[i]] = lca;
                if (vb[i] == lca)  break;
                --bridges;
        }
}


void add_edge (int a, int b) {
        a = get(a);   b = get(b);
        if (a == b)  return;

        int ca = get_comp(a),
                cb = get_comp(b);
        if (ca != cb) {
                ++bridges;
                if (size[ca] > size[cb]) {
                        swap (a, b);
                        swap (ca, cb);
                }
                make_root (a);
                par[a] = comp[a] = b;
                size[cb] += size[a];
        }
        else
                merge_path (a, b);
}
```

Прокомментируем код более подробно.

Система непересекающихся множеств для компонент двусвязности хранится в массиве $\mathrm{bl}[]$, а функция, возвращающая лидера компоненты двусвязности — это $\mathrm{get}(v)$. Эту функцию используется много раз в остальном коде, поскольку нужно помнить о том, что после сжатия нескольких вершин в одну все эти вершины перестают существовать, а вместо них существует только их лидер, у которого и хранятся корректные данные (предок par, предок в системе непересекающихся множеств для компонент связности, и т.д.).

Система непересекающихся множеств для компонент связности хранится в массиве $\mathrm{comp}[]$, также есть дополнительный массив $\mathrm{size}[]$ для хранения размеров компонент. Функция $\mathrm{get_comp}(v)$ возвращает лидера компоненты связности (который на самом деле является корнем дерева).

Функция переподвешивания дерева $\mathrm{make_root}(v)$ работает, как и было описано выше: она идёт от вершины v по предкам до корня, каждый раз перенаправляя предка par в обратную сторону (вниз, по направлению к вершине v). Также обновляется указатель comp в системе непересекающихся множеств для компонент связности, чтобы он указывал на новый корень. После переподвешивания у нового корня проставляется размер size компоненты связности. Обратим внимание, что при реализации мы каждый раз вызываем функцию $\mathrm{get}()$, чтобы получить доступ именно к лидеру компоненты сильной связности, а не к какой-то вершине, которая возможно уже была сжата.

Функция обнаружения и сжатия пути $\mathrm{merge_path}(a, b)$, как и было описано выше, ищет LCA вершин a и b, для чего поднимается от них параллельно вверх, пока какая-то вершина не встретится во второй раз. В целях эффективности пройденные вершины помечаются с помощью техники "числового used", что работает за $O(1)$ вместо применения set. Пройденные пути сохраняются в векторах va и vb, чтобы потом пройтись по ним

второй раз до LCA, получив тем самым все вершины цикла. Все вершины цикла сжимаются, путём присоединения их к LCA (здесь возникает асимптотика $O(\log n)$, поскольку при сжатии мы не используем ранговую эвристику). Попутно считается число пройденных рёбер, которое равно количеству мостов в обнаруженном цикле (это количество отнимается от $\mathrm{bridges}$).

Наконец, **функция обработки запросов** $\mathrm{add_edge}(a, b)$ определяет компоненты связности, в которых лежат вершины a и b, и если они лежат в разных компонентах связности, то меньшее дерево переподвешивается за новый корень и затем присоединяется к большему дереву. Иначе же, если вершины a и b лежат в одном дереве, но в разных компонентах двусвязности, то вызывается функция $\mathrm{merge_path}(a, b)$, которая обнаружит цикл и сожмёт его в одну компоненту двусвязности.

Нахождение кратчайших путей от заданной вершины до всех остальных вершин алгоритмом Дейкстры

Постановка задачи

Дан ориентированный или неориентированный взвешенный граф с n вершинами и m рёбрами. Веса всех рёбер неотрицательны. Указана некоторая стартовая вершина s. Требуется найти длины кратчайших путей из вершины s во все остальные вершины, а также предоставить способ вывода самих кратчайших путей.

Эта задача называется "задачей о кратчайших путях с единственным источником" (single-source shortest paths problem).

Алгоритм

Здесь описывается алгоритм, который предложил датский исследователь **Дейкстра** (Dijkstra) в 1959 г.

Заведём массив $d[]$, в котором для каждой вершины v будем хранить текущую длину $d[v]$ кратчайшего пути из s в v. Изначально $d[s] = 0$, а для всех остальных вершин эта длина равна бесконечности (при реализации на компьютере обычно в качестве бесконечности выбирают просто достаточно большое число, заведомо большее возможной длины пути):

$$d[v] = \infty, v \neq s$$

Кроме того, для каждой вершины v будем хранить, помечена она ещё или нет, т.е. заведём булевский массив $u[]$. Изначально все вершины не помечены, т.е.

$$u[v] = \text{false}$$

Сам алгоритм Дейкстры состоит из n **итераций**. На очередной итерации выбирается вершина v с наименьшей величиной $d[v]$ среди ещё не помеченных, т.е.:

$$d[v] = \min_{p:\ u[p]=\text{false}} d[p]$$

(Понятно, что на первой итерации выбрана будет стартовая вершина s.)

Выбранная таким образом вершина v отмечается помеченной. Далее, на текущей итерации, из вершины v производятся **релаксации**: просматриваются все рёбра (v, to), исходящие из вершины v, и для каждой такой вершины to алгоритм пытается улучшить значение $d[to]$. Пусть длина текущего ребра равна len, тогда в виде кода релаксация выглядит как:

$$d[to] = \min(d[to], d[v] + \text{len})$$

На этом текущая итерация заканчивается, алгоритм переходит к следующей итерации (снова выбирается вершина с наименьшей величиной d, из неё производятся релаксации, и т.д.). При этом в конце концов, после n итераций, все вершины графа станут помеченными, и алгоритм свою работу завершает. Утверждается, что найденные значения $d[v]$ и есть искомые длины кратчайших путей из s в v.

Стоит заметить, что, если не все вершины графа достижимы из вершины s, то значения $d[v]$ для них так и останутся бесконечными. Понятно, что несколько последних итераций алгоритма будут как раз выбирать эти вершины, но никакой полезной работы производить эти итерации не будут (поскольку бесконечное расстояние не сможет прорелаксировать другие, даже тоже бесконечные расстояния). Поэтому алгоритм можно сразу останавливать, как только в качестве выбранной вершины берётся вершина с бесконечным расстоянием.

Восстановление путей. Разумеется, обычно нужно знать не только длины кратчайших путей, но и получить сами пути. Покажем, как сохранить информацию, достаточную для последующего восстановления кратчайшего пути из s до любой вершины. Для этого достаточно так называемого **массива предков**: массива $p[]$, в котором для каждой вершины $v \neq s$ хранится номер вершины $p[v]$, являющейся предпоследней в кратчайшем пути до вершины v. Здесь используется тот факт, что если мы возьмём кратчайший путь до какой-то вершины v, а затем удалим из этого пути последнюю вершину, то получится путь, оканчивающийся некоторой вершиной $p[v]$, и этот путь будет кратчайшим для вершины $p[v]$. Итак, если мы будем обладать этим массивом предков, то кратчайший путь можно будет восстановить по нему, просто каждый раз беря предка от текущей вершины, пока мы не придём в стартовую вершину s — так мы получим искомый кратчайший путь, но записанный в обратном порядке. Итак,

кратчайший путь P до вершины v равен:

$$P = (s, \dots, p[p[p[v]]], p[p[v]], p[v], v)$$

Осталось понять, как строить этот массив предков. Однако это делается очень просто: при каждой успешной релаксации, т.е. когда из выбранной вершины v происходит улучшение расстояния до некоторой вершины to, мы записываем, что предком вершины to является вершина v:

$$p[to] = v$$

Доказательство

Основное утверждение, на котором основана корректность алгоритма Дейкстры, следующее. Утверждается, что после того как какая-либо вершина v становится помеченной, текущее расстояние до неё $d[v]$ уже является кратчайшим, и, соответственно, больше меняться не будет.

Доказательство будем производить по индукции. Для первой итерации справедливость его очевидна — для вершины s имеем $d[s] = 0$, что и является длиной кратчайшего пути до неё. Пусть теперь это утверждение выполнено для всех предыдущих итераций, т.е. всех уже помеченных вершин; докажем, что оно не нарушается после выполнения текущей итерации. Пусть v — вершина, выбранная на текущей итерации, т.е. вершина, которую алгоритм собирается пометить. Докажем, что $d[v]$ действительно равно длине кратчайшего пути до неё (обозначим эту длину через $l[v]$).

Рассмотрим кратчайший путь P до вершины v. Понятно, этот путь можно разбить на два пути: P_1, состоящий только из помеченных вершин (как минимум стартовая вершина s будет в этом пути), и остальная часть пути P_2 (она тоже может включать помеченные вершины, но начинается обязательно с непомеченной). Обозначим через p первую вершину пути P_2, а через q — последнюю вершины пути P_1.

Докажем сначала наше утверждение для вершины p, т.е. докажем равенство $d[p] = l[p]$. Однако это практически очевидно: ведь на одной из предыдущих итераций мы выбирали вершину q и выполняли релаксацию из неё. Поскольку (в силу самого выбора вершины p) кратчайший путь до p равен кратчайшему пути до q плюс ребро (p, q), то при выполнении релаксации из q величина $d[p]$ действительно установится в требуемое значение.

Вследствие неотрицательности стоимостей рёбер длина кратчайшего пути $l[p]$ (а она по только что доказанному равна $d[p]$) не превосходит длины $l[v]$ кратчайшего пути до вершины v. Учитывая, что $l[v] \le d[v]$ (ведь алгоритм Дейкстры не мог найти более короткого пути, чем это вообще возможно), в итоге получаем соотношения:

$$d[p] = l[p] \le l[v] \le d[v]$$

С другой стороны, поскольку и p, и v — вершины непомеченные, то так как на текущей итерации была выбрана именно вершина v, а не вершина p, то получаем другое неравенство:

$$d[p] \ge d[v]$$

Из этих двух неравенств заключаем равенство $d[p] = d[v]$, а тогда из найденных до этого соотношений получаем и:

$$d[v] = l[v]$$

что и требовалось доказать.

Реализация

Итак, алгоритм Дейкстры представляет собой n итераций, на каждой из которых выбирается непомеченная вершина с наименьшей величиной $d[v]$, эта вершина помечается, и затем просматриваются все рёбра, исходящие из данной вершины, и вдоль каждого ребра делается попытка улучшить значение $d[]$ на другом конце ребра.

Время работы алгоритма складывается из:

- n раз поиск вершины с наименьшей величиной $d[v]$ среди всех непомеченных вершин, т.е. среди $O(n)$ вершин
- m раз производится попытка релаксаций

При простейшей реализации этих операций на поиск вершины будет затрачиваться $O(n)$ операций, а на одну релаксацию — $O(1)$ операций, и итоговая **асимптотика** алгоритма составляет:

$$O(n^2 + m)$$

Реализация:

```
const int INF = 1000000000;

int main() {
        int n;
        ... чтение n ...
        vector < vector < pair<int,int> > > g (n);
        ... чтение графа ...
        int s = ...; // стартовая вершина

        vector<int> d (n, INF),  p (n);
        d[s] = 0;
        vector<char> u (n);
        for (int i=0; i<n; ++i) {
                int v = -1;
                for (int j=0; j<n; ++j)
                        if (!u[j] && (v == -1 || d[j] < d[v]))
                                v = j;
                if (d[v] == INF)
                        break;
                u[v] = true;

                for (size_t j=0; j<g[v].size(); ++j) {
                        int to = g[v][j].first,
                                len = g[v][j].second;
                        if (d[v] + len < d[to]) {
                                d[to] = d[v] + len;
                                p[to] = v;
                        }
                }
        }
}
```

Здесь граф g хранится в виде списков смежности: для каждой вершины v список $g[v]$ содержит список рёбер, исходящих из этой вершины, т.е. список пар $\mathrm{pair} < \mathrm{int}, \mathrm{int} >$, где первый элемент пары — вершина, в которую ведёт ребро, а второй элемент — вес ребра.

После чтения заводятся массивы расстояний $d[]$, меток $u[]$ и предков $p[]$. Затем выполняются n итераций. На каждой итерации сначала находится вершина v, имеющая наименьшее расстояние $d[]$ среди непомеченных вершин. Если расстояние до выбранной вершины v оказывается равным бесконечности, то алгоритм останавливается. Иначе вершина помечается как помеченная, и просматриваются все рёбра, исходящие из данной вершины, и вдоль каждого ребра выполняются релаксации. Если релаксация успешна (т.е. расстояние $d[to]$ меняется), то пересчитывается расстояние $d[to]$ и сохраняется предок $p[]$.

После выполнения всех итераций в массиве $d[]$ оказываются длины кратчайших путей до всех вершин, а в массиве $p[]$ — предки всех вершин (кроме стартовой s). Восстановить путь до любой вершины t можно следующим образом:

```
vector<int> path;
for (int v=t; v!=s; v=p[v])
        path.push_back (v);
path.push_back (s);
reverse (path.begin(), path.end());
```

Литература

- Томас Кормен, Чарльз Лейзерсон, Рональд Ривест, Клиффорд Штайн. **Алгоритмы**: **Построение и анализ** [2005]
- Edsger Dijkstra. A note on two problems in connexion with graphs [1959]

Нахождение кратчайших путей от заданной вершины до всех остальных вершин алгоритмом Дейкстры для разреженных графов

Постановку задачи, алгоритм и его доказательство см. в статье об общем алгоритме Дейкстры.

Алгоритм

Напомним, что сложность алгоритма Дейкстры складывается из двух основных операций: время нахождения вершины с наименьшей величиной расстояния $d[v]$, и время совершения релаксации, т.е. время изменения величины $d[to]$.

При простейшей реализации эти операции потребуют соответственно $O(n)$ и $O(1)$ времени. Учитывая, что первая операция всего выполняется $O(n)$ раз, а вторая — $O(m)$, получаем асимптотику простейшей реализации алгоритма Дейкстры: $O(n^2 + m)$.

Понятно, что эта асимптотика является оптимальной для плотных графов, т.е. когда $m \approx n^2$. Чем более разрежен граф (т.е. чем меньше m по сравнению с максимальным количество рёбер n^2), тем менее оптимальной становится эта оценка, и по вине первого слагаемого. Таким образом, надо улучшать время выполнения операций первого типа, не сильно ухудшая при этом время выполнения операций второго типа.

Для этого надо использовать различные вспомогательные структуры данных. Наиболее привлекательными являются **Фибоначчиевы кучи**, которые позволяют производить операцию первого вида за $O(\log n)$, а второго — за $O(1)$. Поэтому при использовании Фибоначчиевых куч время работы алгоритма Дейкстры составит $O(n \log n + m)$, что является практически теоретическим минимумом для алгоритма поиска кратчайшего пути. Кстати говоря, эта оценка является оптимальной для алгоритмов, основанных на алгоритме Дейкстры, т.е. Фибоначчиевы кучи являются оптимальными с этой точки зрения (это утверждение об оптимальности на самом деле основано на невозможности существования такой "идеальной" структуры данных — если бы она существовала, то можно было бы выполнять сортировку за линейное время, что, как известно, в общем случае невозможно; впрочем, интересно, что существует алгоритм Торупа (Thorup), который ищет кратчайший путь с оптимальной, линейной, асимптотикой, но основан он на совсем другой идее, чем алгоритм Дейкстры, поэтому никакого противоречия здесь нет). Однако, Фибоначчиевы кучи довольно сложны в реализации (и, надо отметить, имеют немалую константу, скрытую в асимптотике).

В качестве компромисса можно использовать структуры данных, позволяющие выполнять **оба типа операций** (фактически, это извлечение минимума и обновление элемента) за $O(\log n)$. Тогда время работы алгоритма Дейкстры составит:

$$O(n \log n + m \log n) = O(m \log n)$$

В качестве такой структуры данных программистам на C++ удобно взять стандартный контейнер set или priority_queue. Первый основан на красно-чёрном дереве, второй — на бинарной куче. Поэтому priority_queue имеет меньшую константу, скрытую в асимпотике, однако у него есть и недостаток: он не поддерживает операцию удаления элемента, из-за чего приходится делать "обходной манёвр", который фактически приводит к замене в асимптотике $\log n$ на $\log m$ (с точки зрения асимптотики это на самом деле ничего не меняет, но скрытую константу увеличивает).

Реализация

set

Начнём с контейнера set. Поскольку в контейнере нам надо хранить вершины, упорядоченные по их величинам $d[]$, то удобно в контейнер помещать пары: первый элемент пары — расстояние, а второй — номер вершины. В результате в set будут храниться пары, автоматически упорядоченные по расстояниям, что нам и нужно.

```
const int INF = 1000000000;

int main() {
        int n;
        ... чтение n ...
```

```
        vector < vector < pair<int,int> > > g (n);
        ... чтение графа ...
        int s = ...; // стартовая вершина

        vector<int> d (n, INF),  p (n);
        d[s] = 0;
        set < pair<int,int> > q;
        q.insert (make_pair (d[s], s));
        while (!q.empty()) {
                int v = q.begin()->second;
                q.erase (q.begin());

                for (size_t j=0; j<g[v].size(); ++j) {
                        int to = g[v][j].first,
                                len = g[v][j].second;
                        if (d[v] + len < d[to]) {
                                q.erase (make_pair (d[to], to));
                                d[to] = d[v] + len;
                                p[to] = v;
                                q.insert (make_pair (d[to], to));
                        }
                }
        }
}
```

В отличие от обычного алгоритма Дейкстры, становится ненужным массив $u[]$. Его роль, как и функцию нахождения вершины с наименьшим расстоянием, выполняет set. Изначально в него помещаем стартовую вершину s с её расстоянием. Основной цикл алгоритма выполняется, пока в очереди есть хоть одна вершина. Из очереди извлекается вершина с наименьшим расстоянием, и затем из неё выполняются релаксации. Перед выполнением каждой успешной релаксации мы сначала удаляем из set старую пару, а затем, после выполнения релаксации, добавляем обратно новую пару (с новым расстоянием $d[to]$).

priority_queue

Принципиально здесь отличий от set нет, за исключением того момента, что удалять из $\mathrm{priority_queue}$ произвольные элементы невозможно (хотя теоретически кучи поддерживают такую операцию, в стандартной библиотеке она не реализована). Поэтому приходится совершать "обходной манёвр": при релаксации просто не будем удалять старые пары из очереди. В результате в очереди могут находиться одновременно несколько пар для одной и той же вершины (но с разными расстояниями). Среди этих пар нас интересует только одна, для которой элемент first равен $d[v]$, а все остальные являются фиктивными. Поэтому надо сделать небольшую модификацию: в начале каждой итерации, когда мы извлекаем из очереди очередную пару, будем проверять, фиктивная она или нет (для этого достаточно сравнить first и $d[v]$). Следует отметить, что это важная модификация: если не сделать её, то это приведёт к значительному ухудшению асимптотики (до $O(nm)$).

Ещё нужно помнить о том, что $\mathrm{priority_queue}$ упорядочивает элементы по убыванию, а не по возрастанию, как обычно. Проще всего преодолеть эту особенность не указанием своего оператора сравнения, а просто помещая в качестве элементов first расстояния со знаком минус. В результате в корне кучи будут оказываться элементы с наименьшим расстоянием, что нам и нужно.

```
const int INF = 1000000000;

int main() {
        int n;
        ... чтение n ...
        vector < vector < pair<int,int> > > g (n);
        ... чтение графа ...
        int s = ...; // стартовая вершина

        vector<int> d (n, INF),  p (n);
        d[s] = 0;
        priority_queue < pair<int,int> > q;
        q.push (make_pair (0, s));
        while (!q.empty()) {
                int v = q.top().second,  cur_d = -q.top().first;
                q.pop();
                if (cur_d > d[v])  continue;

                for (size_t j=0; j<g[v].size(); ++j) {
                        int to = g[v][j].first,
```

```
                                len = g[v][j].second;
                        if (d[v] + len < d[to]) {
                                d[to] = d[v] + len;
                                p[to] = v;
                                q.push (make_pair (-d[to], to));
                        }
                }
        }
}
```

Как правило, на практике версия с $\mathrm{priority_queue}$ оказывается несколько быстрее версии с set.

Избавление от pair

Можно ещё немного улучшить производительность, если в контейнерах всё же хранить не пары, а только номера вершин. При этом, понятно, надо перегрузить оператор сравнения для вершин: сравнивать две вершины надо по расстояниям до них $d[]$.

Поскольку в результате релаксации величина расстояния до какой-то вершины меняется, то надо понимать, что "сама по себе" структура данных не перестроится. Поэтому, хотя может показаться, что удалять/добавлять элементы в контейнер в процессе релаксации не надо, это приведёт к разрушению структуры данных. По-прежнему перед релаксацией надо удалить из структуры данных вершину to, а после релаксации вставить её обратно — тогда никакие соотношения между элементами структуры данных не нарушатся.

А поскольку удалять элементы можно из set, но нельзя из $\mathrm{priority_queue}$, то получается, что этот приём применим только к set. На практике он заметно увеличивает производительность, особенно когда для хранения расстояний используются большие типы данных (как $\mathrm{long\ long}$ или double).

Алгоритм Форда-Беллмана

Пусть дан ориентированный взвешенный граф G с n вершинами и m рёбрами, и указана некоторая вершина v. Требуется найти **длины кратчайших путей** от вершины v до всех остальных вершин.

В отличие от алгоритма Дейкстры, этот алгоритм применим также и к графам, содержащим рёбра отрицательного веса. Впрочем, если граф содержит отрицательный цикл, то, понятно, кратчайшего пути до некоторых вершин может не существовать (по причине того, что вес кратчайшего пути должен быть равен минус бесконечности); впрочем, этот алгоритм можно модифицировать, чтобы он сигнализировал о наличии цикла отрицательного веса, или даже выводил сам этот цикл.

Алгоритм носит имя двух американских учёных: Ричарда **Беллмана** (Richard Bellman) и Лестера **Форда** (Lester Ford). Форд фактически изобрёл этот алгоритм в 1956 г. при изучении другой математической задачи, подзадача которой свелась к поиску кратчайшего пути в графе, и Форд дал набросок решающего эту задачу алгоритма. Беллман в 1958 г. опубликовал статью, посвящённую конкретно задаче нахождения кратчайшего пути, и в этой статье он чётко сформулировал алгоритм в том виде, в котором он известен нам сейчас.

Описание алгоритма

Мы считаем, что граф не содержит цикла отрицательного веса. Случай наличия отрицательного цикла будет рассмотрен ниже в отдельном разделе.

Заведём массив расстояний $d[0 \ldots n-1]$, который после отработки алгоритма будет содержать ответ на задачу. В начале работы мы заполняем его следующим образом: $d[v] = 0$, а все остальные элементы $d[]$ равны бесконечности ∞.

Сам алгоритм Форда-Беллмана представляет из себя несколько фаз. На каждой фазе просматриваются все рёбра графа, и алгоритм пытается произвести **релаксацию** (relax, ослабление) вдоль каждого ребра (a, b) стоимости c. Релаксация вдоль ребра — это попытка улучшить значение $d[b]$ значением $d[a] + c$. Фактически это значит, что мы пытаемся улучшить ответ для вершины b, пользуясь ребром (a, b) и текущим ответом для вершины a.

Утверждается, что достаточно $n-1$ фазы алгоритма, чтобы корректно посчитать длины всех кратчайших путей в графе (повторимся, мы считаем, что циклы отрицательного веса отсутствуют). Для недостижимых вершин расстояние $d[]$ останется равным бесконечности ∞.

Реализация

Для алгоритма Форда-Беллмана, в отличие от многих других графовых алгоритмов, более удобно представлять граф в виде одного списка всех рёбер (а не n списков рёбер — рёбер из каждой вершины). В приведённой реализации заводится структура данных edge для ребра. Входными данными для алгоритма являются числа n, m, список e рёбер, и номер стартовой вершины v. Все номера вершин нумеруются с 0 по $n-1$.

Простейшая реализация

Константа INF обозначает число "бесконечность" — её надо подобрать таким образом, чтобы она заведомо превосходила все возможные длины путей.

```
struct edge {
        int a, b, cost;
};

int n, m, v;
vector<edge> e;
const int INF = 1000000000;

void solve() {
        vector<int> d (n, INF);
        d[v] = 0;
        for (int i=0; i<n-1; ++i)
                for (int j=0; j<m; ++j)
                        if (d[e[j].a] < INF)
                                d[e[j].b] = min (d[e[j].b], d[e[j].a] + e
[j].cost);
```

```
        // вывод d, например, на экран
}
```

Проверка "if (d[e[j].a] < INF)" нужна, только если граф содержит рёбра отрицательного веса: без такой проверки бы происходили релаксации из вершин, до которых пути ещё не нашли, и появлялись бы некорректные расстояния вида $\infty - 1, \infty - 2$, и т.д.

Улучшенная реализация

Этот алгоритм можно несколько ускорить: зачастую ответ находится уже за несколько фаз, а оставшиеся фазы никакой полезной работы не происходит, лишь впустую просматриваются все рёбра. Поэтому будем хранить флаг того, изменилось что-то на текущей фазе или нет, и если на какой-то фазе ничего не произошло, то алгоритм можно останавливать. (Эта оптимизация не улучшает асимптотику, т.е. на некоторых графах по-прежнему будут нужны все $n - 1$ фаза, но значительно ускоряет поведение алгоритма "в среднем", т.е. на случайных графах.)

С такой оптимизацией становится вообще ненужным ограничивать вручную число фаз алгоритма числом $n - 1$ — он сам остановится через нужное число фаз.

```
void solve() {
        vector<int> d (n, INF);
        d[v] = 0;
        for (;;) {
                bool any = false;
                for (int j=0; j<m; ++j)
                        if (d[e[j].a] < INF)
                                if (d[e[j].b] > d[e[j].a] + e[j].cost) {
                                        d[e[j].b] = d[e[j].a] + e[j].cost;
                                        any = true;
                                }
                if (!any)  break;
        }
        // вывод d, например, на экран
}
```

Восстановление путей

Рассмотрим теперь, как можно модифицировать алгоритм Форда-Беллмана так, чтобы он не только находил длины кратчайших путей, но и позволял восстанавливать сами кратчайшие пути.

Для этого заведём ещё один массив $p[0 \dots n - 1]$, в котором для каждой вершины будем хранить её "предка", т. е. предпоследнюю вершину в кратчайшем пути, ведущем в неё. В самом деле, кратчайший путь до какой-то вершины a является кратчайшим путём до какой-то вершины $p[a]$, к которому приписали в конец вершину a.

Заметим, что алгоритм Форда-Беллмана работает по такой же логике: он, предполагая, что кратчайшее расстояние до одной вершины уже посчитано, пытается улучшить кратчайшее расстояние до другой вершины. Следовательно, в момент улучшения нам надо просто запоминать в $p[]$, из какой вершины это улучшение произошло.

Приведём реализацию Форда-Беллмана с восстановлением пути до какой-то заданной вершины t:

```
void solve() {
        vector<int> d (n, INF);
        d[v] = 0;
        vector<int> p (n, -1);
        for (;;) {
                bool any = false;
                for (int j=0; j<m; ++j)
                        if (d[e[j].a] < INF)
                                if (d[e[j].b] > d[e[j].a] + e[j].cost) {
                                        d[e[j].b] = d[e[j].a] + e[j].cost;
                                        p[e[j].b] = e[j].a;
                                        any = true;
                                }
                if (!any)  break;
        }

        if (d[t] == INF)
                cout << "No path from " << v << " to " << t << ".";
        else {
                vector<int> path;
```

```
                for (int cur=t; cur!=-1; cur=p[cur])
                        path.push_back (cur);
                reverse (path.begin(), path.end());

                cout << "Path from " << v << " to " << t << ": ";
                for (size_t i=0; i<path.size(); ++i)
                        cout << path[i] << ' ';
        }
}
```

Здесь мы сначала проходимся по предкам, начиная с вершины t, и сохраняем весь пройденный путь в списке path. В этом списке получается кратчайший путь от v до t, но в обратном порядке, поэтому мы вызываем $\mathrm{reverse}$ от него и затем выводим.

Доказательство алгоритма

Во-первых, сразу заметим, что для недостижимых из v вершин алгоритм отработает корректно: для них метка $d[]$ так и останется равной бесконечности (т.к. алгоритм Форда-Беллмана найдёт какие-то пути до всех достижимых из s вершин, а релаксация во всех остальных вершинах не произойдёт ни разу).

Докажем теперь следующее **утверждение**: после выполнения i фаз алгоритм Форда-Беллмана корректно находит все кратчайшие пути, длина которых (по числу рёбер) не превосходит i.

Иными словами, для любой вершины a обозначим через k число рёбер в кратчайшем пути до неё (если таких путей несколько, можно взять любой). Тогда это утверждение говорит о том, что после k фаз этот кратчайший путь будет найден гарантированно.

Доказательство. Рассмотрим произвольную вершину a, до которой существует путь из стартовой вершины v, и рассмотрим кратчайший путь до неё: $(p_0 = v, p_1, \ldots, p_k = a)$. Перед первой фазой кратчайший путь до вершины $p_0 = v$ найден корректно. Во время первой фазы ребро (p_0, p_1) было просмотрено алгоритмом Форда-Беллмана, следовательно, расстояние до вершины p_1 было корректно посчитано после первой фазы. Повторяя эти утверждения k раз, получаем, что после k-й фазы расстояние до вершины $p_k = a$ посчитано корректно, что и требовалось доказать.

Последнее, что надо заметить — это то, что любой кратчайший путь не может иметь более $n - 1$ ребра. Следовательно, алгоритму достаточно произвести только $n - 1$ фазу. После этого ни одна релаксация гарантированно не может завершиться улучшением расстояния до какой-то вершины.

Случай отрицательного цикла

Выше мы везде считали, что отрицательного цикла в графе нет (уточним, нас интересует отрицательный цикл, достижимый из стартовой вершины v, а недостижимые циклы ничего в вышеописанном алгоритме не меняют). При его наличии возникают дополнительные сложности, связанные с тем, что расстояния до всех вершин на этом цикле, а также расстояния до достижимых из этого цикла вершин не определены — они должны быть равны минус бесконечности.

Нетрудно понять, что алгоритм Форда-Беллмана сможет **бесконечно делать релаксации** среди всех вершин этого цикла и вершин, достижимых из него. Следовательно, если не ограничивать число фаз числом $n - 1$, то алгоритм будет работать бесконечно, постоянно улучшая расстояния до этих вершин.

Отсюда мы получаем **критерий наличия достижимого цикла отрицательного веса**: если после $n - 1$ фазы мы выполним ещё одну фазу, и на ней произойдёт хотя бы одна релаксация, то граф содержит цикл отрицательного веса, достижимый из v; в противном случае, такого цикла нет.

Более того, если такой цикл обнаружился, то алгоритм Форда-Беллмана можно модифицировать таким образом, чтобы он выводил сам этот цикл в виде последовательности вершин, входящих в него. Для этого достаточно запомнить номер вершины x, в которой произошла релаксация на n-ой фазе. Эта вершина будет либо лежать на цикле отрицательного веса, либо она достижима из него. Чтобы получить вершину, которая гарантированно лежит на цикле, достаточно, например, n раз пройти по предкам, начиная от вершины x. Получив номер y вершины, лежащей на цикле, надо пройтись от этой вершины по предкам, пока мы не вернёмся в эту же вершину y (а это обязательно произойдёт, потому что релаксации в цикле отрицательного веса происходят по кругу).

Реализация:

```
void solve() {
        vector<int> d (n, INF);
        d[v] = 0;
        vector<int> p (n, -1);
        int x;
        for (int i=0; i<n; ++i) {
```

```
                x = -1;
                for (int j=0; j<m; ++j)
                        if (d[e[j].a] < INF)
                                if (d[e[j].b] > d[e[j].a] + e[j].cost) {
                                        d[e[j].b] = max (-INF, d[e[j].a] + e
[j].cost);
                                        p[e[j].b] = e[j].a;
                                        x = e[j].b;
                                }
        }

        if (x == -1)
                cout << "No negative cycle from " << v;
        else {
                int y = x;
                for (int i=0; i<n; ++i)
                        y = p[y];

                vector<int> path;
                for (int cur=y; ; cur=p[cur]) {
                        path.push_back (cur);
                        if (cur == y && path.size() > 1)  break;
                }
                reverse (path.begin(), path.end());

                cout << "Negative cycle: ";
                for (size_t i=0; i<path.size(); ++i)
                        cout << path[i] << ' ';
        }
}
```

Поскольку при наличии отрицательного цикла за n итераций алгоритма расстояния могли уйти далеко в минус (по всей видимости, до отрицательных чисел порядка -2^n), в коде приняты дополнительные меры против такого целочисленного переполнения:

```
d[e[j].b] = max (-INF, d[e[j].a] + e[j].cost);
```

В приведённой выше реализации ищется отрицательный цикл, достижимый из некоторой стартовой вершины v; однако алгоритм можно модифицировать, чтобы он искал просто **любой отрицательный цикл** в графе. Для этого надо положить все расстояния $d[i]$ равными нулю, а не бесконечности — так, как будто бы мы ищем кратчайший путь изо всех вершин одновременно; на корректность обнаружения отрицательного цикла это не повлияет.

Дополнительно на тему этой задачи — см. отдельную статью "**Поиск отрицательного цикла в графе**".

Задачи в online judges

Список задач, которые можно решить с помощью алгоритма Форда-Беллмана:

- E-OLIMP #1453 "**Форд-Беллман**" [сложность: низкая]
- UVA #423 "MPI Maelstrom" [сложность: низкая]
- UVA #534 "Frogger" [сложность: средняя]
- UVA #10099 "The Tourist Guide" [сложность: средняя]
- UVA #515 "King" [сложность: средняя]

См. также список задач в статье "**Поиск отрицательного цикла**".

Алгоритм Левита нахождения кратчайших путей от заданной вершины до всех остальных вершин

Пусть дан граф с N вершинами и M ребрами, для каждого из которых указан его вес L_i. Также дана стартовая вершина V_0. Требуется найти кратчайшие пути от вершины V_0 до всех остальных вершин.

Алгоритм Левита решает эту задачу весьма эффективно (по поводу асимптотики и скорости работы см. ниже).

Описание

Пусть массив D[1..N] будет содержать текущие кратчайшие длины путей, т.е. D_i - это текущая длина кратчайшего пути от вершины V_0 до вершины i. Изначально массив D заполнен значениями "бесконечность", кроме $D_{V_0} = 0$. По окончании работы алгоритма этот массив будет содержать окончательные кратчайшие расстояния.

Пусть массив P[1..N] содержит текущих предков, т.е. P_i - это вершина, предшествующая вершине i в кратчайшем пути от вершины V_0 до i. Так же как и массив D, массив P изменяется постепенно по ходу алгоритма и к концу его принимает окончательные значения.

Теперь собственно сам алгоритм Левита. На каждом шаге поддерживается три множества вершин:

- M_0 - вершины, расстояние до которых уже вычислено (но, возможно, не окончательно);
- M_1 - вершины, расстояние до которых вычисляется;
- M_2 - вершины, расстояние до которых ещё не вычислено.

Вершины в множестве M_1 хранятся в виде двунаправленной очереди (deque).

Изначально все вершины помещаются в множество M_2, кроме вершины V_0, которая помещается в множество M_1.

На каждом шаге алгоритма мы берём вершину из множества M_1 (достаём верхний элемент из очереди). Пусть V - это выбранная вершина. Переводим эту вершину во множество M_0. Затем просматриваем все рёбра, выходящие из этой вершины. Пусть T - это второй конец текущего ребра (т.е. не равный V), а L - это длина текущего ребра.

- Если T принадлежит M_2, то T переносим во множество M_1 в конец очереди. D_T полагаем равным $D_V + L$.
- Если T принадлежит M_1, то пытаемся улучшить значение D_T: $D_T = \min(D_T, D_V + L)$. Сама вершина T никак не передвигается в очереди.
- Если T принадлежит M_0, и если D_T можно улучшить ($D_T > D_V + L$), то улучшаем D_T, а вершину T возвращаем в множество M_1, помещая её в начало очереди.

Разумеется, при каждом обновлении массива D следует обновлять и значение в массиве P.

Подробности реализации

Создадим массив ID[1..N], в котором для каждой вершины будем хранить, какому множеству она принадлежит: 0 - если M_2 (т.е. расстояние равно бесконечности), 1 - если M_1 (т.е. вершина находится в очереди), и 2 - если M_0 (некоторый путь уже был найден, расстояние меньше бесконечности).

Очередь обработки можно реализовать стандартной структурой данных deque. Однако есть более эффективный способ. Во-первых, очевидно, в очереди в любой момент времени будет храниться максимум N элементов. Но, во-вторых, мы можем добавлять элементы и в начало, и в конец очереди. Следовательно, мы можем организовать очередь на массиве размера N, однако нужно зациклить его. Т.е. делаем массив Q[1..N], указатели (int) на первый элемент QH и на элемент после последнего QT. Очередь пуста, когда QH == QT. Добавление в конец - просто запись в Q[QT] и увеличение QT на 1; если QT после этого вышел за пределы очереди (QT == N), то делаем QT = 0. Добавление в начало очереди - уменьшаем QH на 1, если она вышла за пределы очереди (QH == -1), то делаем QH = N-1.

Сам алгоритм реализуем в точности по описанию выше.

Асимптотика

Мне не известна более-менее хорошая асимптотическая оценка этого алгоритма. Я встречал только оценку O (N M) у похожего алгоритма.

Однако на практике алгоритма зарекомендовал себя очень хорошо: время его работы я оцениваю как O (M log N), хотя, повторюсь, это исключительно **экспериментальная** оценка.

Реализация

```
typedef pair<int,int> rib;
typedef vector < vector<rib> > graph;

const int inf = 1000*1000*1000;


int main()
{
        int n, v1, v2;
        graph g (n);

        ... чтение графа ...

        vector<int> d (n, inf);
        d[v1] = 0;
        vector<int> id (n);
        deque<int> q;
        q.push_back (v1);
        vector<int> p (n, -1);

        while (!q.empty())
        {
                int v = q.front(),  q.pop_front();
                id[v] = 1;
                for (size_t i=0; i<g[v].size(); ++i)
                {
                        int to = g[v][i].first, len = g[v][i].second;
                        if (d[to] > d[v] + len)
                        {
                                d[to] = d[v] + len;
                                if (id[to] == 0)
                                        q.push_back (to);
                                else if (id[to] == 1)
                                        q.push_front (to);
                                p[to] = v;
                                id[to] = 1;
                        }
                }
        }

        ... вывод результата ...

}
```

Алгоритм Флойда-Уоршелла нахождения кратчайших путей между всеми парами вершин

Дан ориентированный или неориентированный взвешенный граф G с n вершинами. Требуется найти значения всех величин d_{ij} — длины кратчайшего пути из вершины i в вершину j.

Предполагается, что граф не содержит циклов отрицательного веса (тогда ответа между некоторыми парами вершин может просто не существовать — он будет бесконечно маленьким).

Этот алгоритм был одновременно опубликован в статьях Роберта Флойда (Robert Floyd) и Стивена Уоршелла (Варшалла) (Stephen Warshall) в 1962 г., по имени которых этот алгоритм и называется в настоящее время. Впрочем, в 1959 г. Бернард Рой (Bernard Roy) опубликовал практически такой же алгоритм, но его публикация осталась незамеченной.

Описание алгоритма

Ключевая идея алгоритма — разбиение процесса поиска кратчайших путей на **фазы**.

Перед k-ой фазой ($k = 1 \dots n$) считается, что в матрице расстояний $d[][]$ сохранены длины таких кратчайших путей, которые содержат в качестве внутренних вершин только вершины из множества $\{1, 2, \dots, k-1\}$ (вершины графа мы нумеруем, начиная с единицы).

Иными словами, перед k-ой фазой величина $d[i][j]$ равна длине кратчайшего пути из вершины i в вершину j, если этому пути разрешается заходить только в вершины с номерами, меньшими k (начало и конец пути не считаются).

Легко убедиться, что чтобы это свойство выполнилось для первой фазы, достаточно в матрицу расстояний $d[][]$ записать матрицу смежности графа: $d[i][j] = g[i][j]$ — стоимости ребра из вершины i в вершину j. При этом, если между какими-то вершинами ребра нет, то записать следует величину "бесконечность" ∞. Из вершины в саму себя всегда следует записывать величину 0, это критично для алгоритма.

Пусть теперь мы находимся на k-ой фазе, и хотим **пересчитать** матрицу $d[][]$ таким образом, чтобы она соответствовала требованиям уже для $k+1$-ой фазы. Зафиксируем какие-то вершины i и j. У нас возникает два принципиально разных случая:

- Кратчайший путь из вершины i в вершину j, которому разрешено дополнительно проходить через вершины $\{1, 2, \dots, k\}$, **совпадает** с кратчайшим путём, которому разрешено проходить через вершины множества $\{1, 2, \dots, k-1\}$.

  В этом случае величина $d[i][j]$ не изменится при переходе с k-ой на $k+1$-ую фазу.

- "Новый" кратчайший путь стал **лучше** "старого" пути.

  Это означает, что "новый" кратчайший путь проходит через вершину k. Сразу отметим, что мы не потеряем общности, рассматривая далее только простые пути (т.е. пути, не проходящие по какой-то вершине дважды).

  Тогда заметим, что если мы разобьём этот "новый" путь вершиной k на две половинки (одна идущая $i \Rightarrow k$, а другая — $k \Rightarrow j$), то каждая из этих половинок уже не заходит в вершину k. Но тогда получается, что длина каждой из этих половинок была посчитана ещё на $k-1$-ой фазе или ещё раньше, и нам достаточно взять просто сумму $d[i][k] + d[k][j]$, она и даст длину "нового" кратчайшего пути.

Объединяя эти два случая, получаем, что на k-ой фазе требуется пересчитать длины кратчайших путей между всеми парами вершин i и j следующим образом:

```
new_d[i][j] = min (d[i][j], d[i][k] + d[k][j]);
```

Таким образом, вся работа, которую требуется произвести на k-ой фазе — это перебрать все пары вершин и пересчитать длину кратчайшего пути между ними. В результате после выполнения n-ой фазы в матрице расстояний $d[i][j]$ будет записана длина кратчайшего пути между i и j, либо ∞, если пути между этими вершинами не существует.

Последнее замечание, которое следует сделать, — то, что можно **не создавать отдельную матрицу** $\text{new_d}[][]$ для временной матрицы кратчайших путей на k-ой фазе: все изменения можно делать сразу в матрице $d[][]$. В самом деле, если мы улучшили (уменьшили) какое-то значение в матрице расстояний, мы не могли ухудшить тем самым длину кратчайшего пути для каких-то других пар вершин, обработанных позднее.

Асимптотика алгоритма, очевидно, составляет $O(n^3)$.

Реализация

На вход программе подаётся граф, заданный в виде матрицы смежности — двумерного массива $d[][]$ размера $n \times n$, в котором каждый элемент задаёт длину ребра между соответствующими вершинами.

Требуется, чтобы выполнялось $d[i][i] = 0$ для любых i.

```
for (int k=0; k<n; ++k)
        for (int i=0; i<n; ++i)
                for (int j=0; j<n; ++j)
                        d[i][j] = min (d[i][j], d[i][k] + d[k][j]);
```

Предполагается, что если между двумя какими-то вершинами **нет ребра**, то в матрице смежности было записано какое-то большое число (достаточно большое, чтобы оно было больше длины любого пути в этом графе); тогда это ребро всегда будет невыгодно брать, и алгоритм сработает правильно.

Правда, если не принять специальных мер, то при наличии в графе рёбер **отрицательного веса**, в результирующей матрице могут появиться числа вида $\infty - 1, \infty - 2$, и т.д., которые, конечно, по-прежнему означают, что между соответствующими вершинами вообще нет пути. Поэтому при наличии в графе отрицательных рёбер алгоритм Флойда лучше написать так, чтобы он не выполнял переходы из тех состояний, в которых уже стоит "нет пути":

```
for (int k=0; k<n; ++k)
        for (int i=0; i<n; ++i)
                for (int j=0; j<n; ++j)
                        if (d[i][k] < INF && d[k][j] < INF)
                                d[i][j] = min (d[i][j], d[i][k] + d[k][j]);
```

Восстановление самих путей

Легко поддерживать дополнительную информацию — так называемых "предков", по которым можно будет восстанавливать сам кратчайший путь между любыми двумя заданными вершинами **в виде последовательности вершин**.

Для этого достаточно кроме матрицы расстояний $d[][]$ поддерживать также **матрицу предков** $p[][]$, которая для каждой пары вершин будет содержать номер фазы, на которой было получено кратчайшее расстояние между ними. Понятно, что этот номер фазы является не чем иным, как "средней" вершиной искомого кратчайшего пути, и теперь нам просто надо найти кратчайший путь между вершинами i и $p[i][j]$, а также между $p[i][j]$ и j. Отсюда получается простой рекурсивный алгоритм восстановления кратчайшего пути.

Случай вещественных весов

Если веса рёбер графа не целочисленные, а вещественные, то следует учитывать погрешности, неизбежно возникающие при работе с типами с плавающей точкой.

Применительно к алгоритму Флойда неприятным спецэффектом этих погрешностей становится то, что найденные алгоритмом расстояния могут уйти сильно в минус из-за **накопившихся ошибок**. В самом деле, если на первой фазе имела место ошибка Δ, то на второй итерации эта ошибка уже может превратиться в 2Δ, на третьей — в 4Δ, и так далее.

Чтобы этого не происходило, сравнения в алгоритме Флойда следует делать с учётом погрешности:

```
if (d[i][k] + d[k][j] < d[i][j] - EPS)
        d[i][j] = d[i][k] + d[k][j];
```

Случай отрицательных циклов

Если в графе есть циклы отрицательного веса, то формально алгоритм Флойда-Уоршелла неприменим к такому графу.

На самом же деле, для тех пар вершин i и j, между которыми нельзя зайти в цикл отрицательного вес,

алгоритм отработает корректно.

Для тех же пар вершин, ответа для которых не существует (по причине наличия отрицательного цикла на пути между ними), алгоритм Флойда найдёт в качестве ответа какое-то число (возможно, сильно отрицательное, но не обязательно). Тем не менее, можно улучшить алгоритм Флойда, чтобы он аккуратно обрабатывал такие пары вершин и выводил для них, например, $-\infty$.

Для этого можно сделать, например, следующий **критерий** "не существования пути". Итак, пусть на данном графе отработал обычный алгоритм Флойда. Тогда между вершинами i и j не существует кратчайшего пути тогда и только тогда, когда найдётся такая вершина t, достижимая из i и из которой достижима j, для которой выполняется $d[t][t] < 0$.

Кроме того, при использовании алгоритма Флойда для графов с отрицательными циклами следует помнить, что возникающие в процессе работы расстояния могут сильно уходить в минус, экспоненциально с каждой фазой. Поэтому следует принять меры против целочисленного переполнения, ограничив все расстояния снизу какой-нибудь величиной (например, $-\mathrm{INF}$).

Более подробно об этой задаче см. отдельную статью: "**Нахождение отрицательного цикла в графе**".

Кратчайшие пути фиксированной длины, количества путей фиксированной длины

Ниже описываются решения этих двух задач, построенные на одной и той же идее: сведение задачи к возведению матрицы в степень (с обычной операцией умножения, и с модифицированной).

Количество путей фиксированной длины

Пусть задан ориентированный невзвешенный граф G с n вершинами, и задано целое число k. Требуется для каждой пары вершин i и j найти количество путей между этими вершинами, состоящих ровно из k рёбер. Пути при этом рассматриваются произвольные, не обязательно простые (т.е. вершины могут повторяться сколько угодно раз).

Будем считать, что граф задан **матрицей смежности**, т.е. матрицей $g[][]$ размера $n \times n$, где каждый элемент $g[i][j]$ равен единице, если между этими вершинами есть ребро, и нулю, если ребра нет. Описываемый ниже алгоритм работает и в случае наличия кратных рёбер: если между какими-то вершинами i и j есть сразу m рёбер, то в матрицу смежности следует записать это число m. Также алгоритм корректно учитывает петли в графе, если таковые имеются.

Очевидно, что в таком виде **матрица смежности** графа является **ответом на задачу при** $k = 1$ — она содержит количества путей длины 1 между каждой парой вершин.

Решение будем строить **итеративно**: пусть ответ для некоторого k найден, покажем, как построить его для $k + 1$. Обозначим через d_k найденную матрицу ответов для k, а через d_{k+1} — матрицу ответов, которую необходимо построить. Тогда очевидна следующая формула:

$$d_{k+1}[i][j] = \sum_{p=1}^{n} d_k[i][p] \cdot g[p][j].$$

Легко заметить, что записанная выше формула — не что иное, как произведение двух матриц d_k и g в самом обычном смысле:

$$d_{k+1} = d_k \cdot g.$$

Таким образом, **решение** этой задачи можно представить следующим образом:

$$d_k = \underbrace{g \cdot \ldots \cdot g}_{k \text{ times}} = g^k.$$

Осталось заметить, что возведение матрицы в степень можно произвести эффективно с помощью алгоритма **Бинарного возведения в степень**.

Итак, полученное решение имеет асимптотику $O(n^3 \log k)$ и заключается в бинарном возведении в k-ую степень матрицы смежности графа.

Кратчайшие пути фиксированной длины

Пусть задан ориентированный взвешенный граф G с n вершинами, и задано целое число k. Требуется для каждой пары вершин i и j найти длину кратчайшего пути между этими вершинами, состоящего ровно из k рёбер.

Будем считать, что граф задан **матрицей смежности**, т.е. матрицей $g[][]$ размера $n \times n$, где каждый элемент $g[i][j]$ содержит длину ребра из вершины i в вершину j. Если между какими-то вершинами ребра нет, то соответствующий элемент матрицы считаем равным бесконечности ∞.

Очевидно, что в таком виде **матрица смежности** графа является **ответом на задачу при** $k = 1$ — она содержит длины кратчайших путей между каждой парой вершин, или ∞, если пути длины 1 не существует.

Решение будем строить **итеративно**: пусть ответ для некоторого k найден, покажем, как построить его для $k + 1$. Обозначим через d_k найденную матрицу ответов для k, а через d_{k+1} — матрицу ответов, которую необходимо построить. Тогда очевидна следующая формула:

$$d_{k+1}[i][j] = \min_{p=1\ldots n} (d_k[i][p] + g[p][j]).$$

Внимательно посмотрев на эту формулу, легко провести аналогию с матричным умножением: фактически, матрица

d_k умножается на матрицу g, только в операции умножения вместо суммы по всем p берётся минимум по всем p:

$$d_{k+1} = d_k \odot g,$$

где операция $\odot$ умножения двух матриц определяется следующим образом:

$$A \odot B = C \quad \Longleftrightarrow \quad C_{ij} = \min_{p=1\ldots n} (A_{ip} + B_{pj}).$$

Таким образом, **решение** этой задачи можно представить с помощью этой операции умножения следующим образом:

$$d_k = \underbrace{g \odot \ldots \odot g}_{k \text{ times}} = g^{\odot k}.$$

Осталось заметить, что возведение в степень с этой операцией умножения можно произвести эффективно с помощью алгоритма **Бинарного возведения в степень**, поскольку единственное требуемое для него свойство — ассоциативность операции умножения — очевидно, имеется.

Итак, полученное решение имеет асимптотику $O(n^3 \log k)$ и заключается в бинарном возведении в k-ую степень матрицы смежности графа с изменённой операцией умножения матриц.

Минимальное остовное дерево. Алгоритм Прима

Дан взвешенный неориентированный граф G с n вершинами и m рёбрами. Требуется найти такое поддерево этого графа, которое бы соединяло все его вершины, и при этом обладало наименьшим возможным весом (т.е. суммой весов рёбер). Поддерево — это набор рёбер, соединяющих все вершины, причём из любой вершины можно добраться до любой другой ровно одним простым путём.

Такое поддерево называется минимальным остовным деревом или просто **минимальным остовом**. Легко понять, что любой остов обязательно будет содержать $n - 1$ ребро.

В **естественной постановке** эта задача звучит следующим образом: есть n городов, и для каждой пары известна стоимость соединения их дорогой (либо известно, что соединить их нельзя). Требуется соединить все города так, чтобы можно было доехать из любого города в другой, а при этом стоимость прокладки дорог была бы минимальной.

Алгоритм Прима

Этот алгоритм назван в честь американского математика Роберта Прима (Robert Prim), который открыл этот алгоритм в 1957 г. Впрочем, ещё в 1930 г. этот алгоритм был открыт чешским математиком Войтеком Ярником (Vojtěch Jarník). Кроме того, Эдгар Дейкстра (Edsger Dijkstra) в 1959 г. также изобрёл этот алгоритм, независимо от них.

Описание алгоритма

Сам **алгоритм** имеет очень простой вид. Искомый минимальный остов строится постепенно, добавлением в него рёбер по одному. Изначально остов полагается состоящим из единственной вершины (её можно выбрать произвольно). Затем выбирается ребро минимального веса, исходящее из этой вершины, и добавляется в минимальный остов. После этого остов содержит уже две вершины, и теперь ищется и добавляется ребро минимального веса, имеющее один конец в одной из двух выбранных вершин, а другой — наоборот, во всех остальных, кроме этих двух. И так далее, т.е. всякий раз ищется минимальное по весу ребро, один конец которого — уже взятая в остов вершина, а другой конец — ещё не взятая, и это ребро добавляется в остов (если таких рёбер несколько, можно взять любое). Этот процесс повторяется до тех пор, пока остов не станет содержать все вершины (или, что то же самое, $n - 1$ ребро).

В итоге будет построен остов, являющийся минимальным. Если граф был изначально не связен, то остов найден не будет (количество выбранных рёбер останется меньше $n - 1$).

Доказательство

Пусть граф G был связным, т.е. ответ существует. Обозначим через T остов, найденный алгоритмом Прима, а через S — минимальный остов. Очевидно, что T действительно является остовом (т.е. поддеревом графа G). Покажем, что веса S и T совпадают.

Рассмотрим первый момент времени, когда в T происходило добавление ребра, не входящего в оптимальный остов S. Обозначим это ребро через e, концы его — через a и b, а множество входящих на тот момент в остов вершин — через V (согласно алгоритму, $a \in V$, $b \notin V$, либо наоборот). В оптимальном остове S вершины a и b соединяются каким-то путём P; найдём в этом пути любое ребро g, один конец которого лежит в V, а другой — нет. Поскольку алгоритм Прима выбрал ребро e вместо ребра g, то это значит, что вес ребра g больше либо равен весу ребра e.

Удалим теперь из S ребро g, и добавим ребро e. По только что сказанному, вес остова в результате не мог увеличиться (уменьшиться он тоже не мог, поскольку S было оптимальным). Кроме того, S не перестало быть остовом (в том, что связность не нарушилась, нетрудно убедиться: мы замкнули путь P в цикл, и потом удалили из этого цикла одно ребро).

Итак, мы показали, что можно выбрать оптимальный остов S таким образом, что он будет включать ребро e. Повторяя эту процедуру необходимое число раз, мы получаем, что можно выбрать оптимальный остов S так, чтобы он совпадал с T. Следовательно, вес построенного алгоритмом Прима T минимален, что и требовалось доказать.

Реализации

Время работы алгоритма существенно зависит от того, каким образом мы производим поиск очередного минимального ребра среди подходящих рёбер. Здесь могут быть разные подходы, приводящие к разным асимптотикам и разным реализациям.

Тривиальная реализация: алгоритмы за $O(nm)$ и $O(n^2 + m \log n)$

Если искать каждый раз ребро простым просмотром среди всех возможных вариантов, то асимптотически будет требоваться просмотр $O(m)$ рёбер, чтобы найти среди всех допустимых ребро с наименьшим весом. Суммарная асимптотика алгоритма составит в таком случае $O(nm)$, что в худшем случае есть $O(n^3)$, — слишком медленный алгоритм.

Этот алгоритм можно улучшить, если просматривать каждый раз не все рёбра, а только по одному ребру из каждой уже выбранной вершины. Для этого, например, можно отсортировать рёбра из каждой вершины в порядке возрастания весов, и хранить указатель на первое допустимое ребро (напомним, допустимы только те рёбра, которые ведут в множество ещё не выбранных вершин). Тогда, если пересчитывать эти указатели при каждом добавлении ребра в остов, суммарная асимптотика алгоритма будет $O(n^2 + m)$, но предварительно потребуется выполнить сортировку всех рёбер за $O(m \log n)$, что в худшем случае (для плотных графов) даёт асимптотику $O(n^2 \log n)$.

Ниже мы рассмотрим два немного других алгоритма: для плотных и для разреженных графов, получив в итоге заметно лучшую асимптотику.

Случай плотных графов: алгоритм за $O(n^2)$

Подойдём к вопросу поиска наименьшего ребра с другой стороны: для каждой ещё не выбранной будем хранить минимальное ребро, ведущее в уже выбранную вершину.

Тогда, чтобы на текущем шаге произвести выбор минимального ребра, надо просто просмотреть эти минимальные рёбра у каждой не выбранной ещё вершины — асимптотика составит $O(n)$.

Но теперь при добавлении в остов очередного ребра и вершины эти указатели надо пересчитывать. Заметим, что эти указатели могут только уменьшаться, т.е. у каждой не просмотренной ещё вершины надо либо оставить её указатель без изменения, либо присвоить ему вес ребра в только что добавленную вершину. Следовательно, эту фазу можно сделать также за $O(n)$.

Таким образом, мы получили вариант алгоритма Прима с асимптотикой $O(n^2)$.

В частности, такая реализация особенно удобна для решения так называемой **евклидовой задачи о минимальном остове**: когда даны n точек на плоскости, расстояние между которыми измеряется по стандартной евклидовой метрике, и требуется найти остов минимального веса, соединяющий их все (причём добавлять новые вершины где-либо в других местах запрещается). Эта задача решается описанным здесь алгоритмом за $O(n^2)$ времени и $O(n)$ памяти, чего не получится добиться алгоритмом Крускала.

Реализация алгоритма Прима для графа, заданного матрицей смежности $g[][]$:

```
// входные данные
int n;
vector < vector<int> > g;
const int INF = 1000000000; // значение "бесконечность"

// алгоритм
vector<bool> used (n);
vector<int> min_e (n, INF), sel_e (n, -1);
min_e[0] = 0;
for (int i=0; i<n; ++i) {
        int v = -1;
        for (int j=0; j<n; ++j)
                if (!used[j] && (v == -1 || min_e[j] < min_e[v]))
                        v = j;
        if (min_e[v] == INF) {
                cout << "No MST!";
                exit(0);
        }

        used[v] = true;
        if (sel_e[v] != -1)
                cout << v << " " << sel_e[v] << endl;

        for (int to=0; to<n; ++to)
                if (g[v][to] < min_e[to]) {
                        min_e[to] = g[v][to];
                        sel_e[to] = v;
                }
}
```

На вход подаются число вершин n и матрица $g[][]$ размера $n \times n$, в которой отмечены веса рёбер, и стоят числа INF, если соответствующее ребро отсутствует. Алгоритм поддерживает три массива: флаг $\text{used}[i] = \text{true}$ означает, что вершина i включена в остов, величина $\min_e[i]$ хранит вес наименьшего допустимого ребра из вершины i, а элемент $\text{sel_e}[i]$ содержит конец этого наименьшего ребра (это нужно для вывода рёбер в ответе). Алгоритм делает n шагов, на каждом из которых выбирает вершину v с наименьшей меткой $\min_e$, помечает её used, и затем просматривает все рёбра из этой вершины, пересчитывая их метки.

Случай разреженных графов: алгоритм за $O(m \log n)$

В описанном выше алгоритме можно увидеть стандартные операции нахождения минимума в множестве и изменение значений в этом множестве. Эти две операции являются классическими, и выполняются многими структурами данных, например, реализованным в языке C++ красно-чёрным деревом set.

По смыслу алгоритм остаётся точно таким же, однако теперь мы можем найти минимальное ребро за время $O(\log n)$. С другой стороны, время на пересчёт n указателей теперь составит $O(n \log n)$, что хуже, чем в вышеописанном алгоритме.

Если учесть, что всего будет $O(m)$ пересчётов указателей и $O(n)$ поисков минимального ребра, то суммарная асимптотика составит $O(m \log n)$ — для разреженных графов это лучше, чем оба вышеописанных алгоритма, но на плотных графах этот алгоритм будет медленнее предыдущего.

Реализация алгоритма Прима для графа, заданного списками смежности $g[]$:

```
// входные данные
int n;
vector < vector < pair<int,int> > > g;
const int INF = 1000000000; // значение "бесконечность"

// алгоритм
vector<int> min_e (n, INF), sel_e (n, -1);
min_e[0] = 0;
set < pair<int,int> > q;
q.insert (make_pair (0, 0));
for (int i=0; i<n; ++i) {
        if (q.empty()) {
                cout << "No MST!";
                exit(0);
        }
        int v = q.begin()->second;
        q.erase (q.begin());

        if (sel_e[v] != -1)
                cout << v << " " << sel_e[v] << endl;

        for (size_t j=0; j<g[v].size(); ++j) {
                int to = g[v][j].first,
                        cost = g[v][j].second;
                if (cost < min_e[to]) {
                        q.erase (make_pair (min_e[to], to));
                        min_e[to] = cost;
                        sel_e[to] = v;
                        q.insert (make_pair (min_e[to], to));
                }
        }
}
```

На вход подаются число вершин n и n списков смежности: $g[i]$ — это список всех рёбер, исходящих из вершины i, в виде пар (второй конец ребра, вес ребра). Алгоритм поддерживает два массива: величина $\min_e[i]$ хранит вес наименьшего допустимого ребра из вершины i, а элемент $\text{sel_e}[i]$ содержит конец этого наименьшего ребра (это нужно для вывода рёбер в ответе). Кроме того, поддерживается очередь q из всех вершин в порядке увеличения их меток $\min_e$. Алгоритм делает n шагов, на каждом из которых выбирает вершину v с наименьшей меткой $\min_e$ (просто извлекая её из начала очереди), и затем просматривает все рёбра из этой вершины, пересчитывая их метки (при пересчёте мы удаляем из очереди старую величину, и затем кладём обратно новую).

Аналогия с алгоритмом Дейкстры

В двух описанных только что алгоритмах прослеживается вполне чёткая аналогия с алгоритмом Дейкстры: он имеет такую же структуру ($n - 1$ фаза, на каждой из которых сначала выбирается оптимальное ребро, добавляется в ответ, а затем пересчитываются значения для всех не выбранных ещё вершин). Более того, алгоритм Дейкстры тоже имеет

два варианта реализации: за $O(n^2)$ и $O(m \log n)$ (мы, конечно, здесь не учитываем возможность использования сложных структур данных для достижения ещё меньших асимптотик).

Если взглянуть на алгоритмы Прима и Дейкстры более формально, то получается, что они вообще идентичны друг другу, за исключением **весовой функции** вершин: если в алгоритме Дейкстры у каждой вершины поддерживается длина кратчайшего пути (т.е. сумма весов некоторых рёбер), то в алгоритме Прима каждой вершине приписывается только вес минимального ребра, ведущего в множество уже взятых вершин.

На уровне реализации это означает, что после добавления очередной вершины v в множество выбранных вершин, когда мы начинаем просматривать все рёбра (v, to) из этой вершины, то в алгоритме Прима указатель to обновляется весом ребра (v, to), а в алгоритме Дейкстры — метка расстояния $d[to]$ обновляется суммой метки $d[v]$ и веса ребра (v, to). В остальном эти два алгоритма можно считать идентичными (хоть они и решают совсем разные задачи).

Свойства минимальных остовов

- **Максимальный** остов также можно искать алгоритмом Прима (например, заменив все веса рёбер на противоположные: алгоритм не требует неотрицательности весов рёбер).
- Минимальный остов **единственен**, если веса всех рёбер различны. В противном случае, может существовать несколько минимальных остовов (какой именно будет выбран алгоритмом Прима, зависит от порядка просмотра рёбер/вершин с одинаковыми весами/указателями)
- Минимальный остов также является остовом, **минимальным по произведению** всех рёбер (предполагается, что все веса положительны). В самом деле, если мы заменим веса всех рёбер на их логарифмы, то легко заметить, что в работе алгоритма ничего не изменится, и будут найдены те же самые рёбра.
- Минимальный остов является остовом с минимальным весом **самого тяжёлого ребра**. Яснее всего это утверждение понятно, если рассмотреть работу алгоритма Крускала.
- **Критерий минимальности** остова: остов является минимальным тогда и только тогда, когда для любого ребра, не принадлежащего остову, цикл, образуемый этим ребром при добавлении к остову, не содержит рёбер тяжелее этого ребра. В самом деле, если для какого-то ребра оказалось, что оно легче некоторых рёбер образуемого цикла, то можно получить остов с меньшим весом (добавив это ребро в остов, и удалив самое тяжелое ребро из цикла). Если же это условие не выполнилось ни для одного ребра, то все эти рёбра не улучшают вес остова при их добавлении.

Минимальное остовное дерево. Алгоритм Крускала

Дан взвешенный неориентированный граф. Требуется найти такое поддерево этого графа, которое бы соединяло все его вершины, и при этом обладало наименьшим весом (т.е. суммой весов рёбер) из всех возможных. Такое поддерево называется минимальным остовным деревом или простом минимальным остовом.

Здесь будут рассмотрены несколько важных фактов, связанных с минимальными остовами, затем будет рассмотрен алгоритм Крускала в его простейшей реализации.

Свойства минимального остова

- Минимальный остов **уникален**, **если веса всех рёбер различны**. В противном случае, может существовать несколько минимальных остовов (конкретные алгоритмы обычно получают один из возможных остовов).
- Минимальный остов является также и **остовом с минимальным произведением** весов рёбер. (доказывается это легко, достаточно заменить веса всех рёбер на их логарифмы)
- Минимальный остов является также и **остовом с минимальным весом самого тяжелого ребра**. (это утверждение следует из справедливости алгоритма Крускала)
- **Остов максимального веса** ищется аналогично остову минимального веса, достаточно поменять знаки всех рёбер на противоположные и выполнить любой из алгоритм минимального остова.

Алгоритм Крускала

Данный алгоритм был описан Крускалом (Kruskal) в 1956 г.

Алгоритм Крускала изначально помещает каждую вершину в своё дерево, а затем постепенно объединяет эти деревья, объединяя на каждой итерации два некоторых дерева некоторым ребром. Перед началом выполнения алгоритма, все рёбра сортируются по весу (в порядке неубывания). Затем начинается процесс объединения: перебираются все рёбра от первого до последнего (в порядке сортировки), и если у текущего ребра его концы принадлежат разным поддеревьям, то эти поддеревья объединяются, а ребро добавляется к ответу. По окончании перебора всех рёбер все вершины окажутся принадлежащими одному поддереву, и ответ найден.

Простейшая реализация

Этот код самым непосредственным образом реализует описанный выше алгоритм, и выполняется за $O(M \log N + N^2)$. Сортировка рёбер потребует O (M log N) операций. Принадлежность вершины тому или иному поддереву хранится просто с помощью массива tree_id - в нём для каждой вершины хранится номер дерева, которому она принадлежит. Для каждого ребра мы за O (1) определяем, принадлежат ли его концы разным деревьям. Наконец, объединение двух деревьев осуществляется за O (N) простым проходом по массиву tree_id. Учитывая, что всего операций объединения будет N-1, мы и получаем асимптотику $O(M \log N + N^2)$.

```
int m;
vector < pair < int, pair<int,int> > > g (m); // вес - вершина 1 - вершина 2

int cost = 0;
vector < pair<int,int> > res;

sort (g.begin(), g.end());
vector<int> tree_id (n);
for (int i=0; i<n; ++i)
        tree_id[i] = i;
for (int i=0; i<m; ++i)
{
        int a = g[i].second.first,  b = g[i].second.second,  l = g[i].first;
        if (tree_id[a] != tree_id[b])
        {
                cost += l;
                res.push_back (make_pair (a, b));
                int old_id = tree_id[b],  new_id = tree_id[a];
                for (int j=0; j<n; ++j)
                        if (tree_id[j] == old_id)
                                tree_id[j] = new_id;
        }
}
```

Улучшенная реализация

С использованием структуры данных "Система непересекающихся множеств" можно написать более быструю реализацию алгоритма Крускала с асимптотикой O (M log N).

Минимальное остовное дерево. Алгоритм Крускала с системой непересекающихся множеств

Постановку задачи и описание алгоритма Крускала см. здесь.

Здесь будет рассмотрена реализация с использованием структуры данных "система непересекающихся множеств" (DSU), которая позволит достигнуть асимптотики O (M log N).

Описание

Так же, как и в простой версии алгоритма Крускала, отсортируем все рёбра по неубыванию веса. Затем поместим каждую вершину в своё дерево (т.е. своё множество) с помощью вызова функции DSU MakeSet - на это уйдёт в сумме O (N). Перебираем все рёбра (в порядке сортировки) и для каждого ребра за O (1) определяем, принадлежат ли его концы разным деревьям (с помощью двух вызовов FindSet за O (1)). Наконец, объединение двух деревьев будет осуществляться вызовом Union - также за O (1). Итого мы получаем асимптотику O (M log N + N + M) = O (M log N).

Реализация

Для уменьшения объёма кода реализуем все операции не в виде отдельных функций, а прямо в коде алгоритма Крускала.

Здесь будет использоваться рандомизированная версия DSU.

```
vector<int> p (n);

int dsu_get (int v) {
        return (v == p[v]) ? v : (p[v] = dsu_get (p[v]));
}

void dsu_unite (int a, int b) {
        a = dsu_get (a);
        b = dsu_get (b);
        if (rand() & 1)
                swap (a, b);
        if (a != b)
                p[a] = b;
}

... в функции main(): ...

int m;
vector < pair < int, pair<int,int> > > g; // вес - вершина 1 - вершина 2
... чтение графа ...

int cost = 0;
vector < pair<int,int> > res;

sort (g.begin(), g.end());
p.resize (n);
for (int i=0; i<n; ++i)
        p[i] = i;
for (int i=0; i<m; ++i) {
        int a = g[i].second.first,  b = g[i].second.second,  l = g[i].first;
        if (dsu_get(a) != dsu_get(b)) {
                cost += l;
                res.push_back (g[i].second);
                dsu_unite (a, b);
        }
}
```

Матричная теорема Кирхгофа.
Нахождение количества остовных деревьев

Задан связный неориентированный граф своей матрицей смежности. Кратные рёбра в графе допускаются. Требуется посчитать количество различных остовных деревьев этого графа.

Приведённая ниже формула принадлежит Кирхгофу (Kirchhoff), который доказал её в 1847 г.

Матричная теорема Кирхгофа

Возьмём матрицу смежности графа G, заменим каждый элемент этой матрицы на противоположный, а на диагонале вместо элемента $A_{i,i}$ поставим степень вершины i (если имеются кратные рёбра, то в степени вершины они учитываются со своей кратностью). Тогда, согласно матричной теореме Кирхгофа, все алгебраические дополнения этой матрицы равны между собой, и равны количеству остовных деревьев этого графа. Например, можно удалить последнюю строку и последний столбец этой матрицы, и модуль её определителя будет равен искомому количеству.

Определитель матрицы можно найти за $O(N^3)$ с помощью метода Гаусса или метода Краута.

Доказательство этой теоремы достаточно сложно и здесь не приводится (см., например, Приезжев В.Б. "Задача о димерах и теорема Кирхгофа").

Связь с законами Кирхгофа в электрической цепи

Между матричной теоремой Кирхгофа и законами Кирхгофа для электрической цепи имеется удивительная связь.

Можно показать (как следствие из закона Ома и первого закона Кирхгофа), что сопротивление R_{ij} между точками i и j электрической цепи равно:

$$R_{ij} = |T^{(i,j)}| \ / \ |T^{j}|$$

где матрица T получена из матрицы A *обратных* сопротивлений проводников (A_{ij} - обратное число к сопротивлению проводника между точками i и j) преобразованием, описанным в матричной теореме Кирхгофа, а обозначение $T^{(i)}$ обозначает вычёркивание строки и столбца с номером i, а $T^{(i,j)}$ - вычёркивание двух строк и столбцов i и j.

Теорема Кирхгофа придаёт этой формуле геометрический смысл.

Код Прюфера. Формула Кэли. Количество способов сделать граф связным

В данной статье мы рассмотрим так называемый **код Прюфера**, который представляет из себя способ однозначного кодирования помеченного дерева с помощью последовательности чисел.

С помощью кодов Прюфера демонстрируется доказательство **формулы Кэли** (задающей количество остовных деревьев в полном графе), а также решение задачи о количестве способов добавить в заданный граф рёбра, чтобы превратить его в связный.

Примечание. Мы не будем рассматривать деревья, состоящие из единственной вершины, — это особый случай, на котором многие утверждения вырождаются.

Код Прюфера

Код Прюфера — это способ взаимно однозначного кодирования помеченных деревьев с n вершинами с помощью последовательности $n-2$ целых чисел в отрезке $[1;n]$. Иными словами, код Прюфера — это **биекция** между всеми остовными деревьями полного графа и числовыми последовательностями.

Хотя использовать код Прюфера для хранения и оперирования с деревьями нецелесообразно из-за специфичности представления, коды Прюфера находят применения в решении комбинаторных задач.

Автор — Хейнц Прюфер (Heinz Prüfer) — предложил этот код в 1918 г. как доказательство формулы Кэли (см. ниже).

Построение кода Прюфера для данного дерева

Код Прюфера строится следующим образом. Будем $n-2$ раза проделывать процедуру: выбираем лист дерева с наименьшим номером, удаляем его из дерева, и добавляем к коду Прюфера номер вершины, которая была связана с этим листом. В конце концов в дереве останется только 2 вершины, и алгоритм на этом завершается (номер этих вершин явным образом в коде не записываются).

Таким образом, код Прюфера для заданного дерева — это последовательность из $n-2$ чисел, где каждое число — номер вершины, связанной с наименьшим на тот момент листом — т.е. это число в отрезке $[1;n]$.

Алгоритм вычисления кода Прюфера легко реализовать с асимптотикой $O(n \log n)$, просто поддерживая структуру данных для извлечения минимума (например, $\text{set}<>$ или $\text{priority_queue}<>$ в языке C++), содержащую в себе список всех текущих листьев:

```
const int MAXN = ...;
int n;
vector<int> g[MAXN];
int degree[MAXN];
bool killed[MAXN];

vector<int> prufer_code() {
        set<int> leaves;
        for (int i=0; i<n; ++i) {
                degree[i] = (int) g[i].size();
                if (degree[i] == 1)
                        leaves.insert (i);
                killed[i] = false;
        }

        vector<int> result (n-2);
        for (int iter=0; iter<n-2; ++iter) {
                int leaf = *leaves.begin();
                leaves.erase (leaves.begin());
                killed[leaf] = true;

                int v;
                for (size_t i=0; i<g[leaf].size(); ++i)
                        if (!killed[g[leaf][i]])
                                v = g[leaf][i];

                result[iter] = v;
```

```
                if (--degree[v] == 1)
                        leaves.insert (v);
        }
        return result;
}
```

Впрочем, построение кода Прюфера можно реализовать и за линейное время, что описывается в следующем разделе.

Построение кода Прюфера для данного дерева за линейное время

Приведём здесь простой алгоритм, имеющий асимптотику $O(n)$.

Суть алгоритма заключается в хранении **движущегося указателя** ptr, который всегда будет продвигаться только в сторону увеличения номеров вершин.

На первый взгляд, такое невозможно, ведь в процессе построения кода Прюфера номера листьев могут как увеличиваться, так и **уменьшаться**. Однако легко заметить, что уменьшения происходят только в единственном случае: кода при удалении текущего листа его предок имеет меньший номер (этот предок станет минимальным листом и удалится из дерева на следующем же шаге кода Прюфера). Таким образом, случаи уменьшения можно обработать за время $O(1)$, и ничего не мешает построению алгоритма с **линейной асимптотикой**:

```
const int MAXN = ...;
int n;
vector<int> g[MAXN];
int parent[MAXN], degree[MAXN];

void dfs (int v) {
        for (size_t i=0; i<g[v].size(); ++i) {
                int to = g[v][i];
                if (to != parent[v]) {
                        parent[to] = v;
                        dfs (to);
                }
        }
}

vector<int> prufer_code() {
        parent[n-1] = -1;
        dfs (n-1);

        int ptr = -1;
        for (int i=0; i<n; ++i) {
                degree[i] = (int) g[i].size();
                if (degree[i] == 1 && ptr == -1)
                        ptr = i;
        }

        vector<int> result;
        int leaf = ptr;
        for (int iter=0; iter<n-2; ++iter) {
                int next = parent[leaf];
                result.push_back (next);
                --degree[next];
                if (degree[next] == 1 && next < ptr)
                        leaf = next;
                else {
                        ++ptr;
                        while (ptr<n && degree[ptr] != 1)
                                ++ptr;
                        leaf = ptr;
                }
        }
        return result;
}
```

Прокомментируем этот код. Основная функция здесь — $\mathrm{prufer_code}()$, которая возвращает код Прюфера для дерева, заданного в глобальных переменных n (количество вершин) и g (списки смежности, задающие граф). Вначале мы находим для каждой вершины её предка $\mathrm{parent}[i]$ — т.е. того предка, которого эта вершина будет иметь в момент удаления из дерева (всё это мы можем найти заранее, пользуясь тем, что максимальная вершина $n-1$

никогда не удалится из дерева). Также мы находим для каждой вершины её степень $\mathrm{degree}[i]$. Переменная ptr — это движущийся указатель ("кандидат" на минимальный лист), который изменяется всегда только в сторону увеличения. Переменная leaf — это текущий лист с минимальным номером. Таким образом, каждая итерация кода Прюфера заключается в добавлении leaf в ответ, а также проверке, не оказалось ли $\mathrm{parent}[\mathrm{leaf}]$ меньше текущего кандидата ptr: если оказалось меньше, то мы просто присваиваем $\mathrm{leaf} = \mathrm{parent}[\mathrm{leaf}]$, а в противном случае — двигаем указатель ptr до следующего листа.

Как легко видно по коду, асимптотика алгоритма действительно составляет $O(n)$: указатель ptr претерпит лишь $O(n)$ изменений, а все остальные части алгоритма очевидно работают за линейное время.

Некоторые свойства кодов Прюфера

- По окончании построения кода Прюфера в дереве останутся неудалёнными две вершины.

  Одной из них точно будет вершина с максимальным номером — $n - 1$, а вот про другую вершину ничего определённого сказать нельзя.

- Каждая вершина встречается в коде Прюфера определённое число раз, равное её степени минус один.

  Это легко понять, если заметить, что вершина удаляется из дерева в момент, когда её степень равна единице — т.е. к этому моменту все смежные с ней рёбра, кроме одного, были удалены. (Для двух оставшихся после построения кода вершин это утверждение тоже верно.)

Восстановление дерева по его коду Прюфера

Для восстановления дерева достаточно заметить из предыдущего пункта, что степени всех вершин в искомом дереве мы уже знаем (и можем посчитать и сохранить в некотором массиве $degree[]$). Следовательно, мы можем найти все листья, и, соответственно, номер наименьшего листа — который был удалён на первом шаге. Этот лист был соединён с вершиной, номер которой записан в первой ячейке кода Прюфера.

Таким образом, мы нашли первое ребро, удалённое кодом Прюфера. Добавим это ребро в ответ, затем уменьшим степени $degree[]$ у обоих концов ребра.

Будем повторять эту операцию, пока не просмотрим весь код Прюфера: искать минимальную вершину с $degree = 1$, соединять её с очередной вершиной кода Прюфера, уменьшать $degree[]$ у обоих концов.

В конце концов у нас останется только две вершины с $degree = 1$ — это те вершины, который алгоритм Прюфера оставил неудалёнными. Соединим их ребром.

Алгоритм завершён, искомое дерево построено.

Реализовать этот алгоритм легко за время $O(n \log n)$: поддерживая в структуре данных для извлечения минимума (например, $\mathrm{set} <>$ или $\mathrm{priority_queue} <>$ в C++) номера всех вершин, имеющих $degree = 1$, и извлекая из него каждый раз минимум.

Приведём соответствующую реализацию (где функция $prufer_decode()$ возвращает список из рёбер искомого дерева):

```
vector < pair<int,int> > prufer_decode (const vector<int> & prufer_code) {
        int n = (int) prufer_code.size() + 2;
        vector<int> degree (n, 1);
        for (int i=0; i<n-2; ++i)
                ++degree[prufer_code[i]];

        set<int> leaves;
        for (int i=0; i<n; ++i)
                if (degree[i] == 1)
                        leaves.insert (i);

        vector < pair<int,int> > result;
        for (int i=0; i<n-2; ++i) {
                int leaf = *leaves.begin();
                leaves.erase (leaves.begin());

                int v = prufer_code[i];
                result.push_back (make_pair (leaf, v));
                if (--degree[v] == 1)
                        leaves.insert (v);
        }
        result.push_back (make_pair (*leaves.begin(), *--leaves.end()));
        return result;
}
```

Восстановление дерева по коду Прюфера за линейное время

Для получения алгоритма с линейной асимптотикой можно применить тот же самый приём, что применялся для получения линейного алгоритма вычисления кода Прюфера.

В самом деле, для нахождения листа с наименьшим номером необязательно заводить структуру данных для извлечения минимума. Вместо этого можно заметить, что, после того как мы находим и обрабатываем текущий лист, он добавляет в рассмотрение только одну новую вершину. Следовательно, мы можем обойтись одним движущимся указателем вместе с переменной, хранящей в себе текущий минимальный лист:

```
vector < pair<int,int> > prufer_decode_linear (const vector<int> &
prufer_code) {
        int n = (int) prufer_code.size() + 2;
        vector<int> degree (n, 1);
        for (int i=0; i<n-2; ++i)
                ++degree[prufer_code[i]];

        int ptr = 0;
        while (ptr < n && degree[ptr] != 1)
                ++ptr;
        int leaf = ptr;

        vector < pair<int,int> > result;
        for (int i=0; i<n-2; ++i) {
                int v = prufer_code[i];
                result.push_back (make_pair (leaf, v));

                --degree[leaf];
                if (--degree[v] == 1 && v < ptr)
                        leaf = v;
                else {
                        ++ptr;
                        while (ptr < n && degree[ptr] != 1)
                                ++ptr;
                        leaf = ptr;
                }
        }
        for (int v=0; v<n-1; ++v)
                if (degree[v] == 1)
                        result.push_back (make_pair (v, n-1));
        return result;
}
```

Взаимная однозначность соответствия между деревьями и кодами Прюфера

С одной стороны, для каждого дерева существует ровно один код Прюфера, соответствующий ему (это следует из определения кода Прюфера).

С другой стороны, из корректности алгоритма восстановления дерева по коду Прюфера следует, что любому коду Прюфера (т.е. последовательности из $n-2$ чисел, где каждое число лежит в отрезке $[1;n]$) соответствует некоторое дерево.

Таким образом, все деревья и все коды Прюфера образуют **взаимно однозначное соответствие**.

Формула Кэли

Формула Кэли гласит, что **количество остовных деревьев в полном помеченном графе** из n вершин равно:

$$n^{n-2}.$$

Имеется много **доказательств** этой формулы, но доказательство с помощью кодов Прюфера наглядно и конструктивно.

В самом деле, любому набору из $n-2$ чисел из отрезка $[1;n]$ однозначно соответствует некоторое дерево из n вершин. Всего различных кодов Прюфера n^{n-2}. Поскольку в случае полного графа из n вершин в качестве

остова подходит любое дерево, то и количество остовных деревьев равно n^{n-2}, что и требовалось доказать.

Количество способов сделать граф связным

Мощь кодов Прюфера заключается в том, что они позволяют получить более общую формулу, чем формулу Кэли.

Итак, дан граф из n вершин и m рёбер; пусть k — количество компонент связности в этом графе. Требуется найти число способов добавить $k-1$ ребро, чтобы граф стал связным (очевидно, $k-1$ ребро — минимально необходимое количество рёбер, чтобы сделать граф связным).

Выведем готовую формулу для решения этой задачи.

Обозначим через $s_1, \ldots, s_k$ размеры компонент связности этого графа. Поскольку добавлять рёбра внутри компонент связности запрещено, то получается, что задача очень похожа на поиск количества остовных деревьев в полном графе из k вершин: но отличие здесь в том, что каждая вершина имеет свой "вес" s_i: каждое ребро, смежное с i-ой вершиной, умножает ответ на s_i.

Таким образом, для подсчёта количества способов оказывается важным, какие степени имеют все k вершин в остове. Для получения формулы для задачи надо просуммировать ответы по всем возможным степеням.

Пусть $d_1, \ldots, d_k$ — степени вершин в остове. Сумма степеней вершин равна удвоенному количеству рёбер, поэтому:

$$\sum_{i=1}^{k} d_i = 2k-2.$$

Если i-я вершина имеет степень d_i, то в код Прюфера она входит $d_i - 1$ раз. Код Прюфера для дерева из k вершин имеет длину $k-2$. Количество способов выбрать набор $k-2$ чисел, где число i встречается ровно $d_i - 1$ раз, равно **мультиномиальному коэффициенту** (по аналогии с биномиальным коэффициентом):

$$\binom{k-2}{d_1-1,\ d_2-1,\ \ldots\ ,d_k-1} = \frac{(k-2)!}{(d_1-1)!\,(d_2-1)!\ \ldots\ (d_k-1)!}.$$

С учётом того, что каждое ребро, смежное с i-ой вершиной, умножает ответ на s_i, получаем, что ответ, при условии, что степени вершин равны $d_1, \ldots, d_k$, равен:

$$s_1^{d_1} \cdot s_2^{d_2} \cdot \ldots \cdot s_k^{d_k} \cdot \frac{(k-2)!}{(d_1-1)!\,(d_2-1)!\ \ldots\ (d_k-1)!}.$$

Для получения ответа на задачу надо просуммировать эту формулу по всевозможным допустимым наборам $\{d_i\}_{i=1}^{i=k}$:

$$\sum_{\substack{d_i \geq 1, \\ \sum_{i=1}^{k} d_i = 2k-2}} s_1^{d_1} \cdot s_2^{d_2} \cdot \ldots \cdot s_k^{d_k} \cdot \frac{(k-2)!}{(d_1-1)!\,(d_2-1)!\ \ldots\ (d_k-1)!}.$$

Для свёртывания этой формулы воспользуемся определением мультиномиального коэффициента:

$$(x_1 + \ldots x_m)^p = \sum_{\substack{c_i \geq 0, \\ \sum_{i=1}^{m} c_i = p}} x_1^{c_1} \cdot x_2^{c_2} \cdot \ldots \cdot x_m^{c_m} \cdot \binom{m}{c_1,\ c_2,\ \ldots\ ,c_k}.$$

Сравнивая эту формулу с предыдущей, получаем, что если ввести обозначение $e_i = d_i - 1$:

$$\sum_{\substack{e_i \geq 0, \\ \sum_{i=1}^{k} e_i = k-2}} s_1^{e_1+1} \cdot s_2^{e_2+1} \cdot \ldots \cdot s_k^{e_k+1} \cdot \frac{(k-2)!}{e_1!\ e_2!\ \ldots\ e_k!},$$

то после сворачивания **ответ на задачу** равен:

$$s_1 \cdot s_2 \cdot \ldots \cdot s_k \cdot (s_1 + s_2 + \ldots + s_k)^{k-2} = s_1 \cdot s_2 \cdot \ldots \cdot s_k \cdot n^{k-2}.$$

(Эта формула верна и при $k = 1$, хотя формально из доказательства это не следовало.)

Задачи в online judges

Задачи в online judges, в которых применяются коды Прюфера:

- UVA #10843 "Anne's game" [сложность: низкая]
- TIMUS #1069 "**Код Прюфера**" [сложность: низкая]
- CODEFORCES 110D "**Улики**" [сложность: средняя]
- TopCoder SRM 460 "TheCitiesAndRoadsDivTwo" [сложность: средняя]

Нахождение отрицательного цикла в графе

Дан ориентированный взвешенный граф G с n вершинами и m рёбрами. Требуется найти в нём любой **цикл отрицательного веса**, если таковой имеется.

При другой постановке задачи — требуется найти **все пары вершин** такие, что между ними существует путь сколько угодно малой длины.

Эти два варианта задачи удобно решать разными алгоритмами, поэтому ниже будут рассмотрены оба из них.

Одна из распространённых "жизненных" постановок этой задачи — следующая: известны **курсы валют**, т.е. курсы перевода из одной валюты в другую. Требуется узнать, можно ли некоторой последовательностью обменов получить выгоду, т.е. стартовав с одной единицы какой-либо валюты, получить в итоге больше чем одну единицу этой же валюты.

Решение с помощью алгоритма Форда-Беллмана

Алгоритм Форда-Беллмана позволяет проверить наличие или отсутствие цикла отрицательного веса в графе, а при его наличии — найти один из таких циклов.

Не будем вдаваться здесь в подробности (которые описаны в статье по алгоритму Форда-Беллмана), а приведём лишь итог — то, как работает алгоритм.

Делается n итераций алгоритма Форда-Беллмана, и если на последней итерации не произошло никаких изменений — то отрицательного цикла в графе нет. В противном случае возьмём вершину, расстояние до которой изменилось, и будем идти от неё по предкам, пока не войдём в цикл; этот цикл и будет искомым отрицательным циклом.

Реализация:

```
struct edge {
        int a, b, cost;
};

int n, m;
vector<edge> e;
const int INF = 1000000000;

void solve() {
        vector<int> d (n);
        vector<int> p (n, -1);
        int x;
        for (int i=0; i<n; ++i) {
                x = -1;
                for (int j=0; j<m; ++j)
                        if (d[e[j].b] > d[e[j].a] + e[j].cost) {
                                d[e[j].b] = max (-INF, d[e[j].a] + e[j].cost);
                                p[e[j].b] = e[j].a;
                                x = e[j].b;
                        }
        }

        if (x == -1)
                cout << "No negative cycle found.";
        else {
                int y = x;
                for (int i=0; i<n; ++i)
                        y = p[y];

                vector<int> path;
                for (int cur=y; ; cur=p[cur]) {
                        path.push_back (cur);
                        if (cur == y && path.size() > 1)  break;
                }
                reverse (path.begin(), path.end());

                cout << "Negative cycle: ";
                for (size_t i=0; i<path.size(); ++i)
```

```
                        cout << path[i] << ' ';
        }
}
```

Решение с помощью алгоритма Флойда-Уоршелла

Алгоритм Флойда-Уоршелла позволяет решать вторую постановку задачи — когда надо найти все пары вершин (i, j), между которыми кратчайшего пути не существует (т.е. он имеет бесконечно малую величину).

Опять же, более подробные объяснения содержатся в описании алгоритма Флойда-Уоршелла, а здесь мы приведём только итог.

После того, как алгоритм Флойда-Уоршелла отработает для входного графа, переберём все пары вершин (i, j), и для каждой такой пары проверим, бесконечно мал кратчайший путь из i в j или нет. Для этого переберём третью вершину t, и если для неё оказалось $d[t][t] < 0$ (т.е. она лежит в цикле отрицательного веса), а сама она достижима из i и из неё достижима j — то путь (i, j) может иметь бесконечно малую длину.

Реализация:

```
for (int i=0; i<n; ++i)
        for (int j=0; j<n; ++j)
                for (int t=0; t<n; ++t)
                        if (d[i][t] < INF && d[t][t] < 0 && d[t][j] < INF)
                                d[i][j] = -INF;
```

Задачи в online judges

Список задач, в которых требуется искать цикл отрицательного веса:

- UVA #499 "Wormholes" [сложность: низкая]
- UVA #104 "Arbitrage" [сложность: средняя]
- UVA #10557 "XYZZY" [сложность: средняя]

Нахождение Эйлерова пути за O (M)

Эйлеров путь - это путь в графе, проходящий через все его рёбра. Эйлеров цикл - это эйлеров путь, являющийся циклом.

Задача заключается в том, чтобы найти эйлеров путь в **неориентированном мультиграфе с петлями**.

Алгоритм

Сначала проверим, существует ли эйлеров путь. Затем найдём все простые циклы и объединим их в один - это и будет эйлеровым циклом. Если граф таков, что эйлеров путь не является циклом, то, добавим недостающее ребро, найдём эйлеров цикл, потом удалим лишнее ребро.

Чтобы проверить, существует ли эйлеров путь, нужно воспользоваться следующей теоремой. Эйлеров цикл существует тогда и только тогда, когда степени всех вершин чётны. Эйлеров путь существует тогда и только тогда, когда количество вершин с нечётными степенями равно двум (или нулю, в случае существования эйлерова цикла).

Кроме того, конечно, граф должен быть достаточно связным (т.е. если удалить из него все изолированные вершины, то должен получиться связный граф).

Искать все циклы и объединять их будем одной рекурсивной процедурой:

```
procedure FindEulerPath (V)
        1. перебрать все рёбра, выходящие из вершины V;
                каждое такое ребро удаляем из графа, и
                вызываем FindEulerPath из второго конца этого ребра;
        2. добавляем вершину V в ответ.
```

Сложность этого алгоритма, очевидно, является линейной относительно числа рёбер.

Но этот же алгоритм мы можем записать в **нерекурсивном** варианте:

```
stack St;
в St кладём любую вершину (стартовая вершина);
пока St не пустой
        пусть V - значение на вершине St;
        если степень(V) = 0, то
                добавляем V к ответу;
                снимаем V с вершины St;
        иначе
                находим любое ребро, выходящее из V;
                удаляем его из графа;
                второй конец этого ребра кладём в St;
```

Несложно проверить эквивалентность этих двух форм алгоритма. Однако вторая форма, очевидно, быстрее работает, причём кода будет не больше.

Задача о домино

Приведём здесь классическую задачу на эйлеров цикл - задачу о домино.

Имеется N доминошек, как известно, на двух концах доминошки записано по одному числу (обычно от 1 до 6, но в нашем случае не важно). Требуется выложить все доминошки в ряд так, чтобы у любых двух соседних доминошек числа, записанные на их общей стороне, совпадали. Доминошки разрешается переворачивать.

Переформулируем задачу. Пусть числа, записанные на донимошках, - вершины графа, а доминошки - рёбра этого графа (каждая доминошка с числами (a,b) - это ребра (a,b) и (b,a)). Тогда наша задача **сводится к** задаче нахождения **эйлерова пути** в этом графе.

Реализация

Приведенная ниже программа ищет и выводит эйлеров цикл или путь в графе, или выводит -1, если его не существует.

Сначала программа проверяет степени вершин: если вершин с нечётной степенью нет, то в графе есть эйлеров цикл, если есть 2 вершины с нечётной степенью, то в графе есть только эйлеров путь (эйлерова цикла нет), если же таких вершин больше 2, то в графе нет ни эйлерова цикла, ни эйлерова пути. Чтобы найти эйлеров путь (не цикл), поступим таким образом: если V1 и V2 - это две вершины нечётной степени, то просто добавим ребро (V1,V2), в полученном графе найдём эйлеров цикл (он, очевидно, будет существовать), а затем удалим из ответа "фиктивное" ребро (V1,V2). Эйлеров цикл будем искать в точности так, как описано выше (нерекурсивной версией), и заодно по окончании этого алгоритма проверим, связный был граф или нет (если граф был не связный, то по

окончании работы алгоритма в графе останутся некоторые рёбра, и в этом случае нам надо вывести -1).
Наконец, программа учитывает, что в графе могут быть изолированные вершины.

```
int main() {

        int n;
        vector < vector<int> > g (n, vector<int> (n));
        ... чтение графа в матрицу смежности ...

        vector<int> deg (n);
        for (int i=0; i<n; ++i)
                for (int j=0; j<n; ++j)
                        deg[i] += g[i][j];

        int first = 0;
        while (!deg[first])  ++first;

        int v1 = -1,  v2 = -1;
        bool bad = false;
        for (int i=0; i<n; ++i)
                if (deg[i] & 1)
                        if (v1 == -1)
                                v1 = i;
                        else if (v2 == -1)
                                v2 = i;
                        else
                                bad = true;

        if (v1 != -1)
                ++g[v1][v2],  ++g[v2][v1];

        stack<int> st;
        st.push (first);
        vector<int> res;
        while (!st.empty())
        {
                int v = st.top();
                int i;
                for (i=0; i<n; ++i)
                        if (g[v][i])
                                break;
                if (i == n)
                {
                        res.push_back (v);
                        st.pop();
                }
                else
                {
                        --g[v][i];
                        --g[i][v];
                        st.push (i);
                }
        }

        if (v1 != -1)
                for (size_t i=0; i+1<res.size(); ++i)
                        if (res[i] == v1 && res[i+1] == v2 || res[i] == v2
&& res[i+1] == v1)
                        {
                                vector<int> res2;
                                for (size_t j=i+1; j<res.size(); ++j)
                                        res2.push_back (res[j]);
                                for (size_t j=1; j<=i; ++j)
                                        res2.push_back (res[j]);
                                res = res2;
                                break;
                        }

        for (int i=0; i<n; ++i)
                for (int j=0; j<n; ++j)
                        if (g[i][j])
```

```
                                bad = true;

        if (bad)
                puts ("-1");
        else
                for (size_t i=0; i<res.size(); ++i)
                        printf ("%d ", res[i]+1);

}
```

Проверка графа на ацикличность и нахождение цикла

Пусть дан ориентированный или неориентированный граф без петель и кратных рёбер. Требуется проверить, является ли он ациклическим, а если не является, то найти любой цикл.

Решим эту задачу с помощью поиска в глубину за O (M).

Алгоритм

Произведём серию поисков в глубину в графе. Т.е. из каждой вершины, в которую мы ещё ни разу не приходили, запустим поиск в глубину, который при входе в вершину будет красить её в серый цвет, а при выходе - в чёрный. И если поиск в глубину пытается пойти в серую вершину, то это означает, что мы нашли цикл (если граф неориентированный, то случаи, когда поиск в глубину из какой-то вершины пытается пойти в предка, не считаются).

Сам цикл можно восстановить проходом по массиву предков.

Реализация

Здесь приведена реализация для случая ориентированного графа.

```
int n;
vector < vector<int> > g;
vector<char> cl;
vector<int> p;
int cycle_st, cycle_end;

bool dfs (int v) {
        cl[v] = 1;
        for (size_t i=0; i<g[v].size(); ++i) {
                int to = g[v][i];
                if (cl[to] == 0) {
                        p[to] = v;
                        if (dfs (to))  return true;
                }
                else if (cl[to] == 1) {
                        cycle_end = v;
                        cycle_st = to;
                        return true;
                }
        }
        cl[v] = 2;
        return false;
}

int main() {
        ... чтение графа ...

        p.assign (n, -1);
        cl.assign (n, 0);
        cycle_st = -1;
        for (int i=0; i<n; ++i)
                if (dfs (i))
                        break;

        if (cycle_st == -1)
                puts ("Acyclic");
        else {
                puts ("Cyclic");
                vector<int> cycle;
                cycle.push_back (cycle_st);
                for (int v=cycle_end; v!=cycle_st; v=p[v])
                        cycle.push_back (v);
                cycle.push_back (cycle_st);
                reverse (cycle.begin(), cycle.end());
                for (size_t i=0; i<cycle.size(); ++i)
```

```
                        printf ("%d ", cycle[i]+1);
        }
}
```

Наименьший общий предок. Нахождение за O (sqrt (N)) и O (log N) с препроцессингом O (N)

Пусть дано дерево G. На вход поступают запросы вида (V1, V2), для каждого запроса требуется найти их наименьшего общего предка, т.е. вершину V, которая лежит на пути от корня до V1, на пути от корня до V2, и из всех таких вершин следует выбирать самую нижнюю. Иными словами, искомая вершина V - предок и V1, и V2, и среди всех таких общих предков выбирается нижний. Очевидно, что наименьший общий предок вершин V1 и V2 - это их общий предок, лежащий на кратчайшем пути из V1 в V2. В частности, например, если V1 является предком V2, то V1 является их наименьшим общим предком.

На английском эта задача называется задачей LCA - Least Common Ancestor.

Идея алгоритма

Перед тем, как отвечать на запросы, выполним так называемый **препроцессинг**. Запустим обход в глубину из корня, который будет строить список посещения вершин Order (текущая вершина добавляется в список при входе в эту вершину, а также после каждого возвращения из её сына), нетрудно заметить, что итоговый размер этого списка будет O (N). И построим массив First[1..N], в котором для каждой вершины будет указана позиция в массиве Order, в которой стоит эта вершина, т.е. Order[First[I]] = I для всех I. Также с помощью поиска в глубину найдём высоту каждой вершины (расстояние от корня до неё) - H[1..N].

Как теперь отвечать на запросы? Пусть имеется текущий запрос - пара вершин V1 и V2. Рассмотрим список Order между индексами First[V1] и First[V2]. Нетрудно заметить, что в этом диапазоне будет находиться и искомое LCA (V1, V2), а также множество других вершин. Однако LCA (V1, V2) будет отличаться от остальных вершин тем, что это будет вершина с наименьшей высотой.

Таким образом, чтобы ответить на запрос, нам нужно просто **найти вершину с наименьшей высотой** в массиве Order в диапазоне между First[V1] и First[V2]. Таким образом, **задача** `LCA` **сводится к задаче** `RMQ` ("минимум на отрезке"). А последняя задача решается с помощью структур данных (см. задача RMQ).

Если использовать `sqrt`-**декомпозицию**, то можно получить решение, отвечающее на запрос за `O (sqrt (N))` и выполняющее препроцессинг за O (N).

Если использовать **дерево отрезков**, то можно получить решение, отвечающее на запрос за `O (log (N))` и выполняющее препроцессинг за O (N).

Реализация

Здесь будет приведена готовая реализация LCA с использованием дерева отрезков:

```
typedef vector < vector<int> > graph;
typedef vector<int>::const_iterator const_graph_iter;


vector<int> lca_h, lca_dfs_list, lca_first, lca_tree;
vector<char> lca_dfs_used;

void lca_dfs (const graph & g, int v, int h = 1)
{
        lca_dfs_used[v] = true;
        lca_h[v] = h;
        lca_dfs_list.push_back (v);
        for (const_graph_iter i = g[v].begin(); i != g[v].end(); ++i)
                if (!lca_dfs_used[*i])
                {
                        lca_dfs (g, *i, h+1);
                        lca_dfs_list.push_back (v);
                }
}

void lca_build_tree (int i, int l, int r)
{
        if (l == r)
                lca_tree[i] = lca_dfs_list[l];
        else
        {
                int m = (l + r) >> 1;
```

```
                lca_build_tree (i+i, l, m);
                lca_build_tree (i+i+1, m+1, r);
                if (lca_h[lca_tree[i+i]] < lca_h[lca_tree[i+i+1]])
                        lca_tree[i] = lca_tree[i+i];
                else
                        lca_tree[i] = lca_tree[i+i+1];
        }
}

void lca_prepare (const graph & g, int root)
{
        int n = (int) g.size();
        lca_h.resize (n);
        lca_dfs_list.reserve (n*2);
        lca_dfs_used.assign (n, 0);

        lca_dfs (g, root);

        int m = (int) lca_dfs_list.size();
        lca_tree.assign (lca_dfs_list.size() * 4 + 1, -1);
        lca_build_tree (1, 0, m-1);

        lca_first.assign (n, -1);
        for (int i = 0; i < m; ++i)
        {
                int v = lca_dfs_list[i];
                if (lca_first[v] == -1)
                        lca_first[v] = i;
        }
}

int lca_tree_min (int i, int sl, int sr, int l, int r)
{
        if (sl == l && sr == r)
                return lca_tree[i];
        int sm = (sl + sr) >> 1;
        if (r <= sm)
                return lca_tree_min (i+i, sl, sm, l, r);
        if (l > sm)
                return lca_tree_min (i+i+1, sm+1, sr, l, r);
        int ans1 = lca_tree_min (i+i, sl, sm, l, sm);
        int ans2 = lca_tree_min (i+i+1, sm+1, sr, sm+1, r);
        return lca_h[ans1] < lca_h[ans2] ? ans1 : ans2;
}

int lca (int a, int b)
{
        int left = lca_first[a],
                right = lca_first[b];
        if (left > right)  swap (left, right);
        return lca_tree_min (1, 0, (int)lca_dfs_list.size()-1, left, right);
}

int main()
{
        graph g;
        int root;
        ... чтение графа ...

        lca_prepare (g, root);

        for (;;)
        {
                int v1, v2; // поступил запрос
                int v = lca (v1, v2); // ответ на запрос
        }
}
```

Наименьший общий предок. Нахождение за O (log N) (метод двоичного подъёма)

Пусть дано дерево G. На вход поступают запросы вида (V1, V2), для каждого запроса требуется найти их наименьшего общего предка, т.е. вершину V, которая лежит на пути от корня до V1, на пути от корня до V2, и из всех таких вершин следует выбирать самую нижнюю. Иными словами, искомая вершина V - предок и V1, и V2, и среди всех таких общих предков выбирается нижний. Очевидно, что наименьший общий предок вершин V1 и V2 - это их общий предок, лежащий на кратчайшем пути из V1 в V2. В частности, например, если V1 является предком V2, то V1 является их наименьшим общим предком.

На английском эта задача называется задачей LCA - Least Common Ancestor.

Здесь будет рассмотрен алгоритм, который пишется намного быстрее, чем описанный здесь.

Асимптотика полученного алгоритма будет равна: препроцессинг за O (N log N) и ответ на каждый запрос за O (log N).

Алгоритм

Предпосчитаем для каждой вершины её 1-го предка, 2-го предка, 4-го, и т.д. Обозначим этот массив через P, т.е. P[i][j] - это 2^j-й предок вершины i, i = 1..N, j = 0..•logN•. Также для каждой вершины найдём времена захода в неё и выхода поиска в глубину (см. "Поиск в глубину") - это нам понадобится, чтобы определять за O (1), является ли одна вершина предком другой (не обязательно непосредственным). Такой препроцессинг можно выполнить за O (N log N).

Пусть теперь поступил очередной запрос - пара вершин (A,B). Сразу проверим, не является ли одна вершина предком другой - в таком случае она и является результатом. Если A не предок B, и B не предок A, то будем подниматься по предкам A, пока не найдём самую высокую (т.е. наиболее близкую к корню) вершину, которая ещё не является предком (не обязательно непосредственным) B (т.е. такую вершину X, что X не предок B, а P[X][0] - предок B). При этом находить эту вершину X будем за O (log N), пользуясь массивом P.

Опишем этот процесс подробнее. Пусть L = •logN•. Пусть сначала I = L. Если P[A][I] не является предком B, то присваиваем A = P[A][I], и уменьшаем I. Если же P[A][I] является предком B, то просто уменьшаем I. Очевидно, что когда I станет меньше нуля, вершина A как раз и будет являться искомой вершиной - т.е. такой, что A не предок B, но P[A][0] - предок B.

Теперь, очевидно, ответом на LCA будет являться P[A][0] - т.е. наименьшая вершина среди предков исходной вершины A, являющаяся также и предком B.

Асимптотика. Весь алгоритм ответа на запрос состоит из изменения I от L = •logN• до 0, а также проверки на каждом шаге за O(1), является ли одна вершина предком другой. Следовательно, на каждый запрос будет найден ответ за O (log N).

Реализация

```
int n, l;
vector < vector<int> > g;
vector<int> tin, tout;
int timer;
vector < vector<int> > up;

void dfs (int v, int p = 0) {
        tin[v] = ++timer;
        up[v][0] = p;
        for (int i=1; i<=l; ++i)
                up[v][i] = up[up[v][i-1]][i-1];
        for (size_t i=0; i<g[v].size(); ++i) {
                int to = g[v][i];
                if (to != p)
                        dfs (to, v);
        }
        tout[v] = ++timer;
}

bool upper (int a, int b) {
        return tin[a] <= tin[b] && tout[a] >= tout[b];
}

int lca (int a, int b) {
```

```
        if (upper (a, b))  return a;
        if (upper (b, a))  return b;
        for (int i=l; i>=0; --i)
                if (! upper (up[a][i], b))
                        a = up[a][i];
        return up[a][0];
}

int main() {

        ... чтение n и g ...

        tin.resize (n),  tout.resize (n),  up.resize (n);
        l = 1;
        while ((1<<l) <= n)  ++l;
        for (int i=0; i<n; ++i)  up[i].resize (l+1);
        dfs (0);

        for (;;) {
                int a, b; // текущий запрос
                int res = lca (a, b); // ответ на запрос
        }

}
```

Наименьший общий предок. Нахождение за O (1) с препроцессингом O (N) (алгоритм Фарах-Колтона и Бендера)

Пусть дано дерево G. На вход поступают запросы вида (V1, V2), для каждого запроса требуется найти их наименьшего общего предка, т.е. вершину V, которая лежит на пути от корня до V1, на пути от корня до V2, и из всех таких вершин следует выбирать самую нижнюю. Иными словами, искомая вершина V - предок и V1, и V2, и среди всех таких общих предков выбирается нижний. Очевидно, что наименьший общий предок вершин V1 и V2 - это их общий предок, лежащий на кратчайшем пути из V1 в V2. В частности, например, если V1 является предком V2, то V1 является их наименьшим общим предком.

На английском эта задача называется задачей LCA - Least Common Ancestor.

Описываемый здесь алгоритм Фарах-Колтона и Бендера (Farach-Colton, Bender) является асимптотически оптимальным, и при этом сравнительно простым (по сравнению с другими алгоритмами, например, Шибера-Вишкина).

Алгоритм

Воспользуемся классическим сведением задачи LCA **к задаче** RMQ (минимум на отрезке) (более подробно см. Наименьший общий предок. Нахождение за O (sqrt (N)) и O (log N) с препроцессингом O (N)). Научимся теперь решать задачу RMQ в данном частном случае с препроцессингом O (N) и O (1) на запрос.

Заметим, что задача RMQ, к которой мы свели задачу LCA, является весьма специфичной: любые два соседних элемента в массиве **отличаются ровно на единицу** (поскольку элементы массива - это не что иное как высоты вершин, посещаемых в порядке обхода, и мы либо идём в потомка, тогда следующий элемент будет на 1 больше, либо идём в предка, тогда следующий элемент будет на 1 меньше). Собственно алгоритм Фарах-Колтона и Бендера как раз и представляет собой решение такой задачи RMQ.

Обозначим через A массив, над которым выполняются запросы RMQ, а N - размер этого массива.

Построим сначала алгоритм, решающий эту задачу **с препроцессингом** O (N log N) **и** O (1) **на запрос**. Это сделать легко: создадим так называемую Sparse Table T[l,i], где каждый элемент T[l,i] равен минимуму A на промежутке [l; l+2^i). Очевидно, 0 <= i <= •log N•, и потому размер Sparse Table будет O (N log N). Построить её также легко за O (N log N), если заметить, что T[l,i] = min (T[l,i-1], T[l+2^{i-1},i-1]). Как теперь отвечать на каждый запрос RMQ за O (1)? Пусть поступил запрос (l,r), тогда ответом будет min (T[l,sz], T[r-2^{sz}+1,sz]), где sz - наибольшая степень двойки, не превосходящая r-l+1. Действительно, мы как бы берём отрезок (l,r) и покрываем его двумя отрезками длины 2^{sz} - один начинающийся в l, а другой заканчивающийся в r (причём эти отрезки перекрываются, что в данном случае нам нисколько не мешает). Чтобы действительно достигнуть асимптотики O (1) на запрос, мы должны предпосчитать значения sz для всех возможных длин от 1 до N.

Теперь опишем, **как улучшить** этот алгоритм до асимптотики O (N).

Разобьём массив A на блоки размером K = 0.5 $\log_2$ N. Для каждого блока посчитаем минимальный элемент в нём и его позицию (поскольку для решения задачи LCA нам важны не сами минимумы, а их позиции). Пусть B - это массив размером N / K, составленный из этих минимумов в каждом блоке. Построим по массиву B Sparse Table, как описано выше, при этом размер Sparse Table и время её построения будут равны:

```
N/K log N/K = (2N / log N) log (2N / log N) =
= (2N / log N) (1 + log (N / log N)) <= 2N / log N + 2N = O (N)
```

Теперь нам осталось только научиться быстро отвечать на запросы RMQ **внутри каждого блока**. В самом деле, если поступил запрос RMQ(l,r), то, если l и r находятся в разных блоках, то ответом будет минимум из следующих значений: минимум в блоке l, начиная с l и до конца блока, затем минимум в блоках после l и до r (не включительно), и наконец минимум в блоке r, от начала блока до r. На запрос "минимум в блоках" мы уже можем отвечать за O (1) с помощью Sparse Table, остались только запросы RMQ внутри блоков.

Здесь мы воспользуемся "+-1 свойством". Заметим, что, если внутри каждого блока от каждого его элемента отнять первый элемент, то все блоки будут однозначно определяться последовательностью длины K-1, состоящей из чисел +-1. Следовательно, количество различных блоков будет равно:

$$2^{K-1} = 2^{0.5 \log N - 1} = 0.5 \text{ sqrt}(N)$$

Итак, количество различных блоков будет O (sqrt (N)), и потому мы можем предпосчитать результаты RMQ внутри всех различных блоков за O (sqrt(N) K^2) = O (sqrt(N) $\log^2$ N) = O (N). С точки зрения реализации, мы можем каждый блок характеризовать битовой маской длины K-1 (которая, очевидно, поместится в стандартный тип int), и хранить предпосчитанные RMQ в некотором массиве R[mask,l,r] размера O (sqrt(N) $\log^2$ N).

Итак, мы научились предпосчитывать результаты RMQ внутри каждого блока, а также RMQ над самими блоками, всё в сумме за O (N), а отвечать на каждый запрос RMQ за O (1) - пользуясь только предвычисленными значениями, в худшем случае четырьмя: в блоке l, в блоке r, и на блоках между l и r не включительно.

Реализация

В начале программы указаны константы MAXN, LOG_MAXLIST и SQRT_MAXLIST, определяющие максимальное число вершин в графе, которые при необходимости надо увеличить.

```
const int MAXN = 100*1000;
const int MAXLIST = MAXN * 2;
const int LOG_MAXLIST = 18;
const int SQRT_MAXLIST = 447;
const int MAXBLOCKS = MAXLIST / ((LOG_MAXLIST+1)/2) + 1;

int n, root;
vector<int> g[MAXN];
int h[MAXN]; // vertex height
vector<int> a; // dfs list
int a_pos[MAXN]; // positions in dfs list
int block; // block size = 0.5 log A.size()
int bt[MAXBLOCKS][LOG_MAXLIST+1]; // sparse table on blocks (relative
minimum positions in blocks)
int bhash[MAXBLOCKS]; // block hashes
int brmq[SQRT_MAXLIST][LOG_MAXLIST/2][LOG_MAXLIST/2]; // rmq inside each
block, indexed by block hash
int log2[2*MAXN]; // precalced logarithms (floored values)

// walk graph
void dfs (int v, int curh) {
        h[v] = curh;
        a_pos[v] = (int)a.size();
        a.push_back (v);
        for (size_t i=0; i<g[v].size(); ++i)
                if (h[g[v][i]] == -1) {
                        dfs (g[v][i], curh+1);
                        a.push_back (v);
                }
}

int log (int n) {
        int res = 1;
        while (1<<res < n)  ++res;
        return res;
}

// compares two indices in a
inline int min_h (int i, int j) {
        return h[a[i]] < h[a[j]] ? i : j;
}

// O(N) preprocessing
void build_lca() {
        int sz = (int)a.size();
        block = (log(sz) + 1) / 2;
        int blocks = sz / block + (sz % block ? 1 : 0);

        // precalc in each block and build sparse table
        memset (bt, 255, sizeof bt);
        for (int i=0, bl=0, j=0; i<sz; ++i, ++j) {
                if (j == block)
                        j = 0,  ++bl;
                if (bt[bl][0] == -1 || min_h (i, bt[bl][0]) == i)
                        bt[bl][0] = i;
        }
        for (int j=1; j<=log(sz); ++j)
                for (int i=0; i<blocks; ++i) {
                        int ni = i + (1<<(j-1));
                        if (ni >= blocks)
                                bt[i][j] = bt[i][j-1];
```

```
                        else
                                bt[i][j] = min_h (bt[i][j-1], bt[ni][j-1]);
                }

        // calc hashes of blocks
        memset (bhash, 0, sizeof bhash);
        for (int i=0, bl=0, j=0; i<sz||j<block; ++i, ++j) {
                if (j == block)
                        j = 0,  ++bl;
                if (j > 0 && (i >= sz || min_h (i-1, i) == i-1))
                        bhash[bl] += 1<<(j-1);
        }

        // precalc RMQ inside each unique block
        memset (brmq, 255, sizeof brmq);
        for (int i=0; i<blocks; ++i) {
                int id = bhash[i];
                if (brmq[id][0][0] != -1)  continue;
                for (int l=0; l<block; ++l) {
                        brmq[id][l][l] = l;
                        for (int r=l+1; r<block; ++r) {
                                brmq[id][l][r] = brmq[id][l][r-1];
                                if (i*block+r < sz)
                                        brmq[id][l][r] =
                                                min_h (i*block+brmq[id][l]
[r], i*block+r) - i*block;
                        }
                }
        }

        // precalc logarithms
        for (int i=0, j=0; i<sz; ++i) {
                if (1<<(j+1) <= i)  ++j;
                log2[i] = j;
        }
}

// answers RMQ in block #bl [l;r] in O(1)
inline int lca_in_block (int bl, int l, int r) {
        return brmq[bhash[bl]][l][r] + bl*block;
}

// answers LCA in O(1)
int lca (int v1, int v2) {
        int l = a_pos[v1],  r = a_pos[v2];
        if (l > r)  swap (l, r);
        int bl = l/block,  br = r/block;
        if (bl == br)
                return a[lca_in_block(bl,l%block,r%block)];
        int ans1 = lca_in_block(bl,l%block,block-1);
        int ans2 = lca_in_block(br,0,r%block);
        int ans = min_h (ans1, ans2);
        if (bl < br - 1) {
                int pw2 = log2[br-bl-1];
                int ans3 = bt[bl+1][pw2];
                int ans4 = bt[br-(1<<pw2)][pw2];
                ans = min_h (ans, min_h (ans3, ans4));
        }
        return a[ans];
}
```

Задача RMQ (Range Minimum Query - минимум на отрезке). Решение за O (1) с препроцессингом O (N)

Дан массив A[1..N]. Поступают запросы вида (L, R), на каждый запрос требуется найти минимум в массиве A, начиная с позиции L и заканчивая позицией R. Массив A изменяться в процессе работы не может, т.е. здесь описано решение статической задачи RMQ.

Здесь описано асимтпотически оптимальное решение. Оно несколько стоит особняком от других алгоритмов решения RMQ, поскольку оно сильно отличается от них: оно сводит задачу RMQ к задаче LCA, а затем использует алгоритм Фарах-Колтона и Бендера, который сводит задачу LCA обратно к RMQ (но уже частного вида) и решает её.

Алгоритм

Построим по массиву A декартово дерево, где у каждой вершины ключом будет позиция i, а приоритетом - само число A [i] (предполагается, что в декартовом дереве приоритеты упорядочены от меньшего в корне к большим). Такое дерево можно построить за O (N). Тогда запрос RMQ(l,r) эквивалентен запросу LCA(l',r'), где l' - вершина, соответствующая элементу A[l], r' - соответствующая A[r]. Действительно, LCA найдёт вершину, которая по ключу находится между l' и r', т.е. по позиции в массиве A будет между l и r, и при этом вершину, наиболее близкую к корню, т.е. с наименьшим приоритетом, т.е. наименьшим значением.

Задачу LCA мы можем решать за O (1) с препроцессингом O (N) с помощью алгоритма Фарах-Колтона и Бендера, который, что интересно, сводит задачу LCA обратно к задаче RMQ, но уже частного вида.

Наименьший общий предок. Нахождение за $O(1)$ в оффлайн (алгоритм Тарьяна)

Дано дерево G с n вершинами и дано m запросов вида (a_i, b_i). Для каждого запроса (a_i, b_i) требуется найти наименьшего общего предка вершин a_i и b_i, т.е. такую вершину c_i, которая наиболее удалена от корня дерева, и при этом является предком обеих вершин a_i и b_i.

Мы рассматриваем задачу в режиме оффлайн, т.е. считая, что все запросы известны заранее. Описываемый ниже алгоритм позволяет ответить на все m запросов за суммарное время $O(n + m)$, т.е. при достаточно большом m за $O(1)$ на запрос.

Алгоритм Тарьяна

Основой для алгоритма является структура данных "Система непересекающихся множеств", которая и была изобретена Тарьяном (Tarjan).

Алгоритм фактически представляет собой обход в глубину из корня дерева, в процессе которого постепенно находятся ответы на запросы. А именно, ответ на запрос (v, u) находится, когда обход в глубину находится в вершине u, а вершина v уже была посещена, или наоборот.

Итак, пусть обход в глубину находится в вершине v (и уже были выполнены переходы в её сыновей), и оказалось, что для какого-то запроса (v, u) вершина u уже была посещена обходом в глубину. Научимся тогда находить LCA этих двух вершин.

Заметим, что $\mathrm{LCA}(v, u)$ является либо самой вершиной v, либо одним из её предков. Получается, нам надо найти самую нижнюю вершину среди предков v (включая её саму), для которой вершина u является потомком. Заметим, что при фиксированном v по такому признаку (т.е. какой наименьший предок v является и предком какой-то вершины) вершины дерева дерева распадаются на совокупность непересекающихся классов. Для каждого предка $p \neq v$ вершины v её класс содержит саму эту вершину, а также все поддеревья с корнями в тех её сыновьях, которые лежат "слева" от пути до v (т.е. которые были обработаны ранее, чем была достигнута v).

Нам надо научиться эффективно поддерживать все эти классы, для чего мы и применим структуру данных "Система непересекающихся множеств". Каждому классу будет соответствовать в этой структуре множество, причём для представителя этого множества мы определим величину $\mathrm{ANCESTOR}$ — ту вершину p, которая и образует этот класс.

Рассмотрим подробно реализацию обхода в глубину. Пусть мы стоим в некоторой вершине v. Поместим её в отдельный класс в структуре непересекающихся множеств, $\mathrm{ANCESTOR}[v] = v$. Как обычно в обходе в глубину, перебираем все исходящие рёбра (v, to). Для каждого такого to мы сначала должны вызвать обход в глубину из этой вершины, а потом добавить эту вершину со всем её поддеревом в класс вершины v. Это реализуется операцией Union структуры данных "система непересекающихся множеств", с последующей установкой $\mathrm{ANCESTOR} = v$ для представителя множества (т.к. после объединения представитель класса мог измениться). Наконец, после обработки всех рёбер мы перебираем все запросы вида (v, u), и если u была помечена как посещённая обходом в глубину, то ответом на этот запрос будет вершина $\mathrm{LCA}(v, u) = \mathrm{ANCESTOR}[\mathrm{FindSet}(u)]$. Нетрудно заметить, что для каждого запроса это условие (что одна вершина запроса является текущей, а другая была посещена ранее) выполнится ровно один раз.

Оценим **асимптотику**. Она складывается из нескольких частей. Во-первых, это асимптотика обхода в глубину, которая в данном случае составляет $O(n)$. Во-вторых, это операции по объединению множеств, которые в сумме для всех разумных n затрачивают $O(n)$ операций. В-третьих, это для каждого запроса проверка условия (два раза на запрос) и определение результата (один раз на запрос), каждое, опять же, для всех разумных n выполняется за $O(1)$. Итоговая асимптотика получается $O(n + m)$, что означает для достаточно больших m ($n = O(m)$) ответ за $O(1)$ на один запрос.

Реализация

Приведём полную реализацию данного алгоритма, включая слегка изменённую (с поддержкой $\mathrm{ANCESTOR}$) реализацию системы пересекающихся множеств (рандомизированный варианта).

```
const int MAXN = максимальное число вершин в графе;
vector<int> g[MAXN], q[MAXN]; // граф и все запросы
int dsu[MAXN], ancestor[MAXN];
bool u[MAXN];
```

```
int dsu_get (int v) {
        return v == dsu[v] ? v : dsu[v] = dsu_get (dsu[v]);
}

void dsu_unite (int a, int b, int new_ancestor) {
        a = dsu_get (a),  b = dsu_get (b);
        if (rand() & 1)  swap (a, b);
        dsu[a] = b,  ancestor[b] = new_ancestor;
}

void dfs (int v) {
        dsu[v] = v,  ancestor[v] = v;
        u[v] = true;
        for (size_t i=0; i<g[v].size(); ++i)
                if (!u[g[v][i]]) {
                        dfs (g[v][i]);
                        dsu_unite (v, g[v][i], v);
                }
        for (size_t i=0; i<q[v].size(); ++i)
                if (u[q[v][i]]) {
                        printf ("%d %d -> %d\n", v+1, q[v][i]+1,
                                ancestor[ dsu_get(q[v][i]) ]+1);
}

int main() {
        ... чтение графа ...

        // чтение запросов
        for (;;) {
                int a, b = ...; // очередной запрос
                --a, --b;
                q[a].push_back (b);
                q[b].push_back (a);
        }

        // обход в глубину и ответ на запросы
        dfs (0);
}
```

Максимальный поток методом Эдмондса-Карпа за O (N M^2)

Пусть дан граф G, в котором выделены две вершины: исток S и сток T, а у каждого ребра определена пропускная способность $C_{u,v}$. Поток F можно представить как поток вещества, которое могло бы пройти по сети от истока к стоку, если рассматривать граф как сеть труб с некоторыми пропускными способностями. Т.е. поток - функция $F_{u,v}$, определённая на множестве рёбер графа.

Задача заключается в нахождении максимального потока. Здесь будет рассмотрен метод Эдмондса-Карпа, работающий за O (N M^2), или (другая оценка) O (F M), где F - величина искомого потока. Алгоритм был предложен в 1972 году.

Алгоритм

Остаточной пропускной способностью называется пропускная способность ребра за вычетом текущего потока вдоль этого ребра. При этом надо помнить, что если некоторый поток протекает по ориентированному ребру, то возникает так называемое обратное ребро (направленное в обратную сторону), которое будет иметь нулевую пропускную способность, и по которому будет протекать тот же по величине поток, но со знаком минус. Если же ребро было неориентированным, то оно как бы распадается на два ориентированных ребра с одинаковой пропускной способностью, и каждое из этих рёбер является обратным для другого (если по одному протекает поток F, то по другому протекает -F).

Общая схема **алгоритма Эдмондса-Карпа** такова. Сначала полагаем поток равным нулю. Затем ищем дополняющий путь, т.е. простой путь из S в T по тем рёбрам, у которых остаточная пропускная способность строго положительна. Если дополняющий путь был найден, то производится увеличение текущего потока вдоль этого пути. Если же пути не было найдено, то текущий поток является максимальным. Для поиска дополняющего пути может использоваться как Обход в ширину, так и Обход в глубину.

Рассмотрим более точно процедуру увеличения потока. Пусть мы нашли некоторый дополняющий путь, тогда пусть C - наименьшая из остаточных пропускных способностей рёбер этого пути. Процедура увеличения потока заключается в следующем: для каждого ребра (u, v) дополняющего пути выполним: $F_{u,v}$ += C, а $F_{v,u}$ = - $F_{u,v}$ (или, что то же самое, $F_{v,u}$ -= C).

Величиной потока будет сумма всех неотрицательных величин $F_{S,v}$, где v - любая вершина, соединённая с истоком.

Реализация

```
const int inf = 1000*1000*1000;


typedef vector<int> graf_line;
typedef vector<graf_line> graf;

typedef vector<int> vint;
typedef vector<vint> vvint;


int main()
{
        int n;
        cin >> n;
        vvint c (n, vint(n));
        for (int i=0; i<n; i++)
                for (int j=0; j<n; j++)
                        cin >> c[i][j];
        // исток - вершина 0, сток - вершина n-1

        vvint f (n, vint(n));
        for (;;)
        {

                vint from (n, -1);
                vint q (n);
                int h=0, t=0;
```

```
                q[t++] = 0;
                from[0] = 0;
                for (int cur; h<t;)
                {
                        cur = q[h++];
                        for (int v=0; v<n; v++)
                                if (from[v] == -1 &&
                                        c[cur][v]-f[cur][v] > 0)
                                {
                                        q[t++] = v;
                                        from[v] = cur;
                                }
                }

                if (from[n-1] == -1)
                        break;
                int cf = inf;
                for (int cur=n-1; cur!=0; )
                {
                        int prev = from[cur];
                        cf = min (cf, c[prev][cur]-f[prev][cur]);
                        cur = prev;
                }

                for (int cur=n-1; cur!=0; )
                {
                        int prev = from[cur];
                        f[prev][cur] += cf;
                        f[cur][prev] -= cf;
                        cur = prev;
                }

        }

        int flow = 0;
        for (int i=0; i<n; i++)
                if (c[0][i])
                        flow += f[0][i];

        cout << flow;

}
```

Максимальный поток методом Проталкивания предпотока за O (N^4)

Пусть дан граф G, в котором выделены две вершины: исток S и сток T, а у каждого ребра определена пропускная способность $C_{u,v}$. Поток F можно представить как поток вещества, которое могло бы пройти по сети от истока к стоку, если рассматривать граф как сеть труб с некоторыми пропускными способностями. Т.е. поток - функция $F_{u,v}$, определённая на множестве рёбер графа.

Задача заключается в нахождении максимального потока. Здесь будет рассмотрен метод Проталкивания предпотока, работающий за O (N^4), или, точнее, за O (N^2 M). Алгоритм был предложен Гольдбергом в 1985 году.

Алгоритм

Общая схема алгоритма такова. На каждом шаге будем рассматривать некоторый предпоток - т.е. функцию, которая по свойствам напоминает поток, но не обязательно удовлетворяет закону сохранения потока. На каждом шаге будем пытаться применить какую-либо из двух операций: проталкивание потока или поднятие вершины. Если на каком-то шаге станет невозможно применить какую-либо из двух операций, то мы нашли требуемый поток.

Для каждой вершины определена её высота H_u, причём $H_S = N$, $H_T = 0$, и для любого остаточного ребра (u, v) имеем $H_u <= H_v + 1$.

Для каждой вершины (кроме S) можно определить её избыток: $E_u = F_{V,u}$. Вершина с положительным избытком называется переполненной.

Операция проталкивания Push (u, v) применима, если вершина u переполнена, остаточная пропускная способность $Cf_{u,v} > 0$ и $H_u = H_v + 1$. Операция проталкивания заключается в максимальном увеличении потока из u в v, ограниченном избытком E_u и остаточной пропускной способностью $Cf_{u,v}$.

Операция поднятия Lift (u) поднимает переполненную вершину u на максимально допустимую высоту. Т.е. $H_u = 1 + \min \{ H_v \}$, где (u, v) - остаточное ребро.

Осталось только рассмотреть инициализацию потока. Нужно инициализировать только следующие значения: $F_{S,v} = C_{S,v}$, $F_{u,S} = - C_{u,S}$, остальные значения положить равными нулю.

Реализация

```
const int inf = 1000*1000*1000;


typedef vector<int> graf_line;
typedef vector<graf_line> graf;

typedef vector<int> vint;
typedef vector<vint> vvint;


void push (int u, int v, vvint & f, vint & e, const vvint & c)
{
        int d = min (e[u], c[u][v] - f[u][v]);
        f[u][v] += d;
        f[v][u] = - f[u][v];
        e[u] -= d;
        e[v] += d;
}

void lift (int u, vint & h, const vvint & f, const vvint & c)
{
        int d = inf;
        for (int i = 0; i < (int)f.size(); i++)
                if (c[u][i]-f[u][i] > 0)
                        d = min (d, h[i]);
```

```
        if (d == inf)
                return;
        h[u] = d + 1;
}


int main()
{
        int n;
        cin >> n;
        vvint c (n, vint(n));
        for (int i=0; i<n; i++)
                for (int j=0; j<n; j++)
                        cin >> c[i][j];
        // исток - вершина 0, сток - вершина n-1

        vvint f (n, vint(n));
        for (int i=1; i<n; i++)
        {
                f[0][i] = c[0][i];
                f[i][0] = -c[0][i];
        }

        vint h (n);
        h[0] = n;

        vint e (n);
        for (int i=1; i<n; i++)
                e[i] = f[0][i];

        for ( ; ; )
        {
                int i;
                for (i=1; i<n-1; i++)
                        if (e[i] > 0)
                                break;
                if (i == n-1)
                        break;

                int j;
                for (j=0; j<n; j++)
                        if (c[i][j]-f[i][j] > 0 && h[i]==h[j]+1)
                                break;
                if (j < n)
                        push (i, j, f, e, c);
                else
                        lift (i, h, f, c);
        }

        int flow = 0;
        for (int i=0; i<n; i++)
                if (c[0][i])
                        flow += f[0][i];

        cout << max(flow,0);

}
```

Модификация метода Проталкивания предпотока для нахождения максимального потока за O (N^3)

Предполагается, что вы уже прочитали Метод Проталкивания предпотока нахождения максимального потока за O (N^4).

Описание

Модификация чрезвычайно проста: на каждой итерации среди всех переполненных вершин мы выбираем только те вершины, которые имеют **набольшую высоту**, и применяем проталкивание/поднятие только к этим вершинам. Более того, для выбора вершин с наибольшей высотой нам не понадобятся никакие структуры данных, достаточно просто хранить список вершин с наибольшей высотой и просто пересчитывать его, если все вершины из этого списка были обработаны (тогда в список добавятся вершины с уже меньшей высотой), а при появлении новой переполненной вершины с большей высотой, чем в списке, очищать список и добавлять вершину в список.

Несмотря на простоту, эта модификация позволяет снизить асимптотику на целый порядок. Если быть точным, асимптотика получившего алгоритма равна O ($N\ M + N^2\ sqrt\ (M)$), что в худшем случае составляет O (N^3).

Эта модификация была предложена Черияном (Cheriyan) и Махешвари (Maheshvari) в 1989 г.

Реализация

Здесь приведена готовая реализация этого алгоритма.

Отличие от обычного алгоритма проталкивания - только в наличии массива maxh, в котором будут храниться номера переполненных вершин с максимальной высотой. Размер массива указан в переменной sz. Если на какой-то итерации оказывается, что этот массив пустой (sz==0), то мы заполняем его (просто проходя по всем вершинам); если после этого массив по-прежнему пустой, то переполненных вершин нет, и алгоритм останавливается. Иначе мы проходим по вершинам в этом списке, применяя к ним проталкивание или поднятие. После выполнения операции проталкивания текущая вершина может перестать быть переполненной, в этом случае удаляем её из списка maxh. Если после какой-то операции поднятия высота текущей вершины становится больше высоты вершин в списке maxh, то мы очищаем список (sz=0), и сразу переходим к следующей итерации алгоритма проталкивания (на которой будет построен новый список maxh).

```
const int INF = 1000*1000*1000;

int main() {

        int n;
        vector < vector<int> > c (n, vector<int> (n));
        int s, t;
        ... чтение n, c, s, t ...

        vector<int> e (n);
        vector<int> h (n);
        h[s] = n-1;
        vector < vector<int> > f (n, vector<int> (n));

        for (int i=0; i<n; ++i) {
                f[s][i] = c[s][i];
                f[i][s] = -f[s][i];
                e[i] = c[s][i];
        }

        vector<int> maxh (n);
        int sz = 0;
        for (;;) {
                if (!sz)
                        for (int i=0; i<n; ++i)
                                if (i != s && i != t && e[i] > 0) {
                                        if (sz && h[i] > h[maxh[0]])
                                                sz = 0;
                                        if (!sz || h[i] == h[maxh[0]])
                                                maxh[sz++] = i;
```

```
                                }
                        if (!sz)  break;
                        while (sz) {
                                int i = maxh[sz-1];
                                bool pushed = false;
                                for (int j=0; j<n && e[i]; ++j)
                                        if (c[i][j]-f[i][j] > 0 && h[i] == h[j]+1) {
                                                pushed = true;
                                                int addf = min (c[i][j]-f[i][j], e[i]);
                                                f[i][j] += addf,  f[j][i] -= addf;
                                                e[i] -= addf,  e[j] += addf;
                                                if (e[i] == 0)  --sz;
                                        }
                                if (!pushed) {
                                        h[i] = INF;
                                        for (int j=0; j<n; ++j)
                                                if (c[i][j]-f[i][j] > 0 && h[j]+1 <
h[i])
                                                        h[i] = h[j]+1;
                                        if (h[i] > h[maxh[0]]) {
                                                sz = 0;
                                                break;
                                        }
                                }
                        }
        }

        ... вывод потока f ...

