added: 10 Jun 2008 17:57
Edited: 23 Mar 2012 3:53

Sieve of Eratosthenes

Sieve of Eratosthenes - an algorithm for finding all prime numbers in the interval $[1; n]$for the $O(n \log \log n)$operations.

The idea is simple - write a series of numbers $1 \ldots n$, and will strike out again all numbers divisible by 2, except for the numbers 2 , then dividing by 3, except for the number 3, then on 5, then $7, 11$ and all other simple to n.

Contents [hide]

Implementation

Immediately we present implementation of the algorithm:

```
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;
```

This code first checks all numbers except zero and one, as simple, and then begins the process of sifting composite numbers. To do this, we loop through all the numbers from 2before n, and if the current number $i$is simple, then mark all the numbers that are multiples of him as a constituent.

At the same time we are beginning to go from i^2, as all lower multiples i, be sure to have a prime divisor less i, which means that they have already been screened before. (But as i^2can easily overwhelm type intin the code before the second nested loop is an additional check using the type $long\ long$.)

With this implementation, the algorithm consumes $O(n)$memory (obviously) and performs the $O(n \log \log n)$ action (this is proved in the next section).

Asymptotics

We prove that the asymptotic behavior of the algorithm is $O(n \log \log n)$.

So, for each prime $p \leq n$will run the inner loop, which make $\frac{n}{p}$actions. Therefore, we need to estimate the following value:

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

Let us recall here two known facts: that the number of primes less than or equal to $n$approximately equal $\frac{n}{\ln n}$, and that $k$flashover prime approximately equal $k \ln k$(this follows from the first statement). Then the sum can be written as follows:

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

Here we have identified the first prime of the sum, since the $k = 1$approximation of the $k \ln k$turn 0, that will lead to division by zero.

We now estimate a sum by the integral of the same function on $k$from 2before $\frac{n}{\ln n}$(we can produce such an approximation, because, in fact, refers to the sum of the integral as its approximation by the formula of rectangles):

Primitive of the integrand there . Performing substitution and removing members of the lower order, we get:

Now, returning to the initial sum, we obtain the approximate its assessment:

QED.

More rigorous proof (and giving a more accurate estimate, up to constant factors) can be found in the book of Hardy and Wright "An Introduction to the Theory of Numbers" (p. 349).

Various optimizations sieve of Eratosthenes

The biggest drawback of the algorithm - that he "walks" on the memory permanently departing from the cache, which is why the constant hidden in the $O(n \log \log n)$relatively large.

In addition, for sufficiently large $n$bottleneck becomes the volume of consumed memory.

The following are the methods to both reduce the number of operations performed, as well as significantly reduce the memory consumption.

Sifting simple to the root

The most obvious point - that in order to find all simple to $n$sufficiently perform simple screening only, not exceeding a root n.

This will change the outer loop of the algorithm:

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

On the asymptotic behavior of this optimization does not affect (indeed, repeating above proof, we obtain an estimate that, by the properties of the logarithm, is asymptotically the same), although the number of transactions decreased markedly.

Sieve only odd numbers

Since all even numbers, except 2, - components, we can not process any way at all even numbers and odd numbers operate only.

Firstly, it will halve the amount of memory required. Second, it will reduce the number of operations makes the algorithm by about half.

To reduce the amount of memory consumed

Note that Eratosthenes algorithm actually operates on $n$bits of memory. Therefore, it can save significant memory consumption, storing not $n$bytes - booleans and $n$bits, ie bytes of memory.

However, this approach - **"bit compression"** - substantially complicate handling these bits. Any read or write bits will be of a few arithmetic operations, which ultimately will lead to a slowdown of the algorithm.

Thus, this approach is justified only if $n$sufficiently large that $n$bytes of memory to allocate anymore. Saving memory (in time), we will pay for it a substantial slowing of the algorithm.

In conclusion, it is worth noting that in C ++ containers have already been implemented, automatically performing the bit compression: vector <bool> and bitset <>. However, if speed is important, it is better to implement the compression bit manually, using bit operations - today compilers still not able to generate a reasonably fast code.

