added: 10 Jun 2008 17:57
edited by: Mar 23, 2012 3:53 am

# Eratosthenes sieve

The Eratosthenes sieve is an algorithm that allows you to find all the prime numbers in a segment $[1; n]$for $O(n \log \log n)$operations.

The idea is simple - to write the series of numbers $1 \ldots n$, and we will strike out the first all numbers divisible by $2$, except for the numbers $2$, and then dividing by $3$, except for the numbers $3$, then on $5$, then $7$, $11$and everything else is just up $n$.

## Implementation

Immediately give the 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 marks all numbers, except for zero and one, as simple, and then begins the process of screening out composite numbers. To do this, we cycle through all the numbers from $2$to $n$, and if the current number is $i$prime, then mark all the numbers that are multiples of it, as compound numbers.

At the same time, we start from $i^2$, since all smaller numbers that are multiples $i$have necessarily a smaller divisor $i$, which means that all of them have already been eliminated before. (But since $i^2$it is easy to overflow a type $int$, in the code before the second nested loop an additional test is performed using the type $long\ long$.)

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

## Asymptotics

Let us prove that the asymptotics of the algorithm is $O(n \log \log n)$.

So, for each simple one, the $p \le n$inner loop will be executed, which will perform the $\frac{n}{p}$action. Therefore, we need to estimate the following value:

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

Recall here two well-known facts: that the number of simple, less than or equal $n$, is approximately equal $\frac{n}{\ln n}$, and that $k$the prime number is approximately equal $k \ln k$(this follows from the first statement). Then the sum can be written thus:

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

Here we have identified the first prime from the sum, since, $k = 1$according to the approximation , it $k \ln k$turns out $0$that it leads to division by zero.

Now we estimate this sum using the integral of the same function with respect $k$to $2$w $\frac{n}{\ln n}$(we can produce such an approximation, since, in fact, the sum refers to the integral as its approximation by the rectangle formula):

$$\sum_{k=2}^{\frac{n}{\ln n}} \frac{1}{k \ln k} \approx \int_2^{\frac{n}{\ln n}} \frac{1}{k \ln k} \, dk.$$

There is a primitive for the integrand $\ln \ln k$. Performing the substitution and removing the terms of smaller order, we obtain:

$$\int_2^{\frac{n}{\ln n}} \frac{1}{k \ln k} \, dk = \ln \ln \frac{n}{\ln n} - \ln \ln 2 = \ln(\ln n - \ln \ln n) - \ln \ln 2 \approx \ln \ln n.$$

Now, returning to the original amount, we get its approximate estimate:

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

Q.E.D.

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

## Various optimizations of sieve Eratosthenes

The biggest drawback of the algorithm is that it "walks" through memory, constantly going beyond the cache memory, because of which the constant hidden in $O(n \log \log n)$is relatively large.

In addition, for a large enough $n$bottleneck becomes the amount of memory consumed.

The following methods are considered, allowing both to reduce the number of operations performed and to significantly reduce memory consumption.

### Sieving simple to root

The most obvious point is that in order to find all the simple ones $n$, it is enough to perform sieving with simple ones that do not exceed the root of $n$.

Thus, the external cycle of the algorithm will change:

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

This optimization does not affect the asymptotics (in fact, after repeating the proof given above, we get an estimate $n \ln \ln \sqrt{n} + o(n)$that, according to the properties of the logarithm, it is asymptotically the same), although the number of operations will noticeably decrease.

### Sieve only by odd numbers

Since all even numbers, except for $2$, are compound, you can not even process even numbers at all, and operate only with odd numbers.

First, it will allow to reduce the amount of required memory by half. Secondly, it will reduce the number of operations performed by the algorithm by about half.

### Reducing the amount of memory consumed

Note that the Eratosthenes algorithm actually operates with $n$memory bits. Consequently, it is possible to save a significant amount of memory consumption by storing non- $n$bytes-variables of the Boolean type, and a $n$bit, ie, $n/8$byte of memory.

However, such an approach - **"bit compression"** - will significantly complicate the operation of these bits. Any reading or writing a bit will be a few arithmetic operations, which in the end will lead to a slowdown of the algorithm.

Thus, this approach is justified only if it is $n$so large that it $n$is impossible to allocate memory bytes anymore. Having saved memory (at $8$times), we will pay for this by a significant slowdown of the algorithm.

