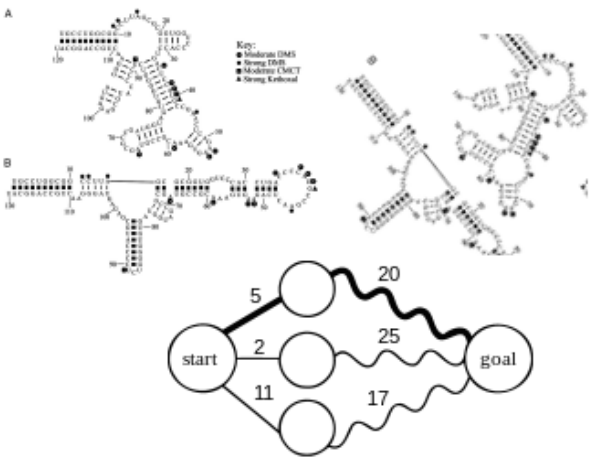
Динамическое программирование для начинающих

7 июня 2016 в 15:04, Статьи (https://tproger.ru/category/articles/) 15 минут 38 657



Динамическое программирование — тема, которой в рунете посвящено довольно мало статей, поэтому мы решили ею заняться. В этой статье будут разобраны классические задачи на последовательности, одномерную и двумерную динамику, будет дано обоснование решениям и описаны разные подходы к их реализации. Весь приведённый в статье код написан на Java.

О чём вообще речь? Что такое динамическое программирование?

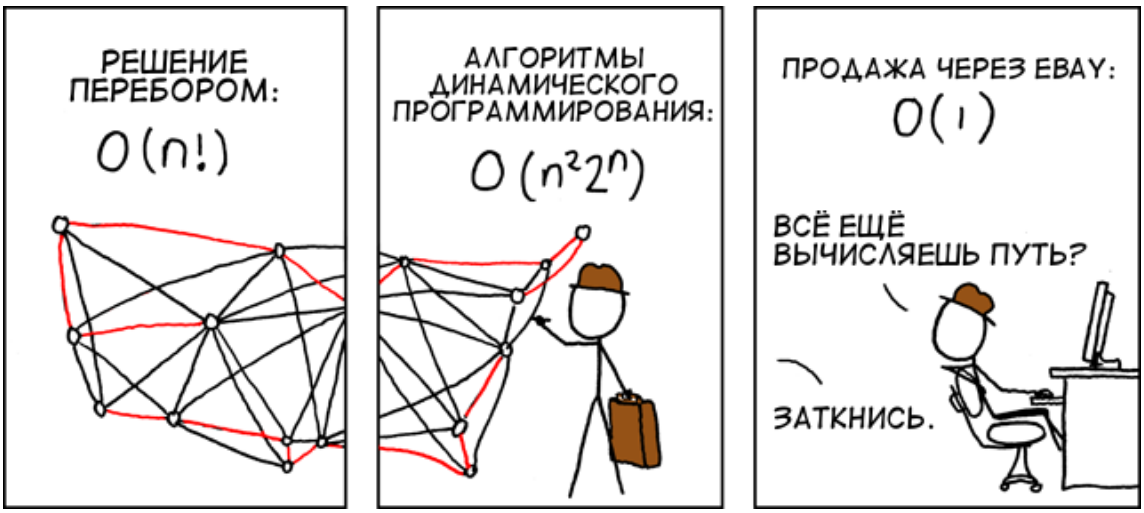
Динамическое программирование — метод решения задачи путём её разбиения на несколько одинаковых подзадач, рекуррентно связанных между собой. Самым простым примером будут числа Фибоначчи (https://ru.wikipedia.org/wiki/Числа_Фибоначчи) — чтобы вычислить некоторое число в этой последовательности, нам нужно сперва вычислить третье число, сложив первые два, затем четвёртое *таким же образом* на основе второго и третьего, и так далее (да, мы слышали про замкнутую формулу).

Хорошо, как это использовать?

