

Артём @Zealint

Пользователь

109,0 карма

-1,8 рейтинг

Профиль

8 Публикации

171 Комментарии

8 Избранное

23 Подписчики

13 февраля в 14:48

Разработка → Обстоятельно о подсчёте единичных битов

tutorial

Спортивное программирование*, Программирование*, Алгоритмы*

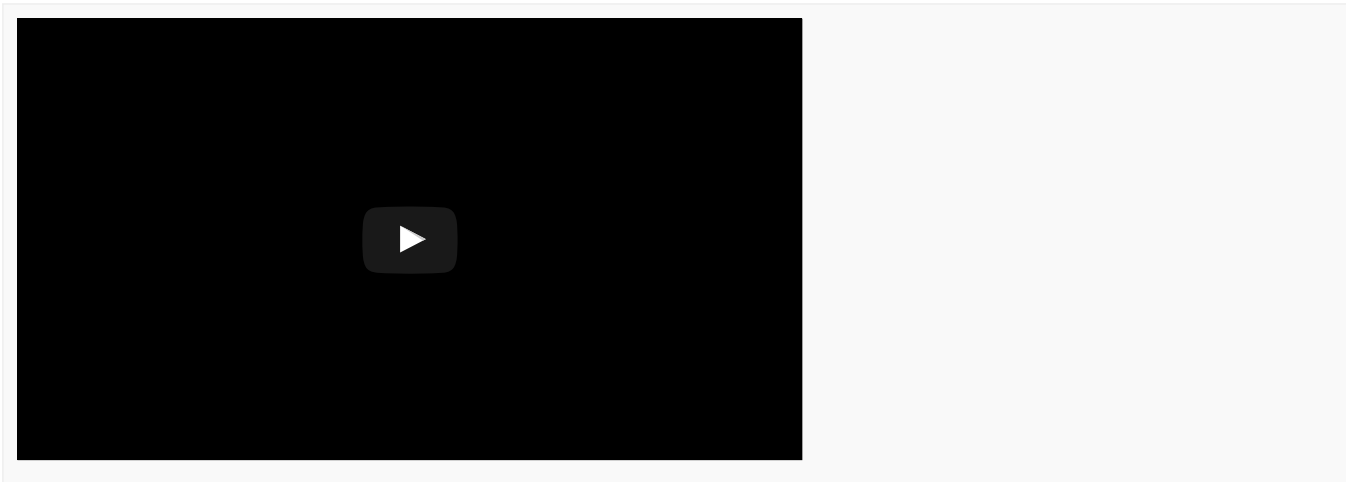
Я хотел бы подарить сообществу Хабра статью, в которой стараюсь дать достаточно полное описание подходов к алгоритмам подсчёта единичных битов в переменных размером от 8 до 64 битов. Эти алгоритмы относятся к разделу так называемой «битовой магии» или «битовой алхимии», которая завораживает своей красотой и неочевидностью многих программистов. Я хочу показать, что в основах этой алхимии нет ничего сложного, и вы даже сможете разработать собственные методы подсчёта единичных битов, познакомившись с фундаментальными приёмами, составляющими подобные алгоритмы.

Прежде чем мы начнём, я сразу хочу предупредить, что это статья не для новичков в программировании. Мне необходимо, чтобы читатель в общих чертах представлял себе простейшие битовые операции (побитовое «и», «или», сдвиг), хорошо владел шестнадцатеричной системой счисления и достаточно уверенно пользовался воображением, представляя в нём не всегда короткие битовые последовательности. По возможности, всё будет сопровождаться картинками, но сами понимаете, они лишь упрощают, но не заменяют полное представление.

Все описанные приёмы были реализованы на языке Си и протестированы в двух режимах: 32 и 64 бита. Таким образом, для более полного понимания статьи будет лучше, чтобы вы хотя бы приблизительно понимали язык Си. Тестирование проходило на процессоре Core 2 Duo E8400 @3GHz на 64-х битовой Windows 7. Измерение чистого времени работы программ проводилось с помощью утилиты [runexex](#). Все исходные коды описываемых алгоритмов доступны [в архиве](#) на Яндекс диске, их компиляция проверена для компиляторов Visual C++, Intel C++, GCC и CLang, так что в принципе, проблем у вас быть не должно, если кто-то захочет перепроверить результаты у себя. Пользователи Linux, думаю, лучше меня знают, как им тестировать время работы программы у себя в системе, поэтому им советов не даю.

Среди читателей, возможно, будут такие, кому проще посмотреть всё то же самое на видео. Я записал такое видео (58 минут), в котором в формате презентации изложено в точности всё то же самое, что будет ниже по тексту, но может немного в другом стиле, более сухо и строго, тогда как текст я попытался немного оживить. Поэтому изучайте материал так, как кому удобнее.

Смотреть видео



Сейчас будут последовательно описаны алгоритмы, порождаемые тем или иным набором алхимических приёмов, в каждом разделе будет таблица сравнения времени работы для переменных разного размера, а в конце будет сводная таблица по всем алгоритмам. Во всех алгоритмах используются псевдонимы для чисел без знака от 8 до 64 бит.

```
typedef unsigned char    u8;
typedef unsigned short int u16;
typedef unsigned int     u32;
typedef unsigned long long u64;
```

Наивный подход

Очевидно, что битовая алхимия применяется вовсе не для того, чтобы блистать на собеседовании, а с целью существенного ускорения программ. Ускорения по отношению к чему? По отношению к тривиальным приёмам, которые могут прийти в голову, когда нет времени более детально вникнуть в задачу. Таковым приёмом и является наивный подход к подсчёту битов: мы просто «откусываем» от числа один бит за другим и суммируем их, повторяя процедуру до тех пор, пока число не станет равным нулю.

```
u8 CountOnes0 (u8 n) {
    u8 res = 0;
    while (n) {
        res += n&1;
        n >>= 1;
    }
    return res;
}
```

Я не вижу смысла что-либо комментировать в этом тривиальном цикле. Невооружённым взглядом ясно, что если старший бит числа n равен 1, то цикл вынужден будет пройти по всем битам числа, прежде чем доберётся до старшего.

Меняя тип входного параметра `u8` на `u16`, `u32` и `u64` мы получим 4 различные функции. Давайте протестируем каждую из них на потоке из 2^{32} чисел, подаваемых в хаотичном порядке. Понятно, что для `u8` у нас 256 различных входных данных, но для единообразия мы всё равно прогоняем 2^{32} случайных чисел для всех этих и всех последующих функций, причём всегда в одном и том же порядке (за подробностями можно обратиться к коду учебной программы из архива).

Время в таблице ниже указано в секундах. Для тестирования программа запускалась трижды и выбиралось среднее время. Погрешность едва ли превышает 0,1 секунды. Первый столбец отражает режим компилятора (32-х битовый исходный код или 64-х битовый), далее 4 столбца отвечают за 4 варианта входных данных.

Режим	u8	u16	u32	u64
x86	38,18	72,00	130,49	384,76
x64	37,72	71,51	131,47	227,46

Как мы видим, скорость работы вполне закономерно возрастает с ростом размера входного параметра. Немного выбивается из общей закономерности вариант, когда числа имеют размер 64 бита, а подсчёт идёт режиме `x86`. Ясное дело, что процессор вынужден делать четырёхкратную работу при удвоении входного параметра и даже хорошо, что он справляется всего лишь втрое медленнее.

Первая польза этого подхода в том, что при его реализации трудно ошибиться, поэтому написанная таким образом программа может стать эталонной для проверки более сложных алгоритмов (именно так и было сделано в моём случае). Вторая польза в универсальности и относительно простой переносимости на числа любого размера.

Трюк с «откусыванием» младших единичных битов

Этот алхимический приём основан на идее обнуления младшего единичного бита. Имея число n , мы можем произнести заклинание $n = n \& (n - 1)$, забирая у числа n его младшую единицу. Картинка ниже для $n = 232$ прояснит ситуацию для людей, впервые узнавших об этом трюке.

n	1	1	1	0	1	0	0	0
$n - 1$	1	1	1	0	0	1	1	1
$n \& (n - 1)$	1	1	1	0	0	0	0	0

Код программы не сильно изменился.

```
u8 CountOnes1 (u8 n) {
    u8 res = 0;
    while (n) {
        res++;
        n &= n-1; // Забираем младшую единицу.
    }
}
```

```
    }  
    return res;  
}
```

Теперь цикл выполнится ровно столько раз, сколько единиц в числе n. Это не избавляет от худшего случая, когда все биты в числе единичные, но значительно сокращает среднее число итераций. Сильно ли данный подход облегчит страдания процессора? На самом деле не очень, а для 8 бит будет даже хуже. Напомню, что сводная таблица результатов будет в конце, а здесь в каждом разделе будет своя таблица.

Режим	u8	u16	u32	u64
x86	44,73	55,88	72,02	300,78
x64	40,96	69,16	79,13	126,72

Предподсчёт

Не будем торопиться переходить к «жёстким» заклинаниям, рассмотрим последний простой приём, который может спасти даже самого неопытного мага. Данный вариант решения задачи не относится напрямую к битовой алхимии, однако для полноты картины должен быть рассмотрен в обязательном порядке. Заведём две таблицы на 256 и 65536 значений, в которых заранее посчитаны ответы для всех возможных 1-байтовых и 2-байтовых величин соответственно.

```
u8 BitsSetTableFF[256]; // Здесь все ответы для одного байта  
u8 BitsSetTableFFFF[65536]; // Здесь все ответы для двух байт
```

Теперь программа для 1 байта будет выглядеть так

```
u8 CountOnes2_FF (u8 n) {  
    return BitsSetTableFF[n];  
}
```

Чтобы рассчитать число бит в более крупных по размеру числах, их нужно разбить на байты. Например, для u32 может быть вот такой код:

```
u8 CountOnes2_FF (u32 n) {  
    u8 *p = (u8*)&n;  
    n = BitsSetTableFF[p[0]]  
      + BitsSetTableFF[p[1]]  
      + BitsSetTableFF[p[2]]  
      + BitsSetTableFF[p[3]];  
    return n;  
}
```

Или такой, если мы применяем таблицу предподсчёта для 2-х байт:

```
u8 CountOnes2_FFFF (u32 n) {  
    u16 *p = (u16*)&n;  
    n = BitsSetTableFFFF[p[0]]  
      + BitsSetTableFFFF[p[1]];  
    return n;  
}
```

Ну а дальше вы догадались, для каждого варианта размера входного параметра n (кроме 8 бит) может существовать два варианта предподсчёта, в зависимости от того, которую из двух таблиц мы применяем. Думаю, читателю понятно, почему мы не можем просто так взять и завести таблицу BitsSetTableFFFFFFFF, однако вполне могут существовать задачи, где и это будет оправданным.

