# Basic Number Theory-2

The concept of prime numbers is a very important concept in math. This article discusses the concept of prime numbers and related properties.

**What are prime numbers and composite numbers?**

Prime numbers are those numbers that are greater than 1 and have only two factors 1 and itself.

Composite numbers are also numbers that are greater than 1 but they have at least one more divisor other than 1 and itself.

For example, 5 is a prime number because 5 is divisible only by 1 and 5 only. However, 6 is a composite number as 6 is divisible by 1, 2, 3, and 6.

There are various methods to check whether a number is prime.

**Naive approach**

Traverse all the numbers from $1$ to $N$ and count the number of divisors. If the number of divisors is equal to 2 then the given number is prime, else it is not.

```
void checkprime(int N){
        int count = 0;
        for( int i = 1;i <= N;++i )
            if( N % i == 0 )
                count++;
        if(count == 2)
            cout << N << “ is a prime number.” << endl;
        else
            cout << N << “ is not a prime number.” << endl;
    }
```

*Time complexity*

The time complexity of this function is *O(N)* because you traverse from $1$ to $N$.

*Better approach*

If you have two positive numbers $N$ and $D$, such that $N$ is divisible by $D$ and $D$ is less than the square root of $N$.

- $(N/D)$ must be greater than the square root of $N$.
- $N$ is also divisible by $(N/D)$. If there is a divisor of $N$ that is less than the square root of $N$, then there will be a divisor of $N$ that is greater than square root of $N$. You will have to traverse till the square root of $N$.

**Note**: You are generating all the divisors of $N$ and if the count of divisors is greater than 2, then the number is composite.

For example, if N=50, $\sqrt{N}$=7 (floor value). You will iterate from 1 to 7 and count the number of divisors of N. The divisors of $N$ are 1, 50; 2, 25; 5,10. You have 6 divisors of 50, and therefore, it is not prime.

```
void checkprime(int N) {
        int count = 0;
        for( int i = 1;i * i <=N;++i ) {
             if( N % i == 0) {
                 if( i * i == N )
                         count++;
                 else       // i < sqrt(N) and (N / i) > sqrt(N)
                         count += 2;
```

```
                }
            }
            if(count == 2)
                cout << N << “ is a prime number.” << endl;
            else
                cout << N << “ is not a prime number.” << endl;
    }
```

*Time complexity*

The time complexity of this function is $O(\sqrt{N})$ because you traverse from 1 to $\sqrt{N}$.

**Sieve of Eratosthenes**

You can use the *Sieve of Eratosthenes* to find all the prime numbers that are less than or equal to a given number N or to find out whether a number is a prime number.

The basic idea behind the Sieve of Eratosthenes is that at each iteration one prime number is picked up and all its multiples are eliminated. After the elimination process is complete, all the unmarked numbers that remain are prime.

*Pseudo code*

1. Mark all the numbers as prime numbers except 1
2. Traverse over each prime numbers smaller than sqrt(N)
3. For each prime number, mark its multiples as composite numbers
4. Numbers, which are not the multiples of any number, will remain marked as prime number and others will change to composite numbers.

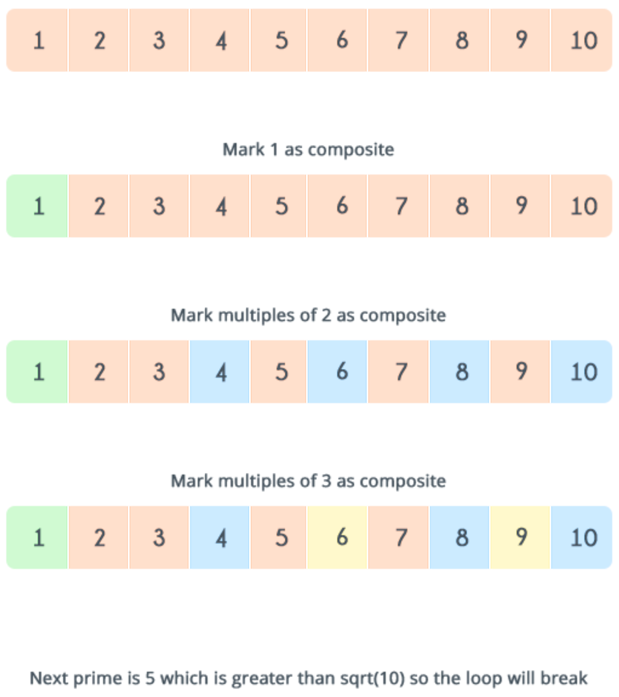
```
void sieve(int N) {
            bool isPrime[N+1];
            for(int i = 0; i <= N;++i) {
                isPrime[i] = true;
            }
            isPrime[0] = false;
            isPrime[1] = false;
            for(int i = 2; i * i <= N; ++i) {
                 if(isPrime[i] == true) {                    //Mark all the
multiples of i as composite numbers
                     for(int j = i * i; j <= N ;j += i)
                         isPrime[j] = false;
                }
            }
    }
```

This code will compute all the prime numbers that are smaller than or equal to N.