Решение задачи динамическим программированием должно содержать следующее:

- **Зависимость элементов динамики друг от друга**. Такая зависимость может быть прямо дана в условии (так часто бывает, если это задача на числовые последовательности). В противном случае вы можете попытаться узнать какой-то известный числовой ряд (вроде тех же чисел Фибоначчи), вычислив первые несколько значений вручную. Если вам совсем не повезло – придётся думать

- **Значение начальных состояний**. В результате долгого разбиения на подзадачи вам необходимо свести функцию либо к уже известным значениям (как в случае с Фибоначчи — заранее определены первые два члена), либо к задаче, решаемой элементарно.



(https://cdn.tproger.ru/wp-content/uploads/2016/06/399_v1.png)

И что, мне для решения рекурсивный метод писать надо? Я слышал, они медленные.

Конечно, не надо, есть и другие подходы к реализации динамики. Разберём их на примере следующей задачи:

> Вычислить n-й член последовательности, заданной формулами:
> $a_{2n} = a_n + a_{n-1}$,
> $a_{2n+1} = a_n — a_{n-1}$,
> $a_0 = a_1 = 1$.

Идея решения

Здесь нам даны и начальные состояния ($a_0 = a_1 = 1$), и зависимости. Единственная сложность, которая может возникнуть — понимание того, что 2n — условие чётности числа, а 2n+1 — нечётности. Иными словами, нам нужно проверять, чётно ли число, и считать его в зависимости от этого по разным формулам.

Рекурсивное решение

Очевидная реализация состоит в написании следующего метода:

```
private static int f(int n){
        if(n==0 || n==1) return 1; // Проверка на начальное значение

        if(n%2==0){ //Проверка на чётность
                return f(n/2)+f(n/2-1); // Вычисляем по формуле для чётных инде
                                        //  ссылаясь на предыдущие значения
        }else{
                return f((n-1)/2)-f((n-1)/2-1);  // Вычисляем по формуле для не
                                                 //индексов, ссылаясь на предыду
        }
}
```

И она отлично работает, но есть нюансы. Если мы захотим вычислить `f(12)`, то метод будет вычислять сумму `f(6)+f(5)`. В то же время, `f(6)=f(3)+f(2)` и `f(5)=f(2)-f(1)`, т.е. значение `f(2)` мы будем вычислять дважды. Спасение от этого есть — мемоизация (кеширование значений).

Рекурсивное решение с кэшированием значений

Идея мемоизации очень проста — единожды вычисляя значение, мы заносим его в какую-то структуру данных. Перед каждым вычислением мы проверяем, есть ли вычисляемое значение в этой структуре, и если есть, используем его. В качестве структуры данных можно использовать массив, заполненный флаговыми значениями. Если значение элемента по индексу N равно значению флага, значит, мы его ещё не вычисляли. Это создаёт определённые трудности, т.к. значение флага не должно принадлежать множеству значений функции, которое не всегда очевидно. Лично я предпочитаю использовать хэш-таблицу (https://ru.wikipedia.org/wiki/Хеш-таблица) — все действия в ней выполняются за `O(1)`, что очень удобно. Однако, при большом количестве значений два числа могут иметь одинаковый хэш, что, естественно, порождает проблемы. В таком случае стоит использовать, например, красно-чёрное дерево (https://ru.wikipedia.org/wiki/Красно-чёрное_дерево).

Для уже написанной функции `f(int)` кэширование значений будет выглядеть следующим образом:

```
private static HashMap<Integer, Integer> cache = new HashMap<Integer, Integer>(

private static int fcashe(int n){
        if(!cache.containsKey(n)){//Проверяем, находили ли мы данное значение
            cache.put(n, f(n)); //Если нет, то находим и записываем в таблицу
        }
        return cache.get(n);
}
```

Не слишком сложно, согласитесь? Зато это избавляет от огромного числа операций. Платите вы за это лишним расходом памяти.

Последовательное вычисление

Теперь вернёмся к тому, с чего начали — рекурсия работает медленно. Не слишком медленно, чтобы это приносило действительные неприятности в настоящей жизни, но на соревнованиях по спортивному программированию каждая миллисекунда на счету.

Метод последовательного вычисления подходит, только если функция ссылается исключительно на элементы перед ней — это его основной, но не единственный минус. Наша задача этому условию удовлетворяет.