}
```

Нахождение потока в графе, в котором у каждого ребра указано минимальное и максимальное значение потока

Пусть дан граф G, в котором для каждого ребра помимо пропускной способности (максимального значения потока вдоль этого ребра) указано и минимальное значение потока, который должен проходить по этому ребру.

Здесь мы рассмотрим две задачи: 1) требуется найти произвольный поток, удовлетворяющий всем ограничениям, и 2) требуется найти минимальный поток, удовлетворяющий всем ограничениям.

Решение задачи 1

Обозначим через L_i минимальную величину потока, которая может проходить по i-му ребру, а через R_i - его максимальная величина.

Произведём в графе следующие **изменения**. Добавим новый исток S' и сток T'. Рассмотрим все рёбра, у которых L_i отлично от нуля. Пусть i - номер такого ребра. Пусть концы этого ребра (ориентированного) - это вершины A_i и B_i. Добавим ребро (S', B_i), у которого L = 0, R = L_i, добавим ребро (A_i, T'), у которого L = 0, R = L_i, а у самого i-го ребра положим $R_i = R_i - L_i$, а $L_i = 0$. Наконец, добавим в граф ребро из T в S (старых стока и истока), у которого L = 0, R = INF.

После выполнения этих преобразований все рёбра графа будут иметь $L_i = 0$, т.е. мы свели эту задачу к обычной задаче нахождения максимального потока (но уже в модифицированном графе с новыми истоком и стоком) (чтобы понять, почему именно максимального - читайте нижеследующее объяснение).

Корректность этих преобразований понять сложнее. Неформальное **объяснение** такое. Каждое ребро, у которого L_i отлично от нуля, мы заменяем на два ребра: одно с пропускной способностью L_i, а другое - с R_i-L_i. Нам требуется найти поток, который бы обязательно насытил первое ребро из этой пары (т.е. поток вдоль этого ребра должен быть равен L_i); второе ребро нас волнует меньше - поток вдоль него может быть любым, лишь бы он не превосходил его пропускной способности. Итак, нам требуется найти такой поток, который бы обязательно насытил некоторое множество рёбер. Рассмотрим каждое такое ребро, и выполним такую операцию: подведём к его концу ребро из нового истока S', подведём ребро из его начала к стоку T', само ребро удалим, а из старого стока T к старому истоку S проведём ребро бесконечной пропускной способности. Этими действиями мы проимитируем тот факт, что это ребро насыщено - из ребра будет вытекать L_i единиц потока (мы имитируем это с помощью нового истока, который подаёт на конец ребра нужное количество потока), а втекать в него будет опять же L_i единиц потока (но вместо ребра этот поток попадёт в новый сток). Поток из нового истока протекает по одной части графа, дотекает до старого стока T, из него протекает в старый исток S, затем течёт по другой части графа, и наконец приходит к началу нашего ребра, и попадает в новый сток T'. Т.е., если мы найдём в этом модифицированном графе максимальный поток (и в сток попадёт нужное количество потока, т.е. сумма всех значений L_i - иначе величина потока будет меньше, и ответа попросту не существует), то мы одновременно найдём поток в исходном графе, который будет удовлетворять все ограничениям минимума, и, разумеется, всем ограничениям максимума.

Решение задачи 2

Заметим, что по ребру из старого стока в старый исток с пропускной способностью INF протекает весь старый поток, т. е. пропускная способность этого ребра влияет на величину старого потока. При достаточно большой величине пропускной способности этого ребра (т.е. INF) старый поток ничем не ограничен. Если мы будем уменьшать пропускную способность, то и, начиная с некоторого момента, будет уменьшаться и величина старого потока. Но при слишком малом значении величина потока станет недостаточной, чтобы обеспечить выполнение ограничений (на минимальное значение потока вдоль рёбер). Очевидно, здесь можно применить **бинарный поиск по значению** INF, и найти такое её наименьшее значение, при котором все ограничения ещё будут удовлетворяться, но старый поток будет иметь минимальное значение.

Поток минимальной стоимости (min-cost-flow). Алгоритм увеличивающих путей

Дана сеть G, состоящая из N вершин и M рёбер. У каждого ребра (вообще говоря, ориентированному, но по этому поводу см. ниже) указана пропускная способность (целое неотрицательное число) и стоимость единицы потока вдоль этого ребра (некоторое целое число). В графе указан исток S и сток T. Даётся некоторая величина K потока, требуется найти поток этой величины, причём среди всех потоков этой величины выбрать поток с наименьшей стоимостью ("задача min-cost-flow").

Иногда задачу ставят немного по-другому: требуется найти максимальный поток наименьшей стоимости ("задача min-cost-max-flow").

Обе эти задачи достаточно эффективно решаются описанным ниже алгоритмом увеличивающих путей.

Описание

Алгоритм очень похож на алгоритм Эдмондса-Карпа вычисления максимального потока.

Простейший случай

Рассмотрим для начала простейший случай, когда граф - ориентированный, и между любой парой вершин не более одного ребра (если есть ребро (i,j), то ребра (j,i) быть не должно).

Пусть U_{ij} - пропускная способность ребра (i,j), если это ребро существует. Пусть C_{ij} - стоимость единицы потока вдоль ребра (i,j). Пусть F_{ij} - величина потока вдоль ребра (i,j), изначально все величины потоков равны нулю.

Модифицируем сеть следующим образом: для каждого ребра (i,j) добавим в сеть так называемое **обратное** ребро (j,i) с пропускной способностью $U_{ji} = 0$ и стоимостью $C_{ji} = - C_{ij}$. Поскольку, по нашему предположению, ребра (j,i) до этого в сети не было, то модифицированная таким образом сеть по-прежнему не будет мультиграфом. Кроме того, на всём протяжении работы алгоритма будем поддерживать верным условие: $F_{ji} = - F_{ij}$.

Определим **остаточную сеть** для некоторого зафиксированного потока F следующим образом (собственно, так же, как и в алгоритме Форда-Фалкерсона): остаточной сети принадлежат только ненасыщенные рёбра (т.е. у которых $F_{ij} < U_{ij}$), а остаточную пропускную способность каждого такого ребра как $UPI_{ij} = U_{ij} - F_{ij}$.

Собственно **алгоритм** min-cost-flow заключается в следующем. На каждой итерации алгоритма находим кратчайший путь в остаточной сети из S в T (кратчайший относительно стоимостей C_{ij}). Если путь не был найден, то алгоритм завершается, поток F - искомый. Если же путь был найден, то мы увеличиваем поток вдоль него настолько, насколько это возможно (т.е. проходим вдоль этого пути, находим минимальную остаточную пропускную способность MIN_UPI среди рёбер этого пути, и затем увеличиваем поток вдоль каждого ребра пути на величину MIN_UPI, не забывая уменьшать на такую же величину поток вдоль обратных рёбер). Если в какой-то момент величина потока достигла величины K (данной нам по условию величины потока), то мы также останавливаем алгоритм (следует учесть, что тогда на последней итерации алгоритма при увеличении потока вдоль пути нужно увеличивать поток на такую величину, чтобы итоговый поток не превзошёл K, но это выполнить легко).

Нетрудно заметить, что если положить K равным бесконечности, то алгоритм найдёт максимальный поток минимальной стоимости, т.е. один и тот же алгоритм без изменений решает обе задачи min-cost-flow и min-cost-max-flow.

Случай неориентированных графов, мультиграфов

Случай неориентированных графов и мультиграфов в концептуальном плане ничем не отличается от вышеописанного, поэтому собственно алгоритм будет работать и на таких графах. Однако возникают некоторые сложности в реализации, на которые следует обратить внимание.

Неориентированное ребро (i,j) - это фактически два ориентированных ребра (i,j) и (j,i) с одинаковыми пропускными способностями и стоимостями. Поскольку вышеописанный алгоритм min-cost-flow требует для каждого неориентированного ребра создать обратное ему ребро, то в итоге получается, что неориентированное ребро расщепляется на 4 ориентированных ребра, и мы фактически получаем случай **мультиграфа**.

Какие проблемы вызывают **кратные рёбра**? Во-первых, поток по каждому из кратных рёбер должен сохраняться отдельно. Во-вторых, при поиске кратчайшего пути нужно учитывать, что важно то, какое именно из кратных рёбер выбрать при восстановлении пути по предкам. Т.е. вместо обычного массива предков для каждой вершины мы должны хранить вершину-предка и номер ребра, по котором мы из неё пришли. В-третьих, при увеличении потока вдоль некоторого ребра нужно, согласно алгоритму, уменьшить поток вдоль обратного ребра. Поскольку у нас могут быть кратные рёбра, то придётся для каждого ребра хранить номер ребра, обратного ему.

Других сложностей с неориентированными графами и мультиграфами нет.

Анализ времени работы

По аналогии с анализом алгоритма Эдмондса-Карпа, мы получаем такую оценку: O (N M) * T (N, M), где T (N, M) - время, необходимое для нахождения кратчайшего пути в графе с N вершинами и M рёбрами. Если это реализовать с помощью простейшего варианта алгоритма Дейкстры, то для всего алгоритма min-cost-flow получится оценка $O(N^3 M)$, правда, алгоритм Дейкстры придётся модифицировать, чтобы он работал на графах с отрицательными весами (это называется алгоритм Дейкстры с потенциалами).

Вместо этого можно использовать алгоритм Левита, который, хотя и асимптотически намного хуже, но на практике работает очень быстро (примерно за то же время, что и алгоритм Дейкстры).

Реализация

Здесь приведена реализация алгоритма min-cost-flow, базирующаяся на алгоритме Левита.

На вход алгоритма подаётся сеть (неориентированный мультиграф) с N вершинами и M рёбрами, и K - величина потока, который нужно найти. Алгоритм находит поток величины K минимальной стоимости, если такой существует. Иначе он находит поток максимальной величины минимальной стоимости.

В программе есть специальная функция для добавления ориентированного ребра. Если нужно добавить неориентированное ребро, то эту функцию нужно вызывать для каждого ребра (i,j) дважды: от (i,j) и от (j,i).

```
const int INF = 1000*1000*1000;

struct rib {
        int b, u, c, f;
        size_t back;
};

void add_rib (vector < vector<rib> > & g, int a, int b, int u, int c) {
        rib r1 = { b, u, c, 0, g[b].size() };
        rib r2 = { a, 0, -c, 0, g[a].size() };
        g[a].push_back (r1);
        g[b].push_back (r2);
}

int main()
{
        int n, m, k;
        vector < vector<rib> > g (n);
        int s, t;
        ... чтение графа ...

        int flow = 0,  cost = 0;
        while (flow < k) {
                vector<int> id (n, 0);
                vector<int> d (n, INF);
                vector<int> q (n);
                vector<int> p (n);
                vector<size_t> p_rib (n);
                int qh=0, qt=0;
                q[qt++] = s;
                d[s] = 0;
                while (qh != qt) {
                        int v = q[qh++];
                        id[v] = 2;
                        if (qh == n)  qh = 0;
                        for (size_t i=0; i<g[v].size(); ++i) {
                                rib & r = g[v][i];
                                if (r.f < r.u && d[v] + r.c < d[r.b]) {
                                        d[r.b] = d[v] + r.c;
                                        if (id[r.b] == 0) {
                                                q[qt++] = r.b;
                                                if (qt == n)  qt = 0;
                                        }
                                        else if (id[r.b] == 2) {
                                                if (--qh == -1)  qh = n-1;
                                                q[qh] = r.b;
                                        }
                                        id[r.b] = 1;
                                        p[r.b] = v;
```

```
                                        p_rib[r.b] = i;
                                }
                        }
                }
                if (d[t] == INF)  break;
                int addflow = k - flow;
                for (int v=t; v!=s; v=p[v]) {
                        int pv = p[v];  size_t pr = p_rib[v];
                        addflow = min (addflow, g[pv][pr].u - g[pv][pr].f);
                }
                for (int v=t; v!=s; v=p[v]) {
                        int pv = p[v];  size_t pr = p_rib[v],  r = g[pv]
[pr].back;
                        g[pv][pr].f += addflow;
                        g[v][r].f -= addflow;
                        cost += g[pv][pr].c * addflow;
                }
                flow += addflow;
        }

        ... вывод результата ...

}
```

Задача о назначениях. Решение с помощью min-cost-flow

Задача имеет две эквивалентные постановки:

- Дана квадратная матрица A[1..N,1..N]. Нужно выбрать в ней N элементов так, чтобы в каждой строке и столбце был выбран ровно один элемент, а сумма значений этих элементов была наименьшей.
- Имеется N заказов и N станков. Про каждый заказ известна стоимость его изготовления на каждом станке. На каждом станке можно выполнять только один заказ. Требуется распределить все заказы по станкам так, чтобы минимизировать суммарную стоимость.

Здесь мы рассмотрим решение задачи на основе алгоритма нахождения потока минимальной стоимости (min-cost-flow), решив задачу о назначениях за $O(N^5)$.

Описание

Построим двудольную сеть: имеется исток S, сток T, в первой доле находятся N вершин (соответствующие строкам матрицы или заказам), во второй - тоже N вершин (соответствующие столбцам матрицы или станкам). Между каждой вершиной i первой доли и каждой вершиной j второй доли проведём ребро с пропускной способностью 1 и стоимостью A_{ij}. От истока S проведём рёбра ко всем вершинам i первой доли с пропускной способностью 1 и стоимостью 0. От каждой вершины второй доли j к стоку T проведём ребро с пропускной способностью 1 и стоимостью 0.

Найдём в полученной сети максимальный поток минимальной стоимости. Очевидно, величина потока будет равна N. Далее, очевидно, что для каждой вершины i из первой доли найдётся ровно одна вершина j из второй доли, такая, что поток $F_{ij} = 1$. Наконец, очевидно, это взаимно однозначное соответствие между вершинами первой доли и вершинами второй доли является решением задачи (поскольку найденный поток имеет минимальную стоимость, то сумма стоимостей выбранных рёбер будет наименьшей из возможных, что и является критерием оптимальности).

Асимптотика этого решения задачи о назначениях зависит от того, каким алгоритмом производится поиск максимального потока минимальной стоимости. Асимптотика составит $O(N^3)$ при использовании алгоритма Дейкстры или $O(N^4)$ при использовании алгоритма Форда-Беллмана.

Реализация

Приведённая здесь реализация длинноватая, возможно, её можно значительно сократить.

```
typedef vector<int> vint;
typedef vector<vint> vvint;

const int INF = 1000*1000*1000;


int main()
{
        int n;
        vvint a (n, vint (n));
        ... чтение a ...

        int m = n * 2 + 2;
        vvint f (m, vint (m));
        int s = m-2, t = m-1;
        int cost = 0;
        for (;;)
        {
                vector<int> dist (m, INF);
                vector<int> p (m);
                vector<int> type (m, 2);
                deque<int> q;
                dist[s] = 0;
                p[s] = -1;
                type[s] = 1;
                q.push_back (s);
                for (; !q.empty(); )
                {
                        int v = q.front(); q.pop_front();
                        type[v] = 0;
```

```
                        if (v == s)
                        {
                                for (int i=0; i<n; ++i)
                                        if (f[s][i] == 0)
                                        {
                                                dist[i] = 0;
                                                p[i] = s;
                                                type[i] = 1;
                                                q.push_back (i);
                                        }
                        }
                        else
                        {
                                if (v < n)
                                {
                                        for (int j=n; j<n+n; ++j)
                                                if (f[v][j] < 1 && dist[j]
> dist[v] + a[v][j-n])
                                                {
                                                        dist[j] = dist[v] + a
[v][j-n];
                                                        p[j] = v;
                                                        if (type[j] == 0)
                                                                q.
push_front (j);
                                                        else if (type[j] == 2)
                                                                q.push_back (j);
                                                        type[j] = 1;
                                                }
                                }
                                else
                                {
                                        for (int j=0; j<n; ++j)
                                                if (f[v][j] < 0 && dist[j]
> dist[v] - a[j][v-n])
                                                {
                                                        dist[j] = dist[v] - a
[j][v-n];
                                                        p[j] = v;
                                                        if (type[j] == 0)
                                                                q.
push_front (j);
                                                        else if (type[j] == 2)
                                                                q.push_back (j);
                                                        type[j] = 1;
                                                }
                                }
                        }
                }

                int curcost = INF;
                for (int i=n; i<n+n; ++i)
                        if (f[i][t] == 0 && dist[i] < curcost)
                        {
                                curcost = dist[i];
                                p[t] = i;
                        }
                if (curcost == INF) break;
                cost += curcost;
                for (int cur=t; cur!=-1; cur=p[cur])
                {
                        int prev = p[cur];
                        if (prev!=-1)
                                f[cur][prev] = - (f[prev][cur] = 1);
                }

        }

        printf ("%d\n", cost);
        for (int i=0; i<n; ++i)
                for (int j=0; j<n; ++j)
```

```
                        if (f[i][j+n] == 1)
                                printf ("%d ", j+1);

}
```

Венгерский алгоритм решения задачи о назначениях

Постановка задачи о назначениях

Задача о назначениях ставится весьма естественно.

Приведём несколько **вариантов постановки** (как легко видеть, все они эквивалентны друг другу):

- Есть n рабочих и n заданий. Для каждого рабочего известно, сколько денег он запросит за выполнение того или иного задания. Каждый рабочий может взять себе только одно задание. Требуется распределить задания по рабочим так, чтобы минимизировать суммарные расходы.
- Дана матрица a размера $n \times n$. Требуется в каждой её строке выбрать по одному числу так, чтобы в любом столбце также было выбрано ровно по одному числу, и при этом сумма выбранных чисел была бы минимальной.
- Дана матрица a размера $n \times n$. Требуется найти такую перестановку p длины n, что величина $\sum a[i][p[i]]$ — минимальна.
- Дан полный двудольный граф с n вершинами; каждому ребру приписан некоторый вес. Требуется найти совершенное паросочетание минимального веса.

Отметим, что все приведённые выше постановки "**квадратны**": в них обе размерности всегда совпадают (и равны n). На практике часто встречаются аналогичные "**прямоугольные**" постановки, когда $n \neq m$, и надо выбрать $\min(n, m)$ элементов. Впрочем, как легко заметить, от "прямоугольной" задачи всегда можно перейти к "квадратной", добавив строки/столбцы с нулевыми/бесконечными значениями соответственно.

Также заметим, что по аналогии с поиском **минимального** решения также ставить задачу поиска **максимального** решения. Впрочем, эти две задачи эквивалентны друг другу: достаточно все веса умножить на -1.

Венгерский алгоритм

Историческая справка

Алгоритм был разработан и опубликован Гарольдом **Куном** (Harold Kuhn) в 1955 г. Сам Кун дал алгоритму название "венгерский", потому что он был в значительной степени основан на более ранних работах двух венгерских математиков: Денеша **Кёнига** (Dénes Kőnig) и Эйгена **Эгервари** (Jenő Egerváry).

В 1957 г. Джеймс **Манкрес** (James Munkres) показал, что этот алгоритм работает за (строго) полиномиальное время (т.е. за время порядка полинома от n, не зависящего от величины стоимостей).

Поэтому в литературе данный алгоритм известен не только как "венгерский", но и как "алгоритм Куна-Манкреса" или "алгоритм Манкреса".

Впрочем, недавно (в 2006 г.) выяснилось, что точно такой же алгоритм был изобретён **за век до Куна** немецким математиком Карлом Густавом **Якоби** (Carl Gustav Jacobi). Дело в том, что его работа "About the research of the order of a system of arbitrary ordinary differential equations", напечатанная посмертно в 1890 г., содержавшая помимо прочих результатов и полиномиальный алгоритм решения задачи о назначениях, была написана на латыни, а её публикация **прошла незамеченной** среди математиков.

Также стоит отметить, что первоначальный алгоритм Куна имел асимптотику $O(n^4)$, и лишь позже Джек **Эдмондс** (Jack Edmonds) и Ричард **Карп** (Richard Karp) (и независимо от них **Томидзава** (Tomizawa)) показали, каким образом улучшить его до асимптотики $O(n^3)$.

Построение алгоритма за $O(n^4)$

Сразу отметим во избежание неоднозначностей, что мы в основном рассматриваем здесь задачу о назначениях в матричной постановке (т.е. дана матрица a, и надо выбрать из неё n ячеек, находящихся в разных строках и столбцах). Индексацию массивов мы начинаем с единицы, т.е., например, матрица a имеет индексы $a[1 \ldots n][1 \ldots n]$.

Также мы будем считать, что все числа в матрице $a[][]$ **неотрицательны** (если это не так, то всегда можно перейти к неотрицательной матрице, прибавив ко всем числам некоторое число).

Назовём **потенциалом** два произвольных массива чисел $u[1 \dots n]$ и $v[1 \dots n]$ таких, что выполняется условие:

$$u[i] + v[j] \le a[i][j] \qquad (i = 1 \dots n, \quad j = 1 \dots n).$$

(Как видно, числа $u[i]$ соответствуют строкам, а числа $v[j]$ — столбцам матрицы.)

Назовём **значением f потенциала** сумму его чисел:

$$f = \sum_{i=1}^{n} u[i] + \sum_{i=1}^{n} v[i].$$

С одной стороны, легко заметить, что стоимость искомого решения sol **не меньше** значения любого потенциала:

$$sol \ge f.$$

(Доказательство. Искомое решение задачи представляет из себя n ячеек матрицы, и для каждой из них выполняется условие $u[i] + v[j] \le a[i][j]$. Поскольку все элементы находятся в разных строках и столбцах, то, суммируя эти неравенства по всем выбранным $a[i][j]$, в левой части неравенства получаем f, а в правой — sol, что и требовалось доказать.)

С другой стороны, оказывается, что всегда существует решение и потенциал, на которых это неравенство **обращается в равенство**. Венгерский алгоритм, описанный ниже, будет конструктивным доказательством этого факта. Пока же лишь обратим внимание на то, что если какое-либо решение имеет стоимость, равную по величине какому-либо потенциалу, то это решение — **оптимально**.

Зафиксируем некоторый потенциал. Назовём ребро (i, j) **жёстким**, если выполняется:

$$u[i] + v[j] = a[i][j].$$

Вспомним об альтернативной постановке задачи о назначениях, с помощью двудольного графа. Обозначим через H двудольный граф, составленный только из жёстких рёбер. Фактически, венгерский алгоритм поддерживает для текущего потенциала **максимальное по количеству рёбер паросочетание** M графа H: и как только это паросочетание станет содержать n рёбер, рёбра этого паросочетания и будут являться искомым оптимальным решением.

Перейдём непосредственно к **описанию алгоритма**.

- В начале алгоритма потенциал полагается равным нулю $u[i] = v[i] = 0$, и паросочетание M полагается пустым.
- Далее, на каждом шаге алгоритма мы пытаемся, не меняя потенциала, увеличить мощность текущего паросочетания M на единицу (напоминаем, паросочетание ищется в графе жёстких рёбер H).

  Для этого фактически используется обычный алгоритм Куна поиска максимального паросочетания в двудольных графах. Напомним здесь этот алгоритм.

  Все рёбра паросочетания M ориентируются по направлению от второй доли к первой, все остальные рёбра графа H ориентируются в противоположную сторону.

  Напомним (из терминологии поиска паросочетаний), что вершина называется насыщенной, если ей смежно ребро из текущего паросочетания. Вершина, которой не смежно ни одно ребро из текущего паросочетания, называется ненасыщенной. Путь нечётной длины, в котором первое ребро не принадлежит паросочетанию, а для всех последующих рёбер происходит чередование (принадлежит/не принадлежит) — называется увеличивающим путём.

  Из всех ненасыщенных вершин первой доли запускается обход в глубину/в ширину. Если в результате обхода удалось достигнуть ненасыщенной вершины второй доли, то это означает, что мы нашли увеличивающий путь из первой доли во вторую. Если прочередовать рёбра вдоль этого пути (т.е. первое ребро включить в паросочетание, второе исключить, третье включить, и т.д.), то тем самым мы увеличим мощность паросочетания на единицу.

  Если же увеличивающего пути не было, то это означает, что текущее паросочетание M — максимально в графе H, поэтому в таком случае переходим к следующему пункту.
- Если на текущем шаге не удалось увеличить мощность текущего паросочетания, то производится некий пересчёт потенциала таким образом, чтобы на следующих шагах появилось больше возможностей для увеличения паросочетания.

  Обозначим через Z_1 множество вершин первой доли, которые были посещены обходом алгоритма Куна при попытке поиска увеличивающей цепи; через Z_2 — множество посещённых вершин второй доли. Обозначим через R_1 множество ненасыщенных вершин первой доли, через R_2 — множество ненасыщенных вершин второй доли.

  Посчитаем величину Δ:

  $$\Delta = \min_{i \in Z_1, j \notin Z_2} \{a[i][j] - u[i] - v[j]\}.$$

  Эта величина строго положительна.

  (Доказательство. Предположим, что $\Delta = 0$. Тогда существует жёсткое ребро (i, j), причём $i \in Z_1$ и $j \notin Z_2$.

Из этого следует, что ребро (i, j) должно было быть ориентированным от второй доле к первой, т.е. это жёсткое ребро (i, j) должно входить в паросочетание M. Однако это невозможно, т.к. мы не могли попасть в насыщенную вершину i, кроме как пройдя по ребру из j в i. Пришли к противоречию, значит, $\Delta > 0$.)

Теперь **пересчитаем потенциал** таким образом: для всех вершин $i \in Z_1$ сделаем $u[i] += \Delta$, а для всех вершин $j \in Z_2$ — сделаем $v[j] -= \Delta$. Получившийся потенциал по-прежнему останется корректным потенциалом.

(Доказательство. Для этого надо показать, что по-прежнему для всех i и j выполняется: $u[i] + v[j] \le a[i][j]$. Для случаев, когда $i \in Z_1 \& j \in Z_2$ или $i \notin Z_1 \& j \notin Z_2$ — это так, поскольку для них сумма $u[i]$ и $v[j]$ не изменилась. Когда $i \notin Z_1 \& j \in Z_2$ неравенство только усилилось. Наконец, для случая $i \in Z_1 \& j \notin Z_2$ — хотя левая часть неравенства и увеличивается, но всё равно неравенство сохраняется, поскольку величина Δ, согласно её определению — это как раз максимальное увеличение, не приводящее к нарушению неравенства.)

Кроме того, старое паросочетание M из жёстких рёбер можно будет оставить, т.е. все рёбра паросочетания останутся жёсткими.

(Доказательство. Чтобы некоторое жёсткое ребро (i, j) перестало быть жёстким в результате изменения потенциала, надо, чтобы равенство $u[i] + v[j] = a[i][j]$ превратилось в неравенство $u[i] + v[j] < a[i][j]$. Однако левая часть могла уменьшиться только в одном случае: когда $i \notin Z_1 \& j \in Z_2$. Но раз $i \notin Z_1$, то это означает, что ребро (i, j) не могло быть ребром паросочетания, что и требовалось доказать.)

Наконец, чтобы показать, что изменения потенциала **не могут происходить бесконечно**, заметим, что при каждом таком изменении потенциала количество вершин, достижимых обходом, т.е. $|Z_1| + |Z_2|$, строго увеличивается. (При этом нельзя утверждать, что увеличивается количество жёстких рёбер.)

(Доказательство. Во-первых, любая вершина, которая была достижимой, достижимой и останется. В самом деле, если некоторая вершина достижима, то до неё есть некоторый путь из достижимых вершин, начинающийся в ненасыщенной вершине первой доли; а поскольку для рёбер вида $(i, j), i \in Z_1 \& j \in Z_2$ сумма $u[i] + v[j]$ не меняется, то весь этот путь сохранится и после изменения потенциала, что и требовалось доказать. Во-вторых, покажем, что в результате пересчёта потенциала появилась хотя бы одна новая достижимая вершина. Но это почти очевидно, если вернуться к определению Δ: в самом деле, если минимум был достигнут на ребре (i, j), то это ребро теперь станет жёстким, а, значит, вершина j станет достижимой благодаря этому ребру и вершине i.)

Таким образом, всего может происходить не более n пересчётов потенциала, прежде чем обнаружится увеличивающая цепочка и мощность паросочетания M будет увеличена.

Таким образом, рано или поздно будет найден потенциал, которому соответствует совершенное паросочетание M, являющееся ответом на задачу.

Если говорить об **асимптотике** алгоритма, то она составляет $O(n^4)$, поскольку всего должно произойти n увеличений паросочетания, перед каждым из которых происходит не более n пересчётов потенциала, каждый из которых выполняется за время $O(n^2)$.

Реализацию за $O(n^4)$ мы здесь приводить не будем, поскольку она всё равно получится не короче, чем описанная ниже реализация за $O(n^3)$.

Построение алгоритма за $O(n^3)$ ($O(n^2 m)$)

Научимся теперь реализовывать тот же алгоритм за асимптотику $O(n^3)$ (для прямоугольных задач $n \times m$ — $O(n^2 m)$).

Ключевая идея: теперь мы будем **добавлять в рассмотрение строки матрицы одну за одной**, а не рассматривать их все сразу. Таким образом, описанный выше алгоритм примет вид:

- Добавляем в рассмотрение очередную строку матрицы a.
- Пока нет увеличивающей цепи, начинающейся в этой строке, пересчитываем потенциал.
- Как только появляется увеличивающая цепь, чередуем паросочетание вдоль неё (включая тем самым последнюю строку в паросочетание), и переходим к началу (к рассмотрению следующей строки).

Чтобы достичь требуемой асимптотики, надо реализовать шаги 2-3, выполняющиеся для каждой строки матрицы, за время $O(n^2)$ (для прямоугольных задач — за $O(nm)$).

Для этого мы вспомним два факта, доказанных нами выше:

- При изменении потенциала вершины, которые были достижимы обходом Куна, достижимыми и останутся.
- Всего могло произойти лишь $O(n)$ пересчётов потенциала, прежде чем будет найдена увеличивающая цепь.

Отсюда вытекают **ключевые идеи**, позволяющие достичь требуемой асимптотики:

- Для проверки наличия увеличивающей цепочки нет необходимости запускать обход Куна заново после каждого пересчёта потенциала. Вместо этого можно оформить обход Куна в **итеративном** виде: после каждого пересчёта потенциала мы просматриваем добавившиеся жёсткие рёбра и, если их левые концы были

достижимыми, помечаем их правые концы также как достижимые и продолжаем обход из них.

- Развивая эту идею дальше, можно прийти к такому представлению алгоритма: это цикл, на каждом шаге которого сначала пересчитывается потенциал, затем находится столбец, ставший достижимым (а таковой всегда найдётся, поскольку после пересчёта потенциала всегда появляются новые достижимые вершины), и если этот столбец был ненасыщен, то найдена увеличивающая цепь, а если столбец был насыщен — то соответствующая ему в паросочетании строка также становится достижимой.

  Теперь алгоритм принимает вид: цикл добавления столбцов, на каждом из которых сначала пересчитывается потенциал, а затем какой-то новый столбец помечается как достижимый.

- Чтобы быстро пересчитывать потенциал (быстрее, чем наивный вариант за $O(n^2)$), надо поддерживать вспомогательные минимумы по каждому из столбцов j:

  $$minv[j] = \min_{i \in Z_1} \{a[i][j] - u[i] - v[j]\}.$$

  Как легко видеть, искомая величина Δ выражается через них следующим образом:

  $$\Delta = \min_{j \notin Z_2} \{minv[j]\}.$$

  Таким образом, нахождение Δ теперь можно произвести за $O(n)$.

  Поддерживать этот массив $minv[]$ необходимо при появлении новых посещённых строк. Это, очевидно, можно сделать за $O(n)$ на одну добавляемую строку (что в сумме даст $O(n^2)$). Также обновлять массив $minv[]$ надо при пересчёте потенциала, что также делается за время $O(n)$ на один пересчёт потенциала (поскольку $minv[]$ меняется только для недостигнутых пока столбцов: а именно, уменьшается на Δ).

Таким образом, алгоритм принимает такой вид: во внешнем цикле мы добавляем в рассмотрение строки матрицы одну за другой. Каждая строка обрабатывается за время $O(n^2)$, поскольку при этом могло происходить лишь $O(n)$ пересчётов потенциала (каждый — за время $O(n)$), для чего за время $O(n^2)$ поддерживается массив $minv[]$; алгоритм Куна суммарно отработает за время $O(n^2)$ (поскольку он представлен в форме $O(n)$ итераций, на каждой из которых посещается новый столбец).

Итоговая асимптотика составляет $O(n^3)$ — или, если задача прямоугольна, $O(n^2m)$.

Реализация венгерского алгоритма за $O(n^3)$ ($O(n^2m)$)

Приведённая реализация фактически была разработана **Андреем Лопатиным** несколько лет назад. Её отличает удивительная лаконичность: весь алгоритм помещается в 30 **строк кода**.

Данная реализация ищет решение для прямоугольной входной матрицы $a[1 \ldots n][1 \ldots m]$, где $n \le m$. Матрица хранится в 1-индексации в целях удобства и краткости кода. Дело в том, что в данной реализации вводятся фиктивные нулевая строка и нулевой столбец, что позволяет написать многие циклы в общем виде, без дополнительных проверок.

Массивы $u[0 \ldots n]$ и $v[0 \ldots m]$ хранят потенциал. Изначально он нулевой, что верно для матрицы, состоящей из нуля строк. (Отметим, что для данной реализации не важно, имеются или нет в матрице $a[][]$ отрицательные числа.)

Массив $p[0 \ldots m]$ содержит паросочетание: для каждого столбца $i = 1 \ldots m$ он хранит номер соответствующей выбранной строки $p[i]$ (или 0, если пока ничего не выбрано). При этом $p[0]$ для удобства реализации полагается равным номеру текущей рассматриваемой строки.

Массив $minv[1 \ldots m]$ содержит для каждого столбца j вспомогательные минимумы, необходимые для быстрого пересчёта потенциала:

$$minv[j] = \min_{i \in Z_1} \{a[i][j] - u[i] - v[j]\}.$$

Массив $way[1 \ldots m]$ содержит информацию о том, где эти минимумы достигаются, чтобы мы впоследствии смогли восстановить увеличивающую цепочку. На первый взгляд кажется, что в массиве $way[]$ для каждого столбца надо хранить номер строки, а также завести ещё один массив: для каждой строки запомнить номер столбца, из которого мы в неё пришли. Однако вместо этого можно заметить, что алгоритм Куна всегда попадает в строки, проходя по ребру паросочетания из столбцов, поэтому номера строк для восстановления цепочки всегда можно взять из паросочетания (т.е. из массива $p[]$). Таким образом, $way[j]$ для каждого столбца j содержит номер предшествующего столбца (или 0, если такого нет).

Сам алгоритм представляет из себя внешний **цикл по строкам матрицы**, внутри которого происходит добавление в рассмотрение i-ой строки матрицы. Внутренняя часть представляет собой цикл "do-while (p[j0] != 0)", который работает, пока не будет найден свободный столбец $j0$. Каждая итерация цикла помечает посещённым новый столбец с номером $j0$ (посчитанным на прошлой итерации; а изначально равным нулю — т. е. стартуем мы с фиктивного столбца), а также новую строку $i0$ — смежную ему в паросочетании (т.е. $p[j0]$; а изначально при $j0 = 0$ берётся i-ая строка). Из-за появления новой посещённой строки $i0$ нужно

соответствующим образом пересчитать массив $minv[]$, заодно мы находим минимум в нём — величину $delta$, и в каком столбце $j1$ этот минимум был достигнут (заметим, что при такой реализации $delta$ могло оказаться равной нулю, что означает, что на текущем шаге потенциал можно не менять: новый достижимый столбец есть и без того). После этого производится пересчёт потенциала $u[]$, $v[]$, соответствующее изменение массива $minv[]$. По окончании цикла "do-while" мы нашли увеличивающую цепочку, оканчивающуюся в столбце $j0$, "раскрутить" которую можно, пользуясь массивом предков $way[]$.

Константа INF — это "бесконечность", т.е. некоторое число, заведомо большее всех возможных чисел во входной матрице $a[][]$.

```
vector<int> u (n+1), v (m+1), p (m+1), way (m+1);
for (int i=1; i<=n; ++i) {
        p[0] = i;
        int j0 = 0;
        vector<int> minv (m+1, INF);
        vector<char> used (m+1, false);
        do {
                used[j0] = true;
                int i0 = p[j0],  delta = INF,  j1;
                for (int j=1; j<=m; ++j)
                        if (!used[j]) {
                                int cur = a[i0][j]-u[i0]-v[j];
                                if (cur < minv[j])
                                        minv[j] = cur,  way[j] = j0;
                                if (minv[j] < delta)
                                        delta = minv[j],  j1 = j;
                        }
                for (int j=0; j<=m; ++j)
                        if (used[j])
                                u[p[j]] += delta,  v[j] -= delta;
                        else
                                minv[j] -= delta;
                j0 = j1;
        } while (p[j0] != 0);
        do {
                int j1 = way[j0];
                p[j0] = p[j1];
                j0 = j1;
        } while (j0);
}
```

Восстановление ответа в более привычной форме, т.е. нахождение для каждой строки $i = 1 \dots n$ номера выбранного в ней столбца $ans[i]$, делается следующим образом:

```
vector<int> ans (n+1);
for (int j=1; j<=m; ++j)
        ans[p[j]] = j;
```

Стоимость найденного паросочетания можно просто взять как потенциал нулевого столбца (взятый с противоположным знаком). В самом деле, как легко проследить по коду, $-v[0]$ содержит в себе сумму всех величин $delta$, т.е. суммарное изменение потенциала. Хотя при каждом изменении потенциала изменяться могли сразу несколько величин $u[i]$ и $v[j]$, суммарное изменение величины потенциала в точности равно $delta$, поскольку пока нет увеличивающей цепи, число достижимых строк ровно на единицу больше числа достижимых столбцов (только текущая строка i не имеет себе "пары" в виде посещённого столбца):

```
int cost = -v[0];
```

Примеры задач

Приведём здесь несколько примеров на решение задачи о назначениях: начиная от совсем тривиальных, и заканчивая менее очевидными задачами:

- Дан двудольный граф, требуется найти в нём паросочетание **максимальное паросочетание минимального веса** (т.е. в первую очередь максимизируется размер паросочетания, во вторую — минимизируется его стоимость).

Для решения просто строим задачу о назначениях, ставя на месте отсутствующих рёбер число "бесконечность". После этого решаем задачу венгерским алгоритмом, и удаляем из ответа рёбра бесконечного веса (они могли войти в ответ, если у задачи нет решения в виде совершенного паросочетания).

- Дан двудольный граф, требуется найти в нём паросочетание **максимальное паросочетание максимального веса**.

  Решение опять же очевидно, только все веса надо умножить на минус единицу (либо в венгерском алгоритме заменить все минимумы на максимумы, а бесконечности — на минус бесконечности).

- Задача **детектирования движущихся объектов по снимкам**: было произведено два снимка, по итогам которых было получено два набор координат. Требуется соотнести объекты на первом и втором снимке, т. е. определить для каждой точки второго снимка, какой точке первого снимка она соответствовала. При этом требуется минимизировать сумму расстояний между сопоставленными точками (т.е. мы ищем решение, в котором объекты суммарно прошли наименьший путь).

  Для решения мы просто строим и решаем задачу о назначениях, где в качестве весов рёбер выступают евклидовы расстояния между точками.

- Задача **детектирования движущихся объектов по локаторам**: есть два локатора, которые умеют определять не положение объекта в пространстве, а лишь направление на него. С обоих локаторов (расположенных в различных точках) поступила информация в виде n таких направлений. Требуется определить положение объектов, т.е. определить предполагаемые положения объектов и соответствующие им пары направлений так, чтобы минимизировать сумму расстояний от объектов до лучей-направлений.

  Решение — опять же, просто строим и решаем задачу о назначениях, где вершинами первой доли являются n направлений с первого локатора, вершинами второй доли — n направлений со второго локатора, а весами рёбер — расстояния между соответствующими лучами.

- Покрытие **ориентированного ациклического графа путями**: дан ориентированный ациклический граф, требуется найти наименьшее число путей (при равенстве — с наименьшим суммарным весом), чтобы каждая вершина графа лежала бы ровно в одном пути.

  Решение — построить по данному графу соответствующий двудольный граф, и найти в нём максимальное паросочетание минимального веса. Более подробно см. отдельную статью.

- **Раскраска дерева**. Дано дерево, в котором каждая вершина, кроме листьев, имеет ровно $k-1$ сыновей. Требуется выбрать для каждой вершины некоторый цвет из k цветов так, чтобы никакие две смежные вершины не имели одинакового цвета. Кроме того, для каждой вершины и каждого цвета известна стоимость покраски этой вершины в этот цвет, и требуется минимизировать суммарную стоимость.

  Для решения воспользуемся методом динамического программирования. А именно, научимся считать величину $d[v][c]$, где v — номер вершины, c — номер цвета, а само значение $d[v][c]$ — это минимальная стоимость раскраски вершины v вместе с её потомками, причём сама вершина v имеет цвет c. Чтобы посчитать такую величину $d[v][c]$, надо распределить остальные $k-1$ цветов по сыновьям вершины v, а для этого надо построить и решить задачу о назначениях (в которой вершины одной доли — цвета, вершины другой доли — вершины-сыновья, а веса рёбер — это значения соответствующих динамик $d[][]$).

  Таким образом, каждая величина $d[v][c]$ считается с помощью решения задачи о назначениях, что в итоге даёт асимптотику $O(nk^4)$.

- Если в задаче о назначениях веса заданы не у рёбер, а у вершин, причём только **у вершин одной доли**, то можно обойтись без венгерского алгоритма, а достаточно лишь отсортировать вершины по весу и запустить обычный алгоритм Куна (более подробно см. отдельную статью).

- Рассмотрим следующий **частный случай**. Пусть каждой вершине первой доли приписано некоторое число $\alpha[i]$, а каждой вершине второй доли — $\beta[j]$. Пусть вес любого ребра (i,j) равен $\alpha[i] \cdot \beta[j]$ (числа $alpha[i]$ и $beta[j]$ нам известны). Решить задачу о назначениях.

  Для решения без венгерского алгоритма рассмотрим сначала случай, когда в обеих долях по две вершины. В этом случае, как нетрудно убедиться, выгодно соединять вершины в обратном порядке: вершину с меньшей $\alpha[i]$ соединить с вершиной с большей $\beta[j]$. Это правило легко обобщить на произвольное количество вершин: надо отсортировать вершины первой доли в порядке увеличения $\alpha[i]$, второй доли — в порядке уменьшения $\beta[j]$, и соединять вершины попарно в таком порядке. Таким образом, мы получаем решение с асимптотикой $O(n \log n)$.

- **Задача о потенциалах**. Дана матрица $a[1 \dots n][1 \dots m]$. Требуется найти два массива $u[1 \dots n]$ и $v[1 \dots m]$ такие, что для любых i и j выполняется $u[i] + v[j] \le a[i][j]$, но при этом сумма элементов массивов $u[]$ и $v[]$ максимальна.

  Зная венгерский алгоритм, решение этой задачи не составит никакого труда: венгерский алгоритм как раз находит именно такой потенциал $u[], v[]$, который удовлетворяет условию задачи. С другой стороны, без знания венгерского алгоритма решить такую задачу представляется почти невозможным.

Литература

- Ravindra Ahuja, Thomas Magnanti, James Orlin. Network Flows [1993]
- Harold Kuhn. The Hungarian Method for the Assignment Problem [1955]
- James Munkres. Algorithms for Assignment and Transportation Problems [1957]

Задачи в online judges

Список задач на решение задачи о назначениях:

- UVA #10746 "Crime Wave – The Sequel" [сложность: низкая]
- UVA #10888 "Warehouse" [сложность: средняя]
- SGU #210 "Beloved Sons" [сложность: средняя]
- UVA #3276 "The Great Wall Game" [сложность: высокая]
- UVA #10296 "Jogging Trails" [сложность: высокая]

Нахождение минимального разреза. Алгоритм Штор-Вагнера

Постановка задачи

Дан неориентированный взвешенный граф G с n вершинами и m рёбрами. Разрезом C называется некоторое подмножество вершин (фактически, разрез — разбиение вершин на два множества: принадлежащие C и все остальные). Весом разреза называется сумма весов рёбер, проходящих через разрез, т.е. таких рёбер, ровно один конец которых принадлежит C:

$$w(C) = \sum_{\substack{(v,u) \in E, \\ u \in C, v \notin C}} c(v, u),$$

где через E обозначено множество всех рёбер графа G, а через $c(v, u)$ — вес ребра (v, u).

Требуется найти **разрез минимального веса**.

Иногда эту задачу называют "глобальным минимальным разрезом" — по контрасту с задачей, когда заданы вершины-сток и исток, и требуется найти минимальный разрез C, содержащий сток и не содержащий исток.
Глобальный минимальный разрез равен минимуму среди разрезов минимальной стоимости по всевозможным парам исток-сток.

Хотя эту задачу можно решить с помощью алгоритма нахождения максимального потока (запуская его $O(n^2)$ раз для всевозможных пар истока и стока), однако ниже описан гораздо более простой и быстрый алгоритм, предложенный Матильдой Штор (Mechthild Stoer) и Франком Вагнером (Frank Wagner) в 1994 г.

В общем случае допускаются петли и кратные рёбра, хотя, понятно, петли абсолютно никак не влияют на результат, а все кратные рёбра можно заменить одним ребром с их суммарным весом. Поэтому мы для простоты будем считать, что во входном графе петли и кратные рёбра отсутствуют.

Описание алгоритма

Базовая идея алгоритма очень проста. Будем итеративно повторять следующий процесс: находить минимальный разрез между какой-нибудь парой вершин s и t, а затем объединять эти две вершины в одну (соединяя списки смежности). В конце концов, после $n - 1$ итерации, граф сожмётся в единственную вершину и процесс остановится. После этого ответом будет являться минимальный среди всех $n - 1$ найденных разрезов. Действительно, на каждой i-ой стадии найденный минимальный разрез C_i между вершинами s_i и t_i либо окажется искомым глобальным минимальным разрезом, либо же, напротив, вершины s_i и t_i невыгодно относить к разным множествам, поэтому мы ничего не ухудшаем, объединяя эти две вершины в одну.

Таким образом, мы свели задачу к следующей: для данного графа найти **минимальный разрез между какой-нибудь**, **произвольной**, **парой вершин** s и t. Для решения этой задачи был предложен следующий, тоже итеративный процесс. Вводим некоторое множество вершин A, которое изначально содержит единственную произвольную вершину. На каждом шаге находится вершина, **наиболее сильно связанная** с множеством A, т.е. вершина $v \notin A$, для которой следующая величина максимальна:

$$w(v, A) = \sum_{\substack{(v,u) \in E, \\ u \in A}} c(v, u)$$

(т.е. максимальна сумма весов рёбер, один конец которых v, а другой принадлежит A).

Опять же, этот процесс завершится через $n - 1$ итерацию, когда все вершины перейдут в множество A (кстати говоря, этот процесс очень напоминает алгоритм Прима). Тогда, как утверждает **теорема Штор-Вагнера**, если мы обозначим через s и t последние две добавленные в A вершины, то минимальный разрез между вершинами s и t будет состоять из единственной вершины — t. Доказательство этой теоремы будет приведено в следующем разделе (как это часто бывает, само по себе оно никак не способствует пониманию алгоритма).

Таким образом, общая **схема алгоритма** Штор-Вагнера такова. Алгоритм состоит из $n - 1$ фазы. На каждой фазе множество A сначала полагается состоящим из какой-либо вершины; подсчитываются стартовые веса вершин $w(v, A)$. Затем происходит $n - 1$ итерация, на каждой из которых выбирается вершина u с наибольшим значением $w(v, A)$ и добавляется в множество A, после чего пересчитываются значения w

для оставшихся вершин (для чего, очевидно, надо пройтись по всем рёбрам списка смежности выбранной вершины u). После выполнения всех итераций мы запоминаем в s и t номера последних двух добавленных вершин, а в качестве стоимости найденного минимального разреза между s и t можно взять значение $w(t, A \setminus t)$. Затем надо сравнить найденный минимальный разрез с текущим ответом, если меньше, то обновить ответ. Перейти к следующей фазе.

Если не использовать никаких сложных структур данных, то самой критичной частью будет нахождение вершины с наибольшей величиной w. Если производить это за $O(n)$, то, учитывая, что всего фаз $n-1$, и по $n-1$ итерации в каждой, итоговая **асимптотика алгоритма** получается $O(n^3)$.

Если для нахождения вершины с наибольшей величиной w использовать **Фибоначчиевы кучи** (которые позволяют увеличивать значение ключа за $O(1)$ в среднем и извлекать максимум за $O(\log n)$ в среднем), то все связанные с множеством A операции на одной фазе выполнятся за $O(m + n \log n)$. Итоговая асимптотика алгоритма в таком случае составит $O(nm + n^2 \log n)$.

Доказательство теоремы Штор-Вагнера

Напомним условие этой теоремы. Если добавить в множество A по очереди все вершины, каждый раз добавляя вершину, наиболее сильно связанную с этим множеством, то обозначим предпоследнюю добавленную вершину через s, а последнюю — через t. Тогда минимальный s-t разрез состоит из единственной вершины — t.

Для доказательства рассмотрим произвольный s-t разрез C и покажем, что его вес не может быть меньше веса разреза, состоящего из единственной вершины t:

$$w(\{t\}) \le w(C).$$

Для этого докажем следующий факт. Пусть A_v — состояние множества A непосредственно перед добавлением вершины v. Пусть C_v — разрез множества $A_v \cup v$, индуцированный разрезом C (проще говоря, C_v равно пересечению этих двух множеств вершин). Далее, вершина v называется активной (по отношению к разрезу C), если вершина v и предыдущая добавленная в A вершина принадлежат разным частям разреза C. Тогда, утверждается, для любой активной вершины v выполняется неравенство:

$$w(v, A_v) \le w(C_v).$$

В частности, t является активной вершиной (т.к. перед ним добавлялась вершина s), и при $v = t$ это неравенство превращается в утверждение теоремы:

$$w(t, A_t) = w(\{t\}) \le w(C_t) = w(C).$$

Итак, будем доказывать неравенство, для чего воспользуемся методом математической индукции.

Для первой активной вершины v это неравенство верно (более того, оно обращается в равенство) — поскольку все вершины A_v принадлежат одной части разреза, а v — другой.

Пусть теперь это неравенство выполнено для всех активных вершин вплоть до некоторой вершины v, докажем его для следующей активной вершины u. Для этого преобразуем левую часть:

$$w(u, A_u) \equiv w(u, A_v) + w(u, A_u \setminus A_v).$$

Во-первых, заметим, что:

$$w(u, A_v) \le w(v, A_v),$$

— это следует из того, что когда множество A было равно A_v, в него была добавлена именно вершина v, а не u, значит, она имела наибольшее значение w.

Далее, поскольку $w(v, A_v) \le w(C_v)$ по предположению индукции, то получаем:

$$w(u, A_v) \le w(C_v),$$

откуда имеем:

$$w(u, A_u) \le w(C_v) + w(u, A_u \setminus A_v).$$

Теперь заметим, что вершина u и все вершины $A_u \setminus A_v$ находятся в разных частях разреза C, поэтому эта величина $w(u, A_u \setminus A_v)$ обозначает сумму весов рёбер, которые учтены в $w(C_u)$, но ещё не были учтены в $w(C_v)$, откуда получаем:

$$w(u, A_u) \le w(C_v) + w(u, A_u \setminus A_v) \le w(C_u),$$

что и требовалось доказать.

Мы доказали соотношение $w(v, A_v) \le w(C_v)$, а из него, как уже говорилось выше, следует и вся теорема.

Реализация

Для наиболее простой и ясной реализации (с асимптотикой $O(n^3)$) было выбрано представление графа в виде матрицы смежности. Ответ хранится в переменных best_cost и best_cut (искомые стоимость минимального разреза и сами вершины, содержащиеся в нём).

Для каждой вершины в массиве exist хранится, существует ли она, или она была объединена с какой-то другой вершиной. В списке $\text{v}[i]$ для каждой сжатой вершины i хранятся номера исходных вершин, которые были сжаты в эту вершину i.

Алгоритм состоит из $n-1$ фазы (цикл по переменной ph). На каждой фазе сначала все вершины находятся вне множества A, для чего массив in_a заполняется нулями, и связности w всех вершин нулевые. На каждой из $n - \text{ph}$ итерации находится вершина sel с наибольшей величиной w. Если это итерация последняя, то ответ, если надо, обновляется, а предпоследняя prev и последняя sel выбранные вершины объединяются в одну. Если итерация не последняя, то sel добавляется в множество A, после чего пересчитываются веса всех остальных вершин.

Следует заметить, что алгоритм в ходе своей работы "портит" граф g, поэтому, если он ещё понадобится позже, надо сохранять его копию перед вызовом функции.

```
const int MAXN = 500;
int n, g[MAXN][MAXN];
int best_cost = 1000000000;
vector<int> best_cut;

void mincut() {
        vector<int> v[MAXN];
        for (int i=0; i<n; ++i)
                v[i].assign (1, i);
        int w[MAXN];
        bool exist[MAXN], in_a[MAXN];
        memset (exist, true, sizeof exist);
        for (int ph=0; ph<n-1; ++ph) {
                memset (in_a, false, sizeof in_a);
                memset (w, 0, sizeof w);
                for (int it=0, prev; it<n-ph; ++it) {
                        int sel = -1;
                        for (int i=0; i<n; ++i)
                                if (exist[i] && !in_a[i] && (sel == -1 || w
[i] > w[sel]))
                                        sel = i;
                        if (it == n-ph-1) {
                                if (w[sel] < best_cost)
                                        best_cost = w[sel],  best_cut = v[sel];
                                v[prev].insert (v[prev].end(), v[sel].begin(),
v[sel].end());
                                for (int i=0; i<n; ++i)
                                        g[prev][i] = g[i][prev] += g[sel][i];
                                exist[sel] = false;
                        }
                        else {
                                in_a[sel] = true;
                                for (int i=0; i<n; ++i)
                                        w[i] += g[sel][i];
                                prev = sel;
                        }
                }
        }
}
```

Литература

- Mechthild Stoer, Frank Wagner. A Simple Min-Cut Algorithm [1997]
- Kurt Mehlhorn, Christian Uhrig. The minimum cut algorithm of Stoer and Wagner [1995]

Поток минимальной стоимости, циркуляция минимальной стоимости. Алгоритм удаления циклов отрицательного веса

Постановка задач

Пусть G — сеть (network), то есть ориентированный граф, в котором выбраны вершины-исток s и сток t. Множество вершин обозначим через V, множество рёбер — через E. Каждому ребру $(i, j) \in E$ сопоставлены его пропускная способность $u_{ij} \geq 0$ и стоимость единицы потока c_{ij}. Если какого-то ребра (i, j) в графе нет, то предполагается, что $u_{ij} = c_{ij} = 0$.

Потоком (flow) в сети G называется такая действительнозначная функция f, сопоставляющая каждой паре вершин (i, j) поток f_{ij} между ними, и удовлетворяющая трём условиям:

- Ограничение пропускной способности (выполняется для любых $i, j \in V$):

$$f_{ij} \leq u_{ij}$$

- Антисимметричность (выполняется для любых $i, j \in V$):

$$f_{ij} = -f_{ji}$$

- Сохранение потока (выполняется для любых $i \in V$, кроме $i = s, i = t$):

$$\sum_{j \in V} f_{ij} = 0$$

Величиной потока называется величина

$$|f| = \sum_{i \in V} f_{si}$$

Стоимостью потока называется величина

$$z(f) = \sum_{i,j \in V} c_{ij} f_{ij}$$

Задача нахождения **потока минимальной стоимости** заключается в том, что по заданной величине потока $|f|$ требуется найти поток, обладающий минимальной стоимостью $z(f)$. Стоит обратить внимание на то, что стоимости c_{ij}, приписанные рёбрам, отвечают за стоимость единицы потока вдоль этого ребра; иногда встречается задача, когда рёбрам сопоставляются стоимости протекания потока вдоль этого ребра (т.е. если протекает поток любой величины, то взимается эта стоимость, независимо от величины потока) — эта задача не имеет ничего общего с рассматриваемой здесь и, более того, является NP-полной.

Задача нахождения **максимального потока минимальной стоимости** заключается в том, чтобы найти поток наибольшей величины, а среди всех таких — с минимальной стоимостью. В частном случае, когда веса всех рёбер одинаковы, эта задача становится эквивалентной обычной задаче о максимальном потоке.

Задача нахождения **циркуляции минимальной стоимости** заключается в том, чтобы найти поток нулевой величины с минимальной стоимостью. Если все стоимости неотрицательные, то, понятно, ответом будет нулевой поток $f_{ij} = 0$; если же есть рёбра отрицательного веса (а, точнее, циклы отрицательного веса), то даже при нулевом потоке возможно найти поток отрицательной стоимости. Задачу нахождения циркуляции минимальной стоимости можно, разумеется, поставить и на сети без истока и стока, поскольку никакой смысловой нагрузки они не несут (впрочем, в такой граф можно добавить исток и сток в виде изолированных вершин и получить обычную по формулировке задачу). Иногда ставится задача нахождения циркуляции максимальной стоимости — понятно, достаточно изменить стоимости рёбер на противоположные и получим задачу нахождения циркуляции уже минимальной стоимости.

Все эти задачи, разумеется, можно перенести и на неориентированные графы. Впрочем, перейти от неориентированного графа к ориентированному легко: каждое неориентированное ребро (i, j) с пропускной способностью u_{ij} и стоимостью c_{ij} следует заменить двумя ориентированными рёбрами (i, j) и (j, i) с одинаковыми пропускными способностями и стоимостями.

Остаточная сеть

Концепция **остаточной сети** G^f основана на следующей простой идее. Пусть есть некоторый поток f; вдоль каждого ребра $(i, j) \in E$ протекает некоторый поток $f_{ij} \le u_{ij}$. Тогда вдоль этого ребра можно (теоретически) пустить ещё $u_{ij} - f_{ij}$ единиц потока; эту величину и назовём **остаточной пропускной способностью**:

$$r_{ij}^f = u_{ij} - f_{ij}$$

Стоимость этих дополнительных единиц потока будет такой же:

$$c_{ij}^f = c_{ij}$$

Однако помимо этого, **прямого** ребра (i, j), в остаточной сети G^f появляется и **обратное ребро** (j, i). Интуитивный смысл этого ребра в том, что мы можем в будущем отменить часть потока, протекавшего по ребру (i, j). Соответственно, пропускание потока вдоль этого обратного ребра (j, i) фактически, и формально, означает уменьшение потока вдоль ребра (i, j). Обратное ребро имеет пропускную способность, равную нулю (чтобы, например, при $f_{ij} = 0$ и по обратному ребру невозможно было бы пропустить поток; при положительной величине $f_{ij} > 0$ для обратного ребра по свойству антисимметричности станет $f_{ji} < 0$, что меньше $c_{ji}^f = 0$, т.е. можно будет пропускать какой-то поток вдоль обратного ребра), остаточную пропускную способность — равную потоку вдоль прямого ребра, а стоимость — противоположную (ведь после отмены части потока мы должны соответственно уменьшить и стоимость):

$$\begin{aligned} u_{ji}^f &= 0 \\ r_{ji}^f &= f_{ij} \\ c_{ji}^f &= -c_{ij} \end{aligned}$$

Таким образом, каждому ориентированному ребру в G соответствует два ориентированных ребра в остаточной сети G^f, и у каждого ребра остаточной сети появляется дополнительная характеристика — остаточная пропускная способность. Впрочем, нетрудно заметить, что выражения для остаточной пропускной способности r_{ij}^f по сути одинаковы как для прямого, так и для обратного ребра, т.е. мы можем записать для любого ребра (i, j) остаточной сети:

$$r_{ij}^f = u_{ij}^f - f_{ij}^f$$

Кстати, при реализации это свойство позволяет не хранить остаточные пропускные способности, а просто вычислять их при необходимости для ребра.

Следует отметить, что из остаточной сети удаляются все рёбра, имеющие нулевую остаточную пропускную способность. Остаточная сеть G^f должна содержать **только рёбра с положительной остаточной пропускной способностью** r_{ij}^f.

Здесь стоит обратить внимание на такой важный момент: если в сети G были одновременно оба ребра (i, j) и (j, i), то в остаточной сети у каждого из них появится по обратному ребру, и в итоге появятся **кратные рёбра**. Например, такая ситуация часто возникает, когда сеть строится по неориентированному графу (и, получается, каждое неориентированное ребро в итоге приведёт к появлению четырёх рёбер в остаточной сети). Эту особенность нужно всегда помнить, она приводит к небольшому усложнению программирования, хотя в общем ничего не меняет. Кроме того, обозначение ребра (i, j) в таком случае становится неоднозначным, поэтому ниже мы везде будем считать, что такой ситуации в сети нет (исключительно в целях простоты и корректности описаний; на правильность идей это никак не влияет).

Критерий оптимальности по наличию циклов отрицательного веса

Теорема. Некоторый поток f является оптимальным (т.е. имеет наименьшую стоимость среди всех потоков такой же величины) тогда и только тогда, когда остаточная сеть G^f не содержит циклов отрицательного веса.

Доказательство: **необходимость**. Пусть поток f является оптимальным. Предположим, что остаточная сеть G^f содержит цикл отрицательного веса. Возьмём этот цикл отрицательного веса и выберем минимум k среди остаточных пропускных способностей рёбер этого цикла (k будет больше нуля). Но тогда можно увеличить поток вдоль каждого ребра цикла на величину k, при этом никакие свойства потока не нарушатся, величина потока не изменится, однако стоимость потока уменьшится (уменьшится на стоимость цикла, умноженную на k). Таким образом, если есть цикл отрицательного веса, то f не может быть оптимальным, ч.т.д.

Доказательство: **достаточность**. Для этого сначала докажем вспомогательные факты.

Лемма 1 (о декомпозиции потока): любой поток f можно представить в виде совокупности путей из истока в сток и циклов, все — имеющие положительный поток. Докажем эту лемму конструктивно: покажем, как разбить поток на совокупность путей и циклов. Если поток имеет ненулевую величину, то, очевидно, из истока s выходит хотя бы одно ребро с положительным потоком; пройдём по этому ребру, окажемся в какой-то вершине v_1. Если эта вершина $v_1 = t$, то останавливаемся — нашли путь из s в t. Иначе, по свойству сохранения потока, из v_1 должно выходить хотя бы одно ребро с положительным потоком; пройдём по нему в какую-то вершину v_2. Повторяя этот процесс, мы либо придём в сток t, либо же придём в какую-то вершину во второй раз. В первом случае мы обнаружим путь из s в t, во втором — цикл. Найденный путь/цикл будет иметь положительный поток k (минимум из потоков рёбер этого пути/цикла). Тогда уменьшим поток вдоль каждого ребра этого пути/цикла на величину k, в результате получим снова поток, к которому снова применим этот процесс. Рано или поздно поток вдоль всех рёбер станет нулевым, и мы найдём его декомпозицию на пути и циклы.

Лемма 2 (о разности потоков): для любых двух потоков f и g одной величины ($|f| = |g|$) поток g можно представить как поток f плюс несколько циклов в остаточной сети G^f. Действительно, рассмотрим разность этих потоков $g - f$ (вычитание потоков — это почленное вычитание, т.е. вычитание потоков вдоль каждого ребра). Нетрудно убедиться, что в результате получится некоторый поток нулевой величины, т.е. циркуляция. Произведём декомпозицию этой циркуляции согласно предыдущей лемме. Очевидно, это декомпозиция не может содержать путей (т.к. наличие s-t-пути с положительным потоком означает, что и величина потока в сети положительна). Таким образом, разность потоков g и f можно представить в виде суммы циклов в сети G. Более того, это будут и циклы в остаточной сети G^f, т.к. $g_{ij} - f_{ij} \le u_{ij} - f_{ij} = r_{ij}^f$, ч.т.д.

Теперь, вооружившись этими леммами, мы легко можем **доказать достаточность**. Итак, рассмотрим произвольный поток f, в остаточной сети которого нет циклов отрицательной стоимости. Рассмотрим также поток той же величины, но минимальной стоимости f^*; докажем, что f и f^* имеют одинаковую стоимость. Согласно лемме 2, поток f^* можно представить в виде суммы потока f и нескольких циклов. Но раз стоимости всех циклов неотрицательны, то и стоимость потока f^* не может быть меньше стоимости потока f: $z(f^*) \ge z(f)$. С другой стороны, т.к. поток f^* является оптимальным, то его стоимость не может быть выше стоимости потока f. Таким образом, $z(f) = z(f^*)$, ч.т.д.

Алгоритм удаления циклов отрицательного веса

Только что доказанная теорема даёт нам простой **алгоритм**, позволяющий найти поток минимальной стоимости: если у нас есть какой-то поток f, то построить для него остаточную сеть, проверить, есть ли в ней цикл отрицательного веса. Если такого цикла нет, то поток f является оптимальным (имеет наименьшую стоимость среди всех потоков такой же величины). Если же был найден цикл отрицательной стоимости, то посчитать поток k, который можно пропустить дополнительно через этот цикл (это k будет равно минимуму из остаточных пропускных способностей рёбер цикла). Увеличив поток на k вдоль каждого ребра цикла, мы, очевидно, не нарушим свойства потока, не изменим величину потока, но уменьшим стоимость этого потока, получив новый поток f', для которого надо повторить весь процесс.

Таким образом, чтобы запустить процесс улучшения стоимости потока, нам предварительно нужно найти **произвольный поток нужной величины** (каким-нибудь стандартным алгоритмом нахождения максимального потока, см., например, алгоритм Эдмондса-Карпа). В частности, если требуется найти циркуляцию наименьшей стоимости, то начать можно просто с нулевого потока.

Оценим **асимптотику** алгоритма. Поиск цикла отрицательной стоимости в графе с n вершинами и m рёбрами производится за $O(nm)$ (см. соответствующую статью). Если мы обозначим через C наибольшее из стоимостей рёбер, через U — наибольшую из пропускных способностей, то максимальное значение стоимости потока не превосходит mCU. Если все стоимости и пропускные способности — целые числа, то каждая итерация алгоритма уменьшает стоимость потока как минимум на единицу; следовательно, всего алгоритм совершит $O(mCU)$ итераций, а итоговая асимптотика составит:

$$O(nm^2CU)$$

Эта асимптотика — не строго полиномиальна (strong polynomial), поскольку зависит от величин пропускных способностей и стоимостей.

Впрочем, если искать не произвольный отрицательный цикл, а использовать какой-то более чёткий подход, то асимптотика может значительно уменьшиться. Например, если каждый раз искать цикл с минимальной средней стоимостью (что также можно производить за $O(nm)$), то время работы всего алгоритма уменьшится до $O(nm^2 \log n)$, и эта асимптотика уже является строго полиномиальной.

Реализация

Сначала введём структуры данных и функции для хранения графа. Каждое ребро хранится в отдельной структуре

$edge$, все рёбра лежат в общем списке $edges$, а для каждой вершины i в векторе $g[i]$ хранятся номера рёбер, выходящих из неё. Такая организация позволяет легко находить номер обратного ребра по номеру прямого ребра — они оказываются в списке $edges$ соседними, и номер одного можно получить по номеру другого операцией "^1" (она инвертирует младший бит). Добавление ориентированного ребра в граф осуществляет функция add_edge, которая добавляет сразу прямое и обратное рёбра.

```
const int MAXN = 100*2;
int n;
struct edge {
        int v, to, u, f, c;
};
vector<edge> edges;
vector<int> g[MAXN];

void add_edge (int v, int to, int cap, int cost) {
        edge e1 = { v, to, cap, 0, cost };
        edge e2 = { to, v, 0, 0, -cost };
        g[v].push_back ((int) edges.size());
        edges.push_back (e1);
        g[to].push_back ((int) edges.size());
        edges.push_back (e2);
}
```

В основной программе после чтения графа идёт бесконечный цикл, внутри которого выполняется алгоритм Форда-Беллмана, и если он обнаруживает цикл отрицательной стоимости, то вдоль этого цикла увеличивается поток. Поскольку остаточная сеть может представлять собой несвязный граф, то алгоритм Форда-Беллмана запускается из каждой не достигнутой ещё вершины. В целях оптимизации алгоритм использует очередь (текущая очередь q и новая очередь nq), чтобы не перебирать на каждой стадии все рёбра. Вдоль обнаруженного цикла каждый раз проталкивается ровно единица потока, хотя, понятно, в целях оптимизации величину потока можно определять как минимум остаточных пропускных способностей.

```
const int INF = 1000000000;
for (;;) {
        bool found = false;

        vector<int> d (n, INF);
        vector<int> par (n, -1);
        for (int i=0; i<n; ++i)
                if (d[i] == INF) {
                        d[i] = 0;
                        vector<int> q, nq;
                        q.push_back (i);
                        for (int it=0; it<n && q.size(); ++it) {
                                nq.clear();
                                sort (q.begin(), q.end());
                                q.erase (unique (q.begin(), q.end()), q.end());
                                for (size_t j=0; j<q.size(); ++j) {
                                        int v = q[j];
                                        for (size_t k=0; k<g[v].size(); ++k) {
                                                int id = g[v][k];
                                                if (edges[id].f < edges[id].u)
                                                        if (d[v] + edges[id].
c < d[edges[id].to]) {
                                                                d[edges[id].
to] = d[v] + edges[id].c;
                                                                par[edges
[id].to] = v;
                                                                nq.
push_back (edges[id].to);
                                                        }
                                        }
                                }
                                swap (q, nq);
                        }
                        if (q.size()) {
                                int leaf = q[0];
                                vector<int> path;
                                for (int v=leaf; v!=-1; v=par[v])
                                        if (find (path.begin(), path.end(),
```

```
v) == path.end())
                                                        path.push_back (v);
                                                else {
                                                        path.erase (path.begin(),
find (path.begin(), path.end(), v));
                                                        break;
                                                }
                                        for (size_t j=0; j<path.size(); ++j) {
                                                int to = path[j],  v = path[(j+1)%
path.size()];
                                                for (size_t k=0; k<g[v].size(); ++k)
                                                        if (edges[ g[v][k] ].to == to) {
                                                                int id = g[v][k];
                                                                edges[id].f += 1;
                                                                edges[id^1].f -= 1;
                                                        }
                                        }
                                        found = true;
                                }
                        }

                if (!found)  break;
}
```

Литература

- Томас Кормен, Чарльз Лейзерсон, Рональд Ривест, Клиффорд Штайн. **Алгоритмы: Построение и анализ** [2005]
- Ravindra Ahuja, Thomas Magnanti, James Orlin. Network flows [1993]
- Andrew Goldberg, Robert Tarjan. Finding Minimum-Cost Circulations by Cancelling Negative Cycles [1989]

Алгоритм Диница

Постановка задачи

Пусть дана сеть, т.е. ориентированный граф G, в котором каждому ребру (u, v) приписана пропускная способность c_{uv}, а также выделены две вершины — исток s и сток t.

Требуется найти в этой сети поток f_{uv} из истока s в сток t максимальной величины.

Немного истории

Этот алгоритм был опубликован советским (израильским) учёным Ефимом **Диницем** (Yefim Dinic, иногда пишется как "Dinitz") в 1970 г., т.е. даже на два года раньше опубликования алгоритма Эдмондса-Карпа (впрочем, оба алгоритма были независимо открыты в 1968 г.).

Кроме того, следует отметить, что некоторые упрощения алгоритма были произведены Шимоном Ивеном (Shimon Even) и его учеником Алоном Итаи (Alon Itai) в 1979 г. Именно благодаря им алгоритм получил свой современный облик: они применили к идее Диница концепцию блокирующих потоков Александра Карзанова (Alexander Karzanov, 1974 г.), а также переформулировали алгоритм к той комбинации обхода в ширину и в глубину, в которой сейчас этот алгоритм и излагается везде.

Развитие идей по отношению к потоковым алгоритмам крайне интересно рассматривать, учитывая "**железный занавес**" тех лет, разделявший СССР и Запад. Видно, как иногда похожие идеи появлялись почти одновременно (как в случае алгоритма Диница и алгоритма Эдмондса-Карпа), правда, имея при этом разную эффективность (алгоритм Диница на один порядок быстрее); иногда же, наоборот, появление идеи по одну сторону "занавеса" опережало аналогичный ход по другую сторону более чем на десятилетие (как алгоритм Карзанова проталкивания в 1974 г. и алгоритм Гольдберга (Goldberg) проталкивания в 1985 г.).

Необходимые определения

Введём три необходимых определения (каждое из них является независимым от остальных), которые затем будут использоваться в алгоритме Диница.

Остаточной сетью G^R по отношению к сети G и некоторому потоку f в ней называется сеть, в которой каждому ребру $(u, v) \in G$ с пропускной способностью c_{uv} и потоком f_{uv} соответствуют два ребра:

- (u, v) с пропускной способностью $c^R_{uv} = c_{uv} - f_{uv}$
- (v, u) с пропускной способностью $c^R_{vu} = f_{uv}$

Стоит отметить, что при таком определении в остаточной сети могут появляться кратные рёбра: если в исходной сети было как ребро (u, v), так и (v, u).

Остаточное ребро можно интуитивно понимать как меру того, насколько ещё можно увеличить поток вдоль какого-то ребра. В самом деле, если по ребру (u, v) с пропускной способностью c_{uv} протекает поток f_{uv}, то потенциально по нему можно пропустить ещё $c_{uv} - f_{uv}$ единиц потока, а в обратную сторону можно пропустить до f_{uv} единиц потока, что будет означать отмену потока в первоначальном направлении.

Блокирующим потоком в данной сети называется такой поток, что любой путь из истока s в сток t содержит насыщенное этим потоком ребро. Иными словами, в данной сети не найдётся такого пути из истока в сток, вдоль которого можно беспрепятственно увеличить поток.

Блокирующий поток не обязательно максимален. Теорема Форда-Фалкерсона говорит о том, что поток будет максимальным тогда и только тогда, когда в остаточной сети не найдётся $s - t$ пути; в блокирующем же потоке ничего не утверждается о существовании пути по рёбрам, появляющимся в остаточной сети.

Слоистая сеть для данной сети строится следующим образом. Сначала определяются длины кратчайших путей из истока s до всех остальных вершин; назовём уровнем $\text{level}[v]$ вершины её расстояние от истока. Тогда в слоистую сеть включают все те рёбра (u, v) исходной сети, которые ведут с одного уровня на какой-либо другой, более поздний, уровень, т.е. $\text{level}[u] + 1 = \text{level}[v]$ (почему в этом случае разница расстояний не может превосходить единицы, следует из свойства кратчайших расстояний). Таким образом, удаляются все рёбра, расположенные целиком внутри уровней, а также рёбра, ведущие назад, к предыдущим уровням.

Очевидно, слоистая сеть ациклична. Кроме того, любой $s - t$ путь в слоистой сети является кратчайшим путём

в исходной сети.

Построить слоистую сеть по данной сети очень легко: для этого надо запустить обход в ширину по рёбрам этой сети, посчитав тем самым для каждой вершины величину $\text{level}[]$, и затем внести в слоистую сеть все подходящие рёбра.

Примечание. Термин "слоистая сеть" в русскоязычной литературе не употребляется; обычно эта конструкция называется просто "вспомогательным графом". Впрочем, на английском языке обычно используется термин "layered network".

Алгоритм

Схема алгоритма

Алгоритм представляет собой несколько **фаз**. На каждой фазе сначала строится остаточная сеть, затем по отношению к ней строится слоистая сеть (обходом в ширину), а в ней ищется произвольный блокирующий поток. Найденный блокирующий поток прибавляется к текущему потоку, и на этом очередная итерация заканчивается.

Этот алгоритм схож с алгоритмом Эдмондса-Карпа, но основное отличие можно понимать так: на каждой итерации поток увеличивается не вдоль одного кратчайшего $s - t$ пути, а вдоль целого набора таких путей (ведь именно такими путями и являются пути в блокирующем потоке слоистой сети).

Корректность алгоритма

Покажем, что если алгоритм завершается, то на выходе у него получается поток именно максимальной величины.

В самом деле, предположим, что в какой-то момент в слоистой сети, построенной для остаточной сети, не удалось найти блокирующий поток. Это означает, что сток t вообще не достижим в слоистой сети из истока s. Но поскольку слоистая сеть содержит в себе все кратчайшие пути из истока в остаточной сети, это в свою очередь означает, что в остаточной сети нет пути из истока в сток. Следовательно, применяя теорему Форда-Фалкерсона, получаем, что текущий поток в самом деле максимален.

Оценка числа фаз

Покажем, что алгоритм Диница всегда выполняет **менее n фаз**. Для этого докажем две леммы:

Лемма 1. Кратчайшее расстояние от истока до каждой вершины не уменьшается с выполнением каждой итерации, т.е.

$$\text{level}_{i+1}[v] \geq \text{level}_i[v]$$

где нижний индекс обозначает номер фазы, перед которой взяты значения этих переменных.

Доказательство. Зафиксируем произвольную фазу i и произвольную вершину v и рассмотрим любой кратчайший $s - v$ путь P в сети G^R_{i+1} (напомним, так мы обозначаем остаточную сеть, взятую перед выполнением $i + 1$-ой фазы). Очевидно, длина пути P равна $\text{level}_{i+1}[v]$.

Заметим, что в остаточную сеть G^R_{i+1} могут входить только рёбра из G^R, а также рёбра, обратные рёбрам из G^R (это следует из определения остаточной сети). Рассмотрим два случая:

- Путь P содержит только рёбра из G^R. Тогда, понятно, длина пути P больше либо равна $\text{level}_i[v]$ (потому что $\text{level}_i[v]$ по определению — длина кратчайшего пути), что и означает выполнение неравенства.
- Путь P содержит как минимум одно ребро, не содержащееся в G^R (но обратное какому-то ребру из G^R). Рассмотрим первое такое ребро; пусть это будет ребро (u, w).

  $$s \Longrightarrow u \to w \Longrightarrow v$$

  Мы можем применить нашу лемму к вершине u, потому что она подпадает под первый случай; итак, мы получаем неравенство $\text{level}_{i+1}[u] \geq \text{level}_i[u]$.

  Теперь заметим, что поскольку ребро (u, w) появилось в остаточной сети только после выполнения i-ой фазы, то отсюда следует, что вдоль ребра (w, u) был дополнительно пропущен какой-то поток; следовательно, ребро (w, u) принадлежало слоистой сети перед i-ой фазой, а потому $\text{level}_i[u] = \text{level}_i[w] + 1$. Учтём, что по свойству кратчайших путей $\text{level}_{i+1}[w] = \text{level}_{i+1}[u] + 1$, и объединяя это равенство с двумя предыдущими неравенствами, получаем:

  $$\text{level}_{i+1}[w] \geq \text{level}_i[w] + 2.$$

  Теперь мы можем применять те же самые рассуждения ко всему оставшемуся пути до v (т.е. что каждое инвертированное ребро добавляет к level как минимум два), и в итоге получим требуемое неравенство.

Лемма 2. Расстояние между истоком и стоком строго увеличивается после каждой фазы алгоритма, т.е.:

$$\text{level}'[t] > \text{level}[t],$$

где штрихом помечено значение, полученное на следующей фазе алгоритма.

Доказательство: от противного. Предположим, что после выполнения текущей фазы оказалось, что $\text{level}'[t] = \text{level}[t]$. Рассмотрим кратчайший путь из истока в сток; по предположению, его длина должна сохраниться неизменной. Однако остаточная сеть на следующей фазе содержит только рёбра остаточной сети перед выполнением текущей фазы, либо обратные к ним. Таким образом, пришли к противоречию: нашёлся $s - t$ путь, который не содержит насыщенных рёбер, и имеет ту же длину, что и кратчайший путь. Этот путь должен был быть "заблокирован" блокирующим потоком, чего не произошло, в чём и заключается противоречие, что и требовалось доказать.

Эту лемму интуитивно можно понимать следующим образом: на i-ой фазе алгоритм Диница выявляет и насыщает все $s - t$ пути длины i.

Поскольку длина кратчайшего пути из s в t не может превосходить $n - 1$, то, следовательно, алгоритм Диница совершает **не более $n - 1$ фазы**.

Поиск блокирующего потока

Чтобы завершить построение алгоритма Диница, надо описать алгоритм нахождения блокирующего потока в слоистой сети — ключевое место алгоритма.

Мы рассмотрим три возможных варианта реализации поиска блокирующего потока:

- Искать $s - t$ пути по одному, пока такие пути находятся. Путь можно найти за $O(m)$ обходом в глубину, а всего таких путей будет $O(m)$ (поскольку каждый путь насыщает как минимум одно ребро). Итоговая асимптотика поиска одного блокирующего потока составит $O(m^2)$.
- Аналогично предыдущей идее, однако удалять в процессе обхода в глубину из графа все "лишние" рёбра, т.е. рёбра, вдоль которых не получится дойти до стока.

  Это очень легко реализовать: достаточно удалять ребро после того, как мы просмотрели его в обходе в глубину (кроме того случая, когда мы прошли вдоль ребра и нашли путь до стока). С точки зрения реализации, надо просто поддерживать в списке смежности каждой вершины указатель на первое неудалённое ребро, и увеличивать этот указать в цикле внутри обхода в глубину.

  Оценим асимптотику этого решения. Каждый обход в глубину завершается либо насыщением как минимум одного ребра (если этот обход достиг стока), либо продвижением вперёд как минимум одного указателя (в противном случае). Можно понять, что один запуск обхода в глубину из основной программы работает за $O(k + n)$, где k — число продвижений указателей. Учитывая, что всего запусков обхода в глубину в рамках поиска одного блокирующего потока будет $O(p)$, где p — число рёбер, насыщенных этим блокирующим потоком, то весь алгоритм поиска блокирующего потока отработает за $O(pk + pn)$, что, учитывая, что все указатели в сумме прошли расстояние $O(m)$, даёт асимптотику $O(m + pn)$. В худшем случае, когда блокирующий поток насыщает все рёбра, асимптотика получается $O(nm)$; эта асимптотика и будет использоваться далее.

  Можно сказать, что этот способ нахождения блокирующего потока чрезвычайно эффективен в том смысле, что на поиск одного увеличивающего пути он тратит $O(n)$ операций в среднем. Именно в этом и кроется разность на целый порядок эффективностей алгоритма Диница и Эдмондса-Карпа (который ищет один увеличивающий путь за $O(m)$).

  Этот способ решения является по-прежнему простым для реализации, но достаточно эффективным, и потому наиболее часто применяется на практике.
- Можно применить специальные структуры данных — динамические деревья Слетора (Sleator) и Тарьяна (Tarjan)). Тогда каждый блокирующий поток можно найти за время $O(m \log n)$.

Асимптотика

Таким образом, весь алгоритм Диница выполняется за $O(n^2 m)$, если блокирующий поток искать описанным выше способом за $O(nm)$. Реализация с использованием динамических деревьев Слетора и Тарьяна будет работать за время $O(nm \log n)$.

Единичные сети

Единичной называется такая сеть, в которой все пропускные способности равны 0 либо 1, и у любой вершины либо входящее, либо исходящее ребро единственно.

Этот случай является достаточно важным, поскольку в задаче поиска **максимального паросочетания** построенная сеть является именно единичной.

Докажем, что на единичных сетях алгоритм Диница даже в простой реализации (которая на произвольных графах отрабатывает за $O(n^2 m)$) работает за время $O(m\sqrt{n})$, достигая на задаче поиска наибольшего паросочетания наилучший из известных алгоритмов — алгоритм Хопкрофта-Карпа. Чтобы доказать

эту оценку, надо рассмотреть два случая:

- Если величина искомого потока не превосходит $\sqrt{n}$.

  В этом случае мы получаем, что число фаз и запусков обхода в глубину есть величина $O(\sqrt{n})$. Вспоминая, что одна фаза в этой реализации работает за $O(m + pn)$, получаем итоговую асимптотику $O(\sqrt{n}m + \sqrt{n}n) = O(\sqrt{n}m)$.

- Если величина $|f|$ искомого потока больше $\sqrt{n}$.

  Заметим, что поток в единичной сети можно представить в виде суммы $|f|$ вершинно-непересекающихся путей, а потому максимальная длина $s - t$ пути имеет величину $O(\sqrt{n})$. Учитывая, что одна фаза алгоритма Диница целиком обрабатывает все пути какой-либо длины, мы снова получаем, что число фаз есть величина $O(\sqrt{n})$. Суммируя асимптотику одной фазы $O(m + pn)$ по всем фазам, получаем $O(\sqrt{n}m + n^2) = O(\sqrt{n}m)$, что и требовалось доказать.

Реализация

Приведём две реализации алгоритма за $O(n^2m)$, работающие на сетях, заданных матрицами смежности и списками смежности соответственно.

Реализация над графами в виде матриц смежности

```
const int MAXN = ...; // число вершин
const int INF = 1000000000; // константа-бесконечность

int n, c[MAXN][MAXN], f[MAXN][MAXN], s, t, d[MAXN], ptr[MAXN], q[MAXN];

bool bfs() {
        int qh=0, qt=0;
        q[qt++] = s;
        memset (d, -1, n * sizeof d[0]);
        d[s] = 0;
        while (qh < qt) {
                int v = q[qh++];
                for (int to=0; to<n; ++to)
                        if (d[to] == -1 && f[v][to] < c[v][to]) {
                                q[qt++] = to;
                                d[to] = d[v] + 1;
                        }
        }
        return d[t] != -1;
}

int dfs (int v, int flow) {
        if (!flow)  return 0;
        if (v == t)  return flow;
        for (int & to=ptr[v]; to<n; ++to) {
                if (d[to] != d[v] + 1)  continue;
                int pushed = dfs (to, min (flow, c[v][to] - f[v][to]));
                if (pushed) {
                        f[v][to] += pushed;
                        f[to][v] -= pushed;
                        return pushed;
                }
        }
        return 0;
}

int dinic() {
        int flow = 0;
        for (;;) {
                if (!bfs())  break;
                memset (ptr, 0, n * sizeof ptr[0]);
                while (int pushed = dfs (s, INF))
                        flow += pushed;
        }
        return flow;
```

```
}
```

Сеть должна быть предварительно считана: должны быть заданы переменные n, s, t, а также считана матрица пропускных способностей $c[][]$. Основная функция решения — $\mathrm{dinic}()$, которая возвращает величину найденного максимального потока.

Реализация над графами в виде списков смежности

```
const int MAXN = ...; // число вершин
const int INF = 1000000000; // константа-бесконечность

struct edge {
        int a, b, cap, flow;
};

int n, s, t, d[MAXN], ptr[MAXN], q[MAXN];
vector<edge> e;
vector<int> g[MAXN];

void add_edge (int a, int b, int cap) {
        edge e1 = { a, b, cap, 0 };
        edge e2 = { b, a, 0, 0 };
        g[a].push_back ((int) e.size());
        e.push_back (e1);
        g[b].push_back ((int) e.size());
        e.push_back (e2);
}

bool bfs() {
        int qh=0, qt=0;
        q[qt++] = s;
        memset (d, -1, n * sizeof d[0]);
        d[s] = 0;
        while (qh < qt && d[t] == -1) {
                int v = q[qh++];
                for (size_t i=0; i<g[v].size(); ++i) {
                        int id = g[v][i],
                                to = e[id].b;
                        if (d[to] == -1 && e[id].flow < e[id].cap) {
                                q[qt++] = to;
                                d[to] = d[v] + 1;
                        }
                }
        }
        return d[t] != -1;
}

int dfs (int v, int flow) {
        if (!flow)  return 0;
        if (v == t)  return flow;
        for (; ptr[v]<(int)g[v].size(); ++ptr[v]) {
                int id = g[v][ptr[v]],
                        to = e[id].b;
                if (d[to] != d[v] + 1)  continue;
                int pushed = dfs (to, min (flow, e[id].cap - e[id].flow));
                if (pushed) {
                        e[id].flow += pushed;
                        e[id^1].flow -= pushed;
                        return pushed;
                }
        }
        return 0;
}

int dinic() {
        int flow = 0;
        for (;;) {
                if (!bfs())  break;
                memset (ptr, 0, n * sizeof ptr[0]);
```

```
                while (int pushed = dfs (s, INF))
                        flow += pushed;
        }
        return flow;
}
```

Сеть должна быть предварительно считана: должны быть заданы переменные n, s, t, а также добавлены все рёбра (ориентированные) с помощью вызовов функции $\mathrm{add_edge}$. Основная функция решения — $\mathrm{dinic}()$, которая возвращает величину найденного максимального потока.

Алгоритм Куна нахождения наибольшего паросочетания в двудольном графе

Дан двудольный граф G, содержащий n вершин и m рёбер. Требуется найти наибольшее паросочетание, т.е. выбрать как можно больше рёбер, чтобы ни одно выбранное ребро не имело общей вершины ни с каким другим выбранным ребром.

Описание алгоритма

Необходимые определения

Паросочетанием M называется набор попарно несмежных рёбер графа (иными словами, любой вершине графа должно быть инцидентно не более одного ребра из множества M). Мощностью паросочетания назовём количество рёбер в нём. Наибольшим (или максимальным) паросочетанием назовём паросочетание, мощность которого максимальна среди всех возможных паросочетаний в данном графе. Все те вершины, у которых есть смежное ребро из паросочетания (т.е. которые имеют степень ровно один в подграфе, образованном M), назовём насыщенными этим паросочетанием.

Цепью длины k назовём некоторый простой путь (т.е. не содержащий повторяющихся вершин или рёбер), содержащий k рёбер.

Чередующейся цепью (в двудольном графе, относительно некоторого паросочетания) назовём цепь, в которой рёбра поочередно принадлежат/не принадлежат паросочетанию.

Увеличивающей цепью (в двудольном графе, относительно некоторого паросочетания) назовём чередующуюся цепь, у которой начальная и конечная вершины не принадлежат паросочетанию.

Теорема Бержа

Формулировка. Паросочетание является максимальным тогда и только тогда, когда не существует увеличивающих относительно него цепей.

Доказательство необходимости. Покажем, что если паросочетание M максимально, то не существует увеличивающей относительно него цепи. Доказательство это будет конструктивным: мы покажем, как увеличить с помощью этой увеличивающей цепи P мощность паросочетания M на единицу.

Для этого выполним так называемое чередование паросочетания вдоль цепи P. Мы помним, что по определению первое ребро цепи P не принадлежит паросочетанию, второе — принадлежит, третье — снова не принадлежит, четвёртое — принадлежит, и т.д. Давайте поменяем состояние всех рёбер вдоль цепи P: те рёбра, которые не входили в паросочетание (первое, третье и т.д. до последнего) включим в паросочетание, а рёбра, которые раньше входили в паросочетание (второе, четвёртое и т.д. до предпоследнего) — удалим из него.

Понятно, что мощность паросочетания при этом увеличилась на единицу (потому что было добавлено на одно ребро больше, чем удалено). Осталось проверить, что мы построили корректное паросочетание, т.е. что никакая вершина графа не имеет сразу двух смежных рёбер из этого паросочетания. Для всех вершин чередующей цепи P, кроме первой и последней, это следует из самого алгоритма чередования: сначала мы у каждой такой вершины удалили смежное ребро, потом добавили. Для первой и последней вершины цепи P также ничего не могло нарушиться, поскольку до чередования они должны были быть ненасыщенными. Наконец, для всех остальных вершин, — не входящих в цепь P, — очевидно, ничего не поменялось. Таким образом, мы в самом деле построили паросочетание, и на единицу большей мощности, чем старое, что и завершает доказательство необходимости.

Доказательство достаточности. Докажем, что если относительно некоторого паросочетания M нет увеличивающих путей, то оно — максимально.

Доказательство проведём от противного. Пусть есть паросочетание M', имеющее бОльшую мощность, чем M. Рассмотрим симметрическую разность Q этих двух паросочетаний, т.е. оставим все рёбра, входящие в M или в M', но не в оба одновременно.

Понятно, что множество рёбер Q — уже наверняка не паросочетание. Рассмотрим, какой вид это множество рёбер имеет; для удобства будем рассматривать его как граф. В этом графе каждая вершина, очевидно, имеет степень не выше 2 (потому что каждая вершина может иметь максимум два смежных ребра — из одного паросочетания и из другого). Легко понять, что тогда этот граф состоит только из циклов или путей, причём ни те, ни другие не пересекаются друг с другом.

Теперь заметим, что и пути в этом графе Q могут быть не любыми, а только чётной длины. В самом деле, в любом пути в графе Q рёбра чередуются: после ребра из M идёт ребро из M', и наоборот. Теперь, если мы рассмотрим какой-то путь нечётной длины в графе Q, то получится, что в исходном графе G это будет увеличивающей цепью либо

для паросочетания M, либо для M'. Но этого быть не могло, потому что в случае паросочетания M это противоречит с условием, а в случае M' — с его максимальностью (ведь мы уже доказали необходимость теоремы, из которой следует, что при существовании увеличивающей цепи паросочетание не может быть максимальным).

Докажем теперь аналогичное утверждение и для циклов: все циклы в графе Q могут иметь только чётную длину. Это доказать совсем просто: понятно, что в цикле рёбра также должны чередоваться (принадлежать по очереди то M, то M'), но это условие не может выполниться в цикле нечётной длины — в нём обязательно найдутся два соседних ребра из одного паросочетания, что противоречит определению паросочетания.

Таким образом, все пути и циклы графа $Q = M \oplus M'$ имеют чётную длину. Следовательно, граф Q содержит равное количество рёбер из M и из M'. Но, учитывая, что в Q содержатся все рёбра M и M', за исключением их общих рёбер, то отсюда следует, что мощность M и M' совпадают. Мы пришли к противоречию: по предположению паросочетание M было не максимальным, значит, теорема доказана.

Алгоритм Куна

Алгоритм Куна — непосредственное применение теоремы Бержа. Его можно кратко описать так: сначала возьмём пустое паросочетание, а потом — пока в графе удаётся найти увеличивающую цепь, — будем выполнять чередование паросочетания вдоль этой цепи, и повторять процесс поиска увеличивающей цепи. Как только такую цепь найти не удалось — процесс останавливаем, — текущее паросочетание и есть максимальное.

Осталось детализировать способ нахождения увеличивающих цепей. **Алгоритм Куна** — просто ищет любую из таких цепей с помощью **обхода в глубину** или **в ширину**. Алгоритм Куна просматривает все вершины графа по очереди, запуская из каждой обход, пытающийся найти увеличивающую цепь, начинающуюся в этой вершине.

Удобнее описывать этот алгоритм, считая, что граф уже разбит на две доли (хотя на самом деле алгоритм можно реализовать и так, чтобы ему не давался на вход граф, явно разбитый на две доли).

Алгоритм просматривает все вершины v первой доли графа: $v = 1 \dots n_1$. Если текущая вершина v уже насыщена текущим паросочетанием (т.е. уже выбрано какое-то смежное ей ребро), то эту вершину пропускаем. Иначе — алгоритм пытается насытить эту вершину, для чего запускается поиск увеличивающей цепи, начинающейся с этой вершины.

Поиск увеличивающей цепи осуществляется с помощью специального обхода в глубину или ширину (обычно в целях простоты реализации используют именно обход в глубину). Изначально обход в глубину стоит в текущей ненасыщенной вершине v первой доли. Просматриваем все рёбра из этой вершины, пусть текущее ребро — это ребро (v, to). Если вершина to ещё не насыщена паросочетанием, то, значит, мы смогли найти увеличивающую цепь: она состоит из единственного ребра (v, to); в таком случае просто включаем это ребро в паросочетание и прекращаем поиск увеличивающей цепи из вершины v. Иначе, — если to уже насыщена каким-то ребром (p, to), то попытаемся пройти вдоль этого ребра: тем самым мы попробуем найти увеличивающую цепь, проходящую через рёбра (v, to), (to, p). Для этого просто перейдём в нашем обходе в вершину p — теперь мы уже пробуем найти увеличивающую цепь из этой вершины.

Можно понять, что в результате этот обход, запущенный из вершины v, либо найдёт увеличивающую цепь, и тем самым насытит вершину v, либо же такой увеличивающей цепи не найдёт (и, следовательно, эта вершина v уже не сможет стать насыщенной).

После того, как все вершины $v = 1 \dots n_1$ будут просмотрены, текущее паросочетание будет максимальным.

Время работы

Итак, алгоритм Куна можно представить как серию из n запусков обхода в глубину/ширину на всём графе. Следовательно, всего этот алгоритм исполняется за время $O(nm)$, что в худшем случае есть $O(n^3)$.

Однако эту оценку можно немного **улучшить**. Оказывается, для алгоритма Куна важно то, какая доля выбрана за первую, а какая — за вторую. В самом деле, в описанной выше реализации запуски обхода в глубину/ширину происходят только из вершин первой доли, поэтому весь алгоритм исполняется за время $O(n_1 m)$, где n_1 — число вершин первой доли. В худшем случае это составляет $O(n_1^2 n_2)$ (где n_2 — число вершин второй доли). Отсюда видно, что выгоднее, когда первая доля содержит меньшее число вершин, нежели вторая. На очень несбалансированных графах (когда n_1 и n_2 сильно отличаются) это выливается в значительную разницу времён работы.

Реализация

Приведём здесь реализацию вышеописанного алгоритма, основанную на обходе в глубину, и принимающей двудольный граф в виде явно разбитого на две доли графа. Эта реализация весьма лаконична, и, возможно, её стоит запомнить именно в таком виде.

Здесь n — число вершин в первой доле, k — во второй доле, $g[v]$ — список рёбер из вершины v первой доли (т.е. список номеров вершин, в которые ведут эти рёбра из v). Вершины в обеих долях занумерованы независимо, т.е. первая доля — с номерами $1 \dots n$, вторая — с номерами $1 \dots k$.

Дальше идут два вспомогательных массива: mt и used. Первый — mt — содержит в себе информацию о текущем паросочетании. Для удобства программирования, информация эта содержится только для вершин второй доли: $mt[i]$ — это номер вершины первой доли, связанной ребром с вершиной i второй доли (или -1, если никакого ребра паросочетания из i не выходит). Второй массив — used — обычный массив "посещённостей" вершин в обходе в глубину (он нужен, просто чтобы обход в глубину не заходил в одну вершину дважды).

Функция $\mathrm{try_kuhn}$ — и есть обход в глубину. Она возвращает true, если ей удалось найти увеличивающую цепь из вершины v, при этом считается, что эта функция уже произвела чередование паросочетания вдоль найденной цепи.

Внутри функции просматриваются все рёбра, исходящие из вершины v первой доли, и затем проверяется: если это ребро ведёт в ненасыщенную вершину to, либо если эта вершина to насыщена, но удаётся найти увеличивающую цепь рекурсивным запуском из $\mathrm{mt}[\mathrm{to}]$, то мы говорим, что мы нашли увеличивающую цепь, и перед возвратом из функции с результатом true производим чередование в текущем ребре: перенаправляем ребро, смежное с to, в вершину v.

В основной программе сначала указывается, что текущее паросочетание — пустое (список mt заполняется числами -1). Затем перебирается вершина v первой доли, и из неё запускается обход в глубину $\mathrm{try_kuhn}$, предварительно обнулив массив used.

Стоит заметить, что размер паросочетания легко получить как число вызовов $\mathrm{try_kuhn}$ в основной программе, вернувших результат true. Само искомое максимальное паросочетание содержится в массиве mt.

```
int n, k;
vector < vector<int> > g;
vector<int> mt;
vector<char> used;

bool try_kuhn (int v) {
        if (used[v])  return false;
        used[v] = true;
        for (size_t i=0; i<g[v].size(); ++i) {
                int to = g[v][i];
                if (mt[to] == -1 || try_kuhn (mt[to])) {
                        mt[to] = v;
                        return true;
                }
        }
        return false;
}

int main() {
        ... чтение графа ...

        mt.assign (k, -1);
        for (int v=0; v<n; ++v) {
                used.assign (n, false);
                try_kuhn (v);
        }

        for (int i=0; i<k; ++i)
                if (mt[i] != -1)
                        printf ("%d %d\n", mt[i]+1, i+1);
}
```

Ещё раз повторим, что алгоритм Куна легко реализовать и так, чтобы он работал на графах, про которые известно, что они двудольные, но явное их разбиение на две доли не найдено. В этом случае придётся отказаться от удобного разбиения на две доли, и всю информацию хранить для всех вершин графа. Для этого массив списков g теперь задаётся не только для вершин первой доли, а для всех вершин графа (понятно, теперь вершины обеих долей занумерованы в общей нумерации — от 1 до n). Массивы mt и used теперь также определены для вершин обеих долей, и, соответственно, их нужно поддерживать в этом состоянии.

Улучшенная реализация

Модифицируем алгоритм следующим образом. До основного цикла алгоритма найдём каким-нибудь простым алгоритмом **произвольное паросочетание** (простым **эвристическим алгоритмом**), и лишь затем будем выполнять цикл с вызовами функции kuhn(), который будет улучшать это паросочетание. В результате алгоритм будет работать заметно быстрее на случайных графах — потому что в большинстве графов можно легко набрать паросочетание достаточно большого веса с помощью эвристики, а потом улучшить найденное паросочетание до максимального уже обычным алгоритмом Куна. Тем самым мы сэкономим на

запусках обхода в глубину из тех вершин, которые мы уже включили с помощью эвристики в текущее паросочетание.

Например, можно просто перебрать все вершины первой доли, и для каждой из них найти произвольное ребро, которое можно добавить в паросочетание, и добавить его. Даже такая простая эвристика способна ускорить алгоритм Куна в несколько раз.

Следует обратить внимание на то, что основной цикл придётся немного модифицировать. Поскольку при вызове функции `try_kuhn` в основном цикле предполагается, что текущая вершина ещё не входит в паросочетание, то нужно добавить соответствующую проверку.

В реализации изменится только код в функции main():

```
int main() {
        ... чтение графа ...

        mt.assign (k, -1);
        vector<char> used1 (n);
        for (int i=0; i<n; ++i)
                for (size_t j=0; j<g[i].size(); ++j)
                        if (mt[g[i][j]] == -1) {
                                mt[g[i][j]] = i;
                                used1[i] = true;
                                break;
                        }
        for (int i=0; i<n; ++i) {
                if (used1[i])  continue;
                used.assign (n, false);
                try_kuhn (i);
        }

        for (int i=0; i<k; ++i)
                if (mt[i] != -1)
                        printf ("%d %d\n", mt[i]+1, i+1);
}
```

Другой хорошей эвристикой является следующая. На каждом шаге будет искать вершину наименьшей степени (но не изолированную), из неё выбирать любое ребро и добавлять его в паросочетание, затем удаляя обе эти вершины со всеми инцидентными им рёбрами из графа. Такая жадность работает очень хорошо на случайных графах, даже в большинстве случаев строит максимальное паросочетание (хотя и против неё есть тест, на котором она найдёт паросочетание значительно меньшей величины, чем максимальное).

Проверка графа на двудольность и разбиение на две доли

Пусть дан неориентированный граф. Требуется проверить, является ли он двудольным, т.е. можно ли разделить его вершины на две доли так, чтобы не было рёбер, соединяющих две вершины одной доли. Если граф является двудольным, то вывести сами доли.

Решим эту задачу с помощью поиска в ширину за O (M).

Признак двудольности

Теорема. Граф является двудольным тогда и только тогда, когда все его простые циклы имеют чётную длину.

Впрочем, с практической точки зрения искать все простые циклы неудобно. Намного проще проверять граф на двудольность следующим алгоритмом:

Алгоритм

Произведём серию поисков в ширину. Т.е. будем запускать поиск в ширину из каждой непосещённой вершины. Ту вершину, из которой мы начинаем идти, мы помещаем в первую долю. В процессе поиска в ширину, если мы идём в какую-то новую вершину, то мы помещаем её в долю, отличную от доли текущей вершину. Если же мы пытаемся пройти по ребру в вершину, которая уже посещена, то мы проверяем, чтобы эта вершина и текущая вершина находились в разных долях. В противном случае граф двудольным не является.

По окончании работы алгоритма мы либо обнаружим, что граф не двудолен, либо найдём разбиение вершин графа на две доли.

Реализация

```
int n;
vector < vector<int> > g;
... чтение графа ...

vector<char> part (n, -1);
bool ok = true;
vector<int> q (n);
for (int st=0; st<n; ++st)
        if (part[st] == -1) {
                int h=0, t=0;
                q[t++] = st;
                part[st] = 0;
                while (h<t) {
                        int v = q[h++];
                        for (size_t i=0; i<g[v].size(); ++i) {
                                int to = g[v][i];
                                if (part[to] == -1)
                                        part[to] = !part[v],  q[t++] = to;
                                else
                                        ok &= part[to] != part[v];
                        }
                }
        }

puts (ok ? "YES" : "NO");
```

Нахождение наибольшего по весу вершинно-взвешенного паросочетания

Дан двудольный граф G. Для каждой вершины первой доли указан её вес. Требуется найти паросочетание наибольшего веса, т.е. с наибольшей суммой весов насыщенных вершин.

Ниже мы опишем и докажем алгоритм, основанный на алгоритме Куна, который будет находить оптимальное решение.

Алгоритм

Сам алгоритм чрезвычайно прост. **Отсортируем** вершины первой доли в порядке убывания (точнее говоря, невозрастания) весов, и применим к полученному графу **алгоритм Куна**.

Утверждается, что полученное при этом максимальное (с точки зрения количества рёбер) паросочетание будет и оптимальным с точки зрения суммы весов насыщенных вершин (несмотря на то, что после сортировки мы фактически больше не используем эти веса).

Таким образом, реализация будет примерно такой:

```
int n;
vector < vector<int> > g (n);
vector used (n);
vector<int> order (n); // список вершин, отсортированный по весу
... чтение ...

for (int i=0; i<n; ++i) {
        int v = order[i];
        used.assign (n, false);
        try_kuhn (v);
}
```

Функция try_kuhn() берётся безо всяких изменений из алгоритма Куна.

Доказательство

Напомним основные положения **теории матроидов**.

Матроид M - это упорядоченная пара (S,I), где S - некоторое множество, I - непустое семейство подмножеств множества S, которые удовлетворяют следующим условиям:

1. Множество S конечное.
2. Семейство I является наследственным, т.е. если какое-то множество принадлежит I, то все его подмножества также принадлежат I.
3. Структура M обладает свойством замены, т.е. если $A \in I$, и $B \in I$, и $|A|<|B|$, то найдётся такой элемент $x \in A-B$, что $A \cup x \in I$.

Элементы семейства I называются независимыми подмножествами.

Матроид называется взвешенным, если для каждого элемента $x \in S$ определён некоторый вес. Весом подмножества называется сумма весов его элементов.

Наконец, важнейшая теорема в теории взвешенных матроидов: чтобы получить оптимальный ответ, т.е. независимое подмножество с наибольшим весом, нужно действовать жадно: начиная с пустого подмножества, будем добавлять (если, конечно, текущий элемент можно добавить без нарушения независимости) все элементы по одному в порядке уменьшения (точнее, невозрастания) их весов:

```
отсортировать множество S по невозрастанию веса;
ans = [];
foreach (x in S)
        if (ans ∪ x ∈ I)
                ans = ans ∪ x;
```

Утверждается, что по окончании этого процесса мы получим подмножество с наибольшим весом.

Теперь **докажем**, что **наша задача** - не что иное, как взвешенный **матроид**.

Пусть S - множество всех вершин первой доли. Чтобы свести нашу задачу в двудольном графе к матроиду относительно вершин первой доли, поставим в соответствие каждому паросочетанию такое подмножество S,

которое равно множеству насыщенных вершин первой доли. Можно также определить и обратное соответствие (из множества насыщенных вершин - в паросочетание), которое, хотя и не будет однозначным, однако вполне нас будет устраивать.

Тогда определим семейство I как семейство таких подмножеств множества S, для которых найдётся хотя бы одно соответствующее паросочетание.

Далее, для каждого элемента S, т.е. для каждой вершины первой доли, по условию определён некоторый вес. Причём вес подмножества, как нам и требуется в рамках теории матроидов, определяется как сумма весов элементов в нём.

Тогда задача о нахождении паросочетания наибольшего веса теперь переформулируется как задача нахождения независимого подмножества наибольшего веса.

Осталось проверить, что выполнены 3 вышеописанных условия, наложенных на матроид. Во-первых, очевидно, что S является конечным. Во-вторых, очевидно, что удаление ребра из паросочетания эквивалентно удалению вершины из множества насыщенных вершин, а потому свойство наследственности выполняется. В-третьих, как следует из корректности алгоритма Куна, если текущее паросочетание не максимально, то всегда найдётся такая вершина, которую можно будет насытить, не удаляя из множества насыщенных вершин другие вершины.

Итак, мы показали, что наша задача является взвешенным матроидом относительно множества насыщенных вершин первой доли, а потому к ней применим жадный алгоритм.

Осталось показать, что **алгоритм Куна является этим жадным алгоритмом**.

Однако это довольно очевидный факт. Алгоритм Куна на каждом шаге пытается насытить текущую вершину - либо просто проводя ребро в ненасыщенную вершину второй доли, либо находя удлиняющую цепь и чередуя паросочетание вдоль неё. И в том, и в другом случае никакие уже насыщенные вершины не перестают быть ненасыщенными, а ненасыщенные на предыдущих шагах вершины первой доли не насыщаются и на этом шаге. Таким образом, алгоритм Куна является жадным алгоритмом, строящим оптимальное независимое подмножества матроида, что и завершает наше доказательство.

Алгоритм Эдмондса нахождения наибольшего паросочетания в произвольных графах

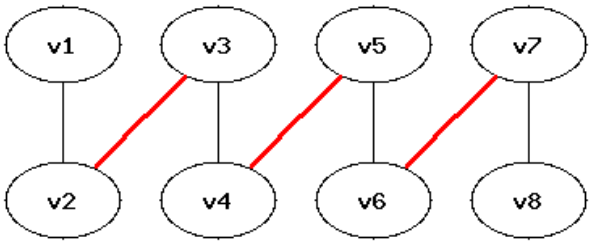
Дан неориентированный невзвешенный граф G с n вершинами. Требуется найти в нём наибольшее паросочетание, т. е. такое наибольшее (по мощности) множество m его рёбер, что никакие два ребра из выбранных не инцидентны друг другу (т.е. не имеют общих вершин).

В отличие от случая двудольного графа (см. Алгоритм Куна), в графе G могут присутствовать циклы нечётной длины, что значительно усложняет поиск увеличивающих путей.

Приведём сначала теорему Бержа, из которой следует, что, как и в случае двудольных графов, наибольшее паросочетание можно находить при помощи увеличивающих путей.

Увеличивающие пути. Теорема Бержа

Пусть зафиксировано некоторое паросочетание M. Тогда простая цепь $P = (v_1, v_2, \ldots, v_k)$ называется чередующейся цепью, если в ней рёбра по очереди принадлежат - не принадлежат паросочетанию M. Чередующаяся цепь называется увеличивающей, если её первая и последняя вершины не принадлежат паросочетанию. Иными словами, простая цепь P является увеличивающей тогда и только тогда, когда вершина $v_1 \notin M$, ребро $(v_2, v_3) \in M$, ребро $(v_4, v_5) \in M$, ..., ребро $(v_{k-2}, v_{k-1}) \in M$, и вершина $v_k \notin M$.



Теорема Бержа (Claude Berge, 1957 г.). Паросочетание M является наибольшим тогда и только тогда, когда для него не существует увеличивающей цепи.

Доказательство необходимости. Пусть для паросочетания M существует увеличивающая цепь P. Покажем, как перейти к паросочетанию большей мощности. Выполним чередование паросочетания M вдоль этой цепи P, т.е. включим в паросочетание рёбра (v_1, v_2), (v_3, v_4), ..., (v_{k-1}, v_k), и удалим из паросочетания рёбра (v_2, v_3), (v_4, v_5), ..., (v_{k-2}, v_{k-1}). В результате, очевидно, будет получено корректное паросочетание, мощность которого будет на единицу выше, чем у паросочетания M (т.к. мы добавили $k/2$ рёбер, а удалили $k/2 - 1$ ребро).

Доказательство достаточности. Пусть для паросочетания M не существует увеличивающей цепи, докажем, что оно является наибольшим. Пусть $\overline{M}$ — наибольшее паросочетание. Рассмотрим симметрическую разность $\overline{G} = M \oplus \overline{M}$ (т.е. множество рёбер, принадлежащих либо M, либо $\overline{M}$, но не обоим одновременно). Покажем, что $\overline{G}$ содержит одинаковое число рёбер из M и $\overline{M}$ (т.к. мы исключили из $\overline{G}$ только общие для них рёбра, то отсюда будет следовать и $|M| = |\overline{M}|$). Заметим, что $\overline{G}$ состоит только из простых цепей и циклов (т.к. иначе одной вершине были бы инцидентны сразу два ребра какого-либо паросочетания, что невозможно). Далее, циклы не могут иметь нечётную длину (по той же самой причине). Цепь в $\overline{G}$ также не может иметь нечётную длину (иначе бы она являлась увеличивающей цепью для M, что противоречит условию, или для $\overline{M}$, что противоречит его максимальности). Наконец, в чётных циклах и цепях чётной длины в $\overline{G}$ рёбра поочерёдно входят в M и $\overline{M}$, что и означает, что в $\overline{G}$ входит одинаковое количество рёбер от M и $\overline{M}$. Как уже упоминалось выше, отсюда следует, что $|M| = |\overline{M}|$, т.е. M является наибольшим паросочетанием.

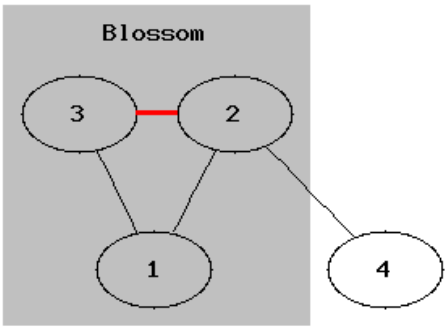
Теорема Бержа даёт основу для алгоритма Эдмондса — поиск увеличивающих цепей и чередование вдоль них, пока увеличивающие цепи находятся.

Алгоритм Эдмондса. Сжатие цветков

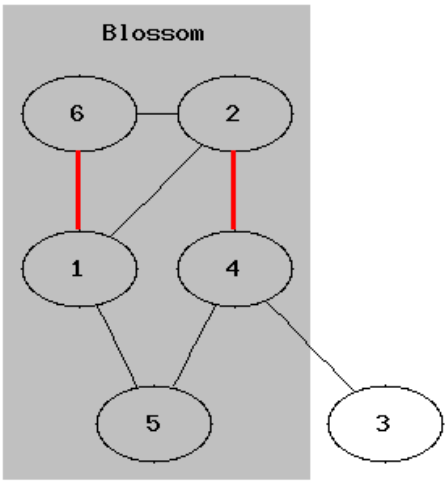
Основная проблема заключается в том, как находить увеличивающий путь. Если в графе имеются циклы нечётной длины, то просто запускать обход в глубину/ширину нельзя.

Можно привести простой контрпример, когда при запуске из одной из вершин алгоритм, не обрабатывающий особо

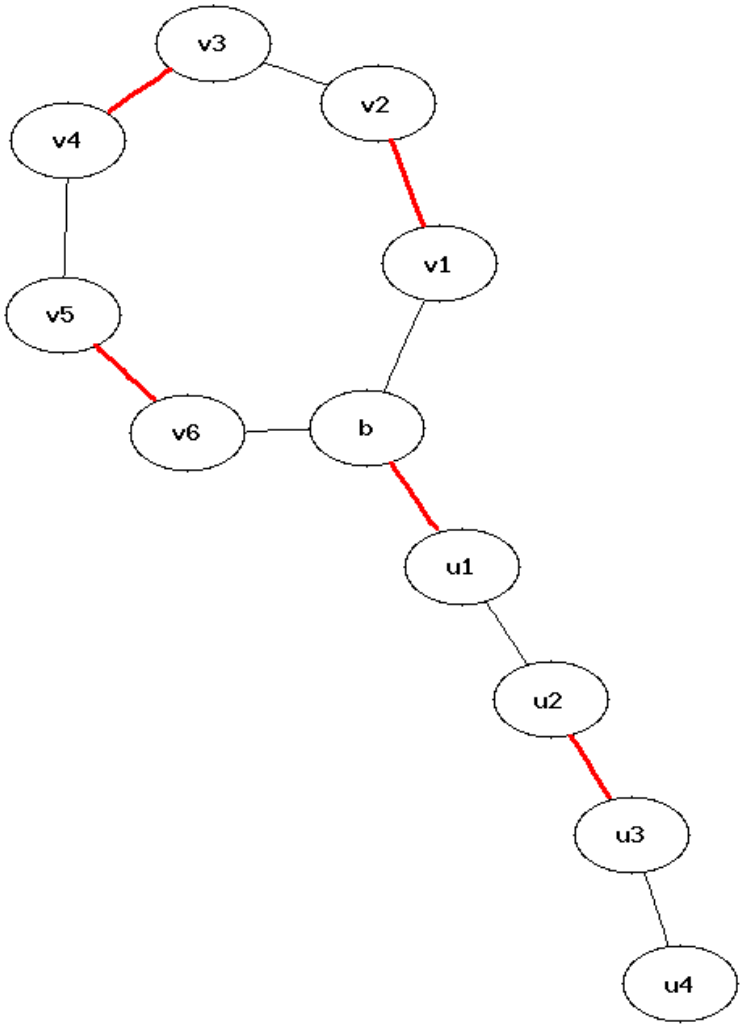
циклы нечётной длины (фактически, Алгоритм Куна) не найдёт увеличивающий путь, хотя должен. Это цикл длины 3 с висящим на нём ребром, т.е. граф 1-2, 2-3, 3-1, 2-4, и ребро 2-3 взято в паросочетание. Тогда при запуске из вершины 1, если обход пойдёт сначала в вершину 2, то он "упрётся" в вершину 3, вместо того чтобы найти увеличивающую цепь 1-3-2-4. Правда, на этом примере при запуске из вершины 4 алгоритм Куна всё же найдёт эту увеличивающую цепь.



Тем не менее, можно построить граф, на котором при определённом порядке в списках смежности алгоритм Куна зайдёт в тупик. В качестве примера можно привести такой граф с 6 вершинами и 7 рёбрами: 1-2, 1-6, 2-6, 2-4, 4-3, 1-5, 4-5. Если применить здесь алгоритм Куна, то он найдёт паросочетание 1-6, 2-4, после чего он должен будет обнаружить увеличивающую цепь 5-1-6-2-4-3, однако может так и не обнаружить её (если из вершины 5 он пойдёт сначала в 4, и только потом в 1, а при запуске из вершины 3 он из вершины 2 пойдёт сначала в 1, и только затем в 6).



Как мы увидели на этом примере, вся проблема в том, что при попадании в цикл нечётной длины обход может пойти по циклу в неправильном направлении. На самом деле, нас интересуют только "насыщенные" циклы, т.е. в которых имеется k насыщенных рёбер, где длина цикла равна $2k+1$. В таком цикле есть ровно одна вершина, не насыщенная рёбрами этого цикла, назовём её **базой** (base). К базовой вершине подходит чередующийся путь чётной (возможно, нулевой) длины, начинающийся в свободной (т.е. не принадлежащей паросочетанию) вершине, и этот путь называется **стеблем** (stem). Наконец, подграф, образованный "насыщенным" нечётным циклом, называется **цветком** (blossom).



Идея алгоритма Эдмондса (Jack Edmonds, 1965 г.) - в **сжатии цветков** (blossom shrinking). Сжатие цветка — это сжатие всего нечётного цикла в одну псевдо-вершину (соответственно, все рёбра, инцидентные вершинам этого цикла, становятся инцидентными псевдо-вершине). Алгоритм Эдмондса ищет в графе все цветки, сжимает их, после чего в графе не остаётся "плохих" циклов нечётной длины, и на таком графе (называемом "поверхностным" (surface) графом) уже можно искать увеличивающую цепь простым обходом в глубину/ширину. После нахождения увеличивающей цепи в поверхностном графе необходимо "развернуть" цветки, восстановив тем самым увеличивающую цепь в исходном графе.

Однако неочевидно, что после сжатия цветка не нарушится структура графа, а именно, что если в графе G существовала увеличивающая цепь, то она существует и в графе $\overline{G}$, полученном после сжатия цветка, и наоборот.

Теорема Эдмондса. В графе $\overline{G}$ существует увеличивающая цепь тогда и только тогда, когда существует увеличивающая цепь в G.

Доказательство. Итак, пусть граф $\overline{G}$ был получен из графа G сжатием одного цветка (обозначим через B цикл цветка, и через $\overline{B}$ соответствующую сжатую вершину), докажем утверждение теоремы. Вначале заметим, что достаточно рассматривать случай, когда база цветка является свободной вершиной (не принадлежащей паросочетанию). Действительно, в противном случае в базе цветка оканчивается чередующийся путь чётной длины, начинающийся в свободной вершине. Прочередовав паросочетание вдоль этого пути, мощность паросочетания не изменится, а база цветка станет свободной вершиной. Итак, при доказательстве можно считать, что база цветка является свободной вершиной.

Доказательство необходимости. Пусть путь P является увеличивающим в графе G. Если он не проходит через B, то тогда, очевидно, он будет увеличивающим и в графе $\overline{G}$. Пусть P проходит через B. Тогда можно не теряя общности считать, что путь P представляет собой некоторый путь P_1, не проходящий по вершинам B, плюс некоторый путь P_2, проходящий по вершинам B и, возможно, другим вершинам. Но тогда путь $P_1 + \overline{B}$ будет являться увеличивающим путём в графе $\overline{G}$, что и требовалось доказать.

Доказательство достаточности. Пусть путь $\overline{P}$ является увеличивающим путём в графе $\overline{G}$. Снова, если путь $\overline{P}$ не проходит через $\overline{B}$, то путь $\overline{P}$ без изменений является увеличивающим путём в G, поэтому этот случай мы рассматривать не будем.

Рассмотрим отдельно случай, когда $\overline{P}$ начинается со сжатого цветка $\overline{B}$, т.е. имеет вид $(\overline{B}, c, \ldots)$. Тогда в цветке B найдётся соответствующая вершина v, которая связана (ненасыщенным) ребром с c. Осталось только заметить, что из базы цветка всегда найдётся чередующийся путь чётной длины до вершины v. Учитывая всё вышесказанное, получаем, что путь $P = (b, \ldots, v, c, \ldots)$ является увеличивающим путём в графе G.

Пусть теперь путь $\overline{P}$ проходит через псевдо-вершину $\overline{B}$, но не начинается и не заканчивается в ней. Тогда в $\overline{P}$ есть два ребра, проходящих через $\overline{B}$, пусть это $(a, \overline{B})$ и $(\overline{B}, c)$. Одно из них обязательно должно принадлежать паросочетанию M, однако, т.к. база цветка не насыщена, а все остальные вершины цикла цветка B насыщены рёбрами цикла, то мы приходим к противоречию. Таким образом, этот случай просто невозможен.

Итак, мы рассмотрели все случаи и в каждом из них показали справедливость теоремы Эдмондса.

Общая схема алгоритма Эдмондса принимает следующий вид:

```
void edmonds() {
        for (int i=0; i<n; ++i)
                if (вершина i не в паросочетании) {
                        int last_v = find_augment_path (i);
                        if (last_v != -1)
                                выполнить чередование вдоль пути из i в last_v;
                }
}

int find_augment_path (int root) {
        обход в ширину:
                int v = текущая_вершина;
                перебрать все рёбра из v
                        если обнаружили цикл нечётной длины, сжать его
                        если пришли в свободную вершину, return
                        если пришли в несвободную вершину, то добавить
                                в очередь смежную ей в паросочетании
        return -1;
}
```

Эффективная реализация

Сразу оценим асимптотику. Всего имеется n итераций, на каждой из которых выполняется обход в ширину за $O(m)$, кроме того, могут происходить операции сжатия цветков — их может быть $O(n)$. Таким образом, если мы научимся сжимать цветок за $O(n)$, то общая асимптотика алгоритма составит $O(n \cdot (n + n^2)) = O(n^3)$.

Основную сложность представляют операции сжатия цветков. Если выполнять их, непосредственно объединяя списки смежности в один и удаляя из графа лишние вершины, то асимптотика сжатия одного цветка будет $O(m)$, кроме того, возникнут сложности при "разворачивании" цветков.

Вместо этого будем для каждой вершины графа G поддерживать указатель на базу цветка, которому она принадлежит (или на себя, если вершина не принадлежит никакому цветку). Нам надо решить две задачи: сжатие цветка за $O(n)$ при его обнаружении, а также удобное сохранение всей информации для последующего чередования вдоль увеличивающего пути.

Итак, одна итерация алгоритма Эдмондса представляет собой обход в ширину, выполняемый из заданной свободной вершины root. Постепенно будет строиться дерево обхода в ширину, причём путь в нём до любой вершины будет являться чередующимся путём, начинающимся со свободной вершины root. Для удобства программирования будем класть в очередь только те вершины, расстояние до которых в дереве путей чётно (будем называть такие вершины чётными — т.е. это корень дерева, и вторые концы рёбер в паросочетании). Само дерево будем хранить в виде массива предков $\mathrm{p}[]$, в котором для каждой нечётной вершины (т.е. до которой расстояние в дереве путей нечётно, т.е. это первые концы рёбер в паросочетании) будем хранить предка - чётную вершину. Таким образом, для восстановления пути из дерева нам надо поочерёдно пользоваться массивами $\mathrm{p}[]$ и $\mathrm{match}[]$, где $\mathrm{match}[]$ — для каждой вершины содержит смежную ей в паросочетании, или -1, если таковой нет.

Теперь становится понятно, как обнаруживать циклы нечётной длины. Если мы из текущей вершины v в процессе обхода в ширину приходим в такую вершину u, являющуюся корнем root или принадлежащую паросочетанию и дереву путей (т.е. $\mathrm{p}[\mathrm{match}[]]$ от которой не равно -1), то мы обнаружили цветок. Действительно, при выполнении этих условий и вершина v, и вершина u являются чётными вершинами. Расстояние от них до их наименьшего общего предка имеет одну чётность, поэтому найденный нами цикл имеет нечётную длину.

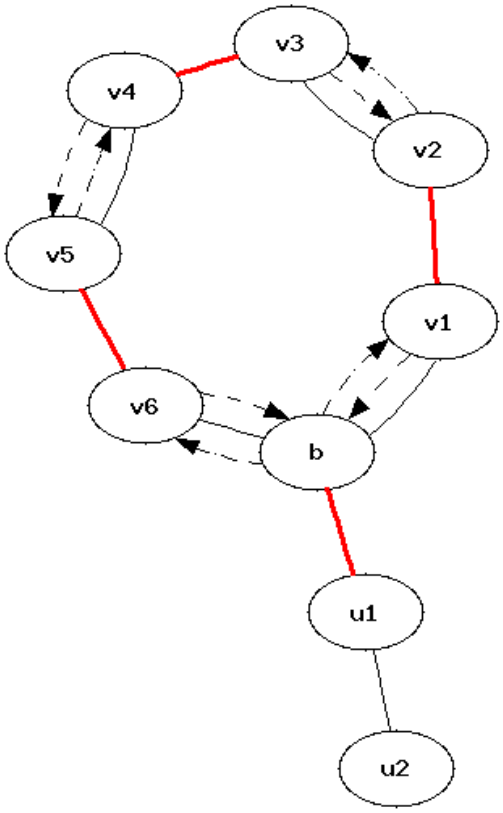
Научимся **сжимать цикл**. Итак, мы обнаружили нечётный цикл при рассмотрении ребра (v, u), где u и v — чётные вершины. Найдём их наименьшего общего предка b, он и будет базой цветка. Нетрудно заметить, что база тоже является чётной вершиной (поскольку у нечётных вершин в дереве путей есть только один сын). Однако надо заметить, что b — это, возможно, псевдовершина, поэтому мы фактически найдём базу цветка, являющегося наименьшим общим предком вершин v и u. Реализуем сразу нахождение наименьшего общего предка (нас вполне устраивает асимптотика $O(n)$):

```
int lca (int a, int b) {
        bool used[MAXN] = { 0 };
```

```
        // поднимаемся от вершины a до корня, помечая все чётные вершины
        for (;;) {
                a = base[a];
                used[a] = true;
                if (match[a] == -1)  break; // дошли до корня
                a = p[match[a]];
        }
        // поднимаемся от вершины b, пока не найдём помеченную вершину
        for (;;) {
                b = base[b];
                if (used[b])  return b;
                b = p[match[b]];
        }
}
```

Теперь нам надо выявить сам цикл — пройтись от вершин v и u до базы b цветка. Будет более удобно, если мы пока просто пометим в каком-то специальном массиве (назовём его $\mathrm{blossom}[]$) вершины, принадлежащие текущему цветку. После этого нам надо будет симитировать обход в ширину из псевдо-вершины — для этого достаточно положить в очередь обхода в ширину все вершины, лежащие на цикле цветка. Тем самым мы избежим явного объединения списков смежности.

Однако остаётся ещё одна проблема: корректное восстановление путей по окончании обхода в ширину. Для него мы сохраняли массив предков $\mathrm{p}[]$. Но после сжатия цветков возникает единственная проблема: обход в ширину продолжился сразу из всех вершин цикла, в том числе и нечётных, а массив предков у нас предназначался для восстановления путей по чётным вершинам. Более того, когда в сжатом графе найдётся увеличивающая цепь, проходящая через цветок, она вообще будет проходить по этому циклу в таком направлении, что в дереве путей это будет представляться движением вниз. Однако все эти проблемы изящно решаются таким манёвром: при сжатии цикла, проставим предков для всех его чётных вершин (кроме базы), чтобы эти "предки" указывали на соседнюю вершину в цикле. Для вершин u и v, если они также не базы, направим указатели предков друг на друга. В результате, если при восстановлении увеличивающего пути мы придём в цикл цветка в нечётную вершину, путь по предкам будет восстановлен корректно, и приведёт в базу цветка (из которой он уже дальше будет восстанавливаться нормально).



Итак, мы готовы реализовать сжатие цветка:

```
int v, u; // ребро (v,u), при рассмотрении которого был обнаружен цветок
int b = lca (v, u);
memset (blossom, 0, sizeof blossom);
mark_path (v, b, u);
mark_path (u, b, v);
```

где функция $\mathrm{mark_path}()$ проходит по пути от вершины до базы цветка, проставляет в специальном массиве $\mathrm{blossom}[]$ для них true и проставляет предков для чётных вершин. Параметр $\mathrm{children}$ — сын для самой вершины v (с помощью этого параметра мы замкнём цикл в предках).

```
void mark_path (int v, int b, int children) {
        while (base[v] != b) {
                blossom[base[v]] = blossom[base[match[v]]] = true;
                p[v] = children;
                children = match[v];
                v = p[match[v]];
        }
}
```

Наконец, реализуем основную функцию — $\mathrm{find_path}\ (\mathrm{int\ root})$, которая будет искать увеличивающий путь из свободной вершины root и возвращать последнюю вершину этого пути, либо -1, если увеличивающий путь не найден.

Вначале произведём инициализацию:

```
int find_path (int root) {
        memset (used, 0, sizeof used);
        memset (p, -1, sizeof p);
        for (int i=0; i<n; ++i)
                base[i] = i;
```

Далее идёт обход в ширину. Рассматривая очередное ребро (v, to), у нас есть несколько вариантов:

- Ребро несуществующее. Под этим мы подразумеваем, что v и to принадлежат одной сжатой псевдо-вершине ($\mathrm{base}[v] == \mathrm{base}[to]$), поэтому в текущем поверхностном графе этого ребра нет. Кроме этого случая, есть ещё один случай: когда ребро (v, to) уже принадлежит текущему паросочетанию; т.к. мы считаем, что вершина v является чётной вершиной, то проход по этому ребру означает в дереве путей подъём к предку вершины v, что недопустимо.

  ```
  if (base[v] == base[to] || match[v] == to)  continue;
  ```

- Ребро замыкает цикл нечётной длины, т.е. обнаруживается цветок. Как уже упоминалось выше, цикл нечётной длины обнаруживается при выполнении условия:

  ```
  if (to == root || match[to] != -1 && p[match[to]] != -1)
  ```

  В этом случае нужно выполнить сжатие цветка. Выше уже подробно разбирался этот процесс, здесь приведём его реализацию:

  ```
  int curbase = lca (v, to);
  memset (blossom, 0, sizeof blossom);
  mark_path (v, curbase, to);
  mark_path (to, curbase, v);
  for (int i=0; i<n; ++i)
          if (blossom[base[i]]) {
                  base[i] = curbase;
                  if (!used[i]) {
                          used[i] = true;
                          q[qt++] = i;
                  }
          }
  ```

- Иначе — это "обычное" ребро, поступаем как и в обычном поиске в ширину. Единственная тонкость — при проверке, что эту вершину мы ещё не посещали, надо смотреть не в массив used, а в массив p — именно он заполняется для посещённых нечётных вершин. Если мы в вершину to ещё не заходили, и она оказалась ненасыщенной, то мы нашли увеличивающую цепь, заканчивающуюся на вершине to, возвращаем её.

