

Number Theory 1

Introduction

According to Wikipedia, number theory is a branch of pure mathematics devoted primarily to the study of the integers and integer-valued functions.

Find Prime Numbers from 1 to N

Example: If we take **N = 10**, then there are a total 4 prime numbers between 1 to 10, i.e., **2, 3, 5, 7**.

Basic Approach: One thing we can do is to iterate from 1 to **N**, and apply some logic **checkPrime(i)** to find out whether **i** is a prime number or not.

For **checkPrime(n)**, we can use the logic that **any prime number has only 2 factors**.

For example,

$2 \rightarrow 1 * 2, 2 * 1$

$3 \rightarrow 1 * 3, 3 * 1$

$5 \rightarrow 1 * 5, 5 * 1$

So for every **n**, we can calculate how many factors of **n** are there from 1 to **n**.

Therefore, in the worst case, **n** can be equal to **N**, and it will take **O(N)** time for **checkPrime()** logic. Additionally, there is an outer loop iterating from 1 to **N**, hence the overall time complexity of this approach will be **O(N²)**.

Optimized Approach:

Consider a pair $\mathbf{a * b = N}$. Here, \mathbf{a} and \mathbf{b} are the factors of \mathbf{N} .

For instance,

$$\mathbf{N = 12}$$

$\mathbf{(a, b)}$ can be (1, 12), (2, 6), (3, 4)

On observing the equation, we can infer that the maximum value of \mathbf{a} and \mathbf{b} can be the square root of \mathbf{N} .

$$\sqrt{\mathbf{N}} * \sqrt{\mathbf{N}} = \mathbf{N}$$

Since, if both of the values go beyond $\mathbf{sqrt(n)}$, then $\mathbf{a * b}$ would be greater than \mathbf{N} .

Also, if \mathbf{a} is less than $\sqrt{\mathbf{N}}$, then \mathbf{b} will be greater than $\sqrt{\mathbf{N}}$. Similarly, if \mathbf{a} is greater than $\sqrt{\mathbf{N}}$, \mathbf{b} will be smaller than $\sqrt{\mathbf{N}}$.

We can conclude that one of the numbers is $\leq \mathbf{sqrt(n)}$, and the other one is $\geq \mathbf{sqrt(n)}$.

To prove that \mathbf{N} is prime, we just need to find one of the numbers : \mathbf{a} or \mathbf{b} .

If no such number exists, it means that \mathbf{N} is not prime.

Hence, to do the primality test, we need not run a loop till \mathbf{N} , this can be done by running the loop till $\sqrt{\mathbf{N}}$ itself.

Example, for 25, if we check for factors from 1 to 5 then we can get all the factors without repeating any answer.

$$25 \rightarrow 1 * 25$$

$$25 \rightarrow 5 * 5$$

While checking the factors, remember to accordingly modify the count.

If the factor is \sqrt{N} then $\text{count} = \text{count} + 1$, otherwise $\text{count} = \text{count} + 2$.

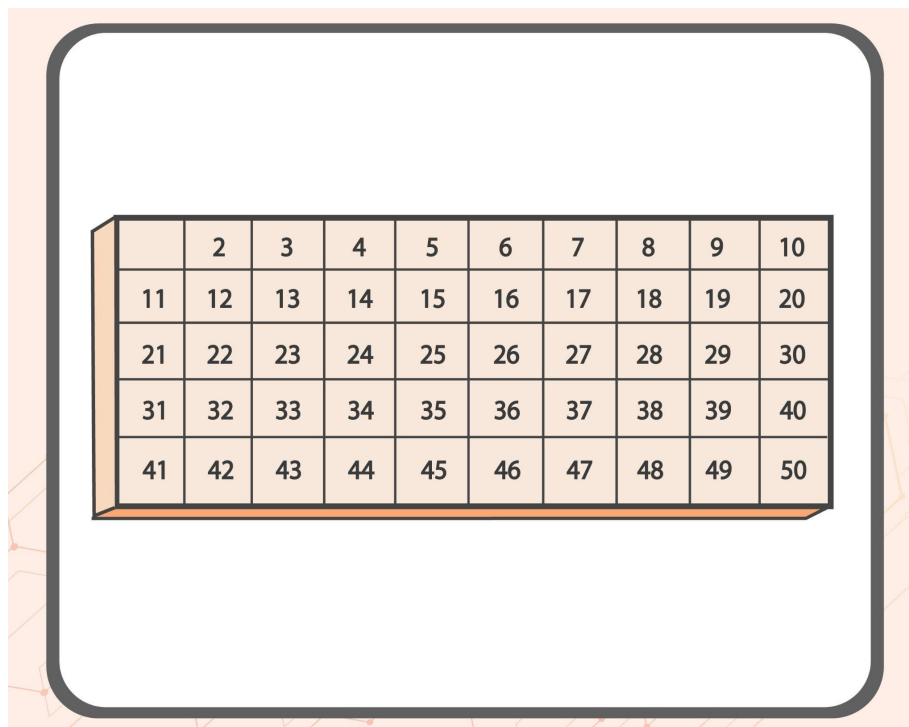
For example, while finding primes for integer 10, we encounter 2 which is a factor. We also encounter a factor 5, for which we won't check because it is greater than \sqrt{N} . Hence +2 is done in count, to consider both these factors.