Суть метода в следующем: мы создаём массив на N элементов и последовательно заполняем его значениями. Вы, наверное, уже догадались, что таким образом мы можем вычислять в том числе те значения, которые для ответа не нужны. В значительной части задач на динамику этот факт можно опустить, так как для ответа часто бывают нужны как раз все значения. Например, при поиске наименьшего пути мы не можем не вычислять путь до какой-то точки, нам нужно пересмотреть все варианты. Но в нашей задаче нам нужно вычислять приблизительно $\log_2(N)$ значений (на практике больше), для 9223372036854775807-го элемента (`MaxLong/10`) нам потребуется 172 вычисления.

```
private static int f(int n){
if(n<2) return 1; //Может, нам и вычислять ничего не нужно?

int[] fs = int[n]; //Создаём массив для значений
fs[0]=fs[1]=1; //Задаём начальные состояния

for(int i=2; i<n; i++){
    if(i%2==0){ //Проверяем чётность
        fs[i]=fs[i/2]+fs[i/2-1];
    }else{
        fs[i]=fs[(i-1)/2]+fs[(i-1)/2-1]
    }
}

return fs[n-1];
}
```

Ещё одним минусом такого подхода является сравнительно большой расход памяти.

Создание стека индексов

Сейчас нам предстоит, по сути, написать свою собственную рекурсию. Идея состоит в следующем — сначала мы проходим «вниз» от N до начальных состояний, запоминая аргументы, функцию от которых нам нужно будет вычислять. Затем возвращаемся

«вверх», последовательно вычисляя значения от этих аргументов, в том порядке, который мы записали.

Зависимости вычисляются следующим образом:

```
LinkedList<Integer> stack = new LinkedList<Integer>();
stack.add(n);

{
LinkedList<Integer> queue = new LinkedList<Integer>();
//Храним индексы, для которых ещё не вычислены зависимости

queue.add(n);
int dum;
while(queue.size()>0){ //Пока есть что вычислять
    dum = queue.removeFirst();

    if(dum%2==0){ //Проверяем чётность
        if(dum/2>1){ //Если вычисленная зависимость не принадлежит начальным с
            stack.addLast(dum/2); //Добавляем в стек
            queue.add(dum/2); //Сохраняем, чтобы
                              //вычислить дальнейшие зависимости
        }
        if(dum/2-1>1){ //Проверяем принадлежность к начальным состояниям
            stack.addLast(dum/2-1); //Добавляем в стек
            queue.add(dum/2-1); //Сохрнаяем, чтобы
                                //вычислить дальнейшие зависимости
        }

    }else{
        if((dum-1)/2>1){ //Проверяем принадлежность к начальным состояниям
            stack.addLast((dum-1)/2); //Добавляем в стек
            queue.add((dum-1)/2); //Сохрнаяем, чтобы
                                  //вычислить дальнейшие зависимости
        }
        if((dum-1)/2-1>1){ //Проверяем принадлежность к начальным состояниям
            stack.addLast((dum-1)/2-1); //Добавляем в стек
            queue.add((dum-1)/2-1); //Сохрнаяем, чтобы
                                    //вычислить дальнейшие зависимости
        }
    }

/*
Конкретно для этой задачи есть более элегантный способ найти все зависимости,
здесь же показан достаточно универсальный
*/

}
}
```

Полученный размер стека – то, сколько вычислений нам потребуется сделать. Именно так я получил упомянутое выше число 172.

Спасибо за внимательность. Опечатка уже отправлена нашим редакторам.

Теперь мы поочередно извлекаем индексы и вычисляем для них значения по формулам – гарантируется, что все необходимые значения уже будут вычислены. Хранить будем как раньше – в хэш-таблице.

```
HashMap<Integer,Integer> values = new HashMap<Integer,Integer>();

values.put(0,1); //Важно добавить начальные состояния
//в таблицу значений
values.put(1,1);

while(stack.size()>0){
    int num = stack.removeLast();

    if(!values.containsKey(num)){ //Эту конструкцию
                                  //вы должны помнить с абзаца о кешировани
    if(num%2==0){ //Проверяем чётность
        int value = values.get(num/2)+values.get(num/2-1); //Вычисляем значени
        values.add(num, value); //Помещаем его в таблицу
    }else{
        int value = values.get((num-1)/2)-values.get((num-1)/2-1); //Вычисляем
        values.add(num, value); //Помещаем его в таблицу
    }
}
```

Все необходимые значения вычислены, осталось только написать

```
return values.get(n);
```

Конечно, такое решение гораздо более трудоёмкое, однако это того стоит.