  ```
  if (p[to] == -1) {
          p[to] = v;
          if (match[to] == -1)
                  return to;
          to = match[to];
          used[to] = true;
  ```

```
        q[qt++] = to;
}
```

Итак, полная реализация функции $\mathrm{find_path}()$:

```
int find_path (int root) {
        memset (used, 0, sizeof used);
        memset (p, -1, sizeof p);
        for (int i=0; i<n; ++i)
                base[i] = i;

        used[root] = true;
        int qh=0, qt=0;
        q[qt++] = root;
        while (qh < qt) {
                int v = q[qh++];
                for (size_t i=0; i<g[v].size(); ++i) {
                        int to = g[v][i];
                        if (base[v] == base[to] || match[v] == to)  continue;
                        if (to == root || match[to] != -1 && p[match[to]] !=
-1) {
                                int curbase = lca (v, to);
                                memset (blossom, 0, sizeof blossom);
                                mark_path (v, curbase, to);
                                mark_path (to, curbase, v);
                                for (int i=0; i<n; ++i)
                                        if (blossom[base[i]]) {
                                                base[i] = curbase;
                                                if (!used[i]) {
                                                        used[i] = true;
                                                        q[qt++] = i;
                                                }
                                        }
                        }
                        else if (p[to] == -1) {
                                p[to] = v;
                                if (match[to] == -1)
                                        return to;
                                to = match[to];
                                used[to] = true;
                                q[qt++] = to;
                        }
                }
        }
        return -1;
}
```

Наконец, приведём определения всех глобальных массивов, и реализацию основной программы нахождения наибольшего паросочетания:

```
const int MAXN = ...; // максимально возможное число вершин во входном графе

int n;
vector<int> g[MAXN];
int match[MAXN], p[MAXN], base[MAXN], q[MAXN];
bool used[MAXN], blossom[MAXN];

...

int main() {
        ... чтение графа ...

        memset (match, -1, sizeof match);
        for (int i=0; i<n; ++i)
                if (match[i] == -1) {
                        int v = find_path (i);
                        while (v != -1) {
                                int pv = p[v],  ppv = match[pv];
                                match[v] = pv,  match[pv] = v;
```

```
                                v = ppv;
                        }
                }
}
```

Оптимизация: предварительное построение паросочетания

Как и в случае Алгоритма Куна, перед выполнением алгоритма Эдмондса можно каким-нибудь простым алгоритмом построить предварительное паросочетание. Например, таким жадным алгоритмом:

```
for (int i=0; i<n; ++i)
        if (match[i] == -1)
                for (size_t j=0; j<g[i].size(); ++j)
                        if (match[g[i][j]] == -1) {
                                match[g[i][j]] = i;
                                match[i] = g[i][j];
                                break;
                        }
```

Такая оптимизация значительно (до нескольких раз) ускорит работу алгоритма на случайных графах.

Случай двудольного графа

В двудольных графах отсутствуют циклы нечётной длины, и, следовательно, код, выполняющий сжатие цветков, никогда не выполнится. Удалив мысленно все части кода, обрабатывающие сжатие цветков, мы получим Алгоритм Куна практически в чистом виде. Таким образом, на двудольных графах алгоритм Эдмондса вырождается в алгоритм Куна и работает за $O(nm)$.

Дальнейшая оптимизация

Во всех вышеописанных операциях с цветками легко угадываются операции с непересекающимися множествами, которые можно выполнять намного эффективнее (см. Система непересекающихся множеств). Если переписать алгоритм с использованием этой структуры, то асимптотика алгоритма понизится до $O(nm)$. Таким образом, для произвольных графов мы получили ту же асимптотическую оценку, что и в случае двудольных графов (алгоритм Куна), но заметно более сложным алгоритмом.

Покрытие путями ориентированного ациклического графа

Дан ориентированный ациклический граф G. Требуется покрыть его наименьшим числом путей, т.е. найти наименьшее по мощности множество непересекающихся по вершинам простых путей, таких, что каждая вершина принадлежит какому-либо пути.

Сведение к двудольному графу

Пусть дан граф G с n вершинами. Построим соответствующий ему двудольный граф H стандартным образом, т.е.: в каждой доле графа H будет по n вершин, обозначим их через a_i и b_i соответственно. Тогда для каждого ребра (i, j) исходного графа G проведём соответствующее ребро (a_i, b_j).

Каждому ребру G соответствует одно ребро H, и наоборот. Если мы рассмотрим в G любой путь $P = (v_1, v_2, \ldots, v_k)$, то ему ставится в соответствие набор рёбер $(a_{v_1}, b_{v_2}), (a_{v_2}, b_{v_3}), ..., (a_{v_{k-1}}, b_{v_k})$.

Более просто для понимания будет, если мы добавим "обратные" рёбра, т.е. образуем граф $\overline{H}$ из графа H добавлением рёбер вида $(b_i, a_i), i = 1 \ldots N$. Тогда пути $P = (v_1, v_2, \ldots, v_k)$ в графе $\overline{H}$ будет соответствовать путь $\overline{Q} = (a_{v_1}, b_{v_2}, a_{v_2}, b_{v_3}, ..., a_{v_{k-1}}, b_{v_k})$.

Обратно, рассмотрим любой путь $\overline{Q}$ в графе $\overline{H}$, начинающийся в первой доле и заканчивающийся во второй доле. Очевидно, $\overline{Q}$ снова будет иметь вид $\overline{Q} = (a_{v_1}, b_{v_2}, a_{v_2}, b_{v_3}, ..., a_{v_{k-1}}, b_{v_k})$, и ему можно поставить в соответствие в графе G путь $P = (v_1, v_2, \ldots, v_k)$. Однако здесь есть одна тонкость: v_1 могло совпадать с v_k, поэтому путь P получился бы циклом. Однако по условию граф G ациклический, поэтому это вообще невозможно (это единственное место, где используется ацикличность графа G; тем не менее, на циклические графы описываемый здесь метод вообще нельзя обобщить).

Итак, всякому простому пути в графе $\overline{H}$, начинающемуся в первой доле и заканчивающемуся во второй, можно поставить в соответствие простой путь в графе G, и наоборот. Но заметим, что такой путь в графе $\overline{H}$ — это **паросочетание** в графе H. Таким образом, любому пути из G можно поставить в соответствие паросочетание в графе H, и наоборот. Более того, непересекающимся путям в G соответствуют непересекающиеся паросочетания в H.

Последний шаг. Заметим, что чем больше путей есть в нашем наборе, тем меньше все эти пути содержат рёбер. А именно, если есть p непересекающихся путей, покрывающих все n вершин графа, то они вместе содержат $r = n - p$ рёбер. Итак, чтобы минимизировать число путей, мы должны **максимизировать число рёбер** в них.

Итак, мы свели задачу к нахождению максимального паросочетания в двудольном графе H. После нахождения этого паросочетания (см. Алгоритм Куна) мы должны преобразовать его в набор путей в G (это делается тривиальным алгоритмом, неоднозначностей здесь не возникает). Некоторые вершины могут остаться ненасыщенными паросочетанием, в таком случае в ответ надо добавить пути нулевой длины из каждой из этих вершин.

Взвешенный случай

Взвешенный случай не сильно отличается от невзвешенного, просто в графе H на рёбрах появляются веса, и требуется найти уже паросочетание наименьшего веса. Восстанавливая ответ аналогично невзвешенному случаю, мы получим покрытие графа наименьшим числом путей, а при равенстве — наименьшим по стоимости.

Матрица Татта

Матрица Татта — это изящный подход к решению задачи о **паросочетании** в произвольном (не обязательно двудольном) графе. Правда, в простом виде алгоритм не выдаёт сами рёбра, входящие в паросочетание, а только размер максимального паросочетания в графе.

Ниже мы сначала рассмотрим результат, полученный Таттом (Tutte) для проверки существования совершенного паросочетания (т.е. паросочетания, содержащего $n/2$ рёбер, и потому насыщающего все n вершин). После этого мы рассмотрим результат, полученный позже Ловасом (Lovasz), который уже позволяет искать размер максимального паросочетания, а не только ограничивается случаем совершенного паросочетания. Затем приводится результат Рабина (Rabin) и Вазирани (Vazirani), которые указали алгоритм восстановления самого паросочетания (как набора входящих в него рёбер).

Определение

Пусть дан граф G с n вершинами (n — чётно).

Тогда **матрицей Татта** (Tutte) называется следующая матрица $n \times n$:

$$\begin{pmatrix} 0 & x_{12} & x_{13} & \dots & x_{1(n-1)} & x_{1n} \\ -x_{12} & 0 & x_{23} & \dots & x_{2(n-1)} & x_{2n} \\ -x_{13} & -x_{23} & 0 & \dots & x_{3(n-1)} & x_{3n} \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ -x_{1(n-1)} & -x_{2(n-1)} & -x_{3(n-1)} & \dots & 0 & x_{(n-1)n} \\ -x_{1n} & -x_{2n} & -x_{3n} & \dots & -x_{(n-1)n} & 0 \end{pmatrix}$$

где x_{ij} ($1 \le i < j \le n$) — это либо независимая переменная, соответствующая ребру между вершинами i и j, либо тождественный ноль, если ребра между этими вершинами нет.

Таким образом, в случае полного графа с n вершинами матрица Татта содержит $n(n-1)/2$ независимых переменных, если же в графе какие-то рёбра отсутствуют, то соответствующие элементы матрицы Татта превращаются в нули. Вообще, число переменных в матрице Татта совпадает с числом рёбер графа.

Матрица Татта антисимметрична (кососимметрична).

Теорема Татта

Рассмотрим определитель $\det(A)$ матрицы Татта. Это, вообще говоря, многочлен относительно переменных x_{ij}.

Теорема Татта гласит: в графе G существует совершенное паросочетание тогда и только тогда, когда многочлен $\det(A)$ не равен нулю тождественно (т.е. имеет хотя бы одно слагаемое с ненулевым коэффициентом). Напомним, что паросочетание называется совершенным, если оно насыщает все вершины, т.е. его мощность равна $n/2$.

Канадский математик Вильям Томас Татт (William Thomas Tutte) первым указал на тесную связь между паросочетаниями в графах и определителями матриц (1947 г.). Более простой вид этой связи позже обнаружил Эдмондс (Edmonds) в случае двудольных графов (1967 г.). Рандомизированные алгоритмы для нахождения величины максимального паросочетания и самих рёбер этого паросочетания были предложены позже, соответственно, Ловасом (Lovasz) (в 1979 г.), и Рабином (Rabin) и Вазирани (Vazirani) (в 1984 г.).

Практическое применение: рандомизированный алгоритм

Непосредственно применять теорему Татта даже в задаче проверки существования совершенного паросочетания нецелесообразно. Причиной этого является то, что при символьном вычислении определителя (т.е. в виде многочленов над переменными x_{ij}) промежуточные результаты являются многочленами, содержащими $O(n^2)$ переменных. Поэтому вычисление определителя матрицы Татта в символьном виде потребует неоправданно много времени.

Венгерский математик Ласло Ловас (Laszlo Lovasz) был первым, указавшим возможность применения здесь **рандомизированного** алгоритма для упрощения вычислений.

Идея очень проста: заменим все переменные x_{ij} случайными числами:

$$x_{ij} = \mathrm{rand}().$$

Тогда, если полином $\det(A)$ был тождественно нулевым, после такой замены он и будет оставаться нулевым; если же он был отличным от нуля, то при такой случайной числовой замене вероятность того, что он обратится в ноль, достаточно мала.

Понятно, что такой тест (подстановка случайных значений и вычисление определителя $\det(A)$) если и ошибается, то только в одну сторону: может сообщить об отсутствии совершенного паросочетания, когда на самом деле оно существует.

Мы можем повторить этот тест несколько раз, подставляя в качестве значений переменных новые случайные числа, и с каждым повторным запуском мы получаем всё большую уверенность в том, что тест выдал правильный ответ. На практике в большинстве случаев достаточно одного теста, чтобы определить, есть ли в графе совершенное паросочетание или нет; несколько таких тестов дают уже весьма высокую вероятность.

Для оценки **вероятности ошибки** можно использовать лемму Шварца-Зиппеля (Schwartz–Zippel), которая гласит, что вероятность обращения в ноль ненулевого полинома P k-ой степени при подстановке в качестве значений переменных случайных чисел, каждое из которых может принимать s вариантов значения, — эта вероятность удовлетворяет неравенству:

$$\Pr[P(r_1, \ldots, r_k) = 0] \le \frac{k}{s}.$$

Например, при использовании стандартной функции случайных чисел C++ $\mathrm{rand}()$ получаем, что эта вероятность при $n = 300$ составляет около процента.

Асимптотика решения получается равной $O(n^3)$ (с использованием, например, алгоритма Гаусса), умноженное на количество итераций теста. Стоит отметить, что по асимптотике такое решение значительно отстаёт от решения алгоритмом Эдмондса сжатия цветков, однако в некоторых случаях более предпочтительно из-за простоты реализации.

Восстановить само совершенное паросочетание как набор рёбер является более сложной задачей. Самым простым, хотя и медленным, вариантом будет восстановление этого паросочетания по одному ребру: перебираем первое ребро ответа, выбираем его так, чтобы в оставшемся графе существовало совершенное паросочетание, и т.д.

Доказательство теоремы Татта

Чтобы хорошо понять доказательство этой теоремы, сначала рассмотрим более простой результат, — полученный Эдмондсом для случая двудольных графов.

Теорема Эдмондса

Рассмотрим двудольный граф, в каждой доле которого по n вершин. Составим матрицу B $n \times n$, в которой, по аналогии с матрицей Татта, b_{ij} является отдельной независимой переменной, если ребро (i, j) присутствует в графе, и является тождественным нулём в противном случае.

Эта матрица похожа на матрицу Татта, однако матрица Эдмондса имеет вдвое меньшую разность, и каждому ребру здесь соответствует только одна ячейка матрицы.

Докажем следующую **теорему**: определитель $\det(B)$ отличен от нуля тогда и только тогда, когда в двудольном графе существует совершенное паросочетание.

Доказательство. Распишем определитель согласно его определению, как сумма по всем перестановкам:

$$\det(B) = \sum_{\pi \in S_n} \mathrm{sgn}(\pi) \cdot b_{1,\pi_1} \cdot b_{2,\pi_2} \cdot \ldots \cdot b_{n,\pi_n}.$$

Заметим, что поскольку все ненулевые элементы матрицы B — различные независимые переменные, то в этой сумме все ненулевые слагаемые различны, а потому никаких сокращений в процессе суммирования не происходит. Осталось заметить, что любое ненулевое слагаемое в этой сумме означает непересекающийся по вершинам набор рёбер, т.е. некоторое совершенное паросочетание. И наоборот, любому совершенному паросочетанию соответствует ненулевое слагаемое в этой сумме. Вкупе с вышесказанным это доказывает теорему.

Свойства антисимметричных матриц

Для доказательства теоремы Татта необходимо воспользоваться несколькими известными фактами линейной алгебры о свойствах антисимметричных матриц.

Во-первых (этот факт нам не пригодится, но он интересен сам по себе), если антисимметричная матрица имеет нечётный размер, то её определитель всегда равен нулю (теорема Якоби (Jacobi)). Для этого достаточно заметить, что антисимметричная матрица удовлетворяет равенству $A^T = -A$, и теперь получаем цепочку равенств:

$$\det(A) = \det(A^T) = \det(-A) = (-1)^n \det(A),$$

откуда и следует, что при нечётных n определитель необходимо должен быть равен нулю.

Во-вторых, оказывается, что в случае антисимметричных матриц чётного размера их определитель всегда можно записать как квадрат некоторого полинома относительно переменных-элементов этой матрицы (полином называется пфаффианом (pfaffian), а результат принадлежит Мьюру (Muir)):

$$\det(A) = \mathrm{Pf}^2(A).$$

В-третьих, этот пфаффиан представляет собой не произвольный многочлен, а сумму вида:

$$\mathrm{Pf}(A) = \frac{1}{2^n n!} \sum_{\pi \in S_n} \mathrm{sgn}(\pi) \cdot a_{\pi_1,\pi_2} \cdot a_{\pi_3,\pi_4} \cdot \ldots \cdot a_{\pi_{n-1},\pi_n}.$$

Таким образом, каждое слагаемое в пфаффиане — это произведение таких $n/2$ элементов матрицы, что их индексы в совокупности представляют собой разбиение множества n на $n/2$ пар. Перед каждым слагаемым имеется свой коэффициент, но его вид нас здесь не интересует.

Доказательство теоремы Татта

Воспользовавшись вторым и третьим свойством из предыдущего пункта, мы получаем, что определитель $\det(A)$ матрицы Татта представляет собой квадрат от суммы слагаемых такого вида, что каждое слагаемое — произведение элементов матрицы, индексы которых не повторяются и покрывают все номера от 1 до n. Таким образом, снова, как и в доказательстве теоремы Эдмондса, каждое ненулевое слагаемое этой суммы соответствует совершенному паросочетанию в графе, и наоборот.

Теорема Ловаса: обобщение для поиска размера максимального паросочетания

Формулировка

Ранг матрицы Татта совпадает с удвоенной величиной **максимального паросочетания** в данном графе.

Применение

Для применения этой теоремы на практике можно воспользоваться тем же самым приёмом рандомизации, что и в вышеописанном алгоритме для матрицы Татта, а именно: подставить вместо переменных случайные значения, и найти ранг полученной числовой матрицы. Ранг матрицы, опять же, ищется за $O(n^3)$ с помощью модифицированного алгоритма Гаусса, см. здесь.

Впрочем, следует отметить, что приведённая в предыдущем алгоритме лемма Шварца-Зиппеля неприменима в явном виде, и интуитивно кажется, что вероятность ошибки здесь становится выше. Однако утверждается (см. работы Ловаса (Lovasz)), что и здесь вероятность ошибки (т.е. того, что ранг полученной матрицы окажется меньше, чем удвоенный размер максимального паросочетания) не превосходит $\frac{n}{s}$ (где s, как и выше, обозначает размер множества, из которого выбираются случайные числа).

Доказательство

Доказательство будет вытекать из одного **свойства**, известного из линейной алгебры. Пусть дана антисимметричная матрица A размера $n \times n$, и пусть множества S и T — любые два подмножества множества $\{1, \ldots, n\}$, причём размеры этих множеств совпадают. Обозначим через A_{ST} матрицу, полученную из A только строками с номерами из S и столбцами с номерами из T. Тогда выполняется:

$$\det(A_{SS}) \cdot \det(A_{TT}) = \det(A_{ST}) \cdot \det(A_{TS}).$$

Покажем, как это свойство позволяет установить **соответствие** между рангом матрицы Татта A и величиной максимального паросочетания.

С одной стороны, рассмотрим в графе G некоторое максимальное паросочетание, и обозначим множество насыщаемых им вершин через U. Тогда определитель $\det(A_{UU})$ отличен от нуля (по теореме Татта). Следователь, ранг матрицы Татта как минимум не меньше удвоенной величины максимального паросочетания.

В обратную сторону, пусть ранг матрицы A равен r. Это означает, что нашлась такая подматрица A_{ST}, где $|S| = |T| = r$, определитель которой отличен от нуля. Но по приведённому выше свойству это означает, что одна из матриц A_{SS}, A_{TT} имеет ненулевой определитель, что по теореме Татта означает, что в подграфе, индуцированном множеством вершин S или T, имеется совершенное паросочетание (и величина го равна $r/2$). Следовательно, ранг матрицы не может быть больше величины максимального паросочетания, что

и завершает доказательство теоремы.

Алгоритм Рабина-Вазирани нахождения максимального паросочетания

Этот алгоритм является дальнейшим обобщением двух предыдущих теорем, и позволяет, в отличие от них, выдавать не только величину максимального паросочетания, но и сами рёбра, входящие в него.

Формулировка теоремы

Пусть в графе существует совершенное паросочетание. Тогда его матрица Татта невырождена, т.е. $\det(A) \neq 0$. Сгенерируем по ней, как было описано выше, случайную числовую матрицу B. Тогда, с высокой вероятностью, $(B^{-1})_{ji} \neq 0$ тогда и только тогда, когда ребро (i, j) входит в какое-либо совершенное паросочетание.

(Здесь через B^{-1} обозначена матрица, обратная к B. Предполагается, что определитель матрицы B отличен от нуля, поэтому обратная матрица существует.)

Применение

Эту теорему можно применять для восстановления самих рёбер максимального паросочетания. Сначала придётся выделить подграф, в котором содержится искомое максимальное паросочетание (это можно сделать параллельно с алгоритмом поиска ранга матрицы).

После этого задача сводится к поиску совершенного паросочетания по данной числовой матрице, полученной из матрицы Татта. Здесь мы уже применяем теорему Рабина-Вазирани, — находим обратную матрицу (что можно сделать модифицированным алгоритмом Гаусса за $O(n^3)$), находим в ней любой ненулевой элемент, удаляем из графа, и повторяем процесс. Асимптотика такого решения будет не самой быстрой — $O(n^4)$, зато взамен получаем простоту решения (по сравнению, например, с алгоритмом Эдмондса сжатия цветков).

Доказательство теоремы

Вспомним известную формулу для элементов обратной матрицы B^{-1}:

$$(B^{-1})_{ji} = \frac{\operatorname{adj}(B)_{i,j}}{\det(B)},$$

где через $\operatorname{adj}(B)_{i,j}$ обозначено алгебраическое дополнение, т.е. это число $(-1)^{i+j}$, умноженное на определитель матрицы, получаемой из B удалением i-й строки и j-го столбца.

Отсюда сразу получаем, что элемент $(B^{-1})_{ji}$ отличен от нуля тогда и только тогда, когда матрица B с вычеркнутыми i-ой строкой и j-ым столбцом имеет ненулевой определитель, что, применяя теорему Татта, означает с высокой вероятностью, что в графе без вершин i и j по-прежнему существует совершенное паросочетание.

Литература

- William Thomas Tutte. The Factorization of Linear Graphs [1946]
- Laszlo Lovasz. On Determinants, Matchings and Random Algorithms [1979]
- Laszlo Lovasz, M.D. Plummer. Matching Theory [1986]
- Michael Oser Rabin, Vijay V. Vazirani. Maximum matchings in general graphs through randomization [1989]
- Allen B. Tucker. Computer Science Handbook [2004]
- Rajeev Motwani, Prabhakar Raghavan. Randomized Algorithms [1995]
- A.C. Aitken. Determinants and matrices [1944]

Рёберная связность. Свойства и нахождение

Определение

Пусть дан неориентированный граф G с n вершинами и m рёбрами.

Рёберной связностью λ графа G называется наименьшее число рёбер, которое нужно удалить, чтобы граф перестал быть связным.

Например, для несвязного графа рёберная связность равна нулю. Для связного графа с единственным мостом рёберная связность равна единице.

Говорят, что множество S рёбер **разделяет** вершины s и t, если при удалении этих рёбер из графа вершины u и v оказываются в разных компонентах связности.

Ясно, что рёберная связность графа равна минимуму от наименьшего числа рёбер, разделяющих две вершины s и t, взятому среди всевозможных пар (s, t).

Свойства

Соотношение Уитни

Соотношение Уитни (Whitney) (1932 г.) между рёберной связностью λ, вершинной связностью κ и наименьшей из степеней вершин δ:

$$\kappa \le \lambda \le \delta.$$

Докажем это утверждение.

Докажем сначала первое неравенство: $\kappa \le \lambda$. Рассмотрим этот набор из λ рёбер, делающих граф несвязным. Если мы возьмём от каждого из этих ребёр по одному концу (любому из двух) и удалим из графа, то тем самым с помощью $\le \lambda$ удалённых вершин (поскольку одна и та же вершина могла встретиться дважды) мы сделаем граф несвязным. Таким образом, $\kappa \le \lambda$.

Докажем второе неравенство: $\lambda \le \delta$. Рассмотрим вершину минимальной степени, тогда мы можем удалить все δ смежных с ней рёбер и тем самым отделить эту вершину от всего остального графа. Следовательно, $\lambda \le \delta$.

Интересно, что неравенство Уитни **нельзя улучшить**: т.е. для любых троек чисел, удовлетворяющих этому неравенству, существует хотя бы один соответствующий граф. См. задачу "Построение графа с указанными величинами вершинной и рёберной связностей и наименьшей из степеней вершин".

Теорема Форда-Фалкерсона

Теорема Форда-Фалкерсона (1956 г.):

Для любых двух вершин наибольшее число рёберно-непересекающихся цепей, соединяющих их, равно наименьшему числу рёбер, разделяющих эти вершины.

Нахождение рёберной связности

Простой алгоритм на основе поиска максимального потока

Этот способ основан на теореме Форда-Фалекрсона.

Мы должны перебрать все пары вершин (s, t), и между каждой парой найти наибольшее число непересекающихся по рёбрам путей. Эту величину можно найти с помощью алгоритма максимального потока: мы делаем s истоком, t — стоком, а пропускную способность каждого ребра кладём равной 1.

Таким образом, псевдокод алгоритма таков:

```
int ans = INF;
```

```
for (int s=0; s<n; ++s)
        for (int t=s+1; t<n; ++t) {
                int flow = ... величина максимального потока из s в t ...
                ans = min (ans, flow);
        }
```

Асимптотика алгоритма при использовании \edmonds_karp{алгоритма Эдмондса-Карпа нахождения максимального потока} получается $O(n^2 \cdot nm^2) = O(n^3m^2)$, однако следует заметить, что скрытая в асимптотике константа весьма мала, поскольку практически невозможно создать такой граф, чтобы алгоритм нахождения максимального потока работал медленно сразу при всех стоках и истоках.

Особенно быстро такой алгоритм будет работать на случайных графах.

Специальный алгоритм

Используя потоковую терминологию, данная задача — это задача поиска **глобального минимального разреза**.

Для её решения разработаны специальные алгоритмы. На данном сайте представлен один из которых — алгоритм Штор-Вагнера, работающий за время $O(n^3)$ или $O(nm)$.

Литература

- Hassler Whitney. Congruent Graphs and the Connectivity of Graphs [1932]
- Фрэнк Харари. **Теория графов** [2003]

Рёберная связность. Свойства и нахождение

Определение

Пусть дан неориентированный граф G с n вершинами и m рёбрами.

Вершинной связностью λ графа G называется наименьшее число вершин, которое нужно удалить, чтобы граф перестал быть связным.

Например, для несвязного графа вершинная связность равна нулю. Для связного графа с единственной точкой сочленения вершинная связность равна единице. Для полного графа вершинную связность полагают равной $n-1$ (поскольку, какую пару вершин мы ни выберем, даже удаление всех остальных вершин не сделает их несвязными). Для всех графов, кроме полного, вершинная связность не превосходит $n-2$ — поскольку можно найти пару вершин, между которыми нет ребра, и удалить все остальные $n-2$ вершины.

Говорят, что множество S вершин **разделяет** вершины s и t, если при удалении этих вершин из графа вершины u и v оказываются в разных компонентах связности.

Ясно, что вершинная связность графа равна минимуму от наименьшего числа вершин, разделяющих две вершины s и t, взятому среди всевозможных пар (s,t).

Свойства

Соотношение Уитни

Соотношение Уитни (Whitney) (1932 г.) между рёберной связностью λ, вершинной связностью κ и наименьшей из степеней вершин δ:

$$\kappa \le \lambda \le \delta.$$

Докажем это утверждение.

Докажем сначала первое неравенство: $\kappa \le \lambda$. Рассмотрим этот набор из λ рёбер, делающих граф несвязным. Если мы возьмём от каждого из этих ребёр по одному концу (любому из двух) и удалим из графа, то тем самым с помощью $\le \lambda$ удалённых вершин (поскольку одна и та же вершина могла встретиться дважды) мы сделаем граф несвязным. Таким образом, $\kappa \le \lambda$.

Докажем второе неравенство: $\lambda \le \delta$. Рассмотрим вершину минимальной степени, тогда мы можем удалить все δ смежных с ней рёбер и тем самым отделить эту вершину от всего остального графа. Следовательно, $\lambda \le \delta$.

Интересно, что неравенство Уитни **нельзя улучшить**: т.е. для любых троек чисел, удовлетворяющих этому неравенству, существует хотя бы один соответствующий граф. См. задачу "Построение графа с указанными величинами вершинной и рёберной связностей и наименьшей из степеней вершин".

Нахождение вершинной связности

Переберём пару вершин s и t, и найдём минимальное количество вершин, которое надо удалить, чтобы разделить s и t.

Для этого **раздвоим** каждую вершину: т.е. у каждой вершины i создадим по две копии — одна i_1 для входящих рёбер, другая i_2 — для выходящих, и эти две копии связаны друг с другом ребром (i_1, i_2).

Каждое ребро (u,v) исходного графа в этой модифицированной сети превратится в два ребра: (u_2, v_1) и (v_2, u_1).

Всем рёбрам проставим пропускную способность, равную единице. Найдём теперь максимальный поток в этом графе между истоком s и стоком t. По построению графа, он и будет являться минимальным количеством вершин, необходимых для разделения s и t.

Таким образом, если для поиска максимального потока мы выберем алгоритм Эдмондса-Карпа, работающий за время $O(nm^2)$, то общая асимптотика алгоритма составит $O(n^3m^2)$. Впрочем, константа, скрытая в асимптотике, весьма мала: поскольку сделать граф, на котором алгоритмы бы работали долго при любой паре исток-сток, практически невозможно.

Построение графа с указанными величинами вершинной и рёберной связностей и наименьшей из степеней вершин

Даны величины κ, λ, δ — это, соответственно, вершинная связность, рёберная связность и наименьшая из степеней вершин графа. Требуется построить граф, который бы обладал указанными значениями, или сказать, что такого графа не существует.

Соотношение Уитни

Соотношение Уитни (Whitney) (1932 г.) между рёберной связностью λ, вершинной связностью κ и наименьшей из степеней вершин δ:

$$\kappa \leq \lambda \leq \delta.$$

Докажем это утверждение.

Докажем сначала первое неравенство: $\kappa \leq \lambda$. Рассмотрим этот набор из λ рёбер, делающих граф несвязным. Если мы возьмём от каждого из этих ребёр по одному концу (любому из двух) и удалим из графа, то тем самым с помощью $\leq \lambda$ удалённых вершин (поскольку одна и та же вершина могла встретиться дважды) мы сделаем граф несвязным. Таким образом, $\kappa \leq \lambda$.

Докажем второе неравенство: $\lambda \leq \delta$. Рассмотрим вершину минимальной степени, тогда мы можем удалить все δ смежных с ней рёбер и тем самым отделить эту вершину от всего остального графа. Следовательно, $\lambda \leq \delta$.

Интересно, что неравенство Уитни **нельзя улучшить**: т.е. для любых троек чисел, удовлетворяющих этому неравенству, существует хотя бы один соответствующий граф. Это мы докажем конструктивно, показав, как строятся соответствующие графы.

Решение

Проверим, удовлетворяют ли данные числа κ, λ и δ соотношению Уитни. Если нет, то ответа не существует.

В противном случае, построим сам граф. Он будет состоять из $2(\delta + 1)$ вершин, причём первые $\delta + 1$ вершины образуют полносвязный подграф, и вторые $\delta + 1$ вершины также образуют полносвязный подграф. Кроме того, соединим эти две части λ рёбрами так, чтобы в первой части эти рёбра были смежны λ вершинам, а в другой части — κ вершинам. Легко убедиться в том, что полученный граф будет обладать необходимыми характеристиками.

Обратная задача SSSP (inverse-SSSP - обратная задача кратчайших путей из одной вершины)

Имеется взвешенный неориентированный мультиграф G из N вершин и M рёбер. Дан массив P[1..N] и указана некоторая начальная вершина S. Требуется изменить веса рёбер так, чтобы для всех I P[I] было равно длине кратчайшего пути из S в I, причём сумма всех изменений (сумма модулей изменений весов рёбер) была бы наименьшей. Если этого сделать невозможно, то алгоритм должен выдать "No solution". Делать вес ребра отрицательным запрещено.

Описание решения

Мы решим эту задачу за линейное время, просто перебрав все рёбра (т.е. за один проход).

Пусть на текущем шаге мы рассматриваем ребро из вершины A в вершину B длиной R. Мы предполагаем, что для вершины A уже все условия выполнены (т.е. расстояние от S до A действительно равно P[A]), и будем проверять выполнение условий для вершины B. Имеем несколько вариантов ситуации:

- 1. `P[A] + R < P[B]`
  Это означает, что мы нашли путь, более короткий, чем он должен быть. Поскольку P[A] и P[B] мы изменять не можем, то мы обязаны удлинить текущее ребро (независимо от остальных рёбер), а именно выполнить:
  R += P[B] - P[A] - R.
  Кроме того, это означает, что мы нашли уже путь в вершину B из S, длина которого равна требуемому значению P[B], поэтому на последующих шагах нам не придётся укорачивать какие-либо рёбра (см. вариант 2).
- 2. `P[A] + R >= P[B]`
  Это означает, что мы нашли путь, более длинный, чем требуемый. Поскольку таких путей может быть несколько, мы должны выбрать среди всех таких путей (рёбер) то, которое потребует наименьшего изменения. Повторимся, что если мы удлиняли какое-то ребро, ведущее в вершину B (вариант 1), то этим мы фактически построили кратчайший путь в вершину B, а потому укорачивать никакое ребро уже не надо будет. Таким образом, для каждой вершины мы должны хранить ребро, которое собираемся укорачивать, т.е. ребро с наименьшим весом изменения.

Таким образом, просто перебрав все рёбра, и рассмотрев для каждого ребра ситуацию (за O(1)), мы решим обратную задачу SSSP за линейное время.

Если в какой-то момент мы пытаемся изменить уже изменённое ребро, то, очевидно, этого делать нельзя, и следует выдать "No solution". Кроме того, у некоторых вершин может быть так и не достигнута требуемая оценка кратчайшего пути, тогда ответ тоже будет "No solution". Во всех остальных случаях (кроме, конечно, явно некорректных значений в массиве P, т.е. P[S] != 0 или отрицательные значения) ответ будет существовать.

Реализация

Программа выводит "No solution", если решения нет, иначе выводит в первой строке минимальную сумму изменений весов рёбер, а в последующих M строках - новые веса рёбер.

```
const int INF = 1000*1000*1000;
int n, m;
vector<int> p (n);

bool ok = true;
vector<int> cost (m), cost_ch (m), decrease (n, INF), decrease_id (n, -1);
decrease[0] = 0;
for (int i=0; i<m; ++i) {
        int a, b, c; // текущее ребро (a,b) с ценой c
        cost[i] = c;

        for (int j=0; j<=1; ++j) {
                int diff = p[b] - p[a] - c;
                if (diff > 0) {
                        ok &= cost_ch[i] == 0 || cost_ch[i] == diff;
                        cost_ch[i] = diff;
                        decrease[b] = 0;
                }
                else
                        if (-diff <= c && -diff < decrease[b]) {
                                decrease[b] = -diff;
                                decrease_id[b] = i;
                        }
```

```
                swap (a, b);
        }
}

for (int i=0; i<n; ++i) {
        ok &= decrease[i] != INF;
        int r_id = decrease_id[i];
        if (r_id != -1) {
                ok &= cost_ch[r_id] == 0 || cost_ch[r_id] == -decrease[i];
                cost_ch[r_id] = -decrease[i];
        }
}

if (!ok)
        cout << "No solution";
else {
        long long sum = 0;
        for (int i=0; i<m; ++i)  sum += abs (cost_ch[i]);
        cout << sum << '\n';
        for (int i=0; i<m; ++i)
                printf ("%d ", cost[i] + cost_ch[i]);
}
```

Обратная задача MST (inverse-MST - обратная задача минимального остова) за O (N M^2)

Дан взвешенный неориентированный граф G с N вершинами и M рёбрами (без петель и кратных рёбер). Известно, что граф связный. Также указан некоторый остов T этого графа (т.е. выбрано N-1 ребро, которые образуют дерево с N вершинами). Требуется изменить веса рёбер таким образом, чтобы указанный остов T являлся минимальным остовом этого графа (точнее говоря, одним из минимальных остовов), причём сделать это так, чтобы суммарное изменение всех весов было наименьшим.

Решение

Сведём задачу inverse-MST к задаче min-cost-flow, точнее, **к задаче**, **двойственной** `min-cost-flow` (в смысле двойственности задач линейного программирования); затем решим последнюю задачу.

Итак, пусть дан граф G с N вершинами, M рёбрами. Вес каждого ребра обозначим через C_i. Предположим, не теряя общности, что рёбра с номерами с 1 по N-1 являются рёбрами T.

1. Необходимое и достаточное условие MST

Пусть дан некоторый остов S (не обязательно минимальный).

Введём сначала одно обозначение. Рассмотрим некоторое ребро j, не принадлежащее S. Очевидно, в графе S имеется единственный путь, соединяющий концы этого ребра, т.е. единственный путь, соединяющий концы ребра j и состоящий только из рёбер, принадлежащих S. **Обозначим через** `P[j]` множество рёбер, образующих этот путь для j-го ребра.

Для того, чтобы некоторый остов S являлся минимальным, **необходимо и достаточно**, чтобы:

```
Ci <= Cj для всех j ∉ S и каждого i ∈ P[j]
```

Можно заметить, что, поскольку **в нашей задаче** остову T принадлежат рёбра 1..N-1, то мы можем записать это условие таким образом:

```
Ci <= Cj для всех j = N..M и каждого i ∈ P[j]
(причём все i лежат в диапазоне 1..N-1)
```

2. Граф путей

Понятие графа путей непосредственно связано с предыдущей теоремой.

Пусть дан некоторый остов S (не обязательно минимальный).

Тогда **графом путей** H для графа G будет следующий граф:

- Он содержит M вершин, каждая вершина в H взаимно однозначно соответствует некоторому ребру в G.
- Граф H двудольный. В первой его доле находятся вершины i, которые соответствуют рёбрам в G, принадлежащим остову S. Соответственно, во второй доле находятся вершины j, которые соответствуют рёбрам, не принадлежащим S.
- Ребро проводится из вершины i в вершину j тогда и только тогда, когда i принадлежит P[j].
  Иными словами, для каждой вершины j из второй доли в неё входят рёбра из всех вершин первой доли, соответствующих множеству рёбер P[j].

В случае **нашей задачи** мы можем немного упростить описание графа путей:

```
ребро (i,j) существует в H, если i ∈ P[j],  j = N..M, i = 1..N-1
```

3. Математическая формулировка задачи

Чисто формально **задача** `inverse-MST` записывается таким образом:

```
найти массив A[1..M] такой, что
Ci + Ai <= Cj + Aj для всех j = N..M и каждого i ∈ P[j] (i в 1..N-1),
и минимизировать сумму |A1| + |A2| + ... + |Am|
```

здесь под искомым массивом A мы подразумеваем те значения, которые нужно добавить к весам рёбер (т.е., решив задачу inverse-MST, мы заменяем вес C_i каждого ребра i на величину $C_i + A_i$).

Очевидно, что нет смысла увеличивать вес рёбер, принадлежащих T, т.е.

```
Ai <= 0,  i = 1..N-1
```

и нет смысла укорачивать рёбра, не принадлежащие T:

```
Ai >= 0,  i = N..M
```

(поскольку в противном случае мы только ухудшим ответ)

Тогда мы можем немного **упростить** постановку задачи, убрав из суммы модули:

```
найти массив A[1..M] такой, что
Ci + Ai <= Cj + Aj для всех j = N..M и каждого i ∈ P[j] (i в 1..N-1),
Ai <= 0,  i = 1..N-1,
Ai >= 0,  i = N..M,
и минимизировать сумму An + ... + Am - (A1 + ... + An-1)
```

Наконец, просто изменим "минимизацию" на "максимизацию", а в самой сумме изменим все знаки на противоположные:

```
найти массив A[1..M] такой, что
Ci + Ai <= Cj + Aj для всех j = N..M и каждого i ∈ P[j] (i в 1..N-1),
Ai <= 0,  i = 1..N-1,
Ai >= 0,  i = N..M,
и максимизировать сумму A1 + ... + An-1 - (An + ... + Am)
```

4. Сведение задачи inverse-MST к задаче, двойственной задаче о назначениях

Формулировка задачи inverse-MST, которую мы только что дали, является формулировкой задачи **линейного программирования** с неизвестными $A_1..A_m$.

Применим классический приём - рассмотрим **двойственную** ей задачу.

По определению, чтобы получить двойственную задачу, нужно каждому неравенству сопоставить двойственную переменную X_{ij}, поменять ролями целевую функцию (которую нужно было минимизировать)

и коэффициенты в правых частях неравенств, поменять знаки "<=" на ">=" и наоборот, поменять максимизацию на минимизацию.

Итак, **двойственная к** inverse-MST задача:

```
найти все Xij для каждого (i,j) ∈ H, такие что:
все Xij >= 0,
для каждого i=1..N-1 ∑ Xij по всем j: (i,j) ∈ H <= 1,
для каждого j=N..M ∑ Xij по всем i: (i,j) ∈ H <= 1,
и минимизировать ∑ Xij (Cj - Ci) для всех (i,j) ∈ H
```

Последняя задача является **задачей о назначениях**: нам нужно в графе путей H выбрать несколько рёбер так, чтобы ни одно ребро не пересекалось с другим в вершине, а сумма весов рёбер (вес ребра (i,j) определим как C_j - C_i) должна быть наименьшей.

Таким образом, **двойственная задача** inverse-MST **эквивалентна задаче о назначениях**. Если мы научимся решать двойственную задачу о назначениях, то мы автоматически решим задачу inverse-MST.

5. Решение двойственной задачи о назначениях

Сначала уделим немного внимания тому частному случаю задачи о назначениях, который мы получили. Во-первых,

это несбалансированная задача о назначениях, поскольку в одной доле находится N-1 вершин, а в другой - M вершин, т. е. в общем случае число вершин во второй доле больше на целый порядок. Для решения такой двойственной задачи о назначениях есть специализированный алгоритм, который решит её за O (N^3), но здесь этот алгоритм рассматриваться не будет. Во-вторых, такую задачу о назначениях можно назвать задачей о назначениях с взвешенными вершинами: веса рёбер положим равными 0, вес каждой вершины из первой доли положим равным $-C_i$, из второй доли - равным C_j, и решение полученной задачи будет тем же самым.

Мы будем решать задачу двойственную задачу о назначениях с помощью **модифицированного алгоритма** `min-cost-flow`, который будет находить поток минимальной стоимости и одновременно решение двойственной задачи.

Свести задачу о назначениях к задаче min-cost-flow очень легко, но для полноты картины мы опишем этот процесс.

Добавим в граф исток s и сток t. Из s к каждой вершине первой доли проведём ребро с пропускной способностью = 1 и стоимостью = 0. Из каждой вершины второй доли проведём ребро к t с пропускной способностью = 1 и стоимостью = 0. Пропускные способности всех рёбер между первой и второй долями также положим равными 1.

Наконец, чтобы модифицированный алгоритм min-cost-flow (описанный ниже) работал, нужно **добавить ребро из** `s` **в** `t` с пропускной способностью = N+1 и стоимостью = 0.

6. Модифицированный алгоритм min-cost-flow для решения задачи о назначениях

Здесь мы рассмотрим **алгоритм последовательных кратчайших путей с потенциалами**, который напоминает обычный алгоритм min-cost-flow, но использует также понятие **потенциалов**, которые к концу работы алгоритма будут содержать **решение двойственной задачи**.

Введём обозначения. Для каждого ребра (i,j) обозначим через U_{ij} его пропускную способность, через C_{ij} - его стоимость, через F_{ij} - поток вдоль этого ребра.

Также введём понятие потенциалов. Каждая вершина обладает своим потенциалом PI_i. Остаточная стоимость ребра CPI_{ij} определяется как:

```
CPI_ij = C_ij - PI_i + PI_j
```

В любой момент работы алгоритма **потенциалы таковы**, что выполняются условия:

```
если F_ij = 0, то CPI_ij >= 0
если F_ij = U_ij, то CPI_ij <= 0
иначе CPI_ij = 0
```

Алгоритм начинает с нулевого потока, и нам нужно найти некоторые начальные значения потенциалов, которые бы удовлетворяли указанным условиям. Нетрудно проверить, что такой способ является одним из возможных решений:

```
PI_j = 0   для j = N..M
PI_i = min C_ij, где (i,j) ∈ H
PI_s = min PI_i, где i = 1..N-1
PI_t = 0
```

Собственно сам алгоритм min-cost-flow состоит из нескольких итераций. **На каждой итерации** мы находим кратчайший путь из s в t в остаточной сети, причём в качестве весов рёбер используем остаточные стоимости CPI. Затем мы увеличиваем поток вдоль найденного пути на единицу, и обновляем потенциалы следующим образом:

```
PI_i -= D_i
```

где D_i - найденное кратчайшее расстояние от s до i (повторимся, в остаточной сети с весами рёбер CPI).

Рано или поздно мы найдём тот путь из s в t, который состоит из единственного ребра (s,t). Тогда после этой итерации нам следует **завершить** работу алгоритма: действительно, если мы не остановим алгоритм, то дальше уже будут находиться пути с неотрицательной стоимостью, и добавлять их в ответ не надо.

К концу работы алгоритма мы получим решение задачи о назначениях (в виде потока F_{ij}) и решение двойственной задачи о назначениях (в массиве PI_i).

(с Pl_i надо будет провести небольшую модификацию: от всех значений Pl_i отнять Pl_s, поскольку его значения имеют смысл только при $Pl_s = 0$)

6. Итог

Итак, мы решили двойственную задачу о назначениях, а, следовательно, и задачу inverse-MST.

Оценим **асимптотику** получившегося алгоритма.

Сначала мы должны будем построить граф путей. Для этого просто для каждого ребра $j \notin T$ обходом в ширину по остову T найдём путь P[j]. Тогда граф путей мы построим за O (M) * O (N) = O (N M).

Затем мы найдём начальные значения потенциалов за O (N) * O (M) = O (N M).

Затем мы будем выполнять итерации min-cost-flow, всего итераций будет не более N (поскольку из истока выходит N рёбер, каждое с пропускной способностью = 1), на каждой итерации мы ищем в графе путей кратчайшие пути от истока до всех остальных вершин. Поскольку вершин в графе путей равно M+2, а число рёбер - O (N M), то, если реализовать поиск кратчайших путей простейшим вариантом алгоритма Дейкстры, каждая итерация min-cost-flow будет выполнять за O (M^2), а весь алгоритм min-cost-flow выполнится за O (N M^2).

Итоговая асимптотика алгоритма равна $O(N\ M^2)$.

Реализация

Реализуем весь вышеописанный алгоритм. Единственное изменение - вместо алгоритма Дейкстры применяется алгоритм Левита, который на многих тестах должен работать несколько быстрее.

```
const int INF = 1000*1000*1000;

struct rib {
        int v, c, id;
};

struct rib2 {
        int a, b, c;
};

int main() {

        int n, m;
        cin >> n >> m;
        vector < vector<rib> > g (n); // граф в формате списков смежности
        vector<rib2> ribs (m); // все рёбра в одном списке
        ... чтение графа ...

        int nn = m+2,  s = nn-2,  t = nn-1;
        vector < vector<int> > f (nn, vector<int> (nn));
        vector < vector<int> > u (nn, vector<int> (nn));
        vector < vector<int> > c (nn, vector<int> (nn));
        for (int i=n-1; i<m; ++i) {
                vector<int> q (n);
                int h=0, t=0;
                rib2 & cur = ribs[i];
                q[t++] = cur.a;
                vector<int> rib_id (n, -1);
                rib_id[cur.a] = -2;
                while (h < t) {
                        int v = q[h++];
                        for (size_t j=0; j<g[v].size(); ++j)
                                if (g[v][j].id >= n-1)
                                        break;
                                else if (rib_id [ g[v][j].v ] == -1) {
                                        rib_id [ g[v][j].v ] = g[v][j].id;
                                        q[t++] = g[v][j].v;
                                }
                }
                for (int v=cur.b, pv; v!=cur.a; v=pv) {
                        int r = rib_id[v];
                        pv = v != ribs[r].a ? ribs[r].a : ribs[r].b;
                        u[r][i] = n;
                        c[r][i] = ribs[i].c - ribs[r].c;
```

```
                        c[i][r] = -c[r][i];
                }
        }
        u[s][t] = n+1;
        for (int i=0; i<n-1; ++i)
                u[s][i] = 1;
        for (int i=n-1; i<m; ++i)
                u[i][t] = 1;

        vector<int> pi (nn);
        pi[s] = INF;
        for (int i=0; i<n-1; ++i) {
                pi[i] = INF;
                for (int j=n-1; j<m; ++j)
                        if (u[i][j])
                                pi[i] = min (pi[i], ribs[j].c-ribs[i].c);
                pi[s] = min (pi[s], pi[i]);
        }

        for (;;) {
                vector<int> id (nn);
                deque<int> q;
                q.push_back (s);
                vector<int> d (nn, INF);
                d[s] = 0;
                vector<int> p (nn, -1);
                while (!q.empty()) {
                        int v = q.front();  q.pop_front();
                        id[v] = 2;
                        for (int i=0; i<nn; ++i)
                                if (f[v][i] < u[v][i]) {
                                        int new_d = d[v] + c[v][i] - pi[v] +
pi[i];
                                        if (new_d < d[i]) {
                                                d[i] = new_d;
                                                if (id[i] == 0)
                                                        q.push_back (i);
                                                else if (id[i] == 2)
                                                        q.push_front (i);
                                                id[i] = 1;
                                                p[i] = v;
                                        }
                                }
                }
                for (int i=0; i<nn; ++i)
                        pi[i] -= d[i];
                for (int v=t; v!=s; v=p[v]) {
                        int pv = p[v];
                        ++f[pv][v],  --f[v][pv];
                }
                if (p[t] == s)  break;
        }

        for (int i=0; i<m; ++i)
                pi[i] -= pi[s];
        for (int i=0; i<n-1; ++i)
                if (f[s][i])
                        ribs[i].c += pi[i];
        for (int i=n-1; i<m; ++i)
                if (f[i][t])
                        ribs[i].c += pi[i];

        ... вывод графа ...

}
```

Покраска рёбер дерева

Это достаточно часто встречающаяся задача. Дано дерево G. Поступают запросы двух видов: первый вид - покрасить некоторое ребро, второй вид - запрос количества покрашенных рёбер между двумя вершинами.

Здесь будет описано достаточно простое решение (с использованием дерева отрезков), которое будет отвечать на запросы за O (log N), с препроцессингом (предварительной обработкой дерева) за O (M).

Решение

Для начала нам придётся реализовать LCA, чтобы каждый запрос второго вида (i,j) сводить к двум запросам (a,b), где a - предок b.

Теперь опишем **препроцессинг** собственно для нашей задачи. Запустим поиск в глубину из корня дерева, этот поиск в глубину составит некоторый список посещения вершин (каждая вершина добавляется в список, когда поиск заходит в неё, и каждый раз после того, как поиск в глубину возвращается из сына текущей вершины) - кстати говоря, этот же список используется алгоритмом LCA. В этом списке будет присутствовать каждое ребро (в том смысле, что если i и j - концы ребра, то в списке обязательно найдётся место, где i и j идут подряд друг за другом), причём присутствовать ровно 2 раза: в прямом направлении (из i в j, где вершина i ближе к корню, чем вершина j) и в обратном (из j в i).

Построим два дерева отрезков (для суммы, с единичной модификацией) по этому списку: T1 и T2. Дерево T1 будет учитывать каждое ребро только в прямом направлении, а дерево T2 - наоборот, только в обратном.

Пусть поступил очередной **запрос** вида (i,j), где i - предок j, и требуется определить, сколько рёбер покрашено на пути между i и j. Найдём i и j в списке обхода в глубину (нам обязательно нужны позиции, где они встречаются впервые), пусть это некоторые позиции p и q (это можно сделать за O (1), если вычислить эти позиции заранее во время препроцессинга). Тогда **ответом будет сумма** `T1[p..q-1]` - **сумма** `T2[p..q-1]`.

Почему? Рассмотрим отрезок [p;q] в списке обхода в глубину. Он содержит рёбра нужного нам пути из i в j, но также содержит и множество рёбер, которые лежат на других путях из i. Однако между нужными нам рёбрами и остальными рёбрами есть одно большое отличие: нужные рёбра будут содержаться в этом списке только один раз, причём в прямом направлении, а все остальные рёбра будут встречаться дважды: и в прямом, и в обратном направлении. Следовательно, разность T1[p..q-1] - T2[p..q-1] даст нам ответ (минус один нужно, потому что иначе мы захватим ещё лишнее ребро из вершины j куда-то вниз или вверх). Запрос суммы в дереве отрезков выполняется за O (log N).

Ответ на **запрос** вида 1 (о покраске какого-либо ребра) ещё проще - нам просто нужно обновить T1 и T2, а именно выполнить единичную модификацию того элемента, который соответствует нашему ребру (найти ребро в списке обхода, опять же, можно за O (1), если выполнить этот поиск в препроцессинге). Единичная модификация в дереве отрезков выполняется за O (log N).

Реализация

Здесь будет приведена полная реализация решения, включая LCA:

```
const int INF = 1000*1000*1000;

typedef vector < vector<int> > graph;

vector<int> dfs_list;
vector<int> ribs_list;
vector<int> h;

void dfs (int v, const graph & g, const graph & rib_ids, int cur_h = 1)
{
        h[v] = cur_h;
        dfs_list.push_back (v);
        for (size_t i=0; i<g[v].size(); ++i)
                if (h[g[v][i]] == -1)
                {
                        ribs_list.push_back (rib_ids[v][i]);
                        dfs (g[v][i], g, rib_ids, cur_h+1);
                        ribs_list.push_back (rib_ids[v][i]);
                        dfs_list.push_back (v);
                }
}

vector<int> lca_tree;
```

```
vector<int> first;

void lca_tree_build (int i, int l, int r)
{
        if (l == r)
                lca_tree[i] = dfs_list[l];
        else
        {
                int m = (l + r) >> 1;
                lca_tree_build (i+i, l, m);
                lca_tree_build (i+i+1, m+1, r);
                int lt = lca_tree[i+i],  rt = lca_tree[i+i+1];
                lca_tree[i] = h[lt] < h[rt] ? lt : rt;
        }
}

void lca_prepare (int n)
{
        lca_tree.assign (dfs_list.size() * 8, -1);
        lca_tree_build (1, 0, (int)dfs_list.size()-1);

        first.assign (n, -1);
        for (int i=0; i < (int)dfs_list.size(); ++i)
        {
                int v = dfs_list[i];
                if (first[v] == -1)  first[v] = i;
        }
}

int lca_tree_query (int i, int tl, int tr, int l, int r)
{
        if (tl == l && tr == r)
                return lca_tree[i];
        int m = (tl + tr) >> 1;
        if (r <= m)
                return lca_tree_query (i+i, tl, m, l, r);
        if (l > m)
                return lca_tree_query (i+i+1, m+1, tr, l, r);
        int lt = lca_tree_query (i+i, tl, m, l, m);
        int rt = lca_tree_query (i+i+1, m+1, tr, m+1, r);
        return h[lt] < h[rt] ? lt : rt;
}

int lca (int a, int b)
{
        if (first[a] > first[b])  swap (a, b);
        return lca_tree_query (1, 0, (int)dfs_list.size()-1, first[a],
first[b]);
}


vector<int> first1, first2;
vector<char> rib_used;
vector<int> tree1, tree2;

void query_prepare (int n)
{
        first1.resize (n-1, -1);
        first2.resize (n-1, -1);
        for (int i = 0; i < (int) ribs_list.size(); ++i)
        {
                int j = ribs_list[i];
                if (first1[j] == -1)
                        first1[j] = i;
                else
                        first2[j] = i;
        }

        rib_used.resize (n-1);
        tree1.resize (ribs_list.size() * 8);
        tree2.resize (ribs_list.size() * 8);
```

```
}

void sum_tree_update (vector<int> & tree, int i, int l, int r, int j, int delta)
{
        tree[i] += delta;
        if (l < r)
        {
                int m = (l + r) >> 1;
                if (j <= m)
                        sum_tree_update (tree, i+i, l, m, j, delta);
                else
                        sum_tree_update (tree, i+i+1, m+1, r, j, delta);
        }
}

int sum_tree_query (const vector<int> & tree, int i, int tl, int tr, int l,
int r)
{
        if (l > r || tl > tr)  return 0;
        if (tl == l && tr == r)
                return tree[i];
        int m = (tl + tr) >> 1;
        if (r <= m)
                return sum_tree_query (tree, i+i, tl, m, l, r);
        if (l > m)
                return sum_tree_query (tree, i+i+1, m+1, tr, l, r);
        return sum_tree_query (tree, i+i, tl, m, l, m)
                + sum_tree_query (tree, i+i+1, m+1, tr, m+1, r);
}

int query (int v1, int v2)
{
        return sum_tree_query (tree1, 1, 0, (int)ribs_list.size()-1, first
[v1], first[v2]-1)
                - sum_tree_query (tree2, 1, 0, (int)ribs_list.size()-1,
first[v1], first[v2]-1);
}


int main()
{
        // чтение графа
        int n;
        scanf ("%d", &n);
        graph g (n), rib_ids (n);
        for (int i=0; i<n-1; ++i)
        {
                int v1, v2;
                scanf ("%d%d", &v1, &v2);
                --v1, --v2;
                g[v1].push_back (v2);
                g[v2].push_back (v1);
                rib_ids[v1].push_back (i);
                rib_ids[v2].push_back (i);
        }

        h.assign (n, -1);
        dfs (0, g, rib_ids);
        lca_prepare (n);
        query_prepare (n);

        for (;;) {
                if () {
                        // запрос о покраске ребра с номером x;
                        //    если start=true, то ребро красится,
иначе покраска снимается
                        rib_used[x] = start;
                        sum_tree_update (tree1, 1, 0, (int)ribs_list.size()-
1, first1[x], start?1:-1);
                        sum_tree_update (tree2, 1, 0, (int)ribs_list.size()-
1, first2[x], start?1:-1);
```

```
                }
                else {
                        // запрос кол-ва покрашенных рёбер на пути между v1 и v2
                        int l = lca (v1, v2);
                        int result = query (l, v1) + query (l, v2);
                        // result - ответ на запрос
                }
        }

}
```

Задача 2-SAT

Задача 2-SAT (2-satisfiability) - это задача распределения значений булевым переменным таким образом, чтобы они удовлетворяли всем наложенным ограничениям.

Задачу 2-SAT можно представить в виде конъюнктивной нормальной формы, где в каждом выражении в скобках стоит ровно по две переменной; такая форма называется 2-CNF (2-conjunctive normal form). Например:

```
(a || c) && (a || !d) && (b || !d) && (b || !e) && (c || d)
```

Приложения

Алгоритм для решения 2-SAT может быть применим во всех задачах, где есть набор величин, каждая из которых может принимать 2 возможных значения, и есть связи между этими величинами:

- **Расположение текстовых меток на карте или диаграмме**.
  Имеется в виду нахождение такого расположения меток, при котором никакие две не пересекаются.
  Стоит заметить, что в общем случае, когда каждая метка может занимать множество различных позиций, мы получаем задачу general satisfiability, которая является NP-полной. Однако, если ограничиться только двумя возможными позициями, то полученная задача будет задачей 2-SAT.
- **Расположение рёбер при рисовании графа**.
  Аналогично предыдущему пункту, если ограничиться только двумя возможными способами провести ребро, то мы придём к 2-SAT.
- **Составление расписания игр**.
  Имеется в виду такая система, когда каждая команда должна сыграть с каждой по одному разу, а требуется распределить игры по типу домашняя-выездная, с некоторыми наложенными ограничениями.
- и т.д.

Алгоритм

Сначала приведём задачу к другой форме - так называемой импликативной форме. Заметим, что выражение вида a || b эквивалентно !a => b или !b => a. Это можно воспринимать следующим образом: если есть выражение a || b, и нам необходимо добиться обращения его в true, то, если a=false, то необходимо b=true, и наоборот, если b=false, то необходимо a=true.

Построим теперь так называемый **граф импликаций**: для каждой переменной в графе будет по две вершины, обозначим их через x_i и $!x_i$. Рёбра в графе будут соответствовать импликативным связям.

Например, для 2-CNF формы:

```
(a || b) && (b || !c)
```

Граф импликаций будет содержать следующие рёбра (ориентированные):

```
!a => b
!b => a
!b => !c
c => b
```

Стоит обратить внимание на такое свойство графа импликаций, что если есть ребро a => b, то есть и ребро !b => !a.

Теперь заметим, что если для какой-то переменной x выполняется, что из x достижимо !x, а из !x достижимо x, то задача решения не имеет. Действительно, какое бы значение для переменной x мы бы ни выбрали, мы всегда придём к противоречию - что должно быть выбрано и обратное ему значение. Оказывается, что это условие является не только достаточным, но и необходимым (доказательством этого факта будет описанный ниже алгоритм). Переформулируем данный критерий в терминах теории графов. Напомним, что если из одной вершины достижима другая, а из той вершины достижима первая, то эти две вершины находятся в одной сильно связной компоненте. Тогда мы можем сформулировать **критерий существования решения** следующим образом:

Для того, чтобы данная задача 2-SAT **имела решение**, необходимо и достаточно, чтобы для любой переменной x вершины x и !x находились **в разных компонентах сильной связности** графа импликаций.

Этот критерий можно проверить за время O (N + M) с помощью алгоритма поиска сильно связных компонент.

Теперь построим собственно **алгоритм** нахождения решения задачи 2-SAT в предположении, что решение существует.

Заметим, что, несмотря на то, что решение существует, для некоторых переменных может выполняться, что из

x достижимо !x, или (но не одновременно), из !x достижимо x. В таком случае выбор одного из значений переменной x будет приводить к противоречию, в то время как выбор другого - не будет. Научимся выбирать из двух значений то, которое не приводит к возникновению противоречий. Сразу заметим, что, выбрав какое-либо значение, мы должны запустить из него обход в глубину/ширину и пометить все значения, которые следуют из него, т.е. достижимы в графе импликаций. Соответственно, для уже помеченных вершин никакого выбора между x и !x делать не нужно, для них значение уже выбрано и зафиксировано. Нижеописанное правило применяется только к непомеченным ещё вершинам.

Утверждается следующее. Пусть comp[v] обозначает номер компоненты сильной связности, которой принадлежит вершина v, причём номера упорядочены в порядке топологической сортировки компонент сильной связности в графе компонентов (т.е. более ранним в порядке топологической сортировки соответствуют большие номера: если есть путь из v в w, то comp[v] <= comp[w]). Тогда, если comp[x] < comp[!x], то выбираем значение !x, иначе, т. е. если comp[x] > comp[!x], то выбираем x.

Докажем, что при таком выборе значений мы не придём к противоречию. Пусть, для определённости, выбрана вершина x (случай, когда выбрана вершина !x, доказывается симметрично).

Во-первых, докажем, что из x не достижимо !x. Действительно, так как номер компоненты сильной связности comp [x] больше номера компоненты comp[!x], то это означает, что компонента связности, содержащая x, расположена левее компоненты связности, содержащей !x, и из первой никак не может быть достижима последняя.

Во-вторых, докажем, что никакая вершина y, достижимая из x, не является "плохой", т.е. неверно, что из y достижимо !y. Докажем это от противного. Пусть из x достижимо y, а из y достижимо !y. Так как из x достижимо y, то, по свойству графа импликаций, из !y будет достижимо !x. Но, по предположению, из y достижимо !y. Тогда мы получаем, что из x достижимо !x, что противоречит условию, что и требовалось доказать.

Итак, мы построили алгоритм, который находит искомые значения переменных в предположении, что для любой переменной x вершины x и !x находятся в разных компонентах сильной связности. Выше показали корректность этого алгоритма. Следовательно, мы одновременно доказали указанный выше критерий существования решения.

Теперь мы можем собрать **весь алгоритм** воедино:

- Построим граф импликаций.
- Найдём в этом графе компоненты сильной связности за время O (N + M), пусть comp[v] - это номер компоненты сильной связности, которой принадлежит вершина v.
- Проверим, что для каждой переменной x вершины x и !x лежат в разных компонентах, т.е. comp[x] ≠ comp[!x]. Если это условие не выполняется, то вернуть "решение не существует".
- Если comp[x] > comp[!x], то переменной x выбираем значение true, иначе - false.

Реализация

Ниже приведена реализация решения задачи 2-SAT для уже построенного графа импликаций g и обратного ему графа gt (т.е. в котором направление каждого ребра изменено на противоположное).

Программа выводит номера выбранных вершин, либо фразу "NO SOLUTION", если решения не существует.

```
int n;
vector < vector<int> > g, gt;
vector<bool> used;
vector<int> order, comp;

void dfs1 (int v) {
        used[v] = true;
        for (size_t i=0; i<g[v].size(); ++i) {
                int to = g[v][i];
                if (!used[to])
                        dfs1 (to);
        }
        order.push_back (v);
}

void dfs2 (int v, int cl) {
        comp[v] = cl;
        for (size_t i=0; i<gt[v].size(); ++i) {
                int to = gt[v][i];
                if (comp[to] == -1)
                        dfs2 (to, cl);
        }
}

int main() {
        ... чтение n, графа g, построение графа gt ...

        used.assign (n, false);
```

```
	for (int i=0; i<n; ++i)
		if (!used[i])
			dfs1 (i);

	comp.assign (n, -1);
	for (int i=0, j=0; i<n; ++i) {
		int v = order[n-i-1];
		if (comp[v] == -1)
			dfs2 (v, j++);
	}

	for (int i=0; i<n; ++i)
		if (comp[i] == comp[i^1]) {
			puts ("NO SOLUTION");
			return 0;
		}
	for (int i=0; i<n; ++i) {
		int ans = comp[i] > comp[i^1] ? i : i^1;
		printf ("%d ", ans);
	}

}
```

Heavy-light декомпозиция

Heavy-light **декомпозиция** — это достаточно общий приём, который позволяет эффективно решать многие задачи, сводящиеся к **запросам на дереве**.

Простейший **пример** задач такого вида — это следующая задача. Дано дерево, каждой вершине которого приписано какое-то число. Поступают запросы вида (a, b), где a и b — номера вершин дерева, и требуется узнать максимальное число на пути между вершинами a и b.

Описание алгоритма

Итак, пусть дано дерево G с n вершинами, подвешенное за некоторый корень.

Суть этой декомпозиции в том, чтобы **разбить дерево на несколько путей** таким образом, чтобы для любой вершины v получалось, что если мы будем подниматься от v к корню, то по пути сменим не более $\log n$ путей. Кроме того, все пути должны не пересекаться друг с другом по рёбрам.

Понятно, что если мы научимся искать такую декомпозицию для любого дерева, это позволит свести любой запрос вида "узнать что-то на пути из a в b" к нескольким запросам вида "узнать что-то на отрезке $[l; r]$ k-го пути".

Построение heavy-light декомпозиции

Посчитаем для каждой вершины v размер её поддерева $s(v)$ (т.е. это количество вершин в поддереве вершины v, включая саму вершину).

Далее, рассмотрим все рёбра, ведущие к сыновьям какой-либо вершины v. Назовём ребро (v, c) **тяжёлым**, если оно ведёт в вершину c такую, что:

$$s(c) \geq \frac{s(v)}{2} \quad \Leftrightarrow \quad \text{edge } (v, c) \text{ is heavy.}$$

Все остальные рёбра назовём **лёгкими**. Очевидно, что из одной вершины v вниз может исходить максимум одно тяжёлое ребро (т.к. в противном случае у вершины v было бы два сына размера $s(v)/2$, что с учётом самой вершины v даёт размер $2 \cdot s(v)/2 + 1 > s(v)$, т.е. пришли к противоречию).

Теперь построим саму **декомпозицию** дерева на непересекающиеся пути. Рассмотрим все вершины, из которых не выходит вниз ни одного тяжёлого ребра, и будем идти от каждой из них вверх, пока не дойдём до корня дерева или не пройдём лёгкое ребро. В результате мы получим несколько путей — покажем, что это и есть искомые пути heavy-light декомпозиции.

Доказательство корректности алгоритма

Во-первых, заметим, что полученные алгоритмом пути будут **непересекающимися**. В самом деле, если бы два каких-то пути имели бы общее ребро, это бы означало, что из какой-то вершины исходит вниз два тяжёлых ребра, чего быть не может.

Во-вторых, покажем, что спускаясь от корня дерева до произвольной вершины, мы **сменим по пути не более $\log n$ путей**. В самом деле, проход вниз по лёгкому ребру уменьшает размер текущего поддерева более чем вдвое:

$$s(c) < \frac{s(v)}{2} \quad \Leftrightarrow \quad \text{edge } (v, c) \text{ is light.}$$

Таким образом, мы не могли пройти более $\log n$ лёгких рёбер. Однако переходить с одного пути на другой мы можем только через лёгкое ребро (т.к. каждый путь, кроме заканчивающихся в корне, содержит лёгкое ребро в конце; а попасть сразу посередине пути мы не можем).

Следовательно, по пути от корня до любой вершины мы не можем сменить более $\log n$ путей, что и требовалось доказать.

Применения при решении задач

При решении задач иногда бывает удобнее рассматривать heavy-light как набор **вершинно-непересекающихся** путей (а не рёберо-непересекающихся). Для этого достаточно из каждого

пути исключить последнее ребро, если оно являются лёгким ребром — тогда никакие свойства не нарушатся, но теперь каждая вершина будет принадлежать ровно одному пути.

Ниже мы рассмотрим несколько типичных задач, которые можно решать с помощью heavy-light декомпозиции.

Отдельно стоит обратить внимание на задачу **сумма чисел на пути**, поскольку это пример задачи, которая может быть решена и более простыми техниками.

Максимальное число на пути между двумя вершинами

Дано дерево, каждой вершине которого приписано какое-то число. Поступают запросы вида (a, b), где a и b — номера вершин дерева, и требуется узнать максимальное число на пути между вершинами a и b.

Построим заранее heavy-light декомпозицию. Над каждым получившимся путём построим дерево отрезков для максимума, что позволит искать вершину с максимальным приписанным числом в указанном сегменте указанного пути за $O(\log n)$. Хотя число путей в heavy-light декомпозиции может достигать $n - 1$, суммарный размер всех путей есть величина $O(n)$, поэтому и суммарный размер деревьев отрезков также будет линейным.

Теперь, для того чтобы отвечать на поступивший запрос (a, b) найдём наименьшего общего предка l этих вершин (например, методом двоичного подъёма). Теперь задача свелась к двум запросам: (a, l) и (b, l), на каждый из которых мы можем ответить таким образом: найдём, в каком пути лежит нижняя вершина, сделаем запрос к этому пути, перейдём в вершину-конец этого пути, снова определим, в каком мы пути оказались и сделаем запрос к нему, и так далее, пока не дойдём до пути, содержащего l.

Аккуратно следует быть со случаем, когда, например, a и l оказались в одном пути — тогда запрос максимума к этому пути надо делать не на суффиксе, а на внутреннем подотрезке.

Таким образом, в процессе ответа на один подзапрос мы пройдём по $O(\log n)$ путям, в каждом из них сделав запрос максимума на суффиксе или на префиксе/подотрезке (запрос на префиксе/подотрезке мог быть только один раз).

Так мы получили решение за $O(\log^2 n)$ на один запрос.

Если ещё дополнительно предпосчитать на каждом пути максимумы на всех суффиксах, то получится решение за $O(n \log n)$ — т.к. запрос максимума не на суффиксе случается только один раз, когда мы доходим до вершины l.

Сумма чисел на пути между двумя вершинами

Дано дерево, каждой вершине которого приписано какое-то число. Поступают запросы вида (a, b), где a и b — номера вершин дерева, и требуется узнать сумму чисел на пути между вершинами a и b. Возможен вариант этой задачи, когда дополнительно бывают запросы изменения числа, приписанного той или иной вершине.

Хотя эту задачу можно решать с помощью heavy-light декомпозиции, построив над каждым путём дерево отрезков для суммы (или просто предпосчитав частичные суммы, если в задаче отсутствуют запросы изменения), эта задача может быть решена **более простыми техниками**.

Если запросы модификации отсутствуют, то узнавать сумму на пути между двумя вершинами можно параллельно с поиском LCA двух вершин в алгоритме двоичного подъёма — для этого достаточно во время препроцессинга для LCA подсчитывать не только 2^k-ых предков каждой вершины, но и сумму чисел на пути до этого предка.

Есть и принципиально другой подход к этой задаче — рассмотреть эйлеров обход дерева, и построить дерево отрезков над ним. Этот алгоритм рассматривается в статье с решением похожей задачи. (А если запросы модификации отсутствуют — то достаточно обойтись предпосчётом частичных сумм, без дерева отрезков.)

Оба этих способа дают относительно простые решения с асимптотикой $O(\log n)$ на один запрос.

Перекраска рёбер пути между двумя вершинами

Дано дерево, каждое ребро изначально покрашено в белый цвет. Поступают запросы вида (a, b, c), где a и b — номера вершин, c — цвет, что означает, что все рёбра на пути из a в b надо перекрасить в цвет c. Требуется после всех перекрашиваний сообщить, сколько в итоге получилось рёбер каждого цвета.

Решение — просто сделать дерево отрезков с покраской на отрезке над набором путей heavy-light декомпозиции.

Каждый запрос перекраски на пути (a, b) превратится в два подзапроса (a, l) и (b, l), где l — наименьший общий предок вершин a и b (найденный, например, алгоритмом двоичного подъёма), а каждый из этих подзапросов — в $O(\log n)$ запросов к деревьям отрезков над путями.

Итого получается решение с асимптотикой $O(\log^2 n)$ на один запрос.

Задачи в online judges

Список задач, которые можно решить, используя heavy-light декомпозицию:

- TIMUS #1553 "Caves and Tunnels" [сложность: средняя]
- IPSC 2009 L "Let there be rainbows!" [сложность: средняя]
- SPOJ #2798 "Query on a tree again!" [сложность: средняя]
- Codeforces Beta Round #88 E "**Дерево или не дерево**" [сложность: высокая]

Длина объединения отрезков на прямой за O (N log N)

Даны N отрезков на прямой, т.е. каждый отрезок задаётся парой координат (X1, X2). Рассмотрим объединение этих отрезков и найдём его длину.

Алгоритм был предложен Кли (Klee) в 1977 году. Алгоритм работает за O (N log N). Было доказано, что этот алгоритм является быстрейшим (асимптотически).

Описание

Положим все координаты концов отрезков в массив X и отсортируем его по значению координаты. Дополнительное условие при сортировке - при равенстве координат первыми должны идти левые концы. Кроме того, для каждого элемента массива будем хранить, относится он к левому или к правому концу отрезка. Теперь пройдёмся по всему массиву, имея счётчик C перекрывающихся отрезков. Если C отлично от нуля, то к результату добавляем разницу X_i - X_{i-1}. Если текущий элемент относится к левому концу, то увеличиваем счётчик C, иначе уменьшаем его.

Реализация

```
unsigned segments_union_measure (const vector <pair <int,int> > & a)
{
        unsigned n = a.size();
        vector <pair <int,bool> > x (n*2);
        for (unsigned i=0; i<n; i++)
        {
                x[i*2] = make_pair (a[i].first, false);
                x[i*2+1] = make_pair (a[i].second, true);
        }

        sort (x.begin(), x.end());

        unsigned result = 0;
        unsigned c = 0;
        for (unsigned i=0; i<n*2; i++)
        {
                if (c && i)
                        result += unsigned (x[i].first - x[i-1].first);
                if (x[i].second)
                        ++c;
                else
                        --c;
        }
        return result;
}
```

Знаковая площадь треугольника и предикат "По часовой стрелке"

Определение

Пусть даны три точки p_1, p_2, p_3. Найдём значение **знаковой площади** S треугольника $p_1 p_2 p_3$, т.е. площади этого треугольника, взятой со знаком плюс или минус в зависимости от типа поворота, образуемого точками p_1, p_2, p_3: против часовой стрелки или по ней соответственно.

Понятно, что, если мы научимся вычислять такую знаковую ("ориентированную") площадь, то сможем и находить обычную площадь любого треугольника, а также сможем проверять, по часовой стрелке или против направлена какая-либо тройка точек.

Вычисление

Воспользуемся понятием **косого** (псевдоскалярного) произведения векторов. Оно как раз равно удвоенной знаковой площади треугольника:

$$a \wedge b = |a||b| \sin \angle(a, b) = 2S,$$

где угол $\angle(a, b)$ берётся ориентированным, т.е. это угол вращения между этими векторами против часовой стрелки.

(Модуль косого произведения двух векторов равен модулю **векторного** произведения их.)

Косое произведение вычисляется как величина определителя, составленного из координат точек:

$$2S = \begin{vmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x_3 & y_3 & 1 \end{vmatrix}.$$

Раскрывая определитель, можно получить такую формулу:

$$2S = x_1(y_2 - y_3) + x_2(y_3 - y_1) + x_3(y_1 - y_2).$$

Можно сгруппировать третье слагаемое с первыми двумя, избавившись от одного умножения:

$$2S = (x_2 - x_1)(y_3 - y_1) - (y_2 - y_1)(x_3 - x_1).$$

Последнюю формулу удобно записывать и запоминать в матричном виде, как следующий определитель:

$$2S = \begin{vmatrix} x_2 - x_1 & y_2 - y_1 \\ x_3 - x_1 & y_3 - y_1 \end{vmatrix}.$$

Реализация

Функция, вычисляющая удвоенную знаковую площадь треугольника:

```
int triangle_area_2 (int x1, int y1, int x2, int y2, int x3, int y3) {
        return (x2 - x1) * (y3 - y1) - (y2 - y1) * (x3 - x1);
}
```

Функция, возвращающая обычную площадь треугольника:

```
double triangle_area (int x1, int y1, int x2, int y2, int x3, int y3) {
        return abs (triangle_area_2 (x1, y1, x2, y2, x3, y3)) / 2.0;
}
```

Функция, проверяющая, образует ли указанная тройка точек поворот по часовой стрелке:

```
bool clockwise (int x1, int y1, int x2, int y2, int x3, int y3) {
        return triangle_area_2 (x1, y1, x2, y2, x3, y3) < 0;
}
```

Функция, проверяющая, образует ли указанная тройка точек поворот против часовой стрелки:

```
bool counter_clockwise (int x1, int y1, int x2, int y2, int x3, int y3) {
        return triangle_area_2 (x1, y1, x2, y2, x3, y3) > 0;
}
```

Проверка двух отрезков на пересечение

Даны два отрезка AB и CD (они могут вырождаться в точки). Требуется проверить, пересекаются они или нет.

Если дополнительно требуется найти саму точку (точки) пересечения, то см. соответствующую статью.

Первый способ: ориентированная площадь треугольника

Воспользуемся Ориентированной площадью треугольника и предикат 'По часовой стрелке'. Действительно, чтобы отрезки AB и CD пересекались, необходимо и достаточно, чтобы точки A и B находились по разные стороны прямой CD, и, аналогично, точки C и D — по разные стороны прямой AB. Проверить это можно, вычисляя ориентированные площади соответствующих треугольников и сравнивая их знаки.

Единственное, на что следует обратить внимание — граничные случаи, когда какие-то точки попадают на саму прямую. При этом возникает единственный особый случай, когда вышеописанные проверки ничего не дадут — случай, когда оба отрезка лежат **на одной прямой**. Этот случай надо рассмотреть отдельно. Для этого достаточно проверить, что проекции этих двух отрезков на оси X и Y пересекаются (часто эту проверку называют "проверкой на bounding box").

В целом, этот способ — более простой, чем тот, что будет приведён ниже (производящий пересечение двух прямых), и имеет меньше особых случаев, однако главный его недостаток — в том, что он не находит саму точку пересечения.

Реализация:

```
struct pt {
        int x, y;
};

inline int area (pt a, pt b, pt c) {
        return (b.x - a.x) * (c.y - a.y) - (b.y - a.y) * (c.x - a.x);
}

inline bool intersect_1 (int a, int b, int c, int d) {
        if (a > b)  swap (a, b);
        if (c > d)  swap (c, d);
        return max(a,c) <= min(b,d);
}

bool intersect (pt a, pt b, pt c, pt d) {
        return intersect_1 (a.x, b.x, c.x, d.x)
                && intersect_1 (a.y, b.y, c.y, d.y)
                && area(a,b,c) * area(a,b,d) <= 0
                && area(c,d,a) * area(c,d,b) <= 0;
}
```

В целях оптимизации проверка на bounding box вынесена в начало, до вычисления площадей — поскольку это более "лёгкая" проверка.

Само собой, этот код применим и для случая вещественных координат, просто все сравнения с нулём следует производить по эпсилону (и избегать перемножения двух вещественнозначных значений $\mathrm{area}()$, перемножая вместо этого их знаки).

Второй способ: пересечение двух прямых

Вместо пересечения отрезков выполним пересечение двух прямых, в результате, если прямые не параллельны, получим какую-то точку, которую надо проверить на принадлежность обоим отрезкам; для этого достаточно проверить, что эта точка принадлежит обоим отрезкам в проекции на ось X и на ось Y.

Если же прямые оказались параллельными, то, если они не совпадают, то отрезки точно не пересекаются. Если же прямые совпали, то отрезки лежат на одной прямой, и для проверки их пересечения достаточно проверить, что пересекаются их проекции на ось X и Y.

Остаётся ещё особый случай, когда один или оба отрезка **вырождаются** в точки: в таком случае говорить о прямых некорректно, и этот метод будет неприменим (этот случай надо будет разбирать отдельно).

Реализация (без учёта случая вырожденных отрезков):

```
struct pt {
        int x, y;
};

const double EPS = 1E-9;

inline int det (int a, int b, int c, int d) {
        return a * d - b * c;
}

inline bool between (int a, int b, double c) {
        return min(a,b) <= c + EPS && c <= max(a,b) + EPS;
}

inline bool intersect_1 (int a, int b, int c, int d) {
        if (a > b)  swap (a, b);
        if (c > d)  swap (c, d);
        return max(a,c) <= min(b,d);
}

bool intersect (pt a, pt b, pt c, pt d) {
        int A1 = a.y-b.y,  B1 = b.x-a.x,  C1 = -A1*a.x - B1*a.y;
        int A2 = c.y-d.y,  B2 = d.x-c.x,  C2 = -A2*c.x - B2*c.y;
        int zn = det (A1, B1, A2, B2);
        if (zn != 0) {
                double x = - det (C1, B1, C2, B2) * 1. / zn;
                double y = - det (A1, C1, A2, C2) * 1. / zn;
                return between (a.x, b.x, x) && between (a.y, b.y, y)
                        && between (c.x, d.x, x) && between (c.y, d.y, y);
        }
        else
                return det (A1, C1, A2, C2) == 0 && det (B1, C1, B2, C2) == 0
                        && intersect_1 (a.x, b.x, c.x, d.x)
                        && intersect_1 (a.y, b.y, c.y, d.y);
}
```

Здесь сначала вычисляется коэффициент zn — знаменатель в формуле Крамера. Если $\mathrm{zn} = 0$, то коэффициенты A и B прямых пропорциональны, и прямые параллельны или совпадают. В этом случае надо проверить, совпадают они или нет, для чего надо проверить, что коэффициенты C прямых пропорциональны с тем же коэффициентом, для чего достаточно вычислить два следующих определителя, если они оба равны нулю, то прямые совпадают:

$$\begin{vmatrix} A_1 & C_1 \\ A_2 & C_2 \end{vmatrix}, \begin{vmatrix} B_1 & C_1 \\ B_2 & C_2 \end{vmatrix}$$

Если же $\mathrm{zn} \neq 0$, то прямые пересекаются, и по формуле Крамера находим точку пересечения (x, y) и проверяем её принадлежность обоим отрезкам.

Следует отметить, что если исходные координаты точек уже были вещественнозначными, то следует нормировать прямые (т.е. привести их к такому состоянию, что сумма квадратов коэффициентов a и b равна единице), иначе погрешности при сравнении прямых на параллельность и на совпадение могут оказаться слишком большими.

Нахождение уравнения прямой для отрезка

Задача — по заданным координатам конца отрезка построить прямую, проходящую через него.

Мы считаем, что отрезок невырожден, т.е. имеет длину больше нуля (иначе, понятно, через него проходит бесконечно много различных прямых).

Двумерный случай

Пусть дан отрезок PQ, т.е. известны координаты его концов P_x, P_y, Q_x, Q_y.

Требуется построить **уравнение прямой на плоскости**, проходящей через этот отрезок, т.е. найти коэффициенты A, B, C в уравнении прямой:

$$Ax + By + C = 0.$$

Заметим, что искомых троек (A, B, C), проходящих через заданный отрезок, **бесконечно много**: можно умножить все три коэффициента на произвольное ненулевое число и получить ту же самую прямую. Следовательно, наша задача — найти одну из таких троек.

Нетрудно убедиться (подстановкой этих выражений и координат точек P и Q в уравнение прямой), что подходит следующий набор коэффициентов:

$$\begin{aligned} A &= P_y - Q_y, \\ B &= Q_x - P_x, \\ C &= -AP_x - BP_y. \end{aligned}$$

Целочисленный случай

Важным преимуществом такого способа построения прямой является то, что если координаты концов были целочисленными, то и полученные коэффициенты также будут **целочисленными**. В некоторых случаях это позволяет производить геометрические операции, вообще не прибегая к вещественным числам.

Однако есть и небольшой недостаток: для одной и той же прямой могут получаться разные тройки коэффициентов. Чтобы избежать этого, но не уходить от целочисленных коэффициентов, можно применить следующий приём, часто называемый **нормированием**. Найдём наибольший общий делитель чисел $|A|, |B|, |C|$, поделим на него все три коэффициента, а затем произведём нормировку знака: если $A < 0$ или $A = 0, B < 0$, то умножим все три коэффициента на -1. В итоге мы придём к тому, что для одинаковых прямых будут получаться одинаковые тройки коэффициентов, что позволит легко проверять прямые на равенство.

Вещественнозначный случай

При работе с вещественными числами следует всегда помнить о погрешностях.

Коэффициенты A и B получаются у нас порядка исходных координат, коэффициент C — уже порядка квадрата от них. Это уже может быть достаточно большими числами, а, например, при пересечении прямых они станут ещё больше, что может привести к большим ошибкам округления уже при исходных координатах порядка 10^3.

Поэтому при работе с вещественными числами желательно производить так называемую **нормировку** прямой: а именно, делать коэффициенты такими, чтобы $A^2 + B^2 = 1$. Для этого надо вычислить число Z:

$$Z = \sqrt{A^2 + B^2},$$

и разделить все три коэффициента A, B, C на него.

Тем самым, порядок коэффициентов A и B уже не будет зависеть от порядка входных координат, а коэффициент C будет того же порядка, что и входные координаты. На практике это приводит к значительному улучшению точности вычислений.

Наконец, упомянем о **сравнении** прямых — ведь после такой нормировки для одной и той же прямой могут получаться только две тройки коэффициентов: с точностью до умножения на -1. Соответственно, если мы произведём дополнительную нормировку с учётом знака (если $A < -\varepsilon$ или $|A| < \varepsilon, B < -\varepsilon$, то умножать на -1), то получающиеся коэффициенты будут уникальными.

Трёхмерный и многомерный случай

Уже в трёхмерном случае **нет простого уравнения**, описывающего прямую (её можно задать как пересечение двух плоскостей, т.е. систему двух уравнений, но это неудобный способ).

Следовательно, в трёхмерном и многомерном случаях мы должны пользоваться **параметрическим способом задания прямой**, т.е. в виде точки p и вектора v:

$$p + vt, \quad t \in \mathcal{R}.$$

Т.е. прямая — это все точки, которые можно получить из точки p прибавлением вектора v с произвольным коэффициентом.

Построение прямой в параметрическом виде по координатам концов отрезка — тривиально, мы просто берём один конец отрезка за точку p, а вектор из первого до второго конца — за вектор v.

Точка пересечения прямых

Пусть нам даны две прямые, заданные своими коэффициентами A_1, B_1, C_1 и A_2, B_2, C_2. Требуется найти их точку пересечения, или выяснить, что прямые параллельны.

Решение

Если две прямые не параллельны, то они пересекаются. Чтобы найти точку пересечения, достаточно составить из двух уравнений прямых систему и решить её:

$$\begin{cases} A_1 x + B_1 y + C_1 = 0, \\ A_2 x + B_2 y + C_2 = 0. \end{cases}$$

Пользуясь формулой Крамера, сразу находим решение системы, которое и будет искомой **точкой пересечения**:

$$x = -\frac{\begin{vmatrix} C_1 & B_1 \\ C_2 & B_2 \end{vmatrix}}{\begin{vmatrix} A_1 & B_1 \\ A_2 & B_2 \end{vmatrix}} = -\frac{C_1 B_2 - C_2 B_1}{A_1 B_2 - A_2 B_1},$$
$$y = -\frac{\begin{vmatrix} A_1 & C_1 \\ A_2 & C_2 \end{vmatrix}}{\begin{vmatrix} A_1 & B_1 \\ A_2 & B_2 \end{vmatrix}} = -\frac{A_1 C_2 - A_2 C_1}{A_1 B_2 - A_2 B_1}.$$

Если знаменатель нулевой, т.е.

$$\begin{vmatrix} A_1 & B_1 \\ A_2 & B_2 \end{vmatrix} = A_1 B_2 - A_2 B_1 = 0$$

то система решений не имеет (прямые **параллельны** и не совпадают) или имеет бесконечно много (прямые **совпадают**). Если необходимо различить эти два случая, надо проверить, что коэффициенты C прямых пропорциональны с тем же коэффициентом пропорциональности, что и коэффициенты A и B, для чего достаточно посчитать два определителя, если они оба равны нулю, то прямые совпадают:

$$\begin{vmatrix} A_1 & C_1 \\ A_2 & C_2 \end{vmatrix}, \begin{vmatrix} B_1 & C_1 \\ B_2 & C_2 \end{vmatrix}$$

Реализация

```
struct pt {
        double x, y;
};

struct line {
        double a, b, c;
};

const double EPS = 1e-9;

double det (double a, double b, double c, double d) {
        return a * d - b * c;
}

bool intersect (line m, line n, pt & res) {
        double zn = det (m.a, m.b, n.a, n.b);
        if (abs (zn) < EPS)
                return false;
        res.x = - det (m.c, m.b, n.c, n.b) / zn;
```

```
        res.y = - det (m.a, m.c, n.a, n.c) / zn;
        return true;
}

bool parallel (line m, line n) {
        return abs (det (m.a, m.b, n.a, n.b)) < EPS;
}

bool equivalent (line m, line n) {
        return abs (det (m.a, m.b, n.a, n.b)) < EPS
                && abs (det (m.a, m.c, n.a, n.c)) < EPS
                && abs (det (m.b, m.c, n.b, n.c)) < EPS;
}
```

Пересечение двух отрезков

Даны два отрезка AB и CD (они могут вырождаться в точки). Требуется найти их пересечение: оно может быть пустым (если отрезки не пересекаются), может быть одной точкой, и может быть целым отрезком (если отрезки накладываются друг на друга).

Алгоритм

Работать с отрезками будем как с прямыми: построим по двум отрезкам уравнения их прямых, проверим, не параллельны ли прямые. Если прямые не параллельны, то всё просто: находим их точку пересечения и проверяем, что она принадлежит обоим отрезкам (для этого достаточно проверить, что точка принадлежит каждому отрезку в проекции на ось X и на ось Y по отдельности). В итоге в этом случае ответом будет либо "пусто", либо единственная найденная точка.

Более сложный случай — если прямые оказались параллельными (сюда же относится случай, когда один или оба отрезка выродились в точки). В этом случае надо проверить, что оба отрезка лежат на одной прямой (или, в случае когда они оба вырождены в точку — что эта точка совпадает). Если это не так, то ответ — "пусто". Если это так, то ответ — это пересечение двух отрезков, лежащих на одной прямой, что реализуется достаточно просто — надо взять максимум из левых концов и минимум из правых концов.

В самом начале алгоритма напишем так называемую "проверку на bounding box" — во-первых, она необходима для случая, когда два отрезка лежат на одной прямой, а во-вторых, она, как легковесная проверка, позволяет алгоритму работать в среднем быстрее на случайных тестах.

Реализация

Приведём здесь полную реализацию, включая все вспомогательные функции по работе с точками и прямыми.

Главной здесь является функция $\mathrm{intersect}$, которая пересекает два переданных ей отрезка, и если они пересекаются хотя бы по одной точке, то возвращает true, а в аргументах left и right возвращает начало и конец отрезка-ответа (в частности, когда ответ — это единственная точка, возвращаемые начало и конец будут совпадать).

```
const double EPS = 1E-9;

struct pt {
        double x, y;

        bool operator< (const pt & p) const {
                return x < p.x-EPS || abs(x-p.x) < EPS && y < p.y - EPS;
        }
};

struct line {
        double a, b, c;

        line() {}
        line (pt p, pt q) {
                a = p.y - q.y;
                b = q.x - p.x;
                c = - a * p.x - b * p.y;
                norm();
        }

        void norm() {
                double z = sqrt (a*a + b*b);
                if (abs(z) > EPS)
                        a /= z,  b /= z,  c /= z;
        }

        double dist (pt p) const {
                return a * p.x + b * p.y + c;
        }
};
```

```
#define det(a,b,c,d)  (a*d-b*c)

inline bool betw (double l, double r, double x) {
        return min(l,r) <= x + EPS && x <= max(l,r) + EPS;
}

inline bool intersect_1d (double a, double b, double c, double d) {
        if (a > b)  swap (a, b);
        if (c > d)  swap (c, d);
        return max (a, c) <= min (b, d) + EPS;
}

bool intersect (pt a, pt b, pt c, pt d, pt & left, pt & right) {
        if (! intersect_1d (a.x, b.x, c.x, d.x) || ! intersect_1d (a.y, b.y,
c.y, d.y))
                return false;
        line m (a, b);
        line n (c, d);
        double zn = det (m.a, m.b, n.a, n.b);
        if (abs (zn) < EPS) {
                if (abs (m.dist (c)) > EPS || abs (n.dist (a)) > EPS)
                        return false;
                if (b < a)  swap (a, b);
                if (d < c)  swap (c, d);
                left = max (a, c);
                right = min (b, d);
                return true;
        }
        else {
                left.x = right.x = - det (m.c, m.b, n.c, n.b) / zn;
                left.y = right.y = - det (m.a, m.c, n.a, n.c) / zn;
                return betw (a.x, b.x, left.x)
                        && betw (a.y, b.y, left.y)
                        && betw (c.x, d.x, left.x)
                        && betw (c.y, d.y, left.y);
        }
}
```

Нахождение площади простого многоугольника

Пусть дан простой многоугольник (т.е. без самопересечений, но не обязательно выпуклый), заданный координатами своих вершин в порядке обхода по или против часовой стрелки. Требуется найти его площадь.

Способ 1

Это легко сделать, если перебрать все рёбра и сложить площади трапеций, ограниченных каждым ребром.
Площадь нужно брать с тем знаком, с каким она получится (именно благодаря знаку вся "лишняя" площадь сократится).
Т.е. формула такова:

```
S += (X2 - X1) * (Y1 + Y2) / 2
```

Код:

```
double sq (const vector<point> & fig)
{
        double res = 0;
        for (unsigned i=0; i<fig.size(); i++)
        {
                point
                        p1 = i ? fig[i-1] : fig.back(),
                        p2 = fig[i];
                res += (p1.x - p2.x) * (p1.y + p2.y);
        }
        return fabs (res) / 2;
}
```

Способ 2

Можно поступить другим образом. Выберем произвольно точку O, переберём все рёбра, прибавляя к ответу ориентированную площадь треугольника, образованного ребром и точкой O (см. Ориентированная площадь треугольника). Опять же, благодаря знаку, вся лишняя площадь сократится, и останется только ответ.

Этот способ хорош тем, что его проще обобщить на более сложные случаи (например, когда некоторые стороны - не прямые, а дуги окружности).

Теорема Пика. Нахождение площади решётчатого многоугольника

Многоугольник без самопересечений называется решётчатым, если все его вершины находятся в точках с целочисленными координатами (в декартовой системе координат).

Теорема Пика

Формула

Пусть дан некоторый решётчатый многоугольник, с ненулевой площадью.

Обозначим его площадь через S; количество точек с целочисленными координатами, лежащих строго внутри многоугольника — через I; количество точек с целочисленными координатами, лежащих на сторонах многоугольника — через B.

Тогда справедливо соотношение, называемое **формулой Пика**:

$$S = I + \frac{B}{2} - 1.$$

В частности, если известны значения I и B для некоторого многоугольника, то его площадь можно посчитать за $O(1)$, даже не зная координат его вершин.

Это соотношение открыл и доказал австрийский математик Георг Александр Пик (Georg Alexander Pick) в 1899 г.

Доказательство

Доказательство производится в несколько этапов: от самых простых фигур до произвольных многоугольников:

- Единичный квадрат. В самом деле, для него $S = 1, I = 0, B = 4$, и формула верна.
- Произвольный невырожденный прямоугольник со сторонами, параллельными осям координат. Для доказательства формулы обозначим через a и b длины сторон прямоугольника. Тогда находим: $S = ab, I = (a-1)(b-1), B = 2(a+b)$. Непосредственной подстановкой убеждаемся, что формула Пика верна.
- Прямоугольный треугольник с катетами, параллельными осям координат. Для доказательства заметим, что любой такой треугольник можно получить отсечением некоторого прямоугольника его диагональю. Обозначив через c число целочисленных точек, лежащих на диагонали, можно показать, что формула Пика выполняется для такого треугольника, независимо от значения c.
- Произвольный треугольник. Заметим, что любой такой треугольник может быть превращён в прямоугольник приклеиванием к его сторонам прямоугольных треугольников с катетами, параллельными осям координат (при этом понадобится не более 3 таких треугольников). Отсюда можно получить корректность формулы Пика для любого треугольника.
- Произвольный многоугольник. Для доказательства триангулируем его, т.е. разобьём на треугольники с вершинами в целочисленных точках. Для одного треугольника формулу Пика мы уже доказали. Дальше, можно доказать, что при добавлении к произвольному многоугольнику любого треугольника формула Пика сохраняет свою корректность. Отсюда по индукции следует, что она верна для любого многоугольника.

Обобщение на высшие размерности

К сожалению, эта столь простая и красивая формула Пика плохо обобщается на высшие размерности.

Наглядно показал это Рив (Reeve), предложив в 1957 г. рассмотреть тетраэдр (называемый теперь **тетраэдром Рива**) со следующими вершинами:

$$\begin{aligned} A &= (0,0,0), \\ B &= (1,0,0), \\ C &= (0,1,0), \\ D &= (1,1,k), \end{aligned}$$

где k — любое натуральное число. Тогда этот тетраэдр $ABCD$ при любых k не содержит внутри ни одной точки с целочисленными координатами, а на его границе — лежат только четыре точки A, B, C, D и никакие другие. Таким образом, объём и площадь поверхности этого тетраэдра могут быть разными, в то время как число точек внутри и на границе — неизменны; следовательно, формула Пика не допускает обобщений даже на трёхмерный случай.

Тем не менее, некоторое подобное обобщение на пространства большей размерности всё же имеется, — это **многочлены Эрхарта** (Ehrhart Polynomial), но они весьма сложны, и зависят не только от числа точек внутри и на границе фигуры.

Задача о покрытии отрезков точками

Дано N отрезков на прямой. Требуется покрыть их наименьшим числом точек, т.е. найти наименьшее множество точек такое, что каждому отрезку принадлежит хотя бы одна точка.

Также рассмотрим усложнённый вариант этой задачи - когда дополнительно указано "запрещённое" множество отрезков, т.е. никакая точка из ответа не должна принадлежать ни одному запрещённому отрезку.

Следует также заметить, что эту задачу можно рассматривать и как задачу в теории расписаний - требуется покрыть заданный набор мероприятий-отрезков наименьшим числом точек.

Ниже будет описан жадный алгоритм, решающий обе задачи за $O(N \log N)$.

Решение первой задачи

Заметим сначала, что можно рассматривать только те решения, в которых каждая из точек находится на правом конце какого-либо отрезка. Действительно, нетрудно понять, что любое решение, если оно не удовлетворяет этому свойству, можно привести к нему, сдвигая его точки вправо настолько, насколько это возможно.

Попытаемся теперь построить решение, удовлетворяющее указанному свойству. Возьмём точки-правые концы отрезков, отсортируем их, и будем двигаться по ним слева направо. Если текущая точка является правым концом уже покрытого отрезка, то мы пропускаем её. Пусть теперь текущая точка является правым концом текущего отрезка, который ещё не был покрыт до этого. Тогда мы должны добавить в ответ текущую точку, и отметить все отрезки, которым принадлежит эта точка, как покрытые. Действительно, если бы мы пропустили текущую точку и не стали бы добавлять её в ответ, то, так как она является правым концом текущего отрезка, то мы уже не смогли бы покрыть текущий отрезок.

Однако при наивной реализации этот метод будет работать за $O(N^2)$. Опишем **эффективную реализацию** этого метода.

Возьмём все точки-концы отрезков (как левые, так и правые) и отсортируем их. При этом для каждой точки сохраним вместе с ней номер отрезка, а также то, каким концом его она является (левым или правым). Кроме того, отсортируем точки таким образом, что, если есть несколько точек с одной координатой, то сначала будут идти левые концы, и только потом - правые. Заведём стек, в котором будут храниться номера отрезков, рассматриваемых в данный момент; изначально стек пуст. Будем двигаться по точкам в отсортированном порядке. Если текущая точка - левый конец, то просто добавляем номер её отрезка в стек. Если же она является правым концом, то проверяем, не был ли покрыт этот отрезок (для этого можно просто завести массив булевых переменных). Если он уже был покрыт, то ничего не делаем и переходим к следующей точке (забегая вперёд, мы утверждаем, что в этом случае в стеке текущего отрезка уже нет). Если же он ещё не был покрыт, то мы добавляем текущую точку в ответ, и теперь мы хотим отметить для всех текущих отрезков, что они становятся покрытыми. Поскольку в стеке как раз хранятся номера непокрытых ещё отрезков, то будем доставать из стека по одному отрезку и отмечать, что он уже покрыт, пока стек полностью не опустеет. По окончании работы алгоритма все отрезки будут покрыты, и притом наименьшим числом точек (повторимся, здесь важно требование, что при равенстве координат сначала идут левые концы, и только затем правые).

Таким образом, весь алгоритм выполняется за $O(N)$, не считая сортировки точек, а итоговая сложность алгоритма как раз равна $O(N \log N)$.

Решение второй задачи

Здесь уже появляются запрещённые отрезки, поэтому, во-первых, решения вообще может не существовать, а во-вторых, уже нельзя утверждать, что ответ можно составить только из правых концов отрезков. Однако описанный выше алгоритм можно соответствующим образом модифицировать.

Снова возьмём все точки-концы отрезков (как целевых отрезков, так и запрещённых), отсортируем их, сохранив вместе с каждой точкой её тип и отрезок, концом которого она является. Опять же, отсортируем отрезки так, чтобы при равенстве координат левые концы шли перед правыми, а если и типы концов равны, то левые концы запрещённых должны идти перед левыми концами целевых, а правые концы запрещённых - после целевых (чтобы запрещённые отрезки учитывались как можно дольше при равенстве координат). Заведём счётчик запрещённых отрезков, который будет равен числу запрещённых отрезков, покрывающих текущую точку. Заведём очередь (queue), в которой будут храниться номера текущих целевых отрезков. Будем перебирать точки в отсортированном порядке. Если текущая точка - левый конец целевого отрезка, то просто добавим номер её отрезка в очередь. Если текущая точка - правый конец целевого отрезка, то, если счётчик запрещённых отрезков равен нулю, то мы поступаем аналогично предыдущей задаче - ставим точку в текущую точку, и выталкиваем все отрезки из очереди, отмечая, что они покрыты. Если же счётчик запрещённых отрезков больше нуля, то в текущую точку мы стрелять не можем, а потому мы должны найти самую последнюю точку, свободную от запрещённых отрезков; для этого надо поддерживать соответствующий указатель last_free, который будет обновляться при поступлении запрещённых отрезков. Тогда мы стреляем в last_free-EPS (потому что прямо в неё нельзя стрелять - эта точка принадлежит запрещённому отрезку), и выталкивать отрезки из очереди, пока точка last_free-EPS принадлежит им. А именно, если текущая точка - левый конец запрещённого отрезка, то мы увеличиваем счётчик, и если перед этим

счётчик был равен нулю, то присваиваем last_free текущую координату. Если текущая точка - правый конец запрещённого отрезка, то просто уменьшаем счётчик.

Центры тяжести многоугольников и многогранников

Центром тяжести (или **центром масс**) некоторого тела называется точка, обладающая тем свойством, что если подвесить тело за эту точку, то оно будет сохранять свое положение.

Ниже рассмотрены двумерные и трёхмерные задачи, связанные с поиском различных центров масс — в основном с точки зрения вычислительной геометрии.

В рассмотренных ниже решениях можно выделить два основных **факта**. Первый — что центр масс системы материальных точек равен среднему их координат, взятых с коэффициентами, пропорциональными их массам. Второй факт — что если мы знаем центры масс двух непересекающихся фигур, то центр масс их объединения будет лежать на отрезке, соединяющем эти два центра, причём он будет делить его в то же отношении, как масса второй фигуры относится к массе первой.

Двумерный случай: многоугольники

На самом деле, говоря о центре масс двумерной фигуры, можно иметь в виду одну из трёх следующих **задач**:

- Центр масс системы точек — т.е. вся масса сосредоточена только в вершинах многоугольника.
- Центр масс каркаса — т.е. масса многоугольника сосредоточена на его периметре.
- Центр масс сплошной фигуры — т.е. масса многоугольника распределена по всей его площади.

Каждая из этих задач имеет самостоятельное решение, и будет рассмотрена ниже отдельно.

Центр масс системы точек

Это самая простая из трёх задач, и её решение — известная физическая формула центра масс системы материальных точек:

$$\vec{r_c} = \frac{\sum\limits_i \vec{r_i}\ m_i}{\sum\limits_i m_i},$$

где m_i — массы точек, $\vec{r_i}$ — их радиус-векторы (задающие их положение относительно начала координат), и $\vec{r_c}$ — искомый радиус-вектор центра масс.

В частности, если все точки имеют одинаковую массу, то координаты центра масс есть **среднее арифметическое** координат точек. Для **треугольника** эта точка называется **центроидом** и совпадает с точкой пересечения медиан:

$$\vec{r_c} = \frac{\vec{r_1} + \vec{r_2} + \vec{r_3}}{3}.$$

Для **доказательства** этих формул достаточно вспомнить, что равновесие достигается в такой точке r_c, в которой сумма моментов всех сил равна нулю. В данном случае это превращается в условие того, чтобы сумма радиус-векторов всех точек относительно точки r_c, домноженных на массы соответствующих точек, равнялась нулю:

$$\sum_i (\vec{r_i} - \vec{r_c})\, m_i = \vec{0},$$

и, выражая отсюда $\vec{r_c}$, мы и получаем требуемую формулу.

Центр масс каркаса

Будем считать для простоты, что каркас однороден, т.е. его плотность везде одна и та же.

Но тогда каждую сторону многоугольника можно заменить одной точкой — серединой этого отрезка (т.к. центр масс однородного отрезка есть середина этого отрезка), с массой, равной длине этого отрезка.

Теперь мы получили задачу о системе материальных точек, и применяя к ней решение из предыдущего пункта, мы находим:

$$\vec{r_c} = \frac{\sum\limits_i \vec{r_i^l}\, l_i}{P},$$

где $\vec{r_i^l}$ — точка-середина i-ой стороны многоугольника, l_i — длина i-ой стороны, P — периметр, т.е. сумма длин сторон.

Для **треугольника** можно показать следующее утверждение: эта точка является **точкой пересечения биссектрис** треугольника, образованного серединами сторон исходного треугольника. (чтобы показать это, надо воспользоваться приведённой выше формулой, и затем заметить, что биссектрисы делят стороны получившегося треугольника в тех же соотношениях, что и центры масс этих сторон).

Центр масс сплошной фигуры

Мы считаем, что масса распределена по фигуре однородно, т.е. плотность в каждой точке фигуры равна одному и тому же числу.

Случай треугольника

Утверждается, что для треугольника ответом будет всё тот же **центроид**, т.е. точка, образованная средним арифметическим координат вершин:

$$\vec{r_c} = \frac{\vec{r_1} + \vec{r_2} + \vec{r_3}}{3}.$$

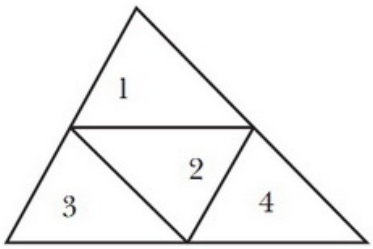
Случай треугольника: доказательство

Приведём здесь элементарное доказательство, не использующее теорию интегралов.

Первым подобное, чисто геометрическое, доказательство привёл Архимед, но оно было весьма сложным, с большим числом геометрических построений. Приведённое здесь доказательство взято из статьи Apostol, Mnatsakanian "Finding Centroids the Easy Way".

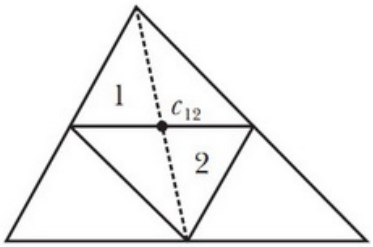
Доказательство сводится к тому, чтобы показать, что центр масс треугольника лежит на одной из медиан; повторяя этот процесс ещё дважды, мы тем самым покажем, что центр масс лежит в точке пересечения медиан, которая и есть центроид.

Разобьём данный треугольник T на четыре, соединив середины сторон, как показано на рисунке:

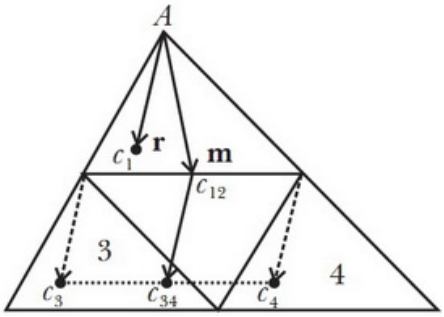


Четыре получившихся треугольника подобны треугольнику T с коэффициентом $1/2$.

Треугольники №1 и №2 вместе образуют параллелограмм, центр масс которого c_{12} лежит в точке пересечения его диагоналей (поскольку это фигура, симметричная относительно обеих диагоналей, а, значит, её центр масс обязан лежать на каждой из двух диагоналей). Точка c_{12} находится посередине общей стороны треугольников №1 и №2, а также лежит на медиане треугольника T:



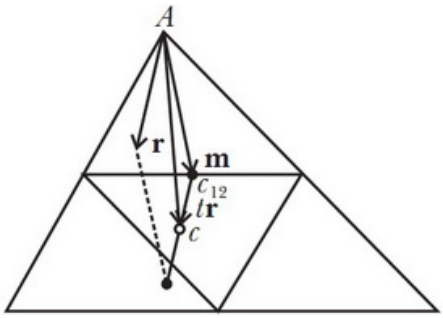
Пусть теперь вектор $\vec{r}$ — вектор, проведённый из вершины A к центру масс c_1 треугольника №1, и пусть вектор $\vec{m}$ — вектор, проведённый из A к точке c_{12} (которая, напомним, является серединой стороны, на которой она лежит):



Наша цель — показать, что вектора $\vec{r}$ и $\vec{m}$ коллинеарны.

Обозначим через c_3 и c_4 точки, являющиеся центрами масс треугольников №3 и №4. Тогда, очевидно, центром масс совокупности этих двух треугольников будет точка c_{34}, являющаяся серединой отрезка $c_3 c_4$. Более того, вектор от точки c_{12} к точке c_{34} совпадает с вектором $\vec{r}$.

Искомый центр масс c треугольника T лежит посередине отрезка, соединяющего точки c_{12} и c_{34} (поскольку мы разбили треугольник T на две части равных площадей: №1-№2 и №3-№4):



Таким образом, вектор от вершины A к центроиду c равен $\vec{m} + \vec{r}/2$. С другой стороны, т.к. треугольник №1 подобен треугольнику T с коэффициентом $1/2$, то этот же вектор равен $2\vec{r}$. Отсюда получаем уравнение:

$$\vec{m} + \vec{r}/2 = 2\vec{r},$$

откуда находим:

$$\vec{r} = \frac{2}{3}\vec{m}.$$

Таким образом, мы доказали, что вектора $\vec{r}$ и $\vec{m}$ коллинеарны, что и означает, что искомый центроид c лежит на медиане, исходящей из вершины A.

Более того, попутно мы доказали, что центроид делит каждую медиану в отношении $2:1$, считая от вершины.

Случай многоугольника

Перейдём теперь к общему случаю — т.е. к случаю **мноугоугольника**. Для него такие рассуждения уже неприменимы, поэтому сведём задачу к треугольной: а именно, разобьём многоугольник на треугольники (т. е. триангулируем его), найдём центр масс каждого треугольника, а затем найдём центр масс получившихся центров масс треугольников.

Окончательная формула получается следующей:

$$\vec{r_c} = \frac{\sum\limits_i \vec{r_i^\circ}\ S_i}{S},$$

где $\vec{r_i^\circ}$ — центроид i-го треугольника в триангуляции заданного многоугольника, S_i — площадь i-го треугольника триангуляции, S — площадь всего многоугольника.

Триангуляция выпуклого многоугольника — тривиальная задача: для этого, например, можно взять треугольники (r_1, r_{i-1}, r_i), где $i = 3 \ldots n$.

Случай многоугольника: альтернативный способ

С другой стороны, применение приведённой формулы не очень удобно для **невыпуклых многоугольников**, поскольку произвести их триангуляцию — сама по себе непростая задача. Но для таких многоугольников можно придумать более простой подход. А именно, проведём аналогию с тем, как можно искать площадь произвольного многоугольника: выбирается произвольная точка z, а затем суммируются знаковые площади треугольников, образованных этой точкой и точками многоугольника:

$S = |\sum_{i=1}^{n} S_{z,p_i,p_{i+1}}|$. Аналогичный приём можно применить и для поиска центра масс: только теперь мы будем суммировать центры масс треугольников (z, p_i, p_{i+1}), взятых с коэффициентами, пропорциональными их площадям, т.е. итоговая формула для центра масс такова:

$$\vec{r_c} = \frac{\sum\limits_i \vec{r}^{\circ}_{z,p_i,p_{i+1}} \; S_{z,p_i,p_{i+1}}}{S},$$

где z — произвольная точка, p_i — точки многоугольника, $\vec{r}^{\circ}_{z,p_i,p_{i+1}}$ — центроид треугольника (z, p_i, p_{i+1}), $S_{z,p_i,p_{i+1}}$ — знаковая площадь этого треугольника, S — знаковая площадь всего многоугольника (т. е. $S = \sum_{i=1}^{n} S_{z,p_i,p_{i+1}}$).

Трёхмерный случай: многогранники

Аналогично двумерному случаю, в 3D можно говорить сразу о четырёх возможных постановках задачи:

- Центр масс системы точек — вершин многогранника.
- Центр масс каркаса — рёбер многогранника.
- Центр масс поверхности — т.е. масса распределена по площади поверхности многогранника.
- Центр масс сплошного многогранника — т.е. масса распределена по всему многограннику.

Центр масс системы точек

Как и в двумерном случае, мы можем применить физическую формулу и получить тот же самый результат:

$$\vec{r_c} = \frac{\sum\limits_i \vec{r_i} \; m_i}{\sum\limits_i m_i},$$

который в случае равных масс превращается в среднее арифметическое координат всех точек.

Центр масс каркаса многогранника

Аналогично двумерному случаю, мы просто заменяем каждое ребро многогранника материальной точкой, расположенной посередине этого ребра, и с массой, равной длине этого ребра. Получив задачу о материальных точках, мы легко находим её решение как взвешенную сумму координат этих точек.

Центр масс поверхности многогранника

Каждая грань поверхности многогранника — двухмерная фигура, центр масс которой мы умеем искать. Найдя эти центры масс и заменив каждую грань её центром масс, мы получим задачу с материальными точками, которую уже легко решить.

Центр масс сплошного многогранника

Случай тетраэдра

Как и в двумерном случае, решим сначала простейшую задачу — задачу для тетраэдра.

Утверждается, что центр масс тетраэдра совпадает с точкой пересечения его медиан (медианой тетраэдра называется отрезок, проведённый из его вершины в центр масс противоположной грани; таким образом, медиана тетраэдра проходит через вершину и через точку пересечения медиан треугольной грани).

Почему это так? Здесь верны рассуждения, аналогичные двумерному случаю: если мы рассечём тетраэдр на два тетраэдра с помощью плоскости, проходящей через вершину тетраэдра и какую-нибудь медиану противоположной грани, то оба получившихся тетраэдра будут иметь одинаковый объём (т.к. треугольная грань разобьётся медианой на два треугольника равной площади, а высота двух тетраэдров не изменится). Повторяя эти рассуждения несколько раз, получаем, что центр масс лежит на точке пересечения медиан тетраэдра.

Эта точка — точка пересечения медиан тетраэдра — называется его **центроидом**. Можно показать, что она на самом деле имеет координаты, равные среднему арифметическому координат вершин тетраэдра:

$$\vec{r_c} = \frac{\vec{r_1} + \vec{r_2} + \vec{r_3} + \vec{r_4}}{4}.$$

(это можно вывести из того факта, что центроид делит медианы в отношении $1 : 3$)

Таким образом, между случаями тетраэдра и треугольника принципиальной разницы нет: точка, равная среднему арифметическому вершин, является центром масс сразу в двух постановках задачи: и когда массы находится только в вершинах, и когда массы распределены по всей площади/объёму. На самом деле, этот результат обобщается на произвольную размерность: центр масс произвольного **симплекса** (simplex) есть среднее арифметическое координат его вершин.

Случай произвольного многогранника

Перейдём теперь к общему случаю — случаю произвольного многогранника.

Снова, как и в двумерном случае, мы производим сведение этой задачи к уже решённой: разбиваем многогранник на тетраэдры (т.е. производим его тетраэдризацию), находим центр масс каждого из них, и получаем окончательный ответ на задачу в виде взвешенной суммы найденных центров масс.

Пересечение окружности и прямой

Дана окружность (координатами своего центра и радиусом) и прямая (своим уравнением). Требуется найти точки их пересечения (одна, две, либо ни одной).

Решение

Вместо формального решения системы двух уравнений подойдём к задаче **с геометрической стороны** (причём, за счёт этого мы получим более точное решение с точки зрения численной устойчивости).

Предположим, не теряя общности, что центр окружности находится в начале координат (если это не так, то перенесём его туда, исправив соответствующе константу C в уравнении прямой). Т.е. имеем окружность с центром в (0,0) радиуса r и прямую с уравнением Ax + By + C = 0.

Сначала найдём **ближайшую к центру точку** прямой - точку с некоторыми координатами (x_0,y_0). Во-первых, эта точка должна находиться на таком расстоянии от начала координат:

```
    |C|
----------
sqrt(A²+B²)
```

Во-вторых, поскольку вектор (A,B) перпендикулярен прямой, то координаты этой точки должны быть пропорциональны координатам этого вектора. Учитывая, что расстояние от начала координат до искомой точки нам известно, нам нужно просто нормировать вектор (A,B) к этой длине, и мы получаем:

```
         A C
x₀ = - -----
       A²+B²

         B C
y₀ = - -----
       A²+B²
```

(здесь неочевидны только знаки 'минус', но эти формулы легко проверить подстановкой в уравнение прямой - должен получиться ноль)

Зная ближайшую к центру окружности точку, мы уже можем определить, сколько точек будет содержать ответ, и даже дать ответ, если этих точек 0 или 1.

Действительно, если расстояние от (x_0, y_0) до начала координат (а его мы уже выразили формулой - см. выше) больше радиуса, то **ответ** - **ноль точек**. Если это расстояние равно радиусу, то **ответом будет одна точка** - (x_0,y_0). А вот в оставшемся случае точек будет две, и их координаты нам предстоит найти.

Итак, мы знаем, что точка (x_0, y_0) лежит внутри круга. Искомые точки (ax,ay) и (bx,by), помимо того что должны принадлежать прямой, должны лежать на одном и том же расстоянии d от точки (x_0, y_0), причём это расстояние легко найти:

```
                  C²
d = sqrt ( r² - ----- )
                A²+B²
```

Заметим, что вектор (-B,A) коллинеарен прямой, а потому искомые точки (ax,ay) и (bx,by) можно получить, прибавив к точке (x_0,y_0) вектор (-B,A), нормированный к длине d (мы получим одну искомую точку), и вычтя этот же вектор (получим вторую искомую точку).

Окончательное решение такое:

```
                d²
mult = sqrt ( ----- )
              A²+B²

ax = x₀ + B mult
```

```
ay = y0 - A mult
bx = x0 - B mult
by = y0 + A mult
```

Если бы мы решали эту задачу чисто алгебраически, то скорее всего получили бы решение в другом виде, которое даёт бОльшую погрешность. Поэтому "геометрический" метод, описанный здесь, помимо наглядности, ещё и более точен.

Реализация

Как и было указано в начале описания, предполагается, что окружность расположена в начале координат.

Поэтому входные параметры - это радиус окружности и коэффициенты A,B,C уравнения прямой.

```
double r, a, b, c; // входные данные

double x0 = -a*c/(a*a+b*b),  y0 = -b*c/(a*a+b*b);
if (c*c > r*r*(a*a+b*b)+EPS)
        puts ("no points");
else if (abs (c*c - r*r*(a*a+b*b)) < EPS) {
        puts ("1 point");
        cout << x0 << ' ' << y0 << '\n';
}
else {
        double d = r*r - c*c/(a*a+b*b);
        double mult = sqrt (d / (a*a+b*b));
        double ax,ay,bx,by;
        ax = x0 + b * mult;
        bx = x0 - b * mult;
        ay = y0 - a * mult;
        by = y0 + a * mult;
        puts ("2 points");
        cout << ax << ' ' << ay << '\n' << bx << ' ' << by << '\n';
}
```

Пересечение двух окружностей

Даны две окружности, каждая определена координатами своего центра и радиусом. Требуется найти все их точки пересечения (либо одна, либо две, либо ни одной точки, либо окружности совпадают).

Решение

Сведём нашу задачу к задаче о **Пересечении окружности и прямой**.

Предположим, не теряя общности, что центр первой окружности - в начале координат (если это не так, то перенесём центр в начало координат, а при выводе ответа будем обратно прибавлять координаты центра). Тогда мы имеем систему двух уравнений:

$$x^2 + y^2 = r_1^2$$
$$(x - x_2)^2 + (y - y_2)^2 = r_2^2$$

Вычтем из второго уравнения первое, чтобы избавиться от квадратов переменных:

$$x^2 + y^2 = r_1^2$$
$$x\,(-2x_2) + y\,(-2y_2) + (x_2^2 + y_2^2 + r_1^2 - r_2^2) = 0$$

Таким образом, мы свели задачу о пересечении двух окружностей к задаче о пересечении первой окружности и следующей прямой:

$$Ax + By + C = 0,$$
$$A = -2x_2,$$
$$B = -2y_2,$$
$$C = x_2^2 + y_2^2 + r_1^2 - r_2^2.$$

А решение последней задачи описано в соответствующей статье.

Единственный **вырожденный случай**, который надо рассмотреть отдельно - когда центры окружностей совпадают. Действительно, в этом случае вместо уравнения прямой мы получим уравнение вида 0 = C, где C - некоторое число, и этот случай будет обрабатываться некорректно. Поэтому этот случай нужно рассмотреть отдельно: если радиусы окружностей совпадают, то ответ - бесконечность, иначе - точек пересечения нет.

Построение выпуклой оболочки обходом Грэхэма

Даны N точек на плоскости. Построить их выпуклую оболочку, т.е. наименьший выпуклый многоугольник, содержащий все эти точки.

Мы рассмотрим метод **Грэхэма** (Graham) (предложен в 1972 г.) с улучшениями Эндрю (Andrew) (1979 г.). С его помощью можно построить выпуклую оболочку за время $O(N \log N)$ с использованием только операций сравнения, сложения и умножения. Алгоритм является асимптотически оптимальным (доказано, что не существует алгоритма с лучшей асимптотикой), хотя в некоторых задачах он неприемлем (в случае параллельной обработки или при online-обработке).

Описание

Алгоритм. Найдём самую левую и самую правую точки A и B (если таких точек несколько, то возьмём самую нижнюю среди левых, и самую верхнюю среди правых). Понятно, что и A, и B обязательно попадут в выпуклую оболочку. Далее, проведём через них прямую AB, разделив множество всех точек на верхнее и нижнее подмножества S1 и S2 (точки, лежащие на прямой, можно отнести к любому множеству - они всё равно не войдут в оболочку). Точки A и B отнесём к обоим множествам. Теперь построим для S1 верхнюю оболочку, а для S2 - нижнюю оболочку, и объединим их, получив ответ. Чтобы получить, скажем, верхнюю оболочку, нужно отсортировать все точки по абсциссе, затем пройтись по всем точкам, рассматривая на каждом шаге кроме самой точки две предыдущие точки, вошедшие в оболочку. Если текущая тройка точек образует не правый поворот (что легко проверить с помощью Ориентированной площади), то ближайшего соседа нужно удалить из оболочки. В конце концов, останутся только точки, входящие в выпуклую оболочку.

Итак, алгоритм заключается в сортировке всех точек по абсциссе и двух (в худшем случае) обходах всех точек, т. е. требуемая асимптотика O (N log N) достигнута.

Реализация

```
struct pt {
        double x, y;
};

bool cmp (pt a, pt b) {
        return a.x < b.x || a.x == b.x && a.y < b.y;
}

bool cw (pt a, pt b, pt c) {
        return a.x*(b.y-c.y)+b.x*(c.y-a.y)+c.x*(a.y-b.y) < 0;
}

bool ccw (pt a, pt b, pt c) {
        return a.x*(b.y-c.y)+b.x*(c.y-a.y)+c.x*(a.y-b.y) > 0;
}

void convex_hull (vector<pt> & a) {
        if (a.size() == 1)  return;
        sort (a.begin(), a.end(), &cmp);
        pt p1 = a[0],  p2 = a.back();
        vector<pt> up, down;
        up.push_back (p1);
        down.push_back (p1);
        for (size_t i=1; i<a.size(); ++i) {
                if (i==a.size()-1 || cw (p1, a[i], p2)) {
                        while (up.size()>=2 && !cw (up[up.size()-2], up[up.
size()-1], a[i]))
                                up.pop_back();
                        up.push_back (a[i]);
                }
                if (i==a.size()-1 || ccw (p1, a[i], p2)) {
                        while (down.size()>=2 && !ccw (down[down.size()-2],
down[down.size()-1], a[i]))
                                down.pop_back();
                        down.push_back (a[i]);
                }
        }
        a.clear();
```

```
        for (size_t i=0; i<up.size(); ++i)
                a.push_back (up[i]);
        for (size_t i=down.size()-2; i>0; --i)
                a.push_back (down[i]);
}
```

Нахождение площади объединения треугольников. Метод вертикальной декомпозиции

Даны N треугольников. Требуется найти площадь их объединения.

Решение

Здесь мы рассмотрим метод **вертикальной декомпозиции**, который в задачах на геометрию часто оказывается очень важным.

Итак, у нас имеется N треугольников, которые могут как угодно пересекаться друг с другом. Избавимся от этих пересечений с помощью вертикальной декомпозиции: найдём все точки пересечения всех отрезков (образующих треугольники), и отсортируем найденные точки по их абсциссе. Пусть мы получили некоторый массив B. Будем двигаться по этому массиву. На i-ом шаге рассматриваем элементы B[i] и B[i+1]. Мы имеем вертикальную полосу между прямыми X = B[i] и X = B[i+1], причём, согласно самому построению массива B, внутри этой полосы отрезки никак не пересекаются друг с другом. Следовательно, внутри этой полосы треугольники обрезаются до трапеций, причём стороны этих трапеций внутри полосы не пересекаются вообще. Будем двигаться по сторонам этих трапеций снизу вверх, и складывать площади трапеций, следя за тем, чтобы каждый кусок был учитан ровно один раз. Фактически, этот процесс очень напоминает обработку вложенных скобок. Сложив площади трапеций внутри каждой полосы, и сложив результаты для всех полос, мы и найдём ответ - площадь объединения треугольников.

Рассмотрим ещё раз процесс сложения площадей трапеций, уже с точки зрения реализации. Мы перебираем все стороны всех треугольников, и если какая-то сторона (не вертикальная, нам вертикальные стороны не нужны, и даже наоборот, будут сильно мешать) попадает в эту вертикальную полосу (полностью или частично), то мы кладём эту сторону в некоторый вектор, удобнее всего это делать в таком виде: координаты Y в точках пересечения стороны с границами вертикальной полосы, и номер треугольника. После того, как мы построили этот вектор, содержащий куски сторон, сортируем его по значению Y: сначала по левой Y, потом по правой Y. В результате первый в векторе элемент будет содержать нижнюю сторону самой нижней трапеции. Теперь мы просто идём по полученному вектору. Пусть i - текущий элемент; это означает, что i-ый кусок - это нижняя сторона некоторой трапеции, некоторого блока (который может содержать несколько трапеций), площадь которого мы хотим сразу прибавить к ответу. Поэтому мы устанавливаем некий счётчик треугольников равным 1, и поднимаемся по отрезкам вверх, и увеличиваем счётчик, если мы встречаем сторону какого-то треугольника в первый раз, и уменьшаем счётчик, если мы встречаем треугольник во второй раз. Если на каком-то отрезке j счётчик стал равным нулю, то мы нашли верхнюю границу блока - на этом мы останавливаемся, прибавляем площадь трапеции, ограниченной отрезками i и j, и i присваиваем j+1, и повторяем весь процесс заново.

Итак, благодаря методу вертикальной декомпозиции мы решили эту задачу, из геометрических примитивов использовав только пересечение двух отрезков.

Реализация

```
struct segment {
        int x1, y1, x2, y2;
};

struct point {
        double x, y;
};

struct item {
        double y1, y2;
        int triangle_id;
};

struct line {
        int a, b, c;
};

const double EPS = 1E-7;

void intersect (segment s1, segment s2, vector<point> & res) {
        line l1 = { s1.y1-s1.y2, s1.x2-s1.x1, l1.a*s1.x1+l1.b*s1.y1 },
                l2 = { s2.y1-s2.y2, s2.x2-s2.x1, l2.a*s2.x1+l2.b*s2.y1 };
        double det1 = l1.a * l2.b - l1.b * l2.a;
        if (abs (det1) < EPS)  return;
```

```
        point p = { (l1.c * 1.0 * l2.b - l1.b * 1.0 * l2.c) / det1,
                (l1.a * 1.0 * l2.c - l1.c * 1.0 * l2.a) / det1 };
        if (p.x >= s1.x1-EPS && p.x <= s1.x2+EPS && p.x >= s2.x1-EPS && p.x
<= s2.x2+EPS)
                res.push_back (p);
}

double segment_y (segment s, double x) {
        return s.y1 + (s.y2 - s.y1) * (x - s.x1) / (s.x2 - s.x1);
}

bool eq (double a, double b) {
        return abs (a-b) < EPS;
}

vector<item> c;

bool cmp_y1_y2 (int i, int j) {
        const item & a = c[i];
        const item & b = c[j];
        return a.y1 < b.y1-EPS || abs (a.y1-b.y1) < EPS && a.y2 < b.y2-EPS;
}

int main() {

        int n;
        cin >> n;
        vector<segment> a (n*3);
        for (int i=0; i<n; ++i) {
                int x1, y1, x2, y2, x3, y3;
                scanf ("%d%d%d%d%d%d", &x1,&y1,&x2,&y2,&x3,&y3);
                segment s1 = { x1,y1,x2,y2 };
                segment s2 = { x1,y1,x3,y3 };
                segment s3 = { x2,y2,x3,y3 };
                a[i*3] = s1;
                a[i*3+1] = s2;
                a[i*3+2] = s3;
        }

        for (size_t i=0; i<a.size(); ++i)
                if (a[i].x1 > a[i].x2)
                        swap (a[i].x1, a[i].x2),  swap (a[i].y1, a[i].y2);

        vector<point> b;
        b.reserve (n*n*3);
        for (size_t i=0; i<a.size(); ++i)
                for (size_t j=i+1; j<a.size(); ++j)
                        intersect (a[i], a[j], b);

        vector<double> xs (b.size());
        for (size_t i=0; i<b.size(); ++i)
                xs[i] = b[i].x;
        sort (xs.begin(), xs.end());
        xs.erase (unique (xs.begin(), xs.end(), &eq), xs.end());

        double res = 0;
        vector<char> used (n);
        vector<int> cc (n*3);
        c.resize (n*3);
        for (size_t i=0; i+1<xs.size(); ++i) {
                double x1 = xs[i],  x2 = xs[i+1];
                size_t csz = 0;
                for (size_t j=0; j<a.size(); ++j)
                        if (a[j].x1 != a[j].x2)
                                if (a[j].x1 <= x1+EPS && a[j].x2 >= x2-EPS) {
                                        item it = { segment_y (a[j],
x1), segment_y (a[j], x2), (int)j/3 };
                                        cc[csz] = (int)csz;
                                        c[csz++] = it;
                                }
                sort (cc.begin(), cc.begin()+csz, &cmp_y1_y2);
```

```
                double add_res = 0;
                for (size_t j=0; j<csz; ) {
                        item lower = c[cc[j++]];
                        used[lower.triangle_id] = true;
                        int cnt = 1;
                        while (cnt && j<csz) {
                                char & cur = used[c[cc[j++]].triangle_id];
                                cur = !cur;
                                if (cur)  ++cnt;  else  --cnt;
                        }
                        item upper = c[cc[j-1]];
                        add_res += upper.y1 - lower.y1 + upper.y2 - lower.y2;
                }
                res += add_res * (x2 - x1) / 2;
        }

        cout.precision (8);
        cout << fixed << res;

}
```

Проверка точки на принадлежность выпуклому многоугольнику

Дан выпуклый многоугольник с N вершинами, координаты всех вершин целочисленны (хотя это не меняет суть решения); вершины заданы в порядке обхода против часовой стрелки (в противном случае нужно просто отсортировать их). Поступают запросы - точки, и требуется для каждой точки определить, лежит она внутри этого многоугольника или нет (границы многоугольника включаются). На каждый запрос будем отвечать в режиме on-line за O (log N). Предварительная обработка многоугольника будет выполняться за O (N).

Алгоритм

Решать будем **бинарным поиском по углу**.

Один из вариантов решения таков. Выберем точку с наименьшей координатой X (если таких несколько, то выбираем самую нижнюю, т.е. с наименьшим Y). Относительно этой точки, обозначим её Zero, все остальные вершины многоугольника лежат в правой полуплоскости. Далее, заметим, что все вершины многоугольника уже упорядочены по углу относительно точки Zero (это вытекает из того, что многоугольник выпуклый, и уже упорядочен против часовой стрелки), причём все углы находятся в промежутке (-π/2 ; π/2].

Пусть поступает очередной запрос - некоторая точка P. Рассмотрим её полярный угол относительно точки Zero. Найдём бинарным поиском две такие соседние вершины L и R многоугольника, что полярный угол P лежит между полярными углами L и R. Тем самым мы нашли тот сектор многоугольника, в котором лежит точка P, и нам остаётся только проверить, лежит ли точка P в треугольнике (Zero,L,R). Это можно сделать, например, с помощью Ориентированной площади треугольника и Предиката "По часовой стрелке", достаточно посмотреть, по часовой стрелке или против находится тройка вершин (R,L,P).

Таким образом, мы за O (log N) находим сектор многоугольника, а затем за O (1) проверяем принадлежность точки треугольнику, и, следовательно, требуемая асимптотика достигнута. Предварительная обработка многоугольника заключается только в том, чтобы предпосчитать полярные углы для всех точек, хотя, эти вычисления тоже можно перенести на этап бинарного поиска.

Замечания по реализации

Чтобы определять полярный угол, можно воспользоваться стандартной функцией atan2. Тем самым мы получим очень короткое и простое решение, однако взамен могут возникнуть проблемы с точностью.

Учитывая, что изначально все координаты являются целочисленными, можно получить решение, вообще не использующее дробной арифметики.

Заметим, что полярный угол точки (X,Y) относительно начала координат однозначно определяется дробью Y/X, при условии, что точка находится в правой полуплоскости. Более того, если у одной точки полярный угол меньше, чем у другой, то и дробь Y1/X1 будет меньше Y2/X2, и обратно.

Таким образом, для сравнения полярных углов двух точек нам достаточно сравнить дроби Y1/X1 и Y2/X2, что уже можно выполнить в целочисленной арифметике.

Реализация

Эта реализация предполагает, что в данном многоугольнике нет повторяющихся вершин, и площадь многоугольника ненулевая.

```
struct pt {
        int x, y;
};

struct ang {
        int a, b;
};

bool operator < (const ang & p, const ang & q) {
        if (p.b == 0 && q.b == 0)
                return p.a < q.a;
        return p.a * 1ll * q.b < p.b * 1ll * q.a;
}

long long sq (pt & a, pt & b, pt & c) {
        return a.x*1ll*(b.y-c.y) + b.x*1ll*(c.y-a.y) + c.x*1ll*(a.y-b.y);
}
```

```
int main() {

        int n;
        cin >> n;
        vector<pt> p (n);
        int zero_id = 0;
        for (int i=0; i<n; ++i) {
                scanf ("%d%d", &p[i].x, &p[i].y);
                if (p[i].x < p[zero_id].x || p[i].x == p[zero_id].x && p[i].y
< p[zero_id].y)
                        zero_id = i;
        }
        pt zero = p[zero_id];
        rotate (p.begin(), p.begin()+zero_id, p.end());
        p.erase (p.begin());
        --n;

        vector<ang> a (n);
        for (int i=0; i<n; ++i) {
                a[i].a = p[i].y - zero.y;
                a[i].b = p[i].x - zero.x;
                if (a[i].a == 0)
                        a[i].b = a[i].b < 0 ? -1 : 1;
        }

        for (;;) {
                pt q; // очередной запрос
                bool in = false;
                if (q.x >= zero.x)
                        if (q.x == zero.x && q.y == zero.y)
                                in = true;
                        else {
                                ang my = { q.y-zero.y, q.x-zero.x };
                                if (my.a == 0)
                                        my.b = my.b < 0 ? -1 : 1;
                                vector<ang>::iterator it = upper_bound (a.
begin(), a.end(), my);
                                if (it == a.end() && my.a == a[n-1].a && my.
b == a[n-1].b)
                                        it = a.end()-1;
                                if (it != a.end() && it != a.begin()) {
                                        int p1 = int (it - a.begin());
                                        if (sq (p[p1], p[p1-1], q) <= 0)
                                                in = true;
                                }
                        }
                puts (in ? "INSIDE" : "OUTSIDE");
        }

}
```

Нахождение вписанной окружности в выпуклом многоугольнике с помощью тернарного поиска

Дан выпуклый многоугольник с N вершинами. Требуется найти координаты центра и радиус наибольшей вписанной окружности.

Здесь описывается простой метод решения этой задачи с помощью двух тернарных поисков, работающий за $O(N \log^2 C)$, где C - коэффициент, определяемый величиной координат и требуемой точностью (см. ниже).

Алгоритм

Определим функцию $Radius(X, Y)$, возвращающую радиус вписанной в данный многоугольник окружности с центром в точке (X;Y). Предполагается, что точки X и Y лежат внутри (или на границе) многоугольника. Очевидно, эту функцию легко реализовать с асимптотикой $O(N)$ - просто проходим по всем сторонам многоугольника, считаем для каждой расстояние до центра (причём расстояние можно брать как от прямой до точки, не обязательно рассматривать как отрезок), и возвращаем минимум из найденных расстояний - очевидно, он и будет наибольшим радиусом.

Итак, нам нужно максимизировать эту функцию. Заметим, что, поскольку многоугольник выпуклый, то эта функция будет пригодна для **тернарного поиска** по обоим аргументам: при фиксированном X_0 (разумеется, таком, что прямая $X=X_0$ пересекает многоугольник) функция $Radius(X_0, Y)$ как функция одного аргумента Y будет сначала возрастать, затем убывать (опять же, мы рассматриваем только такие Y, что точка (X_0, Y) принадлежит многоугольнику). Более того, функция max (по Y) { Radius (X, Y) } как функция одного аргумента X будет сначала возрастать, затем убывать. Эти свойства ясны из геометрических соображений.

Таким образом, нам нужно сделать два тернарных поиска: по X и внутри него по Y, максимизируя значение функции Radius. Единственный особый момент - нужно правильно выбирать границы тернарных поисков, поскольку вычисление функции Radius за пределами многоугольника будет некорректным. Для поиска по X никаких сложностей нет, просто выбираем абсциссу самой левой и самой правой точки. Для поиска по Y находим те отрезки многоугольника, в которые попадает текущий X, и находим ординаты точек этих отрезков при абсциссе X (вертикальные отрезки не рассматриваем).

Осталось оценить **асимптотику**. Пусть максимальное значение, которое могут принимать координаты - это C_1, а требуемая точность - порядка 10^{-C_2}, и пусть $C = C_1 + C_2$. Тогда количество шагов, которые должен будет совершить каждый тернарный поиск, есть величина $O(\log C)$, и итоговая асимптотика получается: $O(N \log^2 C)$.

Реализация

Константа steps определяет количество шагов обоих тернарных поисков.

В реализации стоит отметить, что для каждой стороны сразу предпосчитываются коэффициенты в уравнении прямой, и сразу же нормализуются (делятся на $sqrt(A^2+B^2)$), чтобы избежать лишних операций внутри тернарного поиска.

