Sieve of Eratosthenes is an algorithm for finding all the prime numbers in a segment $[1, n]$ using $O(n \log \log n)$ operations.

The algorithm is very simple: at the beginning we write down all numbers between 2 and n. We mark all proper multiples of 2 (since 2 is the smallest prime number) as composite. A proper multiple of a number x, is a number greater than x and divisible by x. Then we find the next number that hasn't been marked as composite, in this case it is 3. Which means 3 is prime, and we mark all proper multiples of 3 as composite. The next unmarked number is 5, which is the next prime number, and we mark all proper multiples of it. And we continue this procedure until we have processed all numbers in the row.

In the following image you can see a visualization of the algorithm for computing all prime numbers in the range $[1, 16]$. It can be seen that quite often we mark numbers as composite multiple times.

The idea behind is this: A number is prime, if none of the smaller prime numbers divides it. Since we iterate over the prime numbers in order, we already marked all numbers, which are divisible by at least one of the prime numbers, as divisible. Hence if we reach a cell and it is not marked, then it isn't divisible by any smaller prime number and therefore has to be prime.

Implementation

```
int n;
vector<bool> is_prime(n+1, true);
is_prime[0] = is_prime[1] = false;
for (int i = 2; i <= n; i++) {
    if (is_prime[i] && (long long)i * i <= n) {
        for (int j = i * i; j <= n; j += i)
            is_prime[j] = false;
    }
}
```

This code first marks all numbers except zero and one as potential prime numbers, then it begins the process of sifting composite numbers. For this it iterates over all numbers from 2 to n. If the current number i is a prime number, it marks all numbers that are multiples of i as composite numbers, starting from i^2. This is already an optimization over naive way of implementing it, and is allowed as all smaller numbers that are multiples of i necessary also have a prime factor which is less than i, so all of them were already sifted earlier. Since i^2 can easily overflow the type `int`, the additional verification is done using type `long long` before the second nested loop.

Using such implementation the algorithm consumes $O(n)$ of the memory (obviously) and performs $O(n \log \log n)$ (see next section).