Хорошо, математика — это красиво. А что с задачами, в которых не всё дано?

Для больше ясности разберём следующую задачу на одномерную динамику:

> На вершине лесенки, содержащей N ступенек, находится мячик, который начинает прыгать по ним вниз, к основанию. Мячик может прыгнуть на следующую ступеньку, на ступеньку через одну или через 2. (То есть, если мячик лежит на 8-ой ступеньке, то он может переместиться на 5-ую, 6-ую или 7-ую.) Определить число всевозможных «маршрутов» мячика с вершины на землю.

Идея решения

На первую ступеньку можно попасть только одним образом — сделав прыжок с длиной равной единице. На вторую ступеньку можно попасть сделав прыжок длиной 2, или с первой ступеньки — всего 2 варианта. На третью ступеньку можно попасть сделав прыжок длиной три, с первой или со втрой ступенек. Тогда всего 4 варианта (0->3; 0->1->3; 0->2->3; 0->1->2->3). Теперь рассмотрим четвёртую ступеньку. На неё можно

попасть с первой ступеньки — по одному маршруту на каждый маршрут до неё, со второй или с третьей — аналогично. Иными словами, количество путей до 4-й ступеньки есть сумма маршрутов до 1-й, 2-й и 3-й ступенек. Математически выражаясь, `F(N) = F(N-1)+F(N-2)+F(N-3)`. Первые три ступеньки будем считать начальными состояниями.

Реализация через рекурсию

```
private static int f(int n){
    if(n==1) return 1;
    if(n==2) return 2;
    if(n==3) return 4;

    return f(n-1)+f(n-2)+f(n-3);
}
```

Здесь ничего хитрого нет.

Реализация через массив значений

Исходя из того, что, по большому счёту, простое решение на массиве из N элементов очевидно, я продемонстрирую тут решение на массиве всего из трёх.

```
int[] vars = new int[3];
vars[0]=1;vars[1]=2;vars[2]=4;

    for(int i=3; i<N; i++){
        vars[i%3] = vars[0]+vars[1]+vars[2];
    }
}

System.out.println(vars[(N-1)%3]);
```

Так как каждое следующее значение зависит только от трёх предыдущих, ни одно значение под индексом меньше `i-3` нам бы не пригодилось. В приведённом выше коде мы записываем новое значение на место самого старого, не нужного больше. Цикличность остатка от деления на 3 помогает нам избежать кучи условных операторов. Просто, компактно, элегантно.

Там вверху ещё было написано про какую-то двумерную динамику?..

С двумерной динамикой не связано никаких особенностей, однако я, на всякий случай, рассмотрю здесь одну задачу и на неё.

Спасибо за внимательность. Опечатка уже отправлена нашим редакторам.

> В прямоугольной таблице NxM в начале игрок находится в левой верхней клетке. За один ход ему разрешается перемещаться в соседнюю клетку либо вправо, либо вниз (влево и вверх перемещаться запрещено). Посчитайте, сколько есть способов у игрока попасть в правую нижнюю клетку.

Идея решения

Логика решения полностью идентична таковой в задаче про мячик и лестницу — только теперь в клетку `(x,y)` можно попасть из клеток `(x-1,y)` или `(x, y-1)`. Итого `F(x,y) = F(x-1, y)+F(x,y-1)`. Дополнительно можно понять, что все клетки вида `(1,y)` и `(x,1)` имеют только один маршрут — по прямой вниз или по прямой вправо.