```
const double EPS = 1E-9;
int steps = 60;

struct pt {
        double x, y;
};

struct line {
        double a, b, c;
};

double dist (double x, double y, line & l) {
        return abs (x * l.a + y * l.b + l.c);
}

double radius (double x, double y, vector<line> & l) {
        int n = (int) l.size();
        double res = INF;
        for (int i=0; i<n; ++i)
                res = min (res, dist (x, y, l[i]));
        return res;
```

```
}

double y_radius (double x, vector<pt> & a, vector<line> & l) {
        int n = (int) a.size();
        double ly = INF,  ry = -INF;
        for (int i=0; i<n; ++i) {
                int x1 = a[i].x,  x2 = a[(i+1)%n].x,  y1 = a[i].y,  y2 = a
[(i+1)%n].y;
                if (x1 == x2)  continue;
                if (x1 > x2)  swap (x1, x2),  swap (y1, y2);
                if (x1 <= x+EPS && x-EPS <= x2) {
                        double y = y1 + (x - x1) * (y2 - y1) / (x2 - x1);
                        ly = min (ly, y);
                        ry = max (ry, y);
                }
        }
        for (int sy=0; sy<steps; ++sy) {
                double diff = (ry - ly) / 3;
                double y1 = ly + diff,  y2 = ry - diff;
                double f1 = radius (x, y1, l),  f2 = radius (x, y2, l);
                if (f1 < f2)
                        ly = y1;
                else
                        ry = y2;
        }
        return radius (x, ly, l);
}

int main() {

        int n;
        vector<pt> a (n);
        ... чтение a ...

        vector<line> l (n);
        for (int i=0; i<n; ++i) {
                l[i].a = a[i].y - a[(i+1)%n].y;
                l[i].b = a[(i+1)%n].x - a[i].x;
                double sq = sqrt (l[i].a*l[i].a + l[i].b*l[i].b);
                l[i].a /= sq,  l[i].b /= sq;
                l[i].c = - (l[i].a * a[i].x + l[i].b * a[i].y);
        }

        double lx = INF,  rx = -INF;
        for (int i=0; i<n; ++i) {
                lx = min (lx, a[i].x);
                rx = max (rx, a[i].x);
        }

        for (int sx=0; sx<stepsx; ++sx) {
                double diff = (rx - lx) / 3;
                double x1 = lx + diff,  x2 = rx - diff;
                double f1 = y_radius (x1, a, l),  f2 = y_radius (x2, a, l);
                if (f1 < f2)
                        lx = x1;
                else
                        rx = x2;
        }

        double ans = y_radius (lx, a, l);
        printf ("%.7lf", ans);

}
```

Нахождение вписанной окружности в выпуклом многоугольнике методом "сжатия сторон" ("shrinking sides") за $O(n \log n)$

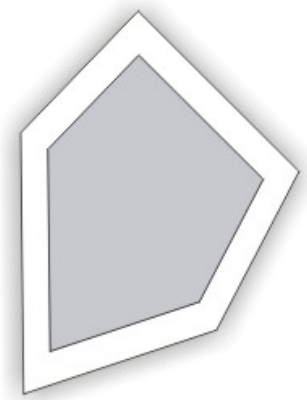
Дан выпуклый многоугольник с n вершинами. Требуется найти вписанную в него окружность максимального радиуса: т. е. найти её радиус и координаты центра. (Если при данном радиусе возможны несколько вариантов центров, то достаточно найти любой из них.)

В отличие от описанного здесь метода двойного тернарного поиска, асимптотика данного алгоритма — $O(n \log n)$ — не зависит от ограничений на координаты и от требуемой точности, и поэтому этот алгоритм подходит при значительно больших n и больших ограничениях на величину координат.

Спасибо **Ивану Красильникову** (mf) за описание этого красивого алгоритма.

Алгоритм

Итак, дан выпуклый многоугольник. Начнём одновременно и с одинаковой скоростью **сдвигать** все его стороны параллельно самим себе внутрь многоугольника:



Пусть, для удобства, это движение происходит со скоростью 1 координатная единица в секунду (т.е. время в каком-то смысле равно расстоянию: спустя единицу времени каждая точка преодолеет расстояние, равное единице).

В процессе этого движения стороны многоугольника будут постепенно исчезать (обращаться в точки). Рано или поздно весь многоугольник сожмётся в точку или отрезок, и этот момент времени t будет являться **ответом на задачу** — искомым радиусом (а центр искомой окружности будет лежать на этом отрезке). В самом деле, если мы сжали многоугольник на толщину t по всем направлениям, и он обратился в точку/отрезок, то это означает, что существует точка, отстоящая от всех сторон многоугольника на расстоянии t, а для бОльших расстояний — такой точки уже не существует.

Итак, нам надо научиться эффективно моделировать этот процесс сжатия. Для этого научимся для каждой стороны **определять время**, через которое она сожмётся в точку.

Для этого рассмотрим внимательно процесс движения сторон. Заметим, что вершины многоугольника всегда движутся по биссектрисам углов (это следует из равенства соответствующих треугольников). Но тогда вопрос о времени, через которое сторона сожмётся, сводится к вопросу об определении высоты H треугольника, в котором известна длина стороны L и два прилежащих к ней угла α и β. Воспользовавшись, например, теоремой синусов, получаем формулу:

$$H = L \cdot \frac{\sin\alpha \cdot \sin\beta}{\sin(\alpha+\beta)}.$$

Теперь мы умеем за $O(1)$ определять время, через которое сторона сожмётся в точку.

Занесём эти времена для каждой стороны в некую **структуру данных для извлечения минимума**, например, красно-чёрное дерево (set в языке C++).

Теперь если мы извлечём сторону с **наименьшим временем** H, то эта сторона первой сожмётся в точку — в момент времени H. Если многоугольник ещё не сжался в точку/отрезок, то эту сторону надо **удалить** из многоугольника, и продолжить алгоритм для оставшихся сторон. При удалении стороны мы должны **соединить** друг с другом её левого и правого соседа, **продлив** их до точки их пересечения. При этом необходимо будет найти эту точку пересечения, пересчитать длины двух сторон и их времена исчезновения.

При реализации для каждой стороны придётся хранить номер её правого и левого соседа (тем самым как бы построив двусвязный список из сторон многоугольника). Это позволяет реализовать удаление стороны и связывание двух её соседей за $O(1)$.

Если при удалении стороны оказывается, что её стороны-соседи **параллельны**, то это означает, что многоугольник после этого сжатия вырождается в точку/отрезок, поэтому мы можем сразу останавливать алгоритм и возвращать в качестве ответа время исчезновения текущей стороны (так что проблем с параллельными сторонами не возникает).

Если же такая ситуация с параллельными сторонами не возникает, то алгоритм доработает до момента, в который в многоугольнике останется только две стороны — и тогда ответом на задачу будет являться время удаления предыдущей стороны.

Очевидно, асимптотика этого алгоритма составляет $O(n \log n)$, поскольку алгоритм состоит из n шагов, на каждом из которых удаляется по одной стороне (для чего производится несколько операций с set за время $O(n \log n)$).

Реализация

Приведём реализацию описанного выше алгоритма. Данная реализация возвращает только радиус искомой окружности; впрочем, добавление вывода центра окружности не составит большого труда.

Данный алгоритм элегантен тем, что из вычислительной геометрии требуется только нахождение угла между двумя сторонами, пересечение двух прямых и проверка двух прямых на параллельность.

Примечание. Предполагается, что подаваемый на вход многоугольник — **строго выпуклый**, т.е. никакие три точки не лежат на одной прямой.

```
const double EPS = 1E-9;
const double PI = ...;

struct pt {
        double x, y;
        pt()  { }
        pt (double x, double y) : x(x), y(y)  { }
        pt operator- (const pt & p) const {
                return pt (x-p.x, y-p.y);
        }
};

double dist (const pt & a, const pt & b) {
        return sqrt ((a.x-b.x)*(a.x-b.x) + (a.y-b.y)*(a.y-b.y));
}

double get_ang (const pt & a, const pt & b) {
        double ang = abs (atan2 (a.y, a.x) - atan2 (b.y, b.x));
        return min (ang, 2*PI-ang);
}

struct line {
        double a, b, c;
        line (const pt & p, const pt & q) {
                a = p.y - q.y;
                b = q.x - p.x;
                c = - a * p.x - b * p.y;
                double z = sqrt (a*a + b*b);
                a/=z, b/=z, c/=z;
        }
};

double det (double a, double b, double c, double d) {
        return a * d - b * c;
}

pt intersect (const line & n, const line & m) {
        double zn = det (n.a, n.b, m.a, m.b);
        return pt (
                - det (n.c, n.b, m.c, m.b) / zn,
                - det (n.a, n.c, m.a, m.c) / zn
        );
}
```

```
bool parallel (const line & n, const line & m) {
        return abs (det (n.a, n.b, m.a, m.b)) < EPS;
}

double get_h (const pt & p1, const pt & p2,
        const pt & l1, const pt & l2, const pt & r1, const pt & r2)
{
        pt q1 = intersect (line (p1, p2), line (l1, l2));
        pt q2 = intersect (line (p1, p2), line (r1, r2));
        double l = dist (q1, q2);
        double alpha = get_ang (l2 - l1, p2 - p1) / 2;
        double beta = get_ang (r2 - r1, p1 - p2) / 2;
        return l * sin(alpha) * sin(beta) / sin(alpha+beta);
}

struct cmp {
        bool operator() (const pair<double,int> & a, const pair<double,int>
& b) const {
                if (abs (a.first - b.first) > EPS)
                        return a.first < b.first;
                return a.second < b.second;
        }
};

int main() {
        int n;
        vector<pt> p;
        ... чтение n и p ...

        vector<int> next (n), prev (n);
        for (int i=0; i<n; ++i) {
                next[i] = (i + 1) % n;
                prev[i] = (i - 1 + n) % n;
        }

        set < pair<double,int>, cmp > q;
        vector<double> h (n);
        for (int i=0; i<n; ++i) {
                h[i] = get_h (
                        p[i], p[next[i]],
                        p[i], p[prev[i]],
                        p[next[i]], p[next[next[i]]]
                );
                q.insert (make_pair (h[i], i));
        }

        double last_time;
        while (q.size() > 2) {
                last_time = q.begin()->first;
                int i = q.begin()->second;
                q.erase (q.begin());

                next[prev[i]] = next[i];
                prev[next[i]] = prev[i];
                int nxt = next[i],   nxt1 = (nxt+1)%n,
                        prv = prev[i],   prv1 = (prv+1)%n;
                if (parallel (line (p[nxt], p[nxt1]), line (p[prv], p[prv1])))
                        break;

                q.erase (make_pair (h[nxt], nxt));
                q.erase (make_pair (h[prv], prv));

                h[nxt] = get_h (
                        p[nxt], p[nxt1],
                        p[prv1], p[prv],
                        p[next[nxt]], p[(next[nxt]+1)%n]
                );
                h[prv] = get_h (
                        p[prv], p[prv1],
                        p[(prev[prv]+1)%n], p[prev[prv]],
```

```
                        p[nxt], p[nxt1]
                );

                q.insert (make_pair (h[nxt], nxt));
                q.insert (make_pair (h[prv], prv));
        }

        cout << last_time << endl;
}
```

Основная функция здесь — это $get_h()$, которая по стороне и её левому и правому соседям вычисляет время исчезновения этой стороны. Для этого ищется точка пересечения этой стороны с соседями, и затем по приведённой выше формуле производится рассчёт искомого времени.

Диаграмма Вороного в 2D

Определение

Даны n точек $P_i(x_i, y_i)$ на плоскости. Рассмотрим разбиение плоскости на n областей V_i (называемых многоугольниками Вороного или ячейками Вороного, иногда — многоугольниками близости, ячейками Дирихле, разбиением Тиссена), где V_i — множество всех точек плоскости, которые находятся ближе к точке P_i, чем ко всем остальным точкам P_k:

$$V_i = \{(x, y) : \rho((x, y), P_i) = \min_{k=1\ldots N} \rho((x, y), P_k)\}$$

Само разбиение плоскости называется диаграммой Вороного данного набора точек P_k.

Здесь $\rho(p, q)$ — заданная метрика, обычно это стандартная Евклидова метрика: $\rho(p, q) = \sqrt{(x_p - x_q)^2 + (y_p - y_q)^2}$, однако ниже будет рассмотрен и случай так называемой манхэттенской метрики. Здесь и далее, если не оговорено иного, будет рассматриваться случай Евклидовой метрики

Ячейки Вороного представляют собой выпуклые многоугольники, некоторые являются бесконечными. Точки, принадлежащие согласно определению сразу нескольким ячейкам Вороного, обычно так и относят сразу к нескольким ячейкам (в случае Евклидовой метрики множество таких точек имеет меру нуль; в случае манхэттенской метрики всё несколько сложнее).

Такие многоугольники впервые были глубоко изучены русским математиком Вороным (1868-1908 гг.).

Свойства

- Диаграмма Вороного является планарным графом, поэтому она имеет $O(n)$ вершин и рёбер.
- Зафиксируем любое $i = 1 \ldots n$. Тогда для каждого $j = 1 \ldots n, j \neq i$ проведём прямую — серединный перпендикуляр отрезка (P_i, P_j); рассмотрим ту полуплоскость, образуемую этой прямой, в которой лежит точка P_i. Тогда пересечение всех полуплоскостей для каждого j даст ячейку Вороного P_i.
- Каждая вершина диаграммы Вороного является центром окружности, проведённой через какие-либо три точки множества P. Эти окружности существенно используются во многих доказательствах, связанных с диаграммами Вороного.
- Ячейка Вороного V_i является бесконечной тогда и только тогда, когда точка P_i лежит на границе выпуклой оболочки множества P_k.
- Рассмотрим граф, двойственный к диаграмме Вороного, т.е. в этом графе вершинами будут точки P_i, а ребро проводится между точками P_i и P_j, если их ячейки Вороного V_i и V_j имеют общее ребро. Тогда, при условии, что никакие четыре точки не лежат на одной окружности, двойственный к диаграмме Вороного граф является триангуляцией Делоне (обладающей множеством интересных свойств).

Применение

Диаграмма Вороного представляет собой компактную структуру данных, хранящую всю необходимую информацию для решения множества задач о близости.

В рассмотренных ниже задачах время, необходимое на построение самой диаграммы Вороного, в асимптотиках не учитывается.

- Нахождение ближайшей точки для каждой.

  Отметим простой факт: если для точки P_i ближайшей является точка P_j, то эта точка P_j имеет "своё" ребро в ячейке V_i. Отсюда следует, что, чтобы найти для каждой точки ближайшую к ней, достаточно просмотреть рёбра её ячейки Вороного. Однако каждое ребро принадлежит ровно двум ячейкам, поэтому будет просмотрено ровно два раза, и вследствие линейности числа рёбер мы получаем решение данной задачи за $O(n)$.

- Нахождение выпуклой оболочки.

  Вспомним, что вершина принадлежит выпуклой оболочке тогда и только тогда, когда её ячейка Вороного бесконечна. Тогда найдём в диаграмме Вороного любое бесконечное ребро, и начнём двигаться в каком-либо фиксированном направлении (например, против часовой стрелки) по ячейке, содержащей это ребро, пока не дойдём

до следующего бесконечного ребра. Тогда перейдём через это ребро в соседнюю ячейку и продолжим обход. В результате все просмотренные рёбра (кроме бесконечных) будут являться сторонами искомой выпуклой оболочки. Очевидно, время работы алгоритма - $O(n)$.

- Нахождение Евклидова минимального остовного дерева.

  Требуется найти минимальное остовное дерево с вершинами в данных точках P, соединяющее все эти точки. Если применять стандартные методы теории графов, то, т.к. граф в данном случае имеет $O(n^2)$ рёбер, даже оптимальный алгоритм будет иметь не меньшую асимптотику.

  Рассмотрим граф, двойственный диаграмме Вороного, т.е. триангуляцию Делоне. Можно показать, что нахождение Евклидова минимального остова эквивалентно построению остова триангуляции Делоне. Действительно, в алгоритме Прима каждый раз ищется кратчайшее ребро между двумя множествами точек; если мы зафиксируем точку одного множества, то ближайшая к ней точка имеет ребро в ячейке Вороного, поэтому в триангуляции Делоне будет присутствовать ребро к ближайшей точке, что и требовалось доказать.

  Триангуляция является планарным графом, т.е. имеет линейное число рёбер, поэтому к ней можно применить алгоритм Крускала и получить алгоритм с временем работы $O(n \log n)$.

- Нахождение наибольшей пустой окружности.

  Требуется найти окружность наибольшего радиуса, не содержащую внутри никакую из точек P_i (центр окружности должен лежать внутри выпуклой оболочки точек P_i). Заметим, что, т.к. функция наибольшего радиуса окружности в данной точке $f(x, y)$ является строго монотонной внутри каждой ячейки Вороного, то она достигает своего максимума в одной из вершин диаграммы Вороного, либо в точке пересечения рёбер диаграммы и выпуклой оболочки (а число таких точек не более чем в два раза больше числа рёбер диаграммы). Таким образом, остаётся только перебрать указанные точки и для каждой найти ближайшую, т.е. решение за $O(n)$.

Простой алгоритм построения диаграммы Вороного за $O(n^4)$

Диаграммы Вороного — достаточно хорошо изученный объект, и для них получено множество различных алгоритмов, работающих за оптимальную асимптотику $O(n \log n)$, а некоторые из этих алгоритмов даже работают в среднем за $O(n)$. Однако все эти алгоритмы весьма сложны.

Рассмотрим здесь самый простой алгоритм, основанный на приведённом выше свойстве, что каждая ячейка Вороного представляет собой пересечение полуплоскостей. Зафиксируем i. Проведём между точкой P_i и каждой точкой P_j прямую — серединный перпендикуляр, затем пересечём попарно все полученные прямые — получим $O(n^2)$ точек, и каждую проверим на принадлежность всем n полуплоскостям. В результате за $O(n^3)$ действий мы получим все вершины ячейки Вороного V_i (их уже будет не более n, поэтому мы можем без ухудшения асимптотики отсортировать их по полярному углу), а всего на построение диаграммы Вороного потребуется $O(n^4)$ действий.

Случай особой метрики

Рассмотрим следующую метрику:

$$\rho(p,q) = \max(|x_p - x_q|, |y_p - y_q|)$$

Начать рассмотрение следует с разбора простейшего случая — случая двух точек A и B.

Если $A_x = B_x$ или $A_y = B_y$, то диаграммой Вороного для них будет соответственно вертикальная или горизонтальная прямая.

Иначе диаграмма Вороного будет иметь вид "уголка": отрезок под углом 45 градусов в прямоугольнике, образованном точками A и B, и горизонтальные/вертикальные лучи из его концов в зависимости от того, длиннее ли вертикальная сторона прямоугольника или горизонтальная.

Особый случай — когда этот прямоугольник имеет одинаковую длину и ширину, т.е. $|A_x - B_x| = |A_y - B_y|$. В этом случае будут иметься две бесконечные области ("уголки", образованные двумя лучами, параллельными осям), которые по определению должны принадлежать сразу обеим ячейкам. В таком случае дополнительно определяют в условии, как следует понимать эти области (иногда искусственно вводят правило, по которому каждый уголок относят к своей ячейке).

Таким образом, уже для двух точек диаграмма Вороного в данной метрике представляет собой нетривиальный объект, а в случае большего числа точек эти фигуры надо будет уметь быстро пересекать.

Нахождение всех граней, внешней грани планарного графа

Дан планарный, уложенный на плоскости граф G с n вершинами. Требуется найти все его грани. Гранью называется часть плоскости, ограниченная рёбрами этого графа.

Одна из граней будет отличаться от остальных тем, что будет иметь бесконечную площадь, такая грань называется внешней гранью. В некоторых задачах требуется находить только внешнюю грань, алгоритм нахождения которой, как мы увидим, по сути ничем не отличается от алгоритма для всех граней.

Теорема Эйлера

Приведём здесь теорему Эйлера и несколько следствий из неё, из которых будет следовать, что число рёбер и граней планарного простого (без петель и кратных рёбер) графа являются величинами порядка $O(n)$.

Пусть планарный граф G является связным. Обозначим через n число вершин в графе, m — число рёбер, f — число граней. Тогда справедлива **теорема Эйлера**:

$$f + n - m = 2$$

Доказать эту формулу легко следующим образом. В случае дерева ($m = n - 1$) формула легко проверяется. Если граф — не дерево, то удалим любое ребро, принадлежащее какому-либо циклу; при этом величина $f + n - m$ не изменится. Будем повторять этот процесс, пока не придём к дереву, для которого тождество $f + n - m = 2$ уже установлено. Таким образом, теорема доказана.

Следствие. Для произвольного планарного графа пусть k — количество компонент связности. Тогда выполняется:

$$f + n - m = 1 + k$$

Следствие. Число рёбер m простого планарного графа является величиной $O(n)$.

Доказательство. Пусть граф G является связным и $n \geq 3$ (в случае $n < 3$ утверждение получаем автоматически). Тогда, с одной стороны, каждая грань ограничена как минимум тремя рёбрами. С другой стороны, каждое ребро ограничивает максимум две грани. Следовательно, $3f \leq 2m$, откуда, подставляя это в формулу Эйлера, получаем:

$$f+n-m=2 \quad \Leftrightarrow \quad 3f=6-3n+3m \quad \Leftrightarrow \quad 6-3n+3m \leq 2m \quad \Leftrightarrow \quad m \leq 3n-6$$

Т.е. $m = O(n)$.

Если граф не является связным, то, суммируя полученные оценки по его компонентам связности, снова получаем $m = O(n)$, что и требовалось доказать.

Следствие. Число граней f простого планарного графа является величиной $O(n)$.

Это следствие вытекает из предыдущего следствия и связи $f = 2 - n + m$.

Обход всех граней

Всегда будем считать, что граф, если он не является связным, уложен на плоскости таким образом, что никакая компонента связности не лежит внутри другой (например, квадрат с лежащим строго внутри него отрезком — некорректный для нашего алгоритма тест).

Разумеется, считается, что граф корректно уложен на плоскости, т.е. никакие две вершины не совпадают, а рёбра не пересекаются в "несанкционированных" точках. Если во входном графе допускаются такие пересекающиеся рёбра, то предварительно надо избавиться от них, вводя в каждую точку пересечения дополнительную вершину (надо заметить, что в результате этого процесса вместо n точек мы можем получить порядка n^2 точек). Более подробно об этом процессе см. ниже в соответствующем разделе.

Пусть для каждой вершины все исходящие из неё рёбра упорядочены по полярному углу. Если это не так, то их следует упорядочить, произведя сортировку каждого списка смежности (т.к. $m = O(n)$, на это потребуется $O(n \log n)$ операций).

Теперь выберем произвольное ребро (a, b) и пустим следующий обход. Приходя в какую-то вершину v по некоторому ребру, выходить из этой вершины мы обязательно должны по следующему в порядке сортировки ребру.

Например, на первом шаге мы находимся в вершине b, и должны найти вершину a в списке смежности вершины b, тогда обозначим через c следующую вершину в списке смежности (если a была последней, то в качестве c возьмём первую вершину), и пройдём по ребру (b, c).

Повторяя этот процесс много раз, мы рано или поздно придём обратно к стартовому ребру (a, b), после чего надо остановиться. Нетрудно заметить, что при таком обходе мы обойдём ровно одну грань. Причём направление обхода будет против часовой стрелки для внешней грани, и по часовой стрелке — для внутренних граней. Иными словами, при таком обходе внутренность грани будет всегда по правую сторону от текущего ребра.

Итак, мы научились обходить одну грань, стартуя с любого ребра на её границе. Осталось научиться выбирать стартовые рёбра таким образом, чтобы получаемые грани не повторялись. Заметим, что у каждого ребра различаются два направления, в которых его можно обходить: при каждом из них будут получаться свои грани. С другой стороны, ясно, что одно такое ориентированное ребро принадлежит ровно одной грани. Таким образом, если мы будет помечать все рёбра каждой обнаруженной грани в некотором массиве used, и не запускать обход из уже помеченных рёбер, то мы обойдём все грани (включая внешнюю), притом ровно по одному разу.

Приведём сразу **реализацию** этого обхода. Будем считать, что в графе G списки смежности уже упорядочены по углу, а кратные рёбра и петли отсутствуют.

Первый вариант реализации упрощённый, следующую вершину в списке смежности он ищет простым поиском. Такая реализация теоретически работает за $O(n^2)$, хотя на практике на многих тестах она работает весьма быстро (со скрытой константой, значительно меньшей единицы).

```
int n; // число вершин
vector < vector<int> > g; // граф

vector < vector<char> > used (n);
for (int i=0; i<n; ++i)
        used[i].resize (g[i].size());
for (int i=0; i<n; ++i)
        for (size_t j=0; j<g[i].size(); ++j)
                if (!used[i][j]) {
                        used[i][j] = true;
                        int v = g[i][j],  pv = i;
                        vector<int> facet;
                        for (;;) {
                                facet.push_back (v);
                                vector<int>::iterator it = find (g[v].begin(),
g[v].end(), pv);
                                if (++it == g[v].end())  it = g[v].begin();
                                if (used[v][it-g[v].begin()])  break;
                                used[v][it-g[v].begin()] = true;
                                pv = v,  v = *it;
                        }
                        ... вывод facet - текущей грани ...
                }
```

Другой, более оптимизированный вариант реализации — пользуется тем, что вершине в списке смежности упорядочены по углу. Если реализовать функцию $\mathrm{cmp_ang}$ сравнения двух точек по полярному углу относительно третьей точки (например, оформив её в виде класса, как в примере ниже), то при поиске точки в списке смежности можно воспользоваться бинарным поиском. В результате получаем реализацию за $O(n \log n)$.

```
class cmp_ang {
        int center;
public:
        cmp_ang (int center) : center(center)
                { }
        bool operator() (int a, int b) const {
                ... должна возвращать true, если точка a имеет
                меньший чем b полярный угол относительно center ...
        }
};


int n; // число вершин
vector < vector<int> > g; // граф

vector < vector<char> > used (n);
for (int i=0; i<n; ++i)
        used[i].resize (g[i].size());
for (int i=0; i<n; ++i)
```

```
        for (size_t j=0; j<g[i].size(); ++j)
                if (!used[i][j]) {
                        used[i][j] = true;
                        int v = g[i][j],  pv = i;
                        vector<int> facet;
                        for (;;) {
                                facet.push_back (v);
                                vector<int>::iterator it = lower_bound (g
[v].begin(), g[v].end(),
                                        pv, cmp_ang(v));
                                if (++it == g[v].end())  it = g[v].begin();
                                if (used[v][it-g[v].begin()])  break;
                                used[v][it-g[v].begin()] = true;
                                pv = v,  v = *it;
                        }
                        ... вывод facet - текущей грани ...
                }
```

Возможен и вариант, основанный на контейнере map, ведь нам нужно всего лишь быстро узнавать позиции чисел в массиве. Разумеется, такая реализация также будет работать $O(n \log n)$.

Следует отметить, что алгоритм не совсем корректно работает с **изолированными** вершинами — такие вершины он просто не обнаружит как отдельные грани, хотя, с математической точки зрения, они должны представлять собой отдельные компоненты связности и грани.

Кроме того, особой гранью является **внешняя грань**. Как её отличать от "обычных" граней, описано в следующем разделе. Следует заметить, что если граф является не связным, то внешняя грань будет состоять из нескольких контуров, и каждый из этих контуров будет найден алгоритмом отдельно.

Выделение внешней грани

Приведённый выше код выводит все грани, не делая различия между внешней гранью и внутренними гранями. На практике обычно, наоборот, требуется найти или только внешнюю грань, или только внутренние. Есть несколько приёмов выделения внешней грани.

Например, её можно определять по площади — внешняя грань должна иметь наибольшую площадь (следует только учесть, что внутренняя грань может иметь ту же площадь, что и внешняя). Этот способ не будет работать, если данный планарный граф G не является связным.

Другой, более надёжный критерий — по направлению обхода. Как уже отмечалось выше, все грани, кроме внешней, обходятся в направлении по часовой стрелки. Внешняя грань, даже если она состоит из нескольких контуров, обойдётся алгоритмом против часовой стрелки. Определить направление обхода можно, просто посчитав знаковую площадь многоугольника. Площадь можно считать прямо по ходу внутреннего цикла. Однако и у этого метода есть своя тонкость — обработка граней нулевой площади. Например, если граф состоит из единственного ребра, то алгоритм найдёт единственную грань, площадь которой будет нулевой. По-видимому, если грань имеет нулевую площадь, то она является внешней гранью.

В некоторых случаях бывает применим и такой критерий, как количество вершин. Например, если граф представляет собой выпуклый многоугольник с проведёнными в нём непересекающимися диагоналями, то его внешняя грань будет содержать все вершины. Но снова надо быть аккуратным со случаем, когда и внешняя, и внутренняя грани имеют одинаковое число вершин.

Наконец есть и следующий метод нахождения внешней грани: можно специально запуститься от такого ребра, что найденная в результате грань будет внешней. Например, можно взять самую левую вершину (если таких несколько, то подойдёт любая) и выбрать из неё ребро, идущее первым в порядке сортировки. В результате обход из этого ребра найдёт внешнюю грань. Этот способ можно распространить и на случай несвязного графа: нужно в каждой компоненте связности найти самую левую вершину и запускать обход из первого ребра из неё.

Приведём реализацию самого простого метода, основанного на знаке площади (сам обход я для примера взял за $O(n^2)$, здесь это неважно). Если граф не связный, то код "... грань является внешней ..." выполнится отдельно для каждого контура, составляющего внешнюю грань.

```
                        ... обычный код по обнаружению граней ...
                        ... сразу после цикла, обнаруживающего
очередную грань: ...

                        // считаем площадь
                        double area = 0;
                        // добавляем фиктивную точку для простоты
подсчёта площади
                        facet.push_back (facet[0]);
```

```
                        for (size_t k=0; k+1<facet.size(); ++k)
                                area += (p[facet[k]].first + p[facet[k
+1]].first)
                                        * (p[facet[k]].second - p[facet[k
+1]].second);
                        if (area < EPS)
                                ... грань является внешней ...
                }
```

Построение планарного графа

Для вышеописанных алгоритмов существенно то, что входной граф является корректно уложенным планарным графом. Однако на практике часто на вход программе подаётся набор отрезков, возможно, пересекающихся между собой в "несанкционированных" точках, и нужно по этим отрезкам построить планарный граф.

Реализовать построение планарного графа можно следующим образом. Зафиксируем какой-либо входной отрезок. Теперь пересечём этот отрезок со всеми остальными отрезками. Найденные точки пересечения, а также концы самого отрезка положим в вектор, и его отсортируем стандартным образом (т.е. сначала по одной координате, при равенстве — по другой). Потом пройдёмся по этому вектору и будет добавлять рёбра между соседними в этом векторе точками (разумеется, следя, чтобы мы не добавили петли). Выполнив этот процесс для всех отрезков, т.е. за $O(n^2 \log n)$, мы построим соответствующий планарный граф (в котором будет $O(n^2)$ точек).

Реализация:

```
const double EPS = 1E-9;

struct point {
        double x, y;
        bool operator< (const point & p) const {
                return x < p.x - EPS || abs (x - p.x) < EPS && y < p.y - EPS;
        }
};

map<point,int> ids;
vector<point> p;
vector < vector<int> > g;

int get_point_id (point pt) {
        if (!ids.count(pt)) {
                ids[pt] = (int)p.size();
                p.push_back (pt);
                g.resize (g.size() + 1);
        }
        return ids[p];
}

void intersect (pair<point,point> a, pair<point,point> b, vector<point> & res) {
        ... стандартная процедура, пересекает два отрезка a и b и
закидывает результат в res ...
        ... если отрезки перекрываются, то закидывает те концы, которые
попали внутрь первого отрезка ...
}

int main() {
        // входные данные
        int m;
        vector < pair<point,point> > a (m);
        ... чтение ...

        // построение графа
        for (int i=0; i<m; ++i) {
                vector<point> cur;
                for (int j=0; j<m; ++j)
                        intersect (a[i], a[j], cur);
                sort (cur.begin(), cur.end());
                for (size_t j=0; j+1<cur.size(); ++j) {
                        int x = get_id (cur[j]),  y = get_id (cur[j+1]);
                        if (x != y) {
```

```
                                g[x].push_back (y);
                                g[y].push_back (x);
                        }
                }
        }
        int n = (int) g.size();
        // сортировка по углу и удаление кратных рёбер
        for (int i=0; i<n; ++i) {
                sort (g[i].begin(), g[i].end(), cmp_ang (i));
                g[i].erase (unique (g[i].begin(), g[i].end()), g[i].end());
        }
}
```

Нахождение пары ближайших точек

Постановка задачи

Даны n точек p_i на плоскости, заданные своими координатами (x_i, y_i). Требуется найти среди них такие две точки, расстояние между которыми минимально:

$$\min_{\substack{i,j=0\ldots n-1,\\ i\neq j}} \rho(p_i, p_j).$$

Расстояния мы берём обычные евклидовы:

$$\rho(p_i, p_j) = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}.$$

Тривиальный алгоритм — перебор всех пар и вычисление расстояния для каждой — работает за $O(n^2)$. Ниже описывается алгоритм, работающий за время $O(n \log n)$. Этот алгоритм был предложен Препаратой (Preparata) в 1975 г. Препарата и Шамос также показали, что в модели дерева решений этот алгоритм асимптотически оптимален.

Алгоритм

Построим алгоритм по общей схеме алгоритмов "**разделяй-и-властвуй**": алгоритм оформляем в виде рекурсивной функции, которой передаётся множество точек; эта рекурсивная функция разбивает это множество пополам, вызывает себя рекурсивно от каждой половины, а затем выполняет какие-то операции по объединению ответов. Операция объединения заключается в обнаружении случаев, когда одна точка оптимального решения попала в одну половину, а другая точка — в другую (в этом случае рекурсивные вызовы от каждой из половинок отдельно обнаружить эту пару, конечно, не смогут). Основная сложность, как всегда, заключается в эффективной реализации этой стадии объединения. Если рекурсивной функции передаётся множество из n точек, то стадия объединения должна работать не более, чем $O(n)$, тогда асимптотика всего алгоритма $T(n)$ будет находиться из уравнения:

$$T(n) = 2T(n/2) + O(n).$$

Решением этого уравнения, как известно, является $T(n) = O(n \log n)$.

Итак, перейдём к построению алгоритма. Чтобы в будущем прийти к эффективной реализации стадии объединения, разбивать множество точек на два будем согласно их x-координатам: фактически мы проводим некоторую вертикальную прямую, разбивающую множество точек на два подмножества примерно одинаковых размеров. Такое разбиение удобно произвести следующим образом: отсортируем точки стандартно как пары чисел, т.е.:

$$p_i < p_j \iff (x_i < x_j) \vee \Big((x_i = x_j) \wedge (y_i < y_j)\Big).$$

Тогда возьмём среднюю после сортировки точку p_m ($m = \lfloor n/2 \rfloor$), и все точки до неё и саму p_m отнесём к первой половине, а все точки после неё — ко второй половине:

$$\begin{aligned} A_1 &= \{p_i \mid i = 0 \ldots m\}, \\ A_2 &= \{p_i \mid i = m+1 \ldots n-1\}. \end{aligned}$$

Теперь, вызвавшись рекурсивно от каждого из множеств A_1 и A_2, мы найдём ответы h_1 и h_2 для каждой из половинок. Возьмём лучший из них: $h = \min(h_1, h_2)$.

Теперь нам надо произвести **стадию объединения**, т.е. попытаться обнаружить такие пары точек, расстояние между которыми меньше h, причём одна точка лежит в A_1, а другая — в A_2. Очевидно, что для этого достаточно рассматривать только те точки, которые отстоят от вертикальной прямой раздела не расстояние, меньшее h, т.е. множество B рассматриваемых на этой стадии точек равно:

$$B = \{p_i \mid |x_i - x_m| < h\}.$$

Для каждой точки из множества B надо попытаться найти точки, находящиеся к ней ближе, чем h. Например, достаточно рассматривать только те точки, координата y которых отличается не более чем на h. Более того, не имеет смысла рассматривать те точки, у которых y-координата больше y-координаты текущей точки. Таким образом,

для каждой точки p_i определим множество рассматриваемых точек $C(p_i)$ следующим образом:

$$C(p_i) = \{p_j \mid p_j \in B, \quad y_i - h < y_j \le y_i\}.$$

Если мы отсортируем точки множества B по y-координате, то находить $C(p_i)$ будет очень легко: это несколько точек подряд до точки p_i.

Итак, в новых обозначениях **стадия объединения** выглядит следующим образом: построить множество B, отсортировать в нём точки по y-координате, затем для каждой точки $p_i \in B$ рассмотреть все точки $p_j \in C(p_i)$, и каждой пары (p_i, p_j) посчитать расстояние и сравнить с текущим наилучшим расстоянием.

На первый взгляд, это по-прежнему неоптимальный алгоритм: кажется, что размеры множеств $C(p_i)$ будут порядка n, и требуемая асимптотика никак не получится. Однако, как это ни удивительно, можно доказать, что размер каждого из множеств $C(p_i)$ есть величина $O(1)$, т.е. не превосходит некоторой малой константы вне зависимости от самих точек. Доказательство этого факта приведено в следующем разделе.

Наконец, обратим внимание на сортировки, которых вышеописанный алгоритм содержит сразу две: сначала сортировка по парам (x,y), а затем сортировка элементов множества B по y. На самом деле, от обеих этих сортировок внутри рекурсивной функции можно избавиться (иначе бы мы не достигли оценки $O(n)$ для стадии объединения, и общая асимптотика алгоритма получилась бы $O(n \log^2 n)$). От первой сортировки избавиться легко — достаточно предварительно, до запуска рекурсии, выполнить эту сортировку: ведь внутри рекурсии сами элементы не меняются, поэтому нет никакой необходимости выполнять сортировку заново. Со второй сортировкой чуть сложнее, выполнить её предварительно не получится. Зато, вспомнив **сортировку слиянием** (merge sort), которая тоже работает по принципу разделяй-и-властвуй, можно просто встроить эту сортировку в нашу рекурсию. Пусть рекурсия, принимая какое-то множество точек (как мы помним, упорядоченное по парам (x, y)) возвращает это же множество, но отсортированное уже по координате y. Для этого достаточно просто выполнить слияние (за $O(n)$) двух результатов, возвращённых рекурсивными вызовами. Тем самым получится отсортированное по y множество.

Оценка асимптотики

Чтобы показать, что вышеописанный алгоритм действительно выполняется за $O(n \log n)$, нам осталось доказать следующий факт: $|C(p_i)| = O(1)$.

Итак, пусть мы рассматриваем какую-то точку p_i; напомним, что множество $C(p_i)$ — это множество точек, y-координата которых не больше y_i, но и не меньше $y_i - h$, а, кроме того, по координате x и сама точка p_i, и все точки множества $C(p_i)$ лежат в полосе шириной $2h$. Иными словами, рассматриваемые нами точки p_i и $C(p_i)$ лежат в прямоугольнике размера $2h \times h$.

Наша задача — оценить максимальное количество точек, которое может лежать в этом прямоугольнике $2h \times h$; тем самым мы оценим и максимальный размер множества $C(p_i)$ (он будет на единицу меньше, т.к. в этом прямоугольнике лежит ещё и точка p_i). При этом надо не забывать, что в общем случае могут встречаться и повторяющиеся точки.

Вспомним, что h получалось как минимум из двух результатов рекурсивных вызовов — от множеств A_1 и A_2, причём A_1 содержит точки слева от линии раздела и частично на ней, A_2 — оставшиеся точки линии раздела и точки справа от неё. Для любой пары точек из A_1, равно как и из A_2, расстояние не может оказаться меньше h — иначе бы это означало некорректность работы рекурсивной функции.

Для оценки максимального количества точек в прямоугольнике $2h \times h$ разобьём его на два квадрата $h \times h$, к первому квадрату отнесём все точки $C(p_i) \cap A_1$, а ко второму — все остальные, т.е. $C(p_i) \cap A_2$. Из приведённых выше соображений следует, что в каждом из этих квадратов расстояние между любыми двумя точками не превосходит h. Но тогда это означает, что в каждом квадрате не более четырёх точек!

Действительно, пусть есть квадрат $h \times h$, и расстояние между любыми двумя точками не превосходит той же h. Докажем, что в квадрате не может быть больше 4 точек. Например, это можно сделать следующим образом: разобьём этот квадрат на 4 квадрата со сторонами $h/2$. Тогда в каждом из этих маленьких квадратов не может быть больше одной точки (т.к. даже диагональ равна $h/\sqrt{2}$, что меньше h). Следовательно, во всём квадрате никак не может быть более 4 точек.

Итак, мы доказали, что в прямоугольнике $2h \times h$ не может быть больше $4 \cdot 2 = 8$ точек, а, следовательно, размер множества $C(p_i)$ не может превосходить 7, что и требовалось доказать.

Реализация

Введём структуру данных для хранения точки (её координаты и некий номер) и операторы сравнения, необходимые для двух видов сортировки:

```
struct pt {
        int x, y, id;
};

inline bool cmp_x (const pt & a, const pt & b) {
        return a.x < b.x || a.x == b.x && a.y < b.y;
}

inline bool cmp_y (const pt & a, const pt & b) {
        return a.y < b.y;
}

pt a[MAXN];
```

Для удобной реализации рекурсии введём вспомогательную функцию $upd_ans()$, которая будет вычислять расстояние между двумя точками и проверять, не лучше ли это текущего ответа:

```
double mindist;
int ansa, ansb;

inline void upd_ans (const pt & a, const pt & b) {
        double dist = sqrt ((a.x-b.x)*(a.x-b.x) + (a.y-b.y)*(a.y-b.y) + .0);
        if (dist < mindist)
                mindist = dist,  ansa = a.id,  ansb = b.id;
}
```

Наконец, реализация самой рекурсии. Предполагается, что перед её вызовом массив $a[]$ уже отсортирован по x-координате. Рекурсии передаётся просто два указателя l, r, которые указывают, что она должна искать ответ для $a[l \ldots r]$. Если расстояние между r и l слишком мало, то рекурсию надо остановить, и выполнить тривиальный алгоритм поиска ближайшей пары и затем отсортировать подмассив по y-координате.

Для слияния двух множеств точек, полученных от рекурсивных вызовов, в одно (упорядоченное по y-координате), мы используем стандартную функцию STL $merge()$, и создаём вспомогательный буфер $t[]$ (один на все рекурсивные вызовы). (Использовать $inplace_merge()$ нецелесообразно, т.к. она в общем случае работает не за линейное время).

Наконец, множество B хранится в том же массиве t.

```
void rec (int l, int r) {
        if (r - l <= 3) {
                for (int i=l; i<=r; ++i)
                        for (int j=i+1; j<=r; ++j)
                                upd_ans (a[i], a[j]);
                sort (a+l, a+r+1, &cmp_y);
                return;
        }

        int m = (l + r) >> 1;
        int midx = a[m].x;
        rec (l, m),  rec (m+1, r);
        static pt t[MAXN];
        merge (a+l, a+m+1, a+m+1, a+r+1, t, &cmp_y);
        copy (t, t+r-l+1, a+l);

        int tsz = 0;
        for (int i=l; i<=r; ++i)
                if (abs (a[i].x - midx) < mindist) {
                        for (int j=tsz-1; j>=0 && a[i].y - t[j].y < mindist;
--j)
                                upd_ans (a[i], t[j]);
                        t[tsz++] = a[i];
                }
}
```

Кстати говоря, если все координаты целые, то на время работы рекурсии можно вообще не переходить к дробным величинам, и хранить в $mindist$ квадрат минимального расстояния.

В основной программе вызывать рекурсию следует так:

```
sort (a, a+n, &cmp_x);
mindist = 1E20;
rec (0, n-1);
```

Обобщение: поиск треугольника с минимальным периметром

Описанный выше алгоритм интересно обобщается и на эту задачу: среди заданного множества точек выбрать три различных точки так, чтобы сумма попарных расстояний между ними была наименьшей.

По сути, для решения этой задачи алгоритм остаётся прежним: мы разделяем поле на две половинки вертикальной прямой, вызываем решение рекурсивно от обеих половинок, выбираем минимум $minper$ из найденных периметров, строим полоску толщиной $minper/2$, и в ней перебираем все треугольники, могущие улучшить ответ. (Отметим, что у треугольника с периметром $\leq minper$ длиннейшая сторона $\leq minper/2$.)

Задачи в online judges

Список задач, которые сводятся к поиску двух ближайших точек:

- UVA 10245 "The Closest Pair Problem" [сложность: низкая]
- SPOJ #8725 CLOPPAIR "Closest Point Pair" [сложность: низкая]
- CODEFORCES Командная олимпиада школьников Саратова - 2011 "**Минимальная сумма**" [сложность: средняя]
- Google CodeJam 2009 Final "Min Perimeter" [сложность: средняя]
- SPOJ #7029 CLOSEST "Closest Triple" [сложность: средняя]

Преобразование геометрической инверсии

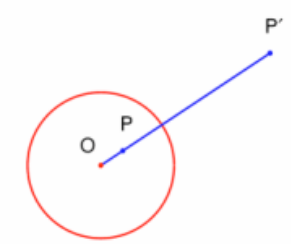
Преобразование геометрической инверсии (inversive geometry) — это особый тип преобразования точек на плоскости. Практическая польза этого преобразования в том, что зачастую оно позволяет свести решение геометрической задачи **с окружностями** к решению соответствующей задачи **с прямыми**, которая обычно имеет гораздо более простое решение.

По всей видимости, основоположником этого направления математики был Людвиг Иммануэль Магнус (Ludwig Immanuel Magnus), который в 1831 г. опубликовал статью об инверсных преобразованиях.

Определение

Зафиксируем окружность с центром в точке O радиуса r. Тогда **инверсией** точки P относительно этой окружности называется такая точка P', которая лежит на луче OP, а на расстояние наложено условие:

$$OP \cdot OP' = r^2.$$



Если считать, что центр O окружности совпадает с началом координат, то можно сказать, что точка P' имеет тот же полярный угол, что и P, а расстояние вычисляется по указанной выше формуле.

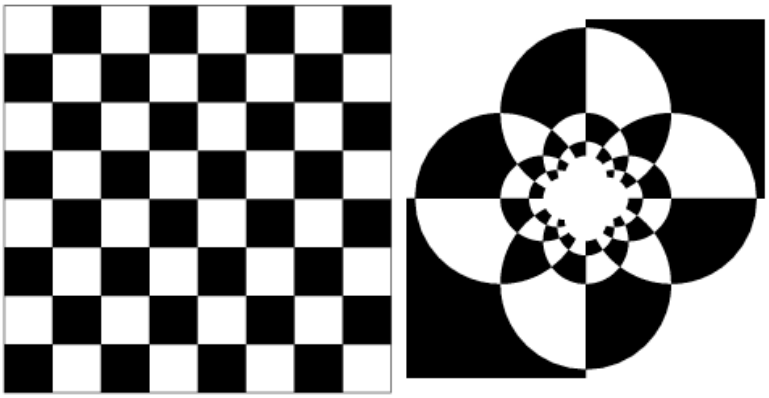
В терминах **комплексных чисел** преобразование инверсии выражается достаточно просто, если считать, что центр O окружности совпадает с началом координат:

$$z' = r^2 \cdot \frac{z}{|z|^2}.$$

С помощью сопряжённого элемента $\overline{z}$ можно получить более простую форму:

$$z' = \frac{r^2}{\overline{z}}.$$

Применение инверсии (в точке-середине доски) к изображению шахматной доски даёт интересную картинку (справа):



Свойства

Очевидно, что любая точка, лежащая **на окружности**, относительно которой производится преобразование инверсии, при отображении переходит в себя же. Любая точка, лежащая **внутри** окружности, переходит во **внешнюю** область, и наоборот. Считается, что центр окружности переходит в точку "бесконечность" ∞, а точка "бесконечность" — наоборот, в центр окружности:

$$(O)' = \infty,$$
$$(\infty)' = O.$$

Очевидно, что повторное применение преобразования инверсии **обращает** первое её применение — все точки возвращаются обратно:

$$(P')' \equiv P.$$

Обобщённые окружности

Обобщённая окружность — это либо окружность, либо прямая (считается, что это тоже окружность, но имеющая бесконечный радиус).

Ключевое свойство преобразования инверсии — что при его применении обобщённая окружность **всегда переходит в обобщённую окружность** (подразумевается, что преобразование инверсии поточечно применяется ко всем точкам фигуры).

Сейчас мы увидим, что именно происходит с прямыми и окружностями при преобразовании инверсии.

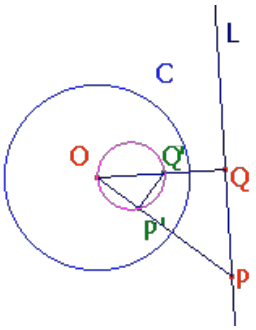
Инверсия прямой, проходящей через точку O

Утверждается, что любая прямая, проходящая через O, после преобразования инверсии **не меняется**.

В самом деле, любая точка этой прямой, кроме O и ∞, переходит по определению тоже в точку этой прямой (причём в итоге получившиеся точки заполнят всю прямую целиком, поскольку преобразование инверсии обратимо). Остаются точки O и ∞, но при инверсии они переходят друг в друга, поэтому доказательство завершено.

Инверсия прямой, не проходящей через точку O

Утверждается, что любая такая прямая перейдёт **в окружность**, проходящую через O.



Рассмотрим любую точку P этой прямой, и рассмотрим также точку Q — ближайшую к O точку прямой. Понятно, что отрезок OQ перпендикулярен прямой, а потому образуемый им угол $\angle PQO$ — прямой.

Воспользуемся теперь **леммой о равных углах**, которую мы докажем чуть позже, эта лемма даёт нам равенство:

$$\angle PQO = \angle Q'P'O.$$

Следовательно, угол $\angle Q'P'O$ тоже прямой. Поскольку мы брали точку P любой, то получается, что точка P' лежит на окружности, построенной на OQ' как на диаметре. Легко понять, что в итоге все точки прямой покроют всю эту окружность целиком, следовательно, утверждение доказано.

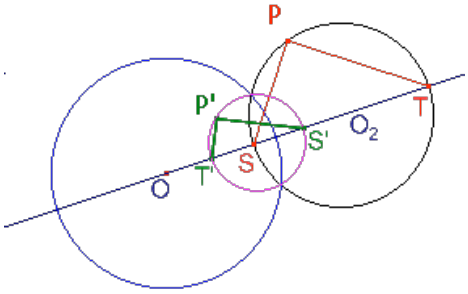
Инверсия окружности, проходящей через точку O

Любая такая окружность перейдёт **в прямую**, не проходящую через точку O.

В самом деле, это сразу следует из предыдущего пункта, если мы вспомним об обратимости преобразования инверсии.

Инверсия окружности, не проходящей через точку O

Любая такая окружность перейдёт **в окружность**, по-прежнему не проходящую через точку O.



В самом деле, рассмотрим любую такую окружность Z с центром в точке O_2. Соединим центры O и O_2 окружностей прямой; эта прямая пересечёт окружность Z в двух точках S и T (очевидно, ST — диаметр Z).

Теперь рассмотрим любую точку P, лежащую на окружности Z. Угол $\angle SPT$ прямой для любой такой точки, но по следствию из **леммы о равных углах** также прямым должен быть и угол $\angle S'P'T'$, откуда и следует, что точка P' лежит на окружности, построенной на отрезке $S'T'$ как на диаметре. Опять же, легко понять, что все образы P' в итоге покроют эту окружность.

Понятно, что эта новая окружность не может проходить через O: иначе бы точка ∞ должна была бы принадлежать старой окружности.

Лемма о равных углах

Это вспомогательное свойство, которое было использовано выше при анализе результатов преобразования инверсии.

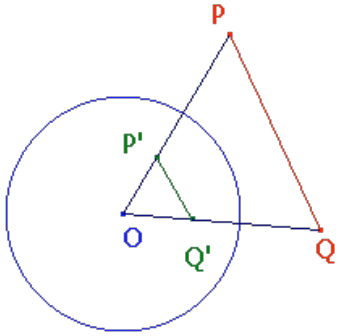
Формулировка

Рассмотрим любые две точки P и Q, и применим к ним преобразование инверсии, получим точки P' и Q'.
Тогда следующие углы равны:

$$\angle PQO = \angle Q'P'O,$$
$$\angle QPO = \angle P'Q'O.$$

Доказательство

Докажем, что треугольники $\triangle PQO$ и $\triangle Q'P'O$ подобны (порядок вершин важен!).



В самом деле, по определению преобразования инверсии имеем:

$$OP \cdot OP' = r^2,$$
$$OQ \cdot OQ' = r^2,$$

откуда получаем равенство:

$$OP \cdot OP' = OQ \cdot OQ',$$
$$\frac{OP}{OQ} = \frac{OQ'}{OP'}.$$

Таким образом, треугольники $\triangle PQO$ и $\triangle Q'P'O$ имеют общий угол, а две прилежащие к нему стороны пропорциональны, следовательно, эти треугольники подобны, а потому соответствующие углы совпадают.

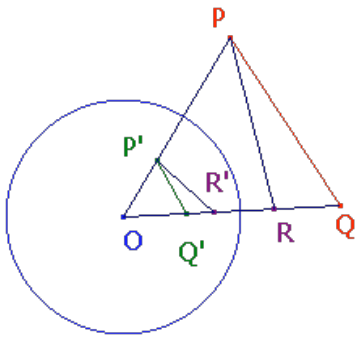
Следствие из леммы

Если даны любые три точки P, Q, R, причём точка R лежит на отрезке OQ, то выполняется:

$$\angle QPR = \angle Q'P'R',$$

причём эти углы ориентированы в разные стороны (т.е. если рассматривать эти два угла как ориентированные, то

они разного знака).



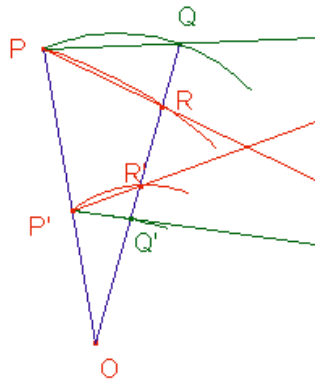
Для доказательства заметим, что $\angle QPR$ — это разность двух углов $\angle QPO$ и $\angle RPO$, к каждому из которых можно применить лемму о равных углах:

$$\angle QPR = \angle QPO - \angle RPO = \angle P'Q'O - \angle P'R'O = \angle R'P'Q' = \angle Q'P'R'.$$

При осуществлении последнего перехода мы изменили порядок следования точек, что и означает, что мы изменили ориентацию угла на противоположную.

Конформность

Преобразование инверсии является конформным, т.е. **сохраняет углы в точках пересечения кривых**. При этом, если углы рассматривать как ориентированные, то ориентация углов при применении инверсии изменяется на противоположную.



Для **доказательства** этого рассмотрим две произвольные кривые, пересекающиеся в точке P и имеющие в ней касательные. Пусть по первой кривой будет идти точка Q, по второй — точка R (мы их устремим в пределе к P).

Очевидно, что после применения инверсии кривые будут по-прежнему пересекаться (если, конечно, они не проходили через точку O, но такой случай мы не рассматриваем), и точкой их пересечения будет P'.

Учитывая, что точка R лежит на прямой, соединяющей O и Q, получаем, что можем применить следствие из леммы о равных углах, из которой мы получаем:

$$\angle QPR = -\angle Q'P'R',$$

где под знаком "минус" мы понимаем то, что углы ориентированы в разных направлениях.

Устремляя точки Q и R к точке P, мы в пределе получаем, что это равенство — выражение угла между пересекающимися кривыми, что и требовалось доказать.

Свойство отражения

Если M — обобщённая окружность, то при преобразовании инверсии она **сохраняется** тогда и только тогда, когда M **ортогональна** окружности C, относительно которой производится инверсия (M и C считаются различными).

Доказательство этого свойства интересно тем, что оно **демонстрирует** применение геометрической инверсии для ухода от окружностей и упрощения задачи.

Первым шагом **доказательства** будет указание того факта, что M и C имеют как минимум две точки пересечения. В самом деле, преобразование инверсии относительно C отображает внутренность окружности в её внешность, и наоборот. Раз M после преобразования не изменилась, то значит, она содержит точки как

из внутренности, так и из внешности окружности C. Отсюда и следует, что точек пересечения две (одна она быть не может — это означает касание двух окружностей, но этого случая, очевидно, быть по условию не может; совпадать окружности также не могут по определению).

Обозначим одну точку пересечения через A, другую — через B. Рассмотрим произвольную окружность с центром в точке A, и выполним преобразование инверсии относительно неё. Заметим, что тогда и окружность C, и обобщённая окружность M необходимо переходят в пересекающиеся прямые. Учитывая конформность преобразования инверсии, получаем, что M и M' совпадали тогда и только тогда, когда угол между двумя этими пересекающимися прямыми прямой (в самом деле, первое преобразование инверсии, — относительно C, — изменяет направление угла между окружностями на противоположное, поэтому если окружность совпадает со своей инверсией, то углы между пересекающимися прямыми с обеих сторон должны совпадать, и равны $\frac{180}{2} = 90$ градусов).

Практическое применение

Сразу стоит отметить, что при применении в расчётах нужно учитывать большую **погрешность**, вносимую преобразованием инверсии: могут появляться дробные числа весьма малых порядков, и обычно из-за высокой погрешности метод инверсии хорошо работает только со сравнительно небольшими координатами.

Построение фигур после инверсии

В программных вычислениях зачастую более удобно и надёжно использовать не готовые формулы для координат и радиусов получающихся обобщённых окружностей, а восстанавливать каждый раз прямые/окружности по двумя точкам. Если для восстановления прямой достаточно взять любые две точки и вычислить их образы и соединить прямой, то с окружностями всё гораздо сложнее.

Если мы хотим найти окружность, получившуюся в результате инверсии прямой, то, согласно приведённым выше выкладкам, надо найти ближайшую к центру инверсии точку Q прямой, применить к ней инверсию (получив некую точку Q'), и тогда искомая окружность будет иметь диаметр OQ'.

Пусть теперь мы хотим найти окружность, получившуюся в результате инверсии другой окружности. Вообще говоря, центр новой окружности — не совпадает с образом центра старой окружности. Для определения центра новой окружности можно воспользоваться таким приёмом: провести через центр инверсии и центр старой окружности прямую, посмотреть её точки пересечения со старой окружностью, — пусть это будут точки S и T. Отрезок ST образует диаметр старой окружности, и легко понять, что после инверсии этот отрезок по-прежнему будет образовывать диаметр. Следовательно, центр новой окружности можно найти как среднее арифметическое точек S' и T'.

Параметры окружности после инверсии

Требуется по заданной окружности (по известным координатам её центра (x_0, y_0) и радиусу r_0) определить, в какую именно окружность она перейдёт после преобразования инверсии относительно окружности с центром в (x_c, y_c) и радиусом r.

Т.е. мы решаем задачу, описанную в предыдущем пункте, но хотим получить решение в аналитическом виде.

Ответ выглядит в виде формул:

$$\begin{aligned} x' &= x_c + s(x_0 - x_c), \\ y' &= y_c + s(y_0 - y_c), \\ r' &= |s| \cdot r_0, \end{aligned}$$

где

$$s = \frac{r^2}{(x_0 - x_c)^2 + (y_0 - y_c)^2 - r_0^2}.$$

Мнемонически эти формулы можно запомнить так: центр окружности переходит "почти" как по преобразованию инверсии, только в знаменателе помимо $|z|^2 = (x_0 - x_c)^2 + (y_0 - y_c)^2$ появилось ещё вычитаемое r_0^2.

Выводятся эти формулы ровно по описанному в предыдущем пункте алгоритму: находятся выражения для двух диаметральных точек S и T, затем к ним применяется инверсия, и затем берётся среднее арифметическое от их координат. Аналогично можно посчитать и радиус как половину длины отрезка ST.

Применение в доказательствах: задача о разбиении точек окружностью

Даны $2n$ различных точек на плоскости, а также произвольная точка O, отличная от всех остальных. Доказать, что найдётся окружность, проходящая через точку O, такая, что внутри и вне неё будет лежать одинаковое число

точек набора, т.е. по n штук.

Для **доказательства** произведём преобразование инверсии относительно выбранной точки O (с любым радиусом, например, $r = 1$). Тогда искомой окружности будет соответствовать прямая, не проходящая через точку O. Причём по одну сторону прямой лежит полуплоскость, соответствующая внутренности окружности, а по другую — соответствующая внешности. Понятно, что всегда найдётся такая прямая, которая разбивает множество из $2n$ точек на две половины по n точек, и при этом не проходит через точку O (например, такую прямую можно получить, повернув всю картину на любой такой угол, чтобы ни у каких из рассматриваемых $2n + 1$ точек не совпали координаты x, а затем просто взяв вертикальную прямую между n-ой и $n + 1$-ой точками). Эта прямая соответствует искомой окружности, проходящей через точку O, а значит, утверждение доказано.

Применение при решении задач вычислительной геометрии

Замечательное свойство геометрической инверсии — в том, что во многих случаях она позволяет упростить поставленную геометрическую задачу, заменяя рассмотрение окружностей только рассмотрением прямых.

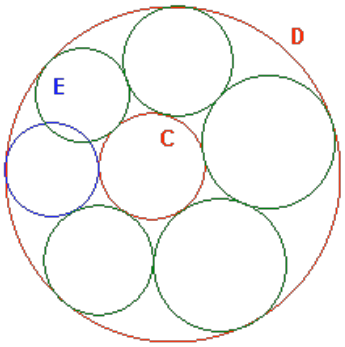
Т.е. если задача имеет достаточно сложный вид различных операций с окружностями, то имеет смысл применить ко входным данным преобразование инверсии, попытаться решить полученную модифицированную задачу без окружностей (или с меньшим их числом), и затем повторным применением инверсии получить решение исходной задачи.

Пример такой задачи описан в следующем разделе.

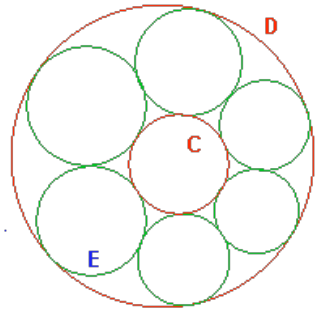
Цепочки Штейнера

Даны две окружности C и D, одна находится строго внутри другой. Затем рисуется третья окружность E, касающаяся этих двух окружностей, после чего запускается итеративный процесс: каждый раз рисуется новая окружность так, чтобы она **касалась** предыдущей нарисованной, и первых двух. Рано или поздно очередная нарисованная окружность пересечётся с какой-то из уже поставленных, или по крайней мере коснётся её.

Случай пересечения:



Случай касания:



Соответственно, наша задача — поставить **как можно больше** окружностей так, чтобы пересечения (т.е. первого из представленных случаев) не было. Первые две окружности (внешняя и внутренняя) фиксированы, мы можем лишь варьировать положение первой касающейся окружности, дальше все касающиеся окружности ставятся однозначно.

В случае касания получающая цепочка окружностей называется **цепочкой Штейнера**.

С этой цепочкой связано так называемое **утверждение Штейнера** (Steiner's porism): если существует хотя бы одна цепочка Штейнера (т.е. существует соответствующее положение стартовой касающейся окружности, приводящее к цепочке Штейнера), то при любом другом выборе стартовой касающейся окружности также будет получаться цепочка Штейнера, причём число окружностей в ней будет таким же.

Из этого утверждения следует, что и при решении задачи максимизации числа окружностей ответ не зависит от позиции первой поставленной окружности.

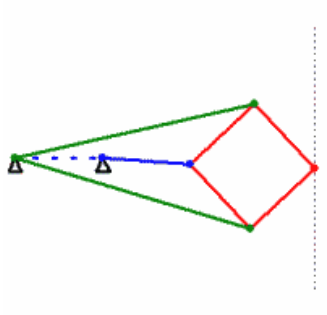
Доказательство и конструктивный алгоритм решения следующие. Заметим, что задача имеет очень

простое решение в случае, когда центры внешней и внутренней окружностей совпадают. Понятно, что в этом случае число поставленных окружностей никак не будет зависеть от первой поставленной. В этом случае все окружности имеют одинаковый радиус, и число их и координаты центров можно посчитать по простым формулам.

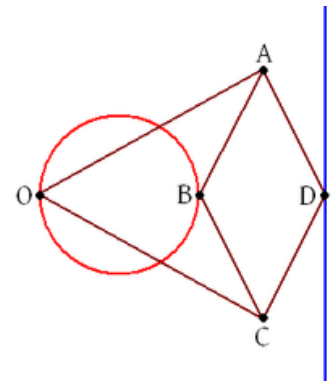
Чтобы перейти к этой простой ситуации из любой подаваемой на вход, применим преобразование инверсии относительно некоторой окружности. Нам нужно, чтобы центр внутренней окружности передвинулся и совпал с центром внешней, поэтому искать точку, относительно которой будем брать инверсию, надо только на прямой, соединяющей центры окружностей. Используя формулы для координат центра окружности после применения инверсии, можно составить уравнение на положение центра инверсии, и решить это уравнение. Тем самым мы от произвольной ситуации можем перейти к простому, симметрическому случаю, а, решив задачу для него, повторно применим преобразование инверсии и получим решение исходной задачи.

Применение в технике: прямило Липкина-Поселье

Долгое время задача преобразования кругового (вращательного) движения в прямолинейное оставалась весьма сложной в машиностроении, удавалось находить в лучшем случае приближённые решения. И лишь в 1864 г. офицер инженерного корпуса французской армии Шарль Никола Поселье (Charles-Nicolas Peaucellier) и в 1868 г. студент Чебышёва Липман Липкин (Lipman Lipkin) изобрели это устройство, основанное на идее геометрической инверсии. Устройство получило название "**прямило Липкина-Поселье**" (Peaucellier–Lipkin linkage).



Чтобы понять работу устройства, отметим на нём несколько точек:



Точка B совершает вращательное движение по окружности (красного цвета), в результате чего точка D необходимо движется по прямой (синего цвета). Наша задача — доказать, что точка D — суть инверсия точки B относительно центра O с некоторым радиусом r.

Формализуем условие задачи: что точка O жёстко закреплена, отрезки OA и OC совпадают, и также совпадает четвёрка отрезков AB, BC, CD, DA. Точка B движется вдоль окружности, проходящей через точку O.

Для **доказательства** заметим вначале, что точки O, B и D лежат на одной прямой (это следует из равенства треугольников). Обозначим через P точку пересечения отрезков AC и BD. Введём обозначения:

$$OB = x, \quad BP = y, \quad AP = h.$$

Нам нужно показать, что величина $OB \cdot OD = \text{const}$:

$$OB \cdot OD = x(x + 2y) = x^2 + 2xy.$$

По теореме Пифагора получаем:

$$OA^2 = (x + y)^2 + h^2,$$
$$AD^2 = y^2 + h^2.$$

Возьмём разность этих двух величин:

$$OA^2 - AD^2 = x^2 + 2xy = OB \cdot OD.$$

Таким образом, мы доказали, что $OB \cdot OD = \text{const}$, что и означает, что D — инверсия точки B.

Поиск общих касательных к двум окружностям

Даны две окружности. Требуется найти все их общие касательные, т.е. все такие прямые, которые касаются обеих окружностей одновременно.

Описанный алгоритм будет работать также в случае, когда одна (или обе) окружности вырождаются в точки. Таким образом, этот алгоритм можно использовать также для нахождения касательных к окружности, проходящих через заданную точку.

Количество общих касательных

Сразу отметим, что мы не рассматриваем **вырожденные** случаи: когда окружности совпадают (в этом случае у них бесконечно много общих касательных), или одна окружность лежит внутри другой (в этом случае у них нет общих касательных, или, если окружности касаются, есть одна общая касательная).

В большинстве случаев, две окружности имеют **четыре** общих касательных.

Если окружности **касаются**, то у них будет три обших касательных, но это можно понимать как вырожденный случай: так, как будто две касательные совпали.

Более того, описанный ниже алгоритм будет работать и в случае, когда одна или обе окружности имеют нулевой радиус: в этом случае будет, соответственно, две или одна общая касательная.

Подводя итог, мы, за исключением описанных в начале случаев, всегда будем искать **четыре касательные**. В вырожденных случаях некоторые из них будут совпадать, однако тем не менее эти случаи также будут вписываться в общую картину.

Алгоритм

В целях простоты алгоритма, будем считать, не теряя общности, что центр первой окружности имеет координаты $(0;0)$. (Если это не так, то этого можно добиться простым сдвигом всей картины, а после нахождения решения — сдвигом полученных прямых обратно.)

Обозначим через r_1 и r_2 радиусы первой и второй окружностей, а через v — координаты центра второй окружности (точка v отлична от начала координат, т.к. мы не рассматриваем случае, когда окружности совпадают, или одна окружность находится внутри другой).

Для решения задачи подойдём к ней чисто **алгебраически**. Нам требуется найти все прямые вида $ax + by + c = 0$, которые лежат на расстоянии r_1 от начала координат, и на расстоянии r_2 от точки v. Кроме того, наложим условие нормированности прямой: сумма квадратов коэффициентов a и b должна быть равна единице (это необходимо, иначе одной и той же прямой будет соответствовать бесконечно много представлений вида $ax + by + c = 0$). Итого получаем такую систему уравнений на искомые a, b, c:

$$\begin{cases} a^2 + b^2 = 1, \\ |a \cdot 0 + b \cdot 0 + c| = r_1, \\ |a \cdot v_x + b \cdot v_y + c| = r_2. \end{cases}$$

Чтобы избавиться от модулей, заметим, что всего есть четыре способа раскрыть модули в этой системе. Все эти способы можно рассмотреть общим случаем, если понимать раскрытие модуля как то, что коэффициент в правой части, возможно, умножается на -1.

Иными словами, мы переходим к такой системе:

$$\begin{cases} a^2 + b^2 = 1, \\ c = \pm r_1, \\ a \cdot v_x + b \cdot v_y + c = \pm r_2. \end{cases}$$

Введя обозначения $d_1 = \pm r_1$ и $d_2 = \pm r_2$, мы приходим к тому, что четыре раза должны решать систему:

$$\begin{cases} a^2 + b^2 = 1, \\ c = d_1, \\ a \cdot v_x + b \cdot v_y + c = d_2. \end{cases}$$

Решение этой системы сводится к решению квадратного уравнения. Мы опустим все громоздкие выкладки, и сразу приведём готовый ответ:

$$\begin{cases} a = \frac{(d_2-d_1)v_x \pm v_y \sqrt{v_x^2+v_y^2-(d_2-d_1)^2}}{v_x^2+v_y^2}, \\ b = \frac{(d_2-d_1)v_y \mp v_x \sqrt{v_x^2+v_y^2-(d_2-d_1)^2}}{v_x^2+v_y^2}, \\ c = d_1. \end{cases}$$

Итого у нас получилось 8 решений вместо 4. Однако легко понять, в каком месте возникают лишние решения: на самом деле, в последней системе достаточно брать только одно решение (например, первое). В самом деле, геометрический смысл того, что мы берём $\pm r_1$ и $\pm r_2$, понятен: мы фактически перебираем, по какую сторону от каждой из окружностей будет прямая. Поэтому два способа, возникающие при решении последней системы, избыточны: достаточно выбрать одно из двух решений (только, конечно, во всех четырёх случаях надо выбрать одно и то же семейство решений).

Последнее, что мы ещё не рассмотрели — это **как сдвигать прямые** в том случае, когда первая окружность не находилась изначально в начале координат. Однако здесь всё просто: из линейности уравнения прямой следует, что от коэффициента c надо отнять величину $a \cdot x_0 + b \cdot y_0$ (где x_0 и y_0 — координаты первоначального центра первой окружности).

Реализация

Опишем сначала все необходимые структуры данных и другие вспомогательные определения:

```
struct pt {
        double x, y;

        pt operator- (pt p) {
                pt res = { x-p.x, y-p.y };
                return res;
        }
};

struct circle : pt {
        double r;
};

struct line {
        double a, b, c;
};

const double EPS = 1E-9;

double sqr (double a) {
        return a * a;
}
```

Тогда само решение можно записать таким образом (где основная функция для вызова — вторая; а первая функция — вспомогательная):

```
void tangents (pt c, double r1, double r2, vector<line> & ans) {
        double r = r2 - r1;
        double z = sqr(c.x) + sqr(c.y);
        double d = z - sqr(r);
        if (d < -EPS)  return;
        d = sqrt (abs (d));
        line l;
        l.a = (c.x * r + c.y * d) / z;
        l.b = (c.y * r - c.x * d) / z;
        l.c = r1;
        ans.push_back (l);
}

vector<line> tangents (circle a, circle b) {
        vector<line> ans;
        for (int i=-1; i<=1; i+=2)
```

```
                for (int j=-1; j<=1; j+=2)
                        tangents (b-a, a.r*i, b.r*j, ans);
        for (size_t i=0; i<ans.size(); ++i)
                ans[i].c -= ans[i].a * a.x + ans[i].b * a.y;
        return ans;
}
```

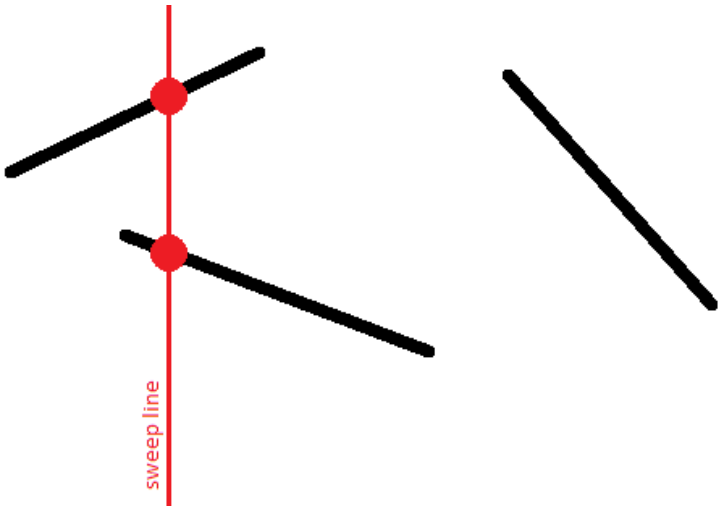
Поиск пары пересекающихся отрезков алгоритмом заметающей прямой за O (N log N)

Даны n отрезков на плоскости. Требуется проверить, пересекаются ли друг с другом хотя бы два из них. (Если ответ положителен — то вывести эту пару пересекающихся отрезков; среди нескольких ответов достаточно выбрать любой из них.)

Наивный алгоритм решения — перебрать за $O(n^2)$ все пары отрезков и проверить для каждой пары, пересекаются они или нет. В данной статье описывается алгоритм с временем работы $O(n \log n)$, который основан на принципе **сканирующей (заметающей) прямой** (по-английски: "sweep line").

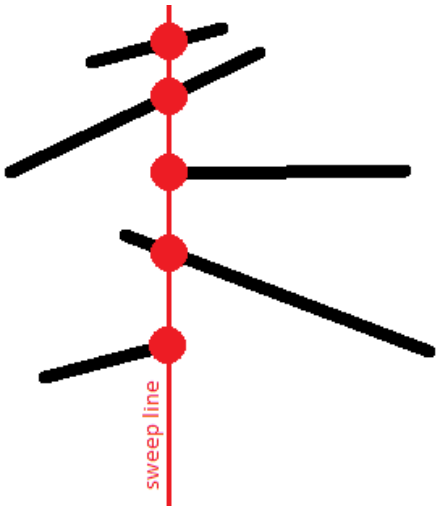
Алгоритм

Проведём мысленно вертикальную прямую $x = -\infty$ и начнём двигать эту прямую вправо. По ходу своего движения эта прямая будет встречаться с отрезками, причём в любой момент времени каждый отрезок будет пересекаться с нашей прямой по одной точке (мы пока будем считать, что вертикальных отрезков нет).

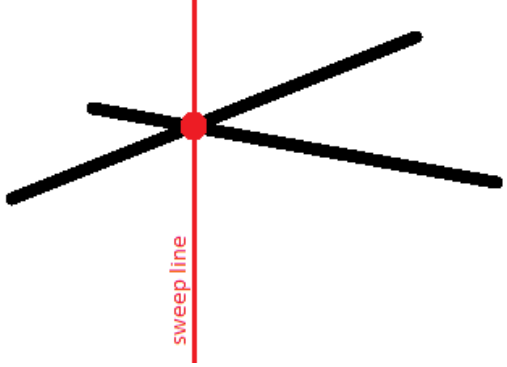


Таким образом, для каждого отрезка в какой-то момент времени его точка появится на сканирующей прямой, затем с движением прямой будет двигаться и эта точка, и, наконец, в какой-то момент отрезок исчезнет с прямой.

Нас интересует **относительный порядок отрезков** по вертикали. А именно, мы будем хранить список отрезков, пересекающих сканирующую прямую в данный момент времени, где отрезки будут отсортированы по их y-координате на сканирующей прямой.



Этот порядок интересен тем, что пересекающиеся отрезки будут иметь одинаковую y-координату хотя бы в один момент времени:



Сформулируем ключевые утверждения:

- Для поиска пересекающейся пары достаточно рассматривать при каждом фиксированном положении сканирующей прямой **только соседние отрезки**.
- Достаточно рассматривать сканирующую прямую не во всех возможных действительных позициях $(-\infty \ldots +\infty)$, а **только в тех позициях**, **когда появляются новые отрезки или исчезают старые**. Иными словами, достаточно ограничиться лишь только положениями, равными абсциссам точек-концов отрезков.
- При появлении нового отрезка достаточно **вставить** его в нужное место в список, полученный для предыдущей сканирующей прямой. Проверять на пересечение надо **только добавляемый отрезок с его непосредственными соседями в списке сверху и снизу**.
- При исчезновении отрезка достаточно **удалить** его из текущего списка. После этого надо **проверить на пересечение с верхним и нижним соседями** в списке.
- Других изменений в порядке следования отрезков в списке, кроме описанных, не существует. Других проверок на пересечения производить не надо.

Для понимания истинности этих утверждений достаточно следующих замечаний:

- Два непересекающихся отрезка никогда не меняют своего **относительного порядка**.

  В самом деле, если один отрезок сначала был выше другого, а затем стал ниже, то между двумя этими моментами произошло пересечение этих двух отрезков.
- Иметь совпадающие y-координаты два непересекающихся отрезка также не могут.
- Из этого следует, что в момент появления отрезка мы можем найти в очереди позицию для этого отрезка, и больше этот отрезок переставлять в очереди не придётся: его **порядок относительно других отрезков в очереди меняться не будет**.
- Два пересекающихся отрезка в момент точки своего пересечения окажутся **соседями** друг друга в очереди.
- Следовательно, для нахождения пары пересекающихся отрезков достаточно проверить на пересечение только все те пары отрезков, которые когда-нибудь за время движения сканирующей прямой хотя бы раз **были соседями друг друга**.

  Легко заметить, что этого достаточно лишь проверять добавляемый отрезок со своими верхним и нижним соседями, а также при удалении отрезка — его верхнего и нижнего соседей (которые после удаления станут соседями друг друга).
- Следует обратить внимание, что при фиксированном положении сканирующей прямой мы **сначала** должны произвести **добавление** всех появляющихся здесь отрезков, и лишь **затем** — **удаление** всех исчезающих здесь отрезков.

  Тем самым, мы не пропустим пересечения отрезков по вершине: т.е. такие случаи, когда два отрезка имеют общую вершину.
- Заметим, что **вертикальные отрезки** на самом деле никак не влияют на корректность алгоритма.

  Эти отрезки выделяются тем, что они появляются и исчезают в один и тот же момент времени. Однако, за счёт предыдущего замечания, мы знаем, что сначала все отрезки будут добавлены в очередь, и лишь затем будут удалены. Следовательно, если вертикальный отрезок пересекается с каким-то другим открытым в этот момент отрезком (в том числе вертикальным), то это будет обнаружено.

  В какое место очереди помещать вертикальные отрезки? Ведь вертикальный отрезок не имеет одной определённой y-координаты, он простирается на целый отрезок по y-координате. Однако легко понять, что в качестве y-координаты можно взять любую координату из этого отрезка.

Таким образом, весь алгоритм совершит не более $2n$ тестов на пересечение пары отрезков, и совершит $O(n)$ операций с очередью отрезков (по $O(1)$ операций в моменты появления и исчезновения каждого отрезка).

Итоговая **асимптотика** алгоритма составляет, таким образом, $O(n \log n)$.

Реализация

Приведём полную реализацию описанного алгоритма:

```
const double EPS = 1E-9;

struct pt {
        double x, y;
};

struct seg {
        pt p, q;
        int id;

        double get_y (double x) const {
                if (abs (p.x - q.x) < EPS)  return p.y;
                return p.y + (q.y - p.y) * (x - p.x) / (q.x - p.x);
        }
};


inline bool intersect1d (double l1, double r1, double l2, double r2) {
        if (l1 > r1)  swap (l1, r1);
        if (l2 > r2)  swap (l2, r2);
        return max (l1, l2) <= min (r1, r2) + EPS;
}

inline int vec (const pt & a, const pt & b, const pt & c) {
        double s = (b.x - a.x) * (c.y - a.y) - (b.y - a.y) * (c.x - a.x);
        return abs(s)<EPS ? 0 : s>0 ? +1 : -1;
}

bool intersect (const seg & a, const seg & b) {
        return intersect1d (a.p.x, a.q.x, b.p.x, b.q.x)
                && intersect1d (a.p.y, a.q.y, b.p.y, b.q.y)
                && vec (a.p, a.q, b.p) * vec (a.p, a.q, b.q) <= 0
                && vec (b.p, b.q, a.p) * vec (b.p, b.q, a.q) <= 0;
}


bool operator< (const seg & a, const seg & b) {
        double x = max (min (a.p.x, a.q.x), min (b.p.x, b.q.x));
        return a.get_y(x) < b.get_y(x) - EPS;
}


struct event {
        double x;
        int tp, id;

        event() { }
        event (double x, int tp, int id)
                : x(x), tp(tp), id(id)
        { }

        bool operator< (const event & e) const {
                if (abs (x - e.x) > EPS)  return x < e.x;
                return tp > e.tp;
        }
};

set<seg> s;
vector < set<seg>::iterator > where;

inline set<seg>::iterator prev (set<seg>::iterator it) {
        return it == s.begin() ? s.end() : --it;
}

inline set<seg>::iterator next (set<seg>::iterator it) {
        return ++it;
```

```
}

pair<int,int> solve (const vector<seg> & a) {
        int n = (int) a.size();
        vector<event> e;
        for (int i=0; i<n; ++i) {
                e.push_back (event (min (a[i].p.x, a[i].q.x), +1, i));
                e.push_back (event (max (a[i].p.x, a[i].q.x), -1, i));
        }
        sort (e.begin(), e.end());

        s.clear();
        where.resize (a.size());
        for (size_t i=0; i<e.size(); ++i) {
                int id = e[i].id;
                if (e[i].tp == +1) {
                        set<seg>::iterator
                                nxt = s.lower_bound (a[id]),
                                prv = prev (nxt);
                        if (nxt != s.end() && intersect (*nxt, a[id]))
                                return make_pair (nxt->id, id);
                        if (prv != s.end() && intersect (*prv, a[id]))
                                return make_pair (prv->id, id);
                        where[id] = s.insert (nxt, a[id]);
                }
                else {
                        set<seg>::iterator
                                nxt = next (where[id]),
                                prv = prev (where[id]);
                        if (nxt != s.end() && prv != s.end() &&
intersect (*nxt, *prv))
                                return make_pair (prv->id, nxt->id);
                        s.erase (where[id]);
                }
        }

        return make_pair (-1, -1);
}
```

Основная функция здесь — $\mathrm{solve}()$, которая возвращает номера найденных пересекающихся отрезков, либо $(-1, -1)$, если пересечения отсутствуют.

Проверка на пересечение двух отрезков осуществляется функцией $\mathrm{intersect}()$, с помощью алгоритма на основе ориентированной площади треугольника.

Очередь отрезков в глобальной переменной s — $\mathrm{set} < \mathrm{event} >$. Итераторы, указывающие положение каждого отрезка в очереди (для удобного удаления отрезков из очереди), хранятся в глобальном массиве where.

Введены также две вспомогательные функции $\mathrm{prev}()$ и $\mathrm{next}()$, которые возвращают итераторы на предыдущий и следующий элементы (либо $\mathrm{end}()$, если такового не существует).

Константа EPS обозначает погрешность сравнения двух вещественных чисел (в основном она используется при проверке двух отрезков на пересечение).

Z-функция строки и её вычисление

Пусть дана строка s длины n. Тогда Z-**функция** ("зет-функция") от этой строки — это массив длины n, i-ый элемент которого равен наибольшему числу символов, начиная с позиции i, совпадающих с первыми символами строки s.

Иными словами, $z[i]$ — это наибольший общий префикс строки s и её i-го суффикса.

Примечание. В данной статье, во избежание неопределённости, мы будем считать строку 0-индексированной — т. е. первый символ строки имеет индекс 0, а последний — $n-1$.

Первый элемент Z-функции, $z[0]$, обычно считают неопределённым. В данной статье мы будем считать, что он равен нулю (хотя ни в алгоритме, ни в приведённой реализации это ничего не меняет).

В данной статье приводится алгоритм вычисления Z-функции за время $O(n)$, а также различные применения этого алгоритма.

Примеры

Приведём для примера подсчитанную Z-функцию для нескольких строк:

- $"aaaaa"$:

$$\begin{aligned} z[0] &= 0, \\ z[1] &= 4, \\ z[2] &= 3, \\ z[3] &= 2, \\ z[4] &= 1. \end{aligned}$$

- $"aaabaab"$:

$$\begin{aligned} z[0] &= 0, \\ z[1] &= 2, \\ z[2] &= 1, \\ z[3] &= 0, \\ z[4] &= 2, \\ z[5] &= 1, \\ z[6] &= 0. \end{aligned}$$

- $"abacaba"$:

$$\begin{aligned} z[0] &= 0, \\ z[1] &= 0, \\ z[2] &= 1, \\ z[3] &= 0, \\ z[4] &= 3, \\ z[5] &= 0, \\ z[6] &= 1. \end{aligned}$$

Тривиальный алгоритм

Формальное определение можно представить в виде следующей элементарной реализации за $O(n^2)$:

```
vector<int> z_function_trivial (string s) {
        int n = (int) s.length();
        vector<int> z (n);
        for (int i=1; i<n; ++i)
                while (i + z[i] < n && s[z[i]] == s[i+z[i]])
                        ++z[i];
        return z;
}
```

Мы просто для каждой позиции i перебираем ответ для неё $z[i]$, начиная с нуля, и до тех пор, пока мы не

обнаружим несовпадение или не дойдём до конца строки.

Разумеется, эта реализация слишком неэффективна, перейдём теперь к построению эффективного алгоритма.

Эффективный алгоритм вычисления Z-функции

Чтобы получить эффективный алгоритм, будем вычислять значения $z[i]$ по очереди — от $i = 1$ до $n - 1$, и при этом постараемся при вычислении очередного значения $z[i]$ максимально использовать уже вычисленные значения.

Назовём для краткости подстроку, совпадающую с префиксом строки s, **отрезком совпадения**. Например, значение искомой Z-функции $z[i]$ — это длиннейший отрезок совпадения, начинающийся в позиции i (и заканчиваться он будет в позиции $i + z[i] - 1$).

Для этого будем поддерживать **координаты** $[l; r]$ **самого правого отрезка совпадения**, т.е. из всех обнаруженных отрезков будем хранить тот, который оканчивается правее всего. В некотором смысле, индекс r — это такая граница, до которой наша строка уже была просканирована алгоритмом, а всё остальное — пока ещё не известно.

Тогда если текущий индекс, для которого мы хотим посчитать очередное значение Z-функции, — это i, мы имеем один из двух вариантов:

- $i > r$ — т.е. текущая позиция лежит **за пределами** того, что мы уже успели обработать.

  Тогда будем искать $z[i]$ **тривиальным алгоритмом**, т.е. просто пробуя значения $z[i] = 0$, $z[i] = 1$, и т. д. Заметим, что в итоге, если $z[i]$ окажется > 0, то мы будем обязаны обновить координаты самого правого отрезка $[l; r]$ — т.к. $i + z[i] - 1$ гарантированно окажется больше r.

- $i \le r$ — т.е. текущая позиция лежит внутри отрезка совпадения $[l; r]$.

  Тогда мы можем использовать уже подсчитанные **предыдущие** значения Z-функции, чтобы проинициализировать значение $z[i]$ не нулём, а каким-то возможно бОльшим числом.

  Для этого заметим, что подстроки $s[l \dots r]$ и $s[0 \dots r - l]$ **совпадают**. Это означает, что в качестве начального приближения для $z[i]$ можно взять соответствующее ему значение из отрезка $s[0 \dots r - l]$, а именно, значение $z[i - l]$.

  Однако значение $z[i - l]$ могло оказаться слишком большим: таким, что при применении его к позиции i оно "вылезет" за пределы границы r. Этого допустить нельзя, т.к. про символы правее r мы ничего не знаем, и они могут отличаться от требуемых.

  Приведём **пример** такой ситуации, на примере строки:

  $$"aaaabaa"$$

  Когда мы дойдём до последней позиции ($i = 6$), текущим самым правым отрезком будет $[5; 6]$. Позиции 6 с учётом этого отрезка будет соответствовать позиция $6 - 5 = 1$, ответ в которой равен $z[1] = 3$. Очевидно, что таким значением инициализировать $z[6]$ нельзя, оно совершенно некорректно. Максимум, каким значением мы могли проинициализировать — это 1, поскольку это наибольшее значение, которое не вылазит за пределы отрезка $[l; r]$.

  Таким образом, в качестве **начального приближения** для $z[i]$ безопасно брать только такое выражение:

  $$z_0[i] = \min(r - i + 1, z[i - l]).$$

  Проинициализировав $z[i]$ таким значением $z_0[i]$, мы снова дальше действуем **тривиальным алгоритмом** — потому что после границы r, вообще говоря, могло обнаружиться продолжение отрезка совпадение, предугадать которое одними лишь предыдущими значениями Z-функции мы не могли.

Таким образом, весь алгоритм представляет из себя два случая, которые фактически различаются только **начальным значением** $z[i]$: в первом случае оно полагается равным нулю, а во втором — определяется по предыдущим значениям по указанной формуле. После этого обе ветки алгоритма сводятся к выполнению **тривиального алгоритма**, стартующего сразу с указанного начального значения.

Алгоритм получился весьма простым. Несмотря на то, что при каждом i в нём так или иначе выполняется тривиальный алгоритм — мы достигли существенного прогресса, получив алгоритм, работающий за линейное время. Почему это так, рассмотрим ниже, после того, как приведём реализацию алгоритма.

Реализация

Реализация получается весьма лаконичной:

```
vector<int> z_function (string s) {
```

```
        int n = (int) s.length();
        vector<int> z (n);
        for (int i=1, l=0, r=0; i<n; ++i) {
                if (i <= r)
                        z[i] = min (r-i+1, z[i-l]);
                while (i+z[i] < n && s[z[i]] == s[i+z[i]])
                        ++z[i];
                if (i+z[i]-1 > r)
                        l = i,  r = i+z[i]-1;
        }
        return z;
}
```

Прокомментируем эту реализацию.

Всё решение оформлено в виде функции, которая по строке возвращает массив длины n — вычисленную Z-функцию.

Массив $z[]$ изначально заполняется нулями. Текущий самый правый отрезок совпадения полагается равным $[0;0]$, т. е. заведомо маленький отрезок, в который не попадёт ни одно i.

Внутри цикла по $i = 1 \dots n-1$ мы сначала по описанному выше алгоритму определяем начальное значение $z[i]$ — оно либо останется нулём, либо вычислится на основе приведённой формулы.

После этого выполняется тривиальный алгоритм, который пытается увеличить значение $z[i]$ настолько, насколько это возможно.

В конце выполняется обновление текущего самого правого отрезка совпадения $[l;r]$, если, конечно, это обновление требуется — т.е. если $i + z[i] - 1 > r$.

Асимптотика алгоритма

Докажем, что приведённый выше алгоритм работает за линейное относительно длины строки время, т.е. за $O(n)$.

Доказательство очень простое.

Нас интересует вложенный цикл while — т.к. всё остальное — лишь константные операции, выполняемые $O(n)$ раз.

Покажем, что **каждая итерация** этого цикла while приведёт к увеличению правой границы r на единицу.

Для этого рассмотрим обе ветки алгоритма:

- $i > r$

  В этом случае либо цикл while не сделает ни одной итерации (если $s[0] \neq s[i]$), либо же сделает несколько итераций, продвигаясь каждый раз на один символ вправо, начиная с позиции i, а после этого — правая граница r обязательно обновится.

  Поскольку $i > r$, то мы получаем, что действительно каждая итерация этого цикла увеличивает новое значение r на единицу.

- $i \le r$

  В этом случае мы по приведённой формуле инициализируем значение $z[i]$ некоторым числом z_0. Сравним это начальное значение z_0 с величиной $r - i + 1$, получаем три варианта:

  - $z_0 < r - i + 1$

    Докажем, что в этом случае ни одной итерации цикл while не сделает.

    Это легко доказать, например, от противного: если бы цикл while сделал хотя бы одну итерацию, это бы означало, что определённое нами значение z_0 было неточным, меньше настоящей длины совпадения. Но т.к. строки $s[l \dots r]$ и $s[0 \dots r-l]$ совпадают, то это означает, что в позиции $z[i-l]$ стоит неправильное значение: меньше, чем должно быть.

    Таким образом, в этом варианте из корректности значения $z[i-l]$ и из того, что оно меньше $r - i + 1$, следует, что это значение совпадает с искомым значением $z[i]$.

  - $z_0 = r - i + 1$

    В этом случае цикл while может совершить несколько итераций, однако каждая из них будет приводить к увеличению нового значения r на единицу: потому что первым же сравниваемым символом будет $s[r+1]$, который вылазит за пределы отрезка $[l;r]$.

  - $z_0 > r - i + 1$

    Этот вариант принципиально невозможен, в силу определения z_0.

Таким образом, мы доказали, что каждая итерация вложенного цикла приводит к продвижению указателя r вправо. Т.к. r не могло оказаться больше $n-1$, это означает, что всего этот цикл сделает не более $n-1$ итерации.

Поскольку вся остальная часть алгоритма, очевидно, работает за $O(n)$, то мы доказали, что и весь алгоритм вычисления Z-функции выполняется за линейное время.

Применения

Рассмотрим несколько применений Z-функции при решении конкретных задач.

Применения эти будут во многом аналогичным применениям префикс-функции.

Поиск подстроки в строке

Во избежании путаницы, назовём одну строку **текстом** t, другую — **образцом** p. Таким образом, задача заключается в том, чтобы найти все вхождения образца p в текст t.

Для решения этой задачи образуем строку $s = p + \# + t$, т.е. к образцу припишем текст через символ-разделитель (который не встречается нигде в самих строках).

Посчитаем для полученной строки Z-функцию. Тогда для любого i в отрезке $[0; length(t) - 1]$ по соответствующему значению $z[i + length(p) + 1]$ можно понять, входит ли образец p в текст t, начиная с позиции i: если это значение Z-функции равно $length(p)$, то да, входит, иначе — нет.

Таким образом, асимптотика решения получилась $O(length(t) + length(p))$. Потребление памяти имеет ту же асимптотику.

Количество различных подстрок в строке

Дана строка s длины n. Требуется посчитать количество её различных подстрок.

Будем решать эту задачу итеративно. А именно, научимся, зная текущее количество различных подстрок, пересчитывать это количество при добавлении в конец одного символа.

Итак, пусть k — текущее количество различных подстрок строки s, и мы добавляем в конец символ c. Очевидно, в результате могли появиться некоторые новые подстроки, оканчивавшиеся на этом новом символе c (а именно, все подстроки, оканчивающиеся на этом символе, но не встречавшиеся раньше).

Возьмём строку $t = s + c$ и инвертируем её (запишем символы в обратном порядке). Наша задача — посчитать, сколько у строки t таких префиксов, которые не встречаются в ней более нигде. Но если мы посчитаем для строки t Z-функцию и найдём её максимальное значение $z_{\max}$, то, очевидно, в строке t встречается (не в начале) её префикс длины $z_{\max}$, но не большей длины. Понятно, префиксы меньшей длины уже точно встречаются в ней.

Итак, мы получили, что число новых подстрок, появляющихся при дописывании символа c, равно $len - z_{\max}$, где len — текущая длина строки после приписывания символа c.

Следовательно, асимптотика решения для строки длины n составляет $O(n^2)$.

Стоит заметить, что совершенно аналогично можно пересчитывать за $O(n)$ количество различных подстрок и при дописывании символа в начало, а также при удалении символа с конца или с начала.

Сжатие строки

Дана строка s длины n. Требуется найти самое короткое её "сжатое" представление, т.е. найти такую строку t наименьшей длины, что s можно представить в виде конкатенации одной или нескольких копий t.

Для решения посчитаем Z-функцию строки s, и найдём первую позицию i такую, что $i + z[i] = n$, и при этом n делится на i. Тогда строку s можно сжать до строки длины i.

Доказательство такого решения практически не отличается от доказательства решения с помощью префикс-функции.

Задачи в online judges

Список задач, которые можно решить, используя Z-функцию:

- UVA #455 "Periodic Strings" [сложность: средняя]
- UVA #11022 "String Factoring" [сложность: средняя]

Префикс-функция. Алгоритм Кнута-Морриса-Пратта

Префикс-функция. Определение

Дана строка $s[0 \dots n-1]$. Требуется вычислить для неё префикс-функцию, т.е. массив чисел $\pi[0 \dots n-1]$, где $\pi[i]$ определяется следующим образом: это такая наибольшая длина наибольшего собственного суффикса подстроки $s[0 \dots i]$, совпадающего с её префиксом (собственный суффикс — значит не совпадающий со всей строкой). В частности, значение $\pi[0]$ полагается равным нулю.

Математически определение префикс-функции можно записать следующим образом:

$$\pi[i] = \max_{k=0 \dots i} \ \{ \ k \ : \ s[0 \dots k-1] = s[i-k+1 \dots i] \ \}.$$

Например, для строки "abcabcd" префикс-функция равна: $[0, 0, 0, 1, 2, 3, 0]$, что означает:

- у строки "a" нет нетривиального префикса, совпадающего с суффиксом;
- у строки "ab" нет нетривиального префикса, совпадающего с суффиксом;
- у строки "abc" нет нетривиального префикса, совпадающего с суффиксом;
- у строки "abca" префикс длины 1 совпадает с суффиксом;
- у строки "abcab" префикс длины 2 совпадает с суффиксом;
- у строки "abcabc" префикс длины 3 совпадает с суффиксом;
- у строки "abcabcd" нет нетривиального префикса, совпадающего с суффиксом.

Другой пример — для строки "aabaaab" она равна: $[0, 1, 0, 1, 2, 2, 3]$.

Тривиальный алгоритм

Непосредственно следуя определению, можно написать такой алгоритм вычисления префикс-функции:

```
vector<int> prefix_function (string s) {
        int n = (int) s.length();
        vector<int> pi (n);
        for (int i=0; i<n; ++i)
                for (int k=0; k<=i; ++k)
                        if (s.substr(0,k) == s.substr(i-k+1,k))
                                pi[i] = k;
        return pi;
}
```

Как нетрудно заметить, работать он будет за $O(n^3)$, что слишком медленно.

Эффективный алгоритм

Этот алгоритм был разработан Кнутом (Knuth) и Праттом (Pratt) и независимо от них Моррисом (Morris) в 1977 г. (как основной элемент для алгоритма поиска подстроки в строке).

Первая оптимизация

Первое важное замечание — что значение $\pi[i+1]$ не более чем на единицу превосходит значение $\pi[i]$ для любого i.

Действительно, в противном случае, если бы $\pi[i+1] > \pi[i] + 1$, то рассмотрим этот суффикс, оканчивающийся в позиции $i+1$ и имеющий длину $\pi[i+1]$ — удалив из него последний символ, мы получим суффикс, оканчивающийся в позиции i и имеющий длину $\pi[i+1] - 1$, что лучше $\pi[i]$, т.е. пришли к противоречию. Иллюстрация этого противоречия (в этом примере $\pi[i-1]$ должно быть равно 3):

$$\underbrace{\overbrace{s_0\ s_1}^{\pi[i-1]=2}\quad s_2\ s_3}_{\pi[i]=4}\ \ldots\ \underbrace{s_{i-2}\ \overbrace{s_{i-2}\ s_{i-1}}^{\pi[i-1]=2}\ s_i}_{\pi[i]=4}$$

(на этой схеме верхние фигурные скобки обозначают две одинаковые подстроки длины 2, нижние фигурные скобки — две одинаковые подстроки длины 4)

Таким образом, при переходе к следующей позиции очередной элемент префикс-функции мог либо увеличиться на единицу, либо не измениться, либо уменьшиться на какую-либо величину. Уже этот факт позволяет нам снизить асимптотику до $O(n^2)$ — поскольку за один шаг значение могло вырасти максимум на единицу, то суммарно для всей строки могло произойти максимум n увеличений на единицу, и, как следствие (т.к. значение никогда не могло стать меньше нуля), максимум n уменьшений. В итоге получится $O(n)$ сравнений строк, т.е. мы уже достигли асимптотики $O(n^2)$.

Вторая оптимизация

Пойдём дальше — **избавимся от явных сравнений подстрок**. Для этого постараемся максимально использовать информацию, вычисленную на предыдущих шагах.

Итак, пусть мы вычислили значение префикс-функции $\pi[i]$ для некоторого i. Теперь, если $s[i+1] = s[\pi[i]]$, то мы можем с уверенностью сказать, что $\pi[i+1] = \pi[i] + 1$, это иллюстрирует схема:

$$\underbrace{\overbrace{s_0\ s_1\ s_2}^{\pi[i]}\quad \overbrace{s_3}^{s_3=s_{i+1}}}_{\pi[i+1]=\pi[i]+1}\ \ldots\ \underbrace{\overbrace{s_{i-2}\ s_{i-1}\ s_i}^{\pi[i]}\quad \overbrace{s_{i+1}}^{s_3=s_{i+1}}}_{\pi[i+1]=\pi[i]+1}$$

(на этой схеме снова одинаковые фигурные скобки обозначают одинаковые подстроки)

Пусть теперь, наоборот, оказалось, что $s[i+1] \neq s[\pi[i]]$. Тогда нам надо попытаться попробовать подстроку меньшей длины. В целях оптимизации хотелось бы сразу перейти к такой (наибольшей) длине $j < \pi[i]$, что по-прежнему выполняется префикс-свойство в позиции i, т.е. $s[0 \ldots j-1] = s[i-j+1 \ldots i]$:

$$\overbrace{\underbrace{s_0\ s_1}_{j}\ s_2\ s_3}^{\pi[i]}\ \ldots\ \overbrace{s_{i-3}\ s_{i-2}\ \underbrace{s_{i-1}\ s_i}_{j}}^{\pi[i]}\ s_{i+1}$$

Действительно, когда мы найдём такую длину j, то нам будет снова достаточно сравнить символы $s[i+1]$ и $s[j]$ — если они совпадут, то можно утверждать, что $\pi[i+1] = j+1$. Иначе нам надо будет снова найти меньшее (следующее по величине) значение j, для которого выполняется префикс-свойство, и так далее. Может случиться, что такие значения j кончатся — это происходит, когда $j = 0$. В этом случае, если $s[i+1] = s[0]$, то $\pi[i+1] = 1$, иначе $\pi[i+1] = 0$.

Итак, общая схема алгоритма у нас уже есть, нерешённым остался только вопрос об эффективном нахождении таких длин j. Поставим этот вопрос формально: по текущей длине j и позиции i (для которых выполняется префикс-свойство, т.е. $s[0 \ldots j-1] = s[i-j+1 \ldots i]$) требуется найти наибольшее $k < j$, для которого по-прежнему выполняется префикс-свойство:

$$\overbrace{\underbrace{s_0\ s_1}_{k}\ s_2\ s_3}^{j}\ \ldots\ \overbrace{s_{i-3}\ s_{i-2}\ \underbrace{s_{i-1}\ s_i}_{k}}^{j}\ s_{i+1}$$

После столь подробного описания уже практически напрашивается, что это значение k есть не что иное, как значение префикс-функции $\pi[j-1]$, которое уже было вычислено нами ранее (вычитание единицы появляется из-за 0-индексации строк). Таким образом, находить эти длины k мы можем за $O(1)$ каждую.

Итоговый алгоритм

Итак, мы окончательно построили алгоритм, который не содержит явных сравнений строк и выполняет $O(n)$ действий.

Приведём здесь итоговую схему алгоритма:

- Считать значения префикс-функции $\pi[i]$ будем по очереди: от $i = 1$ к $i = n-1$ (значение $\pi[0]$ просто присвоим равным нулю).
- Для подсчёта текущего значения $\pi[i]$ мы заводим переменную j, обозначающую длину текущего рассматриваемого образца. Изначально $j = \pi[i-1]$.
- Тестируем образец длины j, для чего сравниваем символы $s[j]$ и $s[i]$. Если они совпадают — то

полагаем $\pi[i] = j + 1$ и переходим к следующему индексу $i + 1$. Если же символы отличаются, то уменьшаем длину j, полагая её равной $\pi[j - 1]$, и повторяем этот шаг алгоритма с начала.

- Если мы дошли до длины $j = 0$ и так и не нашли совпадения, то останавливаем процесс перебора образцов и полагаем $\pi[i] = 0$ и переходим к следующему индексу $i + 1$.

Реализация

Алгоритм в итоге получился удивительно простым и лаконичным:

```
vector<int> prefix_function (string s) {
        int n = (int) s.length();
        vector<int> pi (n);
        for (int i=1; i<n; ++i) {
                int j = pi[i-1];
                while (j > 0 && s[i] != s[j])
                        j = pi[j-1];
                if (s[i] == s[j])  ++j;
                pi[i] = j;
        }
        return pi;
}
```

Как нетрудно заметить, этот алгоритм является **онлайновым** алгоритмом, т.е. он обрабатывает данные по ходу поступления — можно, например, считывать строку по одному символу и сразу обрабатывать этот символ, находя ответ для очередной позиции. Алгоритм требует хранения самой строки и предыдущих вычисленных значений префикс-функции, однако, как нетрудно заметить, если нам заранее известно максимальное значение, которое может принимать префикс-функция на всей строке, то достаточно будет хранить лишь на единицу большее количество первых символов строки и значений префикс-функции.

Применения

Поиск подстроки в строке. Алгоритм Кнута-Морриса-Пратта

Эта задача является классическим применением префикс-функции (и, собственно, она и была открыта в связи с этим).

Дан текст t и строка s, требуется найти и вывести позиции всех вхождений строки s в текст t.

Обозначим для удобства через n длину строки s, а через m — длину текста t.

Образуем строку $s + \# + t$, где символ $\#$ — это разделитель, который не должен нигде более встречаться. Посчитаем для этой строки префикс-функцию. Теперь рассмотрим её значения, кроме первых $n + 1$ (которые, как видно, относятся к строке s и разделителю). По определению, значение $\pi[i]$ показывает наидлиннейшую длину подстроки, оканчивающейся в позиции i и совпадающего с префиксом. Но в нашем случае это $\pi[i]$ — фактически длина наибольшего блока совпадения со строкой s и оканчивающегося в позиции i. Больше, чем n, эта длина быть не может — за счёт разделителя. А вот равенство $\pi[i] = n$ (там, где оно достигается), означает, что в позиции i оканчивается искомое вхождение строки s (только не надо забывать, что все позиции отсчитываются в склеенной строке $s + \# + t$).

Таким образом, если в какой-то позиции i оказалось $\pi[i] = n$, то в позиции $i - (n + 1) - n + 1 = i - 2n$ строки t начинается очередное вхождение строки s в строку t.

Как уже упоминалось при описании алгоритма вычисления префикс-функции, если известно, что значения префикс-функции не будут превышать некоторой величины, то достаточно хранить не всю строку и префикс-функцию, а только её начало. В нашем случае это означает, что нужно хранить в памяти лишь строку $s + \#$ и значение префикс-функции на ней, а потом уже считывать по одному символу строку t и пересчитывать текущее значение префикс-функции.

Итак, алгоритм Кнута-Морриса-Пратта решает эту задачу за $O(n + m)$ времени и $O(n)$ памяти.

Подсчёт числа вхождений каждого префикса

Здесь мы рассмотрим сразу две задачи. Дана строка s длины n. В первом варианте требуется для каждого префикса $s[0 \ldots i]$ посчитать, сколько раз он встречается в самой же строке s. Во втором варианте задачи дана другая строка t, и требуется для каждого префикса $s[0 \ldots i]$ посчитать, сколько раз он встречается в t.

Решим сначала первую задачу. Рассмотрим в какой-либо позиции i значение префикс-функции в ней $\pi[i]$. По определению, оно означает, что в позиции i оканчивается вхождение префикса строки s длины $\pi[i]$, и

никакой больший префикс оканчиваться в позиции i не может. В то же время, в позиции i могло оканчиваться и вхождение префиксов меньших длин (и, очевидно, совсем не обязательно длины $\pi[i]-1$). Однако, как нетрудно заметить, мы пришли к тому же вопросу, на который мы уже отвечали при рассмотрении алгоритма вычисления префикс-функции: по данной длине j надо сказать, какой наидлиннейший её собственный суффикс совпадает с её префиксом. Мы уже выяснили, что ответом на этот вопрос будет $\pi[j-1]$. Но тогда и в этой задаче, если в позиции i оканчивается вхождение подстроки длины $\pi[i]$, совпадающей с префиксом, то в i также оканчивается вхождение подстроки длины $\pi[\pi[i]-1]$, совпадающей с префиксом, а для неё применимы те же рассуждения, поэтому в i также оканчивается и вхождение длины $\pi[\pi[\pi[i]-1]-1]$ и так далее (пока индекс не станет нулевым). Таким образом, для вычисления ответа мы должны выполнить такой цикл:

```
vector<int> ans (n+1);
for (int i=0; i<n; ++i)
        ++ans[pi[i]];
for (int i=n-1; i>0; --i)
        ans[pi[i-1]] += ans[i];
```

Здесь мы для каждого значения префикс-функции сначала посчитали, сколько раз он встречался в массиве $\pi[]$, а затем посчитали такую в некотором роде динамику: если мы знаем, что префикс длины i встречался ровно $\mathrm{ans}[i]$ раз, то именно такое количество надо прибавить к числу вхождений его длиннейшего собственного суффикса, совпадающего с его префиксом; затем уже из этого суффикса (конечно, меньшей чем i длины) выполнится "пробрасывание" этого количества к своему суффиксу, и т.д.

Теперь рассмотрим вторую задачу. Применим стандартный приём: припишем к строке s строку t через разделитель, т. е. получим строку $s+\#+t$, и посчитаем для неё префикс-функцию. Единственное отличие от первой задачи будет в том, что учитывать надо только те значения префикс-функции, которые относятся к строке t, т.е. все $\pi[i]$ для $i \geq n+1$.

Количество различных подстрок в строке

Дана строка s длины n. Требуется посчитать количество её различных подстрок.

Будем решать эту задачу итеративно. А именно, научимся, зная текущее количество различных подстрок, пересчитывать это количество при добавлении в конец одного символа.

Итак, пусть k — текущее количество различных подстрок строки s, и мы добавляем в конец символ c. Очевидно, в результате могли появиться некоторые новые подстроки, оканчивавшиеся на этом новом символе c. А именно, добавляются в качестве новых те подстроки, оканчивающиеся на символе c и не встречавшиеся ранее.

Возьмём строку $t=s+c$ и инвертируем её (запишем символы в обратном порядке). Наша задача — посчитать, сколько у строки t таких префиксов, которые не встречаются в ней более нигде. Но если мы посчитаем для строки t префикс-функцию и найдём её максимальное значение $\pi_{\max}$, то, очевидно, в строке t встречается (не в начале) её префикс длины $\pi_{\max}$, но не большей длины. Понятно, префиксы меньшей длины уж точно встречаются в ней.

Итак, мы получили, что число новых подстрок, появляющихся при дописывании символа c, равно $s.\mathrm{length}()+1-\pi_{\max}$.

Таким образом, для каждого дописываемого символа мы за $O(n)$ можем пересчитать количество различных подстрок строки. Следовательно, за $O(n^2)$ мы можем найти количество различных подстрок для любой заданной строки.

Стоит заметить, что совершенно аналогично можно пересчитывать количество различных подстрок и при дописывании символа в начало, а также при удалении символа с конца или с начала.

Сжатие строки

Дана строка s длины n. Требуется найти самое короткое её "сжатое" представление, т.е. найти такую строку t наименьшей длины, что s можно представить в виде конкатенации одной или нескольких копий t.

Понятно, что проблема является в нахождении длины искомой строки t. Зная длину, ответом на задачу будет, например, префикс строки s этой длины.

Посчитаем по строке s префикс-функцию. Рассмотрим её последнее значение, т.е. $\pi[n-1]$, и введём обозначение $k=n-\pi[n-1]$. Покажем, что если n делится на k, то это k и будет длиной ответа, иначе эффективного сжатия не существует, и ответ равен n.

Действительно, пусть n делится на k. Тогда строку можно представить в виде нескольких блоков длины k, причём, по определению префикс-функции, префикс длины $n-k$ будет совпадать с её суффиксом. Но тогда последний блок должен будет совпадать с предпоследним, предпоследний - с предпредпоследним, и т.д. В итоге получится, что все блоки блоки совпадают, и такое k действительно подходит под ответ.

Покажем, что этот ответ оптимален. Действительно, в противном случае, если бы нашлось меньшее k, то и префикс-функция на конце была бы больше, чем $n-k$, т.е. пришли к противоречию.

Пусть теперь n не делится на k. Покажем, что отсюда следует, что длина ответа равна n. Докажем от противного — предположим, что ответ существует, и имеет длину p (p делитель n). Заметим, что префикс-функция необходимо должна быть больше $n - p$, т.е. этот суффикс должен частично накрывать первый блок. Теперь рассмотрим второй блок строки; т.к. префикс совпадает с суффиксом, и и префикс, и суффикс покрывают этот блок, и их смещение друг относительно друга k не делит длину блока p (а иначе бы k делило n), то все символы блока совпадают. Но тогда строка состоит из одного и того же символа, отсюда $k = 1$, и ответ должен существовать, т. е. так мы придём к противоречию.

$$\overbrace{s_0\ s_1\ s_2\ s_3}^{p}\ \overbrace{s_4\ s_5\ s_6\ s_7}^{p}$$

$$s_0\ s_1\ s_2\ \underbrace{\overbrace{s_3\ s_4\ s_5\ s_6}^{p}\ s_7}_{\pi[7]=5}$$

$$s_4 = s_3, \quad s_5 = s_4, \quad s_6 = s_5, \quad s_7 = s_6 \qquad \Longrightarrow \qquad s_0 = s_1 = s_2 = s_3$$

Построение автомата по префикс-функции

Вернёмся к уже неоднократно использованному приёму конкатенации двух строк через разделитель, т.е. для данных строк s и t вычисление префикс-функции для строки $s + \# + t$. Очевидно, что т.к. символ $\#$ является разделителем, то значение префикс-функции никогда не превысит $s.\mathrm{length}()$. Отсюда следует, что, как упоминалось при описании алгоритма вычисления префикс-функции, достаточно хранить только строку $s + \#$ и значения префикс-функции для неё, а для всех последующих символов префикс-функцию вычислять на лету:

$$\underbrace{s_0\ s_1\ \ldots\ s_{n-1}\ \#}_{\text{need to save}}\underbrace{t_0\ t_1\ \ldots\ t_{m-1}}_{\text{need not to save}}$$

Действительно, в такой ситуации, зная очередной символ $c \in t$ и значение префикс-функции в предыдущей позиции, можно будет вычислить новое значение префикс-функции, никак при этом не используя все предыдущие символы строки t и значения префикс-функции в них.

Другими словами, мы можем построить **автомат**: состоянием в нём будет текущее значение префикс-функции, переходы из одного состояния в другое будут осуществляться под действием символа:

$$s_0\ s_1\ \ldots\ s_{n-1}\ \#\ \underbrace{\ldots}_{\pi[i-1]} \quad \Longrightarrow \quad s_0\ s_1\ \ldots\ s_{n-1}\ \#\ \underbrace{\ldots}_{\pi[i-1]}\ +t_i \quad \Longrightarrow \quad s_0\ s_1\ \ldots\ s_{n-1}\ \#\ldots\underbrace{t_i}_{\pi[i]}$$

Таким образом, даже ещё не имея строки t, мы можем предварительно построить такую таблицу переходов $(\mathrm{old_}\pi, c) \to \mathrm{new_}\pi$ с помощью того же алгоритма вычисления префикс-функции:

```
string s; // входная строка
const int alphabet = 256; // мощность алфавита символов, обычно меньше

s += '#';
int n = (int) s.length();
vector<int> pi = prefix_function (s);
vector < vector<int> > aut (n, vector<int> (alphabet));
for (int i=0; i<n; ++i)
        for (char c=0; c<alphabet; ++c) {
                int j = i;
                while (j > 0 && c != s[j])
                        j = pi[j-1];
                if (c == s[j])  ++j;
                aut[i][c] = j;
        }
```

Правда, в таком виде алгоритм будет работать за $O(n^2 k)$ (k — мощность алфавита). Но заметим, что вместо внутреннего цикла while, который постепенно укорачивает ответ, мы можем воспользоваться уже вычисленной частью таблицы: переходя от значения j к значению $\pi[j-1]$, мы фактически говорим, что переход из состояния (j, c) приведёт в то же состояние, что и переход $(\pi[j-1], c)$, а для него ответ уже точно посчитан (т. к. $\pi[j-1] < j$):

```
string s; // входная строка
const int alphabet = 256; // мощность алфавита символов, обычно меньше

s += '#';
int n = (int) s.length();
vector<int> pi = prefix_function (s);
vector < vector<int> > aut (n, vector<int> (alphabet));
```

```
for (int i=0; i<n; ++i)
        for (char c=0; c<alphabet; ++c)
                if (i > 0 && c != s[i])
                        aut[i][c] = aut[pi[i-1]][c];
                else
                        aut[i][c] = i + (c == s[i]);
```

В итоге получилась крайне простая реализация построения автомата, работающая за $O(nk)$.

Когда может быть полезен такой автомат? Для начала вспомним, что мы считаем префикс-функцию для строки $s + \# + t$, и её значения обычно используют с единственной целью: найти все вхождения строки s в строку t.

Поэтому самая очевидная польза от построения такого автомата — **ускорение вычисления префикс-функции** для строки $s + \# + t$. Построив по строке $s + \#$ автомат, нам уже больше не нужна ни строка s, ни значения префикс-функции в ней, не нужны и никакие вычисления — все переходы (т.е. то, как будет меняться префикс-функция) уже предпосчитаны в таблице.

Но есть и второе, менее очевидное применение. Это случай, когда строка t **является гигантской строкой**, **построенной по какому-либо правилу**. Это может быть, например, строка Грея или строка, образованная рекурсивной комбинацией нескольких коротких строк, поданных на вход.

Пусть для определённости мы решаем **такую задачу**: дан номер $k \le 10^5$ строки Грея, и дана строка s длины $n \le 10^5$. Требуется посчитать количество вхождений строки s в k-ю строку Грея. Напомним, строки Грея определяются таким образом:

$$g_1 = "a"$$
$$g_2 = "aba"$$
$$g_3 = "abacaba"$$
$$g_4 = "abacabadabacaba"$$

В таких случаях даже просто построение строки t будет невозможным из-за её астрономической длины (например, k-ая строка Грея имеет длину $2^k - 1$). Тем не менее, мы сможем посчитать значение префикс-функции на конце этой строки, зная значение префикс-функции, которое было перед началом этой строки.

Итак, помимо самого автомата также посчитаем такие величины: $G[i][j]$ — значение автомата, достигаемое после "скармливания" ему строки g_i, если до этого автомат находился в состоянии j. Вторая величина — $K[i][j]$ — количество вхождений строки s в строку g_i, если до "скармливания" этой строки g_i автомат находился в состоянии j. Фактически, $K[i][j]$ — это количество раз, которое автомат принимал значение $s.\text{length}()$ за время "скармливания" строки g_i. Понятно, что ответом на задачу будет величина $K[k][0]$.

Как считать эти величины? Во-первых, базовыми значениями являются $G[0][j] = j, K[0][j] = 0$. А все последующие значения можно вычислять по предыдущим значениям и используя автомат. Итак, для вычисления этих значений для некоторого i мы вспоминаем, что строка g_i состоит из g_{i-1} плюс i-ый символ алфавита плюс снова g_{i-1}. Тогда после "скармливания" первого куска (g_{i-1}) автомат перейдёт в состояние $G[i-1][j]$, затем после "скармливания" символа char_i он перейдёт в состояние:

$$\text{mid} = \text{aut}[\ G[i-1][j]\ ][\text{char}_i]$$

После этого автомату "скармливается" последний кусок, т.е. g_{i-1}:

$$G[i][j] = G[i-1][\text{mid}]$$

Количества $K[i][j]$ легко считаются как сумма количеств по трём кускам g_i: строка g_{i-1}, символ char_i, и снова строка g_{i-1}:

$$K[i][j] = K[i-1][j] + (\text{mid} == s.\text{length}()) + K[i-1][mid]$$

Итак, мы решили задачу для строк Грея, аналогично можно решить целый класс таких задач. Например, точно таким же методом решается **следующая задача**: дана строка s, и образцы t_i, каждый из которых задаётся следующим образом: это строка из обычных символов, среди которых могут встречаться рекурсивные вставки других строк в форме $t_k[\text{cnt}]$, которая означает, что в это место должно быть вставлено cnt экземпляров строки t_k. Пример такой схемы:

$$t_1 = "abdeca"$$
$$t_2 = "abc" + t_1[30] + "abd"$$
$$t_3 = t_2[50] + t_1[100]$$
$$t_4 = t_2[10] + t_3[100]$$

Гарантируется, что это описание не содержит в себе циклических зависимостей. Ограничения таковы, что если явным образом раскрывать рекурсию и находить строки t_i, то их длины могут достигать порядка 100^{100}.

Требуется найти количество вхождений строки s в каждую из строк t_i.

Задача решается так же, построением автомата префикс-функции, затем надо вычислять и добавлять в него переходы по целым строкам t_i. В общем-то, это просто более общий случай по сравнению с задачей о строках Грея.

Задачи в online judges

Список задач, которые можно решить, используя префикс-функцию:

- UVA #455 "Periodic Strings" [сложность: средняя]
- UVA #11022 "String Factoring" [сложность: средняя]
- UVA #11452 "Dancing the Cheeky-Cheeky" [сложность: средняя]
- SGU #284 "Grammar" [сложность: высокая]

Алгоритмы хэширования в задачах на строки

Алгоритмы хэширования строк помогают решить очень много задач. Но у них есть большой недостаток: что чаще всего они не 100%-ны, поскольку есть множество строк, хэши которых совпадают. Другое дело, что в большинстве задач на это можно не обращать внимания, поскольку вероятность совпадения хэшей всё-таки очень мала.

Определение хэша и его вычисление

Один из лучших способов определить хэш-функцию от строки S следующий:

```
h(S)  =  S[0]  +  S[1] * P  +  S[2] * P^2  +  S[3] * P^3  +  ...  +  S[N] * P^N
```

где P - некоторое число.

Разумно выбирать для P простое число, примерно равное количеству символов во входном алфавите. Например, если строки предполаются состоящими только из маленьких латинских букв, то хорошим выбором будет P = 31. Если буквы могут быть и заглавными, и маленькими, то, например, можно P = 53.

Во всех кусках кода в этой статье будет использоваться P = 31.

Само значение хэша желательно хранить в самом большом числовом типе - int64, он же long long. Очевидно, что при длине строки порядка 20 символов уже будет происходить переполнение значение. Ключевой момент - что мы не обращаем внимание на эти переполнения, как бы беря хэш по модулю 2^64.

Пример вычисления хэша, если допустимы только маленькие латинские буквы:

```
const int p = 31;
long long hash = 0, p_pow = 1;
for (size_t i=0; i<s.length(); ++i)
{
        // желательно отнимать 'a' от кода буквы
        // единицу прибавляем, чтобы у строки вида 'aaaaa' хэш был ненулевой
        hash += (s[i] - 'a' + 1) * p_pow;
        p_pow *= p;
}
```

В большинстве задач имеет смысл сначала вычислить все нужные степени P в каком-либо массиве.

Пример задачи. Поиск одинаковых строк

Уже теперь мы в состоянии эффективно решить такую задачу. Дан список строк S[1..N], каждая длиной не более M символов. Допустим, требуется найти все повторяющиеся строки и разделить их на группы, чтобы в каждой группе были только одинаковые строки.

Обычной сортировкой строк мы бы получили алгоритм со сложностью O (N M log N), в то время как используя хэши, мы получим O (N M + N log N).

Алгоритм. Посчитаем хэш от каждой строки, и отсортируем строки по этому хэшу.

```
vector<string> s (n);
// ... считывание строк ...

// считаем все степени p, допустим, до 10000 - максимальной длины строк
const int p = 31;
vector<long long> p_pow (10000);
p_pow[0] = 1;
for (size_t i=1; i<p_pow.size(); ++i)
        p_pow[i] = p_pow[i-1] * p;

// считаем хэши от всех строк
// в массиве храним значение хэша и номер строки в массиве s
vector < pair<long long, int> > hashes (n);
for (int i=0; i<n; ++i)
{
        long long hash = 0;
```

```
        for (size_t j=0; j<s[i].length(); ++j)
                hash += (s[i][j] - 'a' + 1) * p_pow[j];
        hashes[i] = make_pair (hash, i);
}

// сортируем по хэшам
sort (hashes.begin(), hashes.end());

// выводим ответ
for (int i=0, group=0; i<n; ++i)
{
        if (i == 0 || hashes[i].first != hashes[i-1].first)
                cout << "\nGroup " << ++group << ":";
        cout << ' ' << hashes[i].second;
}
```

Хэш подстроки и его быстрое вычисление

Предположим, нам дана строка S, и даны индексы I и J. Требуется найти хэш от подстроки S[I..J].

По определению имеем:

```
H[I..J]  =  S[I]  +  S[I+1] * P  +  S[I+2] * P^2  +  ...  + S[J] * P^(J-I)
```

откуда:

```
H[I..J] * P[I]  =  S[I] * P[I]  +  ...  +  S[J] * P[J],
H[I..J] * P[I]  =  H[0..J]  -  H[0..I-1]
```

Полученное свойство является очень важным.

Действительно, получается, что, **зная только хэши от всех префиксов строки** `S`, **мы можем за** `O(1)` **получить хэш любой подстроки**.

Единственная возникающая проблема - это то, что нужно уметь делить на P[I]. На самом деле, это не так просто. Поскольку мы вычисляем хэш по модулю 2^64, то для деления на P[I] мы должны найти к нему обратный элемент в поле (например, с помощью Расширенного алгоритма Евклида), и выполнить умножение на этот обратный элемент.

Впрочем, есть и более простой путь. В большинстве случаев, **вместо того чтобы делить хэши на степени** `P`, **можно**, **наоборот**, **умножать их на эти степени**.

Допустим, даны два хэша: один умноженный на P[I], а другой - на P[J]. Если I < J, то умножим перый хэш на P[J-I], иначе же умножим второй хэш на P[I-J]. Теперь мы привели хэши к одной степени, и можем их спокойно сравнивать.

Например, код, который вычисляет хэши всех префиксов, а затем за O (1) сравнивает две подстроки:

```
string s;  int i1, i2, len; // входные данные

// считаем все степени p
const int p = 31;
vector<long long> p_pow (s.length());
p_pow[0] = 1;
for (size_t i=1; i<p_pow.size(); ++i)
        p_pow[i] = p_pow[i-1] * p;

// считаем хэши от всех префиксов
vector<long long> h (s.length());
for (size_t i=0; i<s.length(); ++i)
{
        h[i] = (s[i] - 'a' + 1) * p_pow[i];
        if (i)  h[i] += h[i-1];
}

// получаем хэши двух подстрок
long long h1 = h[i1+len-1];
if (i1)  h1 -= h[i1-1];
long long h2 = h[i2+len-1];
if (i2)  h2 -= h[i2-1];
```

```
// сравниваем их
if (i1 < i2 && h1 * p_pow[i2-i1] == h2 ||
        i1 > i2 && h1 == h2 * p_pow[i1-i2])
        cout << "equal";
else
        cout << "different";
```

Применение хэширования

Вот некоторые типичные применения хэширования:

- Алгоритм Рабина-Карпа поиска подстроки в строке за O (N)
- Определение количества различных подстрок за O (N^2 log N) (см. ниже)
- Определение количества палиндромов внутри строки

Определение количества различных подстрок

Пусть дана строка S длиной N, состоящая только из маленьких латинских букв. Требуется найти количество различных подстрок в этой строке.

Для решения переберём по очереди длину подстроки: L = 1 .. N.

Для каждого L мы построим массив хэшей подстрок длины L, причём приведём хэши к одной степени, и отсортируем этот массив. Количество различных элементов в этом массиве прибавляем к ответу.

Реализация:

```
string s; // входная строка
int n = (int) s.length();

// считаем все степени p
const int p = 31;
vector<long long> p_pow (s.length());
p_pow[0] = 1;
for (size_t i=1; i<p_pow.size(); ++i)
        p_pow[i] = p_pow[i-1] * p;

// считаем хэши от всех префиксов
vector<long long> h (s.length());
for (size_t i=0; i<s.length(); ++i)
{
        h[i] = (s[i] - 'a' + 1) * p_pow[i];
        if (i)  h[i] += h[i-1];
}

int result = 0;

// перебираем длину подстроки
for (int l=1; l<=n; ++l)
{
        // ищем ответ для текущей длины

        // получаем хэши для всех подстрок длины l
        vector<long long> hs (n-l+1);
        for (int i=0; i<n-l+1; ++i)
        {
                long long cur_h = h[i+l-1];
                if (i)  cur_h -= h[i-1];
                // приводим все хэши к одной степени
                cur_h *= p_pow[n-i-1];
                hs[i] = cur_h;
        }

        // считаем количество различных хэшей
        sort (hs.begin(), hs.end());
        hs.erase (unique (hs.begin(), hs.end()), hs.end());
        result += (int) hs.size();
}

cout << result;
```

Алгоритм Рабина-Карпа поиска подстроки в строке за O (N)

Этот алгоритм базируется на хэшировании строк, и тех, кто не знаком с темой, отсылаю к "Алгоритмам хэширования в задачах на строки".

Авторы алгоритма - Рабин (Rabin) и Карп (Karp), 1987 год.

Дана строка S и текст T, состоящие из маленьких латинских букв. Требуется найти все вхождения строки S в текст T за время O (|S| + |T|).

Алгоритм. Посчитаем хэш для строки S. Посчитаем значения хэшей для всех префиксов строки T. Теперь переберём все подстроки T длины |S| и каждую сравним с |S| за время O (1).

Реализация

```
string s, t; // входные данные

// считаем все степени p
const int p = 31;
vector<long long> p_pow (max (s.length(), t.length()));
p_pow[0] = 1;
for (size_t i=1; i<p_pow.size(); ++i)
        p_pow[i] = p_pow[i-1] * p;

// считаем хэши от всех префиксов строки T
vector<long long> h (t.length());
for (size_t i=0; i<t.length(); ++i)
{
        h[i] = (t[i] - 'a' + 1) * p_pow[i];
        if (i)  h[i] += h[i-1];
}

// считаем хэш от строки S
long long h_s = 0;
for (size_t i=0; i<s.length(); ++i)
        h_s += (s[i] - 'a' + 1) * p_pow[i];

// перебираем все подстроки T длины |S| и сравниваем их
for (size_t i = 0; i + s.length() - 1 < t.length(); ++i)
{
        long long cur_h = h[i+s.length()-1];
        if (i)  cur_h -= h[i-1];
        // приводим хэши к одной степени и сравниваем
        if (cur_h == h_s * p_pow[i])
                cout << i << ' ';
}
```

Разбор выражений. Обратная польская нотация

Дана строка, представляющая собой математическое выражение, содержащее числа, переменные, различные операции. Требуется вычислить его значение за $O(n)$, где n — длина строки.

Здесь описан алгоритм, который переводит это выражение в так называемую **обратную польскую нотацию** (явным или неявным образом), и уже в ней вычисляет выражение.

Обратная польская нотация

Обратная польская нотация — это форма записи математических выражений, в которой операторы расположены после своих операндов.

Например, следующее выражение:

$$a + b * c * d + (e - f) * (g * h + i)$$

в обратной польской нотации записывается следующим образом:

$$abc * d * +ef - gh * i + *+$$

Обратная польская нотация была разработана австралийским философом и специалистом в области теории вычислительных машин Чарльзом Хэмблином в середине 1950-х на основе польской нотации, которая была предложена в 1920 г. польским математиком Яном Лукасевичем.

Удобство обратной польской нотации заключается в том, что выражения, представленные в такой форме, очень **легко вычислять**, причём за линейное время. Заведём стек, изначально он пуст. Будем двигаться слева направо по выражению в обратной польской нотации; если текущий элемент — число или переменная, то кладём на вершину стека её значение; если же текущий элемент — операция, то достаём из стека два верхних элемента (или один, если операция унарная), применяем к ним операцию, и результат кладём обратно в стек. В конце концов в стеке останется ровно один элемент - значение выражения.

Очевидно, этот простой алгоритм выполняется за $O(n)$, т.е. порядка длины выражения.

Разбор простейших выражений

Пока мы рассматриваем только простейший случай: все операции **бинарны** (т.е. от двух аргументов), и все **левоассоциативны** (т.е. при равенстве приоритетов выполняются слева направо). Скобки разрешены.

Заведём два стека: один для чисел, другой для операций и скобок (т.е. стек символов). Изначально оба стека пусты. Для второго стека будем поддерживать предусловие, что все операции упорядочены в нём по строгому убыванию приоритета, если двигаться от вершины стека. Если в стеке есть открывающие скобки, то упорядочен каждый блок операций, находящийся между скобками, а весь стек в таком случае не обязательно упорядочен.

Будем идти по строке слева направо. Если текущий элемент — цифра или переменная, то положим в стек значение этого числа/переменной. Если текущий элемент — открывающая скобка, то положим её в стек. Если текущий элемент — закрывающая скобка, то будем выталкивать из стека и выполнять все операции до тех пор, пока мы не извлечём открывающую скобку (т.е., иначе говоря, встречая закрывающую скобку, мы выполняем все операции, находящиеся внутри этой скобки). Наконец, если текущий элемент — операция, то, пока на вершине стека находится операция с таким же или большим приоритетом, будем выталкивать и выполнять её.

После того, как мы обработаем всю строку, в стеке операций ещё могут остаться некоторые операции, которые ещё не были вычислены, и нужно выполнить их все (т.е. действуем аналогично случаю, когда встречаем закрывающую скобку).

Вот реализация данного метода на примере обычных операций $+ - * / \%$:

```
bool delim (char c) {
        return c == ' ';
}

bool is_op (char c) {
        return c=='+' || c=='-' || c=='*' || c=='/' || c=='%';
}

int priority (char op) {
        return
```

```
                op == '+' || op == '-' ? 1 :
                op == '*' || op == '/' || op == '%' ? 2 :
                -1;
}

void process_op (vector<int> & st, char op) {
        int r = st.back();  st.pop_back();
        int l = st.back();  st.pop_back();
        switch (op) {
                case '+':  st.push_back (l + r);  break;
                case '-':  st.push_back (l - r);  break;
                case '*':  st.push_back (l * r);  break;
                case '/':  st.push_back (l / r);  break;
                case '%':  st.push_back (l % r);  break;
        }
}

int calc (string & s) {
        vector<int> st;
        vector<char> op;
        for (size_t i=0; i<s.length(); ++i)
                if (!delim (s[i]))
                        if (s[i] == '(')
                                op.push_back ('(');
                        else if (s[i] == ')') {
                                while (op.back() != '(')
                                        process_op (st, op.back()),  op.
pop_back();
                                op.pop_back();
                        }
                        else if (is_op (s[i])) {
                                char curop = s[i];
                                while (!op.empty() && priority(op.back())
>= priority(s[i]))
                                        process_op (st, op.back()),  op.
pop_back();
                                op.push_back (curop);
                        }
                        else {
                                string operand;
                                while (s[i] >= 'a' && s[i] <= 'z' || isdigit
(s[i]))
                                        operand += s[i++];
                                --i;
                                if (isdigit (operand[0]))
                                        st.push_back (atoi (operand.c_str()));
                                else
                                        st.push_back
(get_variable_val (operand));
                        }
        while (!op.empty())
                process_op (st, op.back()),  op.pop_back();
        return st.back();
}
```

Таким образом, мы научились вычислять значение выражения за $O(n)$, и при этом мы неявно воспользовались обратной польской нотацией: мы расположили операции в таком порядке, когда к моменту вычисления очередной операции оба её операнда уже вычислены. Слегка модифицировав вышеописанный алгоритм, можно получить выражение в обратной польской нотаци и в явном виде.

Унарные операции

Теперь предположим, что выражение содержит унарные операции (т.е. от одного аргумента). Например, особенно часто встречаются унарный плюс и минус.

Одно из отличий этого случая заключается в необходимости определения того, является ли текущая операция унарной или бинарной.

Можно заметить, что перед унарной операцией всегда стоит либо другая операция, либо открывающая скобка,

либо вообще ничего (если она стоит в самом начале строки). Перед бинарной операцией, напротив, всегда стоит либо операнд (число/переменная), либо закрывающая скобка. Таким образом, достаточно завести какой-нибудь флаг для указания того, может ли следующая операция быть унарной или нет.

Ещё чисто реализационная тонкость — как различать унарные и бинарные операции при извлечении из стека и вычислении. Здесь можно, например, для унарных операций вместо символа $s[i]$ класть в стек $-s[i]$.

Приоритет для унарных операций нужно выбирать таким, чтобы он был больше приоритетов всех бинарных операций.

Кроме того, надо заметить, что унарные операции фактически являются правоассоциативными — если подряд идут несколько унарных операций, то они должны обрабатываться справа налево (для описания этого случая см. ниже; приведённый здесь код уже учитывает правоассоциативность).

Реализация для бинарных операций $+ \; - \; * \; /$ и унарных операций $+ \; -$:

```
bool delim (char c) {
        return c == ' ';
}

bool is_op (char c) {
        return c=='+' || c=='-' || c=='*' || c=='/' || c=='%';
}

int priority (char op) {
        if (op < 0)
                return op == -'+' || op == '-' ? 4;
        return
                op == '+' || op == '-' ? 1 :
                op == '*' || op == '/' || op == '%' ? 2 :
                -1;
}

void process_op (vector<int> & st, char op) {
        if (op < 0) {
                int l = st.back();  st.pop_back();
                switch (-op) {
                        case '+':  st.push_back (l);  break;
                        case '-':  st.push_back (-l);  break;
                }
        }
        else {
                int r = st.back();  st.pop_back();
                int l = st.back();  st.pop_back();
                switch (op) {
                        case '+':  st.push_back (l + r);  break;
                        case '-':  st.push_back (l - r);  break;
                        case '*':  st.push_back (l * r);  break;
                        case '/':  st.push_back (l / r);  break;
                        case '%':  st.push_back (l % r);  break;
                }
        }
}

int calc (string & s) {
        bool may_unary = true;
        vector<int> st;
        vector<char> op;
        for (size_t i=0; i<s.length(); ++i)
                if (!delim (s[i]))
                        if (s[i] == '(') {
                                op.push_back ('(');
                                may_unary = true;
                        }
                        else if (s[i] == ')') {
                                while (op.back() != '(')
                                        process_op (st, op.back()),  op.
pop_back();
                                op.pop_back();
                                may_unary = false;
                        }
                        else if (is_op (s[i])) {
                                char curop = s[i];
```

```
                                if (may_unary && isunary (curop))  curop =
-curop;
                                while (!op.empty() && (
                                        curop >= 0 && priority(op.back())
>= priority(curop)
                                        || curop < 0 && priority(op.back())
> priority(curop))
                                        )
                                        process_op (st, op.back()),  op.
pop_back();
                                op.push_back (curop);
                                may_unary = true;
                        }
                        else {
                                string operand;
                                while (s[i] >= 'a' && s[i] <= 'z' || isdigit
(s[i]))
                                        operand += s[i++];
                                --i;
                                st.push_back (get_val (operand));
                                may_unary = false;
                        }
        while (!op.empty())
                process_op (st, op.back()),  op.pop_back();
        return st.back();
}
```

Стоит заметить, что в простейших случаях, например, когда из унарных операций разрешены только + и −, правоассоциативность не играет никакой роли, поэтому в таких ситуациях никаких усложнений в схему можно не вводить. Т.е. цикл:

```
                                while (!op.empty() && (
                                        curop >= 0 && priority(op.back())
>= priority(curop)
                                        || curop < 0 && priority(op.back())
> priority(curop))
                                        )
                                        process_op (st, op.back()),  op.
pop_back();
```

Можно заменить на:

```
                                while (!op.empty() && priority(op.back())
>= priority(curop))
                                        process_op (st, op.back()),  op.
pop_back();
```

Правоассоциативность

Правоассоциативность оператора означает, что при равенстве приоритетов операторы вычисляются справа налево (соотвественно, левоассоциативность - когда слева направо).

Как уже было отмечено выше, унарные операторы обычно являются правоассоциативными. Другой пример - обычно операция возведения в степень считается правоассоциативной (действительно, a^b^c обычно воспринимается как a^(b^c), а не (a^b)^c).

Какие отличия нужно внести в алгоритм, чтобы корректно обрабатывать правоассоциативность? На самом деле, изменения нужны самые минимальные. Единственное отличие будет проявляться только при равенстве приоритетов, и заключается оно в том, что операции с равным приоритетом, находящиеся на вершине стека, не должны выполнять раньше текущей операции.

Таким образом, единственные отличия нужно внести в функцию calc:

```
int calc (string & s) {
...
                                while (!op.empty() && (
                                        left_assoc(curop) && priority(op.
```

```
back()) >= priority(curop)
                                        || !left_assoc(curop) && priority
(op.back()) > priority(curop)))
...
}
```

Суффиксный массив

Дана строка $s[0 \ldots n-1]$ длины n.

i-ым **суффиксом** строки называется подстрока $s[i \ldots n-1]$, $i = 0 \ldots n-1$.

Тогда **суффиксным массивом** строки s называется перестановка индексов суффиксов $p[0 \ldots n-1]$, $p[i] \in [0; n-$, которая задаёт порядок суффиксов в порядке лексикографической сортировки. Иными словами, нужно выполнить сортировку всех суффиксов заданной строки.

Например, для строки $s = abaab$ суффиксный массив будет равен:

$$(2, 3, 0, 4, 1)$$

Построение за $O(n \log n)$

Строго говоря, описываемый ниже алгоритм будет выполнять сортировку не суффиксов, а **циклических сдвигов** строки. Однако из этого алгоритма легко получить и алгоритм сортировки суффиксов: достаточно приписать в конец строки произвольный символ, который заведомо меньше любого символа, из которого может состоять строка (например, это может быть доллар или шарп; в языке C в этих целях можно использовать уже имеющийся нулевой символ).

Сразу заметим, что поскольку мы сортируем циклические сдвиги, то и подстроки мы будем рассматривать **циклические**: под подстрокой $s[i \ldots j]$, когда $i > j$, понимается подстрока $s[i \ldots n-1] + s[0 \ldots j]$. Кроме того, предварительно все индексы берутся по модулю длины строки (в целях упрощения формул я буду опускать явные взятия индексов по модулю).

Рассматриваемый нами алгоритм состоит из примерно $\log n$ фаз. На k-ой фазе ($k = 0 \ldots \lceil \log n \rceil$) сортируются циклические подстроки длины 2^k. На последней, $\lceil \log n \rceil$-ой фазе, будут сортироваться подстроки длины $2^{\lceil \log n \rceil} > n$, что эквивалентно сортировке циклических сдвигов.

На каждой фазе алгоритм помимо перестановки $p[0 \ldots n-1]$ индексов циклических подстрок будет поддерживать для каждой циклической подстроки, начинающейся в позиции i с длиной 2^k, **номер** $c[i]$ **класса эквивалентности**, которому эта подстрока принадлежит. В самом деле, среди подстрок могут быть одинаковые, и алгоритму понадобится информация об этом. Кроме того, номера $c[i]$ классов эквивалентности будем давать таким образом, чтобы они сохраняли и информацию о порядке: если один суффикс меньше другого, то и номер класса он должен получить меньший. Классы будем для удобства нумеровать с нуля. Количество классов эквивалентности будем хранить в переменной classes.

Приведём **пример**. Рассмотрим строку $s = aaba$. Значения массивов $p[]$ и $c[]$ на каждой стадии с нулевой по вторую таковы:

$$\begin{array}{lll} 0: & p = (0,1,3,2) & c = (0,0,1,0) \\ 1: & p = (0,3,1,2) & c = (0,1,2,0) \\ 2: & p = (3,0,1,2) & c = (1,2,3,0) \end{array}$$

Стоит отметить, что в массиве $p[]$ возможны неоднозначности. Например, на нулевой фазе массив мог равняться: $p = (3,1,0,2)$. То, какой именно вариант получится, зависит от конкретной реализации алгоритма, но все варианты одинаково правильны. В то же время, в массиве $c[]$ никаких неоднозначностей быть не могло.

Перейдём теперь к построению **алгоритма**. Входные данные:

```
char *s; // входная строка
int n; // длина строки

// константы
const int maxlen = ...; // максимальная длина строки
const int alphabet = 256; // размер алфавита, <= maxlen
```

На **нулевой фазе** мы должны отсортировать циклические подстроки длины 1, т.е. отдельные символы строки, и разделить их на классы эквивалентности (просто одинаковые символы должны быть отнесены к одному классу эквивалентности). Это можно сделать тривиально, например, сортировкой подсчётом. Для каждого символа посчитаем, сколько раз он встретился. Потом по этой информации восстановим массив $p[]$. После этого, проходом по массиву $p[]$ и сравнением символов, строится массив $c[]$.