Быстро ли работает предподсчёт? Всё сильно зависит от размера, смотрите таблицы ниже. Первая для однобайтового предподсчёта, а вторая для двухбайтового.

Режим	u8	u16	u32	u64
x86	0,01	1,83	21,07	36,25
x64	0,01	1,44	24,79	26,84

Интересный момент: для режима x64 предподсчёт для u64 работает заметно быстрее, возможно, это особенности оптимизации,

Режим	u8	u16	u32	u64
x86	---	0,05	7,95	13,01
x64	---	0,07	8,49	13,01

4/25

$$A \bmod (2^k - 1) = (A_0 \cdot 2^{(N-k)} + A_1 \cdot 2^{(N-1-k)} + \dots + A_{N-1} \cdot 2^{(N-1-k)}) \bmod (2^k - 1) = (A_0 + A_1 + \dots + A_{N-1}) \bmod (2^k - 1).$$

Всё благодаря тому, что $2^k \equiv 1 \pmod{2^k - 1}$ для любого целого неотрицательного i . (Здесь, правда, важно, что трюк имеет смысл, когда $k > 1$, иначе не совсем понятно как нам интерпретировать модуль 1). Вот мы и получили сумму блоков по модулю $2^k - 1$.

То есть от полученного нами числа нужно взять остаток от деления на $2^3 - 1$ (семь) — и мы получаем сумму наших 8-и блоков по модулю 7. Беда в том, что сумма бит может быть равна 7 или 8, в таком случае алгоритм выдаст 0 и 1 соответственно. Но давайте посмотрим: в каком случае мы можем получить ответ 8? Только когда $n=255$. А в каком случае можем получить 0? Только когда $n=0$. Поэтому если алгоритм после взятия остатка на 7 даст 0, то либо мы на входе получили $n=0$, либо в числе ровно 7 единичных бит. Суммируя эту рассуждение, получаем следующий код:

```
u8 CountOnes3 (u8 n) {
    if (n == 0) return 0; // Единственный случай, когда ответ 0.
    if (n == 0xFF) return 8; // Единственный случай, когда ответ 8.
    n = (0x010101*n & 0x249249) % 7; // Считаем число бит по модулю 7.
    if (n == 0) return 7; // Гарантированно имеем 7 единичных битов.
    return n; // Случай, когда в числе от 1 до 6 единичных битов.
}
```

В случае когда n имеет размер 16 бит можно разбить его на две части по 8 бит. Например, так:

```
u8 CountOnes3 (u16 n) {
    return CountOnes3 (u8(n & 0xFF)) + CountOnes3 (u8(n >> 8));
}
```

Для случая 32-х и 64-х бит подобное разбиение не имеет смысла уже даже в теории, умножение и остаток от деления с тремя ветвлениями будут слишком дорого стоить, если выполнять их 4 или 8 раз подряд. Но я оставил для вас пустые места в нижеследующей таблице, поэтому если вы мне не верите — заполните их, пожалуйста, сами. Там скорее всего будут результаты, сравнимые с процедурой CountBits1, если у вас похожий процессор (я не говорю о том, что здесь возможны оптимизации с помощью SSE, это уже будет другой разговор).

Режим	u8	u16	u32	u64
x86	12,42	30,57	---	---
x64	13,88	33,88	---	---

Данный трюк, конечно, можно сделать и без ветвлений, но тогда нам нужно, чтобы при разбиении числа на блоки в блок вместились все числа от 0 до 8, а этого можно добиться лишь в случае 4-битовых блоков (и больше). Чтобы выполнить суммирование 4-битовых блоков, нужно подобрать множитель, который позволит правильно «растиражировать» число и взять остаток от деления на $2^4 - 1 = 15$, чтобы сложить получившиеся блоки. Опытный алхимик (который знает математику) легко подберёт такой множитель: 0x08040201. Почему он выбран таким?

0 a b c	d e f g	h 0 a b	c d e f	g h 0 a	b c d e	f g h 0	a b c d	e f g h
0 0 0 1	0 0 0 1	0 0 0 1	0 0 0 1	0 0 0 1	0 0 0 1	0 0 0 1	0 0 0 1	0 0 0 1

Дело в том, что нам необходимо, чтобы все биты исходного числа заняли правильные позиции в своих 4-битовых блоках (картинка выше), а коль скоро 8 и 4 не являются взаимно простыми числами, обычное копирование 8 битов 4 раза не даст правильного расположения нужных битов. Нам придётся добавить к нашему байту один нолик, то есть тиражировать 9 битов, так как 9 взаимно просто с 4. Так мы получим число, имеющее размер 36 бит, но в котором все биты исходного байта стоят на младших позициях 4-битовых блоков. Осталось только взять побитовое «и» с числом 0x11111111 (вторая строка на картинке выше), чтобы обнулить по три старших бита в каждом блоке. Затем блоки нужно сложить.

При таком подходе программа подсчёта единичных битов в байте будет предельно простой:

```
u8 CountOnes3_x64 (u8 n) {
    return ((u64)0x08040201*n & 0x11111111) % 15;
}
```

Недостаток программы очевиден: требуется выход в 64-битовую арифметику со всеми вытекающими отсюда последствиями. Можно заметить, что в действительности данная программа задействует только 33 бита из 64-х (старшие 3 бита обнуляются), и в принципе можно сообразить, как перенести данные вычисления в 32-х битовую арифметику, но рассказы о подобных оптимизациях не входят в тему этого руководства. Давайте пока просто изучать приёмы, а оптимизировать их вам придётся самим уже под конкретную задачу.

Ответим на вопрос о том, какого размера может быть переменная n , чтобы данный трюк правильно работал для неё. Коль скоро

мы берём остаток от деления на 15, такая переменная не может иметь размер больше 14 бит, в противном случае придётся применить ветвление, как мы делали это раньше. Но для 14 бит приём работает, если добавить к 14-ти битам один нолик, чтобы все биты встали на свои позиции. Теперь я буду считать, что вы в целом усвоили суть приёма и сможете сами без труда подобрать множитель для тиражирования и маску для обнуления ненужных битов. Покажу сразу готовый результат.

```
u8 CountOnes3_x64 (u14 n) { // Это не опечатка (и не должно работать)!
    return (n*0x200040008001llu & 0x11111111111111llu) % 15;
}
```

Эта программа выше показывает, как мог бы выглядеть код, будь у вас переменная размером 14 бит без знака. Этот же код будет работать с переменной в 15 бит, но при условии, что максимум лишь 14 из них равны единице, либо если случай, когда $n=0x7FFF$ мы разберём отдельно. Это всё нужно понимать для того, чтобы написать правильный код для переменной типа `u16`. Идея в том, чтобы сначала «откусить» младший бит, посчитать биты в оставшемся 15-ти битовом числе, а затем обратно прибавить «откушенный» бит.

```
u8 CountOnes3_x64 (u16 n) {
    u8 leastBit = n&1; // Берём младший бит.
    n >>= 1; // Оставляем только 15 бит исходного числа.
    if (n == 0) return leastBit; // Если получился 0, значит ответом будет младший бит.
    if (n == 0x7FFF) return leastBit + 15; // Единственный случай, когда ответ 15+младший бит.
    return leastBit + (n*0x200040008001llu & 0x11111111111111llu) % 15; // Ответ (максимум 14+младший бит).
}
```

Для n размером 32 бита приходится колдовать уже с более серьёзным лицом... Во-первых, ответ влезает только в 6 бит, но можно рассмотреть отдельный случай, когда $n=2^{32}-1$ и спокойно делать расчёты в полях размером 5 бит. Это экономит место и разрешает нам разбить 32-битовое поле числа n на 3 части по 12 бит в каждой (да, последнее поле будет не полным). Поскольку $12 \cdot 5 = 60$, мы можем тиражировать одно 12-битовое поле 5 раз, подобрав множитель, а затем сложить 5-битовые блоки, взяв остаток от деления на 31. Сделать это нужно 3 раза для каждого поля. Суммируя итог, получаем такой код:

```
u8 CountOnes3_x64 (u32 n) {
    if (n == 0) return 0;
    if (n + 1 == 0) return 32;
    u64 res = (n&0xFFF)*0x1001001001001llu & 0x84210842108421llu;
    res += ((n&0xFFFF000)>>12)*0x1001001001001llu & 0x84210842108421llu;
    res += (n>>24)*0x1001001001001llu & 0x84210842108421llu;
    res %= 0x1F;
    if (res == 0) return 31;
    return res;
}
```

Здесь точно также можно было вместо трех ветвлений взять 3 остатка от деления, но я выбрал ветвистый вариант, на моём процессоре он будет работать лучше.

Для n размером 64 бита мне не удалось придумать подходящего заклинания, в котором было бы не так много умножений и сложений. Получалось либо 6, либо 7, а это слишком много для такой задачи. Другой вариант — выход в 128-битовую арифметику, а это уже не пойми каким «откатом» для нас обернётся, неподготовленного мага может и к стенке отшвырнуть :)

Давайте лучше посмотрим на время работы.

