Number Theory: - Prime Number Generation

*“Mathematics is the queen of the sciences, and number theory is the queen of mathematics.”*

Requirements: - Basic School level Arithmetic

# **Basic Definitions**

* Prime Number: - A prime number is a number which has exactly two factors – 1 and the number itself.
* Composite Number: - A number is composite if it has more than two factors.  
  1 and 0 are neither Prime, nor Composite.
* Prime Factorization: - Any composite number can be written as the product of its “prime factors”.   
    
  Consider 24. It can be written as 2\*2\*2\*3, or (2^3)\*3

# **Interesting Info**

One of the most intriguing facts about Number Theory, is that conjectures that very simple to state, are deviously hard to prove. Some Open Conjectures in Number Theory: -

* Goldbach’s Conjecture ( <https://en.wikipedia.org/wiki/Goldbach%27s_conjecture> )
* Collatz Conjecture (<https://en.wikipedia.org/wiki/Collatz_conjecture> )
* Twin Prime Conjecture (<https://www.britannica.com/topic/twin-prime-conjecture> )

# **Prime Number Generation**

Generating prime numbers, or prime factorising composite numbers is one of the most important fields of study in Number Theory.

This problem also comes up often in Competitive Programming, and one of the main applications of Prime Numbers is in Cryptography (RSA Encryption is based on the difficulty of factorising very large composite numbers, think 2048 bits or numbers up to 2^2048, or 616-digit numbers).

So, let’s get on to generating Prime Numbers. Suppose you need to find prime numbers up to n. How would you go about doing it?

# Naïve Method

The naïve method, would be to iterate for all numbers from 2 to n, and trying to divide it by all numbers less than it, if no number less than x can divide x, then x is a prime number, otherwise x is a composite number.

## Time Complexity

Let’s calculate the Time Complexity (approx. order of number of operations required) for this solution.

1 + 2 + 3 + … + n = n\*(n+1)/2 ~ O (n^2)

This means, if we have to find prime numbers upto 10^7, we will require the order of 10^14 operations, which is > 1,000,000 seconds, which is greater than 277 hrs! (Considering standard MIPS for competitive contests)

This means, that if we need to find prime numbers up to 10 million using this method, we will take more than 11 days!

Now, consider if we have to find Prime numbers up to 10^8, we will take 10^16 operations, which is 10^8 seconds, or approx. 28,000 hrs, or more than 3 years!

Fortunately, we can do much better.

# Sieve of Eratosthenes

The sieve of Eratosthenes, is an ancient algorithm, attributed to Eratosthenes (Greek, c. 276 BC – c.195 BC).

It is the most practical techniques of prime number generation, even though some other algorithms (see Sieve of Atkin) have better asymptotic Time Complexity O (N), owing to the fact that it is extremely simple to implement and doesn’t have a high constant. Additionally, the difference between O (N) and O (Nlog(N)) is usually not very significant.

The main idea behind Sieve of Eratosthenes, is that in order to find prime numbers, it is sufficient to just identify the **composite numbers**, and if a number is not marked as composite, it must be prime.

We know that every composite number can be represented as a multiple of a prime number. We use this information to identify all composite numbers. To demonstrate the idea, suppose we have to find primes from 2 to 10.

2, 3, 4, 5, 6, 7, 8, 9, 10 (Identified composites are bolded)

We know that 2 is a prime number, and thus all multiples of 2 **MUST be composite numbers.**

**->** 2, 3, **4**, 5, **6,** 7, **8**, 9, **10**

Primes so far: 2

Now, we go to the next number which is **not composite**, ie 3. Since 3 is not a composite number, it **MUST be a prime number**. Also, all multiples of 3 (6, 9) must be composites.

->2, 3, **4**, 5, **6,** 7, **8**, **9**, **10**

Primes so far: 2, 3

Now, 4 is a composite number, so we skip it.

Now, we go to 5, since 5 is not bolded, we know that it’s a prime, so we repeat the process for 5.

-> 2, 3, **4**, 5, **6,** 7, **8**, **9**, **10**

Primes so far: 2, 3, 5

Now, 6 is a composite number, so we skip it.  
7 is a prime number, but no multiples of it are in the range of 1-10, so we don’t need to mark anything.  
This continues till all the numbers are exhausted.

-> Primes: 2, 3, 5, 7.

We have succeeded in finding all the Prime Numbers from 1 to 10!

## Correctness

I will leave the exercise of proving the correctness of the algorithm to the reader.  
Hint: - You need to prove that the algorithm will properly mark all composite numbers, and that no prime numbers will be marked.

## Optimizations

Now, let’s try to refine the previously defined algorithm.

Ask yourself, are there any numbers we are needlessly visiting and trying to mark?  
The answer is yes, of course.  
Consider when we mark the multiples of 5. We will try to mark 5, 10, 15, 20, 25 …   
We already will have marked: -  
10 (while processing multiples of 2),  
 15 (while processing multiples of 3),  
20 (while processing multiples of 2).

The first multiple of 5, is the number 25 (5\*5). As you will notice, we need not start marking multiples of n starting from 2\*n, but rather n\*n.  
Try to figure out why exactly this property holds.  
(Hint: - When did we mark 2\*n, 3\*n, 4\*n, …, (n-1)\*n before?)

## Time Complexity

Time taken for marking multiples of x from 1 – n : - n/x – (x – 1) (Starting from x\*x)

-> (n/2 + n/3 + n/5 + n/7 + … n/p) – (1 + 2 + 4 + 6 + …)

-> n (1/2 + 1/3 + 1/5 + 1/7 + 1/11 + …) – K

which is > n(1/2 + 1/3 + 1/5 + 1/7 + …)

By using Mertens’ 2nd Theorem (not necessary to know), we get -> n ln(ln n)

Therefore, time complexity = O (n\*log(log n))

This means that for n = 10^7, to find all prime numbers up to n, we will take approx. 0.45s!  
Compared to the previous result (277 hrs), this is lightning fast!

## **Implementation**

* Have a Boolean array, say not\_prime[N+1], where N is the largest number to check, initially initialized to 0.
* not\_prime[2] = 0, ie 2 **IS** a prime
* Now, iterate from i = 2 to i = N, and if is\_not\_prime[i] = 0, we know that i is a prime.
* For **every i that is a prime**, iterate j = i\*i to j <= N, incrementing j by i.
* Mark not\_prime[j] = 1, this means that j is a composite number.
* After the double loop ends, all positions in not\_prime, which are 0, are prime numbers.

## **Problems**

UVa Judge 11408 – Count DePrimes

SPOJ PRIMES-Printing Primes

SPOJ PRIME1-Prime Generator (May require more knowledge on the sieve)