```
int p[maxlen], cnt[maxlen], c[maxlen];
memset (cnt, 0, alphabet * sizeof(int));
for (int i=0; i<n; ++i)
        ++cnt[s[i]];
for (int i=1; i<alphabet; ++i)
        cnt[i] += cnt[i-1];
for (int i=0; i<n; ++i)
        p[--cnt[s[i]]] = i;
c[p[0]] = 0;
int classes = 1;
for (int i=1; i<n; ++i) {
        if (s[p[i]] != s[p[i-1]])  ++classes;
        c[p[i]] = classes-1;
}
```

Далее, пусть мы выполнили $k-1$-ю фазу (т.е. вычислили значения массивов $p[]$ и $c[]$ для неё), теперь научимся за $O(n)$ выполнять **следующую**, k**-ю**, **фазу**. Поскольку фаз всего $O(\log n)$, это даст нам требуемый алгоритм с временем $O(n \log n)$.

Для этого заметим, что циклическая подстрока длины 2^k состоит из двух подстрок длины 2^{k-1}, которые мы можем сравнивать между собой за $O(1)$, используя информацию с предыдущей фазы — номера $c[]$ классов эквивалентности. Таким образом, для подстроки длины 2^k, начинающейся в позиции i, вся необходимая информация содержится в паре чисел $(c[i], c[i+2^k])$ (повторимся, мы используем массив $c[]$ с предыдущей фазы).

$$\ldots \overbrace{\underbrace{s_i \ldots s_{i+2^{k-1}-1}}_{\text{length}=2^{k-1},\text{class}=c[i]} \quad \underbrace{s_{i+2^{k-1}} \ldots s_{i+2^k-1}}_{\text{length}=2^{k-1},\ \text{class}=c[i+2^{k-1}]}}^{\text{length}=2^k} \ldots \overbrace{\underbrace{s_j \ldots s_{j+2^{k-1}-1}}_{\text{length}=2^{k-1},\text{class}=c[j]} \quad \underbrace{s_{j+2^{k-1}} \ldots s_{j+2^k-1}}_{\text{length}=2^{k-1},\text{class}=c[j+2^{k-1}]}}^{\text{length}=2^k} \ldots$$

Это даёт нам весьма простое решение: **отсортировать** подстроки длины 2^k просто **по** этим **парам чисел**, это и даст нам требуемый порядок, т.е. массив $p[]$. Однако обычная сортировка, выполняющаяся за время $O(n \log n)$, нас не устроит — это даст алгоритм построения суффиксного массива с временем $O(n \log^2 n)$ (зато этот алгоритм несколько проще в написании, чем описываемый ниже).

Как быстро выполнить такую сортировку пар? Поскольку элементы пар не превосходят n, то можно выполнить сортировку подсчётом. Однако для достижения лучшей скрытой в асимптотике константы вместо сортировки пар придём к сортировке просто чисел.

Воспользуемся здесь приёмом, на котором основана так называемая **цифровая сортировка**: чтобы отсортировать пары, отсортируем их сначала по вторым элементам, а затем — по первым элементам (но уже обязательно стабильной сортировкой, т.е. не нарушающей относительного порядка элементов при равенстве). Однако отдельно вторые элементы уже упорядочены — этот порядок задан в массиве $p[]$ от предыдущей фазы. Тогда, чтобы упорядочить пары по вторым элементам, надо просто от каждого элемента массива $p[]$ отнять 2^{k-1} — это даст нам порядок сортировки пар по вторым элементам (ведь $p[]$ даёт упорядочение подстрок длины 2^{k-1}, и при переходе к строке вдвое большей длины эти подстроки становятся их вторыми половинками, поэтому от позиции второй половинки отнимается длина первой половинки).

Таким образом, с помощью всего лишь вычитаний от элементов массива $p[]$ мы производим сортировку по вторым элементам пар. Теперь надо произвести стабильную сортировку по первым элементам пар, её уже можно выполнить за $O(n)$ с помощью сортировки подсчётом.

Осталось только пересчитать номера $c[]$ классов эквивалентности, но их уже легко получить, просто пройдя по полученной новой перестановке $p[]$ и сравнивая соседние элементы (опять же, сравнивая как пары двух чисел).

Приведём **реализацию** выполнения всех фаз алгоритма, кроме нулевой. Вводятся дополнительно временные массивы pn и cn (pn — содержит перестановку в порядке сортировки по вторым элементам пар, cn — новые номера классов эквивалентности).

```
int pn[maxlen], cn[maxlen];
for (int h=0; (1<<h)<n; ++h) {
        for (int i=0; i<n; ++i) {
                pn[i] = p[i] - (1<<h);
                if (pn[i] < 0)  pn[i] += n;
        }
        memset (cnt, 0, classes * sizeof(int));
        for (int i=0; i<n; ++i)
                ++cnt[c[pn[i]]];
        for (int i=1; i<classes; ++i)
                cnt[i] += cnt[i-1];
        for (int i=n-1; i>=0; --i)
```

```
                p[--cnt[c[pn[i]]]] = pn[i];
        cn[p[0]] = 0;
        classes = 1;
        for (int i=1; i<n; ++i) {
                int mid1 = (p[i] + (1<<h)) % n,  mid2 = (p[i-1] + (1<<h)) % n;
                if (c[p[i]] != c[p[i-1]] || c[mid1] != c[mid2])
                        ++classes;
                cn[p[i]] = classes-1;
        }
        memcpy (c, cn, n * sizeof(int));
}
```

Этот алгоритм требует $O(n \log n)$ времени и $O(n)$ памяти. Впрочем, если учитывать ещё размер k алфавита, то время работы становится $O((n+k)\log n)$, а размер памяти — $O(n+k)$.

Применения

Нахождение наименьшего циклического сдвига строки

Вышеописанный алгоритм производит сортировку циклических сдвигов (если к строке не приписывать доллар), а потому $p[0]$ даст искомую позицию наименьшего циклического сдвига. Время работы — $O(n \log n)$.

Поиск подстроки в строке

Пусть требуется в тексте t искать строку s в режиме онлайн (т.е. заранее строку s нужно считать неизвестной). Построим суффиксный массив для текста t за $O(|t| \log |t|)$. Теперь подстроку s будем искать следующим образом: заметим, что искомое вхождение должно быть префиксом какого-либо суффикса t. Поскольку суффиксы у нас упорядочены (это даёт нам суффиксный массив), то подстроку s можно искать бинарным поиском по суффиксам строки. Сравнение текущего суффикса и подстроки s внутри бинарного поиска можно производить тривиально, за $O(|p|)$. Тогда асимптотика поиска подстроки в тексте становится $O(|p| \log |t|)$.

Сравнение двух подстрок строки

Требуется по заданной строке s, произведя некоторый её препроцессинг, научиться за $O(1)$ отвечать на запросы сравнения двух произвольных подстрок (т.е. проверка, что первая подстрока равна/меньше/больше второй).

Построим суффиксный массив за $O(|s| \log |s|)$, при этом сохраним промежуточные результаты: нам понадобятся массивы $c[]$ от каждой фазы. Поэтому памяти потребуется тоже $O(|s| \log |s|)$.

Используя эту информацию, мы можем за $O(1)$ сравнивать любые две подстроки длины, равной степени двойки: для этого достаточно сравнить номера классов эквивалентности из соответствующей фазы. Теперь надо обобщить этот способ на подстроки произвольной длины.

Пусть теперь поступил очередной запрос сравнения двух подстрок длины l с началами в индексах i и j. Найдём наибольшую длину блока, помещающегося внутри подстроки такой длины, т.е. наибольшее k такое, что $2^k \le l$. Тогда сравнение двух подстрок можно заменить сравнением двух пар перекрывающихся блоков длины 2^k: сначала надо сравнить два блока, начинающихся в позициях i и j, а при равенстве — сравнить два блока, заканчивающихся в позициях $i+l-1$ и $j+l-1$:

$$\ldots \overbrace{\underbrace{s_i \ldots s_{i+l-2^{k-1}} \ldots s_{i+2^{k-1}-1}}_{2^{k-1}} \ldots s_{i+l-1}}^{\text{first}} \ldots \overbrace{\underbrace{s_j \ldots s_{j+l-2^{k-1}} \ldots s_{j+2^{k-1}-1}}_{2^{k-1}} \ldots s_{j+l-1}}^{\text{second}} \ldots$$

$$\ldots \overbrace{s_i \ldots \underbrace{s_{i+l-2^{k-1}} \ldots s_{i+2^{k-1}-1}}_{2^{k-1}} \ldots s_{i+l-1}}^{\text{first}} \ldots \overbrace{s_j \ldots \underbrace{s_{j+l-2^{k-1}} \ldots s_{j+2^{k-1}-1}}_{2^{k-1}} \ldots s_{j+l-1}}^{\text{second}} \ldots$$

Таким образом, реализация получается примерно такой (здесь считается, что вызывающая процедура сама вычисляет k, поскольку сделать это за константное время не так легко (по-видимому, быстрее всего — предпосчётом), но в любом случае это не имеет отношения к применению суффиксного массива):

```
int compare (int i, int j, int l, int k) {
        pair<int,int> a = make_pair (c[k][i], c[k][i+l-(1<<(k-1))]);
        pair<int,int> b = make_pair (c[k][j], c[k][j+l-(1<<(k-1))]);
        return a == b ? 0 : a < b ? -1 : 1;
}
```

Наибольший общий префикс двух подстрок: способ с дополнительной памятью

Требуется по заданной строке s, произведя некоторый её препроцессинг, научиться за $O(\log |s|)$ отвечать на запросы наибольшего общего префикса (longest common prefix, lcp) для двух произвольных суффиксов с позициями i и j.

Способ, описываемый здесь, требует $O(|s| \log |s|)$ дополнительной памяти; другой способ, использующий линейный объём памяти, но неконстантное время ответа на запрос, описан в следующем разделе.

Построим суффиксный массив за $O(|s| \log |s|)$, при этом сохраним промежуточные результаты: нам понадобятся массивы $c[]$ от каждой фазы. Поэтому памяти потребуется тоже $O(|s| \log |s|)$.

Пусть теперь поступил очередной запрос: пара индексов i и j. Воспользуемся тем, что мы можем за $O(1)$ сравнивать любые две подстроки длины, являющейся степенью двойки. Для этого будем перебирать степень двойки (от большей к меньшей), и для текущей степени проверять: если подстроки такой длины совпадают, то к ответу прибавить эту степень двойки, а наибольший общий префикс продолжим искать справа от одинаковой части, т.е. к i и j надо прибавить текущую степень двойки.

Реализация:

```
int lcp (int i, int j) {
        int ans = 0;
        for (int k=log_n; k>=0; --k)
                if (c[k][i] == c[k][j]) {
                        ans += 1<<k;
                        i += 1<<k;
                        j += 1<<k;
                }
        return ans;
}
```

Здесь через $\log_n$ обозначена константа, равная логарифму n по основанию 2, округлённому вниз.

Наибольший общий префикс двух подстрок: способ без дополнительной памяти. Наибольший общий префикс двух соседних суффиксов

Требуется по заданной строке s, произведя некоторый её препроцессинг, научиться отвечать на запросы наибольшего общего префикса (longest common prefix, lcp) для двух произвольных суффиксов с позициями i и j.

В отличие от предыдущего метода, описываемый здесь будет выполнять препроцессинг строки за $O(n \log n)$ времени с $O(n)$ памяти. Результатом этого препроцессинга будет являться массив (который сам по себе является важным источником информации о строке, и потому использоваться для решения других задач). Ответы же на запрос будут производиться как результат выполнения запроса RMQ (минимум на отрезке, range minimum query) в этом массиве, поэтому при разных реализациях можно получить как логарифмическое, так и константное времена работы.

Базой для этого алгоритма является следующая идея: найдём каким-нибудь образом наибольшие общие префиксы для каждой **соседней в порядке сортировки пары суффиксов**. Иными словами, построим массив $\mathrm{lcp}[0 \dots n-2]$, где $\mathrm{lcp}[i]$ равен наибольшему общему префиксу суффиксов $p[i]$ и $p[i+1]$. Этот массив даст нам ответ для любых двух соседних суффиксов строки. Тогда ответ для любых двух суффиксов, не обязательно соседних, можно получить по этому массиву. В самом деле, пусть поступил запрос с некоторыми номерами суффиксов i и j. Найдём эти индексы в суффиксном массиве, т.е. пусть k_1 и k_2 — их позиции в массиве $p[]$ (упорядочим их, т.е. пусть $k_1 < k_2$). Тогда ответом на данный запрос будет минимум в массиве lcp, взятый на отрезке $[k_1; k_2 - 1]$. В самом деле, переход от суффикса i к суффиксу j можно заменить целой цепочкой переходов, начинающейся с суффикса i и заканчивающейся в суффиксе j, но включающей в себя все промежуточные суффиксы, находящиеся в порядке сортировки между ними.

Таким образом, если мы имеем такой массив lcp, то ответ на любой запрос наибольшего общего префикса сводится к запросу **минимума на отрезке** массива lcp. Эта классическая задача минимума на отрезке (range minimum query, RMQ) имеет множество решений с различными асимптотиками, описанные здесь.

Итак, основная наша задача — **построение** этого массива lcp. Строить его мы будем по ходу алгоритма построения суффиксного массива: на каждой текущей итерации будем строить массив lcp для циклических подстрок текущей длины.

После нулевой итерации массив lcp, очевидно, должен быть нулевым.

Пусть теперь мы выполнили $k-1$-ю итерацию, получили от неё массив lcp', и должны на текущей k-й итерации пересчитать этот массив, получив новое его значение lcp. Как мы помним, в алгоритме

построения суффиксного массива циклические подстроки длины 2^k разбивались пополам на две подстроки длины 2^{k-1}; воспользуемся этим же приёмом и для построения массива lcp.

Итак, пусть на текущей итерации алгоритм вычисления суффиксного массива выполнил свою работу, нашёл новое значение перестановки $p[]$ подстрок. Будем теперь идти по этому массиву и смотреть пары соседних подстрок: $p[i]$ и $p[i+1]$, $i = 0 \dots n-2$. Разбивая каждую подстроку пополам, мы получаем две различных ситуации: 1) первые половинки подстрок в позициях $p[i]$ и $p[i+1]$ различаются, и 2) первые половинки совпадают (напомним, такое сравнение можно легко производить, просто сравнивая номера классов $c[]$ с предыдущей итерации). Рассмотрим каждый из этих случаев отдельно.

1) Первые половинки подстрок различались. Заметим, что тогда на предыдущем шаге эти первые половинки необходимо были соседними. В самом деле, классы эквивалентности не могли исчезать (а могут только появляться), поэтому все различные подстроки длины 2^{k-1} дадут (в качестве первых половинок) на текущей итерации различные подстроки длины 2^k, и в том же порядке. Таким образом, для определения $\mathrm{lcp}[i]$ в этом случае надо просто взять соответствующее значение из массива lcp'.

2) Первые половинки совпадали. Тогда вторые половинки могли как совпадать, так и различаться; при этом, если они различаются, то они совсем не обязательно должны были быть соседними на предыдущей итерации. Поэтому в этом случае нет простого способа определить $\mathrm{lcp}[i]$. Для его определения надо поступить так же, как мы и собираемся потом вычислять наибольший общий префикс для любых двух суффиксов: надо выполнить запрос минимума (RMQ) на соответствующем отрезке массива lcp'.

Оценим **асимптотику** такого алгоритма. Как мы видели при разборе этих двух случаев, только второй случай даёт увеличение числа классов эквивалентности. Иными словами, можно говорить о том, что каждый новый класс эквивалентности появляется вместе с одним запросом RMQ. Поскольку всего классов эквивалентности может быть до n, то и искать минимум мы должны за асимптотику $O(\log n)$. А для этого надо использовать уже какую-то структуру данных для минимума на отрезке; эту структуру данных надо будет строить заново на каждой итерации (которых всего $O(\log n)$). Хорошим вариантом структуры данных будет **Дерево отрезков**: его можно построить за $O(n)$, а потом выполнять запросы за $O(\log n)$, что как раз и даёт нам итоговую асимптотику $O(n \log n)$.

Реализация:

```
int lcp[maxlen], lcpn[maxlen], lpos[maxlen], rpos[maxlen];
memset (lcp, 0, sizeof lcp);
for (int h=0; (1<<h)<n; ++h) {
        for (int i=0; i<n; ++i)
                rpos[c[p[i]]] = i;
        for (int i=n-1; i>=0; --i)
                lpos[c[p[i]]] = i;

        ... все действия по построению суфф. массива, кроме последней
строки (memcpy) ...

        rmq_build (lcp, n-1);
        for (int i=0; i<n-1; ++i) {
                int a = p[i],  b = p[i+1];
                if (c[a] != c[b])
                        lcpn[i] = lcp[rpos[c[a]]];
                else {
                        int aa = (a + (1<<h)) % n,  bb = (b + (1<<h)) % n;
                        lcpn[i] = (1<<h) + rmq (lpos[c[aa]], rpos[c[bb]]-1);
                        lcpn[i] = min (n, lcpn[i]);
                }
        }
        memcpy (lcp, lcpn, (n-1) * sizeof(int));

        memcpy (c, cn, n * sizeof(int));
}
```

Здесь помимо массива $\mathrm{lcp}[]$ вводится временный массив $\mathrm{lcpn}[]$ с его новым значением. Также поддерживается массив $\mathrm{pos}[]$, который для каждой подстроки хранит её позицию в перестановке $p[]$. Функция $\mathrm{rmq_build}$ — некоторая функция, строящая структуру данных для минимума по массиву-первому аргументу, размер его передаётся вторым аргументом. Функция rmq возвращает минимум на отрезке: с первого аргумента по второй включительно.

Из самого алгоритма построения суффиксного массива пришлось только вынести копирование массива $c[]$, поскольку во время вычисления lcp нам понадобятся старые значения этого массива.

Стоит отметить, что наша реализация находит длину общего префикса для **циклических подстрок**, в то время как на практике чаще бывает нужной длина общего префикса для суффиксов в их обычном понимании. В

этом случае надо просто ограничить значения lcp по окончании работы алгоритма:

```
for (int i=0; i<n-1; ++i)
        lcp[i] = min (lcp[i], min (n-p[i], n-p[i+1]));
```

Для **любых** двух суффиксов длину их наибольшего общего префикса теперь можно найти как минимум на соответствующем отрезке массива lcp:

```
for (int i=0; i<n; ++i)
        pos[p[i]] = i;
rmq_build (lcp, n-1);

... поступил запрос (i,j) на нахождение LCP ...
int result = rmq (min(i,j), max(i,j)-1);
```

Количество различных подстрок

Выполним **препроцессинг**, описанный в предыдущем разделе: за $O(n \log n)$ времени и $O(n)$ памяти мы для каждой пары соседних в порядке сортировки суффиксов найдём длину их наибольшего общего префикса. Найдём теперь по этой информации количество различных подстрок в строке.

Для этого будем рассматривать, какие новые подстроки начинаются в позиции $p[0]$, затем в позиции $p[1]$, и т. д. Фактически, мы берём очередной в порядке сортировки суффикс и смотрим, какие его префиксы дают новые подстроки. Тем самым мы, очевидно, не упустим из виду никакие из подстрок.

Пользуясь тем, что суффиксы у нас уже отсортированы, нетрудно понять, что текущий суффикс $p[i]$ даст в качестве новых подстрок все свои префиксы, кроме совпадающих с префиксами суффикса $p[i-1]$. Т.е. все его префиксы, кроме $\mathrm{lcp}[i-1]$ первых, дадут новые подстроки. Поскольку длина текущего суффикса равна $n - p[i]$, то окончательно получаем, что текущий суффикс $p[i]$ даёт $n - p[i] - \mathrm{lcp}[i-1]$ новых подстрок. Суммируя это по всем суффиксам (для самого первого, $p[0]$, отнимать нечего — прибавится просто $n - p[0]$), получаем **ответ** на задачу:

$$\sum_{i=0}^{n} (n - p[i]) - \sum_{i=0}^{n-1} \mathrm{lcp}[i]$$

Задачи в online judges

Задачи, которые можно решить, используя суффиксный массив:

- UVA #10679 "I Love Strings!!!" [сложность: средняя]

Суффиксный автомат

Суффиксный автомат (или **ориентированный ациклический граф слов**) — это мощная структура данных, которая позволяет решать множество строковых задач.

Например, с помощью суффиксного автомата можно искать все вхождения одной строки в другую, или подсчитывать количество различных подстрок данной строки — обе задачи он позволяет решать за линейное время.

На интуитивном уровне, суффиксный автомат можно понимать как сжатую информацию обо **всех подстроках** данной строки. Впечатляющим фактом является то, что суффиксный автомат содержит всю информацию в настолько сжатом виде, что для строки длины n он требует лишь $O(n)$ памяти. Более того, он может быть построен также за время $O(n)$ (если мы считаем размер алфавита k константой; в противном случае — за время $O(n \log k)$).

Исторически, впервые линейность размера суффиксного автомата была открыта в 1983 г. Blumer и др., а в 1985 — 1986 гг. были представлены первые алгоритмы его построения за линейное время (Crochemore, Blumer и др.). Более подробно — см. список литературы в конце статьи.

На английском языке суффиксный автомат называется "suffix automaton" (во множественном числе — "suffix automata"), а ориентированный ациклический граф слов — "directed acyclic word graph" (или просто "DAWG").

Определение суффиксного автомата

Определение. **Суффиксным автоматом** для данной строки s называется такой минимальный детерминированный конечный автомат, который принимает все суффиксы строки s.

Расшифруем это определение.

- Суффиксный автомат представляет собой ориентированный ациклический граф, в котором вершины называются **состояниями**, а дуги графа — это **переходы** между этими состояниями.
- Одно из состояний t_0 называется **начальным состоянием**, и оно должно быть истоком графа (т.е. из него достижимы все остальные состояния).
- Каждый **переход** в автомате — это дуга, помеченная некоторым символом. Все переходы, исходящие из какого-либо состояния, обязаны иметь **разные** метки. (С другой стороны, из состояния может не быть переходов по каким-либо символам.)
- Одно или несколько состояний помечены как **терминальные состояния**. Если мы пройдём из начального состояния t_0 по любому пути до какого-либо терминального состояния, и выпишем при этом метки всех пройденных дуг, то получится строка, которая обязана быть одним из суффиксов строки s.
- Суффиксный автомат содержит минимальное число вершин среди всех автоматов, удовлетворяющих описанным выше условиям. (Минимальность числа переходов не требуется, т.к. при условии минимальности числа состояний в автомате не может быть "лишних" путей — иначе это нарушило бы предыдущее свойство.)

Простейшие свойства суффиксного автомата

Простейшим, и вместе с тем важнейшим свойством суффиксного автомата является то, что он содержит в себе информацию обо всех подстроках строки s. А именно, **любой путь** из начального состояния t_0, если мы выпишем метки дуг вдоль этого пути, образует обязательно **подстроку** строки s. И наоборот, любой подстроке строки s соответствует некоторый путь, начинающийся в начальном состоянии t_0.

В целях упрощения объяснений, мы будем говорить, что подстроке **соответствует** тот путь из начального состояния, метки вдоль которого образуют эту подстроку. И наоборот, мы будем говорить, что любому пути **соответствует** та строка, которую образуют метки его дуг.

В каждое состояние суффиксного автомата ведёт один или несколько путей из начального состояния. Будем говорить, что состоянию **соответствует** набор строк, соответствующих всем этим путям.

Примеры построенных суффиксных автоматов

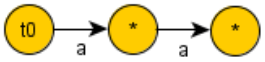
Приведём примеры суффиксных автоматов, построенных для нескольких простых строк.

Начальное состояние мы будем обозначать здесь через $t0$, а терминальные состояния — отмечать звёздочкой.

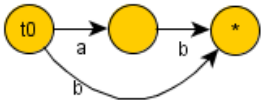
Для строки $s = ""$:

Для строки $s = "a"$:

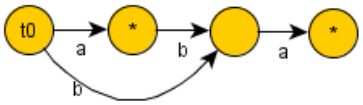
Для строки $s = "aa"$:



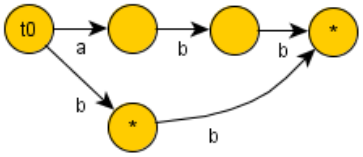
Для строки $s = "ab"$:



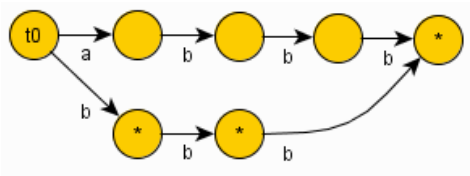
Для строки $s = "aba"$:



Для строки $s = "abb"$:



Для строки $s = "abbb"$:



Алгоритм построения суффиксного автомата за линейное время

Перед тем, как перейти непосредственно к описанию алгоритма построения, надо ввести несколько новых понятий и доказать простые, но очень важные для понимания суффиксного автомата леммы.

Позиции окончаний $endpos$, их свойства и связь с суффиксным автоматом

Рассмотрим любую непустую подстроку t строки s. Тогда назовём **множеством окончаний** $endpos(t)$ множество всех позиций в строке s, в которых оканчиваются вхождения строки t.

Мы будем называть две подстроки t_1 и t_2 $endpos$-эквивалентными, если их множества окончаний совпадают: $endpos(t_1) = endpos(t_2)$. Таким образом, все непустые подстроки строки s можно разбить на несколько **классов эквивалентности** соответственно их множествам $endpos$.

Оказывается, что в суффиксном автомате **$endpos$-эквивалентным подстрокам соответствует одно и то же состояние**. Иными словами, число состояний в суффиксном автомате равно количеству классов $endpos$-эквивалентности среди всех подстрок, плюс одно начальное состояние. Каждому состоянию суффиксного автомата соответствуют одна или несколько подстрок, имеющих одно и то же значение $endpos$.

Это утверждение мы примем как аксиому, и опишем алгоритм построения суффиксного автомата, исходя из этого предположения — как мы затем увидим, все требуемые свойства суффиксного автомата, кроме минимальности, будут выполнены. (А минимальность следует из теоремы Nerode — см. список литературы.)

Приведём также несколько простых, но важных утверждений касательно значений $endpos$.

Лемма 1. Две непустые подстроки u и w ($length(u) \le length(w)$) являются $endpos$-эквивалентными тогда и только тогда, когда строка u встречается в строке s только в виде суффикса строки w.

Доказательство практически очевидно. В одну сторону: если u и w имеют одинаковые позиции окончаний вхождения, то u является суффиксом w, и она присутствует в s только в виде суффикса w. В обратную сторону: если u является суффиксом w и входит только как этот суффикс, то их значения $endpos$ равны по определению.

Лемма 2. Рассмотрим две непустые подстроки u и w ($length(u) \le length(w)$). Тогда их множества $endpos$ либо не пересекаются, либо $endpos(w)$ целиком содержится в $endpos(u)$, причём это зависит от того, является u суффиксом w или нет:

$$\begin{cases} endpos(w) \subset endpos(u) & \text{if } u \text{ — suffix } w, \\ endpos(u) \cap endpos(w) = \emptyset & \text{otherwise.} \end{cases}$$

Доказательство. Предположим, что множества $endpos(u)$ и $endpos(w)$ имеют хотя бы один общий элемент. Тогда это означает, что строки u и w оканчиваются в одном и том же месте, т.е. u — суффикс w. Но тогда каждое вхождение строки w содержит на своём конце вхождение строки u, что и означает, что его множество $endpos(w)$ целиком вкладывается в множество $endpos(u)$.

Лемма 3. Рассмотрим некоторый класс $endpos$-эквивалентности. Отсортируем все подстроки, входящие в этот класс, по невозрастанию длины. Тогда в получившейся последовательности каждая подстрока будет на единицу короче предыдущей, и при этом являться суффиксом предыдущей. Иными словами, **подстроки**, **входящие в один класс эквивалентности**, **на самом деле являются суффиксами друг друга**, **и принимают всевозможные различные длины в некотором отрезке** $[x; y]$.

Доказательство.

Зафиксируем некоторый класс $endpos$-эквивалентности. Если он содержит только одну строку, то корректность леммы очевидна. Пусть теперь количество строк больше одной.

Согласно лемме 1, две различные $endpos$-эквивалентные строки всегда таковы, что одна является собственным суффиксом другой. Следовательно, в одном классе $endpos$-эквивалентности не может быть строк одинаковой длины.

Обозначим через w длиннейшую, а через u — кратчайшую строку в данном классе эквивалентности. Согласно лемме 1, строка u является собственным суффиксом строки w. Рассмотрим теперь любой суффикс строки w с длиной в отрезке $[length(u); length(w)]$, и покажем, что он содержится в этом же классе эквивалентности. В самом деле, этот суффикс может входить в s только в виде суффикса строки w (поскольку более короткий суффикс u входит только в виде суффикса строки w). Следовательно, согласно лемме 1, этот суффикс $endpos$-эквивалентен строке w, что и требовалось доказать.

Суффиксные ссылки

Рассмотрим некоторое состояние автомата $v \ne t_0$. Как мы теперь знаем, состоянию v соответствует некоторый класс строк с одинаковыми значениями $endpos$, причём если мы обозначим через w длиннейшую из этих строк, то все остальные будут суффиксами w.

Также мы знаем, что первые несколько суффиксов строки w (если мы рассматриваем суффиксы в порядке убывания их длины) содержатся в том же самом классе эквивалентности, а все остальные суффиксы (как минимум, пустой суффикс) — в каких-то других классах. Обозначим через t первый такой суффикс — в него мы и проведём суффиксную ссылку.

Иными словами, **суффиксная ссылка** $link(v)$ ведёт в такое состояние, которому соответствует **наидлиннейший суффикс** строки w, находящийся в другом классе $endpos$-эквивалентности.

Здесь мы считаем, что начальному состоянию t_0 соответствует отдельный класс эквивалентности (содержащий только пустую строку), и полагаем $endpos(t_0) = [-1 \dots length(s) - 1]$.

Лемма 4. Суффиксные ссылки образуют **дерево**, корнем которого является начальное состояние t_0.

Доказательство. Рассмотрим произвольное состояние $v \ne t_0$. Суффиксная ссылка $link(v)$ ведёт из него в состояние, которому соответствуют строки строго меньшей длины (это следует из определения суффиксной ссылки и из леммы 3). Следовательно, двигаясь по суффиксным ссылкам, мы рано или поздно придём из состояния v в начальное состояние t_0, которому соответствует пустая строка.

Лемма 5. Если мы построим из всех имеющихся множеств $endpos$ **дерево** (по принципу "множество-родитель содержит как подмножества всех своих детей"), то оно будет совпадать по структуре с деревом суффиксных ссылок.

Доказательство.

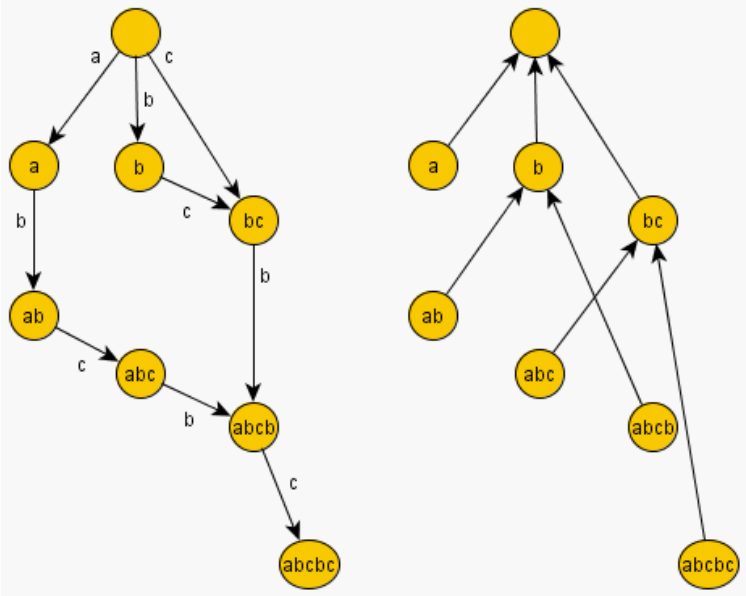
То, что из множеств $endpos$ можно построить дерево, следует из леммы 2 (о том, что любые два множества $endpos$ либо не пересекаются, либо одно содержится в другом).

Рассмотрим теперь произвольное состояние $v \ne t_0$ и его суффиксную ссылку $link(v)$. Из определения суффиксной ссылки и из леммы 2 следует:

$$endpos(v) \subset endpos(link(v)),$$

что вкупе с предыдущей леммой и доказывает наше утверждение: дерево суффиксных ссылок по сути своей есть дерево вкладывающихся множеств $endpos$.

Приведём пример дерева суффиксных ссылок в суффиксном автомате, построенном для строки "$abcbc$":



Промежуточный итог

Перед тем, как приступить к самому алгоритму, систематизируем накопленные выше знания, и введём пару вспомогательных обозначений.

- Множество подстрок строки s можно разбить на классы эквивалентности согласно их множествам окончания $endpos$.
- Суффиксный автомат состоит из начального состояния t_0, а также по одному состоянию на каждый класс $endpos$-эквивалентности.
- Каждому состоянию v соответствует одна или несколько строк. Обозначим через $longest(v)$ длиннейшую из таких строк, через $len(v)$ её длину. Обозначим через $shortest(v)$ кратчайшую из таких строк, а её длину через $minlen(v)$.

  Тогда все строки, соответствующие этому состоянию, являются различными суффиксами строки $longest(v)$ и имеют всевозможные длины в отрезке $[minlen(v); len(v)]$.
- Для каждого состояния $v \neq t_0$ определена суффиксная ссылка, ведущая в такое состояние, которое соответствует суффиксу строки $longest(v)$ длины $minlen(v) - 1$. Суффиксные ссылки образуют дерево с корнем в t_0, причём это дерево, по сути, является деревом отношений включения между множествами $endpos$.
- Таким образом, $minlen(v)$ для $v \neq t_0$ выражается с помощью суффиксной ссылки $link(v)$ как:

  $$minlen(v) = len(link(v)) + 1.$$
- Если мы стартуем из произвольного состояния v_0 и будем идти по суффиксным ссылкам, то рано или поздно дойдём до начального состояния t_0. При этом у нас получится последовательность непересекающихся отрезков $[minlen(v_i); len(v_i)]$, которые в объединении дадут один сплошной отрезок.

Алгоритм построения суффиксного автомата за линейное время

Приступим к описанию самого алгоритма. Алгоритм будет **онлайновым**, т.е. будет добавлять по одному символу строки s, перестраивая соответствующим образом текущий автомат.

Чтобы достичь линейного потребления памяти, в каждом состоянии мы будем хранить только значение $len, link$ и список переходов из этого состояния. Метки терминальных состояний мы поддерживать не будем (мы покажем, как расставить эти метки после построения суффиксного автомата, если имеется необходимость в них).

Изначально автомат состоит из единственного состояния t_0, которое мы условимся считать нулевым состоянием (остальные состояния будут получать номера $1, 2, \ldots$). Присвоим этому состоянию $len = 0$, а значению $link$ присвоим для удобства -1 (означающее ссылку на фиктивное, несуществующее состояние).

Соответственно, вся задача теперь сводится к тому, чтобы реализовать обработку **добавления одного символа** c в конец текущей строки. Опишем этот процесс:

- Пусть $last$ — это состояние, соответствующее всей текущей строке до добавления символа c. (Изначально $last = 0$, а после добавления каждого символа мы будем менять значение $last$.)
- Создадим новое состояние cur, проставив ему $len(cur) = len(last) + 1$. Значение $link(cur)$ пока считаем неопределённым.
- Сделаем такой цикл: изначально мы стоим в состоянии $last$; если из него нет перехода по букве c, то добавляем этот переход по букве c в состояние cur, и затем переходим по суффиксной ссылке, снова проверяя — если нет перехода, то добавляем. Если в какой-то момент случится, что такой переход уже есть, то останавливаемся — и обозначим через p номер состояния, на котором это произошло.
- Если ни разу не случилось, что переход по букве c уже имелся, и мы так и дошли до фиктивного состояния -1 (в которое мы попали по суффиксной ссылке из начального состояния t_0), то мы можем просто присвоить $link(cur) = 0$ и выйти.
- Допустим теперь, что мы остановились на некотором состоянии p, из которого уже был переход по букве c. Обозначим через q то состояние, куда ведёт этот имеющийся переход.
- Теперь у нас два случая в зависимости от того, $len(p) + 1 = len(q)$ или нет.
- Если $len(p) + 1 = len(q)$, то мы можем просто присвоить $link(cur) = q$ и выйти.
- В противном случае, всё несколько сложнее. Необходимо произвести "**клонирование**" состояния q: создать новое состояние $clone$, скопировав в него все данные из вершины q (суффиксную ссылку, переходы), за исключением значения len: надо присвоить $len(clone) = len(p) + 1$.

  После клонирования мы проводим суффиксную ссылку из cur в это состояние $clone$, также перенаправляем суффиксную ссылку из q в $clone$.

  Наконец, последнее, что мы должны сделать — это пройтись от состояния p по суффиксным ссылкам, и для каждого очередного состояния проверять: если имелся переход по букве c в состояние q, то перенаправлять его в состояние $clone$ (а если нет, то останавливаться).
- В любом случае, чем бы ни закончилось выполнение этой процедуры, мы в конце обновляем значение $last$, присваивая ему cur.

Если нам также нужно знать, какие вершины являются **терминальными**, а какие — нет, то мы можем найти все терминальные вершины после построения суффиксного автомата для всей строки. Для этого рассмотрим состояние, соответствующее всей строке (оно, очевидно, у нас сохранено в переменной $last$), и будем идти по его суффиксным ссылкам, пока не дойдём до начального состояния, и помечать каждое пройденное состояние как терминальное. Легко понять, что тем самым мы пометим состояния, соответствующие всем суффиксам строки s, что нам и требовалось.

В следующем разделе мы подробно рассмотрим каждый шаг алгоритма и покажем его **корректность**.

Здесь же лишь отметим, что из алгоритма видно, что добавление одного символа приводит к добавлению одного или двух состояний в автомат. Таким образом, **линейность числа состояний** очевидна.

Линейность числа переходов, да и вообще линейное время работы алгоритма менее понятны, и они будут доказаны ниже, после доказательства корректности алгоритма.

Доказательство корректности алгоритма

- Назовём переход (p, q) **сплошным**, если $len(p) + 1 = len(q)$. В противном случае, т.е. когда $len(p) + 1 < len(q)$, переход будем называть **несплошным**.

  Как можно увидеть из описания алгоритма, сплошные и несплошные переходы приводят к разным ветвям алгоритма. Сплошные переходы называются так потому, что, появившись впервые, они больше никогда не будут меняться. В противоположность им, несплошные переходы могут измениться при добавлении новых букв к строке (измениться может конец дуги-перехода).
- Во избежание неоднозначностей, под строкой s мы будем подразумевать строку, для которой был построен суффиксный автомат до добавления текущего символа c.
- Алгоритм начинается с того, что мы создаём новое состояние cur, которому будет соответствовать вся строка $s + c$. Понятно, почему мы обязаны создать новое состояние — т.к. вместе с добавлением нового символа возникает новый класс эквивалентности — это класс строк, оканчивающихся на добавляемом символе c.
- После создания нового состояния алгоритм проходится по суффиксным ссылкам, начиная с состояния, соответствующего всей строке s, и пытается добавить переход по символу c в состояние cur. Тем самым, мы приписываем к каждому суффиксу строки s символ c. Но добавлять новые переходы мы можем только в том случае, если они не будут конфликтовать с уже имеющимися, поэтому, как только мы встретим уже имеющийся переход по символу c, мы сразу же обязаны остановиться.
- Самый простой случай — если мы так и дошли до фиктивного состояния -1, добавив везде по новому переходу вдоль символа c. Это означает, что символ c в строке s ранее не встречался. Мы успешно добавили все переходы, осталось только проставить суффиксную ссылку у состояния cur — она, очевидно, должна быть равна 0, поскольку состоянию cur в данном случае соответствуют все суффиксы строки $s + c$.

- Второй случай — когда мы наткнулись на уже имеющийся переход (p, q). Это означает, что мы пытались добавить в автомат строку $x + c$ (где x — некоторый суффикс строки s, имеющий длину $len(p)$), а эта строка **уже была ранее добавлена** в автомат (т.е. строка $x + c$ уже входит как подстрока в строку s). Поскольку мы предполагаем, что автомат для строки s построен корректно, то новых переходов мы больше добавлять не должны.

  Однако возникает сложность с тем, куда вести суффиксную ссылку из состояния cur. Нам требуется провести суффиксную ссылку в такое состояние, в котором длиннейшей строкой будет являться как раз эта самая $x + c$, т.е. len для этого состояния должен быть равен $len(p) + 1$. Однако такого состояния могло и не существовать: в таком случае нам надо произвести "**расщепление**" состояния.

- Итак, по одному из возможных сценариев, переход (p, q) оказался сплошным, т.е. $len(q) = len(p) + 1$. В этом случае всё просто, никакого расщепления производить не надо, и мы просто проводим суффиксную ссылку из состояния cur в состояние q.

- Другой, более сложный вариант — когда переход несплошной, т.е. $len(q) > len(p) + 1$. Это означает, что состоянию q соответствует не только нужная нам подстрока $w + c$ длины $len(p) + 1$, но также и подстроки большей длины. Нам ничего не остаётся, кроме как произвести "**расщепление**" состояния q: разбить отрезок строк, соответствующих ей, на два подотрезка, так что первый будет заканчиваться как раз длиной $len(p) + 1$.

  Как производить это расщепление? Мы "**клонируем**" состояние q, делая его копию $clone$ с параметром $len(clone) = len(p) + 1$. Мы копируем в $clone$ из q все переходы, поскольку мы не хотим никоим образом менять пути, проходившие через q. Суффиксную ссылку из $clone$ мы ведём туда, куда вела старая суффиксная ссылка из q, а ссылку из q направляем в $clone$.

  После клонирования мы проводим суффиксную ссылку из cur в $clone$ — то, ради чего мы и производили клонирование.

  Остался последний шаг — перенаправить некоторые входящие в q переходы, перенаправив их на $clone$. Какие именно входящие переходы надо перенаправить? Достаточно перенаправить только переходы, соответствующие всем суффиксам строки $w + c$, т.е. нам надо продолжить двигаться по суффиксным ссылкам, начиная с вершины p, и до тех пор, пока мы не дойдём до фиктивного состояния -1 или не дойдём до состояния, переход из которого ведёт в состояние, отличное от q.

Доказательство линейного числа операций

Во-первых, сразу оговоримся, что мы считаем размер алфавита **константой**. Если это не так, то говорить о линейном времени работы не получится: список переходов из одной вершины надо хранить в виде сбалансированного дерева, позволяющего быстро производить операции поиска по ключу и добавления ключа. Следовательно, если мы обозначим через k размер алфавита, то асимптотика алгоритма составит $O(n \log k)$ при $O(n)$ памяти. Впрочем, если алфавит достаточно мал, то можно, пожертвовав памятью, избежать сбалансированных списков, а хранить переходы в каждой вершине в виде массива длины k (для быстрого поиска по ключу) и динамического списка (для быстрого обхода всех имеющихся ключей). Тем самым мы достигнем $O(n)$ во времени работы алгоритма, но ценой $O(nk)$ потребления памяти.

Итак, мы будем считать размер алфавита константным, т.е. каждая операция поиска перехода по символу, добавления перехода, поиск следующего перехода — все эти операции мы считаем работающими за $O(1)$.

Если мы рассмотрим все части алгоритма, то он содержит три места, линейная асимптотика которых не очевидна:

- Первое место — это проход по суффиксным ссылкам от состояния $last$ с добавлением рёбер по символу c.
- Второе место — копирование переходов при клонировании состояния q в новое состояние $clone$.
- Третье место — перенаправление переходов, ведущих в q, на $clone$.

Воспользуемся известным фактом, что размер суффиксного автомата (как по числу состояний, так и по числу переходов) **линеен**. (Доказательством линейности по числу состояний является сам алгоритм, а доказательство линейности по числу переходов мы приведём ниже, после реализации алгоритма.).

Тогда очевидна линейная суммарная асимптотика **первого и второго места**: ведь каждая операция здесь добавляет в автомат один новый переход.

Осталось оценить суммарную асимптотику **в третьем месте** — в том, где мы перенаправляем переходы, ведущие в q, на $clone$. Обозначим $v = longest(p)$. Это суффикс строки s, и с каждой итерацией его длина убывает — а, значит, и позиция v как суффикса строки s монотонно возрастает с каждой итерацией. При этом, если перед первой итерацией цикла соответствующая строка v была на глубине k ($k \geq 2$) от $last$ (если считать глубиной число суффиксных ссылок, которые надо пройти), то после последней итерации строка $v + c$ станет 2-ой суффиксной ссылкой на пути от cur (которое станет новым значением $last$).

Таким образом, каждая итерация этого цикла приводит к тому, что позиция строки $longest(link(link(last))$ как суффикса всей текущей строки будет монотонно увеличиваться. Следовательно, всего этот цикл не мог отработать более n итераций, **что и требовалось доказать**.

(Стоит заметить, что аналогичные аргументы можно использовать и для доказательства линейности работы первого места, вместо ссылки на доказательство линейности числа состояний.)

Реализация алгоритма

Вначале опишем структуру данных, которая будет хранить всю информацию о конкретном переходе ($len, link$, список переходов). При необходимости сюда можно добавить флаг терминальности, а также другую требуемую информацию. Список переходов мы храним в виде стандартного контейнера map, что позволяет достичь суммарно $O(n)$ памяти и $O(n \log k)$ времени на обработку всей строки.

```
struct state {
        int len, link;
        map<char,int> next;
};
```

Сам суффиксный автомат будем хранить в виде массива этих структур $state$. Как доказывается в следующем разделе, если $MAXN$ — это максимально возможная в программе длина строки, то достаточно завести память под $2 \cdot MAXN - 1$ состояний. Также мы храним переменную $last$ — состояние, соответствующее всей строке на данный момент.

```
const int MAXLEN = 100000;
state st[MAXLEN*2];
int sz, last;
```

Приведём функцию, инициализирующую суффиксный автомат (создающую автомат с единственным начальным состоянием):

```
void sa_init() {
        sz = last = 0;
        st[0].len = 0;
        st[0].link = -1;
        ++sz;
        /*
        // этот код нужен, только если автомат строится много раз для
разных строк:
        for (int i=0; i<MAXLEN*2; ++i)
                st[i].next.clear();
        */
}
```

Наконец, приведём реализацию основной функции — которая добавляет очередной символ в конец текущей строки, перестраивая соответствующим образом автомат:

```
void sa_extend (char c) {
        int cur = sz++;
        st[cur].len = st[last].len + 1;
        int p;
        for (p=last; p!=-1 && !st[p].next.count(c); p=st[p].link)
                st[p].next[c] = cur;
        if (p == -1)
                st[cur].link = 0;
        else {
                int q = st[p].next[c];
                if (st[p].len + 1 == st[q].len)
                        st[cur].link = q;
                else {
                        int clone = sz++;
                        st[clone].len = st[p].len + 1;
                        st[clone].next = st[q].next;
                        st[clone].link = st[q].link;
                        for (; p!=-1 && st[p].next[c]==q; p=st[p].link)
                                st[p].next[c] = clone;
                        st[q].link = st[cur].link = clone;
                }
        }
        last = cur;
}
```

Как уже упоминалось выше, если пожертвовать памятью (до $O(nk)$, где k — размер алфавита), то можно достичь времени построения автомата $O(n)$ даже для любых k — но для этого придётся в каждом состоянии хранить массив размера k (для быстрого поиска перехода по нужной букве) и список всех переходов (для быстрого обхода или копирования всех переходов).

Дополнительные свойства суффиксного автомата

Число состояний

Число состояний в суффиксном автомате, построенном для строки s длины n, **не превышает** $2n - 1$ (для $n \geq 3$).

Доказательством этого является описанный выше алгоритм (поскольку изначально автомат состоит из одного начального состояния, на первом и втором шагах добавляется ровно по одному состоянию, а на каждом из остальных $n - 2$ шагах могло добавляться по две вершины из-за расщепления состояния).

Однако эту оценку **легко показать и без знания алгоритма**. Вспомним о том, что число состояний на единицу больше количества различных значений множеств $endpos$ (поскольку есть начальное состояние, стоящее особняком от остальных). Кроме того, эти множества $endpos$ образуют дерево по принципу "вершина-родитель содержит в себе как подмножества всех детей". Рассмотрим это дерево, и немного преобразуем его: пока в нём есть внутренняя вершина с одним сыном, то это означает, что $endpos$ этого сына не содержит как минимум одно число из $endpos$ родителя; тогда создадим виртуальную вершину с $endpos$, равным этому числу, и привесим этого сына к родителю. В итоге мы получим дерево, в котором каждая внутренняя вершина имеет степень больше единицы, а число листьев не превосходит n. Следовательно, всего в таком дереве не более $2n - 1$ вершины. С учётом начального состояния, $endpos$ которого может совпадать с $endpos$ другого состояния, мы получаем оценку $2n$. Однако эту оценку можно чуть улучшить, заметив, что только на тесте вида "$aaaaa \ldots$" случается так, что имеется состояние, отличное от начального, $endpos$ которого содержит все числа от 1 до n; но на этом тесте оценка $2n$ явно не достигается, а, значит, мы улучшили итоговую оценку до $2n - 1$ (за счёт того, что перестали прибавлять единицу от начального состояния — на всех остальных тестах начальное состояние перестаёт совпадать с другой вершиной).

Итак, мы показали эту оценку независимо, без знания алгоритма.

Интересно заметить, что эта оценка неулучшаема, т.е. существует **тест**, **на котором она достигается**. Этот тест выглядит таким образом:

$$"abbbb \ldots"$$

При обработке этой строки на каждой итерации, начиная с третьей, будет происходить расщепление состояния, и, тем самым, будет достигаться оценка $2n - 1$.

Число переходов

Число переходов в суффиксном автомате, построенном для строки s длины n, **не превышает** $3n - 4$ (для $n \geq 3$).

Докажем это.

Оценим число сплошных переходов. Рассмотрим остовное дерево из длиннейших путей в автомате, начинающихся в состоянии t_0. Этот остов будет состоять только из сплошных рёбер, а, значит, их количество на единицу меньше числа состояний, т.е. не превосходит $2n - 2$.

Оценим теперь число несплошных переходов. Рассмотрим каждый несплошной переход; пусть текущий переход — это переход (p, q) по символу c. Поставим ему в соответствие строку $u + c + w$, где строка u соответствует длиннейшему пути из начального состояния в p, а w — длиннейшему пути из q в какое-либо терминальное состояние. С одной стороны, все такие строки $u + c + w$ для всех несплошных переходов будут различными (поскольку строки u и w образованы только сплошными переходами). С другой стороны, каждая из таких строк $u + c + w$, по определению терминального состояния, будет суффиксом всей строки s. Поскольку непустых суффиксов у строки s всего n штук, и к тому же вся строка s среди этих строк $u + c + w$ не могла содержаться (т.к. всей строке s соответствует путь из n сплошных рёбер), то общее число несплошных переходов не превосходит $n - 1$.

Складывая эти две оценки, мы получаем оценку $3n - 3$. Однако, вспоминая, что максимальное число состояний достигается только на тесте вида "$abbbb \ldots$", и на нём оценка $3n - 3$ явно не достигается, получаем окончательную оценку $3n - 4$, что и требовалось доказать.

Интересно отметить, что также существует **тест**, **на котором эта оценка достигается**:

$$"abbb \ldots bbbc"$$

Связь с суффиксным деревом. Построение суффиксного дерева

по суффиксному автомату и наоборот

Докажем две теоремы, устанавливающие взаимную связь между суффиксным автоматом и суффиксным деревом.

Сразу оговоримся, что мы считаем, что входная строка такова, что каждый суффикс имеет собственную вершину в суффиксном дереве (поскольку для произвольных строк это, вообще говоря, неверно: например, для строки $"aaa\ldots"$). Обычно этого добиваются путём приписывания в конец строки какого-нибудь особого символа (обычно обозначаемого через знак доллара).

Для удобства введём обозначения: $\overline{s}$ — это строка s, записанная в обратном порядке, $DAWG(s)$ — это суффиксный автомат, построенный для строки s, $ST(s)$ — это суффиксное дерево строки s.

Введём понятие **расширяющей ссылки**: зафиксируем вершину суффиксного дерева v и символ c; тогда расширяющая ссылка $ext[c, v]$ ведёт в вершину дерева, соответствующую строке $c + v$ (если этот путь $c + v$ оканчивается посередине ребра, то проведём ссылку в нижний конец этого ребра); если такого пути $c + v$ вообще нет в дереве, то расширяющая ссылка не определена. В некотором смысле, расширяющие ссылки противоположны суффиксным ссылкам.

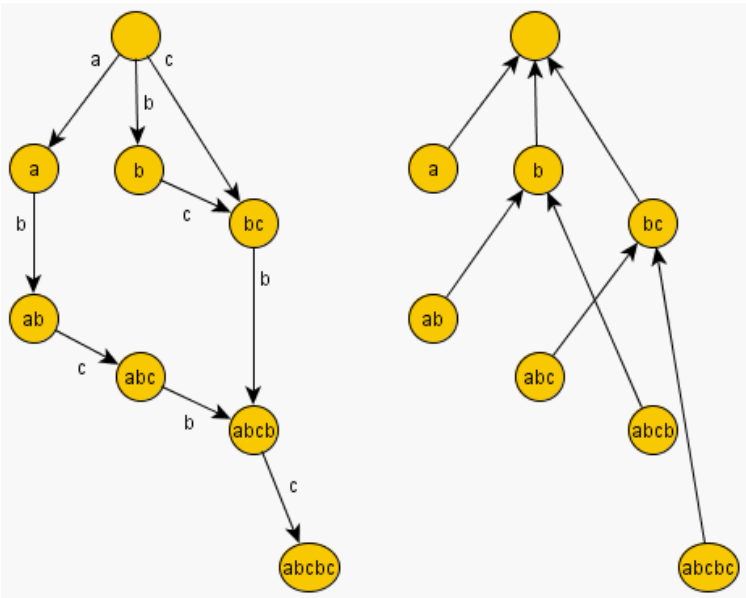
Теорема 1. Дерево, образованное суффиксными ссылками в $DAWG(s)$, является суффиксным деревом $ST(\overline{s})$.

Теорема 2. $DAWG(s)$ — это граф расширяющих ссылок суффиксного дерева $ST(\overline{s})$. Кроме того, сплошные рёбра в $DAWG(s)$ — это инвертированные суффиксные ссылки в $ST(\overline{s})$.

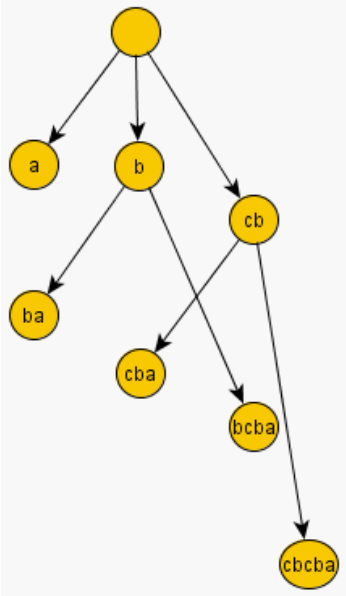
Эти две теоремы позволяют по одной из структур (суффиксному дереву или суффиксному автомату) построить другую за время $O(n)$ — эти два простых алгоритма будут рассмотрены нами ниже в теоремах 3 и 4.

В целях наглядности, приведём суффиксный автомат с его деревом суффиксных ссылок и соответствующее суффиксное дерево для инвертированной строки. Для примера возьмём строку $s = "abcbc"$.

$DAWG("abcbc")$ и его дерево суффиксных ссылок (для наглядности мы подписываем каждое состояние его $longest$-строкой):



$ST("cbcba")$:



Лемма. Следующие три утверждения эквивалентны для любых двух подстрок u и w:

- $endpos(u) = endpos(w)$ в строке s
- $firstpos(\overline{u}) = firstpos(\overline{w})$ в строке $\overline{s}$
- $\overline{u}$ и $\overline{w}$ лежат на одном и том же пути из корня в суффиксном дереве $ST(\overline{s})$.

Доказательство её довольно очевидно: если начала вхождений двух строк совпадают, то одна строка является префиксом другой, а, значит, одна строка лежит в суффиксном дереве на пути другой строки.

Доказательство теоремы 1.

Состояния суффиксного автомата соответствуют вершинам суффиксного дерева.

Рассмотрим произвольную суффиксную ссылку $y = link(x)$. Согласно определению суффиксной ссылки, $longest(y)$ является суффиксом $longest(x)$, причём среди всех таких y выбирается тот, у которого $len(y)$ максимально.

В терминах инвертированной строки $\overline{s}$ это означает, что суффиксная ссылка $link[x]$ ведёт в такой длиннейший префикс строки, соответствующей состоянию x, чтобы этому префиксу соответствовало отдельное состояние y. Иными словами, суффиксная ссылка $link[x]$ ведёт в предка вершины x в суффиксном дереве, что и требовалось доказать.

Доказательство теоремы 2.

Состояния суффиксного автомата соответствуют вершинам суффиксного дерева.

Рассмотрим произвольный переход (x, y, c) в суффиксном автомате $DAWG(s)$. Наличие этого перехода означает, что y — это такое состояние, класс эквивалентности которого содержит подстроку $longest(x) + c$. В инвертированной строке $\overline{s}$ это означает, что y это такое состояние, которому соответствует подстрока, $firstpos$ от которой (в тексте $\overline{s}$) совпадает с $firstpos$ от подстроки $c + \overline{longest(x)}$.

Это как раз и означает, что:

$$\overline{longest(y)} = ext[c, \overline{longest(x)}].$$

Первая часть теоремы доказана, осталось доказать вторую часть: что все сплошные переходы в автомате соответствуют суффиксным ссылкам в дереве. Сплошной переход отличается от несплошного тем, что $length(y) = length(x) + 1$, т.е. после приписывания символа c мы попали в состояние со строкой, максимальной из класса эквивалентности этого состояния. Это означает, что при вычислении соответствующей расширяющей ссылки $ext[c, \overline{longest(x)}]$ мы сразу попали в вершину дерева, а не спускались вниз до ближайшей вершины дерева. Таким образом, приписав один символ в начало, мы попали в другую вершину дерева — значит, если это и есть инвертированная суффиксная ссылка в дереве.

Теорема полностью доказана.

Теорема 3. Имея суффиксный автомат $DAWG(s)$, можно за время $O(n)$ построить суффиксное дерево $ST(\overline{s})$.

Теорема 4. Имея суффиксное дерево $ST(\overline{s})$, можно за время $O(n)$ построить суффиксный автомат $DAWG(s)$.

Доказательство теоремы 3.

Суффиксное дерево $ST(\overline{s})$ будет содержать столько же вершин, сколько состояний в $DAWG(s)$, причём

вершине дерева, получившейся из состояния v автомата, соответствует строка длины $len(v)$.

Согласно теореме 1, рёбра в дереве образуются как инвертированные суффиксные ссылки, и дуговые метки можно найти, исходя из разности len состояний, и дополнительно зная для каждого состояния автомата один любой элемент его множества $endpos$ (этот один элемент множества $endpos$ можно поддерживать при построении автомата).

Суффиксные ссылки в дереве мы можем построить согласно теореме 2: для этого достаточно просмотреть все сплошные переходы в автомате, и для каждого такого перехода (x, y) добавить ссылку $link(y) = x$.

Таким образом, за время $O(n)$ мы можем построить суффиксное дерево вместе с суффиксными ссылками в нём.

(Если мы считаем размер k алфавита не константой, то на всё перестроение потребуется время $O(n \log k)$.)

Доказательство теоремы 4.

Суффиксный автомат $DAWG(s)$ будет содержать столько же состояний, сколько вершин в $ST(\overline{s})$. У каждого состояния v его длиннейшая строка $longest(v)$ будет соответствовать инвертированному пути из корня дерева до вершины v.

Согласно теореме 2, чтобы построить все переходы в суффиксном автомате, нам надо найти все расширяющие ссылки $ext[c, v]$.

Во-первых, заметим, что часть этих расширяющих ссылок получаются непосредственно из суффиксных ссылок в дереве. В самом деле, если для любой вершины x мы рассмотрим её суффиксную ссылку $y = link(x)$, то это означает, что надо провести расширяющую ссылку из y в x по первому символу строки, соответствующей вершине x.

Однако так мы найдём не все расширяющие ссылки. Дополнительно надо пройтись по суффиксному дереву от листьев до корня, и для каждой вершины v просмотреть всех её сыновей, для каждого сына просмотреть все расширяющие ссылки $ext[c, w]$, и скопировать эту ссылку в вершину v, если по этому символу c ссылка из вершины v ещё не была найдена:

$$ext[c, v] = ext[c, w], \qquad \text{if } ext[c, w] = nil.$$

Этот процесс отработает за время $O(n)$, если мы считаем размер алфавита константным.

Наконец, осталось построить суффиксные ссылки в автомате, однако, согласно теореме 1, эти суффиксные ссылки получаются просто как рёбра суффиксного дерева $ST(\overline{s})$.

Таким образом, описанный алгоритм за время $O(n)$ строит суффиксный автомат по суффиксному дереву для инвертированной строки.

(Если же мы считаем, что размер k алфавита — также переменная величина, то асимптотика увеличится до $O(n \log k)$.)

Применения при решении задач

Ниже мы рассмотрим, какие задачи можно решать с помощью суффиксного автомата.

Мы для простоты будем считать размер алфавита k константой, что позволит нам считать асимптотику построения суффиксного автомата и прохода по нему константными.

Проверка вхождения

Условие. Дан текст T, и поступают запросы в виде: дана строка P, требуется проверить, входит или нет строка P в текст T как подстрока.

Асимптотика. Препроцессинг $O(length(T))$ и $O(length(P))$ на один запрос.

Решение. Построим суффиксный автомат по тексту T за время $O(length(T))$.

Как теперь отвечать на один запрос. Пусть текущее состояние — это переменная v, изначально она равна начальному состоянию t_0. Будем идти по символам строки P, соответствующим образом делая переход из текущего состояния v в новое состояние. Если в какой-то момент случилось, что перехода из текущего состояния по нужному символу не оказалось — то ответ на запрос "нет". Если же мы смогли обработать всю строку P, то ответ на запрос "да".

Понятно, что это будет работать за время $O(length(P))$. Более того, алгоритм фактически ищет длину наидлиннейшего префикса P, встречающегося в тексте — и если входные образцы таковы, что эти длины маленькие, то и алгоритм будет работать значительно быстрее, не обрабатывая всю строку целиком.

Количество различных подстрок

Условие. Дана строка S. Требуется узнать количество различных её подстрок.

Асимптотика. $O(length(S))$.

Решение. Построим суффиксный автомат по строке S.

В суффиксном автомате любой подстроке строки S соответствует какой-то путь в автомате. Поскольку повторяющихся строк в автомате быть не может, то ответ на задачу — это **количество различных путей** в автомате, начинающихся в начальной вершине t_0.

Учитывая, что суффиксный автомат представляет собой ациклический граф, количество различных путей можно считать в нём с помощью динамического программирования.

А именно, пусть $d[v]$ — это количество различных путей, начинающихся с состояния v (включая путь длины ноль). Тогда верно:

$$d[v] = 1 + \sum_{\substack{w\ :\\ (v,w,c)\in DAWG}} d[w],$$

т.е. $d[v]$ можно выразить как сумму ответов по всевозможным переходам из состояния v.

Ответом на задачу будет значение $d[t_0] - 1$ (единица отнимается, чтобы не учитывать пустую подстроку).

Суммарная длина различных подстрок

Условие. Дана строка S. Требуется узнать суммарную длину всех различных её подстрок.

Асимптотика. $O(length(S))$.

Решение. Решение задачи аналогично предыдущей, только теперь надо считать в динамике две величины: количество различных подстрок $d[v]$ и их суммарную длину $ans[v]$.

Как считать $d[v]$, описано в предыдущей задаче, а величину $ans[v]$ можно вычислить таким образом:

$$ans[v] = \sum_{\substack{w\ :\\ (v,w,c)\in DAWG}} d[w] + ans[w],$$

т.е. мы берём ответ для каждой вершины w, и прибавляем к нему $d[w]$, тем самым как бы приписывая в начало каждой из строк по одному символу.

Наименьший циклический сдвиг

Условие. Дана строка S. Требуется найти лексикографически наименьший её циклический сдвиг.

Асимптотика. $O(length(S))$.

Решение. Построим суффиксный автомат для строки $S + S$. Тогда этот автомат будет содержать в себе как пути все циклические сдвиги строки S.

Следовательно, задача сведётся к тому, чтобы найти в автомате лексикографически минимальный путь длины $length(S)$, что делается тривиальным образом: мы стартуем в начальном состоянии и каждый раз действуем жадно, переходя по переходу с минимальным символом.

Количество вхождений

Условие. Дан текст T, и поступают запросы в виде: дана строка P, требуется узнать, сколько раз строка P входит в текст T как подстрока (вхождения могут перекрываться).

Асимптотика. Препроцессинг $O(length(T))$ и $O(length(P))$ на один запрос.

Решение. Построим суффиксный автомат по тексту T.

Дальше нам надо сделать такой препроцессинг: для каждого состояния v автомата посчитать число $cnt[v]$, равное размеру множества $endpos(v)$. В самом деле, все строки, соответствующие одному и тому же состоянию, входят в T одинаковое число раз, равное количеству позиций в множестве $endpos$.

Однако явно поддерживать множества $endpos$ для всех состояний мы не можем, поэтому научимся считать только их размеры cnt.

Для этого поступим следующим образом. Для каждого состояния, если оно не было получено путём клонирования (и начальное состояние t_0 мы также не учитываем), изначально присвоим $cnt = 1$. Затем будем идти по всем состояниям в порядке убывания их длины len и пробрасывать текущее значение $cnt[v]$ по суффиксной ссылке:

$$cnt[link(v)] += cnt[v].$$

Утверждается, что в конце концов мы так посчитаем для каждого состояния правильные значения cnt.

Почему это верно? Всего состояний, полученных не путём клонирования, ровно $length(S)$, и i-ое из них появилось, когда мы добавили первые i символов. Следовательно, каждому из этих состояний мы ставим в соответствие эту позицию, при обработке которой оно появилось. Поэтому изначально у каждого такого состояния $cnt = 1$, а у всех остальных состояний $cnt = 0$.

Затем мы выполняем для каждого v такую операцию: $cnt[link(v)] += cnt[v]$. Смысл этого заключается в том, что если строка, соответствующая состоянию v, встречалась $cnt[v]$ раз, то все её суффиксы будут встречаться столько же.

Почему тем самым мы не учтём одну и ту же позицию несколько раз? Потому что из каждого состояния его значение "пробрасывается" только один раз, поэтому не могло так получиться, что из одного состояния его значение "пробросилось" до какого-то другого состояния дважды, двумя разными путями.

Таким образом, мы научились считать эти величины cnt для всех состояний автомата.

После этого ответ на запрос тривиален — надо просто вернуть $cnt[t]$, где t — состояние, соответствующее образцу P.

Позиция первого вхождения

Условие. Дан текст T, и поступают запросы в виде: дана строка P, требуется узнать позицию начала первого вхождения строки P.

Асимптотика. Препроцессинг $O(length(T))$ и $O(length(P))$ на один запрос.

Решение. Построим суффиксный автомат по тексту T.

Для решения задачи нам также надо добавить в препроцессинг нахождение позиций $firstpos$ для всех состояний автомата, т.е. для каждого состояния v мы хотим найти позицию $firstpos[v]$ окончания первого вхождения. Иными словами, мы хотим найти заранее минимальный элемент каждого из множеств $endpos(v)$ (поскольку явно поддерживать все множества $endpos$ мы не можем).

Поддерживать эти позиции $firstpos$ проще всего прямо по ходу построения автомата: когда мы создаём новое состояние cur при входе в функцию $sa_extend()$, то выставляем ему:

$$firstpos(cur) = len(cur) - 1$$

(если мы работаем в 0-индексации).

При клонировании вершины q в $clone$ мы ставим:

$$firstpos(clone) = firstpos(q),$$

(поскольку другой вариант значения только один — это $firstpos(cur)$, что явно больше).

Таким образом, ответ на запрос — это просто $firstpos(t) - length(P) + 1$, где t — состояние, соответствующее образцу P.

Позиции всех вхождений

Условие. Дан текст T, и поступают запросы в виде: дана строка P, требуется вывести позиции всех её вхождений в строку T (вхождения могу перекрываться).

Асимптотика. Препроцессинг $O(length(T))$. Ответ на один запрос за $O(length(P) + answer(P))$, где $answer(P)$ — это размер ответа, т.е. мы будем решать задачу за время порядка размера ввода и вывода.

Решение. Построим суффиксный автомат по тексту T. Аналогично предыдущей задаче, посчитаем в процессе построения автомата для каждого состояния позицию $firstpos$ окончания первого вхождения.

Пусть теперь поступил запрос — строка P. Найдём, какому состоянию t она соответствует.

Понятно, что $firstpos(t)$ точно должно входить в ответ. Какие ещё позиции надо найти? Мы учли состояние автомата, содержащее строку P, однако не учли другие состояния, которым соответствуют такие строки, что P является их суффиксом.

Иными словами, нам требуется найти все состояния, из которых **достижимо по суффиксным ссылкам** состояние t.

Следовательно, для решения задачи нам потребуется сохранить для каждого состояния список суффиксных ссылок, ведущих в него. Ответ на запрос тогда будет заключаться в том, чтобы сделать **обход в глубину/в ширину** по этим инвертированным суффиксным ссылкам, начиная с состояния t.

Этот обход будет работать за время $O(answer(P))$, поскольку мы не посетим одно и то же состояние дважды (потому что из каждого состояния суффиксная ссылка выходит только одна, поэтому не может быть двух путей, ведущих в одно и то же состояние).

Правда, надо учитывать, что у двух состояний их значения $firstpos$ **могут совпадать**: если одно состояние было получено клонированием другого. Однако это не ухудшает асимптотику, поскольку у каждой не-клонированной вершины может быть максимум один клон.

Более того, можно легко избавиться от вывода повторяющихся позиций, если мы не будем добавлять в ответ $firstpos$ от состояний-клонов. В самом деле, в любое состояние-клон ведёт суффиксная ссылка из того первоначального состояния, которое это состояние клонировало. Таким образом, если мы для каждого состояния запомним флаг is_clon и не будем добавлять в ответ $firstpos$ от состояний, для которых $is_clon = true$, то мы тем самым получим все требуемые $answer(P)$ позиций без повторов.

Приведём наброски реализации:

```
struct state {
        ...
        bool is_clon;
        int first_pos;
        vector<int> inv_link;
};


... после построения автомата ...
for (int v=1; v<sz; ++v)
        st[st[v].link].inv_link.push_back (v);
...


// ответ на запрос - вывод всех вхождений (возможно, с повторами)
void output_all_occurences (int v, int P_length) {
        if (! st[v].is_clon)
                cout << st[v].first_pos - P_length + 1 << endl;
        for (size_t i=0; i<st[v].inv_link.size(); ++i)
                output_all_occurences (st[v].inv_link[i], P_length);
}
```

Поиск кратчайшей строки, не входящей в данную

Условие. Дана строка S, и задан определённый алфавит. Требуется найти такую строку наименьшей длины, что она не встречается в S как подстрока.

Асимптотика. Решение за $O(length(S))$.

Решение. Решать будет динамическим программирование по автомату, построенному для строки S.

Пусть $d[v]$ — это ответ для вершины v, т.е. мы уже набрали часть подстроки, оказавшись в состоянии v, и хотим найти наименьшее число символов, которое надо ещё добавить, чтобы выйти за пределы автомата, найдя несуществующий переход.

Считается $d[v]$ очень просто. Если из v нет перехода хотя бы по одному символу из алфавита, то $d[v] = 1$: мы можем приписать такой символ и выйти за пределы автомата, получив тем самым искомую строку.

В противном случае, одним символом обойтись не получится, поэтому надо взять минимум из ответов по всевозможным символам:

$$d[v] = 1 + \min_{\substack{w\ :\\ (v,w,c)\in DAWG}} d[w].$$

Ответ на задачу будет равен $d[t_0]$, а саму строку можно восстановить, восстановив, каким образом в динамике получился этот минимум.

Наидлиннейшая общая подстрока двух строк

Условие. Даны две строки S и T. Требуется найти их наидлиннейшую общую подстроку, т.е. такую строку X, что она является подстрокой и S, и T.

Асимптотика. Решение за $O(length(S) + length(T))$.

Решение. Построим суффиксный автомат по строке S.

Будем теперь идти по строке T, и для каждого префикса искать наидлиннейший суффикс этого префикса, встречающийся в S. Иными словами, мы для каждой позиции в строке T хотим найти наидлиннейшую общую подстроку S и T, заканчивающуюся именно в этой позиции.

Для этого будем поддерживать две переменные: **текущее состояние** v и **текущую длину** l. Эти

две переменные будут описывать текущую совпадающую часть: её длину и состояние, которое соответствует ей (без хранения длины нельзя обойтись, поскольку одному состоянию может соответствовать сразу несколько строк разной длины).

Изначально $p = t_0, l = 0$, т.е. совпадение пустое.

Пусть теперь мы рассматриваем символ $T[i]$ и хотим пересчитать ответ для него.

- Если из состояния v в автомате есть переход по символу $T[i]$, то мы просто совершаем этот переход и увеличиваем l на единицу.
- Если же из состояния v нет требуемого перехода, то мы должны попытаться укоротить текущую совпадающую часть, для чего надо перейти по суффиксной ссылке:

  $$v = link(v).$$

  При этом текущую длину надо укоротить, но оставить максимально возможной. Очевидно, для этого надо присвоить $l = len(v)$, поскольку после прохода по суффиксной ссылке нас удовлетворит подстрока любой длины, соответствующая этому состоянию:

  $$l = len(v).$$

  Если из нового состояния вновь не будет перехода по требуемому символу, то мы снова должны пройти по суффиксной ссылке и уменьшить l, и так далее, пока не найдём переход (тогда перейдём к пункту 1) или мы не попадём в фиктивное состояние -1 (что означает, что символ $T[i]$ вообще не встречается в S, поэтому присваиваем $v = l = 0$ и переходим к следующему i).

Ответом на задачу будет максимум из значений l за всё время обхода.

Асимптотика такого прохода составляет $O(length(T))$, поскольку за один ход мы можем либо увеличить на единицу l, либо сделать несколько проходов по суффиксной ссылке, каждый из которых будет строго уменьшать значение l. Следовательно, уменьшений не могло быть больше $length(T)$, что и означает линейную асимптотику.

Реализация:

```
string lcs (string s, string t) {
        sa_init();
        for (int i=0; i<(int)s.length(); ++i)
                sa_extend (s[i]);

        int v = 0,  l = 0,
                best = 0,  bestpos = 0;
        for (int i=0; i<(int)t.length(); ++i) {
                while (v && ! st[v].next.count(t[i])) {
                        v = st[v].link;
                        l = st[v].length;
                }
                if (st[v].next.count(t[i])) {
                        v = st[v].next[t[i]];
                        ++l;
                }
                if (l > best)
                        best = l,  bestpos = i;
        }
        return t.substr (bestpos-best+1, best);
}
```

Наибольшая общая подстрока нескольких строк.

Условие. Даны K строк S_i. Требуется найти их наидлиннейшую общую подстроку, т.е. такую строку X, что она является подстрокой всех S_i.

Асимптотика. Решение за $O(\sum length(S_i) \cdot K)$.

Решение. Склеим все строки S_i в одну строку T, приписав после каждой строки S_i свой собственный символ-разделитель D_i (т.е. введя K дополнительных спец. символов D_i):

$$T = S_1 \ D_1 \ S_2 \ D_2 \ \ldots \ S_k D_k.$$

Построим для строки T суффиксный автомат.

Теперь нам требуется найти такую строку в автомате, которая содержится во всех строках S_i, и в этом нам помогут добавленные спец. символы. Заметим, что если какая-либо подстрока входит в некоторую строку S_j, то

в суффиксном автомате из этой подстроки найдётся путь, содержащий символ D_j, и не содержащий остальных символов $D_1, \ldots, D_{j-1}, D_{j+1}, \ldots, D_k$.

Таким образом, нам требуется посчитать достижимости: для каждого состояния автомата и каждого символа D_i есть ли путь, содержащий разделитель D_i, и не содержащий других разделителей. Это легко сделать обходом в глубину/ширину или ленивой динамикой. После этого ответом на задачу будет строка $longest(v)$ для состояния v, из которого были найдены пути по всем символам.

Литература

Приведём сначала список первых работ, связанных с суффиксными автоматами:

- A. Blumer, J. Blumer, A. Ehrenfeucht, D. Haussler, R. McConnell. Linear Size Finite Automata for the Set of All Subwords of a Word. An Outline of Results [1983]
- A. Blumer, J. Blumer, A. Ehrenfeucht, D. Haussler. The Smallest Automaton Recognizing the Subwords of a Text [1984]
- Maxime Crochemore. Optimal Factor Transducers [1985]
- Maxime Crochemore. Transducers and Repetitions [1986]
- A. Nerode. Linear automaton transformations [1958]

Помимо этого, в более современных источниках эта тема затрагивается во многих книгах по строковым алгоритмам:

- Maxime Crochemore, Wowjcieh Rytter. Jewels of Stringology [2002]
- Bill Smyth. Computing Patterns in Strings [2003]
- Билл Смит. **Методы и алгоритмы вычислений на строках** [2006]

Нахождение всех подпалиндромов

Постановка задачи

Дана строка s длины n. Требуется найти все такие пары (i, j), где $i < j$, что подстрока $s[i \ldots j]$ является палиндромом (т.е. читается одинаково слева направо и справа налево).

Уточнение постановки

Понятно, что в худшем случае таких подстрок-палиндромов может быть $O(n^2)$, и на первый взгляд кажется, что алгоритма с линейной асимптотикой существовать не может.

Однако информацию о найденных палиндромах можно возвращать более **компактно**: для каждой позиции $i = 0 \ldots n-1$ найдём значения $d_1[i]$ и $d_2[i]$, обозначающие количество палиндромов соответственно нечётной и чётной длины с центром в позиции i.

Например, в строке $s = abababc$ есть три палиндрома нечётной длины с центром в символе $s[3] = b$, т.е. значение $d_1[3] = 3$:

$$a \overbrace{b\ a\ \underbrace{b}_{s_3}\ a\ b}^{d_1[3]=3}\ c$$

А в строке $s = cbaabd$ есть два палиндрома чётной длины с центром в символе $s[3] = a$, т.е. значение $d_2[3] = 2$:

$$c \overbrace{b\ a\ \underbrace{a}_{s_3}\ b}^{d_2[3]=2}\ d$$

Т.е. идея — в том, что если есть подпалиндром длины l с центром в какой-то позиции i, то есть также подпалиндромы длины $l-2, l-4$, и т.д. с центрами в i. Поэтому двух таких массивов $d_1[i]$ и $d_2[i]$ достаточно для хранения информации обо всех подпалиндромах этой строки.

Достаточно неожиданным фактом является то, что существует довольно простой алгоритм, который вычисляет эти "массивы палиндромностей" $d_1[]$ и $d_2[]$ за линейное время. Этот алгоритм и описывается в данной статье.

Решение

Вообще говоря, данная задача имеет несколько известных решений: с помощью техники хэширования её можно решить за $O(n \log n)$, а с помощью суффиксных деревьев и быстрого алгоритма LCA эту задачу можно решить за $O(n)$.

Однако описываемый в данной статье метод значительно проще, и обладает меньшими скрытыми константами в асимптотике времени и памяти. Этот алгоритм был открыт **Гленном Манакером** (Glenn Manacher) в 1975 г.

Тривиальный алгоритм

Во избежание неоднозначностей при дальнейшем описании условимся, что же такое есть "тривиальный алгоритм".

Это алгоритм, который для поиска ответа в позиции i раз за разом пробует увеличить ответ на единицу, каждый раз сравнивая пару соответствующих символов.

Такой алгоритм слишком медленен, весь ответ он может посчитать лишь за время $O(n^2)$.

Приведём для наглядности его реализацию:

```
vector<int> d1 (n),  d2 (n);
for (int i=0; i<n; ++i) {
        d1[i] = 1;
        while (i-d1[i] >= 0 && i+d1[i] < n && s[i-d1[i]] == s[i+d1[i]])
                ++d1[i];
```

```
        d2[i] = 0;
        while (i-d2[i]-1 >= 0 && i+d2[i] < n && s[i-d2[i]-1] == s[i+d2[i]])
                ++d2[i];
}
```

Алгоритм Манакера

Научимся сначала находить все подпалиндромы нечётной длины, т.е. вычислять массив $d_1[]$; решение для палиндромов чётной длины (т.е. нахождение массива $d_2[]$) получится небольшой модификацией этого.

Для быстрого вычисления будем поддерживать **границы** (l, r) самого правого из обнаруженных подпалиндрома (т. е. подпалиндрома с наибольшим значением r). Изначально можно положить $l = 0, r = -1$.

Итак, пусть мы хотим вычислить значение $d_1[i]$ для очередного i, при этом все предыдущие значения $d_1[]$ уже подсчитаны.

- Если i не находится в пределах текущего подпалиндрома, т.е. $i > r$, то просто выполним тривиальный алгоритм.

  Т.е. будем последовательно увеличивать значение $d_1[i]$, и проверять каждый раз — правда ли текущая подстрока $[i - d_1[i]; i + d_1[i]]$ является палиндромом. Когда мы найдём первое расхождение, либо когда мы дойдём до границ строки s — останавливаемся: мы окончательно посчитали значение $d_1[i]$. После этого мы должны не забыть обновить значения (l, r).

- Рассмотрим теперь случай, когда $i \le r$.

  Попробуем извлечь часть информации из уже подсчитанных значений $d_1[]$. А именно, отразим позицию i внутри подпалиндрома (l, r), т.е. получим позицию $j = l + (r - i)$, и рассмотрим значение $d_1[j]$. Поскольку j — позиция, симметричная позиции i, то **почти всегда** мы можем просто присвоить $d_1[i] = d_1[j]$. Иллюстрация этого отражения (палиндром вокруг j фактически "копируется" в палиндром вокруг i):

  $$\ldots \overbrace{s_l \ \ldots \ \underbrace{s_{j-d_1[j]+1} \ \ldots \ s_j \ \ldots \ s_{j+d_1[j]-1}}_{\text{palindrome}} \ \ldots \ \underbrace{s_{i-d_1[j]+1} \ \ldots \ s_i \ \ldots \ s_{i+d_1[j]-1}}_{\text{palindrome}}}^{\text{palindrome}} \ \ldots \ s_r \ \ldots$$

  Однако здесь есть **тонкость**, которую надо обработать правильно: когда "внутренний палиндром" достигает границы внешнего или вылазит за неё, т.е. $j - d_1[j] + 1 \le l$ (или, что то же самое, $i + d_1[j] - 1 \ge r$). Поскольку за границами внешнего палиндрома никакой симметрии не гарантируется, то просто присвоить $d_1[i] = d_1[j]$ будет уже некорректно: у нас недостаточно сведений, чтобы утверждать, что в позиции i подпалиндром имеет такую же длину.

  На самом деле, чтобы правильно обрабатывать такие ситуации, надо "обрезать" длину подпалиндрома, т.е. присвоить $d_1[i] = r - i$. После этого следует пустить тривиальный алгоритм, который будет пытаться увеличить значение $d_1[i]$, пока это возможно.

  Иллюстрация этого случая (на ней палиндром с центром в j изображён уже "обрезанным" до такой длины, что он впритык помещается во внешний палиндром):

  $$\ldots \overbrace{\underbrace{s_l \ \ldots \ s_j \ \ldots \ s_{j+(j-l)}}_{\text{palindrome}} \ \ldots \ \underbrace{s_{i-(r-i)} \ \ldots \ s_i \ \ldots \ s_r}_{\text{palindrome}}}^{\text{palindrome}} \ \underbrace{\ldots\ldots\ldots\ldots}_{\text{try moving here}}$$

  (На этой иллюстрации показано, что, хотя палиндром с центром в позиции j мог быть и более длинным, выходящим за пределы внешнего палиндрома, — но в позиции i мы можем использовать только ту его часть, которая целиком помещается во внешний палиндром. Но ответ для позиции i может быть больше, чем эта часть, поэтому дальше мы должны запустить тривиальный поиск, который будет пытаться раздвинуть его за пределы внешнего палиндрома, т.е. в область "try moving here".)

В завершение описания алгоритма сталось только напомнить, что надо не забывать обновлять значения (l, r) после вычисления очередного значения $d_1[i]$.

Также повторимся, что выше мы описали рассуждения для вычисления массива нечётных палиндромов $d_1[]$; для массива чётных палиндромов $d_2[]$ все рассуждения аналогичны.

Оценка асимптотики алгоритма Манакера

На первый взгляд не очевидно, что данный алгоритм имеет линейную асимптотику: при вычислении ответа для определённой позиции в нём нередко запускается тривиальный алгоритм поиска палиндромов.

Однако более внимательный анализ показывает, что алгоритм всё же линеен. (Стоит сослаться на известный

алгоритм построения Z-функции строки, который внутренне сильно напоминает данный алгоритм, и работает также за линейное время.)

В самом деле, легко проследить по алгоритму, что каждая итерация, производимая тривиальным поиском, приводит к увеличению на один границы r^*. При этом уменьшений r^* по ходу алгоритма происходить не может. Следовательно, тривиальный алгоритм в сумме совершит лишь $O(n)$ действий.

Учитывая, что, кроме тривиальных поисков, все остальные части алгоритма Манакера очевидно работают за линейное время, мы и получаем итоговую асимптотику: $O(n)$.

Реализация алгоритма Манакера

Для случая подпалиндромов нечётной длины, т.е. для вычисления массива $d_1[]$, получаем такой код:

```
vector<int> d1 (n);
int l=0, r=-1;
for (int i=0; i<n; ++i) {
        int k = (i>r ? 0 : min (d1[l+r-i], r-i)) + 1;
        while (i+k < n && i-k >= 0 && s[i+k] == s[i-k])  ++k;
        d1[i] = k--;
        if (i+k > r)
                l = i-k,  r = i+k;
}
```

Для подпалиндромов чётной длины, т.е. для вычисления массива $d_2[]$, лишь немного меняются арифметические выражения:

```
vector<int> d2 (n);
l=0, r=-1;
for (int i=0; i<n; ++i) {
        int k = (i>r ? 0 : min (d2[l+r-i+1], r-i+1)) + 1;
        while (i+k-1 < n && i-k >= 0 && s[i+k-1] == s[i-k])  ++k;
        d2[i] = --k;
        if (i+k-1 > r)
                l = i-k,  r = i+k-1;
}
```

Задачи в online judges

Список задач, которые можно сдать с использованием этого алгоритма:

- UVA #11475 "Extend to Palindrome" [сложность: низкая]

Декомпозиция Линдона. Алгоритм Дюваля. Нахождение наименьшего циклического сдвига

Понятие декомпозиции Линдона

Определим понятие **декомпозиции Линдона** (Lyndon decomposition).

Строка называется **простой**, если она строго **меньше** любого своего собственного **суффикса**. Примеры простых строк: a, b, ab, aab, abb, $ababb$, $abcd$. Можно показать, что строка является простой тогда и только тогда, когда она строго **меньше** всех своих нетривиальных **циклических сдвигов**.

Далее, пусть дана строка s. Тогда **декомпозицией Линдона** строки s называется её разложение $s = w_1 w_2 \dots w_k$, где строки w_i просты, и при этом $w_1 \geq w_2 \geq \dots \geq w_k$.

Можно показать, что для любой строки s это разложение существует и единственно.

Алгоритм Дюваля

Алгоритм Дюваля (Duval's algorithm) находит для данной строки длины n декомпозицию Линдона за время $O(n)$ с использованием $O(1)$ дополнительной памяти.

Работать со строками будем в 0-индексации.

Введём вспомогательное понятие предпростой строки. Строка t называется **предпростой**, если она имеет вид $t = www \dots w\overline{w}$, где w — некоторая простая строка, а $\overline{w}$ — некоторый префикс строки w.

Алгоритм Дюваля является жадным. В любой момент его работы строка S фактически разделена на три строки $s = s_1 s_2 s_3$, где в строке s_1 декомпозиция Линдона уже найдена и s_1 уже больше не используется алгоритмом; строка s_2 — это предпростая строка (причём длину простых строк внутри неё мы также запоминаем); строка s_3 — это ещё не обработанная часть строки s. Каждый раз алгоритм Дюваля берёт первый символ строки s_3 и пытается дописать его к строке s_2. При этом, возможно, для какого-то префикса строки s_2 декомпозиция Линдона становится известной, и эта часть переходит к строке s_1.

Опишем теперь алгоритм **формально**. Во-первых, будет поддерживаться указатель i на начало строки s_2. Внешний цикл алгоритма будет выполняться, пока $i < n$, т.е. пока вся строка s не перейдёт в строку s_1. Внутри этого цикла создаются два указателя: указатель j на начало строки s_3 (фактически указатель на следующий символ-кандидат) и указатель k на текущий символ в строке s_2, с которым будет производиться сравнение. Затем будем в цикле пытаться добавить символ $s[j]$ к строке s_2, для чего необходимо произвести сравнение с символом $s[k]$. Здесь у нас возникают три различных случая:

- Если $s[j] = s[k]$, то мы можем дописать символ $s[j]$ к строке s_2, не нарушив её "предпростоты". Следовательно, в этом случае мы просто увеличиваем указатели j и k на единицу.
- Если $s[j] > s[k]$, то, очевидно, строка $s_2 + s[j]$ станет простой. Тогда мы увеличиваем j на единицу, а k передвигаем обратно на i, чтобы следующий символ сравнивался с первым символом s_2.
- Если $s[j] < s[k]$, то строка $s_2 + s[j]$ уже не может быть предпростой. Поэтому мы разбиваем предпростую строку s_2 на простые строки плюс "остаток" (префикс простой строки, возможно, пустой); простые строки добавляем в ответ (т. е. выводим их позиции, попутно передвигая указатель i), а "остаток" вместе с символом $s[j]$ переводим обратно в строку s_3, и останавливаем выполнение внутреннего цикла. Тем самым мы на следующей итерации внешнего цикла заново обработаем остаток, зная, что он не мог образовать предпростую строку с предыдущими простыми строками. Осталось только заметить, что при выводе позиций простых строк нам нужно знать их длину; но она, очевидно, равна $j - k$.

Реализация

Приведём реализацию алгоритма Дюваля, которая будет выводить искомую декомпозицию Линдона строки s:

```
string s; // входная строка
int n = (int) s.length();
int i=0;
while (i < n) {
```

```
        int j=i+1, k=i;
        while (j < n && s[k] <= s[j]) {
                if (s[k] < s[j])
                        k = i;
                else
                        ++k;
                ++j;
        }
        while (i <= k) {
                cout << s.substr (i, j-k) << ' ';
                i += j - k;
        }
}
```

Асимптотика

Сразу заметим, что для алгоритма Дюваля требуется $O(1)$ **памяти**, а именно три указателя i, j, k.

Оценим теперь **время работы** алгоритма.

Внешний цикл while делает не более n итераций, поскольку в конце каждой его итерации выводится как минимум один символ (а всего символов выводится, очевидно, ровно n).

Оценим теперь количество итераций **первого вложенного цикла** while. Для этого рассмотрим второй вложенный цикл while — он при каждом своём запуске выводит некоторое количество $r \geq 1$ копий одной и той же простой строки некоторой длины $p = j - k$. Заметим, что строка s_2 является предпростой, причём её простые строки имеют длину как раз p, т.е. её длина не превосходит $rp + p - 1$. Поскольку длина строки s_2 равна $j - i$, а указатель j увеличивается по единице на каждой итерации первого вложенного цикла while, то этот цикл выполнит не более $rp + p - 2$ итераций. Худшим случаем является случай $r = 1$, и мы получаем, что первый вложенный цикл while всякий раз выполняет не более $2p - 2$ итераций. Вспоминая, что всего выводится n символов, получаем, что для вывода n символов требуется не более $2n - 2$ итераций первого вложенного while-а.

Следовательно, **алгоритм Дюваля выполняется за** $O(n)$.

Легко оценить и число сравнений символов, выполняемых алгоритмом Дюваля. Поскольку каждая итерация первого вложенного цикла while производит два сравнения символов, а также одно сравнение производится после последней итерации цикла (чтобы понять, что цикл должен остановиться), то общее **число сравнений символов** не превосходит $4n - 3$.

Нахождение наименьшего циклического сдвига

Пусть дана строка s. Построим для строки $s + s$ декомпозицию Линдона (мы можем это сделать за $O(n)$ времени и $O(1)$ памяти (если не выполнять конкатенацию в явном виде)). Найдём предпростой блок, который начинается в позиции, меньшей n (т.е. в первом экземпляре строки s), и заканчивается в позиции, большей или равной n (т.е. во втором экземпляре). Утверждается, что **позиция начала** этого блока и будет началом искомого циклического сдвига (в этом легко убедиться, воспользовавшись определением декомпозиции Линдона).

Начало предпростого блока найти просто — достаточно заметить, что указатель i в начале каждой итерации внешнего цикла while указывает на начало текущего предпростого блока.

Итого мы получаем такую **реализацию** (для упрощения кода она использует $O(n)$ памяти, явным образом дописывая строку к себе):

```
string min_cyclic_shift (string s) {
        s += s;
        int n = (int) s.length();
        int i=0, ans=0;
        while (i < n/2) {
                ans = i;
                int j=i+1, k=i;
                while (j < n && s[k] <= s[j]) {
                        if (s[k] < s[j])
                                k = i;
                        else
                                ++k;
                        ++j;
                }
                while (i <= k)  i += j - k;
        }
```

```
        return s.substr (ans, n/2);
}
```

Задачи в online judges

Список задач, которые можно решить, используя алгоритм Дюваля:

- UVA #719 "Glass Beads" [сложность: низкая]

Алгоритм Ахо-Корасик

Пусть дан набор строк в алфавите размера k суммарной длины m. Алгоритм Ахо-Корасик строит для этого набора строк структуру данных "бор", а затем по этому бору строит автомат, всё за $O(m)$ времени и $O(mk)$ памяти. Полученный автомат уже может использоваться в различных задачах, простейшая из которых — это нахождение всех вхождений каждой строки из данного набора в некоторый текст за линейное время.

Данный алгоритм был предложен канадским учёным Альфредом Ахо (Alfred Vaino Aho) и учёным Маргарет Корасик (Margaret John Corasick) в 1975 г.

Бор. Построение бора

Формально, **бор** — это дерево с корнем в некоторой вершине Root, причём каждое ребро дерево подписано некоторой буквой. Если мы рассмотрим список рёбер, выходящих из данной вершины (кроме ребра, ведущего в предка), то все рёбра должны иметь разные метки.

Рассмотрим в боре любой путь из корня; выпишем подряд метки рёбер этого пути. В результате мы получим некоторую строку, которая соответствует этому пути. Если же мы рассмотрим любую вершину бора, то ей поставим в соответствие строку, соответствующую пути из корня до этой вершины.

Каждая вершина бора также имеет флаг leaf, который равен true, если в этой вершине оканчивается какая-либо строка из данного набора.

Соответственно, **построить бор** по данному набору строк — значит построить такой бор, что каждой leaf-вершине будет соответствовать какая-либо строка из набора, и, наоборот, каждой строке из набора будет соответствовать какая-то leaf-вершина.

Опишем теперь, **как построить бор** по заданному набору строк за линейное время относительно их суммарной длины.

Введём структуру, соответствующую вершинам бора:

```
struct vertex {
        int next[K];
        bool leaf;
};

vertex t[NMAX+1];
int sz;
```

Т.е. мы будем хранить бор в виде массива t (количество элементов в массиве - это sz) структур vertex. Структура vertex содержит флаг leaf, и рёбра в виде массива $\mathrm{next}[]$, где $\mathrm{next}[\mathrm{i}]$ — указатель на вершину, в которую ведёт ребро по символу i, или -1, если такого ребра нет.

Вначале бор состоит только из одной вершины — корня (договоримся, что корень всегда имеет в массиве t индекс 0). Поэтому **инициализация** бора такова:

```
memset (t[0].next, 255, sizeof t[0].next);
sz = 1;
```

Теперь реализуем функцию, которая будет **добавлять в бор** заданную строку s. Реализация крайне проста: мы встаём в корень бора, смотрим, есть ли из корня переход по букве $s[0]$: если переход есть, то просто переходим по нему в другую вершину, иначе создаём новую вершину и добавляем переход в эту вершину по букве $s[0]$. Затем мы, стоя в какой-то вершине, повторяем процесс для буквы $s[1]$, и т.д. После окончания процесса помечаем последнюю посещённую вершину флагом $\mathrm{leaf} = \mathrm{true}$.

```
void add_string (const string & s) {
        int v = 0;
        for (size_t i=0; i<s.length(); ++i) {
                char c = s[i]-'a'; // в зависимости от алфавита
                if (t[v].next[c] == -1) {
                        memset (t[sz].next, 255, sizeof t[sz].next);
                        t[v].next[c] = sz++;
                }
                v = t[v].next[c];
```

```
        }
        t[v].leaf = true;
}
```

Линейное время работы, а также линейное количество вершин в боре очевидны. Поскольку на каждую вершину приходится $O(k)$ памяти, то использование памяти есть $O(nk)$.

Потребление памяти можно уменьшить до линейного ($O(n)$), но за счёт увеличения асимптотики работы до $O(n \log k)$. Для этого достаточно хранить переходы next не массивом, а отображением $\mathrm{map} < \mathrm{char}, \mathrm{int} >$.

Построение автомата

Пусть мы построили бор для заданного набора строк. Посмотрим на него теперь немного с другой стороны. Если мы рассмотрим любую вершину, то строка, которая соответствует ей, является префиксом одной или нескольких строк из набора; т.е. каждую вершину бора можно понимать как позицию в одной или нескольких строках из набора.

Фактически, вершины бора можно понимать как состояния **конечного детерминированного автомата**. Находясь в каком-либо состоянии, мы под воздействием какой-то входной буквы переходим в другое состояние — т.е. в другую позицию в наборе строк. Например, если в боре находится только строка "abc" и мы стоим в состоянии 2 (которому соответствует строка "ab"), то под воздействием буквы "c" мы перейдём в состояние 3.

Т.е. мы можем понимать рёбра бора как переходы в автомате по соответствующей букве. Однако одними только рёбрами бора нельзя ограничиваться. Если мы пытаемся выполнить переход по какой-либо букве, а соответствующего ребра в боре нет, то мы тем не менее должны перейти в какое-то состояние.

Более строго, пусть мы находимся в состоянии p, которому соответствует некоторая строка t, и хотим выполнить переход по символу c. Если в боре из вершины p есть переход по букве c, то мы просто переходим по этому ребру и попадаем в вершину, которой соответствует строка tc. Если же такого ребра нет, то мы должны найти состояние, соответствующее наидлиннейшему собственному суффиксу строки t (наидлиннейшему из имеющихся в боре), и попытаться выполнить переход по букве c из него.

Например, пусть бор построен по строкам "ab" и "bc", и мы под воздействием строки "ab" перешли в некоторое состояние, являющееся листом. Тогда под воздействием буквы "c" мы вынуждены перейти в состояние, соответствующее строке "b", и только оттуда выполнить переход по букве "c".

Суффиксная ссылка для каждой вершины p — это вершина, в которой оканчивается наидлиннейший собственный суффикс строки, соответствующей вершине p. Единственный особый случай — корень бора; для удобства суффиксную ссылку из него проведём в себя же. Теперь мы можем переформулировать утверждение по поводу переходов в автомате так: пока из текущей вершины бора нет перехода по соответствующей букве (или пока мы не придём в корень бора), мы должны переходить по суффиксной ссылке.

Таким образом, мы свели задачу построения автомата к задаче нахождения суффиксных ссылок для всех вершин бора. Однако строить эти суффиксные ссылки мы будем, как ни странно, наоборот, с помощью построенных в автомате переходов.

Заметим, что если мы хотим узнать суффиксную ссылку для некоторой вершины v, то мы можем перейти в предка p текущей вершины (пусть c — буква, по которой из p есть переход в v), затем перейти по его суффиксной ссылке, а затем из неё выполнить переход в автомате по букве c.

Таким образом, задача нахождения перехода свелась к задаче нахождения суффиксной ссылки, а задача нахождения суффиксной ссылки — к задаче нахождения суффиксной ссылки и перехода, но уже для более близких к корню вершин. Мы получили рекурсивную зависимость, но не бесконечную, и, более того, разрешить которую можно за линейное время.

Перейдём теперь к **реализации**. Заметим, что нам теперь понадобится для каждой вершины хранить её предка p, а также символ pch, по которому из предка есть переход в нашу вершину. Также в каждой вершине будем хранить $\mathrm{int\ link}$ — суффиксная ссылка (или -1, если она ещё не вычислена), и массив $\mathrm{int\ go[k]}$ — переходы в автомате по каждому из символов (опять же, если элемент массива равен -1, то он ещё не вычислен). Приведём теперь полную реализацию всех необходимых функций:

```
struct vertex {
        int next[K];
        bool leaf;
        int p;
        char pch;
        int link;
        int go[K];
};

vertex t[NMAX+1];
int sz;
```

```
void init() {
        t[0].p = t[0].link = -1;
        memset (t[0].next, 255, sizeof t[0].next);
        memset (t[0].go, 255, sizeof t[0].go);
        sz = 1;
}

void add_string (const string & s) {
        int v = 0;
        for (size_t i=0; i<s.length(); ++i) {
                char c = s[i]-'a';
                if (t[v].next[c] == -1) {
                        memset (t[sz].next, 255, sizeof t[sz].next);
                        memset (t[sz].go, 255, sizeof t[sz].go);
                        t[sz].link = -1;
                        t[sz].p = v;
                        t[sz].pch = c;
                        t[v].next[c] = sz++;
                }
                v = t[v].next[c];
        }
        t[v].leaf = true;
}

int go (int v, char c);

int get_link (int v) {
        if (t[v].link == -1)
                if (v == 0 || t[v].p == 0)
                        t[v].link = 0;
                else
                        t[v].link = go (get_link (t[v].p), t[v].pch);
        return t[v].link;
}

int go (int v, char c) {
        if (t[v].go[c] == -1)
                if (t[v].next[c] != -1)
                        t[v].go[c] = t[v].next[c];
                else
                        t[v].go[c] = v==0 ? 0 : go (get_link (v), c);
        return t[v].go[c];
}
```

Нетрудно понять, что, за счёт запоминания найденных суффиксных ссылок и переходов, суммарное время нахождения всех суффиксных ссылок и переходов будет линейным.

Применения

Поиск всех строк из заданного набора в тексте

Дан набор строк, и дан текст. Требуется вывести все вхождения всех строк из набора в данный текст за время $O(\mathrm{Len} + \mathrm{Ans})$, где Len — длина текста, Ans — размер ответа.

Построим по данному набору строк бор. Будем теперь обрабатывать текст по одной букве, перемещаясь соответствующим образом по дереву, фактически — по состояниям автомата. Изначально мы находимся в корне дерева. Пусть мы на очередном шаге мы находимся в состоянии v, и очередная буква текста c. Тогда следует переходить в состояние $\mathrm{go}(v, c)$, тем самым либо увеличивая на 1 длину текущей совпадающей подстроки, либо уменьшая её, проходя по суффиксной ссылке.

Как теперь узнать по текущему состоянию v, имеется ли совпадение с какими-то строками из набора? Во-первых, понятно, что если мы стоим в помеченной вершине ($\mathrm{leaf} = \mathrm{true}$), то имеется совпадение с тем образцом, который в боре оканчивается в вершине v. Однако это далеко не единственный возможный случай достижения совпадения: если мы, двигаясь по суффиксным ссылкам, мы можем достигнуть одной или нескольких помеченных вершин, то совпадение также будет, но уже для образцов, оканчивающихся в этих состояниях. Простой пример такой ситуации — когда набор строк — это $\{"dabce", "abc", "bc"\}$, а текст — это $"dabc"$.

Таким образом, если в каждой помеченной вершине хранить номер образца, оканчивающегося в ней (или список номеров, если допускаются повторяющиеся образцы), то мы можем для текущего состояния за $O(n)$ найти номера всех образцов, для которых достигнуто совпадение, просто пройдя по суффиксным ссылкам от текущей вершины до корня. Однако это недостаточно эффективное решение, поскольку в сумме асимптотика получится $O(n \cdot \mathrm{Len})$. Однако можно заметить, что движение по суффиксным ссылкам можно соптимизировать, предварительно посчитав для каждой вершины ближайшую к ней помеченную вершину, достижимую по суффиксным ссылкам (это называется "функцией выхода"). Эту величину можно считать ленивой динамикой за линейное время. Тогда для текущей вершины мы сможем за $O(1)$ находить следующую в суффиксном пути помеченную вершину, т.е. следующее совпадение. Тем самым, на каждое совпадение будет тратиться $O(1)$ действий, и в сумме получится асимптотика $O(\mathrm{Len} + \mathrm{Ans})$.

В более простом случае, когда надо найти не сами вхождения, а только их количество, можно вместо функции выхода посчитать ленивой динамикой количество помеченных вершин, достижимых из текущей вершины v по суффиксным ссылкам. Эта величина может быть посчитана за $O(n)$ в сумме, и тогда для текущего состояния v мы сможем за $O(1)$ найти количество вхождений всех образцов в текст, оканчивающихся в текущей позиции. Тем самым, задача нахождения суммарного количества вхождений может быть решена нами за $O(\mathrm{Len})$.

Нахождение лексикографически наименьшей строки данной длины, не содержащей ни один из данных образцов

Дан набор образцов, и дана длина L. Требуется найти строку длины L, не содержащую ни один из образцов, и из всех таких строк вывести лексикографически наименьшую.

Построим по данному набору строк бор. Вспомним теперь, что те вершины, из которых по суффиксным ссылкам можно достичь помеченных вершин (а такие вершины можно найти за $O(n)$, например, ленивой динамикой), можно воспринимать как вхождение какой-либо строки из набора в заданный текст. Поскольку в данной задаче нам необходимо избегать вхождений, то это можно понимать как то, что в такие вершины нам заходить нельзя. С другой стороны, во все остальные вершины мы заходить можем. Таким образом, мы удаляем из автомата все "плохие" вершины, а в оставшемся графе автомата требуется найти лексикографически наименьший путь длины L. Эту задачу уже можно решить за $O(L)$, например, поиском в глубину.

Нахождение кратчайшей строки, содержащей вхождения одновременно всех образцов

Снова воспользуемся той же идеей. Для каждой вершины будем хранить маску, обозначающую образцы, для которых произошло вхождение в данной вершине. Тогда задачу можно переформулировать так: изначально находясь в состоянии $(v = \mathrm{Root},\ \mathrm{Msk} = 0)$, требуется дойти до состояния $(v,\ \mathrm{Msk} = 2^n - 1)$, где n — количество образцов. Переходы из состояния в состояние будут представлять собой добавление одной буквы к тексту, т. е. переход по ребру автомата в другую вершину с соответствующим изменением маски. Запустив обход в ширину на таком графе, мы найдём путь до состояния $(v,\ \mathrm{Msk} = 2^n - 1)$ наименьшей длины, что нам как раз и требовалось.

Нахождение лексикографически наименьшей строки длины L, содержащей данные образцы в сумме k раз

Как и в предыдущих задачах, посчитаем для каждой вершины количество вхождений, которое соответствует ей (т. е. количество помеченных вершин, достижимых из неё по суффиксным ссылкам). Переформулируем задачу таким образом: текущее состояние определяется тройкой чисел $(v,\ \mathrm{Len},\ \mathrm{Cnt})$, и требуется из состояния $(\mathrm{Root},\ 0,\ 0)$ прийти в состояние $(v,\ L,\ k)$, где v — любая вершина. Переходы между состояниями — это просто переходы по рёбрам автомата из текущей вершины. Таким образом, достаточно просто найти обходом в глубину путь между этими двумя состояниями (если обход в глубину будет просматривать буквы в их естественном порядке, то найденный путь автоматически будет лексикографически наименьшим).

Задачи в online judges

Задачи, которые можно решить, используя бор или алгоритм Ахо-Корасик:

- UVA #11590 "Prefix Lookup" [сложность: низкая]
- UVA #11171 "SMS" [сложность: средняя]
- UVA #10679 "I Love Strings!!!" [сложность: средняя]

Суффиксное дерево. Алгоритм Укконена

Эта статья — временная заглушка, и не содержит никаких описаний.

Описание алгоритма Укконена можно найти, например, в книге Гасфилда "Строки, деревья и последовательности в алгоритмах".

Реализация алгоритма Укконена

Этот алгоритм строит суффиксное дерево для данной строки длины n за время $O(n \log k)$, где k — размер алфавита (если его считать константой, то асимптотика получается $O(n)$).

Входными данными для алгоритма являются строка s и её длина n, которые передаются в виде глобальных переменных.

Основная функция — build_tree, она строит суффиксное дерево. Дерево хранится в виде массива структур node, где $\text{node}[0]$ — корень суффиксного дерева.

В целях простоты кода рёбра хранятся в тех же самых структурах: для каждой вершины в её структуре node записаны данные о ребре, входящем в неё из предка. Итого, в каждой node хранятся: (l, r), определяющие метку $s[l..r-1]$ ребра в предка, par — вершина-предок, link — суффиксная ссылка, next — список исходящих рёбер.

```
string s;
int n;

struct node {
        int l, r, par, link;
        map<char,int> next;

        node (int l=0, int r=0, int par=-1)
                : l(l), r(r), par(par), link(-1) {}
        int len()  {  return r - l;  }
        int &get (char c) {
                if (!next.count(c))  next[c] = -1;
                return next[c];
        }
};
node t[MAXN];
int sz;

struct state {
        int v, pos;
        state (int v, int pos) : v(v), pos(pos)  {}
};
state ptr (0, 0);

state go (state st, int l, int r) {
        while (l < r)
                if (st.pos == t[st.v].len()) {
                        st = state (t[st.v].get( s[l] ), 0);
                        if (st.v == -1)  return st;
                }
                else {
                        if (s[ t[st.v].l + st.pos ] != s[l])
                                return state (-1, -1);
                        if (r-l < t[st.v].len() - st.pos)
                                return state (st.v, st.pos + r-l);
                        l += t[st.v].len() - st.pos;
                        st.pos = t[st.v].len();
                }
        return st;
}

int split (state st) {
        if (st.pos == t[st.v].len())
                return st.v;
```

```
        if (st.pos == 0)
                return t[st.v].par;
        node v = t[st.v];
        int id = sz++;
        t[id] = node (v.l, v.l+st.pos, v.par);
        t[v.par].get( s[v.l] ) = id;
        t[id].get( s[v.l+st.pos] ) = st.v;
        t[st.v].par = id;
        t[st.v].l += st.pos;
        return id;
}

int get_link (int v) {
        if (t[v].link != -1)  return t[v].link;
        if (t[v].par == -1)  return 0;
        int to = get_link (t[v].par);
        return t[v].link = split (go (state(to,t[to].len()), t[v].l + (t
[v].par==0), t[v].r));
}

void tree_extend (int pos) {
        for(;;) {
                state nptr = go (ptr, pos, pos+1);
                if (nptr.v != -1) {
                        ptr = nptr;
                        return;
                }

                int mid = split (ptr);
                int leaf = sz++;
                t[leaf] = node (pos, n, mid);
                t[mid].get( s[pos] ) = leaf;

                ptr.v = get_link (mid);
                ptr.pos = t[ptr.v].len();
                if (!mid)  break;
        }
}

void build_tree() {
        sz = 1;
        for (int i=0; i<n; ++i)
                tree_extend (i);
}
```

Сжатая реализация

Приведём также следующую компактную реализацию алгоритма Укконена, предложенную freopen:

```
const int N=1000000,INF=1000000000;
string a;
int t[N][26],l[N],r[N],p[N],s[N],tv,tp,ts=2;

void ukkadd (int c) {
        suff:;
        if (r[tv]<tp) {
                if (t[tv][c]==-1) { t[tv][c]=ts;  l[ts]=a.size()-1;  r[ts]=INF;
                        p[ts++]=tv;  tv=s[tv];  tp=r[tv]+1;  goto suff; }
                tv=t[tv][c]; tp=l[tv];
        }
        if (tp==-1 || c==a[tp]) tp++; else {
                l[ts+1]=a.size()-1;  r[ts+1]=INF;  p[ts+1]=ts;
                l[ts]=l[tv];  r[ts]=tp-1;  p[ts]=p[tv];  t[ts][c]=ts+1;  t[ts]
[a[tp]]=tv;
                l[tv]=tp;  p[tv]=ts;  t[p[ts]][a[l[ts]]]=ts;  ts+=2;
                tv=s[p[ts-2]];  tp=l[ts-2];
                while (tp<=r[ts-2]) {  tv=t[tv][a[tp]];  tp+=r[tv]-l[tv]+1;}
```

```
                if (tp==r[ts-2]+1)  s[ts-2]=tv;  else s[ts-2]=ts;
                tp=r[tv]-(tp-r[ts-2])+2;  goto suff;
        }
}

void build() {
        fill(r,r+N,INF);
        s[0]=1;
        l[0]=-1;
        r[0]=-1;
        l[1]=-1;
        r[1]=-1;
        memset (t, -1, sizeof t);
        fill(t[1],t[1]+26,0);
        for (int i=0; i<a.size(); ++i)
                ukkadd (a[i]-'a');
}
```

Тот же самый код, прокомментированный:

```
const int N=1000000,    // максимальное число вершин в суффиксном дереве
        INF=1000000000; // константа "бесконечность"
string a;       // входная строка, для которой надо построить дерево
int t[N][26],   // массив переходов (состояние, буква)
        l[N],   // левая
        r[N],   // и правая границы подстроки из a, отвечающие ребру,
входящему в вершину
        p[N],   // предок вершины
        s[N],   // суффиксная ссылка
        tv=0,   // вершина текущего суффикса (если мы посередине ребра,
то нижняя вершина ребра)
        tp=0,   // положение в строке соответствующее месту на ребре (от l
[tv] до r[tv] включительно)
        ts=2;   // количество вершин

void ukkadd(int c) { // дописать к дереву символ c
        suff:;      // будем приходить сюда после каждого перехода к
суффиксу (и заново добавлять символ)
        if (r[tv]<tp) { // проверим, не вылезли ли мы за пределы текущего ребра
                // если вылезли, найдем следующее ребро. Если его нет
- создадим лист и прицепим к дереву
                if (t[tv][c]==-1) {t[tv][c]=ts;l[ts]=la-1;p[ts++]=tv;tv=s
[tv];tp=r[tv]+1;goto suff;}
                tv=t[tv][c];tp=l[tv]; // в противном случае просто перейдем
к следующему ребру
        }
        if (tp==-1 || c==a[tp]) tp++; else { // если буква на ребре совпадает
с c то идем по ребру, а иначе
                // разделяем ребро на два. Посередине - вершина ts
                l[ts]=l[tv];r[ts]=tp-1;p[ts]=p[tv];t[ts][a[tp]]=tv;
                // ставим лист ts+1. Он соответствует переходу по c.
                t[ts][c]=ts+1;l[ts+1]=la-1;p[ts+1]=ts;
                // обновляем параметры текущей вершины. Не забыть про
переход от предка tv к ts.
                l[tv]=tp;p[tv]=ts;t[p[ts]][a[l[ts]]]=ts;ts+=2;
                // готовимся к спуску: поднялись на ребро и перешли
по суффиксной ссылке.
                // tp будет отмечать, где мы в текущем суффиксе.
                tv=s[p[ts-2]];tp=l[ts-2];
                // пока текущий суффикс не кончился, топаем вниз
                while (tp<=r[ts-2]) {tv=t[tv][a[tp]];tp+=r[tv]-l[tv]+1;}
                // если мы пришли в вершину, то поставим в нее
суффиксную ссылку, иначе поставим в ts
                // (ведь на след. итерации мы создадим ts).
                if (tp==r[ts-2]+1) s[ts-2]=tv; else s[ts-2]=ts;
                // устанавливаем tp на новое ребро и идем добавлять букву
к суффиксу.
                tp=r[tv]-(tp-r[ts-2])+2;goto suff;
        }
}
```

```
void build() {
        fill(r,r+N,INF);
        // инициализируем данные для корня дерева
        s[0]=1;
        l[0]=-1;
        r[0]=-1;
        l[1]=-1;
        r[1]=-1;
        memset (t, -1, sizeof t);
        fill(t[1],t[1]+26,0);
        // добавляем текст в дерево по одной букве
        for (int i=0; i<a.size(); ++i)
                ukkadd (a[i]-'a');
}
```

Задачи в online judges

Задачи, которые можно решить, используя суффиксное дерево:

- UVA #10679 "I Love Strings!!!" [сложность: средняя]

char,int> next;

node (int l=0, int r=0, int par=-1) : l(l), r(r), par(par), link(-1) {} int len() { return r - l; } int

Поиск всех тандемных повторов в строке. Алгоритм Мейна-Лоренца

Дана строка s длины n.

Тандемным повтором (tandem repeat) в ней называются два вхождения какой-либо подстроки подряд. Иными словами, тандемный повтор описывается парой индексов $i < j$ такими, что подстрока $s[i \dots j]$ — это две одинаковые строки, записанные подряд.

Задача заключается в том, чтобы **найти все тандемные повторы**. Упрощённые варианты этой задачи: найти **любой** тандемный повтор или найти **длиннейший** тандемный повтор.

Примечание. Во избежание путаницы все строки в статье мы будем считать 0-индексированными, т.е. первый символ строки имеет индекс 0.

Описываемый здесь алгоритм был опубликован в 1982 г. Мейном и Лоренцем (см. список литературы).

Пример

Рассмотрим тандемные повторы на примере какой-нибудь простой строки, например:

$$"acababaee"$$

В этой строке присутствуют следующие тандемные повторы:

- $[2; 5] = "abab"$
- $[3; 6] = "baba"$
- $[7; 8] = "ee"$

Другой пример:

$$"abaaba"$$

Здесь есть только два тандемных повтора:

- $[0; 5] = "abaaba"$
- $[2; 3] = "aa"$

Число тандемных повторов

Вообще говоря, тандемных повторов в строке длины n может быть порядка $O(n^2)$.

Очевидным примером является строка, составленная из n одинаковых букв — в такой строке тандемным повтором является любая подстрока чётной длины, которых примерно $n^2/4$. Вообще, любая периодичная строка с коротким периодом будет содержать очень много тандемных повторов

С другой стороны, сам по себе этот факт никак не препятствует существованию алгоритма с асимптотикой $O(n \log n)$, поскольку алгоритм может выдавать тандемные повторы в том или ином сжатом виде, группами по несколько штук сразу.

Более того, существует понятие **серий** — четвёрок чисел, которые описывают целую группу периодических подстрок. Было доказано, что число серий в любой строке линейно по отношению к длине строки.

Впрочем, описываемый ниже алгоритм не использует понятие серий, поэтому не будем детализировать это понятие.

Приведём здесь и другие интересные результаты, относящиеся к количеству тандемных повторов:

- Известно, что если рассматривать только примитивные тандемные повторы (т.е. такие, половинки которых не являются кратными строками), то их количество в любой строке — $O(n \log n)$.
- Если кодировать тандемные повторы тройками чисел (называемыми тройками Крочемора (Crochemore)) (i, p, r) (где i — позиция начала, p — длина повторяющейся подстроки, r — количество повторов), то все тандемные повторы любой строки можно вывести с помощью $O(n \log n)$ таких троек. (Именно такой результат получается на выходе алгоритма Крочемора нахождения всех тандемных повторов.)
- Строки Фибоначчи, определяемые следующим образом:

  $$t_0 = b,$$

$$t_1 = a,$$
$$t_i = t_{i-1} + t_{i-2},$$

являются "сильно" периодичными.

Число тандемных повторов в i-ой строке Фибоначчи длины f_i, даже сжатых с помощью троек Крочемора, составляет $O(f_n \log f_n)$.

Число примитивных тандемных повторов в строках Фибоначчи — также имеет порядок $O(f_n \log f_n)$.

Алгоритм Мейна-Лоренца

Идея алгоритма Мейна-Лоренца довольно стандартна: это алгоритм "**разделяй-и-властвуй**".

Кратко он заключается в том, что исходная строка разбивается пополам, решение запускается от каждой из двух половинок по отдельности (тем самым мы найдём все тандемные повторы, располагающиеся только в первой или только во второй половинке). Дальше идёт самая сложная часть — это нахождение тандемных повторов, начинающихся в первой половине и заканчивающихся во второй (назовём такие тандемные повторы для удобства **пересекающими**). Как именно это делается — и есть сама суть алгоритма Мейна-Лоренца; это мы подробно опишем ниже.

Асимптотика алгоритма "разделяй-и-властвуй" хорошо исследована. В частности, для нас важно, что если мы научимся искать пересекающие тандемные повторы в строке длины n за $O(n)$, то итоговая асимптотика всего алгоритма получится $O(n \log n)$.

Поиск пересекающих тандемных повторов

Итак, алгоритм Мейна-Лоренца свёлся к тому, чтобы по заданной строке s научиться искать все пересекающие тандемные повторы, т.е. такие, которые начинаются в первой половине строки, а заканчиваются — во второй.

Обозначим через u и v две половинки строки s:

$$s = u + v$$

(их длины примерно равны длине строки s, делённой пополам).

Правые и левые тандемные повторы

Рассмотрим произвольный тандемный повтор и посмотрим на его средний символ (точнее, на тот символ, с которого начинается вторая половинка тандема; т.е. если тандемный повтор — это подстрока $s[i \ldots j]$, то средним символом будет $(i + j + 1)/2$.

Тогда назовём тандемный повтор **левым или правым** в зависимости от того, где находится этот символ — в строке u или в строке v. (Можно сказать и так: тандемный повтор называется левым, если большая его часть лежит в левой половине строки s; иначе — тандемный повтор называется правым.)

Научимся искать **все левые тандемные повторы**; для правых всё будет аналогично.

Центральная позиция $cntr$ тандемного повтора

Обозначим длину искомого левого тандемного повтора через $2k$ (т.е. длина каждой половинки тандемного повтора — это k). Рассмотрим первый символ тандемного повтора, попадающий в строку v (он стоит в строке s в позиции $length(u)$). Он совпадает с символом, стоящим на k позиций раньше него; обозначим эту позицию через $cntr$.

Искать все тандемные повторы мы будем, **перебирая эту позицию** $cntr$: т.е. найдём сначала все тандемные повторы с одним значением $cntr$, затем с другим значением, и т.д. — перебирая все возможные значения $cntr$ от 0 до $length(u) - 1$.

Например, рассмотрим такую строку:

$$s = "cac|ada"$$

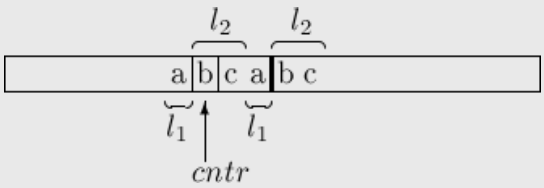
(символ вертикальной черты разделяет две половинки u и v)

Тандемный повтор $"caca"$, содержащийся в этой строке, будет обнаружен, когда мы будем просматривать значение $cntr = 1$ — потому что именно в позиции 1 стоит символ 'a', совпадающий с первым символом тандемного повтора, попавшим в половинку v.

Критерий наличия тандемного повтора с заданным центром $cntr$

Итак, мы должны научиться для зафиксированного значения $cntr$ быстро искать все тандемные повторы, соответствующие ему.

Получаем такую схему (для абстрактной строки, в которой содержится тандемный повтор "$abcabc$"):



Здесь через l_1 и l_2 мы обозначили длины двух кусочков тандемного повтора: l_1 — это длина части тандемного повтора до позиции $cntr - 1$, а l_2 — это длина части тандемного повтора от $cntr$ до конца половинки тандемного повтора. Таким образом, $l_1 + l_2 + l_1 + l_2$ — это длина тандемного повтора.

Взглянув на эту картинку, можно понять, что **необходимое и достаточное** условие того, что с центром в позиции $cntr$ находится тандемный повтор длины $2l = 2(l_1 + l_2) = 2(length(u) - cntr)$, является следующее условие:

- Пусть k_1 — это наибольшее число такое, что k_1 символов перед позицией $cntr$ совпадают с последними k_1 символами строки u:

$$u[cntr - k_1 \dots cntr - 1] == u[length(u) - k_1 \dots length(u) - 1].$$

- Пусть k_2 — это наибольшее число такое, что k_2 символов, начиная с позиции $cntr$, совпадают с первыми k_2 символами строки v:

$$u[cntr \dots cntr + k_2 - 1] == v[0 \dots k_2 - 1].$$

- Тогда должно выполняться:

$$\begin{cases} l_1 \le k_1, \\ l_2 \le k_2. \end{cases}$$

Этот критерий можно **переформулировать** таким образом. Зафиксируем конкретное значение $cntr$, тогда:

- Все тандемные повторы, которые мы будем сейчас обнаруживать, будут иметь длину $2l = 2(length(u) - cntr)$.

  Однако таких тандемных повторов может быть **несколько**: всё зависит от выбора длин кусочков l_1 и $l_2 = l - l_1$.

- Найдём k_1 и k_2, как было описано выше.
- Тогда подходящими будут являться тандемные повторы, для которых длины кусочков l_1 и l_2 удовлетворяют условиям:

$$\begin{cases} l_1 + l_2 = l = length(u) - cntr, \\ l_1 \le k_1, \\ l_2 \le k_2. \end{cases}$$

Алгоритм нахождения длин k_1 и k_2

Итак, вся задача сводится к быстрому вычислению длин k_1 и k_2 для каждого значения $cntr$.

Напомним их определения:

- k_1 — максимальное неотрицательное число, для которого выполнено:

$$u[cntr - k_1 \dots cntr - 1] == u[length(u) - k_1 \dots length(u) - 1].$$

- k_2 — максимальное неотрицательное число, для которого выполнено:

$$u[cntr \dots cntr + k_2 - 1] == v[0 \dots k_2 - 1].$$

На оба этих запроса можно отвечать за $O(1)$, используя **алгоритм нахождения Z-функции**:

- Для быстрого нахождения значений k_1 заранее посчитаем Z-функцию для строки $\overline{u}$ (т.е. строки u, выписанной в обратном порядке).

  Тогда значение k_1 для конкретного $cntr$ будет просто равно соответствующему значению массива Z-функции.

- Для быстрого нахождения значений k_2 заранее посчитаем Z-функцию для строки $v + \# + u$ (т.е. строки u, приписанной к строке v через символ-разделитель).

  Опять же, значение k_2 для конкретного $cntr$ надо будет просто взять из соответствующего элемента Z-функции.

Поиск правых тандемных повторов

До этого момента мы работали только с левыми тандемными повторами.

Чтобы искать правые тандемные повторы, надо действовать аналогично: мы определяем центр $cntr$ как символ, соответствующий последнему символу тандемного повтора, попавшему в первую строку.

Тогда длина k_1 будет определяться как наибольшее число символов до позиции $cntr$ включительно, совпадающих с последними символами строки u. Длина k_2 будет определяться как максимальное число символов, начиная с $cntr + 1$, совпадающих с первыми символами строки v.

Таким образом, для быстрого нахождения k_1 и k_2 надо будет посчитать заранее Z-функцию для строк $\overline{u} + \# + \overline{v}$ и v соответственно. После этого, перебирая конкретное значение $cntr$, мы по тому же самому критерию будем находить все правые тандемные повторы.

Асимптотика

Асимптотика алгоритма Мейна-Лоренца составит, таким образом, $O(n \log n)$: поскольку этот алгоритм представляет собой алгоритм "разделяй-и-властвуй", каждый рекурсивный запуск которого работает за время, линейное относительно длины строки: для четырёх строк за линейное время ищется их Z-функция, а затем перебирается значение $cntr$ и выводятся все группы обнаруженных тандемных повторов.

Тандемные повторы обнаруживаются алгоритмом Мейна-Лоренца в виде своеобразных **групп**: таких четвёрок $(cntr, l, k_1, k_2)$, каждая из которых обозначает группу тандемных повторов с длиной l, центром $cntr$ и с всевозможными длинами кусочков l_1 и l_2, удовлетворяющими условиям:

$$\begin{cases} l_1 + l_2 = l, \\ l_1 \le k_1, \\ l_2 \le k_2. \end{cases}$$

Реализация

Приведём реализацию алгоритма Мейна-Лоренца, которая за время $O(n \log n)$ находит все тандемные повторы данной строки в сжатом виде (в виде групп, описываемых четвёрками чисел).

В целях демонстрации обнаруженные тандемные повторы за время $O(n^2)$ "разжимаются" и выводятся по отдельности. Этот вывод при решении реальных задач легко будет заменить на какие-то другие, более эффективные, действия, например, на поиск наидлиннейшего тандемного повтора или подсчёт количества тандемных повторов.