Режим	u8	u16	u32	u64
x86	39,78	60,48	146,78	---
x64	6,78	12,28	31,12	---

Очевидным выводом из этой таблицы будет то, что 64-х битовая арифметика плохо воспринимается в 32-х битовом режиме исполнения, хотя в целом-то алгоритм неплох. Если вспомнить скорость алгоритма предподсчёта в режиме x64 для однобайтовой таблицы для случая `u32` (24,79 с), то получим, что данный алгоритм отстаёт всего лишь на 25%, а это повод к соревнованию, воплощённому в следующем разделе.

Замена взятия остатка на умножение и сдвиг

Недостаток операции взятия остатка всем очевиден. Это деление, а деление – это долго. Разумеется, современные компиляторы знают алхимию и умеют заменять деление на умножение со сдвигом, а чтобы получить остаток, нужно вычесть из делимого частное, умноженное на делитель. Тем не менее, это всё равно долго! Оказывается, что в древних свитках заклинателей кода сохранился один интересный способ оптимизации предыдущего алгоритма. Мы можем суммировать k -битовые блоки не взятием остатка от деления, а ещё одним умножением на маску, с помощью которой обнуляли лишние биты в блоках. Вот как это

выглядит для n размером в 1 байт.

Для начала снова тиражируем байт трижды и удаляем по два старших бита у каждого 3-битового блока с помощью уже пройденной выше формулы $0x010101 \cdot n \& 0x249249$.

0 0 c	0 0 f	0 0 a	0 0 d	0 0 g	0 0 b	0 0 e	0 0 h
C	F	A	D	G	B	E	H

Каждый трёхбитовый блок я для удобства обозначил заглавной латинской буквой. Теперь умножаем полученный результат на ту же самую маску 0x249249. Маска содержит единичный бит в каждой 3-й позиции, поэтому такое умножение эквивалентно сложению числа самого с собой 8 раз, каждый раз со сдвигом на 3 бита:

$$\begin{array}{cccccccc}
& & & & C & F & A & D & G & B & E & H \\
+ & & & & C & F & A & D & G & B & E & H \\
+ & & & C & F & A & D & G & B & E & H \\
+ & & C & F & A & D & G & B & E & H \\
+ & & C & F & A & D & G & B & E & H \\
+ & C & F & A & D & G & B & E & H \\
+ & C & F & A & D & G & B & E & H \\
+ & C & F & A & D & G & B & E & H
\end{array}$$

Что мы видим? Биты с 21 по 23 и дают нам нужную сумму! При этом переполнения в каком-либо из блоков справа невозможно, так как там ни в одном блоке не будет числа, большего 7. Проблема лишь в том, что если наша сумма равна 8, мы получим 0, но это не страшно, ведь этот единственный случай можно рассмотреть отдельно.

```
u8 CountOnes3_x64_m (u8 n) {  
    if (n == 0xFF) return 8;  
    return ((u64 (0x010101*n & 0x249249) * 0x249249) >> 21) & 0x7;  
}
```

По сути, мы взяли код из предыдущего раздела и заменили в нём взятие остатка от деления на 7 на умножение, сдвиг и побитовое «И» в конце. При этом вместо 3-х ветвлений осталось лишь одно.

Чтобы составить аналогичную программу для 16 бит, нам нужно взять код из предыдущего раздела, в котором показано как это делается с помощью взятия остатка от деления на 15 и заменить данную процедуру умножением. При этом нетрудно заметить то, какие условия можно убрать из кода.

```
u8 CountOnes3_x64_m (u16 n) {
    u8 leastBit = n&1; // Берём младший бит

    n >>= 1; // Оставляем только 15 бит.

    return leastBit

    + (( (n*0x200040008001llu & 0x11111111111111llu)*0x11111111111111llu
        >> 56) & 0xF); // Ответ (максимум 15 + младший бит).
}
```

Для 32-х бит мы делаем то же самое: берём код из предыдущего раздела и, порисовав немного на бумаге, соображаем, каким будет сдвиг, если заменить остаток на умножение.

```

u8 CountOnes3_x64_m ( u32 n ) {

    if (n+1 == 0)    return 32;

    u64 res = (n&0xFFF)*0x10010010010011llu & 0x842108421084211llu;

    res += ((n&0xFFF000)>>12)*0x10010010010011llu & 0x842108421084211llu;

    res += (n>>24)*0x10010010010011llu & 0x842108421084211llu;

    return (res*0x842108421084211llu >> 55) & 0x1F;

}

```

Для 64-х бит я тоже не смог придумать чего-то такого, чтобы не заставляло бы мой процессор выполнять роль печки.

Режим	u8	u16	u32	u64
x86	12,66	42,37	99,90	---
x64	3,54	4,51	18,35	---

Приятно удивили результаты для режима x64. Как и ожидалось, мы обогнали предподсчёт с однобайтовой таблицей для случая u32. Можно ли вообще обогнать предподсчёт? Хороший вопрос :)

Параллельное суммирование

Пожалуй, это самый распространённый трюк, который очень часто повторяют друг за другом не вполне опытные заклинатели, не понимая, как он точно работает.

Начнём с 1 байта. Байт состоит из 4-х полей по 2 бита, сначала просуммируем биты в этих полях, произнесем что-то вроде:

```
n = (n >> 1) & 0x55 + (n & 0x55);
```

Вот пояснительная картинка к данной операции (по-прежнему, обозначаем биты одного байта первыми латинскими буквами):

n	a b	c d	e f	g h
$x = n \& 0x55$	0 b	0 d	0 f	0 h
$y = (n \gg 1) \& 0x55$	0 a	0 c	0 e	0 g
$x + y$	a + b	c + d	e + f	g + h

Одно из побитовых «И» оставляет только младшие биты каждого двухбитового блока, второе оставляет старшие биты, но сдвигает их на позиции, соответствующие младшим битам. В результате суммирования получаем сумму смежных битов в каждом двухбитовом блоке (последняя строка на картинке выше).

Теперь сложим парами числа, находящиеся в двухбитовых полях, помещая результат в 2 четырёхбитовых поля:

```
n = (n >> 2) & 0x33 + (n & 0x33);
```

Нижеследующая картинка поясняет результат. Привожу её теперь без лишних слов:

n	a+b	c+d	e+f	g+h
$x = n \& 0x33$	0	c+d	0	g+h
$y = (n \gg 2) \& 0x33$	0	a+b	0	e+f
$x + y$	a+b+c+d		e+f+g+h	

Наконец, сложим два числа в четырёхбитовых полях:

```
n = (n >> 4) & 0x0F + (n & 0x0F);
```

n	a+b+c+d	e+f+g+h
$x = n \& 0x0F$	0	e+f+g+h
$y = (n \gg 4) \& 0x0F$	0	a+b+c+d
$x + y$	a+b+c+d+e+f+g+h	

Действуя по аналогии, можно распространить приём на любое число бит, равное степени двойки. Число строк заклинания равно двоичному логарифму от числа бит. Уловив идею, взгляните вскользь на 4 функции, записанных ниже, чтобы убедиться в правильности своего понимания.


```
u8 CountOnes4 (u8 n) {
    n = ((n>>1) & 0x55) + (n & 0x55);
    n = ((n>>2) & 0x33) + (n & 0x33);
    n = ((n>>4) & 0x0F) + (n & 0x0F);
    return n;
}
```

```
u8 CountOnes4 (u16 n) {
    n = ((n>>1) & 0x5555) + (n & 0x5555);
    n = ((n>>2) & 0x3333) + (n & 0x3333);
    n = ((n>>4) & 0x0F0F) + (n & 0x0F0F);
    n = ((n>>8) & 0x00FF) + (n & 0x00FF);
    return n;
}
```

```
u8 CountOnes4 (u32 n) {
    n = ((n>>1) & 0x55555555) + (n & 0x55555555);
    n = ((n>>2) & 0x33333333) + (n & 0x33333333);
    n = ((n>>4) & 0x0F0F0F0F) + (n & 0x0F0F0F0F);
    n = ((n>>8) & 0x00FF00FF) + (n & 0x00FF00FF);
    n = ((n>>16) & 0x0000FFFF) + (n & 0x0000FFFF);
    return n;
}
```

```
u8 CountOnes4 (u64 n) {
    n = ((n>>1) & 0x5555555555555555llu) + (n & 0x5555555555555555llu);
    n = ((n>>2) & 0x3333333333333333llu) + (n & 0x3333333333333333llu);
    n = ((n>>4) & 0x0F0F0F0F0F0F0F0Fllu) + (n & 0x0F0F0F0F0F0F0F0Fllu);
    n = ((n>>8) & 0x00FF00FF00FF00FFllu) + (n & 0x00FF00FF00FF00FFllu);
    n = ((n>>16) & 0x0000FFFF0000FFFFllu) + (n & 0x0000FFFF0000FFFFllu);
    n = ((n>>32) & 0x00000000FFFFFFFFllu) + (n & 0x00000000FFFFFFFFllu);
    return n;
}
```

На этом параллельное суммирование не заканчивается. Развить идею позволяет то наблюдение, что в каждой строчке дважды используется одна и та же битовая маска, что как будто наводит на мысль «а нельзя ли как-нибудь только один раз выполнить побитовое «И»?». Можно, но не сразу. Вот что можно сделать, если взять в качестве примера код для u32 (смотрите комментарии).

```
u8 CountOnes4 (u32 n) {
    n = ((n>>1) & 0x55555555) + (n & 0x55555555); // Можно заменить на разность
    n = ((n>>2) & 0x33333333) + (n & 0x33333333); // Нельзя исправить
    n = ((n>>4) & 0x0F0F0F0F) + (n & 0x0F0F0F0F); // Можно вынести & за скобку
    n = ((n>>8) & 0x00FF00FF) + (n & 0x00FF00FF); // Можно вынести & за скобку
    n = ((n>>16) & 0x0000FFFF) + (n & 0x0000FFFF); // Можно вообще убрать &
    return n; // Неявное обрезание по 8-и младшим битам.
}
```

В качестве упражнения я бы хотел предложить доказать самостоятельно то, почему нижеследующий код будет точным отображением предыдущего. Для первой строки я даю подсказку, но не смотрите в неё сразу:

▼ [Подсказка](#)

Двухбитовый блок ab имеет точное значение $2a+b$, значит вычитание сделает его равным...?

```
u8 CountOnes4_opt (u32 n) {
    n -= (n>>1) & 0x55555555;
    n = ((n>>2) & 0x33333333) + (n & 0x33333333);
    n = ((n>>4) + n) & 0x0F0F0F0F;
    n = ((n>>8) + n) & 0x00FF00FF;
    n = ((n>>16) + n);
}
```

```
return n;
}
```

Аналогичные варианты оптимизации возможны и для остальных типов данных.