Реализация через рекурсию

Ради всего святого, не нужно делать двумерную динамику через рекурсию. Уже было упомянуто, что рекурсия менее выгодна, чем цикл по быстродействию, так двумерная рекурсия ещё и читается ужасно. Это только на таком простом примере она смотрится легко и безобидно.

```
private static int f(int i, int j) {
    if(i==1 || j==1) return 1;

    return f(i-1, j)+f(i, j-1);
}
```

Реализация через массив значений

```
int[][] dp = new int[Imax][Jmax];

for(int i=0; i<Imax; i++){
    for(int j=0; j<Jmax; j++){
        if(i==0 || j==0){
            dp[i][j]=1;
        }else{
            dp[i][j]=dp[i-1][j]+dp[i][j-1];
        }
    }
}

System.out.println(dp[Imax-1][Jmax-1]);
```

Классическое решение динамикой, ничего необычного — проверяем, является ли клетка краем, и задаём её значение на основе соседних клеток.

Отлично, я всё понял. На чём мне проверить свои навыки?

Спасибо за внимательность. Опечатка уже отправлена нашим редакторам.

В заключение приведу ряд типичных задач на одномерную и двумерную динамику, разборы прилагаются.

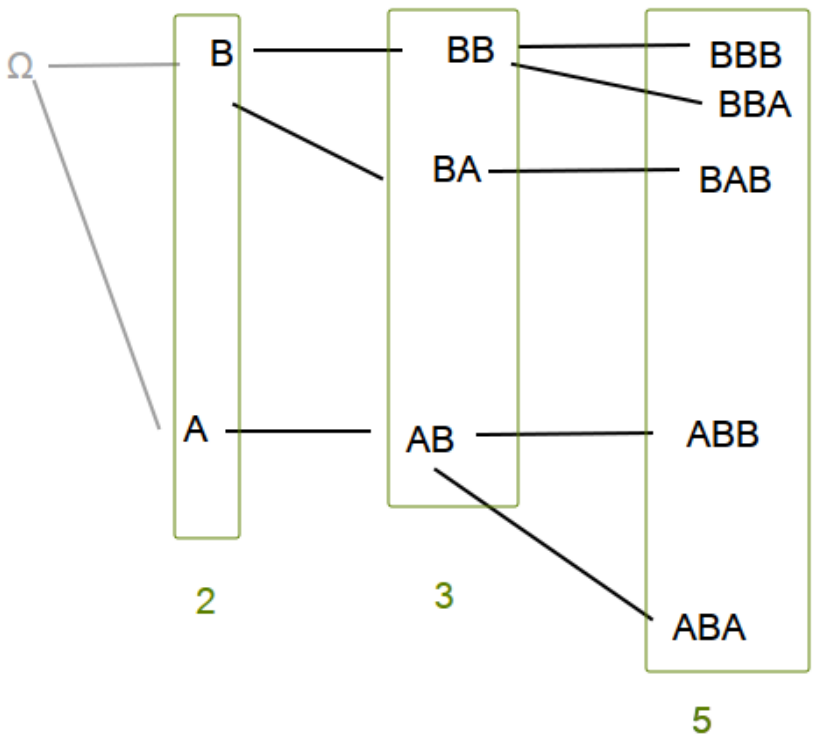
Взрывоопасность

При переработке радиоактивных материалов образуются отходы двух видов — особо опасные (тип A) и неопасные (тип B). Для их хранения используются одинаковые контейнеры. После помещения отходов в контейнеры последние укладываются вертикальной стопкой. Стопка считается взрывоопасной, если в ней подряд идет более одного контейнера типа A. Стопка считается безопасной, если она не является взрывоопасной. Для заданного количества контейнеров N определить количество возможных типов безопасных стопок.

Разбор

Решение

Ответом является (N+1)-е число Фибоначчи. Догадаться можно было, просто вычислив 2-3 первых значения. Строго доказать можно было, построив дерево возможных построений.



Каждый основной элемент делится на два — основной (заканчивается на B) и побочный (заканчивается на A). Побочные элементы превращаются в основные за одну итерацию (к последовательности, заканчивающейся на A, можно дописать

Спасибо за внимательность. Опечатка уже отправлена нашим редакторам.

только B). Это характерно для чисел Фибоначчи.