```
vector<int> z_function (const string & s) {
        int n = (int) s.length();
        vector<int> z (n);
        for (int i=1, l=0, r=0; i<n; ++i) {
                if (i <= r)
                        z[i] = min (r-i+1, z[i-l]);
                while (i+z[i] < n && s[z[i]] == s[i+z[i]])
                        ++z[i];
                if (i+z[i]-1 > r)
                        l = i,  r = i+z[i]-1;
        }
        return z;
}

void output_tandem (const string & s, int shift, bool left, int cntr, int
l, int l1, int l2) {
        int pos;
        if (left)
                pos = cntr-l1;
        else
                pos = cntr-l1-l2-l1+1;
        cout << "[" << shift + pos << ".." << shift + pos+2*l-1 << "] = " <<
s.substr (pos, 2*l) << endl;
}

void output_tandems (const string & s, int shift, bool left, int cntr, int
l, int k1, int k2) {
        for (int l1=1; l1<=l; ++l1) {
                if (left && l1 == l)  break;
                if (l1 <= k1 && l-l1 <= k2)
                        output_tandem (s, shift, left, cntr, l, l1, l-l1);
        }
}
```

```
inline int get_z (const vector<int> & z, int i) {
        return 0<=i && i<(int)z.size() ? z[i] : 0;
}

void find_tandems (string s, int shift = 0) {
        int n = (int) s.length();
        if (n == 1)  return;

        int nu = n/2,  nv = n-nu;
        string u = s.substr (0, nu),
                v = s.substr (nu);
        string ru = string (u.rbegin(), u.rend()),
                rv = string (v.rbegin(), v.rend());

        find_tandems (u, shift);
        find_tandems (v, shift + nu);

        vector<int> z1 = z_function (ru),
                z2 = z_function (v + '#' + u),
                z3 = z_function (ru + '#' + rv),
                z4 = z_function (v);
        for (int cntr=0; cntr<n; ++cntr) {
                int l, k1, k2;
                if (cntr < nu) {
                        l = nu - cntr;
                        k1 = get_z (z1, nu-cntr);
                        k2 = get_z (z2, nv+1+cntr);
                }
                else {
                        l = cntr - nu + 1;
                        k1 = get_z (z3, nu+1 + nv-1-(cntr-nu));
                        k2 = get_z (z4, (cntr-nu)+1);
                }
                if (k1 + k2 >= l)
                        output_tandems (s, shift, cntr<nu, cntr, l, k1, k2);
        }
}
```

Литература

- Michael Main, Richard J. Lorentz. An O (n log n) Algorithm for Finding All Repetitions in a String [1982]
- Bill Smyth. Computing Patterns in Strings [2003]
- Билл Смит. **Методы и алгоритмы вычислений на строках** [2006]

Поиск подстроки в строке с помощью Z- или Префикс-функции

Даны строки S и T. Требуется найти все вхождения строки S в текст T за O (N), где N - суммарная длина строк S и T.

Алгоритм

Образуем строку S$T, где $ - некий разделитель, который не встречается ни в S, ни в T.

Вычислим для полученной строки **префикс-функцию** P за O (N). Пройдёмся по массиву P, и рассмотрим все его элементы, которые равны |S| (длине S). По определению префикс-фунции, это означает, что в это месте оканчивается подстрока, совпадающая с |S|, т.е. искомое вхождение. Таким образом, мы нашли все вхождения. Этот алгоритм называется алгоритмом **КМП** (**Кнута-Морриса-Пратта**).

Теперь решим эту же задачу с помощью Z-**функции**. Построим за O (N) массив Z - Z-функцию строки S$T. Пройдёмся по всем его элементам, и рассмотрим те из них, которые равны |S|. По определению, в этом месте начинается подстрока, совпадающая с S. Таким образом, мы нашли все вхождения.

Решение задачи "сжатие строки" за O (N)

Дана строка S. Требуется найти такую строку T, что строка S получается многократным повторением T. Из всех возможных T нужно выбрать наименьшую по длине.

Эту задачу очень просто решить за O (N) с помощью префикс-функции.

Итак, пусть массив P - префикс-функция строки S, которую можно вычислить за O (N).

Теперь рассмотрим значение последнего элемента P: P[N-1]. Если N делится на (N - P[N-1]), то ответ существует, и это N - P[N-1] первых букв строки S. Если же не делится, то ответа не существует.

Корректность этого метода легко понять. P[N-1] равно длине наидлиннейшего собственного суффикса строки S, совпадающего с префиксом S. Если ответ существует, то, очевидно, начальный кусок строки S длиной (N - P[N-1]) и будет ответом, и, следовательно, N будет делиться на (N - P[N-1]). Если же ответа не существует, то (N - P[N-1]) будет равно какому-то непонятному значению, на которое N делиться не будет (иначе бы ответ существовал).

Реализация

```
int n = (int) s.length();
vector<int> p (n);
// ... здесь вычисление префикс-функции ...
int l = n - p[n-1];
if (n % l == 0)
        cout << s.substr (l);
else
        cout << "No Solution";
```

Sqrt-декомпозиция

Sqrt-декомпозиция — это метод, или структура данных, которая позволяет выполнять некоторые типичные операции (суммирование элементов подмассива, нахождение минимума/максимума и т.д.) за $O(\sqrt{n})$, что значительно быстрее, чем $O(n)$ для тривиального алгоритма.

Сначала мы опишем структуру данных для одного из простейших применений этой идеи, затем покажем, как обобщать её для решения некоторых других задач, и, наконец, рассмотрим несколько иное применение этой идеи: разбиение входных запросов на sqrt-блоки.

Структура данных на основе sqrt-декомпозиции

Поставим задачу. Дан массив $a[0 \dots n-1]$. Требуется реализовать такую структуру данных, которая сможет находить сумму элементов $a[l \dots r]$ для произвольных l и r за $O(\sqrt{n})$ операций.

Описание

Основная идея sqrt-декомпозиции заключается в том, что сделаем следующий **предпосчёт**: разделим массив a на блоки длины примерно $\sqrt{n}$, и в каждом блоке i заранее предпосчитаем сумму $b[i]$ элементов в нём.

Можно считать, что длина одного блока и количество блоков равны одному и тому же числу — корню из n, округлённому вверх:

$$s = \lceil \sqrt{n} \rceil ,$$

тогда массив $a[]$ разбивается на блоки примерно таким образом:

$$\underbrace{a[0]\ a[1]\ \dots\ a[s-1]}_{b[0]}\quad \underbrace{a[s]\ a[s+1]\ \dots\ a[2\cdot s-1]}_{b[1]}\quad \dots\quad \underbrace{a[(s-1)\cdot s]\ \dots\ a[n]}_{b[s-1]}.$$

Хотя последний блок может содержать меньше, чем s, элементов (если n не делится на s), — это не принципиально.

Таким образом, для каждого блока k мы знаем сумму на нём $b[k]$:

$$b[k] = \sum_{i=k\cdot s}^{\min(n-1,(k+1)\cdot s-1)} a[i].$$

Итак, пусть эти значения b_k предварительно подсчитаны (для этого надо, очевидно, $O(n)$ операций). Что они могут дать при вычислении ответа на очередной запрос (l, r)? Заметим, что если отрезок $[l; r]$ длинный, то в нём будут содержаться несколько блоков целиком, и на такие блоки мы можем узнать сумму на них за одну операцию. В итоге от всего отрезка $[l; r]$ останется лишь два блока, попадающие в него лишь частично, и на этих кусках нам придётся произвести суммирование тривиальным алгоритмом.

Иллюстрация (здесь через k обозначен номер блока, в котором лежит l, а через p — номер блока, в котором лежит r):

$$\dots\ \overbrace{a[l]\ \dots\ a[(k+1)\cdot s-1]\ \underbrace{a[(k+1)\cdot s]\ \dots\ a[(k+2)\cdot s-1]}_{b[k+1]}\ \dots\ \underbrace{a[(p-1)\cdot s]\ \dots\ a[p\cdot s-1]}_{b[p]}\ a[p\cdot s]\ \dots a_r}^{sum=?}\ \dots$$

На этом рисунке видно, что для того чтобы посчитать сумму в отрезке $[l \dots r]$, надо просуммировать элементы только в двух "хвостах": $[l \dots (k+1)\cdot s-1]$ и $[p\cdot s \dots r]$, и просуммировать значения $b[i]$ во всех блоках, начиная с $k+1$ и заканчивая $p-1$:

$$\sum_{i=l}^{r} a[i] = \sum_{i=l}^{(k+1)\cdot s-1} a[i] + \sum_{i=k+1}^{p-1} b[i] + \sum_{i=p\cdot s}^{r} a[i]$$

(примечание: эта формула неверна, когда $k = p$: в таком случае некоторые элементы будут просуммированы дважды; в этом случае надо просто просуммировать элементы с l по r)

Тем самым мы экономим значительное количество операций. Действительно, размер каждого из "хвостов", очевидно, не превосходит длины блока s, и количество блоков также не превосходит s. Поскольку s мы выбирали $\approx \sqrt{n}$, то

всего для вычисления суммы на отрезке $[l \ldots r]$ нам понадобится лишь $O(\sqrt{n})$ операций.

Реализация

Приведём сначала простейшую реализацию:

```
// входные данные
int n;
vector<int> a (n);

// предпосчёт
int len = (int) sqrt (n + .0) + 1; // и размер блока, и количество блоков
vector<int> b (len);
for (int i=0; i<n; ++i)
        b[i / len] += a[i];

// ответ на запросы
for (;;) {
        int l, r; // считываем входные данные - очередной запрос
        int sum = 0;
        for (int i=l; i<=r; )
                if (i % len == 0 && i + len - 1 <= r) {
                        // если i указывает на начало блока, целиком лежащего
в [l;r]
                        sum += b[i / len];
                        i += len;
                }
                else {
                        sum += a[i];
                        ++i;
                }
}
```

Недостатком этой реализации является то, что в ней неоправданно много операций деления (которые, как известно, выполняются значительно медленнее других операций). Вместо этого можно посчитать номера блоков c_l и c_r, в которых лежат границы l и r соответственно, и затем сделать цикл по блокам с $c_l + 1$ по $c_r - 1$, отдельно обработав "хвосты" в блоках c_l и c_r. Кроме того, при такой реализации случай $c_l = c_r$ становится особым и требует отдельной обработки:

```
int sum = 0;
int c_l = l / len,   c_r = r / len;
if (c_l == c_r)
        for (int i=l; i<=r; ++i)
                sum += a[i];
else {
        for (int i=l, end=(c_l+1)*len-1; i<=end; ++i)
                sum += a[i];
        for (int i=c_l+1; i<=c_r-1; ++i)
                sum += b[i];
        for (int i=c_r*len; i<=r; ++i)
                sum += a[i];
}
```

Другие задачи

Мы рассматривали задачу нахождения суммы элементов массива в каком-то его подотрезке. Эту задачу можно немного расширить: разрешим также **меняться** отдельным элементам массива A. Действительно, если меняется какой-то элемент a_i, то достаточно обновить значение $b[k]$ в том блоке, в котором этот элемент находится ($k = i/len$):

$$b[k] += a[i] - old_a[i].$$

С другой стороны, вместо задачи о сумме аналогично можно решать задачи о **минимальном**, **максимальном** элементах в отрезке. Если в этих задачах допускать изменения отдельных элементов, то тоже надо будет пересчитывать значение b_k того блока, которому принадлежит изменяемый элемент, но пересчитывать уже полностью, проходом по всем элементам блока за $O(len) = O(\sqrt{n})$ операций.

Аналогичным образом sqrt-декомпозицию можно применять и для множества **других** подобных задач:

нахождение количества нулевых элементов, первого ненулевого элемента, подсчёта количества определённых элементов, и т.д.

Есть и целый класс задач, когда происходят **изменения элементов на целом подотрезке**: прибавление или присвоение элементов на каком-то подотрезке массива A.

Например, нужно выполнять следующие два вида запросов: прибавить ко всем элементам некоторого отрезка $[l; r]$ величину δ, и узнавать значение отдельного элемента a_i. Тогда в качестве b_k положим ту величину, которая должна быть прибавлена ко всем элементам k-го блока (например, изначально все $b_k = 0$); тогда при выполнении запроса "прибавление" нужно будет выполнить прибавление ко всем элементам a_i "хвостов", а затем выполнить прибавление ко всем элементам b_i для блоков, целиком лежащих в отрезке $[l \ldots r]$. А ответом на второй запрос, очевидно, будет просто $a_i + b_k$, где $k = i/len$. Таким образом, прибавление на отрезке будет выполняться за $O(\sqrt{n})$, а запрос отдельного элемента — за $O(1)$.

Наконец, можно комбинировать оба вида задач: изменение элементов на отрезке и ответ на запросы тоже на отрезке. Оба вида операций будут выполняться за $O(\sqrt{n})$. Для этого уже надо будет делать два "блоковых" массива b и c: один — для обеспечения изменений на отрезке, другой — для ответа на запросы.

Можно привести пример и других задач, к которым можно применить sqrt-декомпозицию. Например, можно решать задачу о **поддержании множества чисел** с возможностью добавления/удаления чисел, проверки числа на принадлежность множеству, поиск k-го по порядку числа. Для решения этой задачи надо хранить числа в отсортированном порядке, разделёнными на несколько блоков по $\sqrt{n}$ чисел в каждом. При добавлении или удалении числа надо будет производить "перебалансировку" блоков, перебрасывая числа из начал/концов одних блоков в начала/концы соседних блоков.

Sqrt-декомпозиция входных запросов

Рассмотрим теперь совершенно иное применение идеи об sqrt-декомпозиции.

Предположим, что у нас есть некоторая задача, в которой нам даются некоторые входные данные, а затем поступают k команд/запросов, каждую из которых мы должны дать обработать и выдать ответ. Мы рассматриваем случай, когда запросы бывают как запрашивающие (не меняющие состояния системы, а только запрашивающие некоторую информацию), так и модифицирующие (т.е. влияющие на состояние системы, изначально заданное входными данными).

Конкретная задача может быть весьма сложной, и "честное" её решение (которое считывает один запрос, обрабатывает его, изменяя состояние системы, и возвращает ответ) может быть технически сложным или вовсе быть не по силам для решающего. С другой стороны, решение "оффлайнового" варианта задачи, т.е. когда отсутствуют модифицирующие операции, а имеются только лишь запрашивающие запросы — часто оказывается гораздо проще. Предположим, что мы **умеем решать "оффлайновый" вариант** задачи, т.е. строить за некоторое время $B(n)$ некую структуру данных, которая может отвечать на запросы, но не умеет обрабатывать модифицирующие запросы.

Тогда **разобьём входные запросы на блоки** (какой длины — пока не уточняем; обозначим эту длину через s). В начале обработки каждого блока будем за $B(n)$ строить структуру данных для "оффлайнового" варианта задачи по состоянию данных на момент начала этого блока.

Теперь будем по очереди брать запросы из текущего блока и обрабатывать каждый из них. Если текущий запрос — модифицирующий, то пропустим его. Если же текущий запрос — запрашивающий, то обратимся к структуре данных для оффлайнового варианта задачи, но предварительно **учтя все модифицирующие запросы в текущем блоке**. Такое учитывание модифицирующих запросов бывает возможным далеко не всегда, и оно должно происходить достаточно быстро — за время $O(s)$ или немного хуже; обозначим это время через $Q(s)$.

Таким образом, если всего у нас m запросов, то на их обработку потребуется $B(m)\frac{m}{s} + mQ(s)$ времени. Величину s следует выбирать, исходя из конкретного вида функций $B()$ и $Q()$. Например, если $B(m) = O(m)$ и $Q(s) = O(s)$, то оптимальным выбором будет $s \approx \sqrt{m}$, и итоговая асимптотика получится $O(m\sqrt{m})$.

Поскольку приведённые выше рассуждения слишком абстрактны, приведём несколько примеров задач, к которым применима такая sqrt-декомпозиция.

Пример задачи: прибавление на отрезке

Условие задачи: дан массив чисел $a[1 \ldots n]$, и поступают запросы двух видов: узнать значение в i-ом элементе массива, и прибавить некоторое число x ко всем элементам массива в некотором отрезке $a[l \ldots r]$.

Хотя эту задачу можно решать и без этого приёма с разбиением запросов на блоки, мы приведём её здесь — как простейшее и наглядное применение этого метода.

Итак, разобьём входные запросы на блоки по $\sqrt{m}$ (где m — число запросов). В начале первого блока запросов никаких структур строить не надо, просто храним массив $a[]$. Идём теперь по запросам первого блока. Если текущий запрос — запрос прибавления, то пока пропускаем его. Если же текущий запрос — запрос чтения значения

в некоторой позиции i, то вначале просто возьмём в качестве ответа значение $a[i]$. Затем пройдёмся по всем пропущенным в этом блоке запросам прибавления, и для тех из них, в которые попадает i, применим их увеличения к текущему ответу.

Таким образом, мы научились отвечать на запрашивающие запросы за время $O(\sqrt{m})$.

Осталось только заметить, что в конце каждого блока запросов мы должны применить все модифицирующие запросы этого блока к массиву $a[]$. Но это легко сделать за $O(n)$ — достаточно для каждого запроса прибавления (l, r, x) отметить в вспомогательном массиве в точке l число x, а в точке $r + 1$ — число $-x$, и затем пройтись по этому массиву, прибавляя текущую сумму к массиву $a[]$.

Таким образом, итоговая асимптотика решения составит $O(\sqrt{m}(n + m))$.

Пример задачи: disjoint-set-union с разделением

Есть неориентированный граф с n вершинами и m рёбрами. Поступают запросы трёх видов: добавить ребро (x_i, y_i), удалить ребро (x_i, y_i), и проверить, связаны или нет вершины x_i и y_i путём.

Если бы запросы удаления отсутствовали, то решением задачи была бы известная структура данных disjoint-set-union (система непересекающихся множеств). Однако при наличии удалений задача значительно усложняется.

Сделаем следующим образом. В начале каждого блока запросов посмотрим, какие рёбра в этом блоке будут удаляться, и сразу **удалим** их из графа. Теперь построим систему непересекающихся множеств (dsu) на полученном графе.

Как мы теперь должны отвечать на очередной запрос из текущего блока? Наша система непересекающихся множеств "знает" обо всех рёбрах, кроме тех, что добавляются/удаляются в текущем блоке. Однако удаления из dsu нам делать уже не надо — мы заранее удалили все такие рёбра из графа. Таким образом, всё, что может быть — это дополнительные, добавляющиеся рёбра, которых может быть максимум $\sqrt{m}$ штук.

Следовательно, при ответе на текущий запрашивающий запрос мы можем просто пустить обход в ширину по компонентам связности dsu, который отработает за $O(\sqrt{m})$, поскольку у нас в рассмотрении будут только $O(\sqrt{m})$ рёбер.

Оффлайновые задачи на запросы на подотрезках массива и универсальная sqrt-эвристика для них

Рассмотрим ещё одну интересную вариацию идеи sqrt-декомпозиции.

Пусть у нас есть некоторая задача, в которой есть массив чисел, и поступают запрашивающие запросы, имеющие вид (l, r) — узнать что-то о подотрезке $a[l \ldots r]$. Мы считаем, что запросы не модифицирующие, и известны нам заранее, т.е. задача — оффлайновая.

Наконец, введём последнее **ограничение**: мы считаем, что умеем быстро пересчитывать ответ на запрос при изменении левой или правой границы на единицу. Т.е. если мы знали ответ на запрос (l, r), то быстро сможем посчитать ответ на запрос $(l + 1, r)$ или $(l - 1, r)$ или $(l, r + 1)$ или $(l, r - 1)$.

Опишем теперь **универсальную эвристику** для всех таких задач. Отсортируем запросы по паре: $(l \text{ div } \sqrt{n}, r)$. Т.е. мы отсортировали запросы по номеру sqrt-блока, в котором лежит левый конец, а при равенстве — по правому концу.

Рассмотрим теперь группу запросов с одинаковым значением $l \text{ div } \sqrt{n}$ и посмотрим, как мы можем обрабатывать её. Все такие запросы у нас упорядочены по правой границе, а, значит, мы можем просто стартовать с пустого отрезка $[l; l - 1]$, и раз за разом увеличивая на единицу правую границу — в итоге ответить на все такие запросы.

Хорошим **примером** на данную эвристику является такая задача: узнать количество различных чисел в отрезке массива $[l; r]$. Эта задача трудно поддаётся решению классическими методами.

Чуть более усложнённым вариантом этой задачи является задача с одного из раундов Codeforces.

Дерево Фенвика

Дерево Фенвика - это структура данных, дерево на массиве, обладающее следующими свойствами:

1) позволяет вычислять значение некоторой обратимой операции G на любом отрезке [L; R] за время O (log N);

2) позволяет изменять значение любого элемента за O (log N);

3) требует O (N) **памяти**, а точнее, ровно столько же, сколько и массив из N элементов;

4) легко обобщается на случай многомерных массивов.

Наиболее распространённое применение дерева Фенвика - для вычисления суммы на отрезке, т.е. функция G (X1, ..., Xk) = X1 + ... + Xk.

Дерево Фенвика было впервые описано в статье "A new data structure for cumulative frequency tables" (Peter M. Fenwick, 1994).

Описание

Для простоты описания мы предполагаем, что операция G, по которой мы строим дерево, - это **сумма**.

Пусть дан массив A[0..N-1]. Дерево Фенвика - массив T[0..N-1], в каждом элементе которого хранится сумма некоторых элементов массива A:

```
Ti = сумма Aj для всех F(i) <= j <= i,
```

где F(i) - некоторая функция, которую мы определим несколько позже.

Теперь мы уже можем написать **псевдокод** для функции вычисления суммы на отрезке [0; R] и для функции изменения ячейки:

```
int sum (int r)
{
        int result = 0;
        while (r >= 0) {
                result += t[r];
                r = f(r) - 1;
        }
        return result;
}

void inc (int i, int delta)
{
        для всех j, для которых F(j) <= i <= j
        {
                t[j] += delta;
        }
}
```

Функция sum работает следующим образом. Вместо того чтобы идти по всем элементам массива A, она движется по массиву T, делая "прыжки" через отрезки там, где это возможно. Сначала она прибавляет к ответу значение суммы на отрезке [F(R); R], затем берёт сумму на отрезке [F(F(R)-1); F(R)-1], и так далее, пока не дойдёт до нуля.

Функция inc движется в обратную сторону - в сторону увеличения индексов, обновляя значения суммы T_j только для тех позиций, для которых это нужно, т.е. для всех j, для которых F(j) <= i <= j.

Очевидно, что от выбора функции F будет зависеть скорость выполнения обеих операций. Сейчас мы рассмотрим функцию, которая позволит достичь логарифмической производительности в обоих случаях.

Определим значение F(X) следующим образом. Рассмотрим двоичную запись этого числа и посмотрим на его младший бит. Если он равен нулю, то F(X) = X. Иначе двоичное представление числа X оканчивается на группу из одной или нескольких единиц. Заменим все единицы из этой группы на нули, и присвоим полученное число значению функции F(X).

Этому довольно сложному описанию соответствует очень простая формула:

```
F(X) = X & (X+1),
```

где & - это операция побитового логического "И".

Нетрудно убедиться, что эта формула соответствует словесному описанию функции, данному выше.

Нам осталось только научиться быстро находить такие числа j, для которых F(j) <= i <= j.

Однако нетрудно убедиться в том, что все такие числа j получаются из i последовательными заменами самого правого (самого младшего) нуля в двоичном представлении. Например, для i = 10 мы получим, что j = 11, 15, 31, 63 и т.д.

Как ни странно, такой операции (замена самого младшего нуля на единицу) также соответствует очень простая формула:

```
H(X) = X | (X+1),
```

где | - это операция побитового логического "ИЛИ".

Реализация дерева Фенвика для суммы для одномерного случая

```
vector<int> t;
int n;

void init (int nn)
{
        n = nn;
        t.assign (n, 0);
}

int sum (int r)
{
        int result = 0;
        for (; r >= 0; r = (r & (r+1)) - 1)
                result += t[r];
        return result;
}

void inc (int i, int delta)
{
        for (; i < n; i = (i | (i+1)))
                t[i] += delta;
}

int sum (int l, int r)
{
        return sum (r) - sum (l-1);
}

void init (vector<int> a)
{
        init ((int) a.size());
        for (unsigned i = 0; i < a.size(); i++)
                inc (i, a[i]);
}
```

Реализация дерева Фенвика для минимума для одномерного случая

Следует сразу заметить, что, поскольку дерево Фенвика позволяет найти значение функции в произвольном отрезке [0; R], то мы никак не сможем найти минимум на отрезке [L;R], где L > 0. Далее, все изменения значений должны происходить только в сторону уменьшения (опять же, поскольку никак не получится обратить функцию min).
Это значительные ограничения.

```
vector<int> t;
int n;

const int INF = 1000*1000*1000;

void init (int nn)
```

```
{
        n = nn;
        t.assign (n, INF);
}

int getmin (int r)
{
        int result = INF;
        for (; r >= 0; r = (r & (r+1)) - 1)
                result = min (result, t[r]);
        return result;
}

void update (int i, int new_val)
{
        for (; i < n; i = (i | (i+1)))
                t[i] = min (t[i], new_val);
}

void init (vector<int> a)
{
        init ((int) a.size());
        for (unsigned i = 0; i < a.size(); i++)
                update (i, a[i]);
}
```

Реализация дерева Фенвика для суммы для двумерного случая

Как уже отмечалось, дерево Фенвика легко обобщается на многомерный случай.

```
vector <vector <int> > t;
int n, m;

int sum (int x, int y)
{
        int result = 0;
        for (int i = x; i >= 0; i = (i & (i+1)) - 1)
                for (int j = y; j >= 0; j = (j & (j+1)) - 1)
                        result += t[i][j];
        return result;
}

void inc (int x, int y, int delta)
{
        for (int i = x; i < n; i = (i | (i+1)))
                for (int j = y; j < m; j = (j | (j+1)))
                        t[i][j] += delta;
}
```

Система непересекающихся множеств

В данной статье рассматривается структура данных "**система непересекающихся множеств**" (на английском "disjoint-set-union", или просто "DSU").

Эта структура данных предоставляет следующие возможности. Изначально имеется несколько элементов, каждый из которых находится в отдельном (своём собственном) множестве. За одну операцию можно **объединить два каких-либо множества**, а также можно **запросить**, **в каком множестве** сейчас находится указанный элемент. Также, в классическом варианте, вводится ещё одна операция — создание нового элемента, который помещается в отдельное множество.

Таким образом, базовый интерфейс данной структуры данных состоит всего из трёх операций:

- $\text{make_set}(x)$ — **добавляет** новый элемент x, помещая его в новое множество, состоящее из одного него.
- $\text{union_sets}(x, y)$ — **объединяет** два указанных множества (множество, в котором находится элемент x, и множество, в котором находится элемент y).
- $\text{find_set}(x)$ — **возвращает**, **в каком множестве** находится указанный элемент x. На самом деле при этом возвращается один из элементов множества (называемый **представителем** или **лидером** (в англоязычной литературе "leader")). Этот представитель выбирается в каждом множестве самой структурой данных (и может меняться с течением времени, а именно, после вызовов $\text{union_sets}()$).

  Например, если вызов $\text{find_set}()$ для каких-то двух элементов вернул одно и то же значение, то это означает, что эти элементы находятся в одном и том же множестве, а в противном случае — в разных множествах.

Описываемая ниже структура данных позволяет делать каждую из этих операций почти за $O(1)$ в среднем (более подробно об асимптотике см. ниже после описания алгоритма).

Также в одном из подразделов статьи описан альтернативный вариант реализации DSU, позволяющий добиться асимптотики $O(\log n)$ в среднем на один запрос при $m \geq n$; а при $m >> n$ (т.е. m значительно больше n) — и вовсе времени $O(1)$ в среднем на запрос (см. "Хранение DSU в виде явного списка множеств").

Построение эффективной структуры данных

Определимся сначала, в каком виде мы будем хранить всю информацию.

Множества элементов мы будем хранить в виде **деревьев**: одно дерево соответствует одному множеству. Корень дерева — это представитель (лидер) множества.

При реализации это означает, что мы заводим массив parent, в котором для каждого элемента мы храним ссылку на его предка в дерева. Для корней деревьев будем считать, что их предок — они сами (т.е. ссылка зацикливается в этом месте).

Наивная реализация

Мы уже можем написать первую реализацию системы непересекающихся множеств. Она будет довольно неэффективной, но затем мы улучшим её с помощью двух приёмов, получив в итоге почти константное время работы.

Итак, вся информация о множествах элементов хранится у нас с помощью массива parent.

Чтобы создать новый элемент (операция $\text{make_set}(v)$), мы просто создаём дерево с корнем в вершине v, отмечая, что её предок — это она сама.

Чтобы объединить два множества (операция $\text{union_sets}(a, b)$), мы сначала найдём лидеров множества, в котором находится a, и множества, в котором находится b. Если лидеры совпали, то ничего не делаем — это значит, что множества и так уже были объединены. В противном случае можно просто указать, что предок вершины b равен a (или наоборот) — тем самым присоединив одно дерево к другому.

Наконец, реализация операции поиска лидера ($\text{find_set}(v)$) проста: мы поднимаемся по предкам от вершины v, пока не дойдём до корня, т.е. пока ссылка на предка не ведёт в себя. Эту операцию удобнее реализовать рекурсивно (особенно это будет удобно позже, в связи с добавляемыми оптимизациями).

```
void make_set (int v) {
        parent[v] = v;
}

int find_set (int v) {
        if (v == parent[v])
```

```
                return v;
        return find_set (parent[v]);
}

void union_sets (int a, int b) {
        a = find_set (a);
        b = find_set (b);
        if (a != b)
                parent[b] = a;
}
```

Впрочем, такая реализация системы непересекающихся множеств весьма **неэффективна**. Легко построить пример, когда после нескольких объединений множеств получится ситуация, что множество — это дерево, выродившееся в длинную цепочку. В результате каждый вызов $\text{find_set}()$ будет работать на таком тесте за время порядка глубины дерева, т.е. за $O(n)$.

Это весьма далеко от той асимптотики, которую мы собирались получить (константное время работы). Поэтому рассмотрим две оптимизации, которые позволят (даже применённые по отдельности) значительно ускорить работу.

Эвристика сжатия пути

Эта эвристика предназначена для ускорения работы $\text{find_set}()$.

Она заключается в том, что когда после вызова $\text{find_set}(v)$ мы найдём искомого лидера p множества, то запомним, что у вершины v и всех пройденных по пути вершин — именно этот лидер p. Проще всего это сделать, перенаправив их $\text{parent}[]$ на эту вершину p.

Таким образом, у массива предков $\text{parent}[]$ смысл несколько меняется: теперь это **сжатый массив предков**, т.е. для каждой вершины там может храниться не непосредственный предок, а предок предка, предок предка предка, и т.д.

С другой стороны, понятно, что нельзя сделать, чтобы эти указатели parent всегда указывали на лидера: иначе при выполнении операции $\text{union_sets}()$ пришлось бы обновлять лидеров у $O(n)$ элементов.

Таким образом, к массиву $\text{parent}[]$ следует подходить именно как к массиву предков, возможно, частично сжатому.

Новая реализация операции $\text{find_set}()$ выглядит следующим образом:

```
int find_set (int v) {
        if (v == parent[v])
                return v;
        return parent[v] = find_set (parent[v]);
}
```

Такая простая реализация делает всё, что задумывалось: сначала путём рекурсивных вызовов находится лидера множества, а затем, в процессе раскрутки стека, этот лидер присваивается ссылкам parent для всех пройденных элементов.

Реализовать эту операцию можно и нерекурсивно, но тогда придётся осуществлять два прохода по дереву: первый найдёт искомого лидера, второй — проставит его всем вершинам пути. Впрочем, на практике нерекурсивная реализация не даёт существенного выигрыша.

Оценка асимптотики при применении эвристики сжатия пути

Покажем, что применение одной эвристики сжатия пути **позволяет достичь логарифмическую асимптотику**: $O(\log n)$ на один запрос в среднем.

Заметим, что, поскольку операция $\text{union_sets}()$ представляет из себя два вызова операции $\text{find_set}()$ и ещё $O(1)$ операций, то мы можем сосредоточиться в доказательстве только на оценку времени работы $O(m)$ операций $\text{find_set}()$.

Назовём **весом** $w[v]$ вершины v число потомков этой вершины (включая её саму). Веса вершин, очевидно, могут только увеличиваться в процессе работы алгоритма.

Назовём **размахом ребра** (a, b) разность весов концов этого ребра: $|w[a] - w[b]|$ (очевидно, у вершины-предка вес всегда больше, чем у вершины-потомка). Можно заметить, что размах какого-либо фиксированного ребра (a, b) может только увеличиваться в процессе работы алгоритма.

Кроме того, разобьём рёбра на **классы**: будем говорить, что ребро имеет класс k, если его размах принадлежит отрезку $[2^k; 2^{k+1} - 1]$. Таким образом, класс ребра — это число от 0 до $\lceil \log n \rceil$.

Зафиксируем теперь произвольную вершину x и будем следить, как меняется ребро в её предка: сначала оно

отсутствует (пока вершина x является лидером), затем проводится ребро из x в какую-то вершину (когда множество с вершиной x присоединяется к другому множеству), и затем может меняться при сжатии путей в процессе вызовов find_path. Понятно, что нас интересует асимптотика только последнего случая (при сжатии путей): все остальные случаи требуют $O(1)$ времени на один запрос.

Рассмотрим работу некоторого вызова операции find_set: он проходит в дереве вдоль некоторого **пути**, стирая все рёбра этого пути и перенаправляя их в лидера.

Рассмотрим этот путь и **исключим** из рассмотрения последнее ребро каждого класса (т.е. не более чем по одному ребру из класса $0, 1, \ldots \lceil \log n \rceil$). Тем самым мы исключили $O(\log n)$ рёбер из каждого запроса.

Рассмотрим теперь все **остальные** рёбра этого пути. Для каждого такого ребра, если оно имеет класс k, получается, что в этом пути есть ещё одно ребро класса k (иначе мы были бы обязаны исключить текущее ребро, как единственного представителя класса k). Таким образом, после сжатия пути это ребро заменится на ребро класса как минимум $k + 1$. Учитывая, что уменьшаться вес ребра не может, мы получаем, что для каждой вершины, затронутой запросом find_path, ребро в её предка либо было исключено, либо строго увеличило свой класс.

Отсюда мы окончательно получаем асимптотику работы m запросов: $O((n + m) \log n)$, что (при $m \geq n$) означает логарифмическое время работы на один запрос в среднем.

Эвристика объединения по рангу

Рассмотрим здесь другую эвристику, которая сама по себе способна ускорить время работы алгоритма, а в сочетании с эвристикой сжатия путей и вовсе способна достигнуть практически константного времени работы на один запрос в среднем.

Эта эвристика заключается в небольшом изменении работы union_sets: если в наивной реализации то, какое дерево будет присоединено к какому, определяется случайно, то теперь мы будем это делать на основе **рангов**.

Есть два варианта ранговой эвристики: в одном варианте рангом дерева называется **количество вершин** в нём, в другом — **глубина дерева** (точнее, верхняя граница на глубину дерева, поскольку при совместном применении эвристики сжатия путей реальная глубина дерева может уменьшаться).

В обоих вариантах суть эвристики одна и та же: при выполнении union_sets будем присоединять дерево с меньшим рангом к дереву с большим рангом.

Приведём реализацию **ранговой эвристики на основе размеров деревьев**:

```
void make_set (int v) {
        parent[v] = v;
        size[v] = 1;
}

void union_sets (int a, int b) {
        a = find_set (a);
        b = find_set (b);
        if (a != b) {
                if (size[a] < size[b])
                        swap (a, b);
                parent[b] = a;
                size[a] += size[b];
        }
}
```

Приведём реализацию **ранговой эвристики на основе глубины деревьев**:

```
void make_set (int v) {
        parent[v] = v;
        rank[v] = 0;
}

void union_sets (int a, int b) {
        a = find_set (a);
        b = find_set (b);
        if (a != b) {
                if (rank[a] < rank[b])
                        swap (a, b);
                parent[b] = a;
                if (rank[a] == rank[b])
                        ++rank[a];
        }
}
```

Оба варианта ранговой эвристики являются эквивалентными с точки зрения асимптотики, поэтому на практике можно применять любую из них.

Оценка асимптотики при применении ранговой эвристики

Покажем, что асимптотика работы системы непересекающихся множеств при использовании только ранговой эвристики, без эвристики сжатия путей, будет **логарифмической** на один запрос в среднем: $O(\log n)$.

Здесь мы покажем, что при любом из двух вариантов ранговой эвристики глубина каждого дерева будет величиной $O(\log n)$, что автоматически будет означать логарифмическую асимптотику для запроса find_set, и, следовательно, запроса union_sets.

Рассмотрим **ранговую эвристику по глубине дерева**. Покажем, что если ранг дерева равен k, то это дерево содержит как минимум 2^k вершин (отсюда будет автоматически следовать, что ранг, а, значит, и глубина дерева, есть величина $O(\log n)$). Доказывать будем по индукции: для $k = 0$ это очевидно. При сжатии путей глубина может только уменьшиться. Ранг дерева увеличивается с $k - 1$ до k, когда к нему присоединяется дерево ранга $k - 1$; применяя к этим двум деревьям размера $k - 1$ предположение индукции, получаем, что новое дерево ранга k действительно будет иметь как минимум 2^k вершин, что и требовалось доказать.

Рассмотрим теперь **ранговую эвристику по размерам деревьев**. Покажем, что если размер дерева равен k, то его высота не более $\lfloor \log k \rfloor$. Доказывать будем по индукции: для $k = 1$ утверждение верно. При сжатии путей глубина может только уменьшиться, поэтому сжатие путей ничего не нарушает. Пусть теперь объединяются два дерева размеров k_1 и k_2; тогда по предположению индукции их высоты меньше либо равны, соответственно, $\lfloor \log k_1 \rfloor$ и $\lfloor \log k_2 \rfloor$. Не теряя общности, считаем, что первое дерево — большее ($k_1 \geq k_2$), поэтому после объединения глубина получившегося дерева из $k_1 + k_2$ вершин станет равна:

$$h = \max(\lfloor \log k_1 \rfloor, 1 + \lfloor \log k_2 \rfloor).$$

Чтобы завершить доказательство, надо показать, что:

$$h \overset{?}{\leq} \lfloor \log(k_1 + k_2) \rfloor,$$
$$2^h = \max(2^{\lfloor \log k_1 \rfloor}, 2^{\lfloor \log 2k_2 \rfloor}) \overset{?}{\leq} 2^{\lfloor \log(k_1 + k_2) \rfloor},$$

что есть почти очевидное неравенство, поскольку $k_1 \leq k_1 + k_2$ и $2k_2 \leq k_1 + k_2$.

Объединение эвристик: сжатие пути плюс ранговая эвристика

Как уже упоминалось выше, совместное применение этих эвристик даёт особенно наилучший результат, в итоге достигая практически константного времени работы.

Мы не будем приводить здесь доказательства асимптотики, поскольку оно весьма объёмно (см., например, Кормен, Лейзерсон, Ривест, Штайн "Алгоритмы. Построение и анализ"). Впервые это доказательство было проведено Тарьяном (1975 г.).

Окончательный результат таков: при совместном применении эвристик сжатия пути и объединения по рангу время работы на один запрос получается $O(\alpha(n))$ в среднем, где $\alpha(n)$ — **обратная функция Аккермана**, которая растёт очень медленно, настолько медленно, что для всех разумных ограничений n она **не превосходит** 4 (примерно для $n \leq 10^{600}$).

Именно поэтому про асимптотику работы системы непересекающихся множеств уместно говорить "почти константное время работы".

Приведём здесь **итоговую реализацию системы непересекающихся множеств**, реализующую обе указанные эвристики (используется ранговая эвристика относительно глубин деревьев):

```
void make_set (int v) {
        parent[v] = v;
        rank[v] = 0;
}

int find_set (int v) {
        if (v == parent[v])
                return v;
        return parent[v] = find_set (parent[v]);
}

void union_sets (int a, int b) {
        a = find_set (a);
        b = find_set (b);
        if (a != b) {
                if (rank[a] < rank[b])
```

```
                        swap (a, b);
                parent[b] = a;
                if (rank[a] == rank[b])
                        ++rank[a];
        }
}
```

Применения в задачах и различных улучшения

В этом разделе мы рассмотрим несколько применений структуры данных "система непересекающихся множеств", как тривиальных, так и использующих некоторые улучшения структуры данных.

Поддержка компонент связности графа

Это одно из очевидных приложений структуры данных "система непересекающихся множеств", которое, по всей видимости, и стимулировало изучение этой структуры.

Формально задачу можно сформулировать таким образом: изначально дан пустой граф, постепенно в этот граф могут добавляться вершины и неориентированные рёбра, а также поступают запросы (a, b) — "в одинаковых ли компонентах связности лежат вершины a и b?".

Непосредственно применяя здесь описанную выше структуру данных, мы получаем решение, которое обрабатывает один запрос на добавление вершины/ребра или запрос на проверку двух вершин — за почти константное время в среднем.

Учитывая, что практически в точности такая же задача ставится при использовании **алгоритма Крускала нахождения минимального остовного дерева**, мы сразу же получаем улучшенную версию этого алгоритма, работающую практически за линейное время.

Иногда на практике встречается **инвертированная версия этой задачи**: изначально есть граф с какими-то вершинами и рёбрами, и поступают запросы на удаление рёбер. Если задача дана в оффлайн, т.е. мы заранее можем узнать все запросы, то решать эту задачу можно следующим образом: перевернём задачу задом наперёд: будем считать, что у нас есть пустой граф, в который могут добавляться рёбра (сначала добавим ребро последнего запроса, затем предпоследнего, и т.д.). Тем самым в результате инвертирования этой задачи мы пришли к обычной задаче, решение которой описывалось выше.

Поиск компонент связности на изображении

Одно из лежащих на поверхности применений DSU заключается в решении следующей задачи: имеется изображение $n \times m$ пикселей. Изначально всё изображение белое, но затем на нём рисуется несколько чёрных пикселей. Требуется определить размер каждой "белой" компоненты связности на итоговом изображении.

Для решения мы просто перебираем все белые клетки изображения, для каждой клетки перебираем её четырёх соседей, и если сосед тоже белый — то вызываем $\text{union_sets}()$ от этих двух вершин. Таким образом, у нас будет DSU с nm вершинами, соответствующими пикселям изображения. Получившиеся в итоге деревья DSU — и есть искомые компоненты связности.

Впрочем, такое решение с помощью системы непересекающихся множеств не имеет никаких преимуществ перед решением с помощью обхода в глубину.

Поддержка дополнительной информации для каждого множества

"Система непересекающихся множеств" позволяет легко хранить любую дополнительную информацию, относящуюся ко множествам.

Простой пример — это **размеры множеств**: как их хранить, было описано при описании ранговой эвристики (информация там записывалась для текущего лидера множества).

Таким образом, вместе с лидером каждого множества можно хранить любую дополнительную требуемую в конкретной задаче информацию.

Применение DSU для сжатия "прыжков" по отрезку. Задача о покраске подотрезков в оффлайне

Одно из распространённых применений DSU заключается в том, что если есть набор элементов, и из каждого элемента выходит по одному ребру, то мы можем быстро (за почти константное время) находить конечную точку, в которую мы попадём, если будем двигаться вдоль рёбер из заданной начальной точки.

Наглядным примером этого применения является **задача о покраске подотрезков**: есть отрезок длины L, каждая клетка которого (т.е. каждый кусочек длины 1) имеет нулевой цвет. Поступают запросы вида (l, r, c)

— перекрасить отрезок $[l; r]$ в цвет c. Требуется найти итоговый цвет каждой клетки. Запросы считаются известными заранее, т.е. задача — в оффлайне.

Для решения мы можем завести DSU-структуру, которая для каждой клетки будет хранить ссылку на ближайшую справа непокрашенную клетку. Таким образом, изначально каждая клетка указывает на саму себя, а после покраски первого подотрезка — клетка перед началом подотрезка будет указывать на клетку после конца подотрезка.

Теперь, чтобы решить задачу, мы рассматриваем запросы перекраски **в обратном порядке**: от последнего к первому. Для выполнения запроса мы просто каждый раз с помощью нашего DSU находим самую левую непокрашенную клетку внутри отрезка, перекрашиваем её, и перебрасываем указатель из неё на следующую справа пустую клетку.

Таким образом, мы здесь фактически используем DSU с эвристикой сжатия путей, но без ранговой эвристики (т.к. нам важно, кто станет лидером после объединения). Следовательно, итоговая асимптотика составит $O(\log n)$ на запрос (впрочем, с маленькой по сравнению с другими структурами данных константой).

Реализация:

```
void init() {
        for (int i=0; i<L; ++i)
                make_set (i);
}

void process_query (int l, int r, int c) {
        for (int v=l; ; ) {
                v = find_set (v);
                if (v >= r)  break;
                answer[v] = c;
                parent[v] = v+1;
        }
}
```

Впрочем, можно реализовать это решение **с ранговой эвристикой**: будем хранить для каждого множества в некотором массиве $\mathrm{end}[]$, где это множество заканчивается (т.е. самую правую точку). Тогда можно будет объединять два множества в одно по их ранговой эвристике, проставляя потом получившемуся множеству новую правую границу. Тем самым мы получим решение за $O(\alpha(n))$.

Поддержка расстояний до лидера

Иногда в конкретных приложениях системы непересекающихся множеств всплывает требование поддерживать расстояние до лидера (т.е. длину пути в рёбрах в дереве от текущей вершины до корня дерева).

Если бы не было эвристики сжатия путей, то никаких сложностей бы не возникало — расстояние до корня просто равнялось бы числу рекурсивных вызовов, которые сделала функция $\mathrm{find_set}$.

Однако в результате сжатия путей несколько рёбер пути могли сжаться в одно ребро. Таким образом, вместе с каждой вершиной придётся хранить дополнительную информацию: **длину текущего ребра из вершины в предка**.

При реализации удобно представлять, что массив $\mathrm{parent}[]$ и функция $\mathrm{find_set}$ теперь возвращают не одно число, а пару чисел: вершину-лидера и расстояние до неё:

```
void make_set (int v) {
        parent[v] = make_pair (v, 0);
        rank[v] = 0;
}

pair<int,int> find_set (int v) {
        if (v != parent[v].first) {
                int len = parent[v].second;
                parent[v] = find_set (parent[v].first);
                parent[v].second += len;
        }
        return parent[v];
}

void union_sets (int a, int b) {
        a = find_set (a) .first;
        b = find_set (b) .first;
        if (a != b) {
                if (rank[a] < rank[b])
                        swap (a, b);
```

```
                parent[b] = make_pair (a, 1);
                if (rank[a] == rank[b])
                        ++rank[a];
        }
}
```

Поддержка чётности длины пути и задача о проверке двудольности графа в онлайн

По аналогии с длиной пути до лидера, так же можно поддерживать чётность длины пути до него. Почему же это применение было выделено в отдельный пункт?

Дело в том, что обычно требование хранение чётности пути всплывает в связи со следующей **задачей**: изначально дан пустой граф, в него могут добавляться рёбра, а также поступать запросы вида "является ли компонента связности, содержащая данную вершину, **двудольной**?".

Для решения этой задачи мы можем завести систему непересекающихся множеств для хранения компонент связности, и хранить у каждой вершины чётность длины пути до её лидера. Тем самым, мы можем быстро проверять, приведёт ли добавление указанного ребра к нарушению двудольности графа или нет: а именно, если концы ребра лежат в одной и той же компоненте связности, и при этом имеют одинаковые чётности длины пути до лидера, то добавление этого ребра приведёт к образованию цикла нечётной длины и превращению текущей компоненты в недвудольную.

Главная **сложность**, с которой мы сталкиваемся при этом, — это то, что мы должны аккуратно, с учётом чётностей, производить объединение двух деревьев в функции union_sets.

Если мы добавляем ребро (a, b), связывающее две компоненты связности в одну, то при присоединении одного дерева к другому мы должны указать ему такую чётность, чтобы в результате у вершин a и b получались бы разные чётности длины пути.

Выведём **формулу**, по которой должна получаться эта чётность, выставляемая лидеру одного множества при присоединении его к лидеру другого множества. Обозначим через x чётность длины пути от вершины a до лидера её множества, через y — чётность длины пути от вершины b до лидера её множества, а через t — искомую чётность, которую мы должны поставить присоединяемому лидеру. Если множество с вершиной a присоединяется к множеству с вершиной b, становясь поддеревом, то после присоединения у вершины b её чётность не изменится и останется равной y, а у вершины a чётность станет равной $x \oplus t$ (символом $\oplus$ здесь обозначена операция XOR (симметрическая разность)). Нам требуется, чтобы эти две чётности различались, т.е. их XOR был равен единице. Т.е. получаем уравнение на t:

$$x \oplus t \oplus y = 1,$$

решая которое, находим:

$$t = x \oplus y \oplus 1.$$

Таким образом, независимо от того, какое множество присоединяется к какому, надо использовать указанную формулу для задания чётности ребра, проводимого из одного лидера к другому.

Приведём **реализацию** DSU с поддержкой чётностей. Как и в предыдущем пункте, в целях удобства мы используем пары для хранения предков и результата операции find_set. Кроме того, для каждого множества мы храним в массиве $\text{bipartite}[]$, является ли оно всё ещё двудольным или нет.

```
void make_set (int v) {
        parent[v] = make_pair (v, 0);
        rank[v] = 0;
        bipartite[v] = true;
}

pair<int,int> find_set (int v) {
        if (v != parent[v].first) {
                int parity = parent[v].second;
                parent[v] = find_set (parent[v].first);
                parent[v].second ^= parity;
        }
        return parent[v];
}

void add_edge (int a, int b) {
        pair<int,int> pa = find_set (a);
        a = pa.first;
        int x = pa.second;

        pair<int,int> pb = find_set (b);
```

```
        b = pb.first;
        int y = pb.second;

        if (a == b) {
                if (x == y)
                        bipartite[a] = false;
        }
        else {
                if (rank[a] < rank[b])
                        swap (a, b);
                parent[b] = make_pair (a, x ^ y ^ 1);
                if (rank[a] == rank[b])
                        ++rank[a];
        }
}

bool is_bipartite (int v) {
        return bipartite[ find_set(v) .first ];
}
```

Алгоритм нахождения RMQ (минимум на отрезке) за $O(\alpha(n))$ в среднем в оффлайне

Формально задача ставится следующим образом: нужно реализовать структуру данных, которая поддерживает два вида запросов: добавление указанного числа $\text{insert}(i) (i = 1 \dots n)$ и поиск и извлечение текущего минимального числа $\text{extract_min}()$. Будем считать, что каждое число добавляется ровно один раз.

Кроме того, считаем, что вся последовательность запросов известна нам заранее, т.е. задача — в оффлайне.

Идея решения следующая. Вместо того, чтобы по очереди отвечать на каждый запрос, переберём число $i = 1 \dots n$, и определим, ответом на какой запрос это число должно быть. Для этого нам надо найти первый неотвеченный запрос, идущий после добавления $\text{insert}(i)$ этого числа — легко понять, что это и есть тот запрос, ответом на который является число i.

Таким образом, здесь получается идея, похожая на **задачу о покраске отрезков**.

Можно получить решение за $O(\log n)$ в среднем на запрос, если мы откажемся от ранговой эвристики и будем просто хранить в каждом элементе ссылку на ближайший справа запрос $\text{extract_min}()$, и использовать сжатие пути для поддержания этих ссылок после объединений.

Также можно получить решение и за $O(\alpha(n))$, если мы будем использовать ранговую эвристику и будем хранить в каждом множестве номер позиции, где оно заканчивается (то, что в предыдущем варианте решения достигалось автоматически за счёт того, что ссылки всегда шли только вправо, — теперь надо будет хранить явно).

Алгоритм нахождения LCA (наименьшего общего предка в дереве) за $O(\alpha(n))$ в среднем в оффлайне

Алгоритм Тарьяна нахождения LCA за $O(1)$ в среднем в режиме онлайн описан в соответствующей статье. Этот алгоритм выгодно отличается от других алгоритмов поиска LCA своей простотой (особенно по сравнению с оптимальным алгоритмом Фарах-Колтона-Бендера).

Хранение DSU в виде явного списка множеств. Применение этой идеи при слиянии различных структур данных

Одним из альтернативных способов хранения DSU является сохранение каждого множества в виде **явно хранящегося списка его элементов**. При этом, у каждого элемента также сохраняется ссылка на представителя (лидера) его множества.

На первый взгляд кажется, что это неэффективная структура данных: при объединении двух множеств мы должны будем добавить один список в конец другого, а также обновить лидера у всех элементов одного из двух списков.

Однако, как оказывается, применение **весовой эвристики**, аналогичной описанной выше, позволяет существенно снизить асимптотику работы: до $O(m + n \log n)$ для выполнения m запросов над n элементами.

Под весовой эвристикой подразумевается, что мы всегда **будем добавлять меньшее из двух множеств в большее**. Добавление $\text{union_sets}()$ одного множества в другое легко реализовать за время порядка размера добавляемого множества, а поиск лидера $\text{find_set}()$ — за время $O(1)$ при таком способе хранения.

Докажем **асимптотику** $O(m + n \log n)$ для выполнения m запросов. Зафиксируем произвольный элемент x

и проследим, как на него воздействовали операции объединения union_sets. Когда на элемент x воздействовали первый раз, мы можем утверждать, что размер его нового множества будет как минимум 2. Когда на x воздействовали второй раз — можно утверждать, что он попадёт в множество размера не менее 4 (т.к. мы добавляем меньшее множество в большее). И так далее — получаем, что на элемент x могло воздействовать максимум $\lceil \log n \rceil$ операций объединения. Таким образом, в сумме по всем вершинам это составляет $O(n \log n)$, плюс по $O(1)$ на каждый запрос — что и требовалось доказать.

Приведём пример **реализации**:

```
vector<int> lst[MAXN];
int parent[MAXN];

void make_set (int v) {
        lst[v] = vector<int> (1, v);
        parent[v] = v;
}

int find_set (int v) {
        return parent[v];
}

void union_sets (int a, int b) {
        a = find_set (a);
        b = find_set (b);
        if (a != b) {
                if (lst[a].size() < lst[b].size())
                        swap (a, b);
                while (!lst[b].empty()) {
                        int v = lst[b].back();
                        lst[b].pop_back();
                        parent[v] = a;
                        lst[a].push_back (v);
                }
        }
}
```

Также эту идею добавления элементов меньшего множества в большее можно использовать и вне рамок DSU, при решении других задач.

Например, рассмотрим следующую **задачу**: дано дерево, каждому листу которого приписано какое-либо число (одно и то же число может встречаться несколько раз у разных листьев). Требуется для каждой вершины дерева узнать количество различных чисел в её поддереве.

Применив в этой задаче эту же идею, можно получить такое решение: пустим обход в глубину по дереву, который будет возвращать указатель на set чисел — список всех чисел в поддереве этой вершины. Тогда, чтобы получить ответ для текущей вершины (если, конечно, она не лист) — надо вызвать обход в глубину от всех детей этой вершины, и объединить все полученные set в один, размер которого и будет ответом для текущей вершины. Для эффективного объединения нескольких set в один как раз применим описанный выше приём: будем объединять два множества, просто добавляя по одному элементы меньшего множества в большее. В итоге мы получим решение за $O(n \log^2 n)$, поскольку добавление одного элемента в set производится за $O(\log n)$.

Хранение DSU с сохранением явной структуры деревьев. Переподвешивание. Алгоритм поиска мостов в графе за $O(\alpha(n))$ в среднем в онлайне

Одно из мощных применений структуры данных "системы непересекающихся множеств" заключается в том, что она позволяет хранить одновременно **как сжатую**, **так и несжатую структуру деревьев**. Сжатая структура может использоваться для быстрого объединения деревьев и проверки на принадлежность двух вершин одному дереву, а несжатая — например, для поиска пути между двумя заданными вершинами, или прочих обходов структуры дерева.

При реализации это означает, что помимо обычного для DSU массива сжатых предков $\text{parent}[]$ мы заведём массив обычных, несжатых, предков $\text{real_parent}[]$. Понятно, что поддержание такого массива никак не ухудшает асимптотику: изменения в нём происходят только при объединении двух деревьев, и лишь в одном элементе.

С другой стороны, при применении на практике нередко требуется научиться соединять два дерева указанным ребром, не обязательно выходящим из их корней. Это означает, что у нас нет другого выхода, кроме как **переподвесить** одно из деревьев за указанную вершину, чтобы затем мы смогли присоединить это дерево к другому, сделав корень этого дерева дочерней вершиной ко второму концу добавляемого ребра.

На первый взгляд кажется, что операция переподвешивания — очень затратна и сильно ухудшит

асимптотику. Действительно, для переподвешивания дерева за вершину v мы должны пройтись от этой вершины до корня дерева, обновляя везде указатели $\mathrm{parent}[]$ и $\mathrm{real_parent}[]$.

Однако на самом деле всё не так плохо: достаточно лишь переподвешивать то из двух деревьев, которое меньше, чтобы получить асимпотику одного объединения, равную $O(\log n)$ в среднем.

Более подробно (включая доказательства асимптотики) см. алгоритм поиска мостов в графе за $O(logn)$ в среднем в онлайне.

Историческая ретроспектива

Структура данных "система непересекающихся множеств" была известна сравнительно давно.

Способ хранения этой структуры в виде **леса деревьев** был, по всей видимости, впервые описан Галлером и Фишером в 1964 г. (Galler, Fisher "An Improved Equivalence Algorithm"), однако полный анализ асимптотики был проведён гораздо позже.

Эвристики сжатия путей и объединения по рангу, по-видимому, разработали МакИлрой (McIlroy) и Моррис (Morris), и, независимо от них, Триттер (Tritter).

Некоторое время была известная лишь оценка $O(\log^* n)$ на одну операцию в среднем, данная Хопкрофтом и Ульманом в 1973 г. (Hopcroft, Ullman "Set-merging algomthms") — здесь $\log^* n$ — **итерированный логарифм** (это медленно растущая функция, но всё же не настолько медленно, как обратная функция Аккермана).

Впервые оценку $O(\alpha(n))$, где $\alpha(n)$ — **обратная функция Аккермана** — получил Тарьян в своей статье 1975 г. (Tarjan "Efficiency of a Good But Not Linear Set Union Algorithm"). Позже в 1985 г. он вместе с Льювеном получил эту временную оценку для нескольких различных ранговых эвристик и способов сжатия пути (Tarjan, Leeuwen "Worst-Case Analysis of Set Union Algorithms").

Наконец, Фредман и Сакс в 1989 г. доказали, что в принятой модели вычислений **любой** алгоритм для системы непересекающихся множеств должен работать как минимум за $O(\alpha(n))$ в среднем (Fredman, Saks "The cell probe complexity of dynamic data structures").

Впрочем, следует также отметить, что есть несколько статей, **оспаривающих** эту временную оценку и утверждающих, что система непересекающихся множеств с эвристиками сжатия пути и объединения по рангу работает за $O(1)$ в среднем: Zhang "The Union-Find Problem Is Linear", Wu, Otoo "A Simpler Proof of the Average Case Complexity of Union-Find with Path Compression".

Задачи в online judges

Список задач, которые можно решить с помощью системы непересекающихся множеств:

- TIMUS #1671 "**Паутина Ананси**" [сложность: низкая]
- CODEFORCES 25D "**Дороги не только в Берляндии**" [сложность: средняя]
- TIMUS #1003 "**Чётность**" [сложность: средняя]
- SPOJ #1442 "Chain" [сложность: средняя]

Литература

- Томас Кормен, Чарльз Лейзерсон, Рональд Ривест, Клиффорд Штайн. **Алгоритмы**: **Построение и анализ** [2005]
- Kurt Mehlhorn, Peter Sanders. Algorithms and Data Structures: The Basic Toolbox [2008]
- Robert Endre Tarjan. Efficiency of a Good But Not Linear Set Union Algorithm [1975]
- Robert Endre Tarjan, Jan van Leeuwen. Worst-Case Analysis of Set Union Algorithms [1985]

Дерево отрезков

Дерево отрезков — это структура данных, которая позволяет эффективно (т.е. за асимптотику $O(\log n)$) реализовать операции следующего вида: нахождение суммы/минимума элементов массива в заданном отрезке ($a[l \dots r]$, где l и r поступают на вход алгоритма), при этом дополнительно возможно изменение элементов массива: как изменение значения одного элемента, так и изменение элементов на целом подотрезке массива (т.е. разрешается присвоить всем элементам $a[l \dots r]$ какое-либо значение, либо прибавить ко всем элементам массива какое-либо число).

Вообще, дерево отрезков — очень гибкая структура, и число задач, решаемых ей, теоретически неограниченно. Помимо приведённых выше видов операций с деревьями отрезков, также возможны и гораздо более сложные операции (см. раздел "Усложнённые версии дерева отрезков"). В частности, дерево отрезков легко обобщается на большие размерности: например, для решения задачи о поиске суммы/минимума в некотором подпрямоугольнике данной матрицы (правда, уже только за время $O(\log^2 n)$).

Важной особенностью деревьев отрезков является то, что они потребляют линейный объём памяти: стандартному дереву отрезков требуется порядка $4n$ элементов памяти для работы над массивом размера n.

Описание дерева отрезков в базовом варианте

Для начала рассмотрим простейший случай дерева отрезков — дерево отрезков для сумм. Если ставить задачу формально, то у нас есть массив $a[0..n-1]$, и наше дерево отрезков должно уметь находить сумму элементов с l-го по r-ый (это запрос суммы), а также обрабатывать изменение значения одного указанного элемента массива, т.е. фактически реагировать на присвоение $a[i] = x$ (это запрос модификации). Ещё раз повторимся, дерево отрезков должно обрабатывать оба этих запроса за время $O(\log n)$.

Структура дерева отрезков

Итак, что же представляет из себя дерево отрезков?

Подсчитаем и запомним где-нибудь сумму элементов всего массива, т.е. отрезка $a[0 \dots n-1]$. Также посчитаем сумму на двух половинках этого массива: $a[0 \dots n/2]$ и $a[n/2+1 \dots n-1]$. Каждую из этих двух половинок в свою очередь разобьём пополам, посчитаем и сохраним сумму на них, потом снова разобьём пополам, и так далее, пока текущий отрезок не достигнет длины 1. Иными словами, мы стартуем с отрезка $[0; n-1]$ и каждый раз делим текущий отрезок надвое (если он ещё не стал отрезком единичной длины), вызывая затем эту же процедуру от обеих половинок; для каждого такого отрезка мы храним сумму чисел на нём.

Можно говорить, что эти отрезки, на которых мы считали сумму, образуют дерево: корень этого дерева — отрезок $[0 \dots n-1]$, а каждая вершина имеет ровно двух сыновей (кроме вершин-листьев, у которых отрезок имеет длину 1). Отсюда и происходит название — "дерево отрезков" (хотя при реализации обычно никакого дерева явно не строится, но об этом ниже в разделе реализации).

Итак, мы описали структуру дерева отрезков. Сразу заметим, что оно имеет **линейный размер**, а именно, содержит менее $2n$ вершин. Понять это можно следующим образом: первый уровень дерева отрезков содержит одну вершину (отрезок $[0 \dots n-1]$), второй уровень — в худшем случае две вершины, на третьем уровне в худшем случае будет четыре вершины, и так далее, пока число вершин не достигнет n. Таким образом, число вершин в худшем случае оценивается суммой $n + n/2 + n/4 + n/8 + \dots + 1 < 2n$.

Стоит отметить, что при n, отличных от степеней двойки, не все уровни дерева отрезков будут полностью заполнены. Например, при $n = 3$ левый сын корня есть отрезок $[0 \dots 1]$, имеющий двух потомков, в то время как правый сын корня — отрезок $[2 \dots 2]$, являющийся листом. Никаких особых сложностей при реализации это не составляет, но тем не менее это надо иметь в виду.

Высота дерева отрезков есть величина $O(\log n)$ — например, потому что длина отрезка в корне дерева равна n, а при переходе на один уровень вниз длина отрезков уменьшается примерно вдвое.

Построение

Процесс построения дерева отрезков по заданному массиву a можно делать эффективно следующим образом, снизу вверх: сначала запишем значения элементов $a[i]$ в соответствующие листья дерева, затем на основе них посчитаем значения для вершин предыдущего уровня как сумму значений в двух листьях, затем аналогичным образом посчитаем значения для ещё одного уровня, и т.д. Удобно описывать эту операцию рекурсивно: мы запускаем процедуру построения от корня дерева отрезков, а сама процедура построения, если её вызвали не от листа, вызывает себя от каждого из двух сыновей и суммирует вычисленные значения, а если её вызвали от листа — то просто записывает в себя значение этого элемента массива.

Асимптотика построения дерева отрезков составит, таким образом, $O(n)$.

Запрос суммы

Рассмотрим теперь запрос суммы. На вход поступают два числа l и r, и мы должны за время $O(\log n)$ посчитать сумму чисел на отрезке $a[l \ldots r]$.

Для этого мы будем спускаться по построенному дереву отрезков, используя для подсчёта ответа посчитанные ранее суммы на каждой вершине дерева. Изначально мы встаём в корень дерева отрезков. Посмотрим, в какие из двух его сыновей попадает отрезок запроса $[l \ldots r]$ (напомним, что сыновья корня дерева отрезков — это отрезки $[0 \ldots n/2]$ и $[n/2+1 \ldots n-1]$). Возможны два варианта: что отрезок $[l \ldots r]$ попадает только в одного сына корня, и что, наоборот, отрезок пересекается с обоими сыновьями.

Первый случай прост: просто перейдём в того сына, в котором лежит наш отрезок-запрос, и применим описываемый здесь алгоритм к текущей вершине.

Во втором же случае нам не остаётся других вариантов, кроме как перейти сначала в левого сына и посчитать ответ на запрос в нём, а затем — перейти в правого сына, посчитать в нём ответ и прибавить к нашему ответу. Иными словами, если левый сын представлял отрезок $[l_1 \ldots r_1]$, а правый — отрезок $[l_2 \ldots r_2]$ (заметим, что $l_2 = r_1 + 1$), то мы перейдём в левого сына с запросом $[l \ldots r_1]$, а в правого — с запросом $[l_2 \ldots r]$.

Итак, обработка запроса суммы представляет собой **рекурсивную функцию**, которая всякий раз вызывает себя либо от левого сына, либо от правого (не изменяя границы запроса в обоих случаях), либо от обоих сразу (при этом деля наш запрос на два соответствующих подзапроса). Однако рекурсивные вызовы будем делать не всегда: если текущий запрос совпал с границами отрезка в текущей вершине дерева отрезков, то в качестве ответа будем возвращать предвычисленное значение суммы на этом отрезке, записанное в дереве отрезков.

Иными словами, вычисление запроса представляет собой спуск по дереву отрезков, который распространяется по всем нужным ветвям дерева, и для быстрой работы использующий уже посчитанные суммы по каждому отрезку в дереве отрезков.

Почему же **асимптотика** этого алгоритма будет $O(\log n)$? Для этого посмотрим на каждом уровне дерева отрезков, сколько максимум отрезков могла посетить наша рекурсивная функция при обработке какого-либо запроса. Утверждается, что таких отрезков не могло быть более четырёх; тогда, учитывая оценку $O(\log n)$ для высоты дерева, мы и получаем нужную асимптотику времени работы алгоритма.

Покажем, что эта оценка о четырёх отрезках верна. В самом деле, на нулевом уровне дерева запросом затрагивается единственная вершина — корень дерева. Дальше на первом уровне рекурсивный вызов в худшем случае разбивается на два рекурсивных вызова, но важно здесь то, что запросы в этих двух вызовах будут соседствовать, т.е. число l'' запроса во втором рекурсивном вызове будет на единицу больше числа r' запроса в первом рекурсивном вызове. Отсюда следует, что на следующем уровне каждый из этих двух вызовов мог породить ещё по два рекурсивных вызова, но в таком случае половина этих запросов отработает нерекурсивно, взяв нужное значение из вершины дерева отрезков. Таким образом, всякий раз у нас будет не более двух реально работающих ветвей рекурсии (можно сказать, что одна ветвь приближается к левой границе запроса, а вторая ветвь — к правой), а всего число затронутых отрезков не могло превысить высоты дерева отрезков, умноженной на четыре, т.е. оно есть число $O(\log n)$.

В завершение можно привести и такое понимание работы запроса суммы: входной отрезок $[l \ldots r]$ разбивается на несколько подотрезков, ответ на каждом из которых уже подсчитан и сохранён в дереве. Если делать это разбиение правильным образом, то благодаря структуре дерева отрезков число необходимых подотрезков всегда будет $O(\log n)$, что и даёт эффективность работы дерева отрезков.

Запрос обновления

Напомним, что запрос обновления получает на вход индекс i и значение x, и перестраивает дерево отрезков таким образом, чтобы оно соответствовало новому значению $a[i] = x$. Этот запрос должен также выполняться за время $O(\log n)$.

Это более простой запрос по сравнению с запросом подсчёта суммы. Дело в том, что элемент $a[i]$ участвует только в относительно небольшом числе вершин дерева отрезков: а именно, в $O(\log n)$ вершинах — по одной с каждого уровня.

Тогда понятно, что запрос обновления можно реализовать как рекурсивную функцию: ей передаётся текущая вершина дерева отрезков, и эта функция выполняет рекурсивный вызов от одного из двух своих сыновей (от того, который содержит позицию i в своём отрезке), а после этого — пересчитывает значение суммы в текущей вершине точно таким же образом, как мы это делали при построении дерева отрезков (т.е. как сумма значений по обоим сыновьям текущей вершины).

Реализация

Основной реализационный момент — это то, как **хранить** дерево отрезков в памяти. В целях простоты мы не будем хранить дерево в явном виде, а воспользуемся таким трюком: скажем, что корень дерева имеет номер 1,

его сыновья — номера 2 и 3, их сыновья — номера с 4 по 7, и так далее. Легко понять корректность следующей формулы: если вершина имеет номер i, то пусть её левый сын — это вершина с номером $2i$, а правый — с номером $2i+1$.

Такой приём значительно упрощает программирование дерева отрезков, — теперь нам не нужно хранить в памяти структуру дерева отрезков, а только лишь завести какой-либо массив для сумм на каждом отрезке дерева отрезков.

Стоит только отметить, что размер этого массива при такой нумерации надо ставить не $2n$, а $4n$. Дело в том, что такая нумерация не идеально работает в случае, когда n не является степенью двойки — тогда появляются пропущенные номера, которым не соответствуют никакие вершины дерева (фактически, нумерация ведёт себя подобно тому, как если бы n округлили бы вверх до ближайшей степени двойки). Это не создаёт никаких сложностей при реализации, однако приводит к тому, что размер массива надо увеличивать до $4n$.

Итак, дерево отрезков мы **храним** просто в виде массива $t[]$, размера вчетверо больше размера n входных данных:

```
int n, t[4*MAXN];
```

Процедура **построения дерева отрезков** по заданному массиву $a[]$ выглядит следующим образом: это рекурсивная функция, ей передаётся сам массив $a[]$, номер v текущей вершины дерева, и границы tl и tr отрезка, соответствующего текущей вершине дерева. Из основной программы вызывать эту функцию следует с параметрами $v=1, tl=0, tr=n-1$.

```
void build (int a[], int v, int tl, int tr) {
        if (tl == tr)
                t[v] = a[tl];
        else {
                int tm = (tl + tr) / 2;
                build (a, v*2, tl, tm);
                build (a, v*2+1, tm+1, tr);
                t[v] = t[v*2] + t[v*2+1];
        }
}
```

Далее, функция для **запроса суммы** представляет из себя также рекурсивную функцию, которой таким же образом передаётся информация о текущей вершине дерева (т.е. числа v, tl, tr, которым в основной программе следует передавать значения $1, 0, n-1$ соответственно), а помимо этого — также границы l и r текущего запроса. В целях упрощения кода эта фукнция всегда делает по два рекурсивных вызова, даже если на самом деле нужен один — просто лишнему рекурсивному вызову передастся запрос, у которого $l > r$, что легко отсекается дополнительной проверкой в самом начале функции.

```
int sum (int v, int tl, int tr, int l, int r) {
        if (l > r)
                return 0;
        if (l == tl && r == tr)
                return t[v];
        int tm = (tl + tr) / 2;
        return sum (v*2, tl, tm, l, min(r,tm))
                + sum (v*2+1, tm+1, tr, max(l,tm+1), r);
}
```

Наконец, **запрос модификации**. Ему точно так же передаётся информация о текущей вершине дерева отрезков, а дополнительно указывается индекс меняющегося элемента, а также его новое значение.

```
void update (int v, int tl, int tr, int pos, int new_val) {
        if (tl == tr)
                t[v] = new_val;
        else {
                int tm = (tl + tr) / 2;
                if (pos <= tm)
                        update (v*2, tl, tm, pos, new_val);
                else
                        update (v*2+1, tm+1, tr, pos, new_val);
                t[v] = t[v*2] + t[v*2+1];
        }
}
```

Стоит отметить, что функцию update легко сделать нерекурсивной, поскольку рекурсия в ней хвостовая, т. е. разветвлений никогда не происходит: один вызов может породить только один рекурсивный вызов. При нерекурсивной реализации скорость работы может вырасти в несколько раз.

Из других **оптимизаций** стоит упомянуть, что умножения и деления на два стоит заменить битовыми операциями — это также немного улучшает производительность дерева отрезков.

Усложнённые версии дерева отрезков

Дерево отрезков — очень гибкая структура, и позволяет делать обобщения во многих различных направлениях. Попытаемся ниже классифицировать их.

Более сложные функции и запросы

Улучшения дерева отрезков в этом направлении могут быть как довольно очевидными (как в случае минимума/максимума вместо суммы), так и весьма и весьма нетривиальными.

Поиск минимума/максимума

Немного изменим условие задачи, описанной выше: вместо запроса суммы будем производить теперь запрос минимума/максимума на отрезке.

Тогда дерево отрезков для такой задачи практически ничем не отличается от дерева отрезков, описанного выше. Просто надо изменить способ вычисления $t[v]$ в функциях build и update, а также вычисление возвращаемого ответа в функции sum (заменить суммирование на минимум/максимум).

Поиск минимума/максимума и количества раз, которое он встречается

Задача аналогична предыдущей, только теперь помимо максимума требуется также возвращать количество его вхождений. Эта задача встаёт естественным образом, например, при решении с помощью дерева отрезков такой задачи: найти количество наидлиннейших возрастающих подпоследовательностей в заданном массиве.

Для решения этой задачи в каждой вершине дерева отрезков будем хранить пару чисел: кроме максимума количество его вхождений на соответствующем отрезке. Тогда при построении дерева мы должны просто по двум таким парам, полученным от сыновей текущей вершины, получать пару для текущей вершины.

Объединение двух таких пар в одну стоит выделить в отдельную функцию, поскольку эту операцию надо будет производить и в запросе модификации, и в запросе поиска максимума.

```
pair<int,int> t[4*MAXN];

pair<int,int> combine (pair<int,int> a, pair<int,int> b) {
        if (a.first > b.first)
                return a;
        if (b.first > a.first)
                return b;
        return make_pair (a.first, a.second + b.second);
}

void build (int a[], int v, int tl, int tr) {
        if (tl == tr)
                t[v] = make_pair (a[tl], 1);
        else {
                int tm = (tl + tr) / 2;
                build (a, v*2, tl, tm);
                build (a, v*2+1, tm+1, tr);
                t[v] = combine (t[v*2], t[v*2+1]);
        }
}

pair<int,int> get_max (int v, int tl, int tr, int l, int r) {
        if (l > r)
                return make_pair (-INF, 0);
        if (l == tl && r == tr)
                return t[v];
        int tm = (tl + tr) / 2;
        return combine (
                get_max (v*2, tl, tm, l, min(r,tm)),
                get_max (v*2+1, tm+1, tr, max(l,tm+1), r)
        );
}

void update (int v, int tl, int tr, int pos, int new_val) {
```

```
        if (tl == tr)
                t[v] = make_pair (new_val, 1);
        else {
                int tm = (tl + tr) / 2;
                if (pos <= tm)
                        update (v*2, tl, tm, pos, new_val);
                else
                        update (v*2+1, tm+1, tr, pos, new_val);
                t[v] = combine (t[v*2], t[v*2+1]);
        }
}
```

Поиск наибольшего общего делителя / наименьшего общего кратного

Т.е. мы хотим научиться искать НОД/НОК всех чисел в заданном отрезке массива.

Это довольно интересное обобщение дерева отрезков получается абсолютно таким же путём, как и деревья отрезков для суммы/минимума/максимума: достаточно просто хранить в каждой вершине дерева НОД/НОК всех чисел в соответствующем отрезке массива.

Подсчёт количества нулей, поиск k-го нуля

В этой задаче мы хотим научиться отвечать на запрос количества нулей в заданном отрезке массива, а также на запрос нахождения k-го нулевого элемента.

Снова немного изменим данные, хранящиеся в дереве отрезков: будем хранить теперь в массиве $t[]$ количество нулей, встречающихся в соответствующих отрезках массива. Понятно, как поддерживать и использовать эти данные в функциях $\text{build}, \text{sum}, \text{update}$, — тем самым мы решили задачу о количестве нулей в заданном отрезке массива.

Теперь научимся решать задачу о поиске позиции k-го вхождения нуля в массиве. Для этого будем спускаться по дереву отрезков, начиная с корня, и переходя каждый раз в левого или правого сына в зависимости от того, в каком из отрезков находится искомый k-ый ноль. В самом деле, чтобы понять, в какого сына нам надо переходить, достаточно посмотреть на значение, записанное в левом сыне: если оно больше либо равно k, то переходить надо в левого сына (потому что в его отрезке есть как минимум k нулей), а иначе — переходить в правого сына.

При реализации можно отсечь случай, когда k-го нуля не существует, ещё при входе в функцию, вернув в качестве ответа, например, -1.

```
int find_kth (int v, int tl, int tr, int k) {
        if (k > t[v])
                return -1;
        if (tl == tr)
                return tl;
        int tm = (tl + tr) / 2;
        if (t[v*2] >= k)
                return find_kth (v*2, tl, tm, k);
        else
                return find_kth (v*2+1, tm+1, tr, k - t[v*2]);
}
```

Поиск префикса массива с заданной суммой

Задача такая: требуется по данному значению x быстро найти такое i, что сумма первых i элементов массива $a[]$ больше либо равна x (считая, что массив $a[]$ содержит только неотрицательные числа).

Эту задачу можно решать бинарным поиском, вычисляя каждый раз внутри него сумму на том или ином префиксе массива, но это приведёт к решению за время $O(\log^2 n)$.

Вместо этого можно воспользоваться той же самой идеей, что и в предыдущем пункте, и искать искомую позицию одним спуском по дереву: переходя каждый раз в левого или правого сына в зависимости от величины суммы в левом сыне. Тогда ответ на запрос поиска будет представлять собой один такой спуск по дереву, а, следовательно, будет выполняться за $O(\log n)$.

Поиск подотрезка с максимальной суммой

По-прежнему на вход даётся массив $a[0 \dots n-1]$, и поступают запросы (l, r), которые означают: найти такой подотрезок $a[l' \dots r']$, что $l \le l', r' \le r$, и сумма этого отрезка $a[l' \dots r']$ максимальна.
Запросы модификации отдельных элементов массива допускаются. Элементы массива могут быть отрицательными (и, например, если все числа отрицательны, то оптимальным подотрезком будет пустой — на нём сумма равна нулю).

Это весьма нетривиальное обобщение дерева отрезков получается следующим образом. Будем хранить в каждой вершине дерева отрезков четыре величины: сумму на этом отрезке, максимальную сумму среди всех

префиксов этого отрезка, максимальную сумму среди всех суффиксов, а также максимальную сумму подотрезка на нём. Иными словами, для каждого отрезка дерева отрезков ответ на нём уже предпосчитан, а также дополнительно ответ посчитан среди всех отрезков, упирающихся в левую границу отрезка, а также среди всех отрезков, упирающихся в правую границу.

Как же построить дерево отрезков с такими данными? Снова подойдём к этому с рекурсивной точки зрения: пусть для текущей вершины все четыре значения в левом сыне и в правом сыне уже подсчитаны, посчитаем их теперь для самой вершины. Заметим, что ответ в самой вершине равен:

- либо ответу в левом сыне, что означает, что лучший подотрезок в текущей вершине целиком помещается в отрезок левого сына,
- либо ответу в правом сыне, что означает, что лучший подотрезок в текущей вершине целиком помещается в отрезок правого сына,
- либо сумме максимального суффикса в левом сыне и максимального префикса в правом сыне, что означает, что лучший подотрезок лежит своим началом в левом сыне, а концом — в правом.

Значит, ответ в текущей вершине равен максимуму из этих трёх величин. Пересчитывать же максимальную сумму на префиксах и суффиксах ещё проще. Приведём реализацию функции $\mathrm{combine}$, которой будут передаваться две структуры data, содержащие в себе данные о левом и правом сыновьях, и которая возвращает данные в текущей вершине.

```
struct data {
        int sum, pref, suff, ans;
};

data combine (data l, data r) {
        data res;
        res.sum = l.sum + r.sum;
        res.pref = max (l.pref, l.sum + r.pref);
        res.suff = max (r.suff, r.sum + l.suff);
        res.ans = max (max (l.ans, r.ans), l.suff + r.pref);
        return res;
}
```

Таким образом, мы научились строить дерево отрезков. Отсюда легко получить и реализацию запроса модификации: как и в самом простом дереве отрезков, мы выполняем пересчёт значений во всех изменившихся вершинах дерева отрезков, для чего используем всё ту же функцию $\mathrm{combine}$. Для вычисления значений дерева в листьях также вспомогательную функцию $\mathrm{make_data}$, которая возвращает структуру data, вычисленную по одному числу val.

```
data make_data (int val) {
        data res;
        res.sum = val;
        res.pref = res.suff = res.ans = max (0, val);
        return res;
}

void build (int a[], int v, int tl, int tr) {
        if (tl == tr)
                t[v] = make_data (a[tl]);
        else {
                int tm = (tl + tr) / 2;
                build (a, v*2, tl, tm);
                build (a, v*2+1, tm+1, tr);
                t[v] = combine (t[v*2], t[v*2+1]);
        }
}

void update (int v, int tl, int tr, int pos, int new_val) {
        if (tl == tr)
                t[v] = make_data (new_val);
        else {
                int tm = (tl + tr) / 2;
                if (pos <= tm)
                        update (v*2, tl, tm, pos, new_val);
                else
                        update (v*2+1, tm+1, tr, pos, new_val);
                t[v] = combine (t[v*2], t[v*2+1]);
        }
}
```

Осталось разобраться с ответом на запрос. Для этого мы так же, как и раньше, спускаемся по дереву, разбивая тем самым отрезок запроса $[l \ldots r]$ на несколько подотрезков, совпадающих с отрезками дерева отрезков, и объединяем ответы в них в единый ответ на всю задачу. Тогда понятно, что работа ничем не отличается от работы обычного дерева отрезков, только надо вместо простого суммирования/минимума/максимума значений использовать функцию $\mathrm{combine}$. Приведённая ниже реализация немного отличается от реализации запроса sum: она не допускает случаев, когда левая граница l запроса превышает правую границу r (иначе возникнут неприятные случаи — какую структуру data возвращать, когда отрезок запроса пустой?..).

```
data query (int v, int tl, int tr, int l, int r) {
        if (l == tl && tr == r)
                return t[v];
        int tm = (tl + tr) / 2;
        if (r <= tm)
                return query (v*2, tl, tm, l, r);
        if (l > tm)
                return query (v*2+1, tm+1, tr, l, r);
        return combine (
                query (v*2, tl, tm, l, tm),
                query (v*2+1, tm+1, tr, tm+1, r)
        );
}
```

Сохранение всего подмассива в каждой вершине дерева отрезков

Это отдельный подраздел, стоящий особняком от остальных, поскольку в каждой вершине дерева отрезков мы будем хранить не какую-то сжатую информацию об этом подотрезке (сумму, минимум, максимум и т.п.), а **все** элементы массива, лежащие в этом подотрезке. Таким образом, корень дерева отрезков будет хранить все элементы массива, левый сын корня — первую половину массива, правый сын корня — вторую половину, и так далее.

Самый простой вариант применения этой техники — когда в каждой вершине дерева отрезков хранится отсортированный список всех чисел, встречающихся в соответствующем отрезке. В более сложных вариантах хранятся не списки, а какие-либо структуры данных, построенные над этими списками (set, map и т.д.). Но все эти методы объединяет то, что в каждой вершине дерева отрезков хранится некая структура данных, имеющая в памяти размер порядка длины соответствующего отрезка.

Первый естественный вопрос, встающий при рассмотрении деревьев отрезков этого класса — это **объём потребляемой памяти**. Утверждается, что если в каждой вершине дерева отрезков хранится список всех встречающихся на этом отрезке чисел, либо любая другая структура данных размера того же порядка, то в сумме всё дерево отрезков будет занимать $O(n \log n)$ ячеек памяти. Почему это так? Потому что каждое число $a[i]$ попадает в $O(\log n)$ отрезков дерева отрезков (хотя бы потому, что высота дерева отрезков есть $O(\log n)$).

Итак, несмотря на кажущуюся расточительность такого дерева отрезков, он потребляет памяти не сильно больше обычного дерева отрезков.

Ниже описано несколько типичных применений такой структуры данных. Стоит сразу отметить явную аналогию деревьев отрезков этого типа с **двумерными структурами данных** (собственно, в каком-то смысле это и есть двумерная структура данных, но с довольно ограниченными возможностями).

Поиск наименьшего числа, больше либо равного заданного, в указанном отрезке. Запросов модификации нет

Требуется отвечать на запросы следующего вида: (l, r, x), что означает найти минимальное число в отрезке $a[l \ldots r]$, которое больше либо равно x.

Построим дерево отрезков, в котором в каждой вершине будем хранить отсортированный список всех чисел, встречающихся на соответствующем отрезке. Например, корень будет содержать массив $a[]$ в отсортированном виде. Как построить такое дерево отрезков максимально эффективно? Для этого подойдём к задаче, как обычно, с точки зрения рекурсии: пусть для левого и правого сыновей текущей вершины эти списки уже построены, и нам требуется построить этот список для текущей вершины. При такой постановке вопроса становится почти очевидно, что это можно сделать за линейное время: нам просто надо объединить два отсортированных списка в один, что делается одним проходом по ним с двумя указателями. Пользователям C++ ещё проще, потому что этот алгоритм слияния уже включён в стандартную библиотеку STL:

```
vector<int> t[4*MAXN];

void build (int a[], int v, int tl, int tr) {
        if (tl == tr)
                t[v] = vector<int> (1, a[tl]);
        else {
                int tm = (tl + tr) / 2;
```

```
                build (a, v*2, tl, tm);
                build (a, v*2+1, tm+1, tr);
                merge (t[v*2].begin(), t[v*2].end(), t[v*2+1].begin(), t[v*2
+1].end(),
                        back_inserter (t[v]));
        }
}
```

Мы уже знаем, что построенное таким образом дерево отрезков будет занимать $O(n \log n)$ памяти. А благодаря такой реализации время его построения также есть величина $O(n \log n)$ — ведь каждый список строится за линейное относительно его размера время. (Кстати говоря, здесь прослеживается очевидная аналогия с алгоритмом **сортировки слиянием**: только здесь мы сохраняем информацию со всех этапов работы алгоритма, а не только итог.)

Теперь рассмотрим **ответ на запрос**. Будем спускаться по дереву, как это делает стандартный ответ на запрос в дереве отрезков, разбивая наш отрезок $a[l \ldots r]$ на несколько подотрезков (порядка $O(\log n)$ штук). Понятно, что ответ на всю задачу равен минимуму среди ответов на каждом из этих подотрезков. Поймём теперь, как отвечать на запрос на одном таком подотрезке, совпадающем с некоторой вершиной дерева.

Итак, мы пришли в какую-то вершину дерева отрезков и хотим посчитать ответ на ней, т.е. найти минимальное число, больше либо равное данного x. Для этого нам всего лишь надо выполнить **бинарный поиск** по списку, посчитанному в этой вершине дерева, и вернуть первое число из этого списка, больше либо равное x.

Таким образом, ответ на запрос в одном подотрезке происходит за $O(\log n)$, а весь запрос обрабатывается за время $O(\log^2 n)$.

```
int query (int v, int tl, int tr, int l, int r, int x) {
        if (l > r)
                return INF;
        if (l == tl && tr == r) {
                vector<int>::iterator pos = lower_bound (t[v].begin(), t[v].
end(), x);
                if (pos != t[v].end())
                        return *pos;
                return INF;
        }
        int tm = (tl + tr) / 2;
        return min (
                query (v*2, tl, tm, l, min(r,tm), x),
                query (v*2+1, tm+1, tr, max(l,tm+1), r, x)
        );
}
```

Константа INF равна некоторому большому числу, заведомо большему, чем любое число в массиве. Она несёт смысл "ответа в заданном отрезке не существует".

Поиск наименьшего числа, больше либо равного заданного, в указанном отрезке. Допускаются запросы модификации

Задача аналогична предыдущей, только теперь разрешены запросы модификации: обработать присвоение $a[i] = y$.

Решение также аналогично решению предыдущей задачи, только вместо простых списков в каждой вершине дерева отрезков мы будем хранить сбалансированный список, который позволяет быстро искать требуемое число, удалять его, а также вставлять новое число. Учитывая, что вообще говоря число во входном массиве могут повторяться, оптимальным выбором является структура данных STL $\mathrm{multiset}$.

Построение такого дерева отрезков происходит примерно так же, как и в предыдущей задаче, только теперь надо объединять не отсортированные списки, а $\mathrm{multiset}$, что приведёт к тому, что асимптотика построения ухудшится до $n \log^2 n$ (хотя, по-видимому, красно-чёрные деревья позволяют выполнить слияние двух деревьев за линейное время, однако библиотека STL этого не гарантирует).

Ответ на **запрос поиска** вообще практически эквивалентен приведённому выше коду, только теперь $\mathrm{lower_bound}$ надо вызывать от $t[v]$.

Наконец, **запрос модификации**. Для его обработки мы должны спуститься по дереву, внеся изменения во все $O(\log n)$ списков, содержащих затрагиваемый элемент. Мы просто удаляем старое значение этого элемента (не забыв, что нам не надо удалить вместе с ним все повторы этого числа) и вставляем его новое значение.

```
void update (int v, int tl, int tr, int pos, int new_val) {
        t[v].erase (t[v].find (a[pos]));
        t[v].insert (new_val);
```

```
        if (tl != tr) {
                int tm = (tl + tr) / 2;
                if (pos <= tm)
                        update (v*2, tl, tm, pos, new_val);
                else
                        update (v*2+1, tm+1, tr, pos, new_val);
        }
        else
                a[pos] = new_val;
}
```

Обработка этого запроса происходит также за время $O(\log^2 n)$.

Поиск наименьшего числа, больше либо равного заданного, в указанном отрезке. Ускорение с помощью техники "частичного каскадирования"

Улучшим время ответа на запрос поиска до времени $O(\log n)$ с помощью применения техники "**частичного каскадирования**" ("fractional cascading").

Частичное каскадирование — это простой приём, который позволяет улучшить время работы нескольких бинарных поисков, ведущихся по одному и тому же значению. В самом деле, ответ на запрос поиска заключается в том, что мы разбиваем нашу задачу на несколько подзадач, каждая из которых затем решается бинарным поиском по числу x. Частичное каскадирование позволяет заменить все эти двоичные поиски на один.

Простейшим и самым наглядным примером частичного каскадирования является **следующая задача**: есть несколько отсортированных списков чисел, и мы должны в каждом списке найти первое число, больше либо равное заданного.

Если бы мы решали задачу "в лоб", то вынуждены были бы запустить бинарный поиск по каждому из этих списков, что, если этих списков много, становится весьма существенным фактором: если всего списков k, то асимптотика получится $O(k \log(n/k))$, где n — суммарный размер всех списков (асимптотика такова, потому что худший случай — когда все списки примерно равны друг другу по длине, т.е. равны n/k).

Вместо этого, мы могли бы объединить все эти списки в один отсортированный список, в котором для каждого числа n_i будем хранить список позиций: позицию в первом списке первого числа, больше либо равного n_i, аналогичную позицию во втором списке, и так далее. Иными словами, для каждого встречающегося числа мы храним вместе с этим числом результаты двоичных поисков по нему в каждом из списков. В таком случае асимптотика ответа на запрос получается $O(\log n + k)$, что существенно лучше, однако мы вынуждены расплачиваться большим потреблением памяти: а именно, нам требуется $O(nk)$ ячеек памяти.

Техника частичного каскадирования идёт дальше в решении этой задачи и добивается потребления памяти $O(n)$ при том же самом времени ответа на запрос $O(\log n + k)$. (Для этого мы храним не один большой список длины n, а снова возвращаемся к k спискам, но вместе с каждым списком храним каждый второй элемент из следующего списка; нам снова придётся вместе с каждым числом записывать его позицию в обоих списках (текущем и следующем), однако это позволит по-прежнему эффективно отвечать на запрос: мы делаем двоичный поиск по первому списку, а затем идём по этим спискам по порядку, переходя каждый раз в следующий список с помощью предпосчитанных указателей, и делая один шаг влево, учитывая тем самым, что половина чисел следующего списка учтена не была).

Но нам в нашем приложении к дереву отрезков **не нужна** полная мощь этой техники. Дело в том, что список в текущей вершине содержит все числа, которые могут встречаться в левом и правом сыновьях. Поэтому, чтобы избежать бинарного поиска по списку сына, нам достаточно для каждого списка в дереве отрезков посчитать для каждого числа его позиции в списках левого и правого сыновей (точнее, позиции первого числа, меньшего либо равного текущему).

Таким образом, вместо обычного списка всех чисел мы храним список троек: само число, позиция в списке левого сына, позиция в списке правого сына. Это позволит нам за $O(1)$ определять позицию в списке левого или правого сына, вместо того чтобы делать двоичный список по нему.

Проще всего эту технику применять к задаче, когда запросы модификации отсутствуют, — тогда эти позиции представляют собой просто числа, а подсчитывать их при построении дерева очень легко внутри алгоритма слияния двух отсортированных последовательностей.

В случае, когда разрешены запросы модификации, всё несколько усложняется: эти позиции теперь надо хранить в виде итераторов внутри $\mathrm{multiset}$, а при запросе обновления — правильно уменьшать/увеличивать для тех элементов, для которых это требуется.

Так или иначе, задача уже сводится к чисто реализационным тонкостям, а основная идея — замена $O(\log n)$ бинарных поисков одним бинарным поиском по списку в корне дерева — описана полностью.

Другие возможные направления

Заметим, что эта техника подразумевает под собой целый класс возможных приложений — всё определяется структурой данных, выбранной для хранения в каждой вершине дерева. Выше были рассмотрены приложения

с использованием vector и $\mathrm{multiset}$, в то время как вообще использоваться может любая другая компактная структура данных: другое дерево отрезков (об этом немного сказано ниже в разделе о многомерных деревьях отрезков), дерево Фенвика, декартово дерево и т.д.

Обновление на отрезке

Выше рассматривались только задачи, когда запрос модификации затрагивает единственный элемент массива. На самом деле, дерево отрезков позволяет делать запросы, которые применяются к целым отрезкам подряд идущих элементов, причём выполнять эти запросы за то же время $O(\log n)$.

Прибавление на отрезке

Начнём рассмотрение деревьев отрезков такого рода с самого простого случая: запрос модификации представляет собой прибавление ко всем числам на некотором подотрезке $a[l \ldots r]$ некоторого числа x. Запрос чтения — по-прежнему считывание значения некоторого числа $a[i]$.

Чтобы делать запрос прибавления эффективно, будем хранить в каждой вершине дерева отрезков, сколько надо прибавить ко всем числам этого отрезка целиком. Например, если приходит запрос "прибавить ко всему массиву $a[0 \ldots n-1]$ число 2", то мы поставим в корне дерева число 2. Тем самым мы сможем обрабатывать запрос прибавления на любом подотрезке эффективно, вместо того чтобы изменять все $O(n)$ значений.

Если теперь приходит запрос чтения значения того или иного числа, то нам достаточно спуститься по дереву, просуммировав все встреченные по пути значения, записанные в вершинах дерева.

```
void build (int a[], int v, int tl, int tr) {
        if (tl == tr)
                t[v] = a[tl];
        else {
                int tm = (tl + tr) / 2;
                build (a, v*2, tl, tm);
                build (a, v*2+1, tm+1, tr);
        }
}

void update (int v, int tl, int tr, int l, int r, int add) {
        if (l > r)
                return;
        if (l == tl && tr == r)
                t[v] += add;
        else {
                int tm = (tl + tr) / 2;
                update (v*2, tl, tm, l, min(r,tm), add);
                update (v*2+1, tm+1, tr, max(l,tm+1), r, add);
        }
}

int get (int v, int tl, int tr, int pos) {
        if (tl == tr)
                return t[v];
        int tm = (tl + tr) / 2;
        if (pos <= tm)
                return t[v] + get (v*2, tl, tm, pos);
        else
                return t[v] + get (v*2+1, tm+1, tr, pos);
}
```

Присвоение на отрезке

Пусть теперь запрос модификации представляет собой присвоение всем элементам некоторого отрезка $a[l \ldots r]$ некоторого значения p. В качестве второго запроса будем рассматривать считывание значения массива $a[i]$.

Чтобы делать модификацию на целом отрезке, придётся в каждой вершине дерева отрезков хранить, покрашен ли этот отрезок целиком в какое-либо число или нет (и если покрашен, то хранить само это число). Это позволит нам делать "**запаздывающее**" **обновление** дерева отрезков: при запросе модификации мы, вместо того чтобы менять значения во множестве вершин дерева отрезков, поменяем только некоторые из них, оставив флаги "покрашен" для других отрезков, что означает, что весь этот отрезок вместе со своими подотрезками должен быть покрашен в этот цвет.

Итак, после выполнения запроса модификации дерево отрезков становится, вообще говоря, неактуальным — в нём остались недовыполненными некоторые модификации.

Например, если пришёл запрос модификации "присвоить всему массиву $a[0 \ldots n-1]$ какое-то число", то в дереве отрезков мы сделаем единственное изменение — пометим корень дерева, что он покрашен целиком в это число. Остальные же вершины дерева останутся неизменёнными, хотя на самом деле всё дерево должно быть покрашено в одно и то же число.

Предположим теперь, что в том же дереве отрезков пришёл второй запрос модификации — покрасить первую половину массива $a[0 \ldots n/2]$ в какое-либо другое число. Чтобы обработать такой запрос, мы должны покрасить целиком левого сына корня в этот новый цвет, однако перед тем как сделать это, мы должны разобраться с корнем дерева. Тонкость здесь в том, что в дереве должно сохраниться, что правая половина покрашена в старый цвет, а в данный момент в дереве никакой информации для правой половины не сохранено.

Выход таков: произвести **проталкивание** информации из корня, т.е. если корень дерева был покрашен в какое-либо число, то покрасить в это число его правого и левого сына, а из корня эту отметку убрать. После этого мы можем спокойно красить левого сына корня, не теряя никакой нужной информации.

Обобщая, получаем: при любых запросах с таким деревом (запрос модификации или чтения) во время спуска по дереву мы всегда должны делать проталкивание информации из текущей вершины в обоих её сыновей. Можно понимать это так, что при спуске по дереву мы применяем запаздывающие модификации, но ровно настолько, насколько это необходимо (чтобы не ухудшить асимптотику с $O(\log n)$).

При реализации это означает, что нам надо сделать функцию push, которой будет передаваться вершина дерева отрезков, и она будет производить проталкивание информации из этой вершины в обоих её сыновей. Вызывать эту функцию следует в самом начале функций обработки запросов (но не вызывать её из листьев, ведь из листа проталкивать информацию не надо, да и некуда).

```
void push (int v) {
        if (t[v] != -1) {
                t[v*2] = t[v*2+1] = t[v];
                t[v] = -1;
        }
}

void update (int v, int tl, int tr, int l, int r, int color) {
        if (l > r)
                return;
        if (l == tl && tr == r)
                t[v] = color;
        else {
                push (v);
                int tm = (tl + tr) / 2;
                update (v*2, tl, tm, l, min(r,tm), color);
                update (v*2+1, tm+1, tr, max(l,tm+1), r, color);
        }
}

int get (int v, int tl, int tr, int pos) {
        if (tl == tr)
                return t[v];
        push (v);
        int tm = (tl + tr) / 2;
        if (pos <= tm)
                return get (v*2, tl, tm, pos);
        else
                return get (v*2+1, tm+1, tr, pos);
}
```

Функцию get можно было бы реализовать и по-другому: не делать в ней запаздывающих обновлений, а сразу возвращать ответ, как только она попадает в вершину дерева отрезков, целиком покрашенную в тот или иной цвет.

Прибавление на отрезке, запрос максимума

Пусть теперь запросом модификации снова будет запрос прибавления ко всем числам некоторого подотрезка одного и того же числа, а запросом чтения будет нахождение максимума в некотором подотрезке.

Тогда в каждой вершине дерева отрезков надо будет дополнительно хранить максимум на всём этом подотрезке. Но тонкость здесь заключается в том, как надо пересчитывать эти значения.

Например, пусть произошёл запрос "прибавить ко всей первой половине, т.е. $a[0 \ldots n/2]$, число 2". Тогда в дереве это отразится записью числа 2 в левого сына корня. Как теперь посчитать новое значение максимума в левом сыне и в корне? Здесь становится важно не запутаться — какой максимум хранится в вершине дерева: максимум без учёта прибавления на всей этой вершине, или же с учётом его. Выбрать можно любой из этих подходов, но главное — последовательно использовать его везде. Например, при первом подходе максимум в корне будет получаться

как максимум из двух чисел: максимум в левом сыне плюс прибавление в левом сыне, и максимум в правом сыне плюс прибавление в нём. При втором же подходе максимум в корне будет получаться как прибавление в корне плюс максимум из максимумов в левом и правом сыновьях.

Другие направления

Здесь были рассмотрены только базовые применения деревьев отрезков в задачах с модификациями на отрезке. Остальные задачи получаются на основе тех же самых идей, что описаны здесь.

Важно только быть очень аккуратным при работе с отложенными модификациями: следует помнить, что даже если в текущей вершине мы уже "протолкнули" отложенную модификацию, то в левом и правом сыновьях, скорее всего, этого ещё не сделали. Поэтому часто необходимым является вызывать push также от левого и правого сыновей текущей вершины, либо же аккуратно учитывать отложенные модификации в них.

Обобщение на большие размерности

Дерево отрезков обобщается вполне естественным образом на двумерный и вообще многомерный случай. Если в одномерном случае мы разбивали индексы массива на отрезки, то в двумерном случае теперь будем сначала разбивать всё по первым индексам, а для каждого отрезка по первым индексам — строить обычное дерево отрезков по вторым индексам. Таким образом, основная идея решения — это вкладывание деревьев отрезков по вторым индексам внутрь дерева отрезков по первым индексам.

Поясним эту идею на примере конкретной задачи.

Двумерное дерево отрезков в простейшем варианте

Дана прямоугольная матрица $a[0 \dots n-1, 0 \dots m-1]$, и поступают запросы поиска суммы (или минимума/максимума) на некоторых подпрямоугольниках $a[x_1 \dots x_2, y_1 \dots y_2]$, а также запросы модификации отдельных элементов матрицы (т.е. запросы вида $a[x][y] = p$).

Итак, будем строить двумерное дерево отрезков: сначала дерево отрезков по первой координате (x), затем — по второй (y).

Чтобы **процесс построения** был более понятен, можно на время забыть, что исходный массив был двумерным, и оставить только первую координату. Будем строить обычное одномерное дерево отрезков, работая только с первой координатой. Но в качестве значения каждого отрезка мы будем записывать не какое-то число, как в одномерном случае, а целое дерево отрезков: т.е. в этот момент мы вспоминаем, что у нас есть ещё и вторая координата; но т.к. в этот момент уже зафиксировано, что первая координата есть некоторый отрезок $[l \dots r]$, то мы фактически работаем с такой полосой $a[l \dots r, 0 \dots m-1]$, и для неё строим дерево отрезков.

Приведём реализацию операции построения двумерного дерева. Она фактически представляет собой два отдельных блока: построение дерева отрезков по координате x (build_x) и по координате y (build_y). Если первая функция почти ничем не отличается от обычного одномерного дерева, то вторая вынуждена разбираться отдельно с двумя случаями: когда текущий отрезок по первой координате ($[tlx \dots trx]$) имеет единичную длину, и когда — длину, большую единицы. В первом случае мы просто берём нужное значение из матрицы $a[][]$, а во втором — объединяем значения двух деревьев отрезков из левого сына и правого сына по координате x.

```
void build_y (int vx, int lx, int rx, int vy, int ly, int ry) {
        if (ly == ry)
                if (lx == rx)
                        t[vx][vy] = a[lx][ly];
                else
                        t[vx][vy] = t[vx*2][vy] + t[vx*2+1][vy];
        else {
                int my = (ly + ry) / 2;
                build_y (vx, lx, rx, vy*2, ly, my);
                build_y (vx, lx, rx, vy*2+1, my+1, ry);
                t[vx][vy] = t[vx][vy*2] + t[vx][vy*2+1];
        }
}

void build_x (int vx, int lx, int rx) {
        if (lx != rx) {
                int mx = (lx + rx) / 2;
                build_x (vx*2, lx, mx);
                build_x (vx*2+1, mx+1, rx);
        }
        build_y (vx, lx, rx, 1, 0, m-1);
}
```

Такое дерево отрезков занимает по-прежнему линейный объём памяти, но уже с большей константой: $16nm$

ячеек памяти. Понятно, что строится оно описанной выше процедурой $\mathrm{build_x}$ тоже за линейное время.

Перейдем теперь к **обработке запросов**. Отвечать на двумерный запрос будем по тому же самому принципу: сначала разбивать запрос по первой координате, а затем, когда мы дошли до какой-то вершины дерева отрезков по первой координате — вызывать запрос от соответствующего дерева отрезков по второй координате.

```
int sum_y (int vx, int vy, int tly, int try_, int ly, int ry) {
        if (ly > ry)
                return 0;
        if (ly == tly && try_ == ry)
                return t[vx][vy];
        int tmy = (tly + try_) / 2;
        return sum_y (vx, vy*2, tly, tmy, ly, min(ry,tmy))
                + sum_y (vx, vy*2+1, tmy+1, try_, max(ly,tmy+1), ry);
}

int sum_x (int vx, int tlx, int trx, int lx, int rx, int ly, int ry) {
        if (lx > rx)
                return 0;
        if (lx == tlx && trx == rx)
                return sum_y (vx, 1, 0, m-1, ly, ry);
        int tmx = (tlx + trx) / 2;
        return sum_x (vx*2, tlx, tmx, lx, min(rx,tmx), ly, ry)
                + sum_x (vx*2+1, tmx+1, trx, max(lx,tmx+1), rx, ly, ry);
}
```

Эта функция работает за время $O(\log n \log m)$, поскольку она сначала спускается по дереву по первой координате, а для каждой пройденной вершины этого дерева — делает запрос у обычного дерева отрезков по второй координате.

Наконец, рассмотрим **запрос модификации**. Мы хотим научиться модифицировать дерево отрезков в соответствии с изменением значения какого-либо элемента $a[x][y] = p$. Понятно, что изменения произойдут только в тех вершинах первого дерева отрезков, которые накрывают координату x (а таких будет $O(\log n)$), а для деревьев отрезков, соответствующих им — изменения будут только в тех вершинах, которые накрывают координату y (и таких будет $O(\log m)$). Поэтому реализация запроса модификации не будет сильно отличаться от одномерного случая, только теперь мы сначала спускаемся по первой координате, а затем — по второй.

```
void update_y (int vx, int lx, int rx, int vy, int ly, int ry, int x, int
y, int new_val) {
        if (ly == ry) {
                if (lx == rx)
                        t[vx][vy] = new_val;
                else
                        t[vx][vy] = t[vx*2][vy] + t[vx*2+1][vy];
        }
        else {
                int my = (ly + ry) / 2;
                if (y <= my)
                        update_y (vx, lx, rx, vy*2, ly, my, x, y, new_val);
                else
                        update_y (vx, lx, rx, vy*2+1, my+1, ry, x, y, new_val);
                t[vx][vy] = t[vx][vy*2] + t[vx][vy*2+1];
        }
}

void update_x (int vx, int lx, int rx, int x, int y, int new_val) {
        if (lx != rx) {
                int mx = (lx + rx) / 2;
                if (x <= mx)
                        update_x (vx*2, lx, mx, x, y, new_val);
                else
                        update_x (vx*2+1, mx+1, rx, x, y, new_val);
        }
        update_y (vx, lx, rx, 1, 0, m-1, x, y, new_val);
}
```

Сжатие двумерного дерева отрезков

Пусть задача следующая: есть n точек на плоскости, заданных своими координатами (x_i, y_i), и поступают запросы вида "посчитать количество точек, лежащих в прямоугольнике $((x_1, y_1), (x_2, y_2))$". Понятно, что в случае

такой задачи становится неоправданно расточительным строить двумерное дерево отрезков с $O(n^2)$ элементами. Большая часть этой памяти будет потрачена впустую, поскольку каждая отдельно взятая точка может попасть только в $O(\log n)$ отрезков дерева отрезков по первой координате, а, значит, суммарный "полезный" размер всех деревьев отрезков по второй координате есть величина $O(n \log n)$.

Тогда поступим следующим образом: в каждой вершине дерева отрезков по первой координате будем хранить дерево отрезков, построенное только по тем вторым координатам, которые встречаются в текущем отрезке первых координат. Иными словами, при построении дерева отрезков внутри какой-то вершины с номером vx и границами tlx, trx мы будем рассматривать только те точки, которые попадают в этот отрезок $x \in [tlx; trx]$, и строить дерево отрезков только над ними.

Тем самым мы добьёмся того, что каждое дерево отрезков по второй координате будет занимать ровно столько памяти, сколько и должно. В итоге суммарный **объём памяти** уменьшится до $O(n \log n)$. **Отвечать на запрос** мы будем по-прежнему за $O(\log^2 n)$, просто теперь при вызове запроса от дерева отрезков по второй координате мы должны будем сделать бинарный поиск по второй координате, но асимптотику это не ухудшит.

Но расплатой станет невозможность делать произвольный **запрос модификации**: в самом деле, если появится новая точка, то это приведёт к тому, что мы должны будем в каком-либо дереве отрезков по второй координате добавить новый элемент в середину, что эффективно сделать невозможно.

В завершение отметим, что сжатое описанным образом двумерное дерево отрезков становится практически **эквивалентным** описанной выше модификации одномерного дерева отрезков (см. "Сохранение всего подмассива в каждой вершине дерева отрезков"). В частности, получается, что описываемое здесь двумерное дерево отрезков — это просто частный случай сохранения подмассива в каждой вершине дерева, где подмассив сам хранится в виде дерева отрезков. Отсюда следует, что если приходится отказываться от двумерного дерева отрезков по причине невозможности выполнения того или иного запроса, то имеет смысл попробовать заменить вложенное дерево отрезков на какую-либо более мощную структуру данных, например, декартово дерево.

Дерево отрезков с сохранением истории его значений (улучшение до persistent-структуры данных)

Persistent-структурой данных называется такая структура данных, которая при каждой модификации запоминает своё предыдущее состояние. Это позволяет при необходимости обратиться к любой интересующей нас версии этой структуры данных и выполнить запрос на ней.

Дерево отрезков является одной из тех структур данных, которая может быть превращена в persistent-структуру данных (разумеется, мы рассматриваем эффективную persistent-структуру, а не такую, которая копирует всю себя целиком перед каждым обновлением).

В самом деле, любой запрос изменения в дереве отрезков приводит к изменению данных в $O(\log n)$ вершинах, причём вдоль пути, начинающегося из корня. Значит, если мы будем хранить дерево отрезков на указателях (т. е. указатели на левого и правого сыновей сделать указателями, хранящимися в вершине), то при запросе обновления мы должны просто вместо изменения имеющихся вершин создать новые вершины, ссылки из которых направлять на старые вершины. Тем самым, при запросе обновления будет создано $O(\log n)$ новых вершин, в том числе будет создан новый корень дерева отрезков, а вся предыдущая версия дерева, подвешенная за старый корень, останется без изменений.

Приведём пример реализации для простейшего дерева отрезков: когда есть только запрос подсчёта суммы на подотрезке и запрос модификации единственного числа.

```
struct vertex {
        vertex * l, * r;
        int sum;

        vertex (int val)
                : l(NULL), r(NULL), sum(val)
        { }

        vertex (vertex * l, vertex * r)
                : l(l), r(r), sum(0)
        {
                if (l)  sum += l->sum;
                if (r)  sum += r->sum;
        }
};

vertex * build (int a[], int tl, int tr) {
        if (tl == tr)
                return new vertex (a[tl]);
        int tm = (tl + tr) / 2;
        return new vertex (
```

```
                build (a, tl, tm),
                build (a, tm+1, tr)
        );
}

int get_sum (vertex * t, int tl, int tr, int l, int r) {
        if (l > r)
                return 0;
        if (l == tl && tr == r)
                return t->sum;
        int tm = (tl + tr) / 2;
        return get_sum (t->l, tl, tm, l, min(r,tm))
                + get_sum (t->r, tm+1, tr, max(l,tm+1), r);
}

vertex * update (vertex * t, int tl, int tr, int pos, int new_val) {
        if (tl == tr)
                return new vertex (new_val);
        int tm = (tl + tr) / 2;
        if (pos <= tm)
                return new vertex (
                                update (t->l, tl, tm, pos, new_val),
                                t->r
                        );
        else
                return new vertex (
                                t->l,
                                update (t->r, tm+1, tr, pos, new_val)
                        );
}
```

С помощью этого подхода можно превратить в persistent-структуру данных практически любое дерево отрезков.

Декартово дерево (treap, дерамида)

Декартово дерево - это структура данных, объединяющая в себе бинарное дерево поиска и бинарную кучу (отсюда и второе её название: treap (tree+heap) и дерамида (дерево+пирамида).

Более строго, это структура данных, которая хранит пары (X,Y) в виде бинарного дерева таким образом, что она является бинарным деревом поиска по x и бинарной пирамидой по y. Предполагая, что все X и все Y являются различными, получаем, что если некоторый элемент дерева содержит (X_0,Y_0), то у всех элементов в левом поддереве $X < X_0$, у всех элементов в правом поддереве $X > X_0$, а также и в левом, и в правом поддереве имеем: $Y < Y_0$.

Дерамиды были предложены Сиделем (Siedel) и Арагоном (Aragon) в 1996 г.

Преимущества такой организации данных

В том применении, которое мы рассматриваем (мы будем рассматривать дерамиды, поскольку декартово дерево - это фактически более общая структура данных), X'ы являются ключами (и одновременно значениями, хранящимися в структуре данных), а Y'и - называются **приоритетами**. Если бы приоритетов не было, то было бы обычное бинарное дерево поиска по X, и заданному набору X'ов могло бы соответствовать много деревьев, некоторые из которых являются вырожденными (например, в виде цепочки), а потому чрезвычайно медленными (основные операции выполнялись бы за O (N)).

В то же время, **приоритеты** позволяют **однозначно** указать дерево, которое будет построено (разумеется, не зависящее от порядка добавления элементов) (это доказывается соответствующей теоремой). Теперь очевидно, что если **выбирать приоритеты случайно**, то этим мы добьёмся построения **невырожденных** деревьев в среднем случае, что обеспечит асимптотику O (log N) в среднем. Отсюда и понятно ещё одно название этой структуры данных - **рандомизированное бинарное дерево поиска**.

Операции

Итак, treap предоставляет следующие операции:

- `Insert (X, Y)` - за O (log N) в среднем
  Выполняет добавление в дерево нового элемента.
  Возможен вариант, при котором значение приоритета Y не передаётся функции, а выбирается случайно (правда, нужно учесть, что оно не должно совпадать ни с каким другим Y в дереве).
- `Search (X)` - за O (log N) в среднем
  Ищет элемент с указанным значением ключа X. Реализуется абсолютно так же, как и для обычного бинарного дерева поиска.
- `Erase (X)` - за O (log N) в среднем
  Ищет элемент и удаляет его из дерева.
- `Build` $(X_1, ..., X_N)$ - за O (N)

  Строит дерево из списка значений. Эту операцию можно реализовать за линейное время (в предположении, что значения $X_1, ..., X_N$ отсортированы), но здесь эта реализация рассматриваться не будет.

  Здесь будет использоваться только простейшая реализация - в виде последовательных вызовов Insert, т.е. за O (N log N).
- `Union` (T_1, T_2) - за O (M log (N/M)) в среднем

  Объединяет два дерева, в предположении, что все элементы различны (впрочем, эту операцию можно реализовать с той же асимптотикой, если при объединении нужно удалять повторяющиеся элементы).
- `Intersect` (T_1, T_2) - за O (M log (N/M)) в среднем

  Находит пересечение двух деревьев (т.е. их общие элементы). Здесь реализация этой операции не будет рассматриваться.

Кроме того, за счёт того, что декартово дерево является и бинарным деревом поиска по своим значениям, к нему применимы такие операции, как нахождение K-го по величине элемента, и, наоборот, определение номера элемента.

Описание реализации

С точки зрения реализации, каждый элемент содержит в себе X, Y и указатели на левого L и правого R сына.

Для реализации операций понадобится реализовать две вспомогательные операции: Split и Merge.

`Split (T, X)` - разделяет дерево T на два дерева L и R (которые являются возвращаемым значением) таким образом, что L содержит все элементы, меньшие по ключу X, а R содержит все элементы, большие X. Эта операция выполняется за O (log N). Реализация её довольно проста - очевидная рекурсия.

`Merge` (T_1, T_2) - объединяет два поддерева T_1 и T_2, и возвращает это новое дерево. Эта операция также реализуется за O (log N). Она работает в предположении, что T_1 и T_2 обладают соответствующим порядком (все

значения X в первом меньше значений X во втором). Таким образом, нам нужно объединить их так, чтобы не нарушить порядок по приоритетам Y. Для этого просто выбираем в качестве корня то дерево, у которого Y в корне больше, и рекурсивно вызываем себя от другого дерева и соответствующего сына выбранного дерева.

Теперь очевидна реализация `Insert (X, Y)`. Сначала спускаемся по дереву (как в обычном бинарном дереве поиска по X), но останавливаемся на первом элементе, в котором значение приоритета оказалось меньше Y. Мы нашли позицию, куда будем вставлять наш элемент. Теперь вызываем Split (X) от найденного элемента (от элемента вместе со всем его поддеревом), и возвращаемые ею L и R записываем в качестве левого и правого сына добавляемого элемента.

Также понятна и реализация `Erase (X)`. Спускаемся по дереву (как в обычном бинарном дереве поиска по X), ища удаляемый элемент. Найдя элемент, мы просто вызываем Merge от его левого и правого сыновей, и возвращаемое ею значение ставим на место удаляемого элемента.

Операцию `Build` реализуем за O (N log N) просто с помощью последовательных вызовов Insert.

Наконец, операция `Union` (T_1, T_2). Теоретически её асимптотика O (M log (N/M)), однако на практике она работает очень хорошо, вероятно, с весьма малой скрытой константой. Пусть, не теряя общности, T_1->Y > T_2->Y, т. е. корень T_1 будет корнем результата. Чтобы получить результат, нам нужно объединить деревья T_1->L, T_1->R и T_2 в два таких дерева, чтобы их можно было сделать сыновьями T_1. Для этого вызовем Split (T_2, T_1->X), тем самым мы разобъём T_2 на две половинки L и R, которые затем рекурсивно объединим с сыновьями T_1: Union (T_1->L, L) и Union (T_1->R, R), тем самым мы построим левое и правое поддеревья результата.

Реализация

Реализуем все описанные выше операции. Здесь для удобства введены другие обозначения - приоритет обозначается prior, значения - key.

```
struct item {
        int key, prior;
        item * l, * r;
        item() { }
        item (int key, int prior) : key(key), prior(prior), l(NULL), r(NULL) { }
};
typedef item * pitem;

void split (pitem t, int key, pitem & l, pitem & r) {
        if (!t)
                l = r = NULL;
        else if (key < t->key)
                split (t->l, key, l, t->l),  r = t;
        else
                split (t->r, key, t->r, r),  l = t;
}

void insert (pitem & t, pitem it) {
        if (!t)
                t = it;
        else if (it->prior > t->prior)
                split (t, it->key, it->l, it->r),  t = it;
        else
                insert (it->key < t->key ? t->l : t->r, it);
}

void merge (pitem & t, pitem l, pitem r) {
        if (!l || !r)
                t = l ? l : r;
        else if (l->prior > r->prior)
                merge (l->r, l->r, r),  t = l;
        else
                merge (r->l, l, r->l),  t = r;
}

void erase (pitem & t, int key) {
        if (t->key == key)
                merge (t, t->l, t->r);
        else
                erase (key < t->key ? t->l : t->r, key);
}

pitem unite (pitem l, pitem r) {
```

```
        if (!l || !r)  return l ? l : r;
        if (l->prior < r->prior)  swap (l, r);
        pitem lt, rt;
        split (r, l->key, lt, rt);
        l->l = unite (l->l, lt);
        l->r = unite (l->r, rt);
        return l;
}
```

Поддержка размеров поддеревьев

Чтобы расширить функциональность декартового дерева, очень часто необходимо для каждой вершины хранить количество вершин в её поддереве - некое поле int cnt в структуре item. Например, с его помощью легко будет найти за O (log N) K-ый по величине элемент дерева, или, наоборот, за ту же асимптотику узнать номер элемента в отсортированном списке (реализация этих операций ничем не будет отличаться от их реализации для обычных бинарных деревьев поиска).

При изменении дерева (добавлении или удалении элемента и т.д.) должны соответствующим образом меняться и cnt некоторых вершин. Реализуем две функции - функция cnt() будет возвращать текущее значение cnt или 0, если вершина не существует, а функция upd_cnt() будет обновлять значение cnt для указанной вершины, при условии, что для её сыновей l и r эти cnt уже корректно обновлены. Тогда, понятно, достаточно добавить вызовы функции upd_cnt () в конец каждой из функций insert, erase, split, merge, чтобы постоянно поддерживать корректные значения cnt.

```
int cnt (pitem t) {
        return t ? t->cnt : 0;
}

void upd_cnt (pitem t) {
        if (t)
                t->cnt = 1 + cnt(t->l) + cnt (t->r);
}
```

Построение декартового дерева за O (N) в оффлайн

TODO

Неявные декартовы деревья

Неявное декартово дерево - это простая модификация обычного декартового дерева, которая, тем не менее, оказывается очень мощной структурой данных. Фактически, неявное декартово дерево можно воспринимать как массив, над которым можно реализовать следующие операции (все за O (log N) в режиме онлайн):

- Вставка элемента в массив в любую позицию
- Удаление произвольного элемента
- Сумма, минимум/максимум на произвольном отрезке, и т.д.
- Прибавление, покраска на отрезке
- Переворот (перестановка элементов в обратном порядке) на отрезке

Ключевая идея заключается в том, что в качестве ключей key следует использовать **индексы** элементов в массиве. Однако явно хранить эти значения key мы не будем (иначе, например, при вставке элемента пришлось бы изменять key в O (N) вершинах дерева).

Заметим, что фактически в данном случае ключ для какой-то вершины - это количество вершин, меньших неё. Следует заметить, что вершины, меньшие данной, находятся не только в её левом поддереве, но и, возможно, в левых поддеревьях её предков. Более строго, **неявный ключ** для некоторой вершины t равен количеству вершин cnt(t->l) в левом поддереве этой вершины плюс аналогичные величины cnt(p->l)+1 для каждого предка p этой вершины, при условии, что t находится в правом поддереве для p.

Ясно, как теперь быстро вычислять для текущей вершины её неявный ключ. Поскольку во всех операциях мы приходим в какую-либо вершину, спускаясь по дереву, мы можем просто накапливать эту сумму, передавая её функции. Если мы идём в левое поддерево - накапливаемая сумма не меняется, а если идём в правое - увеличивается на cnt(t->l)+1.

Приведём новые реализации функций split и merge:

```
void merge (pitem & t, pitem l, pitem r) {
        if (!l || !r)
                t = l ? l : r;
        else if (l->prior > r->prior)
                merge (l->r, l->r, r),  t = l;
        else
                merge (r->l, l, r->l),  t = r;
```

```
        upd_cnt (t);
}

void split (pitem t, pitem & l, pitem & r, int key, int add = 0) {
        if (!t)
                return void( l = r = 0 );
        int cur_key = add + cnt(t->l); // вычисляем неявный ключ
        if (key <= cur_key)
                split (t->l, l, t->l, key, add),  r = t;
        else
                split (t->r, t->r, r, key, add + 1 + cnt(t->l)),  l = t;
        upd_cnt (t);
}
```

Теперь перейдём к реализации различных дополнительных операций на неявных декартовых деревьях:

- **Вставка** элемента.
  Пусть нам надо вставить элемент в позицию pos. Разобьём декартово дерево на две половинки: соответствующую массиву [0..pos-1] и массиву [pos..sz]; для этого достаточно вызвать split (t, t1, t2, pos). После этого мы можем объединить дерево t1 с новой вершиной; для этого достаточно вызвать merge (t1, t1, new_item) (нетрудно убедиться в том, что все предусловия для merge выполнены). Наконец, объединим два дерева t1 и t2 обратно в дерево t - вызовом merge (t, t1, t2).
- **Удаление** элемента.
  Здесь всё ещё проще: достаточно найти удаляемый элемент, а затем выполнить merge для его сыновей l и r, и поставить результат объединения на место вершины t. Фактически, удаление из неявного декартова дерева не отличается от удаления из обычного декартова дерева.
- **Сумма**/**минимум** и т.п. на отрезке.
  Во-первых, для каждой вершины создадим дополнительное поле f в структуре item, в котором будет храниться значение целевой функции для поддерева этой вершины. Такое поле легко поддерживать, для этого надо поступить аналогично поддержке размеров cnt (создать функцию, вычисляющую значение этого поля, пользуясь его значениями для сыновей, и вставить вызовы этой функции в конце всех функций, меняющих дерево).
  Во-вторых, нам надо научиться отвечать на запрос на произвольном отрезке [A;B]. Научимся выделять из дерева его часть, соответствующую отрезку [A;B]. Нетрудно понять, что для этого достаточно сначала вызвать split (t, t1, t2, A), а затем split (t2, t2, t3, B-A+1). В результате дерево t2 и будет состоять из всех элементов в отрезке [A;B], и только них. Следовательно, ответ на запрос будет находиться в поле f вершины t2. После ответа на запрос дерево надо восстановить вызовами merge (t, t1, t2) и merge (t, t, t3).
- **Прибавление**/**покраска** на отрезке.
  Здесь мы поступаем аналогично предыдущему пункту, но вместо поля f будем хранить поле add, которое и будет содержать прибавляемую величину (или величину, в которую красят всё поддерево этой вершины). Перед выполнением любой операции эту величину add надо "протолкнуть" - т.е. соответствующим образом изменить t->l->add и t->r->add, а у себя значение add снять. Тем самым мы добьёмся того, что ни при каких изменениях дерева информация не будет потеряна.
- **Переворот** на отрезке.
  Этот пункт почти аналогичен предыдущему - нужно ввести поле bool rev, которое ставить в true, когда требуется произвести переворот в поддереве текущей вершины. "Проталкивание" поля rev заключается в том, что мы обмениваем местами сыновья текущей вершины, и ставим этот флаг для них.

Реализация. Приведём для примера полную реализацию неявного декартова дерева с переворотом на отрезке. Здесь для каждой вершины также хранится поле value - собственно значение элемента, стоящего в массиве на текущей позиции. Приведена также реализация функции output(), которая выводит массив, соответствующий текущему состоянию неявного декартова дерева.

