Sieve of Eratosthenes with linear time work

Given the 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 works in time $O(n \log \log n)$.

Although there is currently a lot of known algorithms running 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" actually computes **a factorization of all the numbers** in the interval $[2; n]$, which can be useful in many practical applications.

Disadvantage driven algorithm is that it uses **more memory** than the classical sieve of Eratosthenes: requires start 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 is only used to sense properties of the order of 10^7no more.

Authorship algorithm apparently belongs Grice and Misra (Gries, Misra, 1978 - see references at the end). (And, in fact, call the algorithm "Sieve of Eratosthenes" correctly: too different these two algorithms.)

Description of the algorithm

Our goal - to count for each number $i$in the interval from $[2; n]$its **minimal prime divisor** $lp[i]$.

In addition, we need to keep a list of all primes found - 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 iterate the current number $i$of 2up n. We can have two cases:

- $lp[i] = 0$- Means by which i- just as for it has not found any other divisors.
- Consequently, 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 of i- a compound, and is the minimum 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 to do it so that in the end each number value $lp[]$would be set more than once.

It is argued that this can do so. Consider the number of the form:

$$x_j = i \cdot p_j,$$

where the sequence p_j- it's simple, not exceeding $lp[i]$(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, in the meantime let its implementation.

Implementation

Sieve is performed before said in a constant number 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 accelerate a bit, getting rid of the vector pr(replacing it with a regular array with counter), as well as getting rid of duplicate multiplying a nested loop for(which result works just have to remember a particular variable).

Proof of correctness

We prove **the correctness of** the algorithm, ie, it puts everything in the correct values $lp[]$, and each of them will be installed only once. This will imply that the algorithm works in linear time - since all other steps of the algorithm obviously work for $O(n)$.

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

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

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

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

Now compare this with the fact that our algorithm does - it actually for each x enumerates all simple for which it can multiply, ie Just prior to $lp[x]$and including that number obtained in the above introduction.

Therefore, the algorithm really pass 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 memory required

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

Given the cost of memory required for this algorithm - array $lp[]$length $n$and an array of all prime $pr[]$length about $n / \ln n$- this algorithm seems inferior to the classical sieve on all counts.

However, it makes that the array $lp[]$, which is calculated by the algorithm, allows you to search any number factorization in the segment $[2; n]$during the order of the size of this factorization. Moreover, the price of one additional array can be made to a factorization of the division operation is not needed.

Knowing 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]