Posted: 6 Sep 2011 1:03
Edited: 23 Mar 2012 3:58

Sieve of Eratosthenes with linear time work

Contents [hide]

- Sieve of Eratosthenes with linear time work
  - Description of the algorithm
  - Implementation
  - Proof of correctness
  - Time and the required memory
  - Literature

Given a number n. You want to find **all the prime** in the interval $[2; n]$.

The classic way to solve this problem - **the sieve of Eratosthenes** . This algorithm is very simple, but it works for a while $O(n \log \log n)$.

Although at the moment we know a lot of algorithms working in sublinear time (ie $o(n)$), the algorithm described below is interesting for its **simplicity** - it is practically difficult to classical sieve of Eratosthenes.

In addition, the algorithm presented here as a "side effect" is actually computes **a factorization of all the numbers** in the interval $[2; n]$, which can be useful in many practical applications.

The drawback of the algorithm is driven by the fact that it uses **more memory** than the classical sieve of Eratosthenes: start requires an array of $n$numbers, while the classical sieve of Eratosthenes only enough $n$bits of memory (which is obtained in 32half).

Thus, the described algorithm should be applied only up to the order of numbers 10^7, no more.

Authorship algorithm apparently belongs Grice and Misra (Gries, Misra, 1978 - see. Bibliography at the end). (And, in fact, to call this algorithm "Sieve of Eratosthenes" correctly too the difference between these two algorithms.)

Description of the algorithm

Our goal - to find each number $i$of the segment in $[2; n]$its **minimal prime divisor** $lp[i]$.

In addition, we need to keep a list of found primes - let's call it an array $pr[]$.

Initially all values are $lp[i]$filled with zeros, which means that we are assuming all the numbers simple. In the course of the algorithm, this array will be gradually filled.

We now sort out the current number $i$of 2up n. We can have two cases:

- $lp[i] = 0$- This means that the number i- easy because for it had not found other subgroups.

  Therefore, it is necessary to assign $lp[i] = i$and add $i$to the end of the list $pr[]$.
- $lp[i] \neq 0$- This means that the current number i- a composite, and it is a minimal prime divisor $lp[i]$.

In both cases, then begins the process of **alignment of values** in the array $lp[]$: we will take the number, **multiples** i , and update their value $lp[]$. However, our goal - to learn how to do this so that in the end each of the value $lp[]$would be set more than once.

Argues that it is possible to do so. Consider the number of the form:

$$x_j = i \cdot p_j,$$

where the sequence p_j- it's simple, do not exceed $lp[i]$(just for this, we need to keep a list of all primes).

All properties of this kind places a new value $lp[x_j]$- obviously, it will be equal p_j.

Why such an algorithm is correct, and why it works in linear time - see. Below, but for now we present its implementation.

Implementation

Sieve performed until the number of the constant N.

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

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

This implementation can speed up a little, getting rid of the vector pr (replacing it with a regular array with counter), as well as getting rid of duplicate multiplication in the nested loop for (which result is the product must be easy to remember in any variable).

Proof of correctness

We prove **the correctness of** the algorithm, ie he correctly puts all the values $lp[]$, and each of them will be set only once. This will imply that the algorithm works in linear time - as all the other steps of the algorithm is obviously working for $O(n)$.

For this, note that any number of **unique representation** of this form: i

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

where $lp[i]$ - (as before) a minimal prime divisor of the number i, and the number x has no divisors less $lp[i]$, ie .:

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

Now compare this to what makes our algorithm - it is actually for everyone x through all simple, for which it can multiply, ie Just prior $lp[x]$ inclusive, to obtain the number of the above representation.

Consequently, the algorithm does take place for each composite number exactly once, putting it the correct value $lp[]$.

This means the correctness of the algorithm and the fact that it runs in linear time.

Time and the required memory

Although the asymptotic behavior of $O(n)$ the asymptotic behavior of the best $O(n \log \log n)$ classical sieve of Eratosthenes, the difference between them is small. In practice this means a two-fold difference in the speed and optimized versions sieve of Eratosthenes and does not lose the algorithm given here.

Given the cost of memory, which requires the algorithm - array $lp[]$ length n and an array of all the simple $pr[]$ length about $n / \ln n$ - this algorithm seems to be inferior to the classical sieve on all counts.

However, it makes the fact that the array $lp[]$, which is calculated by the algorithm, allows you to find the factorization of any number in the interval $[2; n]$ of the time order of the size of the factorization. Moreover, the cost of one more additional array, you can do that in this factorization is not required of the division operation.

Knowledge of the factorization of all the numbers - very useful information for some tasks, and this algorithm is one of the few that allow you to look for it in linear time.

Literature

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