Ниже приводятся две таблицы: одна для обычного параллельного суммирования, а вторая для оптимизированного.

Режим	u8	u16	u32	u64
x86	7,52	14,10	21,12	62,70
x64	8,06	11,89	21,30	22,59

Режим	u8	u16	u32	u64
x86	7,18	11,89	18,86	65,00
x64	8,09	10,27	19,20	19,20

В целом мы видим, что оптимизированный алгоритм работает хорошо, но проигрывает обычному в режиме x86 для u64.

Комбинированный метод

Мы видим, что наилучшие варианты подсчёта единичных битов – это параллельный метод (с оптимизацией) и метод тиражирования с умножением для подсчёта суммы блоков. Мы можем объединить оба метода, получая комбинированный алгоритм.

Первое, что нужно сделать — выполнить первые три строки параллельного алгоритма. Это даст нам точную сумму битов в каждом байте числа. Например, для u32 выполним следующее:

```
n -= (n>>1) & 0x55555555;
n = ((n>>2) & 0x33333333) + (n & 0x33333333);
n = (((n>>4) + n) & 0x0F0F0F0F);
```

Теперь наше число n состоит из 4 байт, которые следует рассматривать как 4 числа, сумму которых мы ищем:

$$n = \begin{array}{|c|c|c|c|} \hline A & B & C & D \\ \hline \end{array}$$

Мы можем найти сумму этих 4-х байт, если умножим число n на 0x01010101. Вы теперь хорошо понимаете, что означает такое умножение, для удобства определения позиции, в которой будет находиться ответ, привожу картинку:

$$\begin{array}{rcccc} & & & A & B & C & D \\ + & & & A & B & C & D \\ + & & A & B & C & D \\ + & A & B & C & D \end{array}$$

Ответ находится в 3-байте (если считать их от 0). Таким образом, комбинированный приём для u32 будет выглядеть так:

```
u8 CountOnes5 ( u32 n ) {
    n -= (n>>1) & 0x55555555;
    n = ((n>>2) & 0x33333333) + (n & 0x33333333);
    n = (((n>>4) + n) & 0x0F0F0F0F) * 0x01010101 >> 24;
    return n; // Здесь происходит неявное обрезание по 8 младшим битам.
}
```

Он же для u16:

```
u8 CountOnes5 (u16 n) {
    n -= (n>>1) & 0x5555;
    n = ((n>>2) & 0x3333) + (n & 0x3333);
    n = (((n>>4) + n) & 0x0F0F) * 0x0101 >> 8;
```

```
return n; // Здесь происходит неявное обрезание по 8 младшим битам.  
}
```

Он же для u64:

```
u8 CountOnes5 ( u64 n ) {  
    n -= (n>>1) & 0x5555555555555555llu;  
    n = ((n>>2) & 0x3333333333333333llu) + (n & 0x3333333333333333llu);  
    n = (((n>>4) + n) & 0x0F0F0F0F0F0F0F0Fllu)  
        * 0x0101010101010101llu >> 56;  
    return n; // Здесь происходит неявное обрезание по 8 младшим битам.  
}
```

Скорость работы этого метода вы можете посмотреть сразу в итоговой таблице.

Итоговое сравнение

Я предлагаю читателю самостоятельно сделать интересующие его выводы, изучив две нижеследующие таблицы. В них я обозначил название методов, программы к которым мы реализовали, а также пометил прямоугольной рамкой те подходы, которые я считаю наилучшими в каждом конкретном случае. Тех, кто думал, что предподсчёт всегда выигрывает, ожидает небольшой сюрприз для режима x64.

Итоговое сравнения для режима компиляции x86.

Функция	Тип n			
	u8	u16	u32	u64
Наивный метод	38,18	72,00	130,49	384,76
Удаление младшей единички	44,73	55,88	72,02	300,78
Предподсчёт - 1 байт	0,01	1,83	21,07	36,25
Предподсчёт - 2 байта	—	0,05	7,95	13,01
Умножение и остаток (32 бита)	12,42	30,57	—	—
Умножение и остаток (64 бита)	39,78	60,48	146,78	—
Умножение и умножение (64 бита)	12,66	42,37	99,90	—
Параллельное суммирование	7,52	14,10	21,12	62,70
Параллельное суммирование (оптимизация)	7,18	11,89	18,86	65,00
Комбинированный метод	—	17,65	13,60	52,65

Итоговое сравнения для режима компиляции x64.

Функция	Тип n			
	u8	u16	u32	u64
Наивный метод	37,72	71,51	131,47	227,46
Удаление младшей единички	40,96	69,16	79,13	126,72
Предподсчёт - 1 байт	0,01	1,44	24,79	26,84
Предподсчёт - 2 байта	—	0,07	8,49	13,01
Умножение и остаток (32 бита)	13,88	33,88	—	—
Умножение и остаток (64 бита)	6,78	12,28	31,12	—
Умножение и умножение (64 бита)	3,54	4,51	18,35	—
Параллельное суммирование	8,06	11,89	21,30	22,59
Параллельное суммирование (оптимизация)	8,09	10,27	19,20	19,20
Комбинированный метод	—	10,53	13,60	9,41

Замечание

Ни в коем случае не рассматривайте итоговую таблицу как доказательство в пользу того или иного подхода. Поверьте, что на вашем процессоре и с вашим компилятором некоторые числа в такой таблице будут совершенно иными. К сожалению, мы никогда не можем точно сказать, который из алгоритмов окажется лучше в том или ином случае. Под каждую задачу нужно заточивать конкретный метод, а универсального быстрого алгоритма, к сожалению, не существует.

Я изложил те идеи, о которых знаю сам, но это лишь идеи, конкретные реализации которых в разных комбинациях могут быть очень разными. Объединяя эти идеи разными способами, вы можете получать огромное количество разных алгоритмов подсчёта единичных битов, каждый из которых вполне может оказаться хорошим в каком-то своём случае.





Спасибо за внимание. До новых встреч!


UPD: Инструкция POPCNT из SSE4.2 не включена в список тестирования, потому что у меня нет процессора, который поддерживает SSE4.2.

🔒 битовые трюки, подсчёт битов, битовая магия, битовая алхимия

↑ +82 ↓

👁 30,6k ⭐ 365





Артём @Zealint

109,0 карма -1,8 рейтинг

Пользователь

Похожие публикации

+62

Визитки 2.0: Добавим немного NFC-магии

👁 61,3k

⭐ 273

💬 42

+39

Эксперименты с бит-реверсными паттернами в двумерных аддитивных клеточных автоматах

👁 12k

⭐ 104

💬 11

+19

Трюки с интерфейсами в Delphi


👁 18,6k

⭐ 118

💬 26

Мой круг

Размести IT-вакансию и получи бесплатно 100 самых подходящих по навыкам специалистов

 Узнать подробности



Сейчас Сутки Неделя Месяц

+5 Одна простенькая задачка. Быстро, красиво или чисто?

 5,6k  38  19

+16 Дайджест свежих материалов из мира фронтенда за последнюю неделю №236 (7 — 13 ноября 2016)

 4k  28  2

+7 CleverScrollbar.js — Сайдбар для понятной навигации

 1,1k  9  4

+20 PHP-Дайджест № 96 — интересные новости, материалы и инструменты (1 — 13 ноября 2016)

 3k  26  0





+270 5 способов, которыми игры пытаются вызвать зависимость

 104k  763  245

Комментарии (91)

 **Big_Lebowski** 13 февраля 2016 в 15:19  0 ↑ ↓





Интересно сравнить какое время получилось бы если хранить таблицу с посчитанными значениями

 **Zealint** 13 февраля 2016 в 15:24    +2 ↑ ↓



Не понял вашего вопроса. Ведь в разделе «Предподсчёт» именно это и сделано.

 **Big_Lebowski** 13 февраля 2016 в 15:42    0 ↑ ↓

тьфу, оборвался комментарий — с посчитанными значениями для 64битовых без трюка с разбиением на меньшие числа. То есть, так же получалось бы 0.01 как и для чара или больше? Таблица для всех возможных, ясно, не влезет, пусть для какого-то диапазона.

 **Zealint** 13 февраля 2016 в 15:53    0 ↑ ↓




Трудно сказать, что будет. Я описал лишь приёмы, работающие со всеми возможными числами каждого диапазона, и то не все возможные комбинации этих приёмов удалось проверить (таких комбинаций будет очень много). А Вы предлагаете некий весьма частный случай задачи, который в общем-то в такой абстрактной постановке не имеет смысла. Я не знаю, что будет, но я в принципе приложил немало усилий, чтобы подобные сравнения каждый мог бы повторить у себя сам.

 **Zealint** 13 февраля 2016 в 15:59  +3 ↑ ↓

Поясню один момент, который может быть непонятным сразу. Есть инструкция POPCNT, входящая в состав SSE4.2, которую по-хорошему мне тоже нужно было включить в список тестирования. Однако у меня нет процессора, который поддерживал бы SSE4.2. Сожалею об этом, но ничего поделать не могу.

 **Beholder** 13 февраля 2016 в 16:08  +3 ↑ ↓

Стоило бы упомянуть книгу «Hacker's Delight», в которой есть этот и многие другие подобные трюки.