Реализация

Например, так:

```
//Ввод числа N с клавиатуры

N+=2;

BigInteger[] fib = new BigInteger[2];
fib[0]=fib[1]=BigInteger.ONE;

for(int i=2; i<N; i++){
    fib[i%2] = fib[0].add(fib[1]);
}


System.out.println(fib[(N-1)%2]);
```

Подъём по лестнице

Мальчик подошел к платной лестнице. Чтобы наступить на любую ступеньку, нужно заплатить указанную на ней сумму. Мальчик умеет перешагивать на следующую ступеньку, либо перепрыгивать через ступеньку. Требуется узнать, какая наименьшая сумма понадобится мальчику, чтобы добраться до верхней ступеньки.

Разбор

Решение

Очевидно, что сумма, к ть сумма, которую он отдал до эт а, которую он отдал до этого» зависит а N-ую — с (N-1)-й или с (N-2)-й. Выби

Реализация

Например, так:

Спасибо за внимательность. Опечатка уже отправлена нашим редакторам.

```
int Imax;
//*ввод с клавиатуры числа ступенек*

DP = new int[Imax];

for(int i=0; i<Imax; i++){
    ///Ввод с клавиатуры стоимости ступеньки DP[i]
}

for(int i=2; i<DP.length; i++){
    DP[i]+=Math.min(DP[i-1], DP[i-2]);
}

System.out.println(DP[Imax-1]);
```

Калькулятор

Имеется калькулятор, который выполняет три операции:

- Прибавить к числу X единицу;
- Умножить число X на 2;
- Умножить число X на 3.

Определите, какое наименьшее число операций необходимо для того, чтобы получить из числа 1 заданное число N. Выведите это число, и, на следующей строке, набор исполненных операций вида «111231».

Разбор

Решение

Наивное решение состоит в том, чтобы делить число на 3, пока это возможно, иначе на 2, если это возможно, иначе вычитать единицу, и так до тех пор, пока оно не обратится в единицу. Это неверное решение, т.к. оно исключает, например, возможность убавить число на единицу, а затем разделить на три, из-за чего на больших числах (например, 32718) возникают ошибки.

Правильное решение заключается в нахождении для каждого числа от 2 до N минимального количества действий на основе предыдущих элементов, иначе говоря: `F(N) = min(F(N-1), F(N/2), F(N/3)) + 1`. Следует помнить, что все индексы должны быть целыми.

Для воссоздания списка действий необходимо идти в обратном направлении и искать такой индекс i, что `F(i)=F(N)`, где N – номер рассматриваемого элемента. Если `i=N-1`, записываем в начало строки 1, если `i=N/2` – двойку, иначе – тройку.

Реализация

```
int N;
//Ввод с клавиатуры

int[] a = new int[N+1];
a[1]= 0;

{
int min;
for(int i=2; i<N+1; i++){
    min=a[i-1]+1;
    if(i%2==0) min=Math.min(min,a[i/2]+1);
    if(i%3==0) min=Math.min(min,a[i/3]+1);

    a[i] = min;
}
}

StringBuilder ret = new StringBuilder();

{
    int i=N;
    while(i>1){
        if(a[i]==a[i-1]+1){
            ret.insert(0, 1);
            i--;
            continue;
        }

        if(i%2==0&&a[i]==a[i/2]+1){
            ret.insert(0, 2);
            i/=2;
            continue;
        }

        ret.insert(0, 3);
        i/=3;
    }
}

System.out.println(a[N]);
System.out.println(ret);
```

Самый дешёвый путь

Спасибо за внимательность.
Опечатка уже отправлена нашим редакторам.

В каждой клетке прямоугольной таблицы N*M записано некоторое число. Изначально игрок находится в левой верхней клетке. За один ход ему разрешается перемещаться в соседнюю клетку либо вправо, либо вниз (влево и вверх перемещаться запрещено). При проходе через клетку с игрока берут столько килограммов еды, какое число записано в этой клетке (еду берут также за первую и последнюю клетки его пути).

Требуется найти минимальный вес еды в килограммах, отдав которую игрок может попасть в правый нижний угол.