Time Complexity of this approach is $O(N\sqrt{N})$.

Now, let's take a much more optimized approach in the next section.


Sieve of Eratosthenes

Suppose, we have to find the prime numbers from 1 to 50, then we make an array of size 50. Then we start from 2, and on each encounter of a prime number, we mark its multiples as composite.

A diagram showing a 5x10 grid of numbers from 2 to 50, representing an array used in the Sieve of Eratosthenes. The grid is enclosed in a rounded rectangle with a thick border. The numbers are arranged in rows: the first row contains 2-10, the second 11-20, the third 21-30, the fourth 31-40, and the fifth 41-50. The grid is shown from a slightly elevated perspective with a 3D effect on the left and bottom edges.

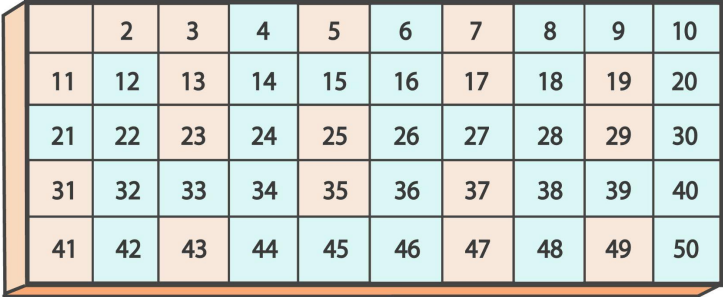
	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50

Now , when we encounter 2 and find out that it is a prime number, we mark all the multiples of 2 in the array.




	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50

When we encounter 3 and find out that it is a prime number, we mark all its multiples. Now, if we look carefully, 6 is a multiple of 3 but it is already marked. This is because it was marked during marking the multiples of 2. So whenever we start marking the multiples of a prime number, we start from its square.



	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50

Similarly for 5 and 7 and all the remaining prime numbers.



	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50

Now, we are left with all the prime numbers as unmarked.

Time Complexity: $O(N \log \log N)$

Space Complexity: $O(N)$

Code:

```
#include<iostream>
using namespace std;

int makeSieve(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){
            for(int j=i*i;j<=n;j+=i){
                isPrime[j] = false;
            }
        }
    }

    int count = 0;
    for(int i=0;i<=n;i++){
        if(isPrime[i] == true){
            count++;
        }
    }
    return count;
}

int main(){
    int n;
    cin >> n;
```

```
int nPrimes = makeSieve(n);  
cout<<nPrimes <<endl;  
return 0;  
}
```

Greatest Common Divisor

Let's try to understand the algorithm to finding GCD with the help of an example:

Find GCD (10, 6).

Basic Approach: Now, one common divisor of both 10 and 6 is 1. So, we start from one and we go till 6, and for every number we iterate on, we are going to check whether it divides both 10 and 6, while maintaining a maximum variable.

Now, let's say that we have **GCD (16131, 13733)**. In this case we have to iterate till 13733 which is a very huge number, so let's try to come up with a better solution.

Euclid's Algorithm:

Euclid's algorithm states that **$\text{GCD}(A, B) = \text{GCD}(B, A \% B)$** , given that **$A > B$** .

In the above algorithm, $\text{GCD}(B, A \% B)$ is a recursive call, so our **base case is when the second parameter i.e, $A \% B$ becomes zero, we return B.**

For example,

$$\text{GCD}(16, 10) = \text{GCD}(10, 6)$$

$$\text{GCD}(10, 6) = \text{GCD}(6, 4)$$

$$\text{GCD}(6, 4) = \text{GCD}(4, 2)$$

$$\text{GCD}(4, 2) = \text{GCD}(2, 0) \rightarrow \text{Ans} = 2$$

Time Complexity of Euclid's algorithm :

$\text{GCD}(A, B) \rightarrow \text{GCD}(B, A \% B)$

Let's say that $R = A \% B$

$\text{GCD}(B, R) \rightarrow \text{GCD}(R, B \% R)$

Now, we have $0 \leq R \leq B-1$

Also, $R = A \% B = A - B * \text{int}(A/B)$

Example, $10 \% 7 = 10 - 7 * \text{int}(10/7) = 10 - 7 * 1 = 3$

Also, the minimum value of $A - B = 1$, so $R \leq A - B$ and we have $R < B$.