 **Zealint** 13 февраля 2016 в 16:15    +2 ↑ ↓

Спасибо за совет, но я не стал давать никаких ссылок, потому что все объяснения в этой статье мои собственные (даже если они вдруг случайно повторяют чьи-то другие варианты объяснений, которых я даже не видел), а сами приёмы я узнавал из совершенно разных мест, которые сейчас и не собрать. Я просто поделился своим опытом.

В указанной Вами книге есть только один трюк для подсчёта единичных битов (боюсь соврать, но в моей редакции есть только параллельное суммирование) и один трюк касательно удаления младшего единичного бита.



hellman 13 февраля 2016 в 16:40 # h ↑

+4 ↑ ↓

Тогда ещё graphics.stanford.edu/~seander/bithacks.html



Zealint 13 февраля 2016 в 16:48 # h ↑

0 ↑ ↓

Да, спасибо, там почти весь набор приёмов, что в статье. Но всё равно «почти» и без пояснений.



vk2 14 февраля 2016 в 00:45 # h ↑

0 ↑ ↓

Приятная ссылка, спасибо!

(по теме: я бы все-таки поставил в статью ссылку на [Hamming Weight](#))



Pavlusha 14 февраля 2016 в 01:04 (комментарий был изменён) # h ↑

0 ↑ ↓

Не много кривовато описано



deniskreshikhin 13 февраля 2016 в 16:40 #

+2 ↑ ↓

Аналоги POPCNT кстати есть и на GPU: в CUDA это `__popc` и `__popcll`, и в OpenCL 1.2 `popcount`.



Vapaamies 15 февраля 2016 в 04:05 # h ↑

0 ↑ ↓

Аналог `popcnt` легко реализовать на обычном ассемблере x86, используя `bsf` и сдвиги.



zagayevskiy 13 февраля 2016 в 16:45 #

+5 ↑ ↓

Исходники на Я.Диске — это моветон в 2016-то году. Залейте на гитхаб, будет круто.

За статью респект, очень круто. Кстати, наблюдение — по-моему, вам в карму плюсанули примерно столько же людей, сколько статью:)



Zealint 13 февраля 2016 в 16:51 # h ↑

+4 ↑ ↓

Ну там не только исходники, ещё презентация. Я не вижу смысла специально залезать на GitHub, чтобы просто положить 5 файлов и быть в тренде, и не вижу проблемы скачать архив, смотря на него просто как на файл. Спасибо за отзыв :)



zagayevskiy 13 февраля 2016 в 17:56 (комментарий был изменён) # h ↑

+7 ↑ ↓

У гитхаба по сравнению с просто облачным хранилищем файлов куча преимуществ. Например, вы удалите у себя инфу или переместите её куда-то — в статье останется битая ссылка. Или вы придумаете новый крутой алгоритм — при коммите подписавшиеся увидят это. Также может кто-то ещё дополнить ваши данные пулл-реквестом. Множество других людей, при должном оформлении, сможет увидеть ваши изыскания, даже не зная о существовании хабра. И так далее, и тому подобное. Сплошные плюсы и никаких минусов.



Zealint 13 февраля 2016 в 18:01 # h ↑

+2 ↑ ↓

Что ж, спасибо за совет. Я теперь задумался, может мне имело бы смысл создать специальную ветку на GitHub для моих «Бесед о программировании», ведь я постоянно выкладываю учебный материал к ним пока что на Я.Диск. Сейчас я не планировал как-то его развивать, но, возможно, какой-то код будет совершенствоваться и будет иметь смысл делать обновления, а также давать другим что-то туда дописывать. Я подумаю над этим ещё.



mikhaelkh 13 февраля 2016 в 18:44 #

0 ↑ ↓

В gcc есть функция `__builtin_popcount`, интересно посмотреть на ее положение в таблице.



Zealint 13 февраля 2016 в 19:39 # h ↑

0 ↑ ↓

Конечно, интересно, но у меня есть подозрение, что там реализован либо вызов `POPCNT` из SSE4.2, либо (когда данная функция не поддерживается процессором), что-то из моего набора приёмов. Так что она займёт место рядом с какой-то из уже записанных в таблице строк. Буду приятно удивлён, если кто-то обнаружит, что там написано что-то совершенно необычное, у кого есть исходники GCC, можете проверить.



khim 13 февраля 2016 в 23:17 # h ↑

0 ↑ ↓

Ну исходники-то не секретны, да. Там табличка `__popcount_tabi` сопутствующие функции:
github.com/gcc-mirror/gcc/blob/master/libgcc/libgcc2.c

Основное всё-таки — это дать возможность вызвать «железную» реализацию, если она есть. Она на каком-нибудь PowerPC появилась в незапамятные годы...



RPG 14 февраля 2016 в 00:50 # h ↑

0 ↑ ↓

`POPCNT` не будет использоваться, если не указать флаг `-msse4.2`, что означает либо компиляцию под определённую платформу либо шаманство с weak-функциями и потеря производительности на вызове функции. Если это узкое место, то нужно отдельно компилировать кучу версий — одну для `sse4.2`, одну для `sse4.1`... и так до `sse2` который гарантированно поддерживается для 64-разрядной архитектуры.

К сожалению представленные на ЯД исходники не позволяют мне проверить все варианты, так как я не понял, что с ними нужно делать. Могу сказать только, что на i5 __builtin_popcount выдаёт 0.1 сек для 100000000 итераций с типом int для sse4.2 и 0.3 сек без него. На Xeon паршивый результат даже с sse4.2 — 0.25, без sse — 2 секунды. На core i7 лучший результат — 0.08 сек. На power хоть и есть инструкция popcntw, но её результат тоже не впечатляет — 0.3 сек.



Zealint 14 февраля 2016 в 10:08 # h ↑

0 ↑ ↓

Странно, но я постарался в комментариях описать, что делать. В любом случае, в статье написано, что число тестов равно 2^{32} .

Порядок действий таков: сначала запускаем «пустой» цикл, который не делает ничего, кроме генерации входного потока из 2^{32} чисел. Например, в моём коде генерация выполняется по формуле $n = 19993 * n + 1$, где n имеет размер `u32`. Начав генерацию от $n=0$, мы закончим её на $n=0$, гарантированно пройдя по всем значениям. Измеряем время работы этого пустого цикла.

Затем запускаем этот же цикл, но с вызовом функции подсчёта битов. Измеряем время работы и вычитаем из него время работы пустого цикла — получаем некую оценку на почти чистое время работы функции. Немного грубо, но в пределах получающихся значений времени вполне достаточно для выводов.



RPG 14 февраля 2016 в 14:18 # h ↑

+2 ↑ ↓

Я немного не об этом. Нельзя загрузить этот код, выполнить `make` и получить такую же табличку, верно? У меня есть возможность протестировать код на совершенно разных машинах, но нет времени его детально изучать. Но по крайней мере я выяснил, что ваш код считает время пустого цикла:) Быстрый взгляд показывает, что в программе не хватает интерфейса чтобы прогнать все тесты разом.

Именно поэтому код следует выложить на GitHub, так как его ещё дорабатывать и дорабатывать. Чтобы не считать время пустого цикла я использовал разворот цикла на 8, после чего время, затрачиваемое на перебор значений, стало ничтожно мало.

> гарантированно пройдя по всем значениям

А почему цикл от 0 до N не пройдёт гарантированно все значения?

> чтобы компилятор вообще не удалил цикл за ненужностью

есть `volatile`.

Но хабр не для Pull Request-ов. А про гитхаб уже сказали.



Zealint 14 февраля 2016 в 16:21 # h ↑

0 ↑ ↓

Конечно нельзя так сделать, и не было цели разрабатывать полноценный тестирующий проект, который будет работать у всех и всегда. Раньше я испытывал иллюзии, что это можно сделать, но потом понял, что нельзя. Для измерения процессорного времени использовалась утилита `gipex`, все измерения проводились вручную. Если ставить целью сделать код, который будет измерять процессорное время на любой машине и везде запускаться, то я затратил уйму времени.

Цикл от 0 до N не подходит, потому что мне нужен именно хаотичный поток данных (в противном случае алгоритм предподсчёта, возможно, будет работать неадекватно быстро при последовательном обращении к памяти и это не будет отражать ситуацию в реальных проектах).

Разворачивание цикла в моём случае затруднительно, так как процедура тестирования заключается в последовательном откомментировании одной строки за другой и ручной проверки времени работы. Если развернуть цикл, придётся 8 раз откомментировать и комментировать обратно эти же строки.

В целом, я понимаю, чего Вы хотите, но для меня затраты времени на это сейчас будут настолько катастрофическими, что они не оправдают результата.

По поводу GitHub я уже говорил, что подумаю. Нужно разобраться, привыкнуть — это не быстро для меня.



Randi 15 февраля 2016 в 21:17 # h ↑

0 ↑ ↓

Чем

`std::chrono::high_resolution_clock::now()`
плох?



Zealint 15 февраля 2016 в 21:32 # h ↑

0 ↑ ↓

Как он позволяет определить процессорное время? Именно то время, которое процессор тратил только на мою задачу, а не на ещё полсотни процессов в ОС?



MacIn 15 февраля 2016 в 21:35 # h ↑

0 ↑ ↓

Поскольку ваша задача — "горячая", процессор будет занят постоянно. Объем данных у вас велик, так что погрешность можно опустить. Или нет?



Zealint 15 февраля 2016 в 21:41 # h ↑

0 ↑ ↓

Нет, разница между астрономическим временем и процессорным на моей машине могла составлять от 1 до 3 секунд. Для моих измерений это много. Конечно, я мог бы увеличить число тестов раз в 10, но тогда статья вышла бы где-нибудь только к концу февраля :)



MacIn 15 февраля 2016 в 21:48 # h ↑

0 ↑ ↓

Здесь, возможно, стоит раскидать по ядрам. Но это так, лирика.



Zealint 15 февраля 2016 в 22:00 # h ↑

0 ↑ ↓

Я бы раскидал, если бы ОС позволяла это сделать относительно простым методом. В принципе, в идеале надо было вообще запускать компьютер, сразу перехватывать управление, без запуска ОС, и запускать тестирование функций. Это не сложно, в принципе, но не слишком ли много Вы хотите от обычного учебного материала?