Block sieve

Optimization of "simple screening to the root" implies that there is no need to store all the time the whole array . To perform screening sufficient to store only the simple to the root of n, ie , the remainder of the array to build a block by block, keeping the current time, only one block.

Let s- a constant determining the size of the block, then only will the blocks kth block ( ) contains a number in the interval . We processed blocks of the queue, i.e. for each kth block will go through all the simple (as before) and

perform their screening only within the current block. Carefully handle should first block - firstly, from simple do not have to remove themselves, and secondly, the number $0$and should be marked as not particularly easy. When processing the last block should also not forget the fact that the latter desired number $n$is not necessarily the end of the block.

We present the implementation of the sieve block. The program reads the number $n$and finds a number of simple to 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;

}
```

Asymptotic behavior of the sieve block is the same as usual and the sieve of Eratosthenes (unless, of course, the size of $s$the blocks is not very small), but the amount of memory used will be reduced to decrease and "walk" from memory. On the other hand, for each block for each of the simple division is performed, which will strongly affect in a smaller unit. Consequently, the choice of the constant $s$need to keep a balance.

Experiments show that the best performance is achieved when the $s$value is about to .

Upgrade to a linear-time work

Eratosthenes algorithm can be converted to a different algorithm, which is already operational in linear time - see. Article "Sieve of Eratosthenes with linear time work" . (However, this algorithm has some limitations.)

Присоединиться к обсуждению...

Sanat • 2 года назад
а если N=10^9 то программа сколько времени будет работать
2 • Ответить • Поделиться ›

anon • 2 года назад
Столкнулся с тем, что дальше кол-ва 54400028 простых чисел не считает
1 • Ответить • Поделиться ›

anon ➚ anon • 2 года назад
Это про блочную реализацию
• Ответить • Поделиться ›

e_maxx Модератор ➚ anon • 2 года назад
Изменяли ли Вы константу SQRT_MAXN? Не происходит ли переполнений 32-битных чисел? Код написан в предположении, что мы работаем в пределах 32-битных чисел, если это не так - в нескольких местах надо заменить тип на 64-битные числа.
• Ответить • Поделиться ›

anon ➚ e_maxx • 2 года назад
Извините, тут все верно. Обнаружилась ошибка
• Ответить • Поделиться ›

Guest • 2 года назад
Случайно нет ошибки в доказательстве асимтотики? (в третьем абзаце) Ведь если n - k-тое простое число, то n будет приблизительно равен k*ln(n), а не k*ln(k).
1 • Ответить • Поделиться ›

e_maxx Модератор ➚ Guest • 2 года назад
Приблизительно (т.е. асимптотически) это одно и то же. В самом деле, они отличаются между собой в ln(k)/ln(n) раз, что, подставляя сюда k~n/ln(n) и раскрывая логарифм от дроби, приблизительно равно 1 - ln(ln(n))/ln(n) - а эта величина стремится к 1 при n->inf.
1 • Ответить • Поделиться ›

guest • 2 года назад
"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."Неправда, в gcc vector<bool> работает быстрее обычных массивов. Поэтому в случае с этим контейнером время работы только уменьшится.
• Ответить • Поделиться ›

e_maxx Модератор ➚ guest • 2 года назад
Да ладно? Вот за 5 минут набросал кое-какую реализацию ручного bitset, и в сравнении с std::vector и bitset на g++ 4.4.6 -O2 она оказалась на 30% быстрее:
http://pastebin.com/jas5cVdm
std::vector<bool>: 2.62 sec
std::bitset<>: 2.59 sec
my_bvector: 1.84 sec
Та же программа на MS Visual Studio 2010 даёт вообще колоссальную разницу (только это запускалось на другом компьютере, поэтому цифры g++ и MS VS не надо сравнивать между собой):
std::vector<bool>: 3.197 sec
std::bitset<>: 4.982 sec
my_bvector: 2.127 sec