Sieve of Eratosthenes Having Linear Time Complexity

Table of Contents

Given a number n, find all prime numbers in a segment $[2; n]$.

The standard way of solving a task is to use the sieve of Eratosthenes. This algorithm is very simple, but it has runtime $O(n \log \log n)$.

Although there are a lot of known algorithms with sublinear runtime (i.e. $o(n)$), the algorithm described below is interesting by its simplicity: it isn't any more complex than the classic sieve of Eratosthenes.

Besides, the algorithm given here calculates **factorizations of all numbers** in the segment $[2; n]$ as

a side effect, and that can be helpful in many practical applications.

The weakness of the given algorithm is in using more memory than the classic sieve of Eratosthenes': it requires an array of n numbers, while for the classic sieve of Eratosthenes it is enough to have n bits of memory (which is 32 times less).

Thus, it makes sense to use the described algorithm only until for numbers of order 10^7 and not greater.

The algorithm's authorship appears to belong to Gries & Misra (Gries, Misra, 1978: see references in the end of the article). And, strictly speaking, this algorithm shouldn't be called "sieve of Eratosthenes" since it's too different from the classic one.

Algorithm

Our goal is to calculate **minimum prime factor** $lp[i]$ for every number i in the segment $[2; n]$.

Besides, we need to store the list of all the found prime numbers - let's call it $pr[]$.

We'll initialize the values $lp[i]$ with zeros, which means that we assume all numbers are prime. During the algorithm execution this array will be filled gradually.

Now we'll go through the numbers from 2 to n. We have two cases for the current number i:

- $lp[i] = 0$ - that means that i is prime, i.e. we haven't found any smaller factors for it.
  Hence, we assign $lp[i] = i$ and add i to the end of the list $pr[]$.

- $lp[i] \neq 0$ - that means that i is composite, and its minimum prime factor is $lp[i]$.

In both cases we update values of $lp[]$ for the numbers that are divisible by i. However, our goal is to learn to do so as to set a value $lp[]$ at most once for every number. We can do it as follows:

Let's consider numbers $x_j = i \cdot p_j$, where p_j are all prime numbers less than or equal to $lp[i]$ (this is why we need to store the list of all prime numbers).

We'll set a new value $lp[x_j] = p_j$ for all numbers of this form.

The proof of correctness of this algorithm and its runtime can be found after the implementation.

Implementation

```
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]<=l
            lp[i * pr[j]] = pr[j];
    }
```

We can speed it up a bit by replacing vector pr with a simple array and a counter, and by getting rid of the second multiplication in the nested `for` loop (for that we just need to remember the product in a variable).

Correctness Proof

We need to prove that the algorithm sets all values $lp[]$ correctly, and that every value will be set exactly once. Hence, the algorithm will have linear runtime, since all the remaining actions of the algorithm, obviously, work for $O(n)$.

Notice that every number i has exactly one representation in form:

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

where $lp[i]$ is the minimal prime factor of i, and the number x doesn't have any prime factors less than $lp[i]$, i.e.

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

Now, let's compare this with the actions of our algorithm: in fact, for every x it goes through all prime numbers it

could be multiplied by, i.e. all prime numbers up to $lp[x]$ inclusive, in order to get the numbers in the form given above.

Hence, the algorithm will go through every composite number exactly once, setting the correct values $lp[]$ there. Q.E.D.

Runtime and Memory

Although the running time of $O(n)$ is better than $O(n \log \log n)$ of the classic sieve of Eratosthenes, the difference between them is not so big. In practice that means just double difference in speed, and the optimized versions of the sieve run as fast as the algorithm given here.

Considering the memory requirements of this algorithm - an array $lp[]$ of length n, and an array of $pr[]$ of length $\frac{n}{\ln n}$, this algorithm seems to worse that the classic sieve in every way.

However, its redeeming quality is that this algorithm calculates an array $lp[]$, which allows us to find factorization of any number in the segment $[2; n]$ in the time of the size order of this factorization. Moreover, using just one extra array will allow us to avoid divisions when looking for factorization.

Knowing the factorizations of all numbers is very useful for some tasks, and this algorithm is one of the few which allow to find them in linear time.

References

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