MacIn 15 февраля 2016 в 22:12 # h ↑

0 ↑ ↓

Это не требование к вам, просто мысль вслух.



Randl 15 февраля 2016 в 21:42 # h ↑

0 ↑ ↓

Он позволяет прогнать все тесты одним нажатием клавиши.

Что, кроме собственно вашего удобства, позволит каждому желающему прогнать свои тесты, прогнать ваши тесты на своем компьютере и т.д.

Точность теоретически упадет, но думаю не так уж сильно — полсотни процессов в ОС одни и те же.



Zealint 15 февраля 2016 в 21:58 # h ↑

0 ↑ ↓

Может Вы и правы в том, что я мог бы так сделать, но всё-таки я в конце-концов решил, что пусть каждый делает сравнения так, как ему удобно, беря ответственность на себя. Измерять астрономическое время на данном уровне ответственности мне не позволит мой опыт подобной работы и куча граблей на этой почве, поэтому советовать делать то же самое остальным пользователям я бы не стал. Я не вижу проблемы прикрутить упомянутые выше функции, но сам так делать не буду. Всё-таки у меня не готовый продукт, а описание методов. Кто хочет их изучить подробнее по-любому захочет всё переписать и сделать по-своему (по крайней мере, я бы так сделал). Давать что-то, что имеет (по моим представлениям) большой риск не работать или работать неправильно я просто не могу по личным причинам, потому я и ограничился самым простым минимальным кодом. Думаю, тут можно не спорить, ведь я разрешил делать с моим кодом всё, что угодно.

В принципе, можно ещё было бы много недостатков найти в моей работе, но только если забыть о целях. Изначальная цель — показать методы. Сравнение — это просто как бонус. Именно поэтому если кто-то проведёт полноценное сравнение с более научных позиций — это будет заслуживать отдельной хорошей статьи. Такой человек мог бы взять несколько процессоров с разной архитектурой (от настольных ПК до мобильных), несколько ОС, несколько режимов работы компиляторов, написать варианты с использованием POPCNT, создать скрипт, который везде, всегда и у всех будет по нажатию кнопки проводить тестирование и выдавать таблицы с результатом (а также параллельно варить кофе и делать бутерброды), — в общем, полно интересной работы. Я лишь начал тему, но продолжать её можно ещё долго.



RPG 15 февраля 2016 в 21:47 # h ↑

0 ↑ ↓

Пользуясь случаем, сообщу, что гупехе по ссылке мертва, а ещё меня терзают смутные сомнения, что я смогу повторить запуск этих тестов, например, на power.



Zealint 15 февраля 2016 в 22:04 # h ↑

0 ↑ ↓

Мертва в том смысле, что проект не разрабатывается? В любом случае, там есть рабочая версия для Windows и исходники, у меня всё работает. Сделать что-то большее я не могу, других утилит не знаю.



RPG 15 февраля 2016 в 22:07 # h ↑

0 ↑ ↓

Осталось дожидаться, когда Microsoft сделает Windows для power:)



MacIn 15 февраля 2016 в 22:11 # h ↑

0 ↑ ↓

Есть timeit в Windows Resource Kit. Хотя я не уверен, учитывает ли она время конкретного процесса, или просто измеряет интервал.



encyclopedist 18 февраля 2016 в 17:21 # h ↑

+1 ↑ ↓

i5 i7 и Xeон — это ничего не значащие маркетинговые названия. Гораздо более информативными были бы указания архитектуры (например, Sandy Bridge или Haswell)



gleb_l 14 февраля 2016 в 11:14 #

+2 ↑ ↓

интересно, а нельзя ли использовать флаг четного количества битов в PSW для оптимизации наивного цикла, или цикла с откусыванием единиц?

**alecv** 14 февраля 2016 в 18:30 #

0 ↑ ↓

Надо сделать сопроцессор на FPGA!

**khim** 14 февраля 2016 в 19:04 # h ↑

0 ↑ ↓

Не смешно даже. Обращение к сопроцессору занимает столько времени, что выигрыша никакого не будет, даже если всё будет исполняться за один такт. Такие вещи если и встраивать — то в сам процессор.

P.S. Хотя более сложные вещи реально в FPGA выносят. Но это должно быть что-нибудь как-минимум на несколько тысяч тактов. Иначе смысла нету...

**MacIn** 15 февраля 2016 в 04:41 #

0 ↑ ↓

Ясное дело, что процессор вынужден делать четырёхкратную работу при удвоении входного параметра и даже хорошо, что он справляется всего лишь втрое медленнее.

Там не втрое, там на 70%.

**Randl** 15 февраля 2016 в 06:20 #

+1 ↑ ↓

Если ктонибудь прогнал бы тесты на своем компьютере и сравнил бы с POPCNT, я был бы ему благодарен. К сожалению, компа поддерживающего эту инструкцию под рукой пока нет. Если никто не сподобится, сделаю сам где-то через месяц-полтора.

**Zealint** 15 февраля 2016 в 10:13 # h ↑

0 ↑ ↓

Я тоже был бы рад видеть подобную таблицу с POPCNT. Однако, к сожалению, подобное тестирование занимает очень много времени, человеку, который его проведёт, не стыдно будет написать отдельную статью. Я не запрещаю никому брать и менять свой код по своему усмотрению и с любыми целями. Но и отвечать за него в таком случае не берусь.

**MacIn** 15 февраля 2016 в 19:57 # h ↑

0 ↑ ↓

Более неочевиден проигрыш второго решения с откусыванием единицы. Вы не приведете ассемблерный листинг циклов наивного решения и с откусыванием для вашего компилятора?

**Zealint** 15 февраля 2016 в 20:28 # h ↑

0 ↑ ↓

Пожалуйста, но сомневаюсь, что это чем-то поможет. Выкладываю код, который получается для i8 в режиме компиляции x86. Для наивного метода:

```
81078 mov cl, al
8107A and cl, 1
8107D add dl, cl
8107F shr al, 1
81081 jne main+38h (81078h)
```

Для откусывания единички:

```
281078 lea ecx, [eax-1]
28107B inc dl
28107D and al, cl
28107F jne main+38h (281078h)
```

Для i32 для режима x86. Наивный метод:

```
2C10A0 mov cl, al
2C10A2 and cl, 1
2C10A5 add dl, cl
2C10A7 shr eax, 1
2C10A9 jne main+60h (2C10A0h)
```

Откусывание:

```
FB10A0 lea ecx, [eax-1]
FB10A3 inc dl
FB10A5 and eax, ecx
FB10A7 jne main+60h (FB10A0h)
```

Как видите, тут не всё очевидно :)

**Randl** 15 февраля 2016 в 20:37 # h ↑

0 ↑ ↓

Я ваш код не смотрел, но ИМХО если код организован, много времени это не займет. Допisać буквально пару строк кода, запустить тестирование, подождать пока оно закончится и опубликовать результаты.



zagayevskiy 15 февраля 2016 в 21:05 # h t

0 ↑ ↓

Читайте [выше](#)

Для измерения процессорного времени использовалась утилита `gipex`, все измерения проводились вручную.

процедура тестирования заключается в последовательном откомментировании одной строки за другой и ручной проверки времени работы



Zealint 15 февраля 2016 в 21:34 # h t

0 ↑ ↓

Я измеряю именно процессорное время. Автоматизировать это относительно простыми методами я не могу (не потому что не умею, а потому что это долго и менее важно для меня).



RPG 15 февраля 2016 в 22:05 # h t

0 ↑ ↓

Померил для 32битного `int` — `popcnt` быстрее самого быстрого варианта с таблицей на 64К в 2 с небольшим раза (на `core i5`). Всё подряд предлагаемое прогнать нереально:) Ждём когда автор доберётся до гитхаба. Или найти в гугле готовые тесты... Например, вот этот: http://www.dalkescientific.com/writings/diary/archive/2011/11/02/faster_popcount_update.html



impetus 15 февраля 2016 в 08:07 (комментарий был изменён) #

+3 ↑ ↓

прошу прощения, я абсолютный 0 в программировании на ЯВУ (а статья «не для новичков»), но очень захотелось спросить — вроде же у каждого семейства процессоров и даже микроконтроллеров есть «родные» команды арифметического сдвига вправо с флагом переноса, и сложения с этим же флагом, т.е. подсчёт единичек сводится к циклу соответствующей длины из последовательной пары команд «сдвинуть», «сложить»? Тем более что почти всюду это команды одни из самых простых и быстрых, и к тому же базовые для семейства. т.е. не меняются от версии к версии контроллера или процессора. Просто раз всё равно в статье и комментах речь про компиляции под конкретные процессоры — то насколько хотя бы по порядку величин эти решения медленнее ассемблерных? Или я что-то совсем не правильно понял?

И второй момент — как понимаю у всех тестов низкоуровневого быстродействия результаты буквально кратно меняются в зависимости от того, сидят ли программа и/или её данные в кэше процессора или нет, и соотв это отследить — отдельное мини-исследование — нет ли тут такого момента что измеряются по сути не быстродействие алгоритмов, а просто помещаемость программ или данных в кэш?



Zealint 15 февраля 2016 в 10:06 # h t

0 ↑ ↓

Наивный метод, который идёт в статье первым, компилируется именно в то, о чём вы пишете — последовательное сложение одного бита за другим. Компилятор генерирует вполне оптимальный код для того кода на ЯВУ, что я дал. Как Вы можете убедиться, это самый медленный алгоритм.

В данном случае всё сидит в кэше, можете не сомневаться.



MacIn 15 февраля 2016 в 21:21 (комментарий был изменён) # h t

0 ↑ ↓

Нет, не именно то, это видно по выпрошенным мною у вас листингам. Человек говорит про команду `ADC`. Что-то наподобие

```
xor edx, edx
m1:
shl eax,1
jz m2
adc edx, 0
jmp m1
m2:
adc edx, 0
quit:
```

Т.е. да, все равно по сути длинный "тупой" цикл, но не совсем то же самое.