In conclusion, it should be noted that in C ++, containers are already implemented that automatically perform bit compression: vector <bool> and bitset <>. However, if the speed of work is very important, then it is better to implement bit compression manually, with the help of bit operations - to date, compilers are still unable to generate a sufficiently fast code.

## Block Sieve

From optimization, "simple-to-root sifting" implies that there is no need to store the entire array all the time $prime[1 \ldots n]$. To perform sieving, it is enough to store only simple ones up to the root of $n$, ie. $prime[1 \ldots \sqrt{n}]$, and $prime$build the rest of the array in blocks, storing at the current time only one block.

Let $s$- a constant that determines the size of the block, then there will be only $\lceil \frac{n}{s} \rceil$blocks, the $k$-th block ( $k = 0 \ldots \lfloor \frac{n}{s} \rfloor$) contains numbers in the segment $[ks; ks + s - 1]$. We will process the blocks in turn, i.e. for each $k$block we will go through all the simple (from $1$before $\sqrt{n}$) and perform them sifting only within the current block. Gently should handle the first unit - firstly, from the simple $[1; \sqrt{n}]$do not remove themselves, and secondly, the number $0$and $1$have specifically marked as not simple. When processing the last block, you should also remember that the last desired number is $n$not necessarily at the end of the block.

Let's describe the implementation of a block sieve. The program reads the number $n$and finds the number of simple from $1$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;

}
```

The asymptotics of the block sieve are the same as the conventional sieve of Eratosthenes (unless, of course, the size of the $s$blocks is very small), but the amount of memory used will be reduced to $O(\sqrt{n} + s)$and the "wandering" from memory will decrease. But, on the other hand, for each block, for each prime $[1; \sqrt{n}]$, division will be performed, which will have a big impact with smaller block sizes. Therefore, when choosing a constant, $s$you must balance it.

As shown by experiments, the best speed of work is achieved when it $s$has a value from about $10^4$to $10^5$.

## Improvement to linear operating time

The algorithm of Eratosthenes can be transformed into another algorithm, which will already work for linear time - see the article "Lattice Eratosthenes with linear time" . (However, this algorithm has drawbacks.)

12 Комментариев e-maxx 1 Войти

Рекомендовать 14 Поделиться Лучшее в начале

Присоединиться к обсуждению...

ВОЙТИ С ПОМОЩЬЮ ИЛИ ЧЕРЕЗ DISQUS ?

Имя

**Николай Арзубов** • 4 года назад

На мой взгляд следующая реализация немного проще и работает быстрее:

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

При нахождении количества простых до 10^8 на моём компьютере авторская реализация работает за 1.947s, а моя реализация за 1.331

8 • Ответить • Поделиться ›

**Sanat** • 6 лет назад

а если N=10^9 то программа сколько времени будет работать

3 • Ответить • Поделиться ›

**anon** • 6 лет назад

Столкнулся с тем, что дальше кол-ва 54400028 простых чисел не считает

1 • Ответить • Поделиться ›

**anon** → anon • 6 лет назад

Это про блочную реализацию

• Ответить • Поделиться ›

**e_maxx** Модератор → anon • 6 лет 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.
^ | v • Ответить • Поделиться ›

**anon** ➦ e_maxx • 6 лет назад
Извините, тут все верно. Обнаружилась ошибка
^ | v • Ответить • Поделиться ›

**guest** • 6 лет 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.
1 ^ | v • Ответить • Поделиться ›

**e_maxx** Модератор ➦ guest • 6 лет 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

ЧЯДНТ?
3 ^ | v • Ответить • Поделиться ›

**Guest** • 6 лет 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 ^ | v • Ответить • Поделиться ›

**e_maxx** Модератор ➦ Guest • 6 лет 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 ^ | v • Ответить • Поделиться ›

**Akniyet** • 7 месяцев назад
Почему char? Может bool?
^ | v • Ответить • Поделиться ›

**Диас** • 10 месяцев назад
У меня вопрос .Почему в реализацие(самая верхняя) j=i*i;
Не правильние будет писать j=i*2||j=i+i;?
^ | v • Ответить • Поделиться ›

ТАКЖЕ НА **E-MAXX**