# The Sieve of Eratosthenes

Hello, and welcome to Competitive Programming! Today we are going to talk about calculating prime numbers, in particular using the Sieve of Eratosthenes.

## Objectives

Your objective it to be able to implement the Sieve, and also to explain some of the reasoning behind why it works.

You will also see some applications involving prime numbers.

# The Sieve

## Calculating Primes the Hard Way

Suppose you have a number `p` that you want to factor, or else discover if it is prime or not.

One thing you could do it just loop over all the numbers smaller than `p` and see if they divide `p` or not.

This, of course, is going to be slow.

## Improvement 1 - skip the even numbers

We can double the speed with one simple observation: if we have already ruled out that the number is divisible by 2, then we have also ruled out that it is divisible by

any multiple of 2.

So here we check divisibility by 2 first, and then check all the odd numbers.

## Improvement 2 - Stop at sqrt(p)

Another optimization is that we don't really need to check everything up to `p`. We can check up to `p`'s square root.

Here's why: suppose there is a factor (q) that is bigger than the square root of p. It can't be (p) itself, since it is smaller than $p$. This means it must be multiplied by some factor (k) so that (k q = p).

Now, if (k) is greater than the square root of (p) then (k q) itself would be greater than (p). Therefore, (k) must be smaller than (q). And if that's the case, then we would have already checked (k) by the time we got to (q). So the check on (q) would be redundant.

This is good, so now we have a method that is O(sqrt p). But we are still testing all the integers in that range. What we'd really like to do is only test the primes.

To do that we are going to have to make a list of the prime numbers.

## The Sieve

Here is the code for the Sieve of Eratosthenes, developed by Eratosthenes of Alexandria.

We will use a bitset called `bs` to keep track of the integers and whether or not they are prime. We will also keep a vector `primes` to have the list handy.

When we start off, we set all the bit in `bs` to 1 to indicate that they are prime. The sieve is going to work by zeroing out all the elements that are a multiple of a prime.

To start, we zero out elements 0 and 1, and then enter our main loop. The i loop checks the bit set until it finds the next prime number. Then the j loop starts off at i squared and crosses out the multiples of i. Once that is done, we put that prime i into the list of primes.

That's it. The algorithm runs in O(n log log n), roughly.

On my laptop, purchased in 2017, it could calculate the primes lower than 10 million in one twentieth of a second. It calculated the primes to 100 million in less than one second, but in many problems that will put you dangerously close to the time limit.

## Factoring Integers

One important applications of primes is factoring integers. Here is the code. You do it by starting with 2 and incrementally picking the next largest prime number. If you find a prime that is a factor, divide it off repeatedly until it's not a factor anymore, and then move to the next prime.

Next
Transcript — Stacks and Queues →