Zealint 15 февраля 2016 в 21:39 # h t

0 ↑ ↓

Уверен, что это будет то же самое, а скорее всего даже хуже. Кроме того, это потребует работать с ассемблером, тогда как у меня в планы входило создать только переносимый код, хотя бы на Си. Впрочем, не берусь утверждать что-либо наверняка, любые утверждения требуют доказательств. Но если сравнивать с остальными методами, то цикл по всем битам проиграет однозначно, пусть не на `i8`, то на более крупных типах точно. Думаю, Вы тоже это понимаете.



MacIn 15 февраля 2016 в 21:46 (комментарий был изменён) # h t

0 ↑ ↓

Разумеется. И тем не менее, я прогоню цикл с `ADC`, чисто из любопытства. Но измерять буду по машинному времени. Да, кстати, я писал вам выше — в статье у вас есть фраза "второе медленнее", там где должно быть "на 70%".



Zealint 15 февраля 2016 в 22:02 # h t

0 ↑ ↓

У меня написано правильно. Возможно, вы что-то другое поняли.



MacIn 15 февраля 2016 в 22:07 # h ↑

0 ↑ ↓

Ах, вот оно что. Поскольку написано о режиме x86, мне казалось, что говорится о сравнении с 64 разрядным процессором, мол, при вчетверо большем объеме работы (ведь 64 делает за один шаг, для него это естественный объем данных), он проигрывает 64 разрядному втрое. А вы имеете в виду по отношению к 32 битному своему же тесту. Тогда извините.



MacIn 15 февраля 2016 в 22:17 # h ↑

0 ↑ ↓

Да, кстати, вариант с ADC быстрее — на простом тесте 7118,6мс против 8922,6мс



MacIn 15 февраля 2016 в 22:27 # h ↑

0 ↑ ↓

Возможно, это издержки моей выборки исходных данных, потому что метод отсечения единицы отработал еще быстрее — за 5969мс.



Zealint 15 февраля 2016 в 22:28 # h ↑

0 ↑ ↓

Отлично, буду знать :) Вообще, по-хорошему все эти функции можно переписать на ассемблере и они будут работать лучше, если понимать принципы оптимизации, конечно. Только переносимость станет равной почти нулю.



ErmIg 15 февраля 2016 в 16:37 #

0 ↑ ↓

Провел сравнение производительности 2-х функции на Core i7 4770:
Первая использует аппаратную реализацию из SSE4.2.

```
#include <intrin.h>

uint64_t BitCount(const uint8_t * src, size_t size)
{
    uint64_t sum = 0;
    for (size_t i = 0; i < size; i += 8)
        sum += __popcnt64(*uint64_t*)(src + i);
    return sum;
}
```

Вторая комбинацию предрасчитанного массива результатов и SIMD на основе AVX2:

```
#include <immintrin.h>

const __m256i Z = _mm256_set1_epi8(0x0);
const __m256i M = _mm256_set1_epi8(0xFF);
const __m256i K = _mm256_setr_epi8(0, 1, 1, 2, 1, 2, 2, 3, 1, 2, 2, 3, 2, 3, 3, 4, 0, 1, 1, 2, 1, 2, 2, 3, 1, 2, 2, 3, 2, 3, 3, 4);

uint64_t BitCount(const uint8_t * src, size_t size)
{
    __m256i _sum = _mm256_setzero_si256();
    for (size_t i = 0; i < size; i += 32)
    {
        __m256i _src = _mm256_loadu_si256((__m256i*)(src + i));
        _sum = _mm256_add_epi64(_sum, _mm256_sad_epu8(Z, _mm256_shuffle_epi8(K, _mm256_and_si256(_src, M))));
        _sum = _mm256_add_epi64(_sum, _mm256_sad_epu8(Z, _mm256_shuffle_epi8(K, _mm256_and_si256(_mm256_srli_epi16(_src, 8), M))));
    }
    return _sum.m256i_u64[0] + _sum.m256i_u64[1] + _sum.m256i_u64[2] + _sum.m256i_u64[3];
}
```

Производительность 1-й — 114 микросекунд на обработку 1 GB, 2-й — 48 ms.



khim 15 февраля 2016 в 17:06 # h ↑

0 ↑ ↓

Это не вполне честно, но тоже полезно знать.

P.S. Как я понимаю даже обычный SSE на больших объёмах может "уделать" popcnt64? Но только если есть где "развернуться"?



ErmIg 15 февраля 2016 в 17:43 # h ↑

0 ↑ ↓

Да уделяет в 2 раза (всего на 10% медленнее чем AVX2). Использует SSSE3:

```
const __m128i Z = _mm_set1_epi8(0x0);
const __m128i M = _mm_set1_epi8(0xFF);
const __m128i K = _mm_setr_epi8(0, 1, 1, 2, 1, 2, 2, 3, 1, 2, 2, 3, 2, 3, 3, 4);

uint64_t BitCount(const uint8_t * src, size_t size)
{
    // ...
}
```

```

__m128i _sum = _mm_setzero_si128();
for (size_t i = 0; i < size; i += 32)
{
    __m128i _src = _mm_loadu_si128((__m128i*)(src + i));
    _sum = _mm_add_epi64(_sum, _mm_sad_epu8(Z, _mm_shuffle_epi8(K, _mm_and_si128(_src, M))));
    _sum = _mm_add_epi64(_sum, _mm_sad_epu8(Z, _mm_shuffle_epi8(K, _mm_and_si128(_mm_srli_epi16(_src, 4), M))));
}
return _sum.m128i_u64[0] + _sum.m128i_u64[1];
}

```


RPG 15 февраля 2016 в 22:39 #

+1 ↑ ↓

Результаты этого теста:

► [Xeon E5-2680](#)

► [Core i7-3770](#)

► [POWER](#)


snikulov 16 февраля 2016 в 10:50 #

0 ↑ ↓

Кстати, а `std::bitset::count()` было бы тоже неплохо посчитать.

Насколько я помню он оптимально выражался в `popcnt` ну и сделать выводы.


art0int 16 февраля 2016 в 16:08 #

-4 ↑ ↓

А вот про подсчет битов: всегда было интересно, почему в сжатом файле примерно одинаковое количество 0 и 1?

Или вот пример создания файлов, которые жмутся на примерно 50%

crfile.sh

```

-----
test -f ucfile && rm ucfile
touch ucfile
for ((j=1;j<=10000;j++))
do
s=""
for ((i=1;i<=4;i++))
do
x=$((($RANDOM+1)/16384))
test $x -eq 0 && s="01$s"
test $x -eq 1 && s="10$s"
done

n=$(echo «obase=10; ibase=2; $s»|bc)
printf "\$(printf '%03o' "$n")">>ucfile

```

```

done
stat -c "%s %n" ucfile
gzip ucfile
stat -c "%s %n" ucfile.gz
gunzip ucfile.gz
7z a ucfile.7z ucfile 2>&1 >/dev/null
stat -c "%s %n" ucfile.7z
rm ucfile.7z
bzip2 ucfile
stat -c "%s %n" ucfile.bz2
bunzip2 ucfile.bz2

```

```

sh ./crfile.sh
10000 ucfile
5814 ucfile.gz
5785 ucfile.7z
5185 ucfile.bz2

```


khim 16 февраля 2016 в 19:26 # h ↑

0 ↑ ↓

А вот про подсчет битов: всегда было интересно, почему в сжатом файле примерно одинаковое количество 0 и 1?

Это далеко не всегда правда. Более того: архиватор, для которого это всегда правда будет, вообще говоря, хуже, чем тот, для которого это может быть неправдой.

Что правда — так это то, что на типичных файлах вы будете получать такой результат. Но это-то как раз понятно: если было бы не так, что было бы достаточно приделать небольшой постобработчик, который бы мог, в типичном случае, сжать файл ещё. Кто бы с этим смирился?



art0int 16 февраля 2016 в 19:46 # h i

-1 ↑ ↓

Ну я не пытался утверждать, что это истина. Это всего лишь наблюдение, анализировал биты сжатых файлов. Более того, в сжатых файлах цепочки одинаковых битов становятся длинными. Так почему?



mihaile 16 февраля 2016 в 20:01 # h i

+1 ↑ ↓

Потому что хороший архиватор выдает "случайный" файл (иначе его можно было бы еще сжать). А в случайном файле обычно 0 и 1 примерно одинаково.



art0int 16 февраля 2016 в 20:37 # h i

-3 ↑ ↓

Иными словами, не любую случайную последовательность можно сжать. С этим я согласен, а вот то, что в случайном файле 0 и 1 примерно поровну, это вряд ли.



khim 16 февраля 2016 в 22:53 # h i

+2 ↑ ↓

В "случайном файле" 0 и 1 "примерно" поровну. Посмотрите на [Треугольник Паскаля](#), помедитируйте немного над ним.

Hint: подумайте над тем, сколько существует файлов с нулём единичных битов, с одним, с двумя, без нуликов, с одним нуликом и так далее. Когда поймёте что самая большая вероятность в совсем случайном файле встретить ровно половину нуликов и единичек — продолжайте думать, не останавливайтесь! Возможно рано или поздно поймёте о чём тут люди говорят...



mihaile 17 февраля 2016 в 02:57 # h i

0 ↑ ↓

Конечно, сжать можно любую последовательность (для любого файла можно сделать архиватор, сжимающий этот файл в 1 байт). Нельзя сделать универсальный архиватор, сжимающий любую последовательность.