```
typedef struct item * pitem;
struct item {
        int prior, value, cnt;
        bool rev;
        pitem l, r;
};

int cnt (pitem it) {
        return it ? it->cnt : 0;
}

void upd_cnt (pitem it) {
        if (it)
                it->cnt = cnt(it->l) + cnt(it->r) + 1;
}

void push (pitem it) {
        if (it && it->rev) {
                it->rev = false;
```

```
                swap (it->l, it->r);
                if (it->l)  it->l->rev ^= true;
                if (it->r)  it->r->rev ^= true;
        }
}

void merge (pitem & t, pitem l, pitem r) {
        push (l);
        push (r);
        if (!l || !r)
                t = l ? l : r;
        else if (l->prior > r->prior)
                merge (l->r, l->r, r),  t = l;
        else
                merge (r->l, l, r->l),  t = r;
        upd_cnt (t);
}

void split (pitem t, pitem & l, pitem & r, int key, int add = 0) {
        if (!t)
                return void( l = r = 0 );
        push (t);
        int cur_key = add + cnt(t->l);
        if (key <= cur_key)
                split (t->l, l, t->l, key, add),  r = t;
        else
                split (t->r, t->r, r, key, add + 1 + cnt(t->l)),  l = t;
        upd_cnt (t);
}

void reverse (pitem t, int l, int r) {
        pitem t1, t2, t3;
        split (t, t1, t2, l);
        split (t2, t2, t3, r-l+1);
        t2->rev ^= true;
        merge (t, t1, t2);
        merge (t, t, t3);
}

void output (pitem t) {
        if (!t)  return;
        push (t);
        output (t->l);
        printf ("%d ", t->value);
        output (t->r);
}
```

Модификация стека и очереди для извлечения минимума за O (1)

Здесь мы рассмотрим три задачи: модифицирование стека с добавлением извлечения наименьшего элемента за O (1), аналогичное модифицирование очереди, а также применение их к задаче нахождения минимума во всех подотрезках фиксированной длины данного массива за O (N).

Модификация стека

Требуется добавить возможность извлечения минимума из стека за O (1), сохранив такой же асимптотику добавления и удаления элементов из стека.

Для этого будем хранить в стеке не сами элементы, а пары: элемент и минимум в стеке, начиная с этого элемента и ниже. Иными словами, если представить стек как массив пар, то

```
stack[i].second = min { stack[j].first }
                  j = 0..i
```

Понятно, что тогда нахождение минимума во всём стеке будет заключаться просто во взятии значения stack.top().second.

Также очевидно, что при добавлении нового элемента в стек величина second будет равна min (stack.top().second, new_element). Удаление элемента из стека ничем не отличается от удаления из обычного стека, поскольку удаляемый элемент никак не мог повлиять на значения second для оставшихся элементов.

Реализация:

```
stack< pair<int,int> > st;
```

- Добавление элемента:

```
int minima = st.empty() ? new_element : min (new_element, st.top().second);
st.push (make_pair (new_element, minima));
```

- Извлечение элемента:

```
int result = st.top().first;
st.pop();
```

- Нахождение минимума:

```
minima = st.top().second;
```

Модификация очереди. Способ 1

Здесь рассмотрим простой способ модификации очереди, но имеющий тот недостаток, что модифицированная очередь реально может хранить не все элементы (т.е. при извлечении элемента из очереди нам надо будет знать значение элемента, который мы хотим извлечь). Ясно, что это весьма специфичная ситуация (обычно очередь нужна как раз для того, чтобы узнавать очередной элемент, а не наоборот), однако этот способ привлекателен своей простотой. Также этот метод применим к задаче о нахождении минимума в подотрезках (см. ниже).

Ключевая идея заключается в том, чтобы реально хранить в очереди не все элементы, а только нужные нам для определения минимума. А именно, пусть очередь представляет собой неубывающую последовательность чисел (т.е. в голове хранится наименьшее значение), причём, разумеется, не произвольную, а всегда содержащую минимум. Тогда минимум во всей очереди всегда будет являться первым её элементом. Перед добавлением нового элемента в очередь достаточно произвести "срезку": пока в хвосте очереди находится элемент, больший нового элемента, будем удалять этот элемент из очереди; затем добавим новый элемент в конец очереди. Тем самым мы, с одной стороны, не нарушим порядка, а с другой стороны, не потеряем текущий элемент, если он на каком-либо последующем шаге окажется минимумом. Но при извлечении элемента из головы очереди его там, вообще говоря, может уже не оказаться - наша модифицированная очередь могла выкинуть этот элемент в процессе перестроения. Поэтому при удалении элемента нам надо знать значение извлекаемого элемента - если элемент с этим значением находится в голове очереди, то извлекаем его; иначе просто ничего не делаем.

Рассмотрим реализацию вышеописанных операций:

```
deque<int> q;
```

- Нахождение минимума:

```
current_minimum = q.front();
```

- Добавление элемента:

```
while (!q.empty() && q.back() > added_element)
        q.pop_back();
q.push_back (added_element);
```

- Извлечение элемента:

```
if (!q.empty() && q.front() == removed_element)
        q.pop_front();
```

Понятно, что в среднем время выполнения всех этих операций есть O (1).

Модификация очереди. Способ 2

Рассмотрим здесь другой способ модификации очереди для извлечения минимума за O (1), который несколько более сложен для реализации, однако лишён основного недостатка предыдущего метода: все элементы очереди реально сохраняются в ней, и, в частности, при извлечении элемента не требуется знать его значение.

Идея заключается в том, чтобы свести задачу к задаче на стеках, которая уже была нами решена. Научимся моделировать очередь с помощью двух стеков.

Заведём два стека: s1 и s2; разумеется, имеются в виду стеки, модифицированные для нахождения минимума за O (1). Добавлять новые элементы будет всегда в стек s1, а извлекать элементы - только из стека s2. При этом, если при попытке извлечения элемента из стека s2 он оказался пустым, просто перенесём все элементы из стека s1 в стек s2 (при этом элементы в стеке s2 получатся уже в обратном порядке, что нам и нужно для извлечения элементов; стек s1 же станет пустым). Наконец, нахождение минимума в очереди будет фактически заключаться в нахождении минимума из минимума в стеке s1 и минимума в стеке s2.

Тем самым, мы выполняем все операции по-прежнему за O (1) (по той простой причине, что каждый элемент в худшем случае 1 раз добавляется в стек s1, 1 раз переносится в стек s2 и 1 раз извлекается из стека s2).

Реализация:

```
stack< pair<int,int> > s1, s2;
```

- Нахождение минимума:

```
if (s1.empty() || s2.empty())
        current_minimum = s1.empty ? s2.top().second : s1.top().second;
else
        current_minimum = min (s1.top().second, s2.top().second);
```

- Добавление элемента:

```
int minima = s1.empty() ? new_element : min (new_element, s1.top().second);
s1.push (make_pair (new_element, minima));
```

- Извлечение элемента:

```
if (s2.empty())
        while (!s1.empty()) {
                int element = s1.top().first;
                s1.pop();
                int minima = s2.empty() ? element : min (element, s2.top
().second);
                s2.push (make_pair (element, minima));
        }
result = s2.top().first;
s2.pop();
```

Задача нахождения минимума во всех подотрезках фиксированной длины данного массива

Пусть дан массив A длины N, и дано число M ≤ N. Требуется найти минимум в каждом подотрезке длины M данного массива, т.е. найти:

```
min A[i],    min A[i],    min A[i],    ...,    min A[i]
0≤i≤M-1      1≤i≤M        2≤i≤M+1              N-M≤i≤N-1
```

Решим эту задачу за линейное время, т.е. O (N).

Для этого достаточно завести очередь, модифицированную для нахождения минимума за O (1), что было рассмотрено нами выше, причём в данной задаче подойдёт любой из двух методов реализации такой очереди. Далее решение уже понятно: добавим в очередь первые M элементов массива, найдём в ней минимум и выведем его, затем добавим в очередь следующий элемент, и извлечём из неё первый элемент массива, снова выведем минимум, и т.д. Поскольку все операции с очередью выполняются в среднем за константное время, то и асимптотика всего алгоритма получится O (N).

Стоит заметить, что реализация модифицированной очереди первым методом проще, однако для неё, вероятно, потребуется хранить весь массив (поскольку на i-ом шаге потребуется знать i-ый и (i-M)-ый элементы массива). При реализации очереди вторым методом массив A хранить явно не понадобится - только узнавать очередной, i-ый элемент массива.

Рандомизированная куча

Рандомизированная куча (randomized heap) — это куча, которая за счёт применения генератора случайных чисел позволяет выполнять все необходимые операции за логарифмическое ожидаемое время.

Кучей называется бинарное дерево, для любой вершины которого справедливо, что значение в этой вершине меньше либо равно значений во всех её потомках (это куча для минимума; разумеется, симметрично можно определить кучу для максимума). Таким образом, в корне кучи всегда находится минимум.

Стандартный набор операций, определяемый для куч, следующий:

- Добавление элемента
- Нахождение минимума
- Извлечение минимума (удаление его из дерева и возврат его значения)
- Слияние двух куч (возвращается куча, содержащая элементы обеих куч; дубликаты не удаляются)
- Удаление произвольного элемента (при известной позиции в дереве)

Рандомизированная куча позволяет выполнять все эти операции за ожидаемое время $O(\log n)$ при очень простой реализации.

Структура данных

Сразу опишем структуру данных, описывающую бинарную кучу:

```
struct tree {
        T value;
        tree * l, * r;
};
```

В вершине дерева хранится значение value некоторого типа T, для которого определён оператор сравнения ($\mathrm{operator}\ <$). Кроме того, хранятся указатели на левого и правого сыновей (которые равны 0, если соответствующий сын отсутствует).

Выполнение операций

Нетрудно понять, что все операции над кучей сводятся к одной операции: **слиянию** двух куч в одну. Действительно, добавление элемента в кучу равносильно слиянию этой кучи с кучей, состоящей из единственного добавляемого элемента. Нахождение минимума вообще не требует никаких действий — минимумом просто является корень кучи. Извлечение минимума эквивалентно тому, что куча заменяется результатом слияния левого и правого поддерева корня. Наконец, удаление произвольного элемента аналогично удалению минимума: всё поддерево с корнем в этой вершине заменяется результатом слияния двух поддеревьев-сыновей этой вершины.

Итак, нам фактически надо реализовать только операцию слияния двух куч, все остальные операции тривиально сводятся к этой операции.

Пусть даны две кучи T_1 и T_2, требуется вернуть их объединение. Понятно, что в корне каждой из этих куч находятся их минимумы, поэтому в корне результирующей кучи будет находиться минимум из этих двух значений. Итак, мы сравниваем, в корне какой из куч находится меньшее значение, его помещаем в корень результата, а теперь мы должны объединить сыновей выбранной вершины с оставшейся кучей. Если мы по какому-то признаку выберем одного из двух сыновей, то тогда нам надо будет просто объединить поддерево в корне с этим сыном с кучей. Таким образом, мы снова пришли к операции слияния. Рано или поздно этот процесс остановится (на это понадобится, понятно, не более чем сумма высот куч).

Таким образом, чтобы достичь логарифмической асимптотики в среднем, нам надо указать способ выбора одного из двух сыновей с тем, чтобы в среднем длина проходимого пути получалась бы порядка логарифма от количества элементов в куче. Нетрудно догадаться, что производить этот выбор мы будем **случайно**, таким образом, реализация операции слияния получается такой:

```
tree * merge (tree * t1, tree * t2) {
        if (!t1 || !t2)
                return t1 ? t1 : t2;
        if (t2->value < t1->value)
                swap (t1, t2);
        if (rand() & 1)
                swap (t1->l, t1->r);
```

```
        t1->l = merge (t1->l, t2);
        return t1;
}
```

Здесь сначала проверяется, если хотя бы одна из куч пуста, то никаких действий по слиянию производить не надо. Иначе, мы делаем, чтобы куча $t1$ была кучей с меньшим значением в корне (для чего обмениваем $t1$ и $t2$, если надо). Наконец, мы считаем, что вторую кучу $t2$ будем сливать с левым сыном корня кучи $t1$, поэтому мы случайным образом обмениваем левого и правого сыновей, а затем выполняем слияние левого сына и второй кучи.

Асимптотика

Введём случайную величину $h(T)$, обозначающую **длину случайного пути** от корня до листа (длина в числе рёбер). Понятно, что алгоритм merge выполняется за $O(h(T1) + h(T2))$ операций. Поэтому для исследования асимптотики алгоритма надо исследовать случайную величину $h(T)$.

Математическое ожидание

Утверждается, что математическое ожидание $h(T)$ оценивается сверху логарифмом от числа n вершин в этой куче:

$$Eh(T) \le \log(n+1)$$

Доказывается это легко по индукции. Пусть L и R — соответственно левое и правое поддеревья корня кучи T, а n_L и n_R — количества вершин в них (понятно, что $n = n_L + n_R + 1$).

Тогда справедливо:

$$\begin{aligned}
&Eh(T) = 1 + \frac{1}{2}(Eh(L) + Eh(R)) \le 1 + \frac{1}{2}(\log(n_L+1) + \log(n_R+1)) = \\
&= 1 + \log\sqrt{(n_L+1)(n_R+1)} = \log 2\sqrt{(n_L+1)(n_R+1)} \le \\
&\le \log\frac{2((n_L+1)+(n_R+1))}{2} = \log(n_L+n_R+2) = \log(n+1)
\end{aligned}$$

что и требовалось доказать.

Превышение ожидаемой оценки

Докажем, что вероятность превышения полученной выше оценки мала:

$$P\{h(T) > (c+1)\log n\} < \frac{1}{n^c}$$

для любой положительной константы c.

Обозначим через P множество путей от корня кучи до листьев, длина которых превосходит $(c+1)\log n$. Заметим, что для любого пути p длины $|p|$ вероятность того, в качестве случайного пути будет выбран именно он, равна $2^{-|p|}$. Тогда получаем:

$$P\{h(T) > (c+1)\log n\} = \sum_{p\in P} 2^{-|p|} < \sum_{p\in P} 2^{-(c+1)\log n} = |P| n^{-(c+1)} \le n^{-c}$$

что и требовалось доказать.

Асимптотика алгоритма

Таким образом, алгоритм merge, а, значит, и все остальные выраженные через него операции, выполняется за $O(\log n)$ в среднем.

Более того, для любой положительной константы ϵ найдётся такая положительная константа c, что вероятность того, что операция потребует больше чем $c\log n$ операций, меньше $n^{-\epsilon}$ (это в некотором смысле описывает худшее поведение алгоритма).

Задача RMQ (Range Minimum Query - минимум на отрезке)

Дан массив A[1..N]. Поступают запросы вида (L, R), на каждый запрос требуется найти минимум в массиве A, начиная с позиции L и заканчивая позицией R.

Приложения

Помимо непосредственного применения в самых разных задачах, можно отметить следующие:

- Задача LCA (наименьший общий предок)

Решение

Задача RMQ решается с помощью структур данных.

Из описанных на сайте структур данных можно выбрать:

- Sqrt-**декомпозиция** - отвечает на запрос за O (sqrt (N)), препроцессинг за O (N).
  Преимущество в том, что это очень простая структура данных. Недостаток - асимптотика.
- **Дерево отрезков** - отвечает на запрос за O (log N), препроцессинг за O (N).
  Преимущество - хорошая асимптотика. Недостаток - бОльший объём кода по сравнению с другими структурами данных.
- **Дерево Фенвика** - отвечает на запрос за O (log N), препроцессинг за O (N log N)
  Преимущество - очень быстро пишется и работает тоже очень быстро. Но значительный недостаток - дерево Фенвика может отвечать только на запросы с L = 1, что для многих приложений неприменимо.

Примечание. "Препроцессинг" - это предварительная обработка массива A, фактически это построение структуры данных для данного массива.

Теперь предположим, что массив A **может изменяться** в процессе работы (т.е. также будут поступать запросы об изменении значения в некотором отрезке [L;R]). Тогда полученную задачу можно решить с помощью Sqrt-**декомпозиции** и **Дерева отрезков**.

Нахождение наидлиннейшей возрастающей подпоследовательности

Условие задачи следующее. Дан массив из n чисел: $a[0 \ldots n-1]$. Требуется найти в этой последовательности строго возрастающую подпоследовательность наибольшей длины.

Формально это выглядит следующим образом: требуется найти такую последовательность индексов $i_1 \ldots i_k$, что:

$$i_1 < i_2 < \ldots < i_k,$$
$$a[i_1] < a[i_2] < \ldots < a[i_k].$$

В данной статье рассматриваются различные алгоритмы решения данной задачи, а также некоторые задачи, которые можно свести к данной задаче.

Решение за $O(n^2)$: метод динамического программирования

Динамическое программирование — это весьма общая методика, позволяющая решать огромный класс задач. Здесь мы рассмотрим эту методику применительно к нашей конкретной задаче.

Научимся сначала искать **длину** наидлиннейшей возрастающей подпоследовательности, а восстановлением самой подпоследовательности займёмся чуть позже.

Динамическое программирование для поиска длины ответа

Для этого давайте научимся считать массив $d[0 \ldots n-1]$, где $d[i]$ — это длина наидлиннейшей возрастающей подпоследовательности, оканчивающейся именно в элементе с индексом i. Массив этот (он и есть — сама динамика) будем считать постепенно: сначала $d[0]$, затем $d[1]$ и т.д. В конце, когда этот массив будет подсчитан нами, ответ на задачу будет равен максимуму в массиве $d[]$.

Итак, пусть текущий индекс — i, т.е. мы хотим посчитать значение $d[i]$, а все предыдущие значения $d[0] \ldots d[i-1]$ уже подсчитаны. Тогда заметим, что у нас есть два варианта:

- либо $d[i] = 1$, т.е. искомая подпоследовательность состоит только из числа $a[i]$.
- либо $d[i] > 1$. Тогда перед числом $a[i]$ в искомой подпоследовательности стоит какое-то другое число. Давайте переберём это число: это может быть любой элемент $a[j] (j = 0 \ldots i-1)$, но такой, что $a[j] < a[i]$. Пусть мы рассматриваем какой-то текущий индекс j. Поскольку динамика $d[j]$ для него уже подсчитана, получается, что это число $a[j]$ вместе с числом $a[i]$ даёт ответ $d[j]+1$. Таким образом, $d[i]$ можно считать по такой формуле:

$$d[i] = \max_{\substack{j=0 \ldots i-1, \\ a[j]<a[i]}} (d[j]+1).$$

Объединяя эти два варианта в один, получаем окончательный алгоритм для вычисления $d[i]$:

$$d[i] = \max \left(1, \max_{\substack{j=0 \ldots i-1, \\ a[j]<a[i]}} (d[j]+1)\right).$$

Этот алгоритм — и есть сама динамика.

Реализация

Приведём реализацию описанного выше алгоритма, которая находит и выводит длину наидлиннейшей возрастающей подпоследовательности:

```
int d[MAXN]; // константа MAXN равна наибольшему возможному значению n

for (int i=0; i<n; ++i) {
        d[i] = 1;
        for (int j=0; j<i; ++j)
                if (a[j] < a[i])
                        d[i] = max (d[i], 1 + d[j]);
}
```

```
int ans = d[0];
for (int i=0; i<n; ++i)
        ans = max (ans, d[i]);
cout << ans << endl;
```

Восстановление ответа

Пока мы лишь научились искать длину ответа, но саму наидлиннейшую подпоследовательность мы вывести не можем, т. к. не сохраняем никакой дополнительной информации о том, где достигаются максимумы.

Чтобы суметь восстановить ответ, помимо динамики $d[0 \ldots n-1]$ надо также хранить вспомогательный массив $p[0 \ldots n-1]$ — то, в каком месте достигся максимум для каждого значения $d[i]$. Иными словами, индекс $p[i]$ будет обозначать тот самый индекс j, при котором получилось наибольшее значение $d[i]$. (Этот массив $p[]$ в динамическом программировании часто называют "массивом предков".)

Тогда, чтобы вывести ответ, надо просто идти от элемента с максимальным значением $d[i]$ по его предкам до тех пор, пока мы не выведем всю подпоследовательность, т.е. пока не дойдём до элемента со значением $d = 1$.

Реализация восстановления ответа

Итак, у нас изменится и код самой динамики, и добавится код, производящий вывод наидлиннейшей подпоследовательности (выводятся индексы элементов подпоследовательности, в 0-индексации).

Для удобства мы изначально положили индексы $p[i] = -1$: для элементов, у которых динамика получилась равной единице, это значение предка так и останется минус единицей, что чуть-чуть удобнее при восстановлении ответа.

```
int d[MAXN], p[MAXN]; // константа MAXN равна наибольшему возможному значению n

for (int i=0; i<n; ++i) {
        d[i] = 1;
        p[i] = -1;
        for (int j=0; j<i; ++j)
                if (a[j] < a[i])
                        if (1 + d[j] > d[i]) {
                                d[i] = 1 + d[j];
                                p[i] = j;
                        }
}

int ans = d[0],  pos = 0;
for (int i=0; i<n; ++i)
        if (d[i] > ans) {
                ans = d[i];
                pos = i;
        }
cout << ans << endl;

vector<int> path;
while (pos != -1) {
        path.push_back (pos);
        pos = p[pos];
}
reverse (path.begin(), path.end());
for (int i=0; i<(int)path.size(); ++i)
        cout << path[i] << ' ';
```

Альтернативный способ восстановления ответа

Впрочем, как почти всегда в случае динамического программирования, для восстановления ответа можно не хранить дополнительный массив предков $p[]$, а просто заново пересчитывая текущий элемент динамики и ища, на каком же индексе был достигнут максимум.

Этот способ при реализации приводит к чуть более длинному коду, однако взамен получаем экономию памяти и абсолютное совпадение логики программы в процессе подсчёта динамики и в процессе восстановления.

Решение за $O(n \log n)$: динамическое программирование

с двоичным поиском

Чтобы получить более быстрое решение задачи, построим другой вариант динамического программирования за $O(n^2)$, а затем поймём, как можно этот вариант ускорить до $O(n \log n)$.

Динамика теперь будет такой: пусть $d[i] (i = 0 \dots n)$ — это число, на которое оканчивается возрастающая подпоследовательность длины i (а если таких чисел несколько — то наименьшее из них).

Изначально мы полагаем $d[0] = -\infty$, а все остальные элементы $d[i] = \infty$.

Считать эту динамику мы будем постепенно, обработав число $a[0]$, затем $a[1]$, и т.д.

Приведём реализацию этой динамики за $O(n^2)$:

```
int d[MAXN];
d[0] = -INF;
for (int i=1; i<=n; ++i)
        d[i] = INF;

for (int i=0; i<n; i++)
        for (int j=1; j<=n; j++)
                if (d[j-1] < a[i] && a[i] < d[j])
                        d[j] = a[i];
```

Заметим теперь, что у этой динамики есть одно **очень важное свойство**: $d[i-1] \le d[i]$ для всех $i = 1 \dots n$. Другое свойство — что каждый элемент $a[i]$ обновляет максимум одну ячейку $d[j]$.

Таким образом, это означает, что обрабатывать очередное $a[i]$ мы можем за $O(\log n)$, сделав двоичный поиск по массиву $d[]$. В самом деле, мы просто ищем в массиве $d[]$ первое число, которое строго больше $a[i]$, и пытаемся произвести обновление этого элемента аналогично приведённой выше реализации.

Реализация за $O(n \log n)$

Воспользовавшись стандартным в языке C++ алгоритмом двоичного поиска $upper_bound$ (который возвращает позицию первого элемента, строго большего данного), получаем такую простую реализацию:

```
int d[MAXN];
d[0] = -INF;
for (int i=1; i<=n; ++i)
        d[i] = INF;

for (int i=0; i<n; i++) {
        int j = int (upper_bound (d.begin(), d.end(), a[i]) - d.begin());
        if (d[j-1] < a[i] && a[i] < d[j])
                d[j] = a[i];
}
```

Восстановление ответа

По такой динамике тоже можно восстановить ответ, для чего опять же помимо динамики $d[i]$ также надо хранить массив "предков" $p[i]$ — то, на элементе с каким индексом оканчивается оптимальная подпоследовательность длины i. Кроме того, для каждого элемента массива $a[i]$ надо будет хранить его "предка" — т.е. индекс того элемента, который должен стоять перед $a[i]$ в оптимальной подпоследовательности.

Поддерживая эти два массива по ходу вычисления динамики, в конце будет нетрудно восстановить искомую подпоследовательность.

(Интересно отметить, что применительно к данной динамике ответ можно восстанавливать только так, через массивы предков — а без них восстановить ответ после вычисления динамики будет невозможно. Это один из редких случаев, когда к динамике неприменим альтернативный способ восстановления — без массивов предков).

Решение за $O(n \log n)$: структуры данных

Если приведённый выше способ за $O(n \log n)$ весьма красив, однако не совсем тривиален идейно, то есть и другой путь: воспользоваться одной из известных простых структур данных.

В самом деле, давайте вернёмся к самой первой динамике, где состоянием являлась просто текущая позиция.

Текущее значение динамики $d[i]$ вычисляется как максимум значений $d[i]$ среди всех таких элементов j, что $a[j] < a[i]$.

Следовательно, если мы через $t[]$ обозначим такой **массив**, в который будем записывать значения динамики от чисел:

$$t[a[i]] = d[i],$$

то получается, что всё, что нам надо уметь — это искать **максимум на префиксе** массива t: $t[0 \ldots a[i] - 1]$.

Задача поиска максимума на префиксах массива (с учётом того, что массив может меняться) решается многими стандартными структурами данных, например, деревом отрезков или деревом Фенвика.

Воспользовавшись любой такой структурой данных, мы получим решение за $O(n \log n)$.

У этого способа решения есть явные **недостатки**: по длине и сложности реализации этот путь будет в любом случае хуже, чем описанная выше динамика за $O(n \log n)$. Кроме того, если входные числа $a[i]$ могут быть достаточно большими, то скорее всего их придётся сжимать (т.е. перенумеровывать от 0 до $n - 1$) — без этого многие стандартные структуры данных работать не смогут из-за высокого потребления памяти.

С другой стороны, у данного пути есть и **преимущества**. Во-первых, при таком способе решения не придётся задумываться о хитрой динамике. Во-вторых, этот способ позволяет решать некоторые обобщения нашей задачи (о них см. ниже).

Смежные задачи

Приведём здесь несколько задач, тесно связанных с задачей поиска наидлиннейшей возрастающей подпоследовательности.

Наидлиннейшая неубывающая подпоследовательность

Фактически, это та же самая задача, только теперь в искомой подпоследовательности допускаются одинаковые числа (т. е. мы должны найти нестрого возрастающую подпоследовательность).

Решение этой задачи по сути ничем не отличается от нашей исходной задачи, просто при сравнениях изменятся знаки неравенств, а также надо будет немного изменить двоичный поиск.

Количество наидлиннейших возрастающих подпоследовательностей

Для решения этой задачи можно использовать самую первую динамику за $O(n^2)$ либо подход с помощью структур данных для решения за $O(n \log n)$. И в том, и в том случае все изменения заключаются только в том, что помимо значения динамики $d[i]$ надо также хранить, сколькими способами это значение могло быть получено.

По всей видимости, способ решения через динамику за $O(n \log n)$ к данной задаче применить невозможно.

Наименьшее число невозрастающих подпоследовательностей, покрывающих данную последовательность

Условие таково. Дан массив из n чисел $a[0 \ldots n - 1]$. Требуется раскрасить его числа в наименьшее число цветов так, чтобы по каждому цвету получалась бы невозрастающая подпоследовательность.

Решение. Утверждается, что минимальное количество необходимых цветов равно длине наидлиннейшей возрастающей подпоследовательности.

Доказательство. Фактически, нам надо доказать **двойственность** этой задачи и задачи поиска наидлиннейшей возрастающей подпоследовательности.

Обозначим через x длину наидлиннейшей возрастающей подпоследовательности, а через y — искомое наименьшее число невозрастающих подпоследовательностей. Нам надо доказать, что $x = y$.

С одной стороны, понятно, почему не может быть $y < x$: ведь если у нас есть x строго возрастающих элементов, то никакие два из них не могли попасть в одну невозрастающую подпоследовательность, а, значит, $y \geq x$.

Покажем теперь, что, наоборот, y не может быть $> x$. Докажем это от противного: предположим, что $y > x$. Тогда рассмотрим любой оптимальный набор из y невозрастающих подпоследовательностей. Преобразуем этот набор таким образом: пока есть две таких подпоследовательности, что первая начинается раньше второй, но при этом первая начинается с числа, больше либо равного чем начало второй — отцепим это стартовое число от первой подпоследовательности и прицепим в начало второй. Таким образом, через какое-то конечное число шагов у нас останется y подпоследовательностей, причём их стартовые числа будут образовывать возрастающую подпоследовательность длины y. Но $y > x$, т.е. мы пришли к противоречию (ведь не может быть возрастающих подпоследовательностей длиннее x).

Таким образом, в самом деле, $y = x$, что и требовалось доказать.

Восстановление ответа. Утверждается, что само искомое разбиение на подпоследовательности можно искать жадно, т.е. идя слева направо и относя текущее число в ту подпоследовательность, которая сейчас заканчивается на минимальное число, больше либо равное текущему.

Задачи в online judges

Список задач, которые можно решить по данной тематике:

- MCCME #1793 "**Наибольшая возрастающая подпоследовательность за** O(n*log(n))" [сложность: низкая]
- TopCoder SRM 278 "500 IntegerSequence" [сложность: низкая]
- TopCoder SRM 233 "DIV2 1000 AutoMarket" [сложность: низкая]
- Всеукраинская олимпиада школьников по информатике — задача F "**Турист**" [сложность: средняя]
- Codeforces Beta Round #10 — задача D "**НОВП**" [сложность: средняя]
- ACM.TJU.EDU.CN 2707 "Greatest Common Increasing Subsequence" [сложность: средняя]
- SPOJ #57 "SUPPER. Supernumbers in a permutation" [сложность: средняя]
- ACM.SGU.RU #521 "North-East" [сложность: высокая]
- TopCoder Open 2004 — Round 4 — "1000. BridgeArrangement" [сложность: высокая]

K-ая порядковая статистика за O (N)

Пусть дан массив A длиной N и пусть дано число K. Задача заключается в том, чтобы найти в этом массиве K-ое по величине число, т.е. K-ую порядковую статистику.

Основная идея - использовать идеи алгоритма быстрой сортировки. Собственно, алгоритм несложный, сложнее доказать, что он работает в среднем за O (N), в отличие от быстрой сортировки.

Реализация в виде нерекурсивной функции:

```
template <class T>
T order_statistics (std::vector<T> a, unsigned n, unsigned k)
{
        using std::swap;
        for (unsigned l=1, r=n; ; )
        {

                if (r <= l+1)
                {
                        // текущая часть состоит из 1 или 2 элементов -
                        //       легко можем найти ответ
                        if (r == l+1 && a[r] < a[l])
                                swap (a[l], a[r]);
                        return a[k];
                }

                // упорядочиваем a[l], a[l+1], a[r]
                unsigned mid = (l + r) >> 1;
                swap (a[mid], a[l+1]);
                if (a[l] > a[r])
                        swap (a[l], a[r]);
                if (a[l+1] > a[r])
                        swap (a[l+1], a[r]);
                if (a[l] > a[l+1])
                        swap (a[l], a[l+1]);

                // выполняем разделение
                // барьером является a[l+1], т.е. медиана среди a[l], a[l+1],
a[r]
                unsigned
                        i = l+1,
                        j = r;
                const T
                        cur = a[l+1];
                for (;;)
                {
                        while (a[++i] < cur) ;
                        while (a[--j] > cur) ;
                        if (i > j)
                                break;
                        swap (a[i], a[j]);
                }

                // вставляем барьер
                a[l+1] = a[j];
                a[j] = cur;

                // продолжаем работать в той части,
                //       которая должна содержать искомый элемент
                if (j >= k)
                        r = j-1;
                if (j <= k)
                        l = i;
```

```
        }
}
```

Следует заметить, что в стандартной библиотеке C++ этот алгоритм уже реализован - он называется nth_element.

Нахождение наидлиннейшей возрастающей подпоследовательности

Условие задачи следующее. Дан массив из n чисел: $a[0 \ldots n-1]$. Требуется найти в этой последовательности строго возрастающую подпоследовательность наибольшей длины.

Формально это выглядит следующим образом: требуется найти такую последовательность индексов $i_1 \ldots i_k$, что:

$$i_1 < i_2 < \ldots < i_k,$$
$$a[i_1] < a[i_2] < \ldots < a[i_k].$$

В данной статье рассматриваются различные алгоритмы решения данной задачи, а также некоторые задачи, которые можно свести к данной задаче.

Решение за $O(n^2)$: метод динамического программирования

Динамическое программирование — это весьма общая методика, позволяющая решать огромный класс задач. Здесь мы рассмотрим эту методику применительно к нашей конкретной задаче.

Научимся сначала искать **длину** наидлиннейшей возрастающей подпоследовательности, а восстановлением самой подпоследовательности займёмся чуть позже.

Динамическое программирование для поиска длины ответа

Для этого давайте научимся считать массив $d[0 \ldots n-1]$, где $d[i]$ — это длина наидлиннейшей возрастающей подпоследовательности, оканчивающейся именно в элементе с индексом i. Массив этот (он и есть — сама динамика) будем считать постепенно: сначала $d[0]$, затем $d[1]$ и т.д. В конце, когда этот массив будет подсчитан нами, ответ на задачу будет равен максимуму в массиве $d[]$.

Итак, пусть текущий индекс — i, т.е. мы хотим посчитать значение $d[i]$, а все предыдущие значения $d[0] \ldots d[i-1]$ уже подсчитаны. Тогда заметим, что у нас есть два варианта:

- либо $d[i] = 1$, т.е. искомая подпоследовательность состоит только из числа $a[i]$.
- либо $d[i] > 1$. Тогда перед числом $a[i]$ в искомой подпоследовательности стоит какое-то другое число. Давайте переберём это число: это может быть любой элемент $a[j] (j = 0 \ldots i-1)$, но такой, что $a[j] < a[i]$. Пусть мы рассматриваем какой-то текущий индекс j. Поскольку динамика $d[j]$ для него уже подсчитана, получается, что это число $a[j]$ вместе с числом $a[i]$ даёт ответ $d[j]+1$. Таким образом, $d[i]$ можно считать по такой формуле:

$$d[i] = \max_{\substack{j=0 \ldots i-1, \\ a[j]<a[i]}} (d[j]+1).$$

Объединяя эти два варианта в один, получаем окончательный алгоритм для вычисления $d[i]$:

$$d[i] = \max \left(1, \max_{\substack{j=0 \ldots i-1, \\ a[j]<a[i]}} (d[j]+1)\right).$$

Этот алгоритм — и есть сама динамика.

Реализация

Приведём реализацию описанного выше алгоритма, которая находит и выводит длину наидлиннейшей возрастающей подпоследовательности:

```
int d[MAXN]; // константа MAXN равна наибольшему возможному значению n

for (int i=0; i<n; ++i) {
        d[i] = 1;
        for (int j=0; j<i; ++j)
                if (a[j] < a[i])
                        d[i] = max (d[i], 1 + d[j]);
}
```

```
int ans = d[0];
for (int i=0; i<n; ++i)
        ans = max (ans, d[i]);
cout << ans << endl;
```

Восстановление ответа

Пока мы лишь научились искать длину ответа, но саму наидлиннейшую подпоследовательность мы вывести не можем, т. к. не сохраняем никакой дополнительной информации о том, где достигаются максимумы.

Чтобы суметь восстановить ответ, помимо динамики $d[0 \ldots n-1]$ надо также хранить вспомогательный массив $p[0 \ldots n-1]$ — то, в каком месте достигся максимум для каждого значения $d[i]$. Иными словами, индекс $p[i]$ будет обозначать тот самый индекс j, при котором получилось наибольшее значение $d[i]$. (Этот массив $p[]$ в динамическом программировании часто называют "массивом предков".)

Тогда, чтобы вывести ответ, надо просто идти от элемента с максимальным значением $d[i]$ по его предкам до тех пор, пока мы не выведем всю подпоследовательность, т.е. пока не дойдём до элемента со значением $d = 1$.

Реализация восстановления ответа

Итак, у нас изменится и код самой динамики, и добавится код, производящий вывод наидлиннейшей подпоследовательности (выводятся индексы элементов подпоследовательности, в 0-индексации).

Для удобства мы изначально положили индексы $p[i] = -1$: для элементов, у которых динамика получилась равной единице, это значение предка так и останется минус единицей, что чуть-чуть удобнее при восстановлении ответа.

```
int d[MAXN], p[MAXN]; // константа MAXN равна наибольшему возможному значению n

for (int i=0; i<n; ++i) {
        d[i] = 1;
        p[i] = -1;
        for (int j=0; j<i; ++j)
                if (a[j] < a[i])
                        if (1 + d[j] > d[i]) {
                                d[i] = 1 + d[j];
                                p[i] = j;
                        }
}

int ans = d[0],  pos = 0;
for (int i=0; i<n; ++i)
        if (d[i] > ans) {
                ans = d[i];
                pos = i;
        }
cout << ans << endl;

vector<int> path;
while (pos != -1) {
        path.push_back (pos);
        pos = p[pos];
}
reverse (path.begin(), path.end());
for (int i=0; i<(int)path.size(); ++i)
        cout << path[i] << ' ';
```

Альтернативный способ восстановления ответа

Впрочем, как почти всегда в случае динамического программирования, для восстановления ответа можно не хранить дополнительный массив предков $p[]$, а просто заново пересчитывая текущий элемент динамики и ища, на каком же индексе был достигнут максимум.

Этот способ при реализации приводит к чуть более длинному коду, однако взамен получаем экономию памяти и абсолютное совпадение логики программы в процессе подсчёта динамики и в процессе восстановления.

Решение за $O(n \log n)$: динамическое программирование

с двоичным поиском

Чтобы получить более быстрое решение задачи, построим другой вариант динамического программирования за $O(n^2)$, а затем поймём, как можно этот вариант ускорить до $O(n \log n)$.

Динамика теперь будет такой: пусть $d[i] (i = 0 \dots n)$ — это число, на которое оканчивается возрастающая подпоследовательность длины i (а если таких чисел несколько — то наименьшее из них).

Изначально мы полагаем $d[0] = -\infty$, а все остальные элементы $d[i] = \infty$.

Считать эту динамику мы будем постепенно, обработав число $a[0]$, затем $a[1]$, и т.д.

Приведём реализацию этой динамики за $O(n^2)$:

```
int d[MAXN];
d[0] = -INF;
for (int i=1; i<=n; ++i)
        d[i] = INF;

for (int i=0; i<n; i++)
        for (int j=1; j<=n; j++)
                if (d[j-1] < a[i] && a[i] < d[j])
                        d[j] = a[i];
```

Заметим теперь, что у этой динамики есть одно **очень важное свойство**: $d[i-1] \le d[i]$ для всех $i = 1 \dots n$. Другое свойство — что каждый элемент $a[i]$ обновляет максимум одну ячейку $d[j]$.

Таким образом, это означает, что обрабатывать очередное $a[i]$ мы можем за $O(\log n)$, сделав двоичный поиск по массиву $d[]$. В самом деле, мы просто ищем в массиве $d[]$ первое число, которое строго больше $a[i]$, и пытаемся произвести обновление этого элемента аналогично приведённой выше реализации.

Реализация за $O(n \log n)$

Воспользовавшись стандартным в языке C++ алгоритмом двоичного поиска $upper_bound$ (который возвращает позицию первого элемента, строго большего данного), получаем такую простую реализацию:

```
int d[MAXN];
d[0] = -INF;
for (int i=1; i<=n; ++i)
        d[i] = INF;

for (int i=0; i<n; i++) {
        int j = int (upper_bound (d.begin(), d.end(), a[i]) - d.begin());
        if (d[j-1] < a[i] && a[i] < d[j])
                d[j] = a[i];
}
```

Восстановление ответа

По такой динамике тоже можно восстановить ответ, для чего опять же помимо динамики $d[i]$ также надо хранить массив "предков" $p[i]$ — то, на элементе с каким индексом оканчивается оптимальная подпоследовательность длины i. Кроме того, для каждого элемента массива $a[i]$ надо будет хранить его "предка" — т.е. индекс того элемента, который должен стоять перед $a[i]$ в оптимальной подпоследовательности.

Поддерживая эти два массива по ходу вычисления динамики, в конце будет нетрудно восстановить искомую подпоследовательность.

(Интересно отметить, что применительно к данной динамике ответ можно восстанавливать только так, через массивы предков — а без них восстановить ответ после вычисления динамики будет невозможно. Это один из редких случаев, когда к динамике неприменим альтернативный способ восстановления — без массивов предков).

Решение за $O(n \log n)$: структуры данных

Если приведённый выше способ за $O(n \log n)$ весьма красив, однако не совсем тривиален идейно, то есть и другой путь: воспользоваться одной из известных простых структур данных.

В самом деле, давайте вернёмся к самой первой динамике, где состоянием являлась просто текущая позиция.

Текущее значение динамики $d[i]$ вычисляется как максимум значений $d[i]$ среди всех таких элементов j, что $a[j] < a[i]$.

Следовательно, если мы через $t[]$ обозначим такой **массив**, в который будем записывать значения динамики от чисел:

$$t[a[i]] = d[i],$$

то получается, что всё, что нам надо уметь — это искать **максимум на префиксе** массива t: $t[0 \ldots a[i] - 1]$.

Задача поиска максимума на префиксах массива (с учётом того, что массив может меняться) решается многими стандартными структурами данных, например, деревом отрезков или деревом Фенвика.

Воспользовавшись любой такой структурой данных, мы получим решение за $O(n \log n)$.

У этого способа решения есть явные **недостатки**: по длине и сложности реализации этот путь будет в любом случае хуже, чем описанная выше динамика за $O(n \log n)$. Кроме того, если входные числа $a[i]$ могут быть достаточно большими, то скорее всего их придётся сжимать (т.е. перенумеровывать от 0 до $n - 1$) — без этого многие стандартные структуры данных работать не смогут из-за высокого потребления памяти.

С другой стороны, у данного пути есть и **преимущества**. Во-первых, при таком способе решения не придётся задумываться о хитрой динамике. Во-вторых, этот способ позволяет решать некоторые обобщения нашей задачи (о них см. ниже).

Смежные задачи

Приведём здесь несколько задач, тесно связанных с задачей поиска наидлиннейшей возрастающей подпоследовательности.

Наидлиннейшая неубывающая подпоследовательность

Фактически, это та же самая задача, только теперь в искомой подпоследовательности допускаются одинаковые числа (т. е. мы должны найти нестрого возрастающую подпоследовательность).

Решение этой задачи по сути ничем не отличается от нашей исходной задачи, просто при сравнениях изменятся знаки неравенств, а также надо будет немного изменить двоичный поиск.

Количество наидлиннейших возрастающих подпоследовательностей

Для решения этой задачи можно использовать самую первую динамику за $O(n^2)$ либо подход с помощью структур данных для решения за $O(n \log n)$. И в том, и в том случае все изменения заключаются только в том, что помимо значения динамики $d[i]$ надо также хранить, сколькими способами это значение могло быть получено.

По всей видимости, способ решения через динамику за $O(n \log n)$ к данной задаче применить невозможно.

Наименьшее число невозрастающих подпоследовательностей, покрывающих данную

Условие таково. Дан массив из n чисел $a[0 \ldots n - 1]$. Требуется раскрасить его числа в наименьшее число цветов так, чтобы по каждому цвету получалась бы невозрастающая подпоследовательность.

Решение. Утверждается, что минимальное количество необходимых цветов равно длине наидлиннейшей возрастающей подпоследовательности.

Доказательство. Фактически, нам надо доказать **двойственность** этой задачи и задачи поиска наидлиннейшей возрастающей подпоследовательности. Обозначим через x длину наидлиннейшей возрастающей подпоследовательности, а через y — искомое наименьшее число невозрастающих подпоследовательностей. Нам надо доказать, что $x = y$.

С одной стороны, понятно, почему не может быть $y < x$: ведь если у нас есть x строго возрастающих элементов, то никакие два из них не могли попасть в одну невозрастающую подпоследовательность, а, значит, $y \geq x$.

Покажем теперь, что, наоборот, y не может быть $> x$. Докажем это от противного: предположим, что $y > x$. Тогда рассмотрим любой оптимальный набор из y невозрастающих подпоследовательностей. Преобразуем этот набор таким образом: пока есть две таких подпоследовательности, что первая начинается раньше второй, но при этом первая начинается с числа, больше либо равного чем начало второй — отцепим это стартовое число от первой подпоследовательности и прицепим в начало второй. Таким образом, через какое-то конечное число шагов у нас останется y подпоследовательностей, причём их стартовые числа будут образовывать возрастающую подпоследовательность длины y. Но $y > x$, т.е. мы пришли к противоречию (ведь не может быть возрастающих подпоследовательностей длиннее x).

Таким образом, в самом деле, $y = x$, что и требовалось доказать.

Восстановление ответа. Утверждается, что само искомое разбиение на подпоследовательности можно искать жадно, т.е. идя слева направо и относя текущее число в ту подпоследовательность, которая сейчас заканчивается на минимальное число, больше либо равное текущему.

Динамика по профилю. Задача "паркет"

Типичными задачами на динамику по профилю, являются:

- найти количество способов замощения поля некоторыми фигурами
- найти замощение с наименьшим количеством фигур
- найти замощение с минимальным количеством неиспользованных клеток
- найти замощение с минимальным количеством фигур такое, что в него нельзя добавить ещё одну фигуру

Задача "Паркет"

Имеется прямоугольная площадь размером NxM. Нужно найти количество способов замостить эту площадь фигурами 1x2 (пустых клеток не должно оставаться, фигуры не должны накладываться друг на друга).

Построим такую динамику: D[I][Mask], где I=1..N, Mask=0..2^M-1. I обозначает количество строк в текущем поле, а Mask - профиль последней строки в текущем поле. Если j-й бит в Mask равен нулю, то в этом месте профиль проходит на "нормальном уровне", а если 1 - то здесь "выемка" глубиной 1. Ответом, очевидно, будет D[N][0].

Строить такую динамику будем, просто перебирая все I=1..N, все маски Mask=0..2^M-1, и для каждой маски будем делать переходы вперёд, т.е. добавлять к ней новую фигуру всеми возможными способами.

Реализация:

```
int n, m;
vector < vector<long long> > d;


void calc (int x = 0, int y = 0, int mask = 0, int next_mask = 0)
{
        if (x == n)
                return;
        if (y >= m)
                d[x+1][next_mask] += d[x][mask];
        else
        {
                int my_mask = 1 << y;
                if (mask & my_mask)
                        calc (x, y+1, mask, next_mask);
                else
                {
                        calc (x, y+1, mask, next_mask | my_mask);
                        if (y+1 < m && ! (mask & my_mask) && ! (mask &
(my_mask << 1)))
                                calc (x, y+2, mask, next_mask);
                }
        }
}


int main()
{
        cin >> n >> m;

        d.resize (n+1, vector<long long> (1<<m));
        d[0][0] = 1;
        for (int x=0; x<n; ++x)
                for (int mask=0; mask<(1<<m); ++mask)
                        calc (x, 0, mask, 0);

        cout << d[n][0];

}
```

Нахождение наибольшей нулевой подматрицы

Дана матрица a размером $n \times m$. Требуется найти в ней такую подматрицу, состоящую только из нулей, и среди всех таких — имеющую наибольшую площадь (подматрица — это прямоугольная область матрицы).

Тривиальный алгоритм, — перебирающий искомую подматрицу, — даже при самой хорошей реализации будет работать $O(n^2 m^2)$. Ниже описывается алгоритм, работающий за $O(nm)$, т.е. за линейное относительно размеров матрицы время.

Алгоритм

Для устранения неоднозначностей сразу заметим, что n равно числу строк матрицы a, соответственно, m — это число столбцов. Элементы матрицы будем нумеровать в 0-индексации, т.е. в обозначении $a[i][j]$ индексы i и j пробегают диапазоны $i = 0 \dots n-1$, $j = 0 \dots m-1$.

Шаг 1: Вспомогательная динамика

Сначала посчитаем следующую вспомогательную динамику: $d[i][j]$ — ближайшая сверху единица для элемента $a[i][j]$. Формально говоря, $d[i][j]$ равно наибольшему номеру строки (среди строк диапазоне от -1 до i), в которой в j-ом столбце стоит единица. В частности, если такой строки нет, то $d[i][j]$ полагается равным -1 (это можно понимать как то, что вся матрица как будто ограничена снаружи единицами).

Эту динамику легко считать двигаясь по матрице сверху вниз: пусть мы стоит в i-ой строке, и известно значение динамики для предыдущей строки. Тогда достаточно скопировать эти значения в динамику для текущей строки, изменив только те элементы, в которых в матрице стоят единицы. Понятно, что тогда даже не требуется хранить всю прямоугольную матрицу динамики, а достаточно только одного массива размера m:

```
vector<int> d (m, -1);
for (int i=0; i<n; ++i) {
        for (int j=0; j<m; ++j)
                if (a[i][j] == 1)
                        d[j] = i;

        // вычислили d для i-ой строки, можем здесь использовать эти значения
}
```

Шаг 2: Решение задачи

Уже сейчас мы можем решить задачу за $O(nm^2)$ — просто перебирать в текущей строке номер левого и правого столбцов искомой подматрицы, и с помощью динамики $d[][]$ вычислять за $O(1)$ верхнюю границу нулевой подматрицы. Однако можно пойти дальше и значительно улучшить асимптотику решения.

Ясно, что искомая нулевая подматрица ограничена со всех четырёх сторон какими-то единичками (либо границами поля), — которые и мешают ей увеличиться в размерах и улучшить ответ. Поэтому, утверждается, мы не пропустим ответ, если будем действовать следующим образом: сначала переберём номер i нижней строки нулевой подматрицы, затем переберём, в каком столбце j мы будем упирать вверх нулевую подматрицу. Пользуясь значением $d[i][j]$, мы сразу получаем номер верхней строки нулевой подматрицы. Осталось теперь определить оптимальные левую и правую границы нулевой подматрицы, — т.е. максимально раздвинуть эту подматрицу влево и вправо от j-го столбца.

Что значит раздвинуть максимально влево? Это значит найти такой индекс k_1, для которого будет $d[i][k_1] > d[i][j]$, и при этом k_1 — ближайший такой слева для индекса j. Понятно, что тогда $k_1 + 1$ даёт номер левого столбца искомой нулевой подматрицы. Если такого индекса вообще нет, то положить $k_1 = -1$ (это означает, что мы смогли расширить текущую нулевую подматрицу влево до упора — до границы всей матрицы a).

Симметрично можно определить индекс k_2 для правой границы: это ближайший справа от j индекс такой, что $d[i][k_2] > d[i][j]$ (либо m, если такого индекса нет).

Итак, индексы k_1 и k_2, если мы научимся эффективно их искать, дадут нам всю необходимую информацию о текущей нулевой подматрице. В частности, её площадь будет равна $(i - d[i][j]) \cdot (k_2 - k_1 - 1)$.

Как же искать эти индексы k_1 и k_2 эффективно при фиксированных i и j? Нас удовлетворит только асимптотика $O(1)$, хотя бы в среднем.

Добиться такой асимптотики можно с помощью стека (stack) следующим образом. Научимся сначала искать индекс k_1, и сохранять его значение для каждого индекса j внутри текущей строки i в динамике $d_1[i][j]$. Для этого будем просматривать все столбцы j слева направо, и заведём такой стек, в котором всегда будут лежать только те столбцы, в которых значение динамики $d[][]$ строго больше $d[i][j]$. Понятно, что при переходе от столбца j к следующему столбцу $j+1$ требуется обновить содержимое этого стека. Утверждается, что требуется сначала положить в стек столбец j (поскольку для него стек "хороший"), а затем, пока на вершине стека лежит неподходящий элемент (т.е. у которого значение $d \le d[i][j+1]$), — доставать этот элемент. Легко понять, что удалять из стека достаточно только из его вершины, и ни из каких других его мест (потому что стек будет содержать возрастающую по d последовательность столбцов).

Значение $d_1[i][j]$ для каждого j будет равно значению, лежащему в этот момент на вершине стека.

Ясно, что поскольку добавлений в стек на каждой строчке i происходит ровно m штук, то и удалений также не могло быть больше, поэтому в сумме асимптотика будет линейной.

Динамика $d_2[i][j]$ для нахождения индексов k_2 считается аналогично, только надо просматривать столбцы справа налево.

Также следует отметить, что этот алгоритм потребляет $O(m)$ памяти (не считая входные данные — матрицу $a[][]$).

Реализация

Эта реализация вышеописанного алгоритма считывает размеры матрицы, затем саму матрицу (как последовательность чисел, разделённых пробелами или переводами строк), и затем выводит ответ — размер наибольшей нулевой подматрицы.

Легко улучшить эту реализацию, чтобы она также выводила саму нулевую подматрицу: для этого надо при каждом изменении ans запоминать также номера строк и столбцов подматрицы (ими будут соответственно $d[j]+1$, i, $d1[j]+1$, $d2[j]-1$).

```
int n, m;
cin >> n >> m;
vector < vector<int> > a (n, vector<int> (m));
for (int i=0; i<n; ++i)
        for (int j=0; j<m; ++j)
                cin >> a[i][j];

int ans = 0;
vector<int> d (m, -1), d1 (m), d2 (m);
stack<int> st;
for (int i=0; i<n; ++i) {
        for (int j=0; j<m; ++j)
                if (a[i][j] == 1)
                        d[j] = i;
        while (!st.empty()) st.pop();
        for (int j=0; j<m; ++j) {
                while (!st.empty() && d[st.top()] <= d[j])  st.pop();
                d1[j] = st.empty() ? -1 : st.top();
                st.push (j);
        }
        while (!st.empty()) st.pop();
        for (int j=m-1; j>=0; --j) {
                while (!st.empty() && d[st.top()] <= d[j])  st.pop();
                d2[j] = st.empty() ? m : st.top();
                st.push (j);
        }
        for (int j=0; j<m; ++j)
                ans = max (ans, (i - d[j]) * (d2[j] - d1[j] - 1));
}

cout << ans;
```

Метод Гаусса решения системы линейных уравнений

Дана система n линейных алгебраических уравнений (СЛАУ) с m неизвестными. Требуется решить эту систему: определить, сколько решений она имеет (ни одного, одно или бесконечно много), а если она имеет хотя бы одно решение, то найти любое из них.

Формально задача ставится следующим образом: решить систему:

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + \ldots + a_{1m}x_m = b_1, \\ a_{21}x_1 + a_{22}x_2 + \ldots + a_{2m}x_m = b_2, \\ \ldots \\ a_{n1}x_1 + a_{n2}x_2 + \ldots + a_{nm}x_m = b_n, \end{cases}$$

где коэффициенты $a_{ij}(i = 1 \ldots n, j = 1 \ldots m)$ и $b_i(i = 1 \ldots n)$ известны, а переменные $x_i(i = 1 \ldots m)$ — искомые неизвестные.

Удобно матричное представление этой задачи:

$$Ax = b,$$

где A — матрица $n \times m$, составленная из коэффициентов a_{ij}, x и b — векторы-столбцы высоты m.

Стоит отметить, что СЛАУ может быть не над полем действительных чисел, а над полем **по модулю** какого-либо числа p, т.е.:

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + \ldots + a_{1m}x_m = b_1, \pmod p \\ a_{21}x_1 + a_{22}x_2 + \ldots + a_{2m}x_m = b_2, \pmod p \\ \ldots \\ a_{n1}x_1 + a_{n2}x_2 + \ldots + a_{nm}x_m = b_n \pmod p \end{cases}$$

— алгоритм Гаусса работает и для таких систем тоже (но этот случай будет рассмотрен ниже в отдельном разделе).

Алгоритм Гаусса

Строго говоря, описываемый ниже метод правильно называть методом "Гаусса-Жордана" (Gauss-Jordan elimination), поскольку он является вариацией метода Гаусса, описанной геодезистом Вильгельмом Жорданом в 1887 г. (стоит отметить, что Вильгельм Жордан не является автором ни теоремы Жордана о кривых, ни жордановой алгебры — всё это три разных учёных-однофамильца; кроме того, по всей видимости, более правильной является транскрипция "Йордан", но написание "Жордан" уже закрепилось в русской литературе). Также интересно заметить, что одновременно с Жорданом (а по некоторым данным даже раньше него) этот алгоритм придумал Класен (B.-I. Clasen).

Базовая схема

Кратко говоря, алгоритм заключается в **последовательном исключении** переменных из каждого уравнения до тех пор, пока в каждом уравнении не останется только по одной переменной. Если $n = m$, то можно говорить, что алгоритм Гаусса-Жордана стремится привести матрицу A системы к единичной матрице — ведь после того как матрица стала единичной, решение системы очевидно — решение единственно и задаётся получившимися коэффициентами b_i.

При этом алгоритм основывается на двух простых эквивалентных преобразованиях системы: во-первых, можно обменивать два уравнения, а во-вторых, любое уравнение можно заменить линейной комбинацией этой строки (с ненулевым коэффициентом) и других строк (с произвольными коэффициентами).

На первом шаге алгоритм Гаусса-Жордана делит первую строку на коэффициент a_{11}. Затем алгоритм прибавляет первую строку к остальным строкам с такими коэффициентами, чтобы их коэффициенты в первом столбце обращались в нули — для этого, очевидно, при прибавлении первой строки к i-ой надо домножать её на $-a_{i1}$. При каждой операции с матрицей A (деление на число, прибавление к одной строке другой) соответствующие операции производятся и с вектором b; в некотором смысле, он ведёт себя, как если бы он был $m + 1$-ым столбцом матрицы A.

В итоге, по окончании первого шага первый столбец матрицы A станет единичным (т.е. будет содержать единицу в первой строке и нули в остальных).

Аналогично производится второй шаг алгоритма, только теперь рассматривается второй столбец и вторая строка: сначала вторая строка делится на a_{22}, а затем отнимается от всех остальных строк с такими коэффициентами,

чтобы обнулять второй столбец матрицы A.

И так далее, пока мы не обработаем все строки или все столбцы матрицы A. Если $n = m$, то по построению алгоритма очевидно, что матрица A получится единичной, что нам и требовалось.

Поиск опорного элемента (pivoting)

Разумеется, описанная выше схема неполна. Она работает только в том случае, если на каждом i-ом шаге элемент a_{ii} отличен от нуля — иначе мы просто не сможем добиться обнуления остальных коэффициентов в текущем столбце путём прибавления к ним i-ой строки.

Чтобы сделать алгоритм работающим в таких случаях, как раз и существует процесс **выбора опорного элемента** (на английском языке это называется одним словом "pivoting"). Он заключается в том, что производится перестановка строк и/или столбцов матрицы, чтобы в нужном элементе a_{ii} оказалось ненулевое число.

Заметим, что перестановка строк значительно проще реализуется на компьютере, чем перестановка столбцов: ведь при обмене местами двух каких-то столбцов надо запомнить, что эти две переменных обменялись местами, чтобы затем, при восстановлении ответа, правильно восстановить, какой ответ к какой переменной относится. При перестановке строк никаких таких дополнительных действий производить не надо.

К счастью, для корректности метода достаточно одних только обменов строк (т.н. "partial pivoting", в отличие от "full pivoting", когда обмениваются и строки, и столбцы). Но какую же именно строку следует выбирать для обмена? И правда ли, что поиск опорного элемента надо делать только тогда, когда текущий элемент a_{ii} нулевой?

Общего ответа на этот вопрос не существует. Есть разнообразные эвристики, однако самой эффективной из них (по соотношению простоты и отдачи) является такая **эвристика**: в качестве опорного элемента следует брать наибольший по модулю элемент, причём производить поиск опорного элемента и обмен с ним надо **всегда**, а не только когда это необходимо (т.е. не только тогда, когда $a_{ii} = 0$).

Иными словами, перед выполнением i-ой фазы алгоритма Гаусса-Жордана с эвристикой partial pivoting необходимо найти в i-ом столбце среди элементов с индексами от i до n максимальный по модулю, и обменять строку с этим элементом с i-ой строкой.

Во-первых, эта эвристика позволит решить СЛАУ, даже если по ходу решения будет случаться так, что элемент $a_{ii} = 0$. Во-вторых, что весьма немаловажно, эта эвристика улучшает **численную устойчивость** алгоритма Гаусса-Жордана.

Без этой эвристики, даже если система такова, что на каждой i-ой фазе $a_{ii} \neq 0$ — алгоритм Гаусса-Жордана отработает, но в итоге накапливающаяся погрешность может оказаться настолько огромной, что даже для матриц размера около 20 погрешность будет превосходить сам ответ.

Вырожденные случаи

Итак, если останавливаться на алгоритме Гаусса-Жордана с partial pivoting, то, утверждается, если $m = n$ и система неврождена (т.е. имеет ненулевой определитель, что означает, что она имеет единственное решение), то описанный выше алгоритм полностью отработает и придёт к единичной матрице A (доказательство этого, т.е. того, что ненулевой опорный элемент всегда будет находиться, здесь не приводится).

Рассмотрим теперь **общий случай** — когда n и m не обязательно равны. Предположим, что опорный элемент на i-ом шаге не нашёлся. Это означает, что в i-ом столбце все строки, начиная с текущей, содержат нули. Утверждается, что в этом случае эта i-ая переменная не может быть определена, и является **независимой переменной** (может принимать произвольное значение). Чтобы алгоритм Гаусса-Жордана продолжил свою работу для всех последующих переменных, в такой ситуации надо просто пропустить текущий i-ый столбец, не увеличивая при этом номер текущей строки (можно сказать, что мы виртуально удаляем i-ый столбец матрицы).

Итак, некоторые переменные в процессе работы алгоритма могут оказываться независимыми. Понятно, что когда количество m переменных больше количества n уравнений, то как минимум $m - n$ переменных обнаружатся независимыми.

В целом, если обнаружилась хотя бы одна независимая переменная, то она может принимать произвольное значение, в то время как остальные (зависимые) переменные будут выражаться через неё. Это означает, что, когда мы работаем в поле действительных чисел, система потенциально имеет **бесконечно много решений** (если мы рассматриваем СЛАУ по модулю, то число решений будет равно этому модулю в степени количества независимых переменных). Впрочем, следует быть аккуратным: надо помнить о том, что даже если были обнаружены независимые переменные, тем не менее СЛАУ **может не иметь решений вовсе**. Это происходит, когда в оставшихся необработанными уравнениях (тех, до которых алгоритм Гаусса-Жордана не дошёл, т.е. это уравнения, в которых остались только независимые переменные) есть хотя бы один ненулевой свободный член.

Впрочем, проще это проверить явной подстановкой найденного решения: всем независимыми переменным присвоить нулевые значения, зависимым переменным присвоить найденные значения, и подставить это решение в текущую СЛАУ.

Реализация

Приведём здесь реализацию алгоритма Гаусса-Жордана с эвристикой partial pivoting (выбором опорного элемента как максимума по столбцу).

На вход функции $\mathrm{gauss}()$ передаётся сама матрица системы a. Последний столбец матрицы a — это в наших старых обозначениях столбец b свободных коэффициентов (так сделано для удобства программирования — т.к. в самом алгоритме все операции со свободными коэффициентами b повторяют операции с матрицей A).

Функция возвращает число решений системы (0, 1 или ∞) (бесконечность обозначена в коде специальной константой INF, которой можно задать любое большое значение). Если хотя бы одно решение существует, то оно возвращается в векторе ans.

```
int gauss (vector < vector<double> > a, vector<double> & ans) {
        int n = (int) a.size();
        int m = (int) a[0].size() - 1;

        vector<int> where (m, -1);
        for (int col=0, row=0; col<m && row<n; ++col) {
                int sel = row;
                for (int i=row; i<n; ++i)
                        if (abs (a[i][col]) > abs (a[sel][col]))
                                sel = i;
                if (abs (a[sel][col]) < EPS)
                        continue;
                for (int i=col; i<=m; ++i)
                        swap (a[sel][i], a[row][i]);
                where[col] = row;

                for (int i=0; i<n; ++i)
                        if (i != row) {
                                double c = a[i][col] / a[row][col];
                                for (int j=col; j<=m; ++j)
                                        a[i][j] -= a[row][j] * c;
                        }
                ++row;
        }

        ans.assign (m, 0);
        for (int i=0; i<m; ++i)
                if (where[i] != -1)
                        ans[i] = a[where[i]][m] / a[where[i]][i];
        for (int i=0; i<n; ++i) {
                double sum = 0;
                for (int j=0; j<m; ++j)
                        sum += ans[j] * a[i][j];
                if (abs (sum - a[i][m]) > EPS)
                        return 0;
        }

        for (int i=0; i<m; ++i)
                if (where[i] == -1)
                        return INF;
        return 1;
}
```

В функции поддерживаются два указателя — на текущий столбец col и текущую строку row.

Также заводится вектор where, в котором для каждой переменной записано, в какой строке должна она получиться (иными словами, для каждого столбца записан номер строки, в которой этот столбец отличен от нуля). Этот вектор нужен, поскольку некоторые переменные могли не "определиться" в ходе решения (т.е. это независимые переменные, которым можно присвоить произвольное значение — например, в приведённой реализации это нули).

Реализация использует технику partial pivoting, производя поиск строки с максимальным по модулю элементом, и переставляя затем эту строку в позицию row (хотя явную перестановку строк можно заменить обменом двух индексов в некотором массиве, на практике это не даст реального выигрыша, т.к. на обмены тратится $O(n^2)$ операций).

В реализации в целях простоты текущая строка не делится на опорный элемент — так что в итоге по окончании работы алгоритма матрица становится не единичной, а диагональной (впрочем, по-видимому, деление строки на ведущий элемент позволяет несколько уменьшить возникающие погрешности).

После нахождения решения оно подставляется обратно в матрицу — чтобы проверить, имеет ли система хотя бы одно решение или нет. Если проверка найденного решения прошла успешно, то функция возвращает 1 или ∞ —

в зависимости от того, есть ли хотя бы одна независимая переменная или нет.

Асимптотика

Оценим асимптотику полученного алгоритма. Алгоритм состоит из m фаз, на каждой из которых происходит:

- поиск и перестановка опорного элемента — за время $O(n+m)$ при использовании эвристики "partial pivoting" (поиск максимума в столбце)
- если опорный элемент в текущем столбце был найден — то прибавление текущего уравнения ко всем остальным уравнениям — за время $O(nm)$

Очевидно, первый пункт имеет меньшую асимптотику, чем второй. Заметим также, что второй пункт выполняется не более $\min(n,m)$ раз — столько, сколько может быть зависимых переменных в СЛАУ.

Таким образом, **итоговая асимптотика** алгоритма принимает вид $O(\min(n,m) \cdot nm)$.

При $n = m$ эта оценка превращается в $O(n^3)$.

Заметим, что когда СЛАУ рассматривается не в поле действительных чисел, а в поле по модулю два, то систему можно решать гораздо быстрее — об этом см. ниже в разделе "Решение СЛАУ по модулю".

Более точная оценка числа действий

Для простоты выкладок будем считать, что $n = m$.

Как мы уже знаем, время работы всего алгоритма фактически определяется временем, затрачиваемым на исключение текущего уравнения из остальных.

Это может происходить на каждом из n шагов, при этом текущее уравнение прибавляется ко всем $n-1$ остальным. При прибавлении работа идёт только со столбцами, начиная с текущего. Таким образом, в сумме получается $n^3/2$ операций.

Дополнения

Ускорение алгоритма: разделение его на прямой и обратный ход

Добиться двукратного ускорения алгоритма можно, рассмотрев другую его версию, более классическую, когда алгоритм разбивается на фазы прямого и обратного хода.

В целом, в отличие от описанного выше алгоритма, можно приводить матрицу не к диагональному виду, а к **треугольному виду** — когда все элементы строго ниже главной диагонали равны нулю.

Система с треугольной матрицей решается тривиально — сначала из последнего уравнения сразу находится значение последней переменной, затем найденное значение подставляется в предпоследнее уравнение и находится значение предпоследней переменной, и так далее. Этот процесс и называется **обратным ходом** алгоритма Гаусса.

Прямой ход алгоритма Гаусса — это алгоритм, аналогичный описанному выше алгоритму Гаусса-Жордана, за одним исключением: текущая переменная исключается не из всех уравнений, а только из уравнений после текущего. В результате этого действительно получается не диагональная, а треугольная матрица.

Разница в том, что прямой ход работает **быстрее** алгоритма Гаусса-Жордана — поскольку в среднем он делает в два раза меньше прибавлений одного уравнения к другому. Обратный ход работает за $O(nm)$, что в любом случае асимптотически быстрее прямого хода.

Таким образом, если $n = m$, то данный алгоритм будет делать уже $n^3/4$ операций — что в два раза меньше алгоритма Гаусса-Жордана.

Решение СЛАУ по модулю

Для решения СЛАУ по модулю можно применять описанный выше алгоритм, он сохраняет свою корректность.

Разумеется, теперь становится ненужным использовать какие-то хитрые техники выбора опорного элемента — достаточно найти любой ненулевой элемент в текущем столбце.

Если модуль простой, то никаких сложностей вообще не возникает — происходящие по ходу работы алгоритма Гаусса деления не создают особых проблем.

Особенно замечателен **модуль**, **равный двум**: для него все операции с матрицей можно производить очень эффективно. Например, отнимание одной строки от другой по модулю два — это на самом деле их

симметрическая разность ("xor"). Таким образом, весь алгоритм можно значительно ускорить, сжав всю матрицу в битовые маски и оперируя только ими. Приведём здесь новую реализацию основной части алгоритма Гаусса-Жордана, используя стандартный контейнер C++ "bitset":

```
int gauss (vector < bitset<N> > a, int n, int m, bitset<N> & ans) {
        vector<int> where (m, -1);
        for (int col=0, row=0; col<m && row<n; ++col) {
                for (int i=row; i<n; ++i)
                        if (a[i][col]) {
                                swap (a[i], a[row]);
                                break;
                        }
                if (! a[row][col])
                        continue;
                where[col] = row;

                for (int i=0; i<n; ++i)
                        if (i != row && a[i][col])
                                a[i] ^= a[row];
                ++row;
        }
```

Как можно заметить, реализация стала даже немного короче, при том, что она значительно быстрее старой реализации — а именно, быстрее в 32 раза за счёт битового сжатия. Также следует отметить, что решение систем по модулю два на практике работает очень быстро, поскольку случаи, когда от одной строки надо отнимать другую, происходят достаточно редко (на разреженных матрицах этот алгоритм может работать за время скорее порядка квадрата от размера, чем куба).

Если модуль **произвольный** (не обязательно простой), то всё становится несколько сложнее. Понятно, что пользуясь Китайской теоремой об остатках, мы сводим задачу с произвольным модулем только к модулям вида "степень простого". [дальнейший текст был скрыт, т.к. это непроверенная информация — возможно, неправильный способ решения]

Наконец, рассмотрим вопрос **числа решений СЛАУ по модулю**. Ответ на него достаточно прост: число решений равно p^k, где p — модуль, k — число независимых переменных.

Немного о различных способах выбора опорного элемента

Как уже говорилось выше, однозначного ответа на этот вопрос нет.

Эвристика "partial pivoting", которая заключалась в поиске максимального элемента в текущем столбце, работает на практике весьма неплохо. Также оказывается, что она даёт практически тот же результат, что и "full pivoting" — когда опорный элемент ищется среди элементов целой подматрицы — начиная с текущей строки и с текущего столбца.

Но интересно отметить, что обе эти эвристики с поиском максимального элемента, фактически, очень зависят от того, насколько были промасштабированы исходные уравнения. Например, если одно из уравнений системы умножить на миллион, то это уравнение почти наверняка будет выбрано в качестве ведущего на первом же шаге. Это кажется достаточно странным, поэтому логичен переход к немного более сложной эвристике — так называемому "implicit pivoting".

Эвристика implicit pivoting заключается в том, что элементы различных строк сравниваются так, как если бы обе строки были пронормированы таким образом, что максимальный по модулю элемент в них был бы равен единице. Для реализации этой техники надо просто поддерживать текущий максимум в каждой строке (либо поддерживать каждую строку так, чтобы максимум в ней был равен единице по модулю, но это может привести к увеличению накапливаемой погрешности).

Улучшение найденного ответа

Поскольку, несмотря на различные эвристики, алгоритм Гаусса-Жордана всё равно может приводить к большим погрешностям на специальных матрицах даже размеров порядка $50 \cdot 100$.

В связи с этим, полученный алгоритмом Гаусса-Жордана ответ можно улучшить, применив к нему какой-либо простой численный метод — например, метод простой итерации.

Таким образом, решение превращается в двухшаговое: сначала выполняется алгоритм Гаусса-Жордана, затем — какой-либо численный метод, принимающий в качестве начальных данных решение, полученное на первом шаге.

Такой приём позволяет несколько расширить множество задач, решаемых алгоритмом Гаусса-Жордана с приемлемой погрешностью.

Литература

- William H. Press, Saul A. Teukolsky, William T. Vetterling, Brian P. Flannery. Numerical Recipes: The Art of Scientific Computing [2007]
- Anthony Ralston, Philip Rabinowitz. A first course in numerical analysis [2001]

Нахождение ранга матрицы

Ранг матрицы - это наибольшее число линейно независимых строк/столбцов матрицы. Ранг определён не только для квадратных матриц; пусть матрица прямоугольна и имеет размер NxM.

Также ранг матрицы можно определить как наибольший из порядков миноров матрицы, отличных от нуля.

Заметим, что если матрица квадратная и её определитель отличен от нуля, то ранг равен N(=M), иначе он будет меньше. В общем случае, ранг матрицы не превосходит min(N,M).

Алгоритм

Искать ранг можно с помощью модифицированного метода Гаусса. Будем выполнять абсолютно те же самые операции, что и при решении системы или нахождении её определителя, но если на каком-либо шаге в i-ом столбце среди невыбранных до этого строк нет ненулевых, то мы этот шаг пропускаем, а ранг уменьшаем на единицу (изначально ранг полагаем равным max(N,M)). Иначе, если мы нашли на i-ом шаге строку с ненулевым элементом в i-ом столбце, то помечаем эту строку как выбранную, и выполняем обычные операции отнимания этой строки от остальных.

Реализация

```
const double EPS = 1E-9;

int rank = max(n,m);
vector<char> line_used (n);
for (int i=0; i<m; ++i) {
        int j;
        for (j=0; j<n; ++j)
                if (!line_used[j] && abs(a[j][i]) > EPS)
                        break;
        if (j == n)
                --rank;
        else {
                line_used[j] = true;
                for (int p=i+1; p<m; ++p)
                        a[j][p] /= a[j][i];
                for (int k=0; k<n; ++k)
                        if (k != j && abs (a[k][i]) > EPS)
                                for (int p=i+1; p<m; ++p)
                                        a[k][p] -= a[j][p] * a[k][i];
        }
}
```

Вычисление определителя матрицы методом Гаусса

Пусть дана квадратная матрица A размером NxN. Требуется вычислить её определитель.

Алгоритм

Воспользуемся идеями метода Гаусса решения систем линейных уравнений.

Будем выполнять те же самые действия, что и при решении системы линейных уравнений, исключив только деление текущей строки на a[i][i] (точнее, само деление можно выполнять, но подразумевая, что число выносится за знак определителя). Тогда все операции, которые мы будем производить с матрицей, не будут изменять величину определителя матрицы, за исключением, быть может, знака (мы только обмениваем местами две строки, что меняет знак на противоположный, или прибавляем одну строку к другой, что не меняет величину определителя).

Но матрица, к которой мы приходим после выполнения алгоритма Гаусса, является диагональной, и определитель её равен произведению элементов, стоящих на диагонали. Знак, как уже говорилось, будет определяться количеством обменов строк (если их нечётное, то знак определителя следует изменить на противоположный). Таким образом, мы можем с помощью алгоритма Гаусса вычислять определитель матрицы за O (N^3).

Осталось только заметить, что если в какой-то момент мы не найдём в текущем столбце ненулевого элемента, то алгоритм следует остановить и вернуть 0.

Реализация

```
const double EPS = 1E-9;
int n;
vector < vector<double> > a (n, vector<double> (n));
... чтение n и a ...

double det = 1;
for (int i=0; i<n; ++i) {
        int k = i;
        for (int j=i+1; j<n; ++j)
                if (abs (a[j][i]) > abs (a[k][i]))
                        k = j;
        if (abs (a[k][i]) < EPS) {
                det = 0;
                break;
        }
        swap (a[i], a[k]);
        if (i != k)
                det = -det;
        det *= a[i][i];
        for (int j=i+1; j<n; ++j)
                a[i][j] /= a[i][i];
        for (int j=0; j<n; ++j)
                if (j != i && abs (a[j][i]) > EPS)
                        for (int k=i+1; k<n; ++k)
                                a[j][k] -= a[i][k] * a[j][i];
}

cout << det;
```

Вычисление определителя методом Краута за O (N^3)

Здесь будет рассмотрена модификация метода Краута (Crout), позволяющая вычислить определитель матрицы за O (N^3).

Собственно алгоритм Краута находит разложение матрицы A в виде $A = L\ U$, где L - нижняя, а U - верхняя треугольная матрицы. Без ограничения общности можно считать, что в L все диагональные элементы равны 1. Но, зная эти матрицы, легко вычислить определитель A: он будет равен произведению всех элементов, стоящих на главной диагонали матрицы U.

Имеется теорема, согласно которой любая обратимая матрица обладает LU-разложением, и притом единственным, тогда и только тогда, когда все её главные миноры отличны от нуля. Следует напомнить, что мы рассматриваем только такие разложения, в которых диагональ L состоит только из единиц; иначе же, вообще говоря, разложение не единственно.

Пусть A - матрица, N - её размер. Мы найдём элементы матриц L и U.

Сам алгоритм состоит из следующих шагов:

1. Положим $L_{i\,i} = 1$ для i = 1, 2, ..., N
2. Для каждого j = 1, 2, ..., N выполним:
   1. Для i = 1, 2, ..., j найдём значение $U_{i\,j}$:

      $U_{i\,j} = A_{i\,j}$ - SUM $L_{i\,k}\ U_{k\,j}$,

      где сумма по всем k = 1, 2, ..., i-1.
   2. Далее, для i = j+1, j+2, ..., N имеем:
      $L_{i\,j} = (A_{i\,j}$ - SUM $L_{i\,k}\ U_{k\,j}) / U_{j\,j}$,

      где сумма берётся по всем k = 1, 2, ..., j-1.

Реализация

Код на Java (с использованием дробной длинной арифметики):

```
static BigInteger det (BigDecimal a [][], int n)
{

        try {

        for (int i=0; i<n; i++)
        {
                boolean nonzero = false;
                for (int j=0; j<n; j++)
                        if (a[i][j].compareTo (new BigDecimal
(BigInteger.ZERO)) > 0)
                                nonzero = true;
                if (!nonzero)
                        return BigInteger.ZERO;
        }

        BigDecimal scaling [] = new BigDecimal [n];
        for (int i=0; i<n; i++)
        {
                BigDecimal big = new BigDecimal (BigInteger.ZERO);
                for (int j=0; j<n; j++)
                        if (a[i][j].abs().compareTo (big) > 0)
                                big = a[i][j].abs();
                scaling[i] = (new BigDecimal (BigInteger.ONE)) .divide
                        (big, 100, BigDecimal.ROUND_HALF_EVEN);
        }

        int sign = 1;

        for (int j=0; j<n; j++)
```

```
        {

                for (int i=0; i<j; i++)
                {
                        BigDecimal sum = a[i][j];
                        for (int k=0; k<i; k++)
                                sum = sum.subtract (a[i][k].multiply (a[k][j]));
                        a[i][j] = sum;
                }

                BigDecimal big = new BigDecimal (BigInteger.ZERO);
                int imax = -1;
                for (int i=j; i<n; i++)
                {
                        BigDecimal sum = a[i][j];
                        for (int k=0; k<j; k++)
                                sum = sum.subtract (a[i][k].multiply (a[k][j]));
                        a[i][j] = sum;
                        BigDecimal cur = sum.abs();
                        cur = cur.multiply (scaling[i]);
                        if (cur.compareTo (big) >= 0)
                        {
                                big = cur;
                                imax = i;
                        }
                }

                if (j != imax)
                {

                        for (int k=0; k<n; k++)
                        {
                                BigDecimal t = a[j][k];
                                a[j][k] = a[imax][k];
                                a[imax][k] = t;
                        }

                        BigDecimal t = scaling[imax];
                        scaling[imax] = scaling[j];
                        scaling[j] = t;

                        sign = -sign;
                }

                if (j != n-1)
                        for (int i=j+1; i<n; i++)
                                a[i][j] = a[i][j].divide
                                        (a[j][j], 100,
BigDecimal.ROUND_HALF_EVEN);

        }

        BigDecimal result = new BigDecimal (1);
        if (sign == -1)
                result = result.negate();
        for (int i=0; i<n; i++)
                result = result.multiply (a[i][i]);

        return result.divide
                (BigDecimal.valueOf(1), 0, BigDecimal.
ROUND_HALF_EVEN).toBigInteger();

        }
        catch (Exception e)
        {
                return BigInteger.ZERO;
        }

}
```

Интегрирование по формуле Симпсона

Требуется посчитать значение определённого интеграла:

$$\int_a^b f(x)dx$$

Решение, описываемое здесь, было опубликовано в одной из диссертаций **Томаса Симпсона** (Thomas Simpson) в 1743 г.

Формула Симпсона

Пусть n — некоторое натуральное число. Разобьём отрезок интегрирования $[a; b]$ на $2n$ равных частей:

$$x_i = ih, \quad i = 0 \dots 2n,$$
$$h = \frac{b-a}{2n}.$$

Теперь посчитаем интеграл отдельно на каждом из отрезков $[x_{2i-2}, x_{2i}], i = 1 \dots n$, а затем сложим все значения.

Итак, пусть мы рассматриваем очередной отрезок $[x_{2i-2}, x_{2i}], i = 1 \dots n$. Заменим функцию $f(x)$ на нём параболой, проходящей через 3 точки $(x_{2i-2}, x_{2i-1}, x_{2i})$. Такая парабола всегда существует и единственна. Её можно найти аналитически, затем останется только проинтегрировать выражение для неё, и окончательно получаем:

$$\int_{x_{2i-2}}^{x_{2i}} f(x)dx = (f(x_{2i-2}) + 4f(x_{2i-1}) + f(x_{2i}))\frac{h}{3}.$$

Складывая эти значения по всем отрезкам, получаем окончательную **формулу Симпсона**:

$$\int_a^b f(x)dx = (f(x_0) + 4f(x_1) + 2f(x_2) + 4f(x_3) + 2f(x_4) + \dots + 4f(x_{2N-1}) + f(x_{2N}))\frac{h}{3}.$$

Погрешность

Погрешность, даваемая формулой Симпсона, не превосходит по модулю величины:

$$\frac{1}{180}h^4(b-a)\max_{a \le x \le b}\left|f^{(4)}(x)\right|.$$

Таким образом, погрешность имеет порядок уменьшения как $O(n^4)$.

Реализация

Здесь $f(x)$ — некоторая пользовательская функция.

```
double a, b; // входные данные
const int N = 1000*1000; // количество шагов (уже умноженное на 2)
double s = 0;
double h = (b - a) / N;
for (int i=0; i<=N; ++i) {
        double x = a + h * i;
        s += f(x) * ((i==0 || i==N) ? 1 : ((i&1)==0) ? 2 : 4);
}
s *= h / 3;
```

Метод Ньютона (касательных) для поиска корней

Это итерационный метод, изобретённый **Исааком Ньютоном** (Isaak Newton) около 1664 г. Впрочем, иногда этот метод называют методом Ньютона-Рафсона (Raphson), поскольку Рафсон изобрёл тот же самый алгоритм на несколько лет позже Ньютона, однако его статья была опубликована намного раньше.

Задача заключается в следующем. Дано уравнение:

$$f(x) = 0.$$

Требуется решить это уравнение, точнее, найти один из его корней (предполагается, что корень существует). Предполагается, что $f(x)$ непрерывна и дифференцируема на отрезке $[a; b]$.

Алгоритм

Входным параметром алгоритма, кроме функции $f(x)$, является также **начальное приближение** — некоторое x_0, от которого алгоритм начинает идти.

Пусть уже вычислено x_i, вычислим x_{i+1} следующим образом. Проведём касательную к графику функции $f(x)$ в точке $x = x_i$, и найдём точку пересечения этой касательной с осью абсцисс. x_{i+1} положим равным найденной точке, и повторим весь процесс с начала.

Нетрудно получить следующую формулу:

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}.$$

Интуитивно ясно, что если функция $f(x)$ достаточно "хорошая" (гладкая), а x_i находится достаточно близко от корня, то x_{i+1} будет находиться ещё ближе к искомому корню.

Скорость сходимости является **квадратичной**, что, условно говоря, означает, что число точных разрядов в приближенном значении x_i удваивается с каждой итерацией.

Применение для вычисления квадратного корня

Рассмотрим метод Ньютона на примере вычисления квадратного корня.

Если подставить $f(x) = \sqrt{x}$, то после упрощения выражения получаем:

$$x_{i+1} = \frac{x_i + \frac{n}{x_i}}{2}.$$

Первый типичный вариант задачи — когда дано дробное число n, и нужно подсчитать его корень с некоторой точностью EPS:

```
double n;
cin >> n;
const double EPS = 1E-15;
double x = 1;
for (;;) {
        double nx = (x + n / x) / 2;
        if (abs (x - nx) < EPS)  break;
        x = nx;
}
printf ("%.15lf", x);
```

Другой распространённый вариант задачи — когда требуется посчитать целочисленный корень (для данного n найти наибольшее x такое, что $x^2 \le n$). Здесь приходится немного изменять условие останова алгоритма, поскольку может случиться, что x начнёт "скакать" возле ответа. Поэтому мы добавляем условие, что если значение x на предыдущем шаге уменьшилось, а на текущем шаге пытается увеличиться, то алгоритм надо остановить.

```
int n;
cin >> n;
```

```
int x = 1;
bool decreased = false;
for (;;) {
        int nx = (x + n / x) >> 1;
        if (x == nx || nx > x && decreased)  break;
        decreased = nx < x;
        x = nx;
}
cout << x;
```

Наконец, приведём ещё третий вариант — для случая длинной арифметики. Поскольку число n может быть достаточно большим, то имеет смысл обратить внимание на начальное приближение. Очевидно, что чем оно ближе к корню, тем быстрее будет достигнут результат. Достаточно простым и эффективным будет брать в качестве начального приближения число $2^{\mathrm{bits}/2}$, где bits — количество битов в числе n. Вот код на языке Java, демонстрирующий этот вариант:

```
BigInteger n; // входные данные

BigInteger a = BigInteger.ONE.shiftLeft (n.bitLength() / 2);
boolean p_dec = false;
for (;;) {
        BigInteger b = n.divide(a).add(a).shiftRight(1);
        if (a.compareTo(b) == 0 || a.compareTo(b) < 0 && p_dec)  break;
        p_dec = a.compareTo(b) > 0;
        a = b;
}
```

Например, этот вариант кода выполняется для числа 10^{1000} за 60 миллисекунд, а если убрать улучшенный выбор начального приближения (просто начинать с 1), то будет выполняться примерно 120 миллисекунд.

Тернарный поиск

Постановка задачи

Пусть дана функция $f(x)$, **унимодальная** на некотором отрезке $[l; r]$. Под унимодальностью понимается один из двух вариантов. Первый: функция сначала строго возрастает, потом достигает максимума (в одной точке или целом отрезке), потом строго убывает. Второй вариант, симметричный: функция сначала убывает убывает, достигает минимума, возрастает. В дальнейшем мы будем рассматривать первый вариант, второй будет абсолютно симметричен ему.

Требуется **найти максимум** функции $f(x)$ на отрезке $[l; r]$.

Алгоритм

Возьмём любые две точки m_1 и m_2 в этом отрезке: $l < m_1 < m_2 < r$. Посчитаем значения функции $f(m_1)$ и $f(m_2)$. Дальше у нас получается три варианта:

- Если окажется, что $f(m_1) < f(m_2)$, то искомый максимум не может находиться в левой части, т.е. в части $[l; m_1]$. В этом легко убедиться: если в левой точке функция меньше, чем в правой, то либо эти две точки находятся в области "подъёма" функции, либо только левая точка находится там. В любом случае, это означает, что максимум дальше имеет смысл искать только в отрезке $[m_1; r]$.
- Если, наоборот, $f(m_1) > f(m_2)$, то ситуация аналогична предыдущей с точностью до симметрии. Теперь искомый максимум не может находиться в правой части, т.е. в части $[m_2; r]$, поэтому переходим к отрезку $[l; m_2]$.
- Если $f(m_1) = f(m_2)$, то либо обе эти точки находятся в области максимума, либо левая точка находится в области возрастания, а правая — в области убывания (здесь существенно используется то, что возрастание/убывание строгие). Таким образом, в дальнейшем поиск имеет смысл производить в отрезке $[m_1; m_2]$, но (в целях упрощения кода) этот случай можно отнести к любому из двух предыдущих.

Таким образом, по результату сравнения значений функции в двух внутренних точках мы вместо текущего отрезка поиска $[l; r]$ находим новый отрезок $[l'; r']$. Повторим теперь все действия для этого нового отрезка, снова получим новый, строго меньший, отрезок, и т.д.

Рано или поздно длина отрезка станет маленькой, меньшей заранее определённой константы-точности, и процесс можно останавливать. Этот метод численный, поэтому после остановки алгоритма можно приближённо считать, что во всех точках отрезка $[l; r]$ достигается максимум; в качестве ответа можно взять, например, точку l.

Осталось заметить, что мы не накладывали никаких ограничений на выбор точек m_1 и m_2. От этого способа, понятно, будет зависеть скорость сходимости (но и возникающая погрешность). Наиболее распространённый способ — выбирать точки так, чтобы отрезок $[l; r]$ делился ими на 3 равные части:

$$m_1 = l + \frac{r - l}{3}$$
$$m_2 = r - \frac{r - l}{3}$$

Впрочем, при другом выборе, когда m_1 и m_2 ближе друг к другу, скорость сходимости несколько увеличится.

Случай целочисленного аргумента

Если аргумент функции f целочисленный, то отрезок $[l; r]$ тоже становится дискретным, однако, поскольку мы не накладывали никаких ограничений на выбор точек m_1 и m_2, то на корректность алгоритма это никак не влияет. Можно по-прежнему выбирать m_1 и m_2 так, чтобы они делили отрезок $[l; r]$ на 3 части, но уже равные только приблизительно.

Второй отличающийся момент — критерий остановки алгоритма. В данном случае тернарный поиск надо будет останавливать, когда станет $r - l < 3$, ведь в таком случае уже невозможно будет выбрать точки m_1 и m_2 так, чтобы были различными и отличались от l и r, и это может привести к зацикливанию. После того, как алгоритм тернарного поиска остановится и станет $r - l < 3$, из оставшихся нескольких точек-кандидатов $(l, l + 1, \ldots, r)$ надо выбрать точку с максимальным значением функции.

Реализация

Реализация для непрерывного случая (т.е. функция f имеет вид: $\mathrm{double\ f\ (double\ x)}$):

```
double l = ..., r = ..., EPS = ...; // входные данные
while (r - l > EPS) {
   double m1 = l + (r - l) / 3,
      m2 = r - (r - l) / 3;
   if (f (m1) < f (m2))
      l = m1;
   else
      r = m2;
}
```

Здесь EPS — фактически, **абсолютная погрешность** ответа (не считая погрешностей, связанных с неточным вычислением функции).

Вместо критерия "while (r - l > EPS)" можно выбрать и такой критерий останова:

```
for (int it=0; it<iterations; ++it)
```

С одной стороны, придётся подобрать константу $\mathrm{iterations}$, чтобы обеспечить требуемую точность (обычно достаточно нескольких сотен, чтобы достичь максимальной точности). Но зато, с другой стороны, число итераций перестаёт зависеть от абсолютных величин l и r, т.е. мы фактически с помощью $\mathrm{iterations}$ задаём требуемую **относительную погрешность**.

Биномиальные коэффициенты

Биномиальным коэффициентом C_n^k называется количество способов выбрать набор k предметов из n различных предметов без учёта порядка расположения этих элементов (т.е. количество неупорядоченных наборов).

Также биномиальные коэффициенты - это коффициенты в разложении $(a+b)^n$ (т.н. бином Ньютона):

$$(a+b)^n = C_n^0 a^n + C_n^1 a^{n-1} b + C_n^2 a^{n-2} b^2 + \ldots + C_n^k a^{n-k} b^k + \ldots + C_n^n b^n$$

Считается, что эту формулу, как и треугольник, позволяющий эффективно находить коэффициенты, открыл Блез Паскаль (Blaise Pascal), живший в 17 в. Тем не менее, она была известна ещё китайскому математику Яну Хуэю (Yang Hui), жившему в 13 в. Возможно, её открыл персидский учёный Омар Хайям (Omar Khayyam). Более того, индийский математик Пингала (Pingala), живший ещё в 3 в. до н.э., получил близкие результаты. Заслуга же Ньютона заключается в том, что он обобщил эту формулу для степеней, не являющихся натуральными.

Вычисление

Аналитическая формула для вычисления:

$$C_n^k = \frac{n!}{k!(n-k)!}$$

Эту формулу легко вывести из задачи о неупорядоченной выборке (количество способов неупорядоченно выбрать k элементов из n элементов). Сначала посчитаем количество упорядоченных выборок. Выбрать первый элемент есть n способов, второй — $n-1$, третий — $n-2$, и так далее. В результате для числа упорядоченных выборок получаем формулу: $n(n-1)(n-2)\ldots(n-k+1) = \frac{n!}{(n-k)!}$. К неупорядоченным выборкам легко перейти, если заметить, что каждой неупорядоченной выборке соответствует ровно $k!$ упорядоченных (т.к. это количество всевозможных перестановок k элементов). В результате, деля $\frac{n!}{(n-k)!}$ на $k!$, мы и получаем искомую формулу.

Рекуррентная формула (с которой связан знаменитый "треугольник Паскаля"):

$$C_n^k = C_{n-1}^{k-1} + C_{n-1}^k$$

Её легко вывести через предыдущую формулу.

Стоит заметить особо, при $n < k$ значение C_n^k всегда полагается равным нулю.

Свойства

Биномиальные коэффициенты обладают множеством различных свойств, приведём наиболее простые из них:

- Правило симметрии:

  $$C_n^k = C_n^{n-k}$$

- Внесение-вынесение:

  $$C_n^k = \frac{n}{k} C_{n-1}^{k-1}$$

- Суммирование по k:

  $$\sum_{k=0}^{n} C_n^k = 2^n$$

- Суммирование по n:

  $$\sum_{m=0}^{n} C_m^k = C_{n+1}^{k+1}$$

- Суммирование по n и k:

$$\sum_{k=0}^{m} C_{n+k}^{k} = C_{n+m+1}^{m}$$

- Суммирование квадратов:

$$(C_n^0)^2 + (C_n^1)^2 + \ldots + (C_n^n)^2 = C_{2n}^n$$

- Взвешенное суммирование:

$$1C_n^1 + 2C_n^2 + \ldots + nC_n^n = n2^{n-1}$$

- Связь с числами Фибоначчи:

$$C_n^0 + C_{n-1}^1 + \ldots + C_{n-k}^k + \ldots + C_0^n = F_{n+1}$$

Вычисления в программе

Непосредственные вычисления по аналитической формуле

Вычисления по первой, непосредственной формуле, очень легко программировать, однако этот способ подвержен переполнениям даже при сравнительно небольших значениях n и k (даже если ответ вполне помещается в какой-нибудь тип данных, вычисление промежуточных факториалов может привести к переполнению). Поэтому очень часто этот способ можно применять только вместе с [[Длинная арифметика|Длинной арифметикой]]:

```
int C (int n, int k) {
        int res = 1;
        for (int i=n-k+1; i<=n; ++i)
                res *= i;
        for (int i=2; i<=k; ++i)
                res /= i;
}
```

Улучшенная реализация

Можно заметить, что в приведённой выше реализации в числителе и знаменателе стоит одинаковое количество сомножителей (k), каждый из которых не меньше единицы. Поэтому можно заменить нашу дробь на произведение k дробей, каждая из которых является вещественнозначной. Однако, можно заметить, что после домножения текущего ответа на каждую очередную дробь всё равно будет получаться целое число (это, например, следует из свойства "внесения-вынесения"). Таким образом, получаем такую реализацию:

```
int C (int n, int k) {
        double res = 1;
        for (int i=1; i<=k; ++i)
                res = res * (n-k+i) / i;
        return (int) (res + 0.01);
}
```

Здесь мы аккуратно приводим дробное число к целому, учитывая, что из-за накапливающихся погрешностей оно может оказаться чуть меньше истинного значения (например, 2.99999 вместо трёх).

Треугольник Паскаля

С использованием же рекуррентного соотношения можно построить таблицу биномиальных коэффициентов (фактически, треугольник Паскаля), и из неё брать результат. Преимущество этого метода в том, что промежуточные результаты никогда не превосходят ответа, и для вычисления каждого нового элемента таблицы надо всего лишь одно сложение. Недостатком является медленная работа для больших N и K, если на самом деле таблица не нужна, а нужно единственное значение (потому что для вычисления C_n^k понадобится строить таблицу для всех $C_i^j, \quad 1 \le i \le n, \quad 1 \le j \le n$ или хотя бы до $1 \le j \le \min(i, 2k)$).

```
const int maxn = ...;
int C[maxn+1][maxn+1];
for (int n=0; n<=maxn; ++n) {
        C[n][0] = C[n][n] = 1;
        for (int k=1; k<n; ++k)
                C[n][k] = C[n-1][k-1] + C[n-1][k];
}
```

Если вся таблица значений не нужна, то, как нетрудно заметить, достаточно хранить от неё только две строки (текущую — n-ую строку и предыдущую — $n-1$-ую).

Вычисление за O(1)

Наконец, в некоторых ситуациях оказывается выгодно предпосчитать заранее значения всех факториалов, с тем, чтобы впоследствии считать любой необходимый биномиальный коэффициент, производя лишь два деления. Это может быть выгодно при использовании Длинной арифметики, когда память не позволяет предпосчитать весь треугольник Паскаля, или же когда требуется производить расчёты по некоторому простому модулю (если модуль не простой, то возникают сложности при делении числителя дроби на знаменатель; их можно преодолеть, если факторизовать модуль и хранить все числа в виде векторов из степеней этих простых; см раздел "Длинная арифметика в факторизованном виде").

Числа Каталана

Числа Каталана — числовая последовательность, встречающаяся в удивительном числе комбинаторных задач.

Эта последовательность названа в честь бельгийского математика Каталана (Catalan), жившего в 19 веке, хотя на самом деле она была известна ещё Эйлеру (Euler), жившему за век до Каталана.

Последовательность

Первые несколько чисел Каталана C_n (начиная с нулевого):

$$1,\ 1,\ 2,\ 5,\ 14,\ 42,\ 132,\ 429,\ 1430,\ \ldots$$

Числа Каталана встречаются в большом количестве задач комбинаторики. n-**ое число Каталана** — это:

- Количество корректных скобочных последовательностей, состоящих из n открывающих и n закрывающих скобок.
- Количество корневых бинарных деревьев с $n+1$ листьями (вершины не пронумерованы).
- Количество способов полностью разделить скобками $n+1$ множитель.
- Количество триангуляций выпуклого $n+2$-угольника (т.е. количество разбиений многоугольника непересекающимися диагоналями на треугольники).
- Количество способов соединить $2n$ точек на окружности n непересекающимися хордами.
- Количество неизоморфных полных бинарных деревьев с n внутренними вершинами (т.е. имеющими хотя бы одного сына).
- Количество монотонных путей из точки $(0,0)$ в точку (n,n) в квадратной решётке размером $n \times n$, не поднимающихся над главной диагональю.
- Количество перестановок длины n, которые можно отсортировать стеком (можно показать, что перестановка является сортируемой стеком тогда и только тогда, когда нет таких индексов $i < j < k$, что $a_k < a_i < a_j$).
- Количество непрерывных разбиений множества из n элементов (т.е. разбиений на непрерывные блоки).
- Количество способов покрыть лесенку $1 \ldots n$ с помощью n прямоугольников (имеется в виду фигура, состоящая из n столбцов, i-ый из которых имеет высоту i).

Вычисление

Имеется две формулы для чисел Каталана: рекуррентная и аналитическая. Поскольку мы считаем, что все приведённые выше задачи эквивалентны, то для доказательства формул мы будем выбирать ту задачу, с помощью которой это сделать проще всего.

Рекуррентная формула

$$C_n = \sum_{k=0}^{n-1} C_k C_{n-1-k}$$

Рекуррентную формулу легко вывести из задачи о правильных скобочных последовательностях.

Самой левой открывающей скобке l соответствует определённая закрывающая скобка r, которая разбивает формулу две части, каждая из которых в свою очередь является правильной скобочной последовательностью. Поэтому, если мы обозначим $k = r - l - 1$, то для любого фиксированного r будет ровно $C_k C_{n-1-k}$ способов. Суммируя это по всем допустимым k, мы и получаем рекуррентную зависимость на C_n.

Аналитическая формула

$$C_n = \frac{1}{n+1} C_{2n}^n$$

(здесь через C_n^k обозначен, как обычно, биномиальный коэффициент).

Эту формулу проще всего вывести из задачи о монотонных путях. Общее количество монотонных путей в решётке размером $n \times n$ равно C_{2n}^n. Теперь посчитаем количество монотонных путей, пересекающих диагональ. Рассмотрим какой-либо из таких путей, и найдём первое ребро, которое стоит выше диагонали. Отразим относительно диагонали весь путь, идущий после этого ребра. В результате получим монотонный путь в решётке $(n-1) \times (n+1)$. Но, с другой стороны, любой монотонный путь в решётке $(n-1) \times (n+1)$ обязательно пересекает диагональ, следовательно, он получен как раз таким способом из

какого-либо (причём единственного) монотонного пути, пересекающего диагональ, в решётке $n \times n$. Монотонных путей в решётке $(n-1) \times (n+1)$ имеется C_{2n}^{n-1}. В результате получаем формулу:

$$C_n = C_{2n}^n - C_{2n}^{n-1} = \frac{1}{n+1} C_{2n}^n$$

Ожерелья

Задача "ожерелья" — это одна из классических комбинаторных задач. Требуется посчитать количество различных ожерелий из n бусинок, каждая из которых может быть покрашена в один из k цветов. При сравнении двух ожерелий их можно поворачивать, но не переворачивать (т.е. разрешается сделать циклический сдвиг).

Решение

Решить эту задачу можно, используя лемму Бернсайда и теорему Пойа. [Ниже идёт копия текста из этой статьи]

В этой задаче мы можем сразу найти группу инвариантных перестановок. Очевидно, она будет состоять из n перестановок:

$$\begin{aligned}
\pi_0 &= 1\ 2\ 3\ \ldots\ n \\
\pi_1 &= 2\ 3\ \ldots\ n\ 1 \\
\pi_2 &= 3\ \ldots\ n\ 1\ 2 \\
&\ldots \\
\pi_{n-1} &= n\ 1\ 2\ \ldots\ (n-1)
\end{aligned}$$

Найдём явную формулу для вычисления $C(\pi_i)$. Во-первых, заметим, что перестановки имеют такой вид, что в i-ой перестановке на j-ой позиции стоит $i+j$ (взятое по модулю n, если оно больше n). Если мы будем рассматривать циклическую структуру i-ой перестановки, то увидим, что единица переходит в $1+i$, $1+i$ переходит в $1+2i$, $1+2i$ — в $1+3i$, и т.д., пока не придём в число $1+kn$; для остальных элементов выполняются похожие утверждения. Отсюда можно понять, что все циклы имеют одинаковую длину, равную $\mathrm{lcm}(i,n)/i$, т.е. $n/\gcd(i,n)$ ("gcd" — наибольший общий делитель, "lcm" — наименьшее общее кратное). Тогда количество циклов в i-ой перестановке будет равно просто $\gcd(i,n)$.

Подставляя найденные значения в теорему Пойа, получаем **решение**:

$$\mathrm{Ans} = \frac{1}{n}\sum_{i=1}^{n} k^{\gcd(i,n)}$$

Можно оставить формулу в таком виде, а можно её свернуть ещё больше. Перейдём от суммы по всем i к сумме только по делителям n. Действительно, в нашей сумме будет много одинаковых слагаемых: если i не является делителем n, то таковой делитель найдётся после вычисления $\gcd(i,n)$. Следовательно, для каждого делителя $d|n$ его слагаемое $k^{\gcd(d,n)} = k^d$ учтётся несколько раз, т.е. сумму можно представить в таком виде:

$$\mathrm{Ans} = \frac{1}{n}\sum_{d|n} C_d k^d$$

где C_d — это количество таких чисел i, что $\gcd(i,n) = d$. Найдём явное выражение для этого количества. Любое такое число i имеет вид: $i = dj$, где $\gcd(j, n/d) = 1$ (иначе было бы $\gcd(i,n) > d$). Вспоминая функцию Эйлера, мы находим, что количество таких j — это величина функции Эйлера $\phi(n/d)$. Таким образом, $C_d = \phi(n/d)$, и окончательно получаем **формулу**:

$$\mathrm{Ans} = \frac{1}{n}\sum_{d|n} \phi\left(\frac{n}{d}\right) k^d$$

Расстановка слонов на шахматной доске

Требуется найти количество способов расставить K слонов на доске размером NxN.

Алгоритм

Решать задачу будем с помощью **динамического программирования**.

Пусть D[i][j] - количество способов расставить j слонов на диагоналях до i-ой включительно, причём только тех диагоналях, которые того же цвета, что и i-ая диагональ. Тогда i = 1..2N-1, j = 0..K.

Диагонали занумеруем следующим образом (пример для доски 5x5):

```
черные:     белые:
1 _ 5 _ 9   _ 2 _ 6 _
_ 5 _ 9 _   2 _ 6 _ 8
5 _ 9 _ 7   _ 6 _ 8 _
_ 9 _ 7 _   6 _ 8 _ 4
9 _ 7 _ 3   _ 8 _ 4 _
```

Т.е. нечётные номера соответствуют чёрным диагоналям, чётные - белым; диагонали нумеруем в порядке увеличения количества элементов в них.

При такой нумерации мы можем вычислить каждое D[i][], основываясь только на D[i-2][] (двойка вычитается, чтобы мы рассматривали диагональ того же цвета).

Итак, пусть текущий элемент динамики - D[i][j]. Имеем два перехода. Первый - D[i-2][j], т.е. ставим всех j слонов на предыдущие диагонали. Второй переход - если мы ставим одного слона на текущую диагональ, а остальных j-1 слонов - на предыдущие; заметим, что количество способов поставить слона на текущую диагональ равно количеству клеток в ней минус j-1, т.к. слоны, стоящие на предыдущих диагоналях, будут перекрывать часть направлений. Таким образом, имеем:

```
D[i][j] = D[i-2][j] + D[i-2][j-1] (cells(i) - j + 1)
```

где cells(i) - количество клеток, лежащих на i-ой диагонали. Например, cells можно вычислять так:

```
int cells (int i) {
        if (i & 1)
                return i / 4 * 2 + 1;
        else
                return (i - 1) / 4 * 2 + 2;
}
```

Осталось определить базу динамики, тут никаких сложностей нет: D[i][0] = 1, D[1][1] = 1.

Наконец, вычислив динамику, найти собственно **ответ** к задаче несложно. Перебираем количество i=0..K слонов, стоящих на чёрных диагоналях (номер последней чёрной диагонали - 2N-1), соответственно K-i слонов ставим на белые диагонали (номер последней белой диагонали - 2N-2), т.е. к ответу прибавляем величину D[2N-1][i] * D[2N-2][K-i].

Реализация

```
int n, k; // входные данные
if (k > 2*n-1) {
        cout << 0;
        return 0;
}

vector < vector<int> > d (n*2, vector<int> (k+2));
for (int i=0; i<n*2; ++i)
        d[i][0] = 1;
d[1][1] = 1;
for (int i=2; i<n*2; ++i)
        for (int j=1; j<=k; ++j)
                d[i][j] = d[i-2][j] + d[i-2][j-1] * (cells(i) - j + 1);

int ans = 0;
```

```
for (int i=0; i<=k; ++i)
        ans += d[n*2-1][i] * d[n*2-2][k-i];
cout << ans;
```

Правильные скобочные последовательности

Правильной скобочной последовательностью называется строка, состоящая только из символов "скобки" (чаще всего рассматриваются только круглые скобки, но здесь будет рассматриваться и общий случай нескольких типов скобок), где каждой закрывающей скобке найдётся соответствующая открывающая (причём того же типа).

Здесь мы рассмотрим классические задачи на правильные скобочные последовательности (далее для краткости просто "последовательности"): проверка на правильность, количество последовательностей, генерация всех последовательностей, нахождение лексикографически следующей последовательности, нахождение k-ой последовательности в отсортированном списке всех последовательностей, и, наоборот, определение номера последовательности. Каждая из задач рассмотрена в двух случаях — когда разрешены скобки только одного типа, и когда нескольких типов.

Проверка на правильность

Пусть сначала разрешены скобки только одного типа, тогда проверить последовательность на правильность можно очень простым алгоритмом. Пусть depth — это текущее количество открытых скобок. Изначально $\mathrm{depth} = 0$. Будем двигаться по строке слева направо, если текущая скобка открывающая, то увеличим depth на единицу, иначе уменьшим. Если при этом когда-то получалось отрицательное число, или в конце работы алгоритма depth отлично от нуля, то данная строка не является правильной скобочной последовательностью, иначе является.

Если допустимы скобки нескольких типов, то алгоритм нужно изменить. Вместо счётчика depth следует создать стек, в который будем класть открывающие скобки по мере поступления. Если текущий символ строки — открывающая скобка, то кладём его в стек, а если закрывающая — то проверяем, что стек не пуст, и что на его вершине лежит скобка того же типа, что и текущая, и затем достаём эту скобку из стека. Если какое-либо из условий не выполнилось, или в конце работы алгоритма стек остался не пуст, то последовательность не является правильной скобочной, иначе является.

Таким образом, обе эти задачи мы научились решать за время $O(n)$.

Количество последовательностей

Формула

Количество правильных скобочных последовательностей с одним типом скобок можно вычислить как **число Каталана**. Т.е. если есть n пар скобок (строка длины $2n$), то количество будет равно:

$$\frac{1}{n+1} \cdot C_{2n}^n.$$

Пусть теперь имеется не один, а k типов скобок. Тогда каждая пара скобок независимо от остальных может принимать один из k типов, а потому мы получаем такую формулу:

$$\frac{1}{n+1} \cdot C_{2n}^n \cdot k^n.$$

Динамическое программирование

С другой стороны, к этой задаче можно подойти и с точки зрения **динамического программирования**. Пусть $d[n]$ — количество правильных скобочных последовательностей из n пар скобок. Заметим, что в первой позиции всегда будет стоять открывающая скобка. Понятно, что внутри этой пары скобок стоит какая-то правильная скобочная последовательность; аналогично, после этой пары скобок также стоит правильная скобочная последовательность. Теперь чтобы посчитать $d[n]$, переберём, сколько пар скобок i будет стоять внутри этой первой пары, тогда, соответственно, $n-1-i$ пара скобок будет стоять после этой первой пары. Следовательно, формула для $d[n]$ имеет вид:

$$d[n] = \sum_{i=0}^{n-1} d[i] \cdot d[n-1-i].$$

Начальное значение для этой рекуррентной формулы — это $d[0] = 1$.

Нахождение всех последовательностей

Иногда требуется найти и вывести все правильные скобочные последовательности указанной длины n (в данном случае n — это длина строки).

Для этого можно начать с лексикографически первой последовательности $((\ldots(())\ldots))$, а затем находить каждый раз лексикографически следующую последовательность с помощью алгоритма, описанного в следующем разделе.

Но если ограничения не очень большие (n до $10-12$), то можно поступить значительно проще. Найдём всевозможные перестановки этих скобок (для этого удобно использовать функцию next_permutation()), их будет C_{2n}^n, и каждую проверим на правильность вышеописанным алгоритмом, и в случае правильности выведем текущую последовательность.

Также процесс нахождения всех последовательностей можно оформить в виде рекурсивного перебора с отсечениями (что в идеале можно довести по скорости работы до первого алгоритма).

Нахождение следующей последовательности

Здесь рассматривается только случай одного типа скобок.

По заданной правильной скобочной последовательности требуется найти правильную скобочную последовательность, которая находится следующей в лексикографическом порядке после текущей (или выдать "No solution", если такой не существует).

Понятно, что в целом алгоритм выглядит следующим образом: найдем такую самую правую открывающую скобку, которую мы имеем право заменить на закрывающую (так, чтобы в этом месте правильность не нарушалась), а всю оставшуюся справа строку заменим на лексикографически минимальную: т.е. сколько-то открывающих скобок, затем все оставшиеся закрывающие скобки. Иными словами, мы пытаемся оставить без изменения как можно более длинный префикс исходной последовательности, а в суффиксе эту последовательность заменяем на лексикографически минимальную.

Осталось научиться искать эту самую позицию первого изменения. Для этого будем идти по строке справа налево и поддерживать баланс depth открытых и закрытых скобок (при встрече открывающей скобки будем уменьшать depth, а при закрывающей — увеличивать). Если в какой-то момент мы стоим на открывающей скобке, а баланс после обработки этого символа больше нуля, то мы нашли самую правую позицию, от которой мы можем начать изменять последовательность (в самом деле, $\mathrm{depth} > 0$ означает, что слева имеется не закрытая ещё скобка). Поставим в текущую позицию закрывающую скобку, затем максимально возможное количество открывающих скобок, а затем все оставшиеся закрывающие скобки, — ответ найден.

Если мы просмотрели всю строку и так и не нашли подходящую позицию, то текущая последовательность — максимальна, и ответа не существует.

Реализация алгоритма:

```
string s;
cin >> s;
int n = (int) s.length();
string ans = "No solution";
for (int i=n-1, depth=0; i>=0; --i) {
        if (s[i] == '(')
                --depth;
        else
                ++depth;
        if (s[i] == '(' && depth > 0) {
                --depth;
                int open = (n-i-1 - depth) / 2;
                int close = n-i-1 - open;
                ans = s.substr(0,i) + ')' + string ('(', open) + string
(')', close);
                break;
        }
}
cout << ans;
```

Таким образом, мы решили эту задачу за $O(n)$.

Номер последовательности

Здесь пусть n — количество пар скобок в последовательности. Требуется по заданной правильной скобочной последовательности найти её номер в списке лексикографически упорядоченных правильных скобочных последовательностей.

Научимся считать вспомогательную **динамику** $d[i][j]$, где i — длина скобочной последовательности (она "полуправильна": всякой закрывающей скобке имеется парная открывающая, но не все открытые скобки закрыты), j — баланс (т.е. разность между количеством открывающих и закрывающих скобок), $d[i][j]$ — количество таких последовательностей. При подсчёте этой динамики мы считаем, что скобки бывают только одного типа.

Считать эту динамику можно следующим образом. Пусть $d[i][j]$ — величина, которую мы хотим посчитать. Если $i = 0$, то ответ понятен сразу: $d[0][0] = 1$, все остальные $d[0][j] = 0$. Пусть теперь $i > 0$, тогда мысленно переберём, чему был равен последний символ этой последовательности. Если он был равен '(', то до этого символа мы находились в состоянии $(i-1, j-1)$. Если он был равен ')', то предыдущим было состояние $(i-1, j+1)$. Таким образом, получаем формулу:

$$d[i][j] = d[i-1][j-1] + d[i-1][j+1]$$

(считается, что все значения $d[i][j]$ при отрицательном j равны нулю). Таким образом, эту динамику мы можем посчитать за $O(n^2)$.

Перейдём теперь к решению самой задачи.

Сначала пусть допустимы только скобки **одного** типа. Заведём счётчик depth глубины вложенности в скобки, и будем двигаться по последовательности слева направо. Если текущий символ $s[i] (i = 0 \ldots 2n-1)$ равен '(', то мы увеличиваем depth на 1 и переходим к следующему символу. Если же текущий символ равен ')', то мы должны прибавить к ответу $d[2n-i-1][\text{depth}+1]$, тем самым учитывая, что в этой позиции мог бы стоять символ '(' (который бы привёл к лексикографически меньшей последовательности, чем текущая); затем мы уменьшаем depth на единицу.

Пусть теперь разрешены скобки **нескольких** k типов. Тогда при рассмотрении текущего символа $s[i]$ до пересчёта depth мы должны перебирать все скобки, которые меньше текущего символа, пробовать ставить эту скобку в текущую позицию (получая тем самым новый баланс $\text{ndepth} = \text{depth} \pm 1$), и и прибавлять к ответу количество соответствующих "хвостов" - завершений (которые имеют длину $2n-i-1$, баланс ndepth и k типов скобок). Утверждается, что формула для этого количества имеет вид:

$$d[2n-i-1][\text{ndepth}] \cdot k^{(2n-i-1-\text{ndepth})/2}.$$

Эта формула выводится из следующих соображений. Сначала мы "забываем" про то, что скобки бывают нескольких типов, и просто берём ответ из $d[2n-i-1][\text{ndepth}]$. Теперь посчитаем, как изменится ответ из-за наличия k типов скобок. У нас имеется $2n-i-1$ неопределённых позиций, из которых ndepth являются скобками, закрывающими какие-то из открытых ранее, — значит, тип таких скобок мы варьировать не можем. А вот все остальные скобки (а их будет $(2n-i-1-\text{ndepth})/2$ пар) могут быть любого из k типов, поэтому ответ умножается на эту степень числа k.

Нахождение k-ой последовательности

Здесь пусть n — количество пар скобок в последовательности. В данной задаче по заданному k требуется найти k-ую правильную скобочную последовательность в списке лексикографически упорядоченных последовательностей.

Как и в предыдущем разделе, посчитаем **динамику** $d[i][j]$ — количество правильных скобочных последовательностей длины i с балансом j.

Пусть сначала допустимы только скобки **одного** типа.

Будем двигаться по символам искомой строки, с 0-го по $2n-1$-ый. Как и в предыдущей задаче, будем хранить счётчик depth — текущую глубину вложенности в скобки. В каждой текущей позиции будем перебирать возможный символ - открывающую скобку или закрывающую. Пусть мы хотим поставить сюда открывающую скобку, тогда мы должны посмотреть на значение $d[i+1][\text{depth}+1]$. Если оно $\geq k$, то мы ставим в текущую позицию открывающую скобку, увеличиваем depth на единицу и переходим к следующему символу. Иначе мы отнимаем от k величину $d[i+1][\text{depth}+1]$, ставим закрывающую скобку и уменьшаем значение depth. В конце концов мы и получим искомую скобочную последовательность.

Реализация на языке Java с использованием длинной арифметики:

```
int n;  BigInteger k;  // входные данные

BigInteger d[][] = new BigInteger [n*2+1][n+1];
for (int i=0; i<=n*2; ++i)
```

```
        for (int j=0; j<=n; ++j)
                d[i][j] = BigInteger.ZERO;
d[0][0] = BigInteger.ONE;
for (int i=0; i<n*2; ++i)
        for (int j=0; j<=n; ++j) {
                if (j+1 <= n)
                        d[i+1][j+1] = d[i+1][j+1].add( d[i][j] );
                if (j > 0)
                        d[i+1][j-1] = d[i+1][j-1].add( d[i][j] );
        }

String ans = new String();
if (k.compareTo( d[n*2][0] ) > 0)
        ans = "No solution";
else {
        int depth = 0;
        for (int i=n*2-1; i>=0; --i)
                if (depth+1 <= n && d[i][depth+1].compareTo( k ) >= 0) {
                        ans += '(';
                        ++depth;
                }
                else {
                        ans += ')';
                        if (depth+1 <= n)
                                k = k.subtract( d[i][depth+1] );
                        --depth;
                }
}
```

Пусть теперь разрешён не один, а k **типов** скобок. Тогда алгоритм решения будет отличаться от предыдущего случая только тем, что мы должны домножать значение $D[i+1][\mathrm{ndepth}]$ на величину $k^{(2n-i-1-\mathrm{ndepth})/2}$, чтобы учесть, что в этом остатке могли быть скобки различных типов, а парных скобок в этом остатке будет только $2n-i-1-\mathrm{ndepth}$, поскольку ndepth скобок являются закрывающими для открывающих скобок, находящихся вне этого остатка (а потому их типы мы варьировать не можем).

Реализация на языке Java для случая двух типов скобок - круглых и квадратных:

```
int n;  BigInteger k;  // входные данные

BigInteger d[][] = new BigInteger [n*2+1][n+1];
for (int i=0; i<=n*2; ++i)
        for (int j=0; j<=n; ++j)
                d[i][j] = BigInteger.ZERO;
d[0][0] = BigInteger.ONE;
for (int i=0; i<n*2; ++i)
        for (int j=0; j<=n; ++j) {
                if (j+1 <= n)
                        d[i+1][j+1] = d[i+1][j+1].add( d[i][j] );
                if (j > 0)
                        d[i+1][j-1] = d[i+1][j-1].add( d[i][j] );
        }

String ans = new String();
int depth = 0;
char [] stack = new char[n*2];
int stacksz = 0;
for (int i=n*2-1; i>=0; --i) {
        BigInteger cur;
        // '('
        if (depth+1 <= n)
                cur = d[i][depth+1].shiftLeft( (i-depth-1)/2 );
        else
                cur = BigInteger.ZERO;
        if (cur.compareTo( k ) >= 0) {
                ans += '(';
                stack[stacksz++] = '(';
                ++depth;
                continue;
        }
        k = k.subtract( cur );
```

```
        // ')'
        if (stacksz > 0 && stack[stacksz-1] == '(' && depth-1 >= 0)
                cur = d[i][depth-1].shiftLeft( (i-depth+1)/2 );
        else
                cur = BigInteger.ZERO;
        if (cur.compareTo( k ) >= 0) {
                ans += ')';
                --stacksz;
                --depth;
                continue;
        }
        k = k.subtract( cur );
        // '['
        if (depth+1 <= n)
                cur = d[i][depth+1].shiftLeft( (i-depth-1)/2 );
        else
                cur = BigInteger.ZERO;
        if (cur.compareTo( k ) >= 0) {
                ans += '[';
                stack[stacksz++] = '[';
                ++depth;
                continue;
        }
        k = k.subtract( cur );
        // ']'
        ans += ']';
        --stacksz;
        --depth;
}
```

Количество помеченных графов

Дано число N вершин. Требуется посчитать количество G_N различных помеченных графов с N вершинами (т. е. вершины графа помечены различными числами от 1 до N, и графы сравниваются с учётом этой покраски вершин). Рёбра графа неориентированы, петли и кратные рёбра запрещены.

Рассмотрим множество всех возможных рёбер графа. Для любого ребра (i, j) положим, что $i < j$ (основываясь на неориентированности графа и отсутствии петель). Тогда множество всех возможных рёбер графа имеет мощность C_N^2, т.е. $\frac{N(N-1)}{2}$.

Поскольку любой помеченный граф однозначно определяется своими рёбрами, то количество помеченных графов с N вершинами равно:

$$G_N = 2^{\frac{N(N-1)}{2}}$$

Количество связных помеченных графов

По сравнению с предыдущей задачей мы дополнительно накладываем ограничение, что граф должен быть связным.

Обозначим искомое число через $Conn_N$.

Научимся, наоборот, считать количество **несвязных** графов; тогда количество связных графов получится как G_N минус найденное число. Более того, научимся считать количество **корневых** (т.е. с выделенной вершиной - корнем) **несвязных графов**; тогда количество несвязных графов будет получаться из него делением на N. Заметим, что, так как граф несвязный, то в нём найдётся компонента связности, внутри которой лежит корень, а остальной граф будет представлять собой ещё несколько (как минимум одну) компонент связности.

Переберём количество K вершин в этой компоненте связности, содержащей корень (очевидно, $K = 1 \dots N-1$), и найдём количество таких графов. Во-первых, мы должны выбрать K вершин из N, т.е. ответ умножается на C_N^K. Во-вторых, компонента связности с корнем даёт множитель $Conn_K$. В-третьих, оставшийся граф из $N - K$ вершин является произвольным графом, а потому он даёт множитель G_{N-K}. Наконец, количество способов выделить корень в компоненте связности из K вершин равно K. Итого, при фиксированном K количество **корневых несвязных** графов равно:

$$K\ C_N^K\ Conn_K\ G_{N-K}$$

Значит, количество **несвязных** графов с N вершинами равно:

$$\frac{1}{N} \sum_{K=1}^{N-1} K\ C_N^K\ Conn_K\ G_{N-K}$$

Наконец, искомое количество **связных** графов равно:

$$Conn_N = G_N - \frac{1}{N} \sum_{K=1}^{N-1} K\ C_N^K\ Conn_K\ G_{N-K}$$

Количество помеченных графов с K компонентами связности

Основываясь на предыдущей формуле, научимся считать количество помеченных графов с N вершинами и K компонентами связности.

Сделать это можно с помощью динамического программирования. Научимся считать $D[N][K]$— количество помеченных графов с N вершинами и K компонентами связности.

Научимся вычислять очередной элемент $D[N][K]$, зная предыдущие значения. Воспользуемся стандартным приёмом при решении таких задач: возьмём вершину с номером 1, она принадлежит какой-то компоненте, вот эту компоненту мы и будем перебирать. Переберём размер S этой компоненты, тогда количество способов выбрать такое множество вершин равно C_{N-1}^{S-1} (одну вершину — вершину 1 — перебирать не надо). Количество же

способов построить компоненту связности из S вершин мы уже умеем считать — это $Conn_S$. После удаления этой компоненты из графа у нас остаётся граф с $N - S$ вершинами и $K - 1$ компонентами связности, т.е. мы получили рекуррентную зависимость, по которой можно вычислять значения $D[][]$:

$$D[N][K] = \sum_{S=1}^{N} C_{N-1}^{S-1}\ Conn_S\ D[N-S][K-1]$$

Итого получаем примерно такой код:

```
int d[n+1][k+1]; // изначально заполнен нулями
d[0][0][0] = 1;
for (int i=1; i<=n; ++i)
        for (int j=1; j<=i && j<=k; ++j)
                for (int s=1; s<=i; ++s)
                        d[i][j] += C[i-1][s-1] * conn[s] * d[i-s][j-1];
cout << d[n][k][n];
```

Разумеется, на практике, скорее всего, нужна будет длинная арифметика.

Генерация сочетаний из N элементов

Сочетания из N элементов по K в лексикографическом порядке

Постановка задачи. Даны натуральные числа N и K. Рассмотрим множество чисел от 1 до N. Требуется вывести все различные его подмножества мощности K, причём в лексикографическом порядке.

Алгоритм весьма прост. Первым сочетанием, очевидно, будет сочетание (1,2,...,K). Научимся для текущего сочетания находить лексикографически следующее. Для этого в текущем сочетании найдём самый правый элемент, не достигший ещё своего наибольшего значения; тогда увеличим его на единицу, а всем последующим элементам присвоим наименьшие значения.

```
bool next_combination (vector<int> & a, int n) {
        int k = (int)a.size();
        for (int i=k-1; i>=0; --i)
                if (a[i] < n-k+i+1) {
                        ++a[i];
                        for (int j=i+1; j<k; ++j)
                                a[j] = a[j-1]+1;
                        return true;
                }
        return false;
}
```

С точки зрения производительности, этот алгоритм линеен (в среднем), если K не близко к N (т.е. если не выполняется, что K = N - o(N)). Для этого достаточно доказать, что сравнения "a[i] < n-k+i+1" выполняются в сумме C_{n+1}^{k} раз, т.е. в (N +1) / (N-K+1) раз больше, чем всего есть сочетаний из N элементов по K.

Сочетания из N элементов по K с изменениями ровно одного элемента

Требуется выписать все сочетания из N элементов по K, но в таком порядке, что любые два соседних сочетания будут отличаться ровно одним элементом.

Интуитивно можно сразу заметить, что эта задача похожа на задачу генерации всех подмножеств данного множества в таком порядке, когда два соседних подмножества отличаются ровно одним элементом. Эта задача непосредственно решается с помощью Кода Грея: если мы каждому подмножеству поставим в соответствие битовую маску, то, генерируя с помощью кодов Грея эти битовые маски, мы и получим ответ.

Может показаться удивительным, но задача генерации сочетаний также непосредственно решается с помощью **кода Грея**. А именно, сгенерируем коды Грея для чисел от 0 до 2^N-1, и оставим только те коды, которые содержат ровно K единиц. Удивительный факт заключается в том, что в полученной последовательности любые две соседние маски (а также первая и последняя маски) будут отличаться ровно двумя битами, что нам как раз и требуется.

Докажем это.

Для доказательства вспомним факт, что последовательность G(N) кодов Грея можно получить следующим образом:

$$G(N) = 0G(N-1) \cup 1G(N-1)^R$$

т.е. берём последовательность кодов Грея для N-1, дописываем в начало каждой маски 0, добавляем к ответу; затем снова берём последовательность кодов Грея для N-1, инвертируем её, дописываем в начало каждой маски 1 и добавляем к ответу.

Теперь мы можем произвести доказательство.

Сначала докажем, что первая и последняя маски будут отличаться ровно в двух битах. Для этого достаточно заметить, что первая маска будет иметь вид N-K нулей и K единиц, а последняя маска будет иметь вид: единица, потом N-K-1 нулей, потом K-1 единица. Доказать это легко по индукции по N, пользуясь приведённой выше формулой для последовательности кодов Грея.

Теперь докажем, что любые два соседних кода будут отличаться ровно в двух битах. Для этого снова обратимся к формуле для последовательности кодов Грея. Пусть внутри каждой из половинок (образованных из G(N-1)) утверждение верно, докажем, что оно верно для всей последовательности. Для этого достаточно доказать, что оно верно в месте "склеивания" двух половинок G(N-1), а это легко показать, основываясь на том, что мы знаем первый и последний элементы этих половинок.

Приведём теперь наивную реализацию, работающую за 2^N:

```
int gray_code (int n) {
        return n ^ (n >> 1);
}

int count_bits (int n) {
        int res = 0;
        for (; n; n>>=1)
                res += n & 1;
        return res;
}

void all_combinations (int n, int k) {
        for (int i=0; i<(1<<n); ++i) {
                int cur = gray_code (i);
                if (count_bits (cur) == k) {
                        for (int j=0; j<n; ++j)
                                if (cur & (1<<j))
                                        printf ("%d ", j+1);
                        puts ("");
                }
        }
}
```

Стоит заметить, что возможна и в некотором смысле более эффективная реализация, которая будет строить всевозможные сочетания на ходу, и тем самым работать за O (C_n^k n). С другой стороны, эта реализация представляет собой рекурсивную функцию, и поэтому для небольших n, вероятно, она имеет большую скрытую константу, чем предыдущее решение.

Собственно сама реализация - это непосредственное следование формуле:

$$G(N,K) = 0G(N-1,K) \cup 1G(N-1,K-1)^R$$

Эта формула легко получается из приведённой выше формулы для последовательности Грея - мы просто выбираем подпоследовательность из подходящих нам элементов.