Разбор

Решение

В любую клетку таблицы мы можем попасть либо из клетки, находящейся непосредственно над ней, либо из клетки, находящейся непосредственно слева. Таким образом, F(x,y) = min(F(x-1,y), F(x,y-1)). Чтобы не обрабатывать граничные случаи, можно добавить первую строку и столбец, заполненные некоторой константой — каким-то числом, заведомо большим содержимого любой из клеток.

Реализация

```
///nextInt() --- метод, считывающий с консоли число
int Imax=nextInt();
int Jmax=nextInt();

long[][] dp = new long[Imax][Jmax];

for(int i=0; i<Imax; i++){
    for(int j=0; j<Jmax; j++){
        dp[i][j]= nextInt();
        if(i>0 && j>0){
            dp[i][j]+=Math.min(dp[i-1][j], dp[i][j-1]);
        }else{
            if(i>0){
                dp[i][j]+=dp[i-1][j];
            }else if(j>0){
                dp[i][j]+=dp[i][j-1];
            }
        }
    }
}

System.out.println(dp[Imax-1][Jmax-1]);
```

Спасибо за внимательность. Опечатка уже отправлена нашим редакторам.

Наши тесты для вас:
— Какой язык программирования стоит выбрать для изучения? (https://goo.gl/WT87JG)
— Что вы знаете о работе мозга? (https://goo.gl/SBKoVY)
— Найдите уязвимость в веб-проекте. (https://goo.gl/zroiyY)

Java (https://tproger.ru/tag/java/), Для начинающих (https://tproger.ru/tag/for-beginners/), Рекурсия и динамика (https://tproger.ru/tag/recursion-and-dynamic-programming/)

Также рекомендуем:

8 комментариев

Ваш комментарий...

Спасибо за внимательность. Опечатка уже отправлена нашим редакторам.

Отпр

Старый Боцман
Если я правильно понял последнее решение, вы используете сравнение двух представленных ходов нахождение локальной выгодности (Greedy алгоритм), который может привезти к ошибке, если лока минимум приведет к более дорогому пути. Я бы предложил использовать брут форс перебор. Он до времени, но даст точный результат в любом случае. Как считаете?
16 янв в 14:17 Комментировать

Никита Шемякин
Классная статья. Помогла лучше разобраться с динамикой
13 июн 2017 Комментировать

Илья Минеев
f n fs i j iMax dp jMax
20 ноя 2016 Комментировать

Дима Тырышкин
Шикарная статья. Похоже я недооценивал рекурсию)
29 авг 2016 Комментировать

Артем Верещака
Зачем в задаче про Фибоначчи хеш в виде мапы? Мапы инт к инту... Хватило бы обычного массива, и списка, и по индексу забегать.
Куча недочетов мелких.
10 июн 2016

Пётр Соковых
Разумеется, конкретно в этой задаче, когда нам гаратированно понадобится вычислять все значения, начиная с первого и до N-ого, можно было бы использовать и обычный массив, и д структуры данных. Но это возможно далеко не в каждой задаче. Статья посвещена динамиче программированию вообще, а не конкретной задаче, поэтому все решения сделаны в достат общем виде.
В соседнем абзаце даже есть замечание на эту тему:
/*
Конкретно для этой задачи есть более элегантный способ найти все зависимости,
здесь же показан достаточно универсальный
*/
15 июн 2016 Ответить

Написать комментарий...

Тимур Мишагин
Вам с таким кодом нужно на ACMP задачи решать ;)
7 июн 2016 Комментировать

Женя Александров
—"Весь приведённый в статье код написан на Java"
— Alt+F4
7 июн 2016

Показать все 6 комментариев

Василий Голубцов ответил Никите
Никита, такие вещи нужно рассказывать на алгоритмическом языке, не привязываясь к синта
К примеру в решении со ступеньками есть немного лишней информации.

DP = new int[Imax];

Спасибо за внимательность.
Опечатка уже отправлена нашим редакторам.

for(int i=0; i<lmax; i++){
///Ввод с клавиатуры стоимости ступеньки DP[i]
} ...
10 июн 2016 Ответить

Василий Федорищев ответил Никите
Ага. Обчитаются на своих форумах по PHP всякой фигни про Java и яб... ой пишут всякую фиг другу... 😆😆😆
3 июл 2016 Ответить

Евгений Кулешов
Java - это C подобный язык, а синтаксис C должен знать любой уважающий себя человек.
5 янв 2017 Ответить

Написать комментарий...