Let us compute prime numbers where $N = 10$.

1. Mark all the numbers as prime
2. Mark 1 as composite

In each iteration, check if a number is prime or not, if it is then mark all of its multiple as composite.


1 2 3 4 5 6 7 8 9 10
Mark 1 as composite
1 2 3 4 5 6 7 8 9 10
Mark multiples of 2 as composite
1 2 3 4 5 6 7 8 9 10
Mark multiples of 3 as composite
1 2 3 4 5 6 7 8 9 10
Next prime is 5 which is greater than sqrt(10) so the loop will break

The prime numbers are 2, 3, 5, and 7.

**Time complexity**
The inner loop that runs for each element is as follows:

1. If i = 2, inner loop runs N / 2 times
2. If i = 3, inner loop runs N / 3 times
3. If i = 5, inner loop runs N / 5 times

*Total complexity*

N * (½ + ⅓ + ⅕ + ... ) = O(NloglogN)

Reference for complexity analysis: Sieve of Erastothenes

**Modification of Sieve of Eratosthenes for fast factorization**

*Factorization in* $\sqrt{N}$

```
vector<int> factorize(int n) {
    vector<int> res;
    for (int i = 2; i * i <= n; ++i) {
        while (n % i == 0) {
            res.push_back(i);
            n /= i;
        }
    }
    if (n != 1) {
        res.push_back(n);
    }
    return res;
}
```

For example,

$n = 50, i = 2; res =$ {2}
$n = 25, i = 3; res =$ {2}
$n = 25, i = 4; res =$ {2}

$n = 25, i = 5; res =${2,5,5}
$n = 1$

When you exit the for loop, $res\ vector$ is the factorization of $N = 50$.

At every step, you must look for the prime number of the least value, which divides the current $N$. This is the main idea of this modification.

Let us construct an array which will give us this number in **O(1)** time.

```
int minPrime[n + 1];
for (int i = 2; i * i <= n; ++i) {
    if (minPrime[i] == 0) {         //If i is prime
        for (int j = i * i; j <= n; j += i) {
            if (minPrime[j] == 0) {
                minPrime[j] = i;
            }
        }
    }
}
for (int i = 2; i <= n; ++i) {
    if (minPrime[i] == 0) {
        minPrime[i] = i;
    }
}
```

Now, use this modification to factorize $N$ in *O(log(N))* time.

```
vector<int> factorize(int n) {
    vector<int> res;
    while (n != 1) {
        res.push_back(minPrime[n]);
        n /= minPrime[n];
    }
    return res;
}
```

For example,

$n = 50, minprime[50] = 2, res = 2$
$n = 25, minprime[25] = 5, res = 2, 5$
$n = 5, minprime[5] = 5, res = 2, 5, 5$
$n = 1$

The required factors are in $res$ .

**Conditions**

You can implement this modification only if you are allowed to create an array of integers of size $N$.

**Note**: This approach is useful when you need to factorize not-very-large numbers. It is not necessary to build a modified Sieve for every problem in which factorization is required. Moreover, you cannot build it for large numbers $N$ like $10^9$ or $10^{12}$. Therefore, for such large numbers it is recommended that you factorize in *O(sqrt(N))* instead.

**Fact**

If the factorization of $N$ is $p_1^{q_1} * p_2^{q_2} * \ldots * p_k^{q_k}$ where $p_1, p_2 ... p_k$ are the prime factors of $N$ and $q_1, q_2$... $q_k$ are the powers of the respective prime factors, then $N$ has $(q_1 + 1) * (q_2 + 1) * \ldots * (q_k + 1)$ distinct divisors.

**Sieve of Eratosthenes on the segment:**

Sometimes you need to find all the primes that are in the range $[L \ldots R]$ and not in $[1 ... N]$, where $R$ is a large number.

**Conditions**

You are allowed to create an array of integers with size $(R - L + 1)$.

**Implemention**

```
bool isPrime[r - l + 1]; //filled by true
for (long long i = 2; i * i <= r; ++i) {
    for (long long j = max(i * i, (l + (i - 1)) / i  * i); j <= r; j += i) {
        isPrime[j - l] = false;
    }
}
for (long long i = max(l, 2); i <= r; ++i) {
    if (isPrime[i - l]) {
        //then i is prime
    }
}
```

The approximate comlexity is *O(sqrt(R))*

**Suggestions**

It is recommended that you do not build a Sieve to check several numbers for primality. Use the following function instead, which works in $O(\sqrt{N})$ for every number:

```
bool isPrime(int n) {
    for (int i = 2; i * i <= n; ++i) {
        if (n % i == 0) {
            return false;
        }
    }
    return true;
}
```

*Contributed by: Shubham Gupta*