```
bool ans[MAXN];

void gen (int n, int k, int l, int r, bool rev) {
        if (k > n || k < 0)  return;
        if (!n) {
                for (int i=0; i<n; ++i)
                        printf ("%d", (int)ans[i]);
                puts ("");
                return;
        }
        ans[rev?r:l] = false;
        gen (n-1, k, !rev?l+1:l, !rev?r:r-1, rev);
        ans[rev?r:l] = true;
        gen (n-1, k-1, !rev?l+1:l, !rev?r:r-1, !rev);
}

void all_combinations (int n, int k) {
        gen (n, k, 0, n-1, false);
}
```

Лемма Бернсайда. Теорема Пойа

Лемма Бернсайда

Эта лемма была сформулирована и доказана **Бернсайдом** (Burnside) в 1897 г., однако было установлено, что эта формула была ранее открыта **Фробениусом** (Frobenius) в 1887 г., а ещё раньше - **Коши** (Cauchy) в 1845 г. Поэтому эта формула иногда называется леммой Бернсайда, а иногда - теоремой Коши-Фробениуса.

Лемма Бернсайда позволяет посчитать количество классов эквивалентности в некотором множестве, основываясь на некоторой его внутренней симметрии.

Объекты и представления

Проведём чёткую грань между количеством объектов и количеством представлений.

Одним и тем же объектам могут соответствовать различные представления, но, разумеется, любое представление соответствует ровно одному объекту. Следовательно, множество всех представлений разбивается на классы эквивалентности. Наша задача — в подсчёте именно числа объектов, или, что то же самое, количества классов эквивалентности.

Пример задачи: раскраска бинарных деревьев

Допустим, мы рассматриваем следующую задачу. Требуется посчитать количество способов раскрасить корневые бинарные деревья с n вершинами в 2 цвета, если у каждой вершины мы не различаем правого и левого сына.

Множество объектов здесь — это множество различных в этом понимании раскрасок деревьев.

Определим теперь множество представлений. Каждой раскраске поставим в соответствие задающую её функцию $f(v)$, где $v = 1 \ldots n$, а $f(v) = 0 \ldots 1$. Тогда множество представлений — это множество различных функций такого вида, и размер его, очевидно, равен 2^n. В то же время, на этом множестве представлений мы ввели разбиение на классы эквивалентности.

Например, пусть $n = 3$, а дерево таково: корень — вершина 1, а вершины 2 и 3 — её сыновья. Тогда следующие функции f_1 и f_2 считаются эквивалентными:

$$\begin{array}{ll} f_1(1) = 0 & f_2(1) = 0 \\ f_1(2) = 1 & f_2(2) = 0 \\ f_1(3) = 0 & f_2(3) = 1 \end{array}$$

Инвариантные перестановки

Почему эти две функции f_1 и f_2 принадлежат одному классу эквивалентности? Интуитивно это понятно — потому что мы можем переставить местами сыновей вершины 1, т.е. вершины 2 и 3, а после такого преобразования функции f_1 и f_2 совпадут. Но формально это означает, что найдётся такая **инвариантная перестановка** π (т.е. которая по условию задачи не меняет сам объект, а только его представление), такая, что:

$$f_2\pi \equiv f_1$$

Итак, исходя из условия задачи, мы можем найти все инвариантные перестановки, т.е. применяя которые мы не не переходим из одного класса эквивалентности в другой. Тогда, чтобы проверить, являются ли две функции f_1 и f_2 эквивалентными (т.е. соответствуют ли они на самом деле одному объекту), надо для каждой инвариантной перестановки π проверить, не выполнится ли условие: $f_2\pi \equiv f_1$ (или, что то же самое, $f_1\pi \equiv f_2$). Если хотя бы для одной перестановки обнаружилось это равенство, то f_1 и f_2 эквивалентны, иначе они не эквивалентны.

Нахождение всех таких инвариантных перестановок, относительно которых наша задача инвариантна — это ключевой шаг для применения как леммы Бернсайда, так и теоремы Пойа. Понятно, что эти инвариантные перестановки зависят от конкретной задачи, и их нахождение — процесс чисто эвристический, основанный на интуитивных соображениях. Впрочем, в большинстве случаев достаточно вручную найти несколько "основных" перестановок, из которых все остальные перестановки могут быть получены их всевозможными произведениями (и эту, исключительно механическую, часть работы можно переложить на компьютер; более подробно это будет рассмотрено ниже на примере конкретной задачи).

Нетрудно понять, что инвариантные перестановки образуют **группу** — поскольку произведение любых инвариантных перестановок тоже является инвариантной перестановкой. Обозначим **группу**

инвариантных перестановок через G.

Формулировка леммы

Для формулировки осталось напомнить одно понятие из алгебры. **Неподвижной точкой** f для перестановки π называется такой элемент, который инвариантен относительно этой перестановки: $f \equiv f\pi$. Например, в нашем примере неподвижными точками будут являться те функции f, которые соответствуют раскраскам, не меняющимся при применении к ним перестановки π (не меняющимся именно в формальном смысле равенства двух функций). Обозначим через $I(\pi)$ **количество неподвижных точек** для перестановки π.

Тогда **лемма Бернсайда** звучит следующим образом: количество классов эквивалетности равно сумме количеств неподвижных точек по всем перестановкам из группы G, делённой на размер этой группы:

$$\text{ClassesCount} = \frac{1}{|G|} \sum_{\pi \in G} I(\pi)$$

Хотя лемма Бернсайда сама по себе не так удобна для применения на практике (пока непонятно, как быстро искать величину $I(\pi)$), она наиболее ясно раскрывает математическую суть, на которой основана идея подсчёта классов эквивалентности.

Доказательство леммы Бернсайда

Описанное здесь доказательство леммы Бернсайда не так важно для её понимания и применения на практике, поэтому его можно пропустить при первом чтении.

Приведённое здесь доказательство является самым простым из известных и не использует теорию групп. Это доказательство было опубликовано Богартом (Bogart) и Кеннетом (Kenneth) в 1991 г.

Итак, нам нужно доказать следующее утверждение:

$$\text{ClassesCount}|G| = \sum_{\pi \in G} I(\pi)$$

Величина, стоящая справа — это не что иное, как количество "инвариантных пар" (f, π), т.е. таких пар, что $f\pi \equiv f$. Очевидно, что в формуле мы имеем право изменить порядок суммирования - сделать внешнюю сумму по элементам f, а внутри неё поставить величину $J(f)$ — количество перестановок, относительно которых f инвариантна:

$$\text{ClassesCount}|G| = \sum_{f} J(f)$$

Для доказательства этой формулы составим таблицу, столбцы которой будут подписаны всеми значениями f_i, строки — всеми перестановками π_j, а в клетках таблицы будут стоять произведения $f_i \pi_j$. Тогда, если мы будем рассматривать столбцы этой таблицы как множества, то некоторые из них могут совпасть, и это будет как означать, что соответствующие этим столбцам f также эквивалентны. Таким образом, количество различных как множество столбцов равно искомой величине ClassesCount. Кстати говоря, с точки зрения теории групп столбец таблицы, подписанный некоторым элементом f_i — это орбита этого элемента; для эквивалентных элементов, очевидно, орбиты совпадают, и число различных орбит даёт именно ClassesCount.

Итак, столбцы таблицы сами распадаются на классы эквивалентности; зафиксируем теперь какой-либо класс и рассмотрим столбцы в нём. Во-первых, заметим, что в этих столбцах могут стоять только элементы f_i одного класса эквивалентности (иначе получилось бы, что некоторым эквивалентным преобразованием π_j мы перешли в другой класс эквивалентности, что невозможно). Во-вторых, каждый элемент f_i будет встречаться одинаковое число раз во всех столбцах (это также следует из того, что столбцы соответствуют эквивалентным элементам). Отсюда можно сделать вывод, что все столбцы внутри одного класса эквивалентности совпадают друг с другом как мультимножества.

Теперь зафиксируем произвольный элемент f. С одной стороны, он встречается в своём столбце ровно $J(f)$ раз (по самому определению $J(f)$). С другой стороны, все столбцы внутри одного класса эквивалентности одинаковы как мультимножества. Следовательно, внутри каждого столбца данного класса эквивалентности любой элемент g встречается ровно $J(g)$ раз.

Таким образом, если мы возьмём произвольным образом от каждого класса эквивалентности по одному столбцу и просуммируем количество элементов в них, то получим, с одной стороны, $\text{ClassesCount}|G|$ (это получается, просто умножив количество столбцов на их размер), а с другой стороны — сумму величин $J(f)$ по всем f (это следует из всех предыдущих рассуждений):

$$\text{ClassesCount}|G| = \sum_{f} J(f)$$

что и требовалось доказать.

Теорема Пойа. Простейший вариант

Теорема **Пойа** (Polya) является обобщением леммы Бернсайда, к тому же предоставляющая более удобный инструмент для нахождения количества классов эквивалентности. Следует отметить, что ещё до Пойа эта теорема была открыта и доказана Редфилдом (Redfield) в 1927 г., однако его публикация прошла незамеченной математиками того времени. Пойа независимо пришёл к тому же результату лишь в 1937 г., и его публикация была более удачной.

Здесь мы рассмотрим формулу, получающуюся как частный случай теоремы Пойа, и которую очень удобно использовать для вычислений на практике. Общая теорема Пойа в данной статье рассматриваться не будет.

Обозначим через $C(\pi)$ количество циклов в перестановке π. Тогда выполняется следующая формула (**частный случай теоремы Пойа**):

$$\text{ClassesCount} = \frac{1}{|G|} \sum_{\pi \in G} k^{C(\pi)}$$

где k — количество значений, которые может принимать каждый элемент представления $f(v)$. Например, в нашей задаче-примере (раскраска корневого бинарного дерева в 2 цвета) $k = 2$.

Доказательство

Эта формула является прямым следствием леммы Бернсайда. Чтобы получить её, нам надо просто найти явное выражение для величины $I(\pi)$, фигурирующую в лемме (напомним, это количество неподвижных точек перестановки π).

Итак, рассмотрим некоторую перестановку π и некоторый элемент f. Под действием перестановки π элементы f передвигаются, как известно, по циклам перестановки. Заметим, что так как в результате должно получаться $f \equiv f\pi$, то внутри каждого цикла перестановки должны находиться одинаковые элементы f. В то же время, для разных циклов никакой связи между значениями элементов не возникает. Таким образом, для каждого цикла перестановки π мы выбираем по одному значению (среди k вариантов), и тем самым мы получим все представления f, инвариантные относительно этой перестановки, т.е.:

$$I(\pi) = k^{C(\pi)}$$

где $C(\pi)$ — количество циклов перестановки.

Пример задачи: Ожерелья

Задача "ожерелья" — это одна из классических комбинаторных задач. Требуется посчитать количество различных ожерелий из n бусинок, каждая из которых может быть покрашена в один из k цветов. При сравнении двух ожерелий их можно поворачивать, но не переворачивать (т.е. разрешается сделать циклический сдвиг).

В этой задаче мы можем сразу найти группу инвариантных перестановок. Очевидно, она будет состоять из n перестановок:

$$\begin{aligned}
\pi_0 &= 1\ 2\ 3\ \ldots\ n \\
\pi_1 &= 2\ 3\ \ldots\ n\ 1 \\
\pi_2 &= 3\ \ldots\ n\ 1\ 2 \\
\pi_{n-1} &= n\ 1\ 2\ \ldots\ (n-1)
\end{aligned}$$

Найдём явную формулу для вычисления $C(\pi_i)$. Во-первых, заметим, что перестановки имеют такой вид, что в i-ой перестановке на j-ой позиции стоит $i + j$ (взятое по модулю n, если оно больше n). Если мы будем рассматривать циклическую структуру i-ой перестановки, то увидим, что единица переходит в $1 + i$, $1 + i$ переходит в $1 + 2i$, $1 + 2i$ — в $1 + 3i$, и т.д., пока не придём в число $1 + kn$; для остальных элементов выполняются похожие утверждения. Отсюда можно понять, что все циклы имеют одинаковую длину, равную $\text{lcm}(i, n)/i$, т.е. $n/\gcd(i, n)$ ("gcd" — наибольший общий делитель, "lcm" — наименьшее общее кратное). Тогда количество циклов в i-ой перестановке будет равно просто $\gcd(i, n)$.

Подставляя найденные значения в теорему Пойа, получаем **решение**:

$$\text{Ans} = \frac{1}{n} \sum_{i=1}^{n} k^{\gcd(i,n)}$$

Можно оставить формулу в таком виде, а можно её свернуть ещё больше. Перейдём от суммы по всем i к сумме только по делителям n. Действительно, в нашей сумме будет много одинаковых слагаемых: если i не является делителем n, то таковой делитель найдётся после вычисления $\gcd(i, n)$. Следовательно, для каждого делителя $d|n$ его

слагаемое $k^{\gcd(d,n)} = k^d$ учтётся несколько раз, т.е. сумму можно представить в таком виде:

$$\text{Ans} = \frac{1}{n} \sum_{d|n} C_d k^d$$

где C_d — это количество таких чисел i, что $\gcd(i,n) = d$. Найдём явное выражение для этого количества.
Любое такое число i имеет вид: $i = dj$, где $\gcd(j, n/d) = 1$ (иначе было бы $\gcd(i,n) > d$).
Вспоминая функцию Эйлера, мы находим, что количество таких j — это величина функции Эйлера $\phi(n/d)$.
Таким образом, $C_d = \phi(n/d)$, и окончательно получаем **формулу**:

$$\text{Ans} = \frac{1}{n} \sum_{d|n} \phi\left(\frac{n}{d}\right) k^d$$

Применение леммы Бернсайда совместно с программными вычислениями

Далеко не всегда удаётся чисто аналитическим путём получить явную формулу для количества классов эквивалентности. Во многих задачах количество перестановок, входящих в группу, может быть слишком большим для ручных вычислений, и вычислить аналитически количество циклов в них не представляется возможным.

В таком случае следует вручную найти несколько "основных" перестановок, которых будет достаточно для порождения всей группы G. Далее можно написать программу, которая сгенерирует все перестановки группы G, посчитает в каждой из них количество циклов и подставит их в формулу.

Рассмотрим для примера **задачу о количестве раскрасок тора**. Имеется прямоугольный клетчатый лист бумаги $n \times m (n < m)$, некоторые из клеток покрашены в чёрный цвет. Затем из этого листа получают цилиндр, склеивая две стороны с длинами m. Затем из цилиндра получают тор, склеивая две окружности (базы цилиндра) без перекручивания. Требуется посчитать количество различных торов (лист был изначально покрашен произвольно), считая, что линии склеивания неразличимы, а тор можно поворачивать и переворачивать.

В данной задаче представлением можно считать лист бумаги $n \times m$, некоторые клетки которого покрашены в чёрный цвет. Нетрудно понять, что следующие виды преобразований сохраняют класс эквивалентности: циклический сдвиг строк листа, циклический сдвиг столбцов листа, поворот листа на 180 градусов; также интуитивно можно понять, что этих трёх видов преобразований достаточно для порождения всей группы инвариантных преобразований. Если мы каким-либо образом занумеруем клетки поля, то мы можем записать три перестановки p_1, p_2, p_3, соответствующие этим видам преобразований. Дальше остаётся только сгенерировать все перестановки, получающиеся как произведения этой. Очевидно, что все такие перестановки имеют вид $p_1^{i_1} p_2^{i_2} p_3^{i_3}$, где $i_1 = 0 \dots m-1, i_2 = 0 \dots n-1, i_3 = 0 \dots 1$.

Таким образом, мы можем написать реализацию решения этой задачи:

```
void mult (vector<int> & a, const vector<int> & b) {
        vector<int> aa (a);
        for (size_t i=0; i<a.size(); ++i)
                a[i] = aa[b[i]];
}

int cnt_cycles (vector<int> a) {
        int res = 0;
        for (size_t i=0; i<a.size(); ++i)
                if (a[i] != -1) {
                        ++res;
                        for (size_t j=i; a[j]!=-1; ) {
                                size_t nj = a[j];
                                a[j] = -1;
                                j = nj;
                        }
                }
        return res;
}

int main() {
        int n, m;
        cin >> n >> m;

        vector<int> p (n*m),  p1 (n*m),  p2 (n*m),  p3 (n*m);
        for (int i=0; i<n*m; ++i) {
```

```
		p[i] = i;
		p1[i] = (i % n + 1) % n + i / n * n;
		p2[i] = (i / n + 1) % m * n + i % n;
		p3[i] = (m - 1 - i / n) * n + (n - 1 - i % n);
	}

	int sum = 0,  cnt = 0;
	set < vector<int> > s;
	for (int i1=0; i1<n; ++i1) {
		for (int i2=0; i2<m; ++i2) {
			for (int i3=0; i3<2; ++i3) {
				if (!s.count(p)) {
					s.insert (p);
					++cnt;
					sum += 1 << cnt_cycles(p);
				}
				mult (p, p3);
			}
			mult (p, p2);
		}
		mult (p, p1);
	}

	cout << sum / cnt;
}
```

Принцип включений-исключений

Принцип включений-исключений — это важный комбинаторный приём, позволяющий подсчитывать размер каких-либо множеств, или вычислять вероятность сложных событий.

Формулировки принципа включений-исключений

Словесная формулировка

Принцип включений-исключений выглядит следующим образом:

Чтобы посчитать размер объединения нескольких множеств, надо просуммировать размеры этих множеств **по отдельности**, затем вычесть размеры всех **попарных** пересечений этих множеств, прибавить обратно размеры пересечений всевозможных **троек** множеств, вычесть размеры пересечений **четвёрок**, и так далее, вплоть до пересечения **всех** множеств.

Формулировка в терминах множеств

В математической форме приведённая выше словесная формулировка выглядит следующим образом:

$$\left|\bigcup_{i=1}^{n} A_i\right| = \sum_{i=1}^{n} |A_i| \; - \sum_{\substack{i,j:\\ 1\le i<j\le n}} |A_i \cap A_j| \; + \sum_{\substack{i,j,k:\\ 1\le i<j<k\le n}} |A_i \cap A_j \cap A_k| \; - \ldots + (-1)^{n-1} |A_1 \cap \ldots \cap A_n|.$$

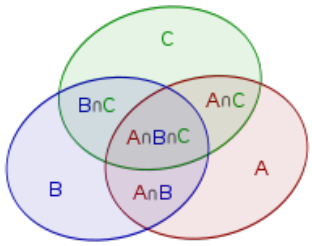
Её можно записать более компактно, через сумму по подмножествам. Обозначим через B множество, элементами которого являются A_i. Тогда принцип включений-исключений принимает вид:

$$\left|\bigcup_{i=1}^{n} A_i\right| = \sum_{C \subseteq B} (-1)^{size(C)-1} \left|\bigcap_{e \in C} e\right|.$$

Эту формулу приписывают Муавру (Abraham de Moivre).

Формулировка с помощью диаграмм Венна

Пусть на диаграмме отмечены три фигуры A, B и C:



Тогда площадь объединения $A \cup B \cup C$ равна сумме площадей A, B и C за вычетом дважды покрытых площадей $A \cap B$, $A \cap C$, $B \cap C$, но с прибавлением трижды покрытой площади $A \cap B \cap C$:

$$S(A \cup B \cup C) = S(A) + S(B) + S(C) - S(A \cap B) - S(A \cap C) - S(B \cap C) + S(A \cap B \cap C).$$

Аналогичным образом это обобщается и на объединение n фигур.

Формулировка в терминах теории вероятностей

Если $A_i (i = 1 \ldots n)$ — это события, $\mathcal{P}(A_i)$ — их вероятности, то вероятность их объединения (т.е. того, что произойдёт хотя бы одно из этих событий) равна:

$$\mathcal{P}\left(\bigcup_{i=1}^{n} A_i\right) = \sum_{i=1}^{n} \mathcal{P}(A_i) - \sum_{\substack{i,j:\\ 1\le i<j\le n}} \mathcal{P}(A_i \cap A_j) + \\ + \sum_{\substack{i,j,k:\\ 1\le i<j<k\le n}} \mathcal{P}(A_i \cap A_j \cap A_k) - \ldots + (-1)^{n-1}\mathcal{P}(A_1 \cap \ldots \cap A_n).$$

Эту сумму также можно записать в виде суммы по подмножествам множества B, элементами которого являются события A_i:

$$\mathcal{P}\left(\bigcup_{i=1}^{n} A_i\right) = \sum_{C \subseteq B} (-1)^{size(C)-1} \cdot \mathcal{P}\left(\bigcap_{e \in C} e\right).$$

Доказательство принципа включений-исключений

Для доказательства удобно пользоваться математической формулировкой в терминах теории множеств:

$$\left|\bigcup_{i=1}^{n} A_i\right| = \sum_{C \subseteq B} (-1)^{size(C)-1} \left|\bigcap_{e \in C} e\right|,$$

где B, напомним, — это множество, состоящее из A_i-ых.

Нам нужно доказать, что любой элемент, содержащийся хотя бы в одном из множеств A_i, учтётся формулой ровно один раз. (Заметим, что остальные элементы, не содержащиеся ни в одном из A_i, никак не могут быть учтены, поскольку отсутствуют в правой части формулы).

Рассмотрим произвольный элемент x, содержащийся ровно в $k \ge 1$ множествах A_i. Покажем, что он посчитается формулой ровно один раз.

Заметим, что:

- в тех слагаемых, у которых $size(C) = 1$, элемент x учтётся ровно k раз, со знаком плюс;
- в тех слагаемых, у которых $size(C) = 2$, элемент x учтётся (со знаком минус) ровно C_k^2 раз — потому что x посчитается только в тех слагаемых, которые соответствуют двум множествам из k множеств, содержащих x;
- в тех слагаемых, у которых $size(C) = 3$, элемент x учтётся ровно C_k^3 раз, со знаком плюс;
- $\ldots$
- в тех слагаемых, у которых $size(C) = k$, элемент x учтётся ровно C_k^k раз, со знаком $(-1)^{k-1}$;
- в тех слагаемых, у которых $size(C) > k$, элемент x учтётся ноль раз.

Таким образом, нам надо посчитать такую сумму биномиальных коэффициентов:

$$T = C_k^1 - C_k^2 + C_k^3 - \ldots + (-1)^{i-1} \cdot C_k^i + \ldots + (-1)^{k-1} \cdot C_k^k.$$

Проще всего посчитать эту сумму, сравнив её с разложением в бином Ньютона выражения $(1-x)^k$:

$$(1-x)^k = C_k^0 - C_k^1 \cdot x + C_k^2 \cdot x^2 - C_k^3 \cdot x^3 + \ldots + (-1)^k \cdot C_k^k \cdot x^k.$$

Видно, что при $x = 1$ выражение $(1-x)^k$ представляет собой не что иное, как $1 - T$.
Следовательно, $T = 1 - (1-1)^k = 1$, что и требовалось доказать.

Применения при решении задач

Принцип включений-исключений сложно хорошо понять без изучения примеров его применений.

Сначала мы рассмотрим три простые задачи "на бумажке", иллюстрирующие применение принципа, затем рассмотрим более практические задачи, которые трудно решить без использования принципа включений-исключений.

Особо следует отметить задачу "поиск числа путей", поскольку в ней демонстрируется, что принцип включений-исключений может иногда приводить к полиномиальным решениям, а не обязательно экспоненциальным.

Простая задачка о перестановках

Сколько есть перестановок чисел от 0 до 9 таких, что первый элемент больше 1, а последний — меньше 8?

Посчитаем число "плохих" перестановок, т.е. таких, у которых первый элемент ≤ 1 и/или последний ≥ 8.

Обозначим через X множество перестановок, у которых первый элемент ≤ 1, а через Y — у которых последний элемент ≥ 8. Тогда количество "плохих" перестановок по формуле включений-исключений равно:

$$|X| + |Y| - |X \cap Y|.$$

Проведя несложные комбинаторные вычисления, получаем, что это равно:

$$2 \cdot 9! + 2 \cdot 9! - 2 \cdot 2 \cdot 8!$$

Отнимая это число от общего числа перестановок $10!$, мы получим ответ.

Простая задачка о (0,1,2)-последовательностях

Сколько существует последовательностей длины n, состоящих только из чисел $0, 1, 2$, причём каждое число встречается хотя бы раз?

Снова перейдём к обратной задаче, т.е. будем считать число последовательностей, в которых не присутствует хотя бы одно из чисел.

Обозначим через $A_i (i = 0 \ldots 2)$ множество последовательностей, в которых не встречается число i. Тогда по формуле включений-исключений число "плохих" последовательностей равно:

$$|A_0| + |A_1| + |A_2| - |A_0 \cap A_1| - |A_0 \cap A_2| - |A_1 \cap A_2| + |A_0 \cap A_1 \cap A_2|.$$

Размеры каждого из A_i равны, очевидно, 2^n (поскольку в таких последовательностях могут встречаться только два вида цифр). Мощности каждого попарного пересечения $A_i \cap A_j$ равны 1 (поскольку остаётся доступной только одна цифра). Наконец, мощность пересечения всех трёх множеств равна 0 (поскольку доступных цифр вообще не остаётся).

Вспоминая, что мы решали обратную задачу, получаем итоговый **ответ**:

$$3^n - 3 \cdot 2^n + 3 \cdot 1 - 0.$$

Количество целочисленных решений уравнения

Дано уравнение:

$$x_1 + x_2 + x_3 + x_4 + x_5 + x_6 = 20,$$

где все $0 \leq x_i \leq 8$ (где $i = 1 \ldots 6$).

Требуется посчитать число решений этого уравнения.

Забудем сначала про ограничение $x_i \leq 8$, и просто посчитаем число неотрицательных решений этого уравнения. Это легко делается через биномиальные коэффициенты — мы хотим разбить 20 элементов на 6 групп, т.е. распределить 5 "стенок", разделяющих группы, по 25 местам:

$$N_0 = C_{25}^5$$

Посчитаем теперь по формуле включений-исключений число "плохих" решений, т.е. таких решений уравнения, в которых один или более x_i больше 9.

Обозначим через A_k (где $k = 1 \ldots 6$) множество таких решений уравнения, в которых $x_k \geq 9$, а все остальные $x_i \geq 0$ (для всех $i \neq k$). Чтобы посчитать размер множества A_k, заметим, что у нас по сути та же комбинаторная задача, что решалась двумя абзацами выше, только теперь 9 элементов исключены из рассмотрения и точно принадлежат первой группе. Таким образом:

$$|A_k| = C_{16}^5$$

Аналогично, мощность пересечения двух множеств A_k и A_p равна числу:

$$|A_k \cap A_p| = C_7^5$$

Мощность каждого пересечения трёх и более множеств равна нулю, поскольку 20 элементов не хватит на три и более переменных, больше либо равных 9.

Объединяя всё это в формулу включений-исключений и учитывая, что мы решали обратную задачу, окончательно получаем **ответ**:

$$C_{25}^5 - C_6^1 \cdot C_{16}^5 + C_6^2 \cdot C_7^5.$$

Количество взаимно простых чисел в заданном отрезке

Пусть даны числа n и r. Требуется посчитать количество чисел в отрезке $[1; r]$, взаимно простых с n.

Сразу перейдём к обратной задаче — посчитаем количество не взаимно простых чисел.

Рассмотрим все простые делители числа n; обозначим их через $p_i (i = 1 \dots k)$.

Сколько чисел в отрезке $[1; r]$, делящихся на p_i? Их количество равно:

$$\left\lfloor \frac{r}{p_i} \right\rfloor$$

Однако если мы просто просуммируем эти числа, то получим неправильный ответ — некоторые числа будут просуммированы несколько раз (те, которые делятся сразу на несколько p_i). Поэтому надо воспользоваться формулой включений-исключений.

Например, можно за 2^k перебрать подмножество множества всех p_i-ых, посчитать их произведение, и прибавить или вычесть в формуле включений-исключений очередное слагаемое.

Итоговая **реализация** для подсчёта количества взаимно простых чисел:

```
int solve (int n, int r) {
        vector<int> p;
        for (int i=2; i*i<=n; ++i)
                if (n % i == 0) {
                        p.push_back (i);
                        while (n % i == 0)
                                n /= i;
                }
        if (n > 1)
                p.push_back (n);

        int sum = 0;
        for (int msk=1; msk<(1<<p.size()); ++msk) {
                int mult = 1,
                        bits = 0;
                for (int i=0; i<(int)p.size(); ++i)
                        if (msk & (1<<i)) {
                                ++bits;
                                mult *= p[i];
                        }

                int cur = r / mult;
                if (bits % 2 == 1)
                        sum += cur;
                else
                        sum -= cur;
        }

        return r - sum;
}
```

Асимптотика решения составляет $O(\sqrt{n})$.

Количество чисел в заданном отрезке, кратных хотя бы одному из заданных чисел

Даны n чисел a_i и число r. Требуется посчитать количество чисел в отрезке $[1; r]$, которые кратны хотя бы одному из a_i.

Алгоритм решения практически совпадает с предыдущей задачей — делаем формулу включений-исключений над числами a_i, т.е. каждое слагаемое в этой формуле — это количество чисел, делящихся на заданный поднабор чисел a_i (иными словами, делящихся на их наименьшее общее кратное).

Таким образом, решение сводится к тому, чтобы за 2^n перебрать поднабор чисел, за $O(n \log r)$ операций найти их наименьшее общее кратное, и прибавить или вычесть из ответа очередное значение.

Количество строк, удовлетворяющих заданному числу паттернов

Дано n паттернов — строк одинаковой длины, состоящих только из букв и знаков вопроса. Также дано число k.

Требуется посчитать количество строк, удовлетворяющих ровно k паттернам либо, в другой постановке, как минимум k паттернам.

Заметим вначале, что мы можем **легко посчитать число строк**, удовлетворяющих сразу всем указанным паттернам. Для этого надо просто "пересечь" эти паттерны: посмотреть на первый символ (во всех ли паттернах на первой позиции стоит вопрос, или не во всех — тогда первый символ определён однозначно), на второй символ, и т.д.

Научимся теперь решать **первый вариант задачи**: когда искомые строки должны удовлетворять ровно k паттернам.

Для этого переберём и зафксируем конкретное подмножество X паттернов размера k — теперь мы должны посчитать количество строк, удовлетворяющих этому набору паттернов и только ему. Для этого воспользуемся формулой включений-исключений: мы суммируем по всем надмножествам множества X, и либо прибавляем к текущему ответу, либо отнимаем от него количество строк, подходящих под текущее множество:

$$ans(X) = \sum_{Y \supseteq X} (-1)^{|Y|-k} \cdot f(Y),$$

где $f(Y)$ обозначает количество строк, подходящих под набор паттернов Y.

Если мы просуммируем $ans(X)$ по всем X, то получим ответ:

$$ans = \sum_{X \ : \ |X|=k} ans(X).$$

Однако тем самым мы получили решение за время порядка $O(3^k \cdot k)$.

Решение можно ускорить, заметив, что в разных $ans(X)$ суммирование зачастую ведётся по одним и тем же множествам Y.

Перевернём формулу включений-исключений и будем вести суммирование по Y. Тогда легко понять, что множество Y учтётся в $C_{|Y|}^k$ формулах включений-исключений, везде с одним и тем же знаком $(-1)^{|Y|-k}$:

$$ans = \sum_{Y \ : \ |Y| \geq k} (-1)^{|Y|-k} \cdot C_{|Y|}^k \cdot f(Y).$$

Решение получилось с асимптотикой $O(2^k \cdot k)$.

Перейдём теперь ко **второму варианту задачи**: когда искомые строки должны удовлетворять как минимум k паттернам.

Понятно, мы можем просто воспользоваться решением первого варианта задачи и просуммировать ответы от k до n. Однако можно заметить, что все рассуждения по-прежнему будут верны, только в этом варианте задачи сумма по X идёт не только по тем множествам, размер которых равен k, а по всем множествам с размером $\geq k$.

Таким образом, в итоговой формуле перед $f(Y)$ будет стоять другой коэффициент: не один биномиальный коэффициент с каким-то знаком, а их сумма:

$$(-1)^{|Y|-k} \cdot C_{|Y|}^k \ + \ (-1)^{|Y|-k-1} \cdot C_{|Y|}^{k+1} \ + \ (-1)^{|Y|-k-2} \cdot C_{|Y|}^{k+2} \ + \ \ldots \ + \ (-1)^{|Y|-|Y|} \cdot C_{|Y|}^{|Y|}.$$

Заглянув в Грэхема (Грэхем, Кнут, Паташник. "**Конкретная математика**" [1998]), мы видим такую известную формулу для биномиальных коэффициентов:

$$\sum_{k=0}^{m} (-1)^k \cdot C_n^k = (-1)^m \cdot C_{n-1}^m.$$

Применяя её здесь, получаем, что вся эта сумма биномиальных коэффициентов сворачивается в:

$$(-1)^{|Y|-k} \cdot C_{|Y|-1}^{|Y|-k}.$$

Таким образом, для этого варианта задачи мы также получили решение с асимптотикой $O(2^k \cdot k)$:

$$ans = \sum_{Y \ : \ |Y| \geq k} (-1)^{|Y|-k} \cdot C_{|Y|-1}^{|Y|-k} \cdot f(Y).$$

Количество путей

Есть поле $n \times m$, некоторые k клеток которого — непроходимые стенки. На поле в клетке $(1, 1)$ (левая нижняя

клетка) изначально находится робот. Робот может двигаться только вправо или вверх, и в итоге он должен попасть в клетку (n, m), избежав все препятствия. Требуется посчитать число путей, которыми он может это сделать.

Предполагаем, что размеры n и m очень большие (скажем, до 10^9), а количество k — небольшое (порядка 100).

Для решения сразу в целях удобства **отсортируем** препятствия в том порядке, в каком мы можем их обойти: т. е., например, по координате x, а при равенстве — по координате y.

Также сразу научимся решать задачу без препятствий: т.е. научимся считать число способов дойти от одной клетки до другой. Если по одной координате нам надо пройти x клеток, а по другой — y клеток, то из несложной комбинаторики мы получаем такую формулу через биномиальные коэффициенты:

$$C_{x+y}^{x}$$

Теперь чтобы посчитать число способов дойти от одной клетки до другой, избежав всех препятствий, можно воспользоваться **формулой включений-исключений**: посчитаем число способов дойти, наступив хотя бы на одно препятствие.

Для этого можно, например, перебрать подмножество тех препятствий, на которые мы точно наступим, посчитать число способов сделать это (просто перемножив число способов дойти от стартовой клетки до первого из выбранных препятствий, от первого препятствия до второго, и так далее), и затем прибавить или отнять это число от ответа, в соответствии со стандартной формулой включений-исключений.

Однако это снова будет неполиномиальное решение — за асимптотику $O(2^k k)$. Покажем, как получить **полиномиальное решение**.

Решать будем **динамическим программированием**: научимся вычислять числа $d[i][j]$ — число способов дойти от i-ой точки до j-ой, не наступив при этом ни на одно препятствие (кроме самих i и j, естественно). Всего у нас будет $k + 2$ точки, поскольку к препятствиям добавляются стартовая и конечная клетки.

Если мы на секунду забудем про все препятствия и просто посчитаем число путей из клетки i в клетку j, то тем самым мы учтём некоторые "плохие" пути, проходящие через препятствия. Научимся считать количество этих "плохих" путей. Переберём первое из препятствий $i < t < j$, на которое мы наступим, тогда количество путей будет равно $d[i][t]$, умноженному на число произвольных путей из t в j. Просуммировав это по всем t, мы посчитаем количество "плохих" путей.

Таким образом, значение $d[i][j]$ мы научились считать за время $O(k)$. Следовательно, решение всей задачи имеет асимптотику $O(k^3)$.

Число взаимно простых четвёрок

Дано n чисел: $a_1, a_2, \ldots, a_n$. Требуется посчитать количество способов выбрать из них четыре числа так, что их совокупный наибольший общий делитель равен единице.

Будем решать обратную задачу — посчитаем число "плохих" четвёрок, т.е. таких четвёрок, в которых все числа делятся на число $d > 1$.

Воспользуемся формулой включений-исключений, суммируя количество четвёрок, делящихся на делитель d (но, возможно, делящихся и на больший делитель):

$$ans = \sum_{d \geq 2} (-1)^{deg(d)-1} \cdot f(d),$$

где $deg(d)$ — это количество простых в факторизации числа d, $f(d)$ — количество четвёрок, делящихся на d.

Чтобы посчитать функцию $f(d)$, надо просто посчитать количество чисел, кратных d, и биномиальным коэффициентом посчитать число способов выбрать из них четвёрку.

Таким образом, с помощью формулы включений-исключений мы суммируем количество четвёрок, делящихся на простые числа, затем отнимаем число четвёрок, делящихся на произведение двух простых, прибавляем четвёрки, делящиеся на три простых, и т.д.

Число гармонических троек

Дано число $n \leq 10^6$. Требуется посчитать число таких троек чисел $2 \leq a < b < c \leq n$, что они являются гармоническими тройками, т.е.:

- либо $\gcd(a, b) = \gcd(a, c) = \gcd(b, c) = 1$,
- либо $\gcd(a, b) > 1, \gcd(a, c) > 1, \gcd(b, c) > 1$.

Во-первых, сразу перейдём к обратной задаче — т.е. посчитаем число негармонических троек.

Во-вторых, заметим, что в любой негармонической тройке ровно два её числа находятся в такой ситуации, что это число взаимно просто с одним числом тройки и не взаимно просто с другим числом тройки.

Таким образом, количество негармонических троек равно сумме по всем числам от 2 до n произведений количества взаимно простых с текущим числом чисел на количество не взаимно простых чисел.

Теперь всё, что нам осталось для решения задачи — это научиться считать для каждого числа в отрезке $[2; n]$ количество чисел, взаимно простых (или не взаимно простых) с ним. Хотя эта задача уже рассматривалась нами выше, описанное выше решение не подходит здесь — оно потребует факторизации каждого из чисел от 2 до n, и затем перебора всевозможных произведений простых чисел из факторизации.

Поэтому нам понадобится более быстрое решение, которое подсчитывает ответы для всех чисел из отрезка $[2; n]$ сразу.

Для этого можно реализовать такую **модификацию решета Эратосфена**:

- Во-первых, нам надо найти все числа в отрезке $[2; n]$, в факторизации которых никакое простое не входит дважды. Кроме того, для формулы включений-исключений нам потребуется знать, сколько простых содержит факторизация каждого такого числа.

  Для этого нам надо завести массивы $deg[]$, хранящие для каждого числа количество простых в его факторизации, и $good[]$ — содержащий для каждого числа $true$ или $false$ — все простые входят в него в степени ≤ 1 или нет.

  После этого во время решета Эратосфена при обработке очередного простого числа мы пройдёмся по всем числам, кратным текущему числу, и увеличим $deg[]$ у них, а у всех чисел, кратных квадрату от текущего простого — поставим $good = false$.

- Во-вторых, нам надо посчитать ответ для всех чисел от 2 до n, т.е. массив $cnt[]$ — количество чисел, не взаимно простых с данным.

  Для этого вспомним, как работает формула включений-исключений — здесь фактически мы реализуем её же, но с перевёрнутой логикой: мы словно перебираем слагаемое и смотрим, в какие формулы включений-исключений для каких чисел это слагаемое входит.

  Итак, пусть у нас есть число i, для которого $good[] = true$, т.е. это число, участвующее в формуле включений-исключений. Переберём все числа, кратные i, и к ответу $cnt[]$ каждого из таких чисел мы должны прибавить или вычесть величину $\lfloor N/i \rfloor$. Знак — прибавление или вычитание — зависит от $deg[i]$: если $deg[i]$ нечётна, то надо прибавлять, иначе вычитать.

Реализация:

```
int n;
bool good[MAXN];
int deg[MAXN], cnt[MAXN];

long long solve() {
        memset (good, 1, sizeof good);
        memset (deg, 0, sizeof deg);
        memset (cnt, 0, sizeof cnt);

        long long ans_bad = 0;
        for (int i=2; i<=n; ++i) {
                if (good[i]) {
                        if (deg[i] == 0)  deg[i] = 1;
                        for (int j=1; i*j<=n; ++j) {
                                if (j > 1 && deg[i] == 1)
                                        if (j % i == 0)
                                                good[i*j] = false;
                                        else
                                                ++deg[i*j];
                                cnt[i*j] += (n / i) * (deg[i]%2==1 ? +1 : -1);
                        }
                }
                ans_bad += (cnt[i] - 1) * 1ll * (n-1 - cnt[i]);
        }

        return (n-1) * 1ll * (n-2) * (n-3) / 6 - ans_bad / 2;
}
```

Асимптотика такого решения составляет $O(n \log n)$, поскольку почти для каждого числа i оно совершает примерно n/i итераций вложенного цикла.

Число перестановок без неподвижных точек

Докажем, что число перестановок длины n без неподвижных точек равно следующему числу:

$$n! - C_n^1 \cdot (n-1)! + C_n^2 \cdot (n-2)! - C_n^3 \cdot (n-3)! + \ldots \pm C_n^n \cdot (n-n)!$$

и приблизительно равно числу:

$$\frac{n!}{e}$$

(более того, если округлить это выражение к ближайшему целому — то получится в точности число перестановок без неподвижных точек)

Обозначим через A_k множество перестановок длины n с неподвижной точкой в позиции k $(1 \le k \le n)$.

Воспользуемся теперь формулой включений-исключений, чтобы посчитать число перестановок хотя бы с одной неподвижной точкой. Для этого нам надо научиться считать размеры множеств-пересечений A_i, они выглядят следующим образом:

$$|A_p| = (n-1)!\ ,$$
$$|A_p \cap A_q| = (n-2)!\ ,$$
$$|A_p \cap A_q \cap A_r| = (n-3)!\ ,$$
$$\ldots$$

поскольку если мы знаем, что число неподвижных точек равно x, то тем самым мы знаем позицию x элементов перестановки, а все остальные $(n-x)$ элементов могут стоять где угодно.

Подставляя это в формулу включений-исключений и учитывая, что число способов выбрать подмножество размера x из n-элементного множества равно C_n^x, получаем формулу для числа перестановок хотя бы с одной неподвижной точкой:

$$C_n^1 \cdot (n-1)! - C_n^2 \cdot (n-2)! + C_n^3 \cdot (n-3)! - \ldots \pm C_n^n \cdot (n-n)!$$

Тогда число перестановок без неподвижных точек равно:

$$n! - C_n^1 \cdot (n-1)! + C_n^2 \cdot (n-2)! - C_n^3 \cdot (n-3)! + \ldots \pm C_n^n \cdot (n-n)!$$

Упрощая это выражение, получаем **точное и приблизительное выражения для количества перестановок без неподвижных точек**:

$$n!\left(1 - \frac{1}{1!} + \frac{1}{2!} - \frac{1}{3!} + \ldots \pm \frac{1}{n!}\right) \approx \frac{n!}{e}.$$

(поскольку сумма в скобках — это первые $n+1$ членов разложения в ряд Тейлора e^{-1})

В заключение стоит отметить, что аналогичным образом решается задача, когда требуется, чтобы неподвижных точек не было среди m первых элементов перестановок (а не среди всех, как мы только что решали). Формула получится такая, как приведённая выше точная формула, только в ней сумма будет идти до k, а не до n.

Задачи в online judges

Список задач, которые можно решить, используя принцип включений-исключений:

- UVA #10325 "The Lottery" [сложность: низкая]
- UVA #11806 "Cheerleaders" [сложность: низкая]
- TopCoder SRM 477 "CarelessSecretary" [сложность: низкая]
- TopCoder TCHS 16 "Divisibility" [сложность: низкая]
- SPOJ #6285 NGM2 "Another Game With Numbers" [сложность: низкая]
- TopCoder SRM 382 "CharmingTicketsEasy" [сложность: средняя]
- TopCoder SRM 390 "SetOfPatterns" [сложность: средняя]
- TopCoder SRM 176 "Deranged" [сложность: средняя]
- TopCoder SRM 457 "TheHexagonsDivOne" [сложность: средняя]
- SPOJ #4191 MSKYCODE "Sky Code" [сложность: средняя]
- SPOJ #4168 SQFREE "Square-free integers" [сложность: средняя]
- CodeChef "Count Relations" [сложность: средняя]

Литература

- Debra K. Borkovitz. "Derangements and the Inclusion-Exclusion Principle"

Игры на произвольных графах

Пусть игра ведётся двумя игроками на некотором графе G. Т.е. текущее состояние игры - это некоторая вершина графа, и из каждой вершины рёбра идут в те вершины, в которые можно пойти следующим ходом.

Мы рассматриваем самый общий случай - случай произвольного ориентированного графа с циклами. Требуется для заданной начальной позиции определить, кто выиграет при оптимальной игре обоих игроков (или определить, что результатом будет ничья).

Мы решим эту задачу очень эффективно - найдём ответы для всех вершин графа за линейное относительно количества рёбер время - $O(M)$.

Описание алгоритма

Про некоторые вершины графа заранее известно, что они являются выигрышными или проигрышными; очевидно, такие вершины не имеют исходящих рёбер.

Имеем следующие **факты**:

- если из некоторой вершины есть ребро в проигрышную вершину, то эта вершина выигрышная;
- если из некоторой вершины все рёбра исходят в выигрышные вершины, то эта вершина проигрышная;
- если в какой-то момент ещё остались неопределённые вершины, но ни одна из них не подходят ни под первое, ни под второе правило, то все эти вершины - ничейные.

Таким образом, уже ясен алгоритм, работающий за асимптотику O (N M) - мы перебираем все вершины, пытаемся к каждой применить первое либо второе правило, и если мы произвели какие-то изменения, то повторяем всё заново.

Однако этот процесс поиска и обновления можно значительно ускорить, доведя асимптотику до линейной.

Переберём все вершины, про которые изначально известно, что они выигрышные или проигрышные. Из каждой из них пустим следующий поиск в глубину. Этот поиск в глубину будет двигаться по обратным рёбрам. Прежде всего, он не будет заходить в вершины, которые уже определены как выигрышные или проигрышные. Далее, если поиск в глубину пытается пойти из проигрышной вершины в некоторую вершину, то её он помечает как выигрышную, и идёт в неё. Если же поиск в глубину пытается пойти из выигрышной вершины в некоторую вершину, то он должен проверить, все ли рёбра ведут из этой вершины в выигрышные. Эту проверку легко осуществить за O (1), если в каждой вершине будем хранить счётчик рёбер, которые ведут в выигрышные вершины. Итак, если поиск в глубину пытается пойти из выигрышной вершины в некоторую вершину, то он увеличивает в ней счётчик, и если счётчик сравнялся с количеством рёбер, исходящих из этой вершины, то эта вершина помечается как проигрышная, и поиск в глубину идёт в эту вершину. Иначе же, если целевая вершина так и не определена как выигрышная или проигрышная, то поиск в глубину в неё не заходит.

Итого, мы получаем, что каждая выигрышная и каждая проигрышная вершина посещается нашим алгоритмом ровно один раз, а ничейные вершины и вовсе не посещаются. Следовательно, асимптотика действительно $O(M)$.

Реализация

Рассмотрим реализацию поиска в глубину, в предположении, что граф игры построен в памяти, степени исхода посчитаны и записаны в degree (это будет как раз счётчиком, он будет уменьшаться, если есть ребро в выигрышную вершину), а также изначально выигрышные или проигрышные вершины уже помечены.

```
vector<int> g [100];
bool win [100];
bool loose [100];
bool used[100];
int degree[100];

void dfs (int v) {
        used[v] = true;
        for (vector<int>::iterator i = g[v].begin(); i != g[v].end(); ++i)
                if (!used[*i]) {
                        if (loose[v])
                                win[*i] = true;
                        else if (--degree[*i] == 0)
                                loose[*i] = true;
                        else
                                continue;
                        dfs (*i);
                }
}
```

Пример задачи. "Полицейский и вор"

Чтобы алгоритм стал более ясным, рассмотрим его на конкретном примере.

Условие задачи. Имеется поле размером MxN клеток, в некоторые клетки заходить нельзя. Известны начальные координаты полицейского и вора. Также на карте может присутствовать выход. Если полицейский окажется в одной клетке с вором, то выиграл полицейский. Если же вор окажется в клетке с выходом (и в этой клетке не стоит полицейский), то выиграет вор. Полицейский может ходить в 8 направлениях, вор - только в 4 (вдоль осей координат). И полицейский, и вор могут пропустить свой ход. Первым ход делает полицейский.

Построение графа. Построим граф игры. Мы должны формализовать правила игры. Текущее состояние игры определяется координатами полицейского P, вора T, а также булева переменная Pstep, которая определяет, кто будет делать следующий ход. Следовательно, вершина графа определена тройкой (P,T,Pstep). Граф построить легко, просто соответствуя условию.

Далее нужно определить, какие вершины являются выигрышными или проигрышными изначально. Здесь есть **тонкий момент**. Выигрышность/проигрышность вершины помимо координат зависит и от Pstep - чей сейчас ход. Если сейчас ход полицейского, то вершина выигрышная, если координаты полицейского и вора совпадают; вершина проигрышная, если она не выигрышная и вор находится на выходе. Если же сейчас ход вора, то вершина выигрышная, если вор находится на выходе, и проигрышная, если она не выигрышная и координаты полицейского и вора совпадают.

Единственный момент, которой нужно решить - строить **граф явно или** делать это "**на ходу**", прямо в поиске в глубину. С одной стороны, если строить граф предварительно, то будет меньше вероятность ошибиться. С другой стороны, это увеличит объём кода, да и время работы будет в несколько раз медленнее, чем если строить граф "на ходу".

Реализация всей программы:

```
struct state {
        char p, t;
        bool pstep;
};

vector<state> g [100][100][2];
// 1 = policeman coords; 2 = thief coords; 3 = 1 if policeman's step or 0
if thief's.
bool win [100][100][2];
bool loose [100][100][2];
bool used[100][100][2];
int degree[100][100][2];

void dfs (char p, char t, bool pstep) {
        used[p][t][pstep] = true;
        for (vector<state>::iterator i = g[p][t][pstep].begin(); i != g[p]
[t][pstep].end(); ++i)
                if (!used[i->p][i->t][i->pstep]) {
                        if (loose[p][t][pstep])
                                win[i->p][i->t][i->pstep] = true;
                        else if (--degree[i->p][i->t][i->pstep] == 0)
                                loose[i->p][i->t][i->pstep] = true;
                        else
                                continue;
                        dfs (i->p, i->t, i->pstep);
                }
}


int main() {

        int n, m;
        cin >> n >> m;
        vector<string> a (n);
        for (int i=0; i<n; ++i)
                cin >> a[i];

        for (int p=0; p<n*m; ++p)
                for (int t=0; t<n*m; ++t)
                        for (char pstep=0; pstep<=1; ++pstep) {
                                int px = p/m, py = p%m, tx=t/m, ty=t%m;
                                if (a[px][py]=='*' || a[tx][ty]=='*')  continue;
```

```
                                bool & wwin = win[p][t][pstep];
                                bool & lloose = loose[p][t][pstep];
                                if (pstep)
                                        wwin = px==tx && py==ty,   lloose
= !wwin && a[tx][ty] == 'E';
                                else
                                        wwin = a[tx][ty] == 'E',   lloose
= !wwin && px==tx && py==ty;
                                if (wwin || lloose)  continue;

                                state st = { p, t, !pstep };
                                g[p][t][pstep].push_back (st);
                                st.pstep = pstep != 0;
                                degree[p][t][pstep] = 1;

                                const int dx[] = { -1, 0, 1, 0,   -1, -1,
1, 1 };
                                const int dy[] = { 0, 1, 0, -1,   -1, 1, -
1, 1 };
                                for (int d=0; d<(pstep?8:4); ++d) {
                                        int ppx=px, ppy=py, ttx=tx, tty=ty;
                                        if (pstep)
                                                ppx += dx[d],  ppy += dy[d];
                                        else
                                                ttx += dx[d],  tty += dy[d];
                                        if (ppx>=0 && ppx<n && ppy>=0 &&
ppy<m && a[ppx][ppy]!='*' &&
                                                ttx>=0 && ttx<n && tty>=0
&& tty<m && a[ttx][tty]!='*')
                                        {
                                                g[ppx*m+ppy][ttx*m+tty][!
pstep].push_back (st);
                                                ++degree[p][t][pstep];
                                        }
                                }
                        }

        for (int p=0; p<n*m; ++p)
                for (int t=0; t<n*m; ++t)
                        for (char pstep=0; pstep<=1; ++pstep)
                                if ((win[p][t][pstep] || loose[p][t]
[pstep]) && !used[p][t][pstep])
                                        dfs (p, t, pstep!=0);

        int p_st, t_st;
        for (int i=0; i<n; ++i)
                for (int j=0; j<m; ++j)
                        if (a[i][j] == 'C')
                                p_st = i*m+j;
                        else if (a[i][j] == 'T')
                                t_st = i*m+j;

        cout << (win[p_st][t_st][true] ? "WIN" : loose[p_st][t_st]
[true] ? "LOSS" : "DRAW");

}
```

Теория Шпрага-Гранди. Ним

Введение

Теория Шпрага-Гранди — это теория, описывающая так называемые **равноправные** (англ. "impartial") игры двух игроков, т.е. игры, в которых разрешённые ходы и выигрышность/проигрышность зависят только от состояния игры. От того, какой именно из двух игроков ходит, не зависит ничего: т.е. игроки полностью равноправны.

Кроме того, предполагается, что игроки располагают всей информацией (о правилах игры, возможных ходах, положении соперника).

Предполагается, что игра **конечна**, т.е. при любой стратегии игроки рано или поздно придут в **проигрышную** позицию, из которой нет переходов в другие позиции. Эта позиция является проигрышной для игрока, который должен делать ход из этой позиции. Соответственно, она является выигрышной для игрока, пришедшего в эту позицию. Понятно, ничейных исходов в такой игре не бывает.

Иными словами, такую игру можно полностью описать **ориентированным ациклическим графом**: вершинами в нём являются состояния игры, а рёбрами — переходы из одного состояния игры в другое в результате хода текущего игрока (повторимся, в этом первой и второй игрок равноправны). Одна или несколько вершин не имеют исходящих рёбер, они является проигрышными вершинами (для игрока, который должен совершать ход из такой вершины).

Поскольку ничейных исходов не бывает, то все состояния игры распадаются на два класса: **выигрышные и проигрышные**. Выигрышные — это такие состояния, что найдётся ход текущего игрока, который приведёт к неминуемому поражению другого игрока даже при его оптимальной игре. Соответственно, проигрышные состояния — это состояния, из которых все переходы ведут в состояния, приводящие к победе второго игрока, несмотря на "сопротивление" первого игрока. Иными словами, выигрышным будет состояние, из которого есть хотя бы один переход в проигрышное состояние, а проигрышным является состояние, из которого все переходы ведут в выигрышные состояния (или из которого вообще нет переходов).

Наша задача — для любой заданной игры провести классификацию состояний этой игры, т.е. для каждого состояния определить, выигрышное оно или проигрышное.

Теорию таких игр независимо разработали Роланд Шпраг (Roland Sprague) в 1935 г. и Патрик Майкл Гранди (Patrick Michael Grundy) в 1939 г.

Игра "Ним"

Эта игра является одним из примеров описываемых выше игр. Более того, как мы увидим чуть позже, **любая** из равноправных игр двух игроков на самом деле эквивалентна игре "ним" (англ. "nim"), поэтому изучение этой игры автоматически позволит нам решать все остальные игры (впрочем, об этом позже).

Исторически, эта игра была популярна ещё в древние времена. Вероятно, игра берёт своё происхождение в Китае — по крайней мере, китайская игра "Jianshizi" очень похожа на ним. В Европе первые упоминания о ниме относятся к XVI в. Само название "ним" придумал математик Чарлз Бутон (Charles Bouton), который в 1901 г. опубликовал полный анализ этой игры. Происхождение названия "ним" доподлинно неизвестно.

Описание игры

Игра "ним" представляет из себя следующую игру.

Есть несколько кучек, в каждой из которых по нескольку камней. За один ход игрок может взять из какой-нибудь одной кучки любое ненулевое число камней и выбросить их. Соответственно, проигрыш наступает, когда ходов больше не осталось, т.е. все кучки пусты.

Итак, состояние игры "ним" однозначно описывается неупорядоченным набором натуральных чисел. За один ход разрешается строго уменьшить любое из чисел (если в результате число станет нулём, то оно удаляется из набора).

Решение нима

Решение этой игры опубликовал в 1901 г. Чарлз Бутон (Charles L. Bouton), и выглядит оно следующим образом.

Теорема. Текущий игрок имеет выигрышную стратегию тогда и только тогда, когда XOR-сумма размеров кучек отлична от нуля. В противном случае текущий игрок находится в проигрышном состоянии. (XOR-суммой чисел a_i называется выражение $a_1 \oplus a_2 \oplus \ldots \oplus a_n$, где $\oplus$ — операция побитового исключающего или)

Доказательство.

Основная суть приведённого ниже доказательства — в наличии **симметричной стратегии для противника**. Мы покажем, что, оказавшись в состоянии с нулевой XOR-суммой, игрок не сможет выйти из этого состояния — при любом его переходе в состояние с ненулевой XOR-суммой у противника найдётся ответный ход, возвращающий XOR-сумму обратно в ноль.

Приступим теперь к формальному доказательству (оно будет конструктивным, т.е. мы покажем, как именно выглядит симметричная стратегия противника — какой именно ход нужно будет ему совершать).

Доказывать теорему будем по индукции.

Для пустого нима (когда размеры всех кучек равны нулю) XOR-сумма равна нулю, и теорема верна.

Пусть теперь мы хотим доказать теорему для некоторого состояния игры, из которого есть хотя бы один переход. Пользуясь предположением индукции (и ацикличностью игры) мы считаем, что теорема доказана для всех состояний, в которые мы можем попасть из текущего.

Тогда доказательство распадается на две части: если XOR-сумма s в текущем состоянии $= 0$, то надо доказать, что текущее состояние проигрышно, т.е. все переходы из него ведут в состояния с XOR-суммой $t \neq 0$. Если же $s \neq 0$, то надо доказать, что найдётся переход, приводящий нас в состояние с $t = 0$.

- Пусть $s = 0$, тогда мы хотим доказать, что текущее состояние — проигрышно. Рассмотрим любой переход из текущего состояния нима: обозначим через p номер изменяемой кучки, через $x_i\,(i = 1 \ldots n)$ — размеры кучек до хода, через $y_i\,(i = 1 \ldots n)$ — после хода. Тогда, пользуясь элементарными свойствами функции $\oplus$, мы имеем:

  $$t = s \oplus x_p \oplus y_p = 0 \oplus x_p \oplus y_p = x_p \oplus y_p.$$

  Но поскольку $y_p < x_p$, то это означает, что $t \neq 0$. Значит, новое состояние будет иметь ненулевую XOR-сумму, т. е., согласно базе индукции, будет выигрышным, что и требовалось доказать.

- Пусть $s \neq 0$. Тогда наша задача — доказать, что текущее состояние — выигрышно, т.е. из него существует ход в проигрышное состояние (с нулевой XOR-суммой).

  Рассмотрим битовую запись числа s. Возьмём старший ненулевой бит, пусть его номер равен d. Пусть k — номер той кучки, у размер x_k которой d-ый бит отличен от нуля (такое k найдётся, иначе бы в XOR-сумме s этот бит не получился бы отличным от нуля).

  Тогда, утверждается, искомый ход — это изменить k-ую кучку, сделав её размера $y_k = x_k \oplus s$.

  Убедимся в этом.

  Сначала надо проверить, что это ход корректный, т.е. что $y_k < x_k$. Однако это верно, поскольку все биты, старшие d-го, у x_k и y_k совпадают, а в d-ом бите у y_k будет ноль, а у x_k будет единица.

  Теперь посчитаем, какая XOR-сумма получится при этом ходе:

  $$t = s \oplus x_k \oplus y_k = s \oplus x_k \oplus (s \oplus x_k) = 0.$$

  Таким образом, указанный нами ход — действительно ход в проигрышное состояние, а это и доказывает, что текущее состояние выигрышно.

Теорема доказана.

Следствие. Любое состояние ним-игры можно заменить эквивалентным состоянием, состоящим из единственной кучки размера, равного XOR-сумме размеров кучек в старом состоянии.

Иными словами, при анализе нима с несколькими кучками можно посчитать XOR-сумму s их размеров, и перейти к анализу нима из единственной кучки размера s — как показывает только что доказанная теорема, выигрышность/проигрышность от этого не изменится.

Эквивалентность любой игры ниму. Теорема Шпрага-Гранди

Здесь мы покажем, как любой игре (равноправной игре двух игроков) поставить в соответствие ним. Иными словами, любому состоянию любой игры мы научимся ставить в соответствие ним-кучку, полностью описывающее состояние исходной игры.

Лемма о ниме с увеличениями

Докажем сначала очень важную лемму — **лемму о ниме с увеличениями**.

А именно, рассмотрим следующий модифицированный ним: всё так же, как и в обычном ниме, однако теперь разрешается дополнительный вид хода: вместо уменьшения, наоборот, **увеличить размер некоторой кучки**. Если быть более точным, то ход игрока теперь заключается в том, что он либо забирает ненулевое количество камней из какой-нибудь кучки, либо увеличивает размер какой-либо кучки (в

соответствии с некими правилами, см. следующий абзац).

Здесь важно понимать, что правила того, как именно игрок может совершать увеличения, **нас не интересуют** — однако такие правила всё же должны быть, чтобы наша игра по-прежнему была **ациклична**. Ниже в разделе "Примеры игр" рассмотрены примеры таких игр: "лестничный ним", "nimble-2", "turning turtles".

Повторимся, лемма будет доказана нами вообще для любых игр такого вида — игр вида "ним с увеличениями"; конкретные правила увеличений в доказательстве никак не используются.

Формулировка леммы. Ним с увеличениями эквивалентен обычному ниму, в том смысле, что выигрышность/проигрышность состояния определяется по теореме Бутона согласно XOR-сумме размеров кучек. (Или, иными словами, суть леммы в том, что увеличения бесполезны, их нет смысла применять в оптимальной стратегии, и они никак не меняют выигрышность/проигрышность по сравнению с обычным нимом.)

Доказательство.

Идея доказательства, как и в теореме Бутона — в наличии **симметричной стратегии**. Мы покажем, что увеличения ничего не меняют, поскольку после того, как один из игроков прибегнет к увеличению, другой сможет симметрично ответить ему.

В самом деле, предположим, что текущий игрок совершает ход-увеличение какой-либо кучки. Тогда его соперник сможет просто ответить ему, уменьшив обратно эту кучку до старого значения — ведь обычные ходы нима у нас по-прежнему могут свободно использоваться.

Таким образом, симметричным ответом на ход-увеличение будет ход-уменьшение обратно до старого размера кучки. Следовательно, после такого ответа игра вернётся обратно к тем же размерам кучек, т.е. игрок, совершивший увеличение, ничего от этого не выиграет. Т.к. игра ациклична, то рано или поздно ходы-увеличения кончатся, и текущему игроку придётся делать ход-уменьшение, а это и означает, что наличие увеличивающих ходов не меняет ровным счётом ничего.

Теорема Шпрага-Гранди об эквивалентности любой игры ниму

Перейдём теперь к самому главному в данной статье факту — теореме об эквивалентности ниму любой равноправной игры двух игроков.

Теорема Шпрага-Гранди. Рассмотрим любое состояние v некоторой равноправной игры двух игроков. Пусть из него есть переходы в некоторые состояния $v_i (i = 1 \dots k)$ (где $k \geq 0$). Утверждается, что состоянию v этой игры можно поставить в соответствие кучку нима некоторого размера x (которая будет полностью описывать состояние v нашей игры — т.е. эти два состояния двух разных игр будут эквивалентны). Это число x — называется **значением Шпрага-Гранди** состояния v.

Более того, это число x можно находить следующим рекурсивным образом: посчитаем значение Шпрага-Гранди x_i по каждому переходу (v, v_i), и тогда выполняется:

$$x = \operatorname{mex}\{x_1, \dots, x_k\},$$

где функция mex от множества чисел возвращает наименьшее неотрицательное число, не встречающееся в этом множестве (название "mex" — это сокращение от "minimum excludant").

Таким образом, мы можем, стартуя от вершин без исходящих рёбер, постепенно **посчитать значения Шпрага-Гранди для всех состояний нашей игры**. Если значение Шпрага-Гранди какого-либо состояния равно нулю, то это состояние проигрышно, иначе — выигрышно.

Доказательство. Доказывать теорему будем по индукции.

Для вершин, из которых нет ни одного перехода, величина x согласно теореме будет получаться как mex от пустого множества, т.е. $x = 0$. Но, в самом деле, состояние без переходов — это проигрышное состояние, и ему действительно должна соответствовать ним-кучка размера 0.

Рассмотрим теперь любое состояние v, из которого есть переходы. По индукции мы можем считать, что для всех состояний v_i, в которые мы можем перейти из текущего состояния, значения x_i уже подсчитаны.

Посчитаем величину $p = \operatorname{mex}\{x_1, \dots, x_k\}$. Тогда, согласно определению функции mex, мы получаем, что для любого числа i в промежутке $[0; p)$ найдётся хотя бы один соответствующий переход в какое-то из v_i-ых состояние. Кроме того, могут существовать также дополнительные переходы — в состояния со значениями Гранди, большими p.

Это означает, что текущее состояние **эквивалентно состоянию ниму с увеличениями с кучкой размера** p: в самом деле, у нас есть переходы из текущего состояния в состояния с кучками всех меньших размеров, а также могут быть переходы в состояния больших размеров.

Следовательно, величина $\operatorname{mex}\{x_1, \dots, x_k\}$ действительно является искомым значением Шпрага-Гранди для текущего состояния, что и требовалось доказать.

Применение теоремы Шпрага-Гранди

Опишем наконец целостный алгоритм, применимый к любой равноправной игре двух игроков для определения выигрышности/проигрышности текущего состояния v.

Функция, которая каждому состоянию игры ставит в соответствие ним-число, называется **функцией Шпрага-Гранди**.

Итак, чтобы посчитать функцию Шпрага-Гранди для текущего состояния некоторой игры, нужно:

- Выписать все возможные переходы из текущего состояния.
- Каждый такой переход может вести либо в одну игру, либо в **сумму независимых игр**.

  В первом случае — просто посчитаем функцию Гранди рекурсивно для этого нового состояния.

  Во втором случае, когда переход из текущего состояния приводит в сумму нескольких независимых игр — рекурсивно посчитаем для каждой из этих игр функцию Гранди, а затем скажем, что функция Гранди суммы игр равна XOR-сумме значений этих игр.
- После того, как мы посчитали функцию Гранди для каждого возможного перехода — считаем mex от этих значений, и найденное число — и есть искомое значение Гранди для текущего состояния.
- Если полученное значение Гранди равно нулю, то текущее состояние проигрышно, иначе — выигрышно.

Таким образом, по сравнению с теоремой Шпрага-Гранди здесь мы учитываем то, что в игре могут быть переходы из отдельных состояний в **суммы нескольких игр**. Чтобы работать с суммами игр, мы сначала заменяем каждую игру её значением Гранди, т.е. одной ним-кучкой некоторого размера. После этого мы приходим к сумме нескольких ним-кучек, т.е. к обычному ниму, ответ для которого, согласно теореме Бутона — XOR-сумма размеров кучек.

Закономерности в значениях Шпрага-Гранди

Очень часто при решении конкретных задач, когда требуется научиться считать функцию Шпрага-Гранди для заданной игры, помогает **изучение таблиц значений** этой функции в поисках закономерностей.

Во многих играх, кажущихся весьма трудными для теоретического анализа, функция Шпрага-Гранди на практике оказывается периодичной, либо же имеющей очень простой вид, который легко заметить "на глаз". В подавляющем большинстве случаев увиденные закономерности являются верными, и при желании доказываемыми с помощью математической индукции.

Впрочем, далеко не всегда функция Шпрага-Гранди содержит простые закономерности, а для некоторых, даже весьма простых по формулировке, игр вопрос о наличии таких закономерностей до сих пор открыт (например, "Grundy's game" ниже).

Примеры игр

Для наглядной демонстрации теории Шпрага-Гранди, разберём несколько задач.

Особо следует обратить внимание на задачи "лестничный ним", "nimble-2", "turning turtles", в которой демонстрируется нетривиальное сведение исходной задачи к ниму с увеличениями.

"Крестики-крестики"

Условие. Рассмотрим клетчатую полоску размера $1 \times n$ клеток. За один ход игроку надо поставить один крестик, но при этом запрещено ставить два крестика рядом (в соседние клетки). Проигрывает тот, кто не может сделать ход. Сказать, кто выиграет при оптимальной игре.

Решение. Когда игрок ставит крестик в какую-либо клетку, можно считать, что вся полоска распадается на две независимые половинки: слева от крестика и справа от него. При этом сама клетка с крестиком, а также её левый и правый сосед уничтожаются — т.к. в них больше ничего нельзя будет поставить.

Следовательно, если мы занумеруем клетки полоски от 1 до n, то, поставив крестик в позицию $1 < i < n$, полоска распадётся на две полоски длины $i - 2$ и $n - i - 1$, т.е. мы переходим в сумму двух игр $i - 2$ и $n - i - 1$. Если же крестик ставится в позицию 1 или n, то это особый случай — мы просто перейдём в состояние $n - 2$.

Таким образом, функция Гранди $g[n]$ имеет вид (для $n \geq 3$):

$$g[n] = \operatorname{mex}\left\{ g[n-2], \bigcup_{i=2}^{n-1} \Big(g[i-2] \oplus g[n-i-1] \Big) \right\}.$$

Т.е. $g[n]$ получается как mex от множества, состоящего из числа $g[n-2]$, а также всевозможных

значений выражения $g[i-2] \oplus g[n-i-1]$.

Итак, мы получили решение этой задачи за $O(n^2)$.

На самом деле, посчитав на компьютере таблицу значений для первой сотни значений n, можно увидеть, что, начиная с $n = 52$, последовательность $g[n]$ становится периодичной с периодом 34. Эта закономерность сохраняется и дальше (что, вероятно, можно доказать по индукции).

"Крестики-крестики — 2"

Условие. Снова игра ведётся на полоске $1 \times n$ клеток, и игроки по очереди ставят по одному крестику. Выигрывает тот, кто поставит три крестика подряд.

Решение. Заметим, что если $n > 2$ и мы оставили после своего хода два крестика рядом или через один пробел, то противник следующим ходом выиграет. Следовательно, если один игрок поставил где-то крестик, то другому игроку невыгодно ставить крестик в соседние с ним клетки, а также в соседние с соседними (т.е. на расстоянии 1 и 2 ставить невыгодно, это приведёт к поражению).

Тогда решение получается практически аналогичным предыдущей задаче, только теперь крестик удаляет у каждой половинки не по одной, а сразу по две клетки.

"Пешки"

Условие. Есть поле $3 \times n$, на котором в первом и третьем ряду стоят по n пешек — белых и чёрных, соответственно. Первый игрок ходит белыми пешками, второй — чёрными. Правила хода и удара — стандартные шахматные, за исключением того, что бить (при наличии такой возможности) обязательно.

Решение. Проследим, что происходит, когда одна пешка сделает ход вперёд. Следующим ходом противник будет обязан съесть её, затем мы будем обязаны съесть пешку противника, затем снова он съест, и, наконец, наша пешка съест вражескую пешку и останется, "упёршись" в пешку противника. Таким образом, если мы в самом начале пошли пешкой в колонке $1 < i < n$, то в результате три колонки $[i-1; i+1]$ доски фактически уничтожатся, и мы перейдём к сумме игр размера $i-2$ и $n-i-1$. Граничные случаи $i = 1$ и $i = n$ приводят нас просто к доске размера $n-2$.

Таким образом, мы получили выражения для функции Гранди, аналогичные рассмотренной выше задаче "Крестики-крестики".

"Lasker's nim"

Условие. Имеется n кучек камней заданных размеров. За один ход игрок может взять любое ненулевое число камней из какой-либо кучки, либо же разделить какую-либо кучку на две непустые кучки. Проигрывает тот, кто не может сделать ход.

Решение. Записывая оба вида переходов, легко получить функцию Шпрага-Гранди как:

$$g[n] = \operatorname{mex}\left\{ \bigcup_{i=0}^{n-1} g[i], \quad \bigcup_{i=1}^{n-1} \Big(g[i] \oplus g[n-i] \Big) \right\}.$$

Однако можно построить таблицу значений для малых n и увидеть простую закономерность:

n	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
$g[n]$	0	1	2	4	3	5	6	8	7	9	10	12	11	13	14	16	15	17	18	20

Здесь видно, что $g[n] = n$ для чисел, равных 1 или 2 по модулю 4, и $g[n] = n \pm 1$ для чисел, равных 3 и 0 по модулю 4. Доказать это можно по индукции.

"The game of Kayles"

Условие. Есть n кегель, выставленных в ряд. За один удар игрок может выбить либо одну кеглю, либо две рядом стоящие кегли. Выигрывает тот, который выбил последнюю кеглю.

Решение. И когда игрок выбивает одну кеглю, и когда он выбивает две — игра распадается на сумму двух независимых игр.

Нетрудно получить такое выражение для функции Шпрага-Гранди:

$$g[n] = \operatorname{mex}\left\{ \bigcup_{i=0}^{n-1} \Big(g[i] \oplus g[n-1-i] \Big), \quad \bigcup_{i=0}^{n-2} \Big(g[i] \oplus g[n-2-i] \Big) \right\}.$$

Посчитаем для него таблицу для нескольких первых десятков элементов:

$g[0\ldots11]:$	0	1	2	3	1	4	3	2	1	4	2	6
$g[12\ldots23]:$	4	1	2	7	1	4	3	2	1	4	6	7
$g[24\ldots35]:$	4	1	2	8	5	4	7	2	1	8	6	7
$g[36\ldots47]:$	4	1	2	3	1	4	7	2	1	8	2	7
$g[48\ldots59]:$	4	1	2	8	1	4	7	2	1	4	2	7
$g[60\ldots71]:$	4	1	2	8	1	4	7	2	1	8	6	7
$g[72\ldots83]:$	4	1	2	8	1	4	7	2	1	8	2	7
$g[84\ldots95]:$	4	1	2	8	1	4	7	2	1	8	2	7
$g[96\ldots107]:$	4	1	2	8	1	4	7	2	1	8	2	7
$g[108\ldots119]:$	4	1	2	8	1	4	7	2	1	8	2	7

Можно заметить, что, начиная с некоторого момента, последовательность становится периодичной с периодом 12. В дальнейшем эта периодичность также не нарушится.

Grundy's game

Условие. Есть n кучек камней, размеры которых мы обозначим через a_i. За один ход игрок может взять какую-либо кучку размера как минимум 3 и разделить её на две непустые кучки неравных размеров. Проигрывает тот, кто не может сделать ход (т.е. когда размеры всех оставшихся кучек меньше либо равны двум).

Решение. Если $n > 1$, то все эти несколько кучек, очевидно, — независимые игры. Следовательно, наша задача — научиться искать функцию Шпрага-Гранди для одной кучки, а ответ для нескольких кучек будет получаться как их XOR-сумма.

Для одной кучки эта функция строится также легко, достаточно просмотреть все возможные переходы:

$$g[n] = \operatorname{mex}\left\{ \bigcup_{\substack{i=[1\ldots n-1],\\ i\neq n-i}} \Big(g[i] \oplus g[n-i]\Big) \right\}.$$

Чем эта игра интересна — тем, что до сих пор для неё не найдено общей закономерности. Несмотря на предположения, что последовательность $g[n]$ должна быть периодичной, она была просчитана вплоть до 2^{35}, и периодов в этой области обнаружено не было.

"Лестничный ним"

Условие. Есть лестница с n ступеньками (занумерованными от 1 до n), на i-ой ступеньке лежит a_i монет. За один ход разрешается переместить некоторое ненулевое число монет с i-ой на $i-1$-ую ступеньку. Проигрывает тот, кто не может сделать хода.

Решение. Если попытаться свести эту задачу к ниму "в лоб", то получится, что ход у нас — это уменьшение одной кучки на сколько-то, и одновременное увеличение другой кучки на столько же. В итоге мы получаем модификацию нима, решить которую весьма сложно.

Поступим по-другому: рассмотрим только ступеньки с нечётными номерами: $a_1, a_3, a_5, \ldots$. Посмотрим, как будет меняться этот набор чисел при совершении одного хода.

Если ход делается с чётным i, то тогда этот ход означает увеличение числа a_{i-1}. Если же ход делается с нечётным i, то это означаем уменьшение a_i.

Получается, что наша задача — это обыкновенный ним с увеличениями с размерами кучек $a_1, a_3, a_5, \ldots$.

Следовательно, функция Гранди от него — это XOR-сумма чисел вида a_{2i+1}.

"Nimble" и "Nimble-2"

Условие. Есть клетчатая полоска $1 \times n$, на которой расположены k монет: i-ая монета находится в a_i-ой клетке. За один ход игрок может взять какую-то монету и подвинуть её влево на произвольное число клеток, но так, чтобы она не вылезла за пределы полоски. В игре "Nimble-2" дополнительно запрещается перепрыгивать другие монеты (или даже ставить две монеты в одну клетку). Проигрывает тот, кто не может сделать ход.

Решение "Nimble". Заметим, что монеты в этой игре независимы друг от друга. Более того, если мы рассмотрим набор чисел $a_i - 1 (i = 1\ldots k)$, то понятно, что за один ход игрок может взять любое из этих чисел и уменьшить его, а проигрыш наступает, когда все числа обращаются в ноль. Следовательно, игра "Nimble" — это **обычный ним**, и ответом на задачу является XOR-сумма чисел $a_i - 1$.

Решение "Nimble-2". Перенумеруем монеты в порядке их следования слева направо. Тогда обозначим через d_i расстояние от i-ой до $i-1$-ой монеты:

$$d_i = a_i - a_{i-1} - 1, \qquad (i = 1\ldots k)$$

(считая, что $a_0 = 0$).

Тогда за один игрок может отнять от какого-нибудь d_p некоторое число q, и прибавить это же число q к d_{p+1}. Таким образом, эта игра — это фактически "**лестничный ним**" над числами d_i (надо лишь изменить порядок этих чисел на противоположный).

"Turning turtles" и "Twins"

Условие. Дана клетчатая полоска размера $1 \times n$. В каждой клетке стоит либо крестик, либо нолик. За один ход можно взять какой-то нолик и превратить его в крестик.

При этом **дополнительно** разрешается выбрать одну из клеток слева от изменяемой и изменить в ней значение на противоположное (т.е. нолик заменить на крестик, а крестик — на нолик). В игре "turning turtles" делать это не обязательно (т.е. ход игрока может ограничиваться превращением нолика в крестик), а в "twins" — обязательно.

Решение "turning turtles". Утверждается, что эта игра — это обычный ним над числами b_i, где b_i — позиция i-го нолика (в 1-индексации). Проверим это утверждение.

- Если игрок просто поменял нолик на крестик, не воспользовавшись дополнительным ходом — то это можно понимать как то, что он просто забрал всю кучку, соответствующую этому нолику. Иными словами, если игрок поменял нолик на крестик в позиции $x (1 \le x \le n)$, то тем самым он взял кучку размера x и сделал её размер равным нулю.
- Если игрок воспользовался дополнительным ходом, т.е. помимо того, что поменял крестик в позиции x на нолик, он ещё изменил клетку в позиции $y (y < x)$, то можно считать, что он уменьшил кучку x до размера y. Действительно, если в позиции y раньше был крестик — то, в самом деле, после хода игрока там станет нолик, т. е. появится кучка размера y. А если в позиции y раньше был нолик, то после хода игрока эта кучка исчезает — или, что то же самое, появилась вторая кучка точно такого же размера y (поскольку в ниме две кучки одинаковых размеров фактически "уничтожают" друг друга).

Таким образом, ответ на задачу — это XOR-сумма чисел — координат все ноликов в 1-индексации.

Решение "twins". Все рассуждения, приведённые выше, остаются верны, за исключением того, что хода "обнулить кучку" теперь у игрока нет. Т.е. если мы от всех координат отнимем единицу — то снова игра превратится в обычный ним.

Таким образом, ответ на задачу — это XOR-сумма чисел — координат все ноликов в 0-индексации.

Northcott's game

Условие. Есть доска размера $n \times m$: n строк и m столбцов. В каждой строке стоят по две фишки: одна чёрная и одна белая. За один ход игрок может взять любую фишку своего цвета и подвинуть её внутри строки вправо или влево на произвольное число шагов, но не перепрыгивая через другую фишку (и не вставая на неё). Проигрывает тот, кто не может сделать хода.

Решение. Во-первых, понятно, что каждая из n строк доски образует независимую игру. Поэтому задача сводится к анализу игры в одной строке, а ответом на задачу будет XOR-сумма значений Шпрага-Гранди для каждой из строк.

Решая задачу для одной строки, обозначим через x расстояние между чёрной и белой фишкой (которое может меняться от нуля до $m - 2$). За один ход каждый игрок может либо уменьшить x на некоторое произвольное значение, либо, возможно, увеличить его до некоторого значения (увеличения доступны не всегда). Таким образом, эта игра — это "**ним с увеличениями**", и, как мы уже знаем, увеличения в этой игре бесполезны. Следовательно, функция Гранди для одной строки — это и есть это расстояние x.

(Следует заметить, что формально такое рассуждение неполно — т.к. в "ниме с увеличениями" предполагается, что игра **конечна**, а здесь правила игры позволяют игрокам играть бесконечно долго. Впрочем, бесконечная игра не может иметь места при оптимальной игре — т.к. стоит одному игроку увеличить расстояние x (ценой приближения к границе поля), как другой игрок приблизится к нему, уменьшив x обратно. Следовательно, при оптимальной игре противника игроку не удастся совершать увеличивающие ходы бесконечно долго, поэтому всё же описанное решение задачи остаётся в силе.)

Триомино

Условие. Дано клетчатое поле размера $2 \times n$. За один ход игрок может поставить на поле одну фигурку в форме буквы "Г" (т.е. связную фигуру из трёх клеток, не лежащих на одной прямой). Запрещено ставить фигурку так, чтобы она пересеклась хотя бы одной клеткой с какой-то из уже поставленных фигурок. Проигрывает тот, кто не может сделать ход.

Решение. Заметим, что постановка одной фигурки разбивает всё поле на два независимых поля. Таким образом, нам надо анализировать не только прямоугольные поля, но и поля, у которых левая и/или правая границы неровные.

Нарисовав различные конфигурации, можно понять, что какой бы ни была конфигурация поля, главное — лишь то, сколько на этом поле клеток. На самом деле, если в текущем поле x свободных клеток, и мы хотим разбить это поле на два поля размером y и z (где $y + z + 3 = x$), то это всегда можно сделать, т.е. всегда можно найти соответствующее место для фигурки.

Таким образом, наша задача превращается в такую: изначально у нас есть кучка камней размера $2n$, и за один ход мы можем выкинуть из некоторой кучки 3 камня и затем разбить эту кучку на две кучки произвольных размеров. Функция Гранди для такой игры имеет вид:

$$g[n] = \operatorname{mex}\left\{ \bigcup_{i=0}^{n-3} \Big(g[i] \oplus g[n-i-3] \Big) \right\}.$$

Фишки на графе

Условие. Дан ориентированный ациклический граф. В некоторых вершинах графа стоят фишки. За один ход игрок может взять какую-то фишку и передвинуть её вдоль какого-либо ребра в новую вершину. Проигрывает тот, кто не может сделать ход.

Также бывает и второй вариант этой задачи: когда считается, что если две фишки приходят в одну вершину, то они обе взаимно уничтожают друг друга.

Решение первого варианта задачи. Во-первых, все фишки — независимы друг от друга, поэтому наша задача — научиться искать функцию Гранди для одной фишки в графе.

Учитывая, что граф ацикличен, мы можем делать это рекурсивно: предположим, что мы посчитали функцию Гранди для всех потомков текущей вершины. Тогда функция Гранди в текущей вершине — это mex от этого множества чисел.

Таким образом, решением задачи является следующее: для каждой вершины рекурсивно посчитать функцию Гранди, если бы фишка стояла именно в этой вершине. После этого ответом на задачу будет XOR-сумма значений Гранди от тех вершин графа, в которых по условию стоят фишки.

Решение второго варианта задачи. На самом деле, второй вариант задачи ничем не отличается от первого. В самом деле, если две фишки стоят в одной и той же вершине графа, то в результирующей XOR-сумме их значения Гранди взаимно уничтожают друг друга. Следовательно, фактически это одна и та же задача.

Реализация

С позиции реализации интерес может представлять реализация функции mex.

Если это не является узким местом в программе, то можно написать какой-нибудь простой вариант за $O(c \log c)$ (где c — количество аргументов):

```
int mex (vector<int> a) {
        set<int> b (a.begin(), a.end());
        for (int i=0; ; ++i)
                if (! b.count (i))
                        return i;
}
```

Впрочем, не так уж и сложным является вариант **за линейное время**, т.е. за $O(c)$, где c — число аргументов функции mex. Обозначим через D константу, равную максимально возможному значению c (т. е. максимальной степени вершины в графе игры). В таком случае результат функции mex не будет превосходить D.

Следовательно, при реализации достаточно завести массив размера $D+1$ (массив глобальный, или статический — главное, чтобы он не создавался при каждом вызове функции). При вызове функции mex мы сначала отметим в этом массиве все c аргументов (пропустив те из них, которые больше D — такие значения, очевидно, не влияют на результат). Затем проходом по этому массиву мы за $O(c)$ найдём первый неотмеченный элемент. Наконец, в конце можно снова пройтись по всем переданным аргументам и обнулить обратно массив для них. Тем самым, мы выполним все действия за $O(c)$, что на практике может оказаться существенно меньше максимальной степени D.

```
int mex (const vector<int> & a) {
        static bool used[D+1] = { 0 };
        int c = (int) a.size();

        for (int 8  i=0; i<c; ++i)
                if (a[i] <= D)
                        used[a[i]] = true;

        int result;
        for (int i=0; ; ++i)
                if (!used[i]) {
                        result = i;
                         2 break;
```

```
                }

        for (int  1 i=0; i<c; ++i)
                if (a[i] <= D)
                        used[a[i]] = false;

        return result;
}
```

Другой вариант — воспользоваться техникой "**числового** `used`". Т.е. сделать used массивом не булевских переменных, а чисел ("версий"), и завести глобальную переменную, обозначающую номер текущей версии. При входе в функцию mex мы увеличиваем номер текущей версии, в первом цикле мы проставляем в массиве used не true, а номер текущей версии. Наконец, во втором цикле мы просто сравниваем $\text{used}[i]$ с номером текущей версии — если они не совпали, то это означает, что текущее число не встречалось в массиве a. Третий цикл (который ранее занулял массив used) в таком решении не нужен.

Обобщение нима: ним Мура (k-ним)

Одно из интересных обобщений обычного нима было дано Муром (Moore) в 1910 г.

Условие. Есть n кучек камней размера a_i. Также задано натуральное число k. За один ход игрок может уменьшить размеры от одной до k кучек (т.е. теперь разрешаются одновременные ходы в нескольких кучках сразу). Проигрывает тот, кто не может сделать хода.

Очевидно, при $k = 1$ ним Мура превращается в обычный ним.

Решение. Решение такой задачи удивительно просто. Запишем размеры каждой кучки в двоичной системе счисления. Затем просуммируем эти числа в $k + 1$-ичной системе счисления без переносов разрядов. Если получилось число ноль, то текущая позиция проигрышная, иначе — выигрышная (и из неё есть ход в позицию с нулевой величиной).

Иными словами, для каждого бита мы смотрим, стоит этот бит или нет в двоичном представлении каждого числа a_i. Затем мы суммируем получившиеся нули/единицы, и сумму берём по модулю $k + 1$. Если в итоге эта сумма для каждого бита получилась нулевой, то текущая позиция — проигрышная, иначе — выигрышная.

Доказательство. Как и для нима, доказательство заключается в описании стратегии игроков: с одной стороны, мы показываем, что из игры с нулевым значением мы можем перейти только в игру с ненулевым значением, а с другой стороны — что из игры с ненулевым значением есть ход в игру с нулевым значением.

Во-первых, покажем, что из игры с нулевым значением можно перейти только в игру с ненулевым значением. Это вполне понятно: если сумма по модулю $k + 1$ была равна нулю, то после изменения от одного до k бит мы не могли получить снова нулевую сумму.

Во-вторых, покажем, как из игры с ненулевой суммы перейти в игру с нулевой суммой. Будем перебирать биты, в которых сумма получилась ненулевой, в порядке от старшего к младшему.

Обозначим через u количество кучек, которые мы уже начали изменять; изначально $u = 0$. Обратим внимание, что в этих u кучках мы уже можем ставить любые биты по нашему желанию (поскольку у любой кучки, которая попадала в число этих u кучек, уже уменьшился один из предыдущих, более старших, битов).

Итак, пусть мы рассматриваем текущий бит, в котором сумма по модулю $k + 1$ получилась ненулевой. Обозначим через s эту сумму, но в которой не учитываются те u кучек, которые мы уже начали изменять. Обозначим через q сумму, которую можно получить, поставив в эти u кучек текущий бит равным единице:

$$q = (s + u) \pmod{(k + 1)}$$

У нас есть два варианта:

- Если $q \le u$.

  Тогда мы можем обойтись только лишь уже выбранными u кучками: достаточно в $k + 1 - s = u - q$ из них поставить текущий бит равным единице, а во всех остальных — равным нулю.

- Если $q > u$.

  В таком случае мы, наоборот, поставим в уже выбранных u кучках текущий бит равным нулю. Тогда сумма в текущем бите будет равна $s > 0$, а, значит, среди невыбранных $n - u$ кучек в текущем бите есть как минимум s единиц. Выберем какие-нибудь s кучек среди них, и уменьшим в них текущий бит с единицы до нуля.

  В результате число u изменяемых кучек увеличится на s, и составит $q \le k$.

Таким образом, мы показали, каким образом выбирать множество изменяемых кучек и какие биты следует в них изменять, чтобы общее их количество u никогда не превысило k.

Следовательно, мы доказали, что искомый переход из состояния с ненулевой суммой в состояние с нулевой

суммой существует, что и требовалось доказать.

"Ним в поддавки"

Тот ним, который мы рассматривали во всей данной статье — называется также "нормальным нимом" ("normal nim"). В противоположность ему, существует также "**ним в поддавки**" ("misère nim") — когда игрок, совершивший последний ход, проигрывает (а не выигрывает).

(кстати говоря, по всей видимости, ним как настольная игра — более популярен именно в версии "в поддавки", а не в "нормальной" версии)

Решение такого нима удивительно просто: будем действовать так же, как и в обычном ниме (т.е. посчитаем XOR-сумму всех размеров кучек, и если она равна нулю, то мы проиграем при любой стратегии, а иначе — выиграем, найдя переход в позицию с нулевым значением Шпрага-Гранди). Но есть одно **исключение**: если размеры всех кучек равны единице, то выигрышность/проигрышность меняются местами по сравнению с обычным нимом.

Таким образом, выигрышность/проигрышность нима "в поддавки" определяются по числу:

$$a_1 \oplus a_2 \oplus \ldots \oplus a_n \oplus z,$$

где через z обозначена булевская переменная, равная единице, если $a_1 = a_2 = \ldots = a_n = 1$.

С учётом этого исключения, **оптимальная стратегия** для игрока в выигрышной позиции определяется следующим образом. Найдём ход, который игрок бы совершил, если бы он играл в нормальный ним. Теперь, если этот ход ведёт в позицию, в которой размеры всех кучек равны единице (и при этом до этого хода была кучка размера, большего единицы), то этот ход надо изменить: изменить так, чтобы количество остающихся непустых кучек изменило свою чётность.

Доказательство. Обратим внимание, что вообще теория Шпрага-Гранди относится к "нормальным" играм, а не к играм в поддавки. Однако ним — одна из тех игр, для которых решение игры "в поддавки" не сильно отличается от решения "нормальной" игры. (Кстати говоря, решение нима "в поддавки" было дано тем же Чарлзом Бутоном, который описал решение "нормального" нима.)

Каким же образом можно объяснить столь странную закономерность — что выигрышность/проигрышность нима "в поддавки" совпадает с выигрышностью/проигрышностью "нормального" нима почти всегда?

Рассмотрим некоторое **течение игры**: т.е. выберем произвольную стартовую позицию и выпишем ходы игроков вплоть до завершения игры. Нетрудно понять, что при условии оптимальной игры соперников — игра завершится тем, что останется одна кучка размера 1, и игрок будет вынужден пойти в ней и проиграть.

Следовательно, в любой игре двух оптимальных игроков рано или поздно наступает **момент**, когда размеры всех непустых кучек равны единице. Обозначим через k число непустых кучек в этот момент — тогда для текущего игрока эта позиция выигрышна тогда и только тогда, когда k чётно. Т.е. мы убедились в том, что в этих случаях выигрышность/проигрышность нима "в поддавки" **противоположна** "нормальному" ниму.

Снова вернёмся к тому моменту, когда впервые в игре все кучки стали размера 1, и откатимся на один ход назад — прямо перед тем, как эта ситуация получилась. Мы оказались в ситуации, что одна кучка имеет размер > 1, а все остальные кучки (возможно, их было ноль штук) — размера 1. Эта позиция очевидно выигрышна (т.к. мы в самом деле всегда можем сделать такой ход, чтобы осталось нечётное число кучек размера 1, т.е. приведём соперника к поражению). С другой стороны, XOR-сумма размеров кучек в этот момент отлична от нуля — значит, здесь "нормальный" ним **совпадает** с нимом "в поддавки".

Далее, если продолжим откатываться по игре назад, то мы будем приходить в состояния, когда в игре было две кучки размера > 1, три кучки, и т.д. Для всех таких состояний выигрышность/проигрышность также будет совпадать с "нормальным" нимом — просто потому, что когда у нас есть более одной кучки размера > 1, то все переходы ведут в состояния с одной и более кучкой размера > 1 — а для всех них, как мы уже показали, ничего по сравнению с "нормальным" нимом **не изменилось**.

Таким образом, изменения в ниме "в поддавки" затрагивают только состояния, когда все кучки имеют размер, равный единице — что и требовалось доказать.

Задачи в online judges

Список задач в online judges, которые можно решать с помощью функции Гранди:

- TIMUS #1465 "**Игра в пешки**" [сложность: низкая]
- UVA #11534 "Say Goodbye to Tic-Tac-Toe" [сложность: средняя]
- SGU #328 "A Coloring Game" [сложность: средняя]

Литература

- John Horton Conway. On numbers and games [1979]
- Bernhard von Stengel. Lecture Notes on Game Theory

Задача Джонсона с одним станком

Это задача составления оптимального расписания обработки n деталей на единственном станке, если i-ая деталь обрабатывается на нём за время t_i, а за t секунд ожидания до обработки этой детали платится штраф $f_i(t)$.

Таким образом, задача заключается в поиске такого переупорядочения деталей, что следующая величина (размер штрафа) минимальна. Если мы обозначим через π перестановку деталей (π_1 — номер первой обрабатываемой детали, π_2 — второй, и т.д.), то размер штрафа $f(\pi)$ равен:

$$F(\pi) = f_{\pi_1}(0) + f_{\pi_2}(t_{\pi_1}) + f_{\pi_3}(t_{\pi_1} + t_{\pi_2}) + \ldots + f_{\pi_n}\left(\sum_{i=1}^{n-1} t_{\pi_i}\right).$$

Иногда эта задача называется задачей однопроцессорного обслуживания множества заявок.

Решение задачи в некоторых частных случаях

Первый частный случай: линейные функции штрафа

Научимся решать эту задачу в случае, когда все $f_i(t)$ линейны, т.е. имеют вид:

$$f_i(t) = c_i \cdot t,$$

где c_i — неотрицательные числа. Заметим, что в этих линейных функциях свободный член равен нулю, т.к. в противном случае к ответу сразу можно прибавить этот свободный член, и решать задачу с нулевым свободным членом.

Зафиксируем некоторое расписание — перестановку π. Зафиксируем какой-то номер $i = 1 \ldots n-1$, и пусть перестановка π' равна перестановке π, в которой обменяли i-ый и $i+1$-ый элементы. Посмотрим, на сколько при этом изменился штраф:

$$F(\pi') - F(\pi) =$$

легко понять, что изменения произошли только с i-ым и $i+1$-ым слагаемыми:

$$\begin{aligned}
&= c_{\pi'_i} \cdot \sum_{k=1}^{i-1} t_{\pi'_k} + c_{\pi'_{i+1}} \cdot \sum_{k=1}^{i} t_{\pi'_k} - c_{\pi_i} \cdot \sum_{k=1}^{i-1} t_{\pi_k} - c_{\pi_{i+1}} \cdot \sum_{k=1}^{i} t_{\pi_k} = \\
&= c_{\pi_{i+1}} \cdot \sum_{k=1}^{i-1} t_{\pi'_k} + c_{\pi_i} \cdot \sum_{k=1}^{i} t_{\pi'_k} - c_{\pi_i} \cdot \sum_{k=1}^{i-1} t_{\pi_k} - c_{\pi_{i+1}} \cdot \sum_{k=1}^{i} t_{\pi_k} = \\
&= c_{\pi_i} \cdot t_{\pi_{i+1}} - c_{\pi_{i+1}} \cdot t_{\pi_i}.
\end{aligned}$$

Понятно, что если расписание π является оптимальным, то любое его изменение приводит к увеличению штрафа (или сохранению прежнего значения), поэтому для оптимального плана можно записать условие:

$$\forall i = 1 \ldots n-1 \quad : \quad c_{\pi_i} \cdot t_{\pi_{i+1}} - c_{\pi_{i+1}} \cdot t_{\pi_i} \geq 0.$$

Преобразуя, получаем:

$$\forall i = 1 \ldots n-1 \quad : \quad \frac{c_{\pi_i}}{t_{\pi_i}} \geq \frac{c_{\pi_{i+1}}}{t_{\pi_{i+1}}}.$$

Таким образом, **оптимальное расписание** можно получить, просто **отсортировав** все детали по отношению c_i к t_i в обратном порядке.

Следует отметить, что мы получили этот алгоритм так называемым **перестановочным приёмом**: мы попробовали обменять местами два соседних элемента расписания, вычислили, насколько при этом изменился штраф, и отсюда вывели алгоритм поиска оптимального расписания.

Второй частный случай: экспоненциальные функции штрафа

Пусть теперь функции штрафа имеют вид:

$$f_i(t) = c_i \cdot e^{\alpha \cdot t},$$

где все числа c_i неотрицательны, константа α положительна.

Тогда, применяя аналогичным образом здесь перестановочный приём, легко получить, что детали надо сортировать в порядке убывания величин:

$$v_i = \frac{1 - e^{\alpha \cdot t_i}}{c_i}.$$

Третий частный случай: одинаковые монотонные функции штрафа

В этом случае считается, что все $f_i(t)$ совпадают с некоторой функцией $\phi(t)$, которая является возрастающей.

Понятно, что в этом случае оптимально располагать детали в порядке увеличения времени обработки t_i.

Теорема Лившица-Кладова

Теорема Лившица-Кладова устанавливает, что перестановочный приём применим только для вышеописанных трёх частных случаев, и только них, т.е.:

- Линейный случай: $f_i(t) = c_i \cdot t + d_i$, где c_i — неотрицательные константы,
- Экспоненциальный случай: $f_i(t) = c_i \cdot e^{\alpha \cdot t} + d_i$, где c_i и α — положительные константы,
- Тождественный случай: $f_i(t) = \phi(t)$, где ϕ — возрастающая функция.

Эта теорема доказана в предположении, что функции штрафа являются достаточно гладкими (существуют третьи производные).

Во всех трёх случаях применим перестановочный приём, благодаря которому искомое оптимальное расписание может быть найдено простой сортировкой, следовательно, за время $O(n \log n)$.

Задача Джонсона с двумя станками

Имеется n деталей и два станка. Каждая деталь должна сначала пройти обработку на первом станке, затем — на втором. При этом i-ая деталь обрабатывается на первом станке за a_i времени, а на втором — за b_i времени. Каждый станок в каждый момент времени может работать только с одной деталью.

Требуется составить такой порядок подачи деталей на станки, чтобы итоговое время обработки всех деталей было бы минимальным.

Эта задача называется иногда задачей двухпроцессорного обслуживания задач, или задачей Джонсона (по имени S. M. Johnson, который в 1954 г. предложил алгоритм для её решения).

Стоит отметить, что когда число станков больше двух, эта задача становится NP-полной (как доказал Гэри (Garey) в 1976 г.).

Построение алгоритма

Заметим вначале, что можно считать, что порядок обработки деталей **на первом и втором станках должен совпадать**. В самом деле, т.к. детали для второго станка становятся доступными только после обработки на первом, а при наличии нескольких доступных для второго станка деталей время их обработки будет равно сумме их b_i независимо от их порядка — то выгоднее всего отправлять на второй станок ту из деталей, которая раньше других прошла обработку на первом станке.

Рассмотрим порядок подачи деталей на станки, совпадающий с их входным порядком: $1, 2, \ldots, n$.

Обозначим через x_i **время простоя** второго станка непосредственно перед обработкой i-ой детали (после обработки $i-1$-ой детали). Наша цель — **минимизировать суммарный простой**:

$$F(x) = \sum x_i \longrightarrow \min .$$

Для первой детали мы имеем:

$$x_1 = a_1 .$$

Для второй — т.к. она становится готовой к отправке на второй станок в момент времени $a_1 + a_2$, а второй станок освобождается в момент времени $x_1 + b_1$, то имеем:

$$x_2 = \max\Big((a_1 + a_2) - (x_1 + b_1), 0\Big).$$

Третья деталь становится доступной для второго станка в момент $a_1 + a_2 + a_3$, а станок освобождается в $x_1 + b_1 + x_2 + b_2$, поэтому:

$$x_3 = \max\Big((a_1 + a_2 + a_3) - (x_1 + b_1 + x_2 + b_2), 0\Big).$$

Таким образом, общий вид для x_i выглядит так:

$$x_k = \max\left(\sum_{i=1}^{k} a_i - \sum_{i=1}^{k-1} b_i - \sum_{i=1}^{k-1} x_i, 0\right).$$

Посчитаем теперь **суммарный простой** $F(x)$. Утверждается, что он имеет вид:

$$F(x) = \max_{k=1\ldots n} K_i,$$

где

$$K_i = \sum_{i=1}^{k} a_i - \sum_{i=1}^{k-1} b_i .$$

(В это можно убедиться по индукции, либо последовательно находя выражения для суммы первых двух, трёх, и т.д. x_i.)

Воспользуемся теперь **перестановочным приёмом**: попробуем обменять какие-либо два соседних элемента j и $j+1$ и посмотрим, как при этом изменится суммарный простой.

По виду функции выражений для K_i понятно, что изменятся только K_j и K_{j+1}; обозначим их новые значения через

K'_j и K'_{j+1}.

Таким образом, чтобы деталь j шла до детали $j+1$, достаточно (хотя и не необходимо), чтобы:

$$\max\left(K_j, K_{j+1}\right) \le \max\left(K'_j, K'_{j+1}\right).$$

(т.е. мы проигнорировали остальные, не изменившиеся, аргументы максимума в выражении для $F(x)$, получив тем самым достаточное, но не необходимое условие того, что старое $F(x)$ меньше либо равно нового значения)

Отняв $\sum_{i=1}^{j+1} a_i - \sum_{i=1}^{j-1} b_i$ от обеих частей этого неравенства, получим:

$$\max(-a_{j+1}, -b_j) \le \max(-b_{j+1}, -a_j),$$

или, избавляясь от отрицательных чисел, получаем:

$$\min(a_j, b_{j+1}) \le \min(b_j, a_{j+1}).$$

Тем самым, мы получили **компаратор**: отсортировав детали по нему, мы, согласно приведённым выше выкладкам, придём к оптимальному порядку деталей, в котором нельзя переставить местами никакие две детали, улучшив итоговое время.

Впрочем, можно ещё больше **упростить** сортировку, если посмотреть на этот компаратор с другой стороны. Фактически он говорит нам о том, что если минимум из четырёх чисел $(a_j, a_{j+1}, b_j, b_{j+1})$ достигается на элементе из массива a, то соответствующая деталь должна идти раньше, а если на элементе из массива b — то позже. Тем самым мы получаем другую форму алгоритма: отсортировать детали по минимуму из (a_i, b_i), и если у текущей детали минимум равен a_i, то эту деталь надо обработать первой из оставшихся, иначе — последней из оставшихся.

Так или иначе, получается, что задача Джонсона с двумя станками сводится к сортировке деталей с определённой функцией сравнения элементов. Таким образом, асимптотика решения составляет $O(n \log n)$.

Реализация

Реализуем второй вариант описанного выше алгоритма, когда детали сортируются по минимуму из (a_i, b_i), и затем отправляются в начало либо в конец текущего списка.

```
struct item {
        int a, b, id;

        bool operator< (item p) const {
                return min(a,b) < min(p.a,p.b);
        }
};


sort (v.begin(), v.end());
vector<item> a, b;
for (int i=0; i<n; ++i)
        (v[i].a<=v[i].b ? a : b) .push_back (v[i]);
a.insert (a.end(), b.rbegin(), b.rend());

int t1=0, t2=0;
for (int i=0; i<n; ++i) {
        t1 += a[i].a;
        t2 = max(t2,t1) + a[i].b;
}
```

Здесь все детали хранятся в виде структур item, каждая из которых содержит значения a и b и исходный номер детали.

Детали сортируются, затем распределяются по спискам a (это те детали, которые были отправлены в начало очереди) и b (те, что были отправлены в конец). После этого два списка объединяются (причём второй список берётся в обратном порядке), и затем по найденному порядку вычисляется искомое минимальное время: поддерживаются две переменные t_1 и t_2 — время освобождения первого и второго станка соответственно.

Литература

- S.M. Johnson. Optimal two- and three-stage production schedules with setup

times included [1954]

- M.R. Garey. The Complexity of Flowshop and Jobshop Scheduling [1976]

Оптимальный выбор заданий при известных временах завершения и длительностях выполнения

Пусть дан набор заданий, у каждого задания известен момент времени, к которому это задание нужно завершить, и длительность выполнения этого задания. Процесс выполнения какого-либо задания нельзя прерывать до его завершения. Требуется составить такое расписание, чтобы выполнить наибольшее число заданий.

Решение

Алгоритм решения — **жадный** (greedy). Отсортируем все задания по их крайнему сроку, и будем рассматривать их по очереди в порядке убывания крайнего срока. Также создадим очередь q, в которую мы будем постепенно помещать задания, и извлекать из очереди задание с наименьшим временем выполнения (например, можно использовать set или priority_queue). Изначально q пустая.

Пусть мы рассматриваем i-ое задание. Сначала поместим его в q. Рассмотрим отрезок времени между сроком завершения i-го задания и сроком завершения $i-1$-го задания — это отрезок некоторой длины T. Будем извлекать из q задания (в порядке увеличения оставшегося времени их выполнения) и помещать на выполнение в этом отрезке, пока не заполним весь отрезок T. Важный момент — если в какой-то момент времени очередное извлечённое из структуры задание можно успеть частично выполнить в отрезке T, то мы выполняем это задание частично — именно настолько, насколько это возможно, т.е. в течение T единиц времени, а оставшуюся часть задания помещаем обратно в q.

По окончании этого алгоритма мы выберем оптимальное решение (или, по крайней мере, одно из нескольких решений). Асимптотика решения — $O(n \log n)$.

Реализация

```
int n;
vector < pair<int,int> > a; // задания в виде пар (крайний срок, длительность)
... чтение n и a ...

sort (a.begin(), a.end());

typedef set < pair<int,int> > t_s;
t_s s;
vector<int> result;
for (int i=n-1; i>=0; --i) {
        int t = a[i].first - (i ? a[i-1].first : 0);
        s.insert (make_pair (a[i].second, i));
        while (t && !s.empty()) {
                t_s::iterator it = s.begin();
                if (it->first <= t) {
                        t -= it->first;
                        result.push_back (it->second);
                }
                else {
                        s.insert (make_pair (it->first - t, it->second));
                        t = 0;
                }
                s.erase (it);
        }
}

for (size_t i=0; i<result.size(); ++i)
        cout << result[i] << ' ';
```

Задача Иосифа

Условие задачи. Даны натуральные n и k. По кругу выписывают все натуральные числа от 1 до n. Сначала отсчитывают k-ое число, начиная с первого, и удаляют его. Затем от него отсчитывают k чисел и k-ое удаляют, и т. д. Процесс останавливается, когда остаётся одно число. Требуется найти это число.

Задача была поставлена **Иосифом Флавием** (Flavius Josephus) ещё в 1 веке (правда, в несколько более узкой формулировке: при $k = 2$).

Решать эту задачу можно моделированием. Простейшее моделирование будет работать $O(n^2)$. Используя Дерево отрезков, можно произвести моделирование за $O(n \log n)$.

Решение за $O(n)$

Попытаемся найти закономерность, выражающую ответ для задачи $J_{n,k}$ через решение предыдущих задач.

С помощью моделирования построим таблицу значений, например, такую:

$$
\begin{array}{c}
n \setminus k \\
1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \\ 9 \\ 10
\end{array}
\begin{array}{c}
\begin{array}{cccccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10
\end{array} \\
\begin{pmatrix}
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
2 & 1 & 2 & 1 & 2 & 1 & 2 & 1 & 2 & 1 \\
3 & 3 & 2 & 2 & 1 & 1 & 3 & 3 & 2 & 2 \\
4 & 1 & 1 & 2 & 2 & 3 & 2 & 3 & 3 & 4 \\
5 & 3 & 4 & 1 & 2 & 4 & 4 & 1 & 2 & 4 \\
6 & 5 & 1 & 5 & 1 & 4 & 5 & 3 & 5 & 2 \\
7 & 7 & 4 & 2 & 6 & 3 & 5 & 4 & 7 & 5 \\
8 & 1 & 7 & 6 & 3 & 1 & 4 & 4 & 8 & 7 \\
9 & 3 & 1 & 1 & 8 & 7 & 2 & 3 & 8 & 8 \\
10 & 5 & 4 & 5 & 3 & 3 & 9 & 1 & 7 & 8
\end{pmatrix}
\end{array}
$$

И здесь достаточно отчётливо видна следующая **закономерность**:

$$
\begin{aligned}
&J_{n,k} = \left(J_{(n-1),k} + k - 1\right) \ \% \ n + 1 \\
&J_{1,k} = 1
\end{aligned}
$$

Здесь 1-индексация несколько портит элегантность формулы, если нумеровать позиции с нуля, то получится очень наглядная формула:

$$
J_{n,k} = \left(J_{(n-1),k} + k\right) \ \% \ n = \sum_{i=1}^{n} k \ \% \ i
$$

Итак, мы нашли решение задачи Иосифа, работающее за $O(n)$ операций.

Простая **рекурсивная реализация** (в 1-индексации):

```
int joseph (int n, int k) {
        return n>1 ? (joseph (n-1, k) + k - 1) % n + 1 : 1;
}
```

Нерекурсивная форма:

```
int joseph (int n, int k) {
        int res = 0;
        for (int i=1; i<=n; ++i)
                res = (res + k) % i;
        return res + 1;
}
```

Решение за $O(k \log n)$

Для сравнительно небольших k можно придумать более оптимальное решение, чем рассмотренное выше рекурсивное решение за $O(n)$. Если k небольшое, то даже интуитивно понятно, что тот алгоритм делает много лишних действий: серьёзные изменения происходят, только когда происходит взятие по модулю n, а до этого момента алгоритм просто несколько раз прибавляет к ответу число k. Соответственно, можно избавиться от этих ненужных шагов,

Небольшая возникающая при этом сложность заключается в том, что после удаления этих чисел у нас получится задача с меньшим n, но стартовой позицией не в первом числе, а где-то в другом месте. Поэтому, вызвав рекурсивно себя от задачи с новым n, мы затем должны аккуратно перевести результат в нашу систему нумерации из его собственной.

Также отдельно надо разбирать случай, когда n станет меньше k — в этом случае вышеописанная оптимизация выродится в бесконечный цикл.

Реализация (для удобства в 0-индексации):

```
int joseph (int n, int k) {
        if (n == 1)  return 0;
        if (k == 1)  return n-1;
        if (k > n)  return (joseph (n-1, k) + k) % n;
        int cnt = n / k;
        int res = joseph (n - cnt, k);
        res -= n % k;
        if (res < 0)  res += n;
        else  res += res / (k - 1);
        return res;
}
```

Оценим **асимптотику** этого алгоритма. Сразу заметим, что случай $n < k$ разбирается у нас старым решением, которое отработает в данном случае за $O(k)$. Теперь рассмотрим сам алгоритм. Фактически, на каждой его итерации вместо n чисел мы получаем примерно $n\left(1-\frac{1}{k}\right)$ чисел, поэтому общее число x итераций алгоритма примерно можно найти из уравнения:

$$n\left(1-\frac{1}{k}\right)^x = 1,$$

логарифмируя его, получаем:

$$\ln n + x \ln\left(1-\frac{1}{k}\right) = 0,$$
$$x = -\frac{\ln n}{\ln\left(1-\frac{1}{k}\right)},$$

пользуясь разложением логарифма в ряд Тейлора, получаем приблизительную оценку:

$$x \approx k \ln n$$

Таким образом, асимптотика алгоритма действительно $O(k \log n)$.

Аналитическое решение для $k = 2$

В этом частном случае (в котором и была поставлена эта задача Иосифом Флавием) задача решается значительно проще.

В случае чётного n получаем, что будут вычеркнуты все чётные числа, а потом останется задача для $\frac{n}{2}$, тогда ответ для n будет получаться из ответа для $\frac{n}{2}$ умножением на два и вычитанием единицы (за счёт сдвига позиций):

$$J_{2n,2} = 2J_{n,2} - 1$$

Аналогично, в случае нечётного n будут вычеркнуты все чётные числа, затем первое число, и останется задача для $\frac{n-1}{2}$, и с учётом сдвига позиций получаем вторую формулу:

$$J_{2n+1,2} = 2J_{n,2} + 1$$

При реализации можно непосредственно использовать эту рекуррентную зависимость. Можно эту закономерность перевести в другую форму: $J_{n,2}$ представляют собой последовательность всех нечётных чисел, "перезапускающуюся" с единицы всякий раз, когда n оказывается степенью двойки. Это можно записать и в виде одной формулы:

$$J_{n,2} = 1 + 2\left(n - 2^{\lfloor \log_2 n \rfloor}\right)$$

Аналитическое решение для $k > 2$

Несмотря на простой вид задачи и большое количество статей по этой и смежным задачам, простого аналитического представления решения задачи Иосифа до сих пор не найдено. Для небольших k выведены некоторые формулы, но, по-видимому, все они труднопримениимы на практике (например, см. Halbeisen, Hungerbuhler "The Josephus Problem" и Odlyzko, Wilf "Functional iteration and the Josephus problem").

Игра Пятнашки: существование решения

Напомним, что игра представляет собой поле 4 на 4, на котором расположены 15 фишек, пронумерованных числами от 1 до 15, а одно поле оставлено пустым. Требуется, передвигая на каждом шаге какую-либо фишку на свободную позицию, прийти в конце концов к следующей позиции:

$$\begin{matrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & \bigcirc \end{matrix}$$

Игру Пятнашки ("15 puzzle") изобрёл в 1880 г. Нойес Чэпман (Noyes Chapman).

Существование решения

Здесь мы рассмотрим такую задачу: по данной позиции на доске сказать, существует ли последовательность ходов, приводящая к решению, или нет.

Пусть дана некоторая позиция на доске:

$$\begin{matrix} a_1 & a_2 & a_3 & a_4 \\ a_5 & a_6 & a_7 & a_8 \\ a_9 & a_{10} & a_{11} & a_{12} \\ a_{13} & a_{14} & a_{15} & a_{16} \end{matrix}$$

где один из элементов равен нулю и обозначает пустую клетку $a_z = 0$.

Рассмотрим перестановку:

$$a_1 a_2 \ldots a_{z-1} a_{z+1} \ldots a_{15} a_{16}$$

(т.е. перестановка чисел, соответствующая позиции на доске, без нулевого элемента)

Обозначим через N количество инверсий в этой перестановке (т.е. количество таких элементов a_i и a_j, что $i < j$, но $a_i > a_j$).

Далее, пусть K — номер строки, в которой находится пустой элемент (т.е. в наших обозначениях $K = (z-1) \text{ div } 4 + 1$).

Тогда, **решение существует тогда и только тогда**, **когда** $N + K$ **чётно**.

Реализация

Проиллюстрируем указанный выше алгоритм с помощью программного кода:

```
int a[16];
for (int i=0; i<16; ++i)
        cin >> a[i];

int inv = 0;
for (int i=0; i<16; ++i)
        if (a[i])
                for (int j=0; j<i; ++j)
                        if (a[j] > a[i])
                                ++inv;
for (int i=0; i<16; ++i)
        if (a[i] == 0)
                inv += 1 + i / 4;

puts ((inv & 1) ? "No Solution" : "Solution Exists");
```

Доказательство

Джонсон (Johnson) в 1879 г. доказал, что если $N + K$ нечётно, то решения не существует, а Стори (Story) в том же году доказал, что все позиции, для которых $N + K$ чётно, имеют решение.

Однако оба эти доказательства были достаточно сложны.

В 1999 г. Арчер (Archer) предложил значительно более простое доказательство (скачать его статью можно здесь).

Дерево Штерна-Броко. Ряд Фарея

Дерево Штерна-Броко

Дерево Штерна-Броко — это изящная конструкция, позволяющая построить множество всех неотрицательных дробей. Она была независимо открыта немецким математиком Морицем Штерном (Moritz Stern) в 1858 г. и французским часовщиком Ахиллом Броко (Achille Brocot) в 1861 г. Впрочем, по некоторым данным, эта конструкция была открыта ещё древнегреческим учёным Эратосфеном (Eratosthenes).

На **нулевой** итерации у нас есть две дроби:

$$\frac{0}{1}, \frac{1}{0}$$

(вторая величина, строго говоря, дробью не является; её можно понимать как несократимую дробь, обозначающую бесконечность)

Дальше, на каждом **последующей** итерации берётся этот список дробей и между каждыми двумя соседними дробями $\frac{a}{b}$ и $\frac{c}{d}$ вставляется их **медианта**, т.е. дробь $\frac{a+c}{b+d}$.

Так, на первой итерации текущее множество будет таким:

$$\frac{0}{1}, \frac{1}{1}, \frac{1}{0}$$

На второй:

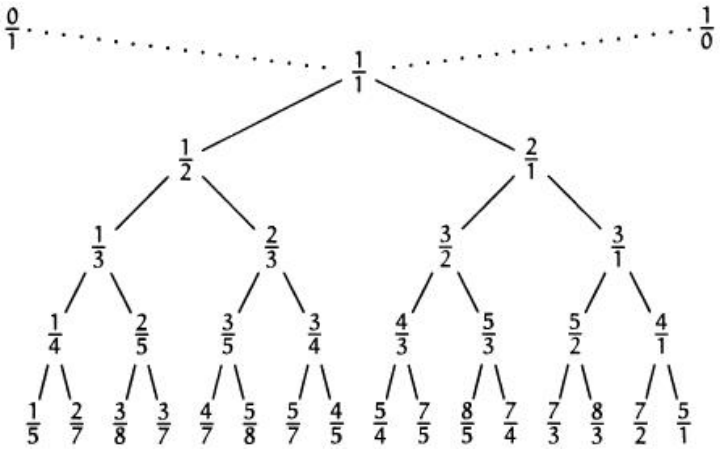
$$\frac{0}{1}, \frac{1}{2}, \frac{1}{1}, \frac{2}{1}, \frac{1}{0}$$

На третьей:

$$\frac{0}{1}, \frac{1}{3}, \frac{1}{2}, \frac{2}{3}, \frac{1}{1}, \frac{3}{2}, \frac{2}{1}, \frac{3}{1}, \frac{1}{0}$$

Продолжая этот процесс до **бесконечности**, утверждается, можно получить множество **всех** неотрицательных дробей. Более того, все получаемые дроби будут **различными** (т.е. в текущем множестве каждая дробь встречается не более одного раза), **несократимыми** (числители и знаменатели будут получаться взаимно простыми). Наконец, все дроби будут автоматически **упорядоченными** по возрастанию. Доказательство всех этих замечательных свойств дерева Штерна-Броко будет приведено чуть ниже.

Осталось только привести изображение самого дерева Штерна-Броко (пока мы описывали его с помощью меняющегося множества). В корне этого бесконечного дерева находится дробь $\frac{1}{1}$, а слева и справа от дерева находятся дроби $\frac{0}{1}$ и $\frac{1}{0}$. Любая вершина дерева имеет двух сыновей, каждый из которых получается как медианта своего левого предка и правого предка:



Доказательство

Упорядоченность. Она доказывается очень просто: заметим, что медианта двух дробей всегда находится

между ними, т.е.:

$$\frac{a}{b} \le \frac{a+c}{b+d} \le \frac{c}{d}$$

при условии, что

$$\frac{a}{b} \le \frac{c}{d}$$

Доказывается это просто приведением трёх дробей к общему знаменателю.

Поскольку на нулевой итерации упорядоченность имела место, то она будет сохраняться и на каждой новой итерации.

Несократимость. Для этого покажем, что на любой итерации для любых двух соседних в списке дробей $\frac{a}{b}$ и $\frac{c}{d}$ выполняется:

$$bc - ad = 1$$

Действительно, вспоминая Диофантовы уравнения с двумя неизвестными ($\boxed{ax + by = c}$), получаем из этого утверждения, что $\gcd(a, b) = \gcd(c, d) = 1$, что нам и требуется.

Итак, нам надо доказать истинность утверждения $bc - ad = 1$ на любой итерации. Докажем его также по индукции. На нулевой итерации это свойство выполнялось (в чём нетрудно убедиться). Теперь пусть оно было выполнено на предыдущей итерации, покажем, что оно выполнено на текущей итерации. Для этого надо рассмотреть тройку дробей-соседей в новом списке:

$$\frac{a}{b}, \frac{a+c}{b+d}, \frac{c}{d}$$

Для них условия принимают вид:

$$b(a+c) - a(b+d) = 1,$$
$$c(b+d) - d(a+c) = 1$$

Однако истинность этих условий очевидна, при условии истинности $bc - ad = 1$. Таким образом, действительно, это свойство выполнено и на текущей итерации, что и требовалось доказать.

Наличие всех дробей. Доказательство этого свойства тесно связано с алгоритмом нахождения дроби в дереве Штерна-Броко. Учитывая, что в дереве Штерна-Броко все дроби упорядочены, получаем, что для любой вершины дерева в её левом поддереве находятся дроби, меньшие её, а в правом — большие её. Отсюда получаем и очевидный алгоритм поиска какой-либо дроби в дереве Штерна-Броко: вначале мы находимся в корне; сравниваем нашу дробь с дробью, записанной в текущей вершине: если наша дробь меньше, то переходим в левое поддерево, если наша дробь больше — переходим в правое, а если совпадает — нашли дробь, поиск завершён.

Чтобы доказать, что бесконечное дерево Штерна-Броко содержит все дроби, достаточно показать, что этот алгоритм поиска дроби завершится за конечное число шагов для любой заданной дроби. Этот алгоритм можно понимать так: у нас есть текущий отрезок $\left[\frac{a}{b}; \frac{c}{d}\right]$, в котором мы ищем нашу дробь $\frac{x}{y}$. Изначально $\frac{a}{b} = \frac{0}{1}, \frac{c}{d} = \frac{1}{0}$. На каждом шаге дробь $\frac{x}{y}$ сравнивается с медиантой концов отрезка, т.е. с $\frac{a+c}{b+d}$, и в зависимости от этого мы либо останавливаем поиск, либо переходим в левую или правую часть отрезка. Если бы алгоритм поиска дроби работал бесконечно долго, то следующие условия были бы выполнены на каждой итерации:

$$\frac{a}{b} < \frac{x}{y} < \frac{c}{d}$$

Но их можно переписать в таком виде:

$$bx - ay \ge 1,$$
$$cy - dx \ge 1$$

(здесь использовалось то, что они целочисленны, поэтому из > 0 следует ≥ 1)

Тогда, умножая первое на $c + d$, а второе — на $a + b$, и складывая их, получаем:

$$(c+d)(bx - ay) + (a+b)(cy - dx) \ge a + b + c + d$$

Раскрывая скобки слева и учитывая, что $bc - ad = 1$ (см. доказательство предыдущего свойства), окончательно получаем:

$$x + y \ge a + b + c + d$$

А поскольку на каждой итерации хотя бы одна из переменных a, b, c, d строго возрастает, то процесс поиска дроби $\frac{x}{y}$ будет содержать не более $x + y$ итераций, что и требовалось доказать.

Алгоритм построения дерева

Чтобы построить любое поддерево дерева Штерна-Броко, достаточно знать только левого и правого предков. Изначально, на первом уровне, левым предком является $\frac{0}{1}$, а правым — $\frac{1}{0}$. По ним можно вычислить дробь в текущей вершине, а затем запуститься от левого и правого сыновей (левому сыну передав себя в качестве правого предка, а правому сыну — в качестве левого предка).

Псевдокод этой процедуры, пытающийся построить всё бесконечное дерево:

```
void build (int a = 0, int b = 1, int c = 1, int d = 0, int level = 1) {
        int x = a+c,  y = b+d;
        ... вывод текущей дроби x/y на уровне дерева level
        build (a, b, x, y, level + 1);
        build (x, y, c, d, level + 1);
}
```

Алгоритм поиска дроби

Алгоритм поиска дроби был уже описан при доказательства того, что дерево Штерна-Броко содержит все дроби, повторим его здесь. Этот алгоритм — фактически алгоритм бинарного поиска, или алгоритм поиска заданного значения в бинарном дереве поиска. Изначально мы стоим в корне дерева. Стоя в текущей вершине, мы сравниваем нашу дробь с дробью в текущей вершине. Если они совпадают, то процесс останавливаем — мы нашли дробь в дереве. Иначе, если наша дробь меньше дроби в текущей вершине, то переходим в левого сына, иначе — в правого.

Как было доказано в свойстве о том, что дерево Штерна-Броко содержит все неотрицательные дроби, при поиске дроби $\frac{x}{y}$ алгоритм совершит не более $x + y$ итераций.

Приведём реализацию, которая возвращает путь до вершины, содержащей заданную дробь $\frac{x}{y}$, возвращая его в виде последовательности символов 'L'/'R': если текущий символ равен 'L', то это обозначает переход в дереве в левого сына, а иначе — в правого (изначально мы стоим в корне дерева, т.е. в вершине с дробью $\frac{1}{1}$). На самом деле, такая последовательность символов, существующая и однозначно определяющая любую неотрицательную дробь, называется **системой счисления Штерна-Броко**.

```
string find (int x, int y, int a = 0, int b = 1, int c = 1, int d = 0) {
        int m = a+c,  n = b+d;
        if (x == m && y == n)
                return "";
        if (x * n < y * m)
                return 'L' + find (x, y, a, b, m, n);
        else
                return 'R' + find (x, y, m, n, c, d);
}
```

Иррациональным числам в системе счисления Штерна-Броко будут соответствовать бесконечные последовательности символов; если известна какая-то наперёд заданная точность, то можно ограничиться некоторым префиксом этой бесконечной последовательности. В процессе этого бесконечного поиска иррациональной дроби в дереве Штерна-Броко алгоритм будет каждый раз находить простую дробь (с постепенно возрастающими знаменателями), обеспечивающую лучшее приближение этого иррационального числа (это применение как раз важно в часовой технике, и в связи с этим Ахилл Броко и открыл это дерево).

Последовательность Фарея

Последовательностью Фарея порядка n называется множество всех несократимых дробей между 0 и 1, знаменатели которых не превосходят n, причём дроби упорядочены в порядке возрастания.

Эта последовательность названа в честь английского геолога Джона Фарея (John Farey), который попытался в 1816 г. доказать, что в ряде Фарея любая дробь является медиантой двух соседних. Насколько известно, его доказательство было неверным, а правильное доказательство предложил несколько позже Коши (Cauchy). Впрочем, ещё в 1802 г. математик Харос (Haros) в одной из своих работ пришёл практически к тем же результатам.

Последовательности Фарея обладают и множеством собственных интересных свойств, однако наиболее очевидна их **связь с деревом Штерна-Броко**: фактически, последовательность Фарея получается удалением некоторых ветвей из дерева. Или можно говорить, что для получения последовательности Фарея нужно взять множество дробей, получаемое при построении дерева Штерна-Броко на бесконечной итерации, и оставить в этом множестве только дроби со знаменателями, не превосходящими n и числителями, не превосходящими знаменатели.

Из алгоритма построения дерева Штерна-Броко следует и аналогичный **алгоритм** для последовательностей Фарея.

На нулевой итерации включим в множество только дроби $\frac{0}{1}$ и $\frac{1}{1}$. На каждой следующей итерации мы между каждыми двумя соседним дробями вставляем их медианту, если её знаменатель не превосходит n. Рано или поздно в множестве перестанут происходить какие-либо изменения, и процесс можно останавливать — мы нашли искомую последовательность Фарея.

Вычислим **длину** последовательности Фарея. Последовательность Фарея порядка n содержит все элементы последовательности Фарея порядка $n-1$, а также все несократимые дроби со знаменателями, равными n, но это количество, как известно, равно $\phi(n)$. Таким образом, длина L_n последовательности Фарея порядка n выражается по формуле:

$$L_n = L_{n-1} + \phi(n)$$

или, раскрывая рекурсию:

$$L_n = 1 + \sum_{k=1}^{n} \phi(k)$$

Литература

- Роналд Грэхем, Дональд Кнут, Орен Паташник. **Конкретная математика**. **Основание информатики** [1998]

Поиск подотрезка массива с максимальной / минимальной суммой

Здесь мы рассмотрим задачу о поиске подотрезка массива с максимальной суммой ("maximum subarray problem" на английском), а также некоторые её вариации (в том числе алгоритм решения варианта этой задачи в режиме онлайн — описанный автором алгоритма — KADR (Ярослав Твердохлеб)).

Постановка задачи

Дан массив чисел $a[1 \dots n]$. Требуется найти такой его подотрезок $a[l \dots r]$, что сумма на нём **максимальна**:

$$\max_{1 \le l \le r \le n} \sum_{i=l}^{r} a[i].$$

Например, если бы все числа массива $a[]$ были бы неотрицательными, то в качестве ответа можно было бы взять весь массив. Решение нетривиально, когда массив может содержать как положительные, так и отрицательные числа.

Понятно, что задача о поиске **минимального** подотрезка — по сути та же самая, достаточно лишь изменить знаки всех чисел на противоположные.

Алгоритм 1

Здесь мы рассмотрим практически очевидный алгоритм. (Дальше мы рассмотрим другой алгоритм, который чуть сложнее придумать, однако его реализация получается ещё короче.)

Описание алгоритма

Алгоритм весьма прост.

Введём для удобства **обозначение**: $s[i] = \sum_{j=1}^{i} a[j]$. Т.е. массив $s[i]$ — это массив частичных сумм массива $a[]$. Также положим значение $s[0] = 0$.

Будем теперь **перебирать** индекс $r = 1 \dots n$, и научимся для каждого текущего значения r быстро находить оптимальное l, при котором достигается максимальная сумма на подотрезке $[l; r]$.

Формально это означает, что нам надо для текущего r найти такое l (не превосходящее r), чтобы величина $s[r] - s[l-1]$ была максимальной. После тривиального преобразования мы получаем, что нам надо найти минимальное значение в массиве $s[]$ минимум на отрезке $[0; r-1]$.

Отсюда мы сразу получаем алгоритм решения: мы просто будем хранить, где в массиве $s[]$ находится текущий минимум. Используя этот минимум, мы за $O(1)$ находим текущий оптимальный индекс l, а при переходе от текущего индекса r к следующему мы просто обновляем этот минимум.

Очевидно, этот алгоритм работает за $O(n)$ и асимптотически оптимален.

Реализация

Для реализации нам даже не понадобится явно хранить массив частичных сумм $s[]$ — от него нам будет требоваться только текущий элемент.

Реализация приводится в 0-индексированных массивах, а не в 1-нумерации, как было описано выше.

Приведём сначала решение, которое находит просто численный ответ, не находя индексы искомого отрезка:

```
int ans = a[0],
        sum = 0,
        min_sum = 0;
for (int r=0; r<n; ++r) {
        sum += a[r];
        ans = max (ans, sum - min_sum);
        min_sum = min (min_sum, sum);
}
```

Теперь приведём полный вариант решения, который параллельно с числовым решением находит границы искомого отрезка:

```
int ans = a[0],
        ans_l = 0,
        ans_r = 0,
        sum = 0,
        min_sum = 0,
        min_pos = -1;
for (int r=0; r<n; ++r) {
        sum += a[r];

        int cur = sum - min_sum;
        if (cur > ans) {
                ans = cur;
                ans_l = min_pos + 1;
                ans_r = r;
        }

        if (sum < min_sum) {
                min_sum = sum;
                min_pos = r;
        }
}
```

Алгоритм 2

Здесь мы рассмотрим другой алгоритм. Его чуть сложнее понять, но зато он более элегантен, чем приведённый выше, и реализуется чуть-чуть короче. Этот алгоритм был предложен Джеем Каданом (Jay Kadane) в 1984 г.

Описание алгоритма

Сам **алгоритм** выглядит следующим образом. Будем идти по массиву и накапливать в некоторой переменной s текущую частичную сумму. Если в какой-то момент s окажется отрицательной, то мы просто присвоим $s = 0$. Утверждается, что максимум из всех значений переменной s, случившихся за время работы, и будет ответом на задачу.

Докажем этот алгоритм.

В самом деле, рассмотрим первый момент времени, когда сумма s стала отрицательной. Это означает, что, стартовав с нулевой частичной суммы, мы в итоге пришли к отрицательной частичной сумме — значит, и весь этот префикс массива, равно как и любой его суффикс имеют отрицательную сумму. Следовательно, от всего этого префикса массива в дальнейшем не может быть никакой пользы: он может дать только отрицательную прибавку к ответу.

Однако этого недостаточно для доказательства алгоритма. В алгоритме мы, фактически, ограничиваемся в поиске ответа только такими отрезками, которые начинаются непосредственно после мест, когда случалось $s < 0$.

Но, в самом деле, рассмотрим произвольный отрезок $[l; r]$, причём l не находится в такой "критической" позиции (т. е. $l > p + 1$, где p — последняя такая позиция, в которой $s < 0$). Поскольку последняя критическая позиция находится строго раньше, чем в $l - 1$, то получается, что сумма $a[p + 1 \ldots l - 1]$ неотрицательна. Это означает, что, сдвинув l в позицию $p + 1$, мы увеличим ответ или, в крайнем случае, не изменим его.

Так или иначе, но получается, что действительно при поиске ответа можно ограничиться только отрезками, начинающимися сразу после позиций, в которых оказывалось $s < 0$. Это доказывает правильность алгоритма.

Реализация

Как и в алгоритме 1, приведём сначала упрощённую реализацию, которая ищет только числовой ответ, не находя границ искомого отрезка:

```
int ans = a[0],
        sum = 0;
for (int r=0; r<n; ++r) {
        sum += a[r];
        ans = max (ans, sum);
        sum = max (sum, 0);
}
```

Полный вариант решения, с поддержанием индексов-границ искомого отрезка:

```
int ans = a[0],
        ans_l = 0,
        ans_r = 0,
        sum = 0,
        minus_pos = -1;
for (int r=0; r<n; ++r) {
        sum += a[r];

        if (sum > ans) {
                ans = sum;
                ans_l = minus_pos + 1;
                ans_r = r;
        }

        if (sum < 0) {
                sum = 0;
                minus_pos = r;
        }
}
```

Смежные задачи

Поиск максимального / минимального подотрезка с ограничениями

Если в условии задачи на искомый отрезок $[l; r]$ накладываются дополнительные ограничения (например, что длина $r - l + 1$ отрезка должна находиться в заданных пределах), то описанный алгоритм скорее всего легко обобщается на эти случаи — так или иначе, задача будет по-прежнему заключаться в поиске минимума в массиве $s[]$ при заданных дополнительных ограничениях.

Двумерный случай задачи: поиск максимальной / минимальной подматрицы

Описанная в данной статье задача естественно обобщается на большие размерности. Например, в двумерном случае она превращается в поиск такой подматрицы $[l_1 \dots r_1; l_2 \dots r_2]$ заданной матрицы, которая имеет максимальную сумму чисел в ней.

Из описанного выше решения для одномерного случая **легко получить** решение за $O(n^3)$: переберём l_1 и r_1, и посчитаем массив сумм с l_1 по r_1 в каждой строке матрицы; мы пришли к одномерной задаче поиска индексов l_2 и r_2 в этом массиве, которую уже можно решать за линейное время.

Более быстрые алгоритмы решения этой задачи хотя и известны, однако они не сильно быстрее $O(n^3)$, и при этом весьма сложны (настолько сложны, что по скрытой константе многие из них уступают тривиальному алгоритму при всех разумных ограничениях). По всей видимости, лучший из известных алгоритмов работает за $O\left(n^3 \frac{\log^3 \log n}{\log^2 n}\right)$ (T. Chan 2007 "More algorithms for all-pairs shortest paths in weighted graphs").

Этот алгоритм Chan, а также многие другие результаты в данной области на самом деле описывают **быстрое умножение** матриц (где под умножением матриц подразумевается модифицированное умножение: вместо сложения используется минимум, а вместо умножения — сложение). Дело в том, что задача о поиске подматрицы с наибольшей суммой сводится к задаче о поиске кратчайших путей между всеми парами вершин, а эта задача, в свою очередь — сводится к такому умножению матриц.

Поиск подотрезка с максимальной / минимальной средней суммой

Эта задача заключается в том, что надо найти такой отрезок $[l; r]$, чтобы среднее значение на нём было максимальным:

$$\max_{l \le r} \frac{1}{r - l + 1} \sum_{i=l} ra[i].$$

Конечно, если на искомый отрезок $[l; r]$ по условию не наложено других условий, то решением всегда будет являться отрезок длины 1 в точке-максимуме массива. Задача имеет смысл, только если имеются **дополнительные ограничения** (например, длина искомого отрезка ограничена снизу).

В таком случае применим **стандартный приём** при работе с задачами о среднем значении: будем подбирать искомую максимальную среднюю величину **двоичным поиском**.

Для этого нам надо научиться решать такую подзадачу: дано число x, и надо проверить, есть ли подотрезок массива $a[]$ (конечно, удовлетворяющий всем дополнительным ограничениям задачи), на котором среднее значение больше x.

Чтобы решить эту подзадачу, отнимем x от каждого элемента массива $a[]$. Тогда наша подзадача фактически превращается в такую: есть или нет в данном массиве подотрезок положительной суммы. А эту задачу мы уже умеем решать.

Таким образом, мы получили решение за асимптотику $O(T(n)\log W)$, где W — требуемая точность, $T(n)$ — время решения подзадачи для массива длины n (которое может варьироваться в зависимости от конкретных накладываемых дополнительных ограничений).

Решение задачи в режиме онлайн

Условие задачи таково: дан массив из n чисел, а также дано число L. Поступают запросы вида (l, r), и в ответ на запрос требуется найти подотрезок отрезка $[l; r]$ длины не менее L с максимально возможным средним арифметическим.

Алгоритм решения этой задачи достаточно сложен. Автор данного алгоритма — KADR (Ярослав Твердохлеб) — описал данный алгоритм в своём сообщении на форуме.

Литература

C++

Visual C++, MFC

- Круглински, Уингоу, Шеферд. **Программирование на** Microsoft Visual C++ 6.0 **для профессионалов** (PDF[in RAR], 54.4 МБ)

C++

- ANSI. C++ International Standard (second edition, 2003-10-15) (PDF, 2.3 МБ)
- Эккель. **Философия** C++. **Введение в стандартный** C++ (2-**е изд**.) (DJVU, 6.4 МБ)
- Эккель, Эллисон. **Философия** C++. **Практическое программирование** (DJVU, 6.5 МБ)
- Саттер. **Решение сложных задач на** C++. 87 **головоломных примеров с решениями** (DJVU, 3.8 МБ)
- Саттер. **Новые сложные задачи на** C++. 40 **новых головоломных примеров с решениями** (DJVU, 3.6 МБ)
- Страуструп. **Язык программирования** C++ (2-**е**, **дополненное изд**.) (PDF, 2.9 МБ)
- Stroustrup. The C++ Programming Language (3rd edition) (PDF, 3.4 МБ)
- Abrahams, Gurtovoy. C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond (CHM, 0.62 МБ)
- Джосьютис. C++. **Стандартная библиотека**. **Для профессионалов** (DJVU, 4.8 МБ)
- Джосьютис. C++. **Стандартная библиотека**. **Для профессионалов** (CD **к книге**) (ZIP, 0.14 МБ)
- Vandervoorde, Josuttis. C++ Templates: The Complete Guide (CHM, 0.72 МБ)
- Sutter, Alexandrescu. C++ Coding Standards: 101 Rules, Guidelines, and Best Practices (CHM, 0.51 МБ)
- Голуб. **Веревка достаточной длины**, **чтобы выстрелить себе в ногу**. **Правила программирования на** C **и** C++ (PDF, 1.29 МБ)
- Meyers. Effective C++. More effective C++ (CHM, 2.0 МБ)
- Дьюхэрст. **Скользкие места** C++. **Как избежать проблем при проектировании и компиляции ваших программ** (DJVU, 9.3 МБ)
- Дьюхэрст. C++. **Священные знания** (DJVU, 6.7 МБ)

Алгоритмы

Фундаментальные пособия

- Кормен, Лейзерсон, Ривест, Штайн. **Алгоритмы**. **Построение и анализ** (2-**е зд**.) (DJVU, 18.3 МБ)
- Knuth. The Art of Computer Programming. Volume 1 (3rd edition) (DJVU, 6.0 МБ)
- Knuth. The Art of Computer Programming. Volume 2 (3rd edition) (DJVU, 7.6 МБ)
- Knuth. The Art of Computer Programming. Volume 3 (2nd edition) (DJVU, 7.7 МБ)
- Кормен, Лейзерсон, Ривест, Штайн. **Алгоритмы**. **Построение и анализ** (1-**е изд**.?) (PDF, 4.5 МБ)
- Седжвик. **Фундаментальные алгоритмы** (3-**я ред**.). **Части** 1-4 (DJVU, 15.0 МБ)
- Седжвик. **Фундаментальные алгоритмы** (3-**я ред**.). **Часть** 5 (DJVU, 16.7 МБ)
- Кнут. **Искусство программирования**. **Том** 1 (DJVU, 5.6 МБ)
- Кнут. **Искусство программирования**. **Том** 2 (DJVU, 6.1 МБ)
- Кнут. **Искусство программирования**. **Том** 3 (DJVU, 6.4 МБ)
- Грэхэм, Кнут, Паташник. **Конкретная математика** (DJVU, 8.9 МБ)
- Пападимитриу, Стайглиц. **Комбинаторная оптимизация**: **алгоритмы и сложность** (DJVU, 5.6 МБ)
- Motwani, Raghavan. Randomized Algorithms (DJVU, 4.4 МБ)
- Tucker. Computer Science Handbook (PDF, 27.0 МБ)
- Mehlhorn, Sanders. Algorithms and Data Structures: The Basic Toolbox (PDF, 2.0 МБ)

Олимпиадные задачи

- Меньшиков. **Олимпиадные задачи по программированию** (DJVU, 4.4 МБ)

- Меньшиков. **Олимпиадные задачи по программированию** (CD **к книге**) (ZIP, 4.0 МБ)
- Окулов. **Программирование в алгоритмах** (DJVU, 3.6 МБ)
- Долинский. **Решение сложных и олимпиадных задач по программированию** (DJVU, 2.9 МБ)
- Скиена, Ревилла. **Олимпиадные задачи по программированию** (DJVU, 5.3 МБ)

Строки

- Гасфилд. **Строки**, **деревья и последовательности в алгоритмах** (DJVU, 12.1 МБ)
- Smyth. Computing patterns in strings (DJVU, 26.4 МБ)
- Crochemore, Rytter. Jewels of Stringology (DJVU, 2.6 МБ)
- Crochemore, Hancart. Automata for matching patterns (pdf, 0.44 МБ)

Компиляция, интерпретация

- Aho, Lam, Sethi, Ullman. Compilers: Principles, Techniques and Tools (DJVU, 5.7 МБ)
- Mogensen. Basics of Compiler Design (PDF, 0.81 МБ)
- Пратт, Зелковиц. **Языки программирования**: **разработка и реализация** (4-**е изд**., 2002) (DJVU, 5.7 МБ)

Теория игр

- Conway. On Numbers and Games (DJVU, 2.1 МБ)

Алгебра, теория чисел

- Ribenboim. The New Book of Prime Number Records (DJVU, 11.0 МБ)
- Shoup. A Computational Introduction to Number Theory and Algebra (version 2) (PDF, 3.5 МБ)
- William H. Press, Saul A. Teukolsky, William T. Vetterling, Brian P. Flannery. Numerical Recipes: The Art of Scientific Computing (PDF, 20.4 МБ)

Вычислительная геометрия

- Препарата, Шеймос. **Вычислительная геометрия**. **Введение** (DJVU, 4.5 МБ)
- Андреева, Егоров. **Вычислительная геометрия на плоскости** (DPF, 0.61 МБ)
- Mount. Lecture notes for the course Computational Geometry (PDF, 0.77 МБ)
- de Berg, van Kreveld, Overmars, Schwarzkopf. Computational Geometry: Algorithms and Applications (2nd, revised edition) (DJVU, 3.7 МБ)
- Chen. Computational Geometry: Methods and Applications (PDF, 1.14 МБ)
- Скворцов. **Триангуляция Делоне и её применение** (PDF, 2.5 МБ)
- Miu. Voronoi Diagrams: lecture slides (PDF, 0.14 МБ)
- Held. Voronoi Diagram: slides (PDF, 1.35 МБ)

Графы

- Ahuja, Magnanti, Orlin. Network flows (DJVU, 13.8 МБ)
- Приезжев. **Задача о димерах и теорема Кирхгофа** (PDF, 1.18 МБ)
- Thorup. Unidirected Single-Source Shortest Paths with Positive Integer Weights in Linear Time (PPT, 1.10 МБ)
- Eppstein. Finding the K Shortest Paths (PDF, 0.18 МБ)
- Sokkalingham, Ahuja, Orlin. Inverse Spanning Tree Problems: Formulations and Algorithms (PDF, 0.07 МБ)
- Ahuja, Orlin. A Faster Algorithm for the Inverse Spanning Tree Problem (PDF, 0.10 МБ)
- Brander, Sinclair. A Comparative Study of K-Shortest Path Algorithms (PDF, 0.16 МБ)
- Gabow. An Efficient Implementation of Edmonds Maximum-Matching Algorithm (PDF, 2.7 МБ)
- Bender, Farach-Colton. The LCA Problem Revisited (PDF, 0.08 МБ)
- Майника. **Алгоритмы оптимизации на сетях и графах** (DJVU, 4.0 МБ)
- Mehlhorn, Uhrig. The minimum cut algorithm of Stoer and Wagner (PDF, 0.12 МБ)
- Оре. **Теория графов** (DJVU, 4.3 МБ)
- Харари. **Теория графов** (DJVU, 8.7 МБ)
- Stoer, Wagner. A Simple Min-Cut Algorithm (PDF, 0.20 МБ)
- Lovasz, Plummer. Matching theory (PDF, 9.9 МБ)
- Tutte. The Factorization of Linear Graphs (PDF, 0.47 МБ)

Комбинаторика

- Степанов. **Лемма Бернсайда и задачи о раскрасках** (DPF, 0.18 МБ)

- Харари. **Перечисление графов** (DJVU, 4.1 МБ)

Теория сложности

- Гэри, Джонсон. **Вычислительные машины и труднорешаемые задачи** (DJVU, 11.5 МБ)

Математика

- Aitken. Determinants and Matrices (PDF, 10.2 МБ)

Структуры данных

- Tarjan. Efficiency of a Good But Not Linear Set Union Algorithm (PDF, 0.63 МБ)
- Tarjan, Leeuwen. Worst-Case Analysis of Set Union Algorithms (PDF, 1.55 МБ)

Оптимизация

- Kaspersky. Code Optimization: Effective Memory Usage (CHM, 10.4 МБ)
- Kaspersky. Code Optimization: Effective Memory Usage (CD **к книге**) (ZIP, 4.6 МБ)
- Fog. Optimization Manuals (Optimizing software in C++, in assembly, processors microarchitecture) (last edited - 2008) (PDF[in ZIP], 2.9 МБ)
- Intel. Intel Architecture Optimization Manual (1997) (PDF, 0.49 МБ)

Java

Java

- Эккель. **Философия** Java (4-**е изд**.) (DJVU, 5.4 МБ)
- Хорстманн, Корнелл. Java 2. **Библиотека профессионала**. **Том** 1 (**Основы**) (7-**е изд**.) (DJVU, 10.5 МБ)
- Хорстманн, Корнелл. Java 2. **Библиотека профессионала**. **Том** 2 (**Тонкости программирования**) (7-**е изд**.) (DJVU, 13.2 МБ)
- Хорстманн, Корнелл. Java 2. **Библиотека профессионала** (7-**е изд**.) (CD **к книге**) (ZIP, 0.66 МБ)

TeX

TeX

- Кнут. **Всё про** TeX (DJVU, 17.1 МБ)
- Abrahams, Hargreaves, Berry. TeX for the Impatient (PDF, 1.36 МБ)

LaTeX

- Gratzer. Math into LaTeX. An Introduction to LaTeX and AMS-LaTeX (DJVU, 0.34 МБ)
- Oetiker. The Not So Short Introduction to LaTeX (version 4.26, 2008-Sep-25) (PDF, 2.3 МБ)

Об авторе

Бессменным автором сайта e-maxx.ru и всех статей по алгоритмам являюсь я, e-maxx, также известный как Максим Иванов :)

Все материалы сайта, в том числе и эта книга, выложены под лицензией Public Domain, т.е. являются общественным достоянием, не охраняются авторским правом, и могут распространяться абсолютно неограниченно. Главная цель этого ресурса — распространение информации и устранение препятствий на пути к этому.

Кроме меня в создании, корректировке и улучшении статей принимали участие множество людей, — спасибо Вам за бесчисленные замечания, предложения по улучшению, указания на неточности, и т.д. К сожалению, пока никто из сообщества не решился на написание полноценной статьи с описанием какого-либо алгоритма. Если пожелаете быть первым — добро пожаловать ;)

Связаться со мной можно по электронной почте: e-maxx@inbox.ru, или на форуме сайта e-maxx.