Если вам неочевидно, что в случайном файле примерно поровну 0 и 1, то вы понимаете под случайным файлом что-то очень странное. Приведите свое определение, что ли.



art0int 17 февраля 2016 в 06:28 # h i

-1 ↑ ↓

Про специализированный/универсальный архиватор — да. Под «случайным» файлом Вы имели в виду, наверное, то, что архиватор на выходе выдает файл с псевдослучайной последовательностью? Я не сразу понял. И распределение там биномиальное, $p \sim 0,5$? М.б. это в виде теоремы оформлено? Если знаете, подскажите, пожалуйста.



mihaile 17 февраля 2016 в 13:48 # h i

0 ↑ ↓

У большинства последовательностей число 0 и 1 примерно одинаково. Более формально — доля последовательностей длины n , в которых меньше чем $0.4n$ (или $0.49999n$, неважно) единиц, экспоненциально мала.

У биномиального распределения при $p=0.5$ энтропия максимальна — следовательно, если архиватор выдает другое распределение, его вход типично можно сжимать.



art0int 17 февраля 2016 в 19:01 # h i

-1 ↑ ↓

Ясно. И вот еще, я рассматриваю биты архивного файла. Вижу, что чередующиеся наборы одинаковых битов удлиняются по сравнению с неким произвольным исходным файлом. Может ли быть объяснение этому явлению? Наблюдается для разных архиваторов и разных алгоритмов.



mihaile 17 февраля 2016 в 19:29 # h i

+3 ↑ ↓

Вообще говоря, если вы действительно находите какую-то алгоритмически выразимую закономерность в результатах работы архиватора — то вы можете его улучшить. Т.к. вряд ли хорошие архиваторы можно улучшить нахалю, то скорее всего закономерность кажущаяся.

Если нет — предъявите конкретные числа.



khim 18 февраля 2016 в 00:46 # h i

0 ↑ ↓

Хорошие архиваторы с лёгкостью улучшаются нанемного для определённых типов файлов. На этом, чёрт побери, построено **вообще всё**: если входные данные у вас равновероятны, то построить архиватор **вообще** нельзя.

Так что для любого, самого-самого лучшего архиватора можно найти какой-то тип файлов, на которых его можно сделать ещё лучше. Как вы сами заметили "для любого файла можно сделать архиватор, сжимающий этот файл в 1 байт" (а почему не 1 бит, кстати?).

Вся проблема обсуждения архиваторов — в этой самой "нечестности": на случайных (совсем случайных... белый шум...) данных никто ничего никогда не сожмёт, всегда нужно как-то отбирать — и вот тут-то собака и порылась. Если зафиксировать, скажем, размер архиватора — тогда можно уже что-то обсуждать... но тоже непросто. Если на каких-то данных архиватор начал выдавать что-то совсем непохожее на белый шум — значит вы задали какой-то тип данных для которых этот архиватор никто не заточивал :-)



mihaile 18 февраля 2016 в 01:27 # h i

+1 ↑ ↓

Да, естественно можно подобрать данные, на которых архиватор выдаст строку из одних нулей. Но случайно это сделать (или наткнуться на то, что какой-то формат типично приводит к такому результату) — маловероятно.

Не в 1 бит — потому что не бывает файлов в 1 бит.

Вообще есть еще колмогоровский декомпрессор, который "сильно" улучшить нельзя. Правда соответствующий ему архиватор невычислим.



khim 18 февраля 2016 в 01:44 # h i

0 ↑ ↓

Да, естественно можно подобрать данные, на которых архиватор выдаст строку из одних нулей. Но случайно это сделать (или наткнуться на то, что какой-то формат типично приводит к такому результату) — маловероятно.

А вот и нет. В большинстве случаев достаточно каких-нибудь тупых последовательностей. Чего-нибудь типа 123456789123456789...

Конечно любой уважающий себя архиватор и так сожмёт такую строку в десятки раз. Но можно-то в 1000, 10000, 1000000 раз... вот только **зачем**? Часто вы такие файлы сжимаете на практике? Файлы из одних нулей довольно популярны или какие-нибудь $N \times 0x\text{DEADBEEF}$ (пометка «неиспользуемой» памяти), но более сложные регулярные паттерны встречаются разве только если кто-то хочет «сломать» архиватор специально...



mihaild 18 февраля 2016 в 18:09 # h i

0 ↑ ↓

Ну например я бы предполагал, что любой уважающий себя архиватор будет работать хотя бы не хуже, чем код Хаффмана, что уже почти гарантирует равномерность битового распределения.



art0int 18 февраля 2016 в 10:25 # h i

-2 ↑ ↓

выглядит несколько маньячно :)
размер файла $785816 \times 8 = 6286528$ битов

длина набора одинаковых битов 0, частота
 $s=2830297$

4 55423 т.е. последовательность битов 0000 встречается 55423 раза
3 209814
2 497036
1 985091

длина набора одинаковых битов 1, частота
 $s=3456231$

8 5327 т.е. последовательность битов 11111111 встречается 5327 раз
7 8680
6 23785
5 53878
4 161519
3 139320
2 521863
1 832993

исходный файл, сжатый bz2, размер $390916 \times 8 = 3127328$ битов

сжатый файл, длина набора одинаковых битов 0, частота
 $s=1555468$

26 2
17 1
16 6
15 11
14 29
13 84
12 180
11 405
10 854
9 1707
8 3125
7 5996
6 11862
5 23550
4 47024
3 94969
2 194755
1 404715

сжатый файл, длина набора одинаковых битов 1, частота
 $s=1571860$

935 1 т.е. 1 раз в файле встречается 935 подряд идущих битов = 1

88 1
84 1
71 1
67 2
62 1
61 1
59 1
57 1
51 1
49 1
45 1
44 1
43 6
42 3
41 2
38 2
37 1
35 1
34 1
33 1
30 1
29 2
27 1
26 2
25 5
24 4
23 2
22 5
21 2
20 9
19 14
18 14
17 22
16 26
15 35
14 72
13 89
12 201
11 380
10 582
9 1444
8 2555
7 5339
6 11821
5 24595
4 50123
3 100066
2 195029
1 396805



mihaile 18 февраля 2016 в 18:10 # h ↑

+2 ↑ ↓

Вы же понимаете, что это невозможно прочитать?



art0int 18 февраля 2016 в 20:41 # h ↑

0 ↑ ↓

Думаю, что прочитать можно, но сделать это непросто.



Randl 5 марта 2016 в 15:01 # h ↑

-4 ↑ ↓

Конечно, сжать можно любую последовательность (для любого файла можно сделать архиватор, сжимающий этот файл в 1 байт).

Предлагаю вам доказать это утверждение и выиграть 50000 евро: <http://prize.hutter1.net/>



zagayevskiy 5 марта 2016 в 15:20 # h ↑

+1 ↑ ↓

Имелось в виду "каждый конкретный файл". И архив будет не самораспаковывающийся. В том смысле, что у вас будет архиватор размером 100 Мб и архив очень маленького размера.



Randl 5 марта 2016 в 20:45 # h ↑

0 ↑ ↓

Там и надо запаковать конкретный файл.

А без ограничения на размер декомпрессора само сжатие смысла не имеет. Можно же полностью файл туда засунуть.



khim 5 марта 2016 в 22:49 # h ↑

0 ↑ ↓

Собственно об том и речь. Любой **реальный** архиватор затачивается на файлы определённого типа. То есть вплоть до того что в каком-нибудь 7zip'е есть определённые куски файлов определённых форматов, которые он "знает" и потому может особенно хорошо паковать.

Предельный случай — как вы правильно заметили — особого смысла не имеет, но если архиватор у вас один, а файлов определённого типа вам нужно архивировать много (на практике куда как более часто встречающаяся задача, чем описанный вам challenge), то это — вполне разумное решение.



Randl 6 марта 2016 в 08:06 # h ↑

+1 ↑ ↓

Речь о том, что рассматривать размер сжатых файлов без декомпрессора смысла не имеет, ни в контексте соревнований по сжатию, ни в контексте хранения/передачи файлов. Просто в повседневной жизни объем сжимаемых файлов гораздо больше размера архиватора/деархиватора, а в соревнованиях приходится обговаривать

И в таких условиях (по-моему, достаточно очевидных), "сжать файл до 1 байта" невозможно. На что я и пытался намекнуть



zagayevskiy 6 марта 2016 в 11:06 # h ↑

0 ↑ ↓

Вы сами по своей ссылке хоть ходили?

The Task

Create a compressed version (**self-extracting archive**) of the 100MB file enwik8

Никто здесь не говорит о "повседневной жизни", или возможности запаковать **произвольный файл** в один байт. Говорится о том, чтобы записать **конкретный** файл целиком в архиватор и далее "запаковать" его в 1 байт.

А вы воюете с ветряными мельницами и выглядите глупо.



Randl 6 марта 2016 в 15:06 # h ↑

0 ↑ ↓

Ну я считаю, что скопировать файл в другое место и заменить его одним байтом не является сжатием.

Если вы считаете, что это глупо, это конечно ваше право.

Если бы в изначальном комментарии было написано "запаковать" в кавычках, я бы прошел мимо.



mihaile 7 марта 2016 в 00:21 # h ↑

0 ↑ ↓

А вы не пробовали читать комментарий полностью прежде чем отвечать?

Там написано "нет архиватора, сжимающего любую последовательность".

Проблема в том, что размер разархиватора зависит от архитектуры, от ОС и т.д.

Как, собственно, вводится колмогоровская сложность — рассматривается фиксированный разархиватор, и берется сложность относительно него (мин. длина архива, который он распаковывает в данное слово). Затем оказывается, что существует "универсальный" разархиватор — который с точностью до константы не хуже любого другого.

Проблема в том, что избавиться от этой константы разумными способами нельзя.

Только зарегистрированные пользователи могут оставлять комментарии. [Войдите](#), пожалуйста.

Интересные публикации



H CleverScrollbar.js — Сайдбар для понятной навигации 4

H Дайджест свежих материалов из мира фронтенда за последнюю неделю №236 (7 — 13 ноября 2016) 2

H Одна простенькая задачка. Быстро, красиво или чисто? 19

H РНР-Дайджест № 96 – интересные новости, материалы и инструменты (1 – 13 ноября 2016) 0

CT Израильские разработчики смогли научить ИИ побеждать человека в Mortal Kombat 10