Иван Левченко
cashe, stek... Отличные названия переменных.
7 июн 2016

Артем Верещака
да и читать дальше "наличных" перестал...
10 июн 2016 Ответить

Написать комментарий...

“
Лень кодить нудную фичу. Убедил заказчика, что она не нужна.
”

Спасибо за внимательность.
Опечатка уже отправлена нашим редакторам.

Рубрики

/dev/null (https://tproger.ru/category/devnull/) Видео (https://tproger.ru/category/video/)

Викторины (https://tproger.ru/category/quiz/) Задачки (https://tproger.ru/category/problems/)

Интервью (https://tproger.ru/category/interview/)

Интересные ссылки (https://tproger.ru/category/links/) Книги (https://tproger.ru/category/books/)

Коротко о главном (https://tproger.ru/category/explain/) Новости (https://tproger.ru/category/news/)

Ответы экспертов (https://tproger.ru/category/experts/)

Переводы (https://tproger.ru/category/translations/) Подборки (https://tproger.ru/category/digest/)

Рассказы о своих проектах (https://tproger.ru/category/projects/)

События (https://tproger.ru/category/events/) Статьи (https://tproger.ru/category/articles/)

Темы

Android (https://tproger.ru/tag/android/) Apple (https://tproger.ru/tag/apple/)

C# (https://tproger.ru/tag/c-sharp/) C++ (https://tproger.ru/tag/cpp/) CSS (https://tproger.ru/tag/css/)

Facebook (https://tproger.ru/tag/facebook/) GitHub (https://tproger.ru/tag/github/)

Google (https://tproger.ru/tag/google/) Google Chrome (https://tproger.ru/tag/google-chrome/)

Hardware (https://tproger.ru/tag/hardware/) IDE (https://tproger.ru/tag/ide/)

iOS (https://tproger.ru/tag/ios/) Java (https://tproger.ru/tag/java/)

JavaScript (https://tproger.ru/tag/javascript/) Linux (https://tproger.ru/tag/linux/)

Microsoft (https://tproger.ru/tag/microsoft/) Open Source (https://tproger.ru/tag/open-source/)

PHP (https://tproger.ru/tag/php/) Python (https://tproger.ru/tag/python/)

Windows (https://tproger.ru/tag/windows/) Windows 10 (https://tproger.ru/tag/windows-10/)

Алгоритмы (https://tproger.ru/tag/algorithms/) Безопасность (https://tproger.ru/tag/security/)

Блокчейн (https://tproger.ru/tag/blockchain/) Браузеры (https://tproger.ru/tag/browsers/)

Веб-разработка (https://tproger.ru/tag/web/)

Виртуальная реальность (https://tproger.ru/tag/virtual-reality/)

Головоломки (https://tproger.ru/tag/brain-teasers/)

Для начинающих (https://tproger.ru/tag/for-beginners/)

Для продолжающих (https://tproger.ru/tag/for-amateurs/) Инструменты (https://tproger.ru/tag/tools/)

Интернет (https://tproger.ru/tag/internet/) Искусственный интеллект (https://tproger.ru/tag/ai/)

Лучшая практика (https://tproger.ru/tag/best-practice/)

Материалы от друзей Tproger (https://tproger.ru/tag/guest/)

Машинное обучение (https://tproger.ru/tag/machine-learning/)

Мобильная разработка (https://tproger.ru/tag/mobiledev/)

Нейронные сети (https://tproger.ru/tag/neural-network/)

Облачные технологии (https://tproger.ru/tag/cloud/)

Обучающие курсы (https://tproger.ru/tag/courses/)

Обучение программированию (https://tproger.ru/tag/learn-programming/)

Разработка игр (https://tproger.ru/tag/gamedev/) Распознавание (https://tproger.ru/tag/recognition/)

Языки C++ (https://tproger.ru/tag/c-language/)

Языки программирования (https://tproger.ru/tag/programming-languages/)