Adding the two equations, we get $R < A/2$.

So at every step the value of the second parameter/ number gets halved. Therefore the time complexity of this algorithm is $O(\log(\max(a, b)))$.

Diophantine Equations

A Diophantine equation is a polynomial equation, usually in two or more unknowns, for which the integral solutions exist.

For example, in the equation $a * x + b * y = c$, if $a = 3$, $b = 6$, $c = 9$, then there exist $x = 1$ and $y = 1$.

For linear Diophantine equation equations, integral solutions exist if and only if the GCD of coefficients of the two variables divides the constant term perfectly. In other words the integral solution exists if $\text{GCD}(a, b)$ divides c .

Extended Euclid's Algorithm

While the Euclidean algorithm calculates only the greatest common divisor (GCD) of two integers a and b , the extended version also finds a way to represent GCD in terms of a and b , i.e. coefficients x and y for which:

$$a \cdot x + b \cdot y = \gcd(a, b)$$

It's important to note, that we can always find such a representation, for instance $\gcd(55, 80) = 5$, therefore we can represent 5 as a linear combination with the terms 55 and 80:

$$55 \cdot 3 + 80 \cdot (-2) = 5$$

We have discussed these equations in the previous topic, Diophantine Equations.

Algorithm:

We will denote the GCD of a and b with g in this section.

The changes to the original algorithm are very simple. If we recall the algorithm, we can see that the recursive algorithm ends with $b = 0$ and $a = g$ (**base case, as when $b = 0$, GDC is $g = a$**). For these parameters we can easily find coefficients, namely

$$g \cdot 1 + 0 \cdot 0 = g.$$

Starting from these coefficients $(x, y) = (1, 0)$, we can go backwards up the recursive calls. All we need to do is to figure out how the coefficients x and y change during the transition from (a, b) to $(b, a \% b)$.

Let us assume we found the coefficients (x_1, y_1) for $(b, a \% b)$:

$$b \cdot x_1 + (a \% b) \cdot y_1 = g$$

and we want to find the pair (x, y) for (a, b) :

$$a \cdot x + b \cdot y = g$$

We can represent $(a \% b)$ as:

$$(a \% b) = a - [a/b] \cdot b$$

Substituting this expression in the coefficient equation of (x_1, y_1) gives:

$$\begin{aligned} g &= b \cdot x_1 + (a \% b) \cdot y_1 \\ &= b \cdot x_1 + (a - [a/b] \cdot b) \cdot y_1 \end{aligned}$$

and after rearranging the terms we get:

$$g = a \cdot y_1 + b \cdot (x_1 - y_1 \cdot [a/b])$$

Thus we have found the values of x and y .

$$x = y_1$$

$$y = x_1 - y_1 \cdot [a/b]$$

Code:

```
int gcdExtended(int a, int b, int *x, int *y)
{
    // Base Case
    if (b == 0)
    {
        *x = 1;
        *y = 0;
        return b;
    }

    int x1, y1; // To store results of recursive call
    int gcd = gcdExtended(b, a % b, &x1, &y1);

    // Update x and y using results of
```

```
// recursive call
*x = y1;
*y = x1 - y1 * (a/b);

return gcd;
}
```

The recursive function above returns the GCD and the values of coefficients to x and y (which are passed by reference to the function).

This implementation of the extended Euclidean algorithm produces correct results for negative integers as well.

Iterative Code:

It's also possible to write the Extended Euclidean algorithm in an iterative way. Because it avoids recursion, the code will run a little bit faster than the recursive one.

```
int gcdExtended(int a, int b, int *x, int *y)
{
    *x = 1;
    *y = 0;
    int x1 = 0, y1 = 1, a1 = a, b1 = b;
    while (b1) {
        int q = a1 / b1;
        tie(x, x1) = make_tuple(x1, x - q * x1);
        tie(y, y1) = make_tuple(y1, y - q * y1);
        tie(a1, b1) = make_tuple(b1, a1 - q * b1);
    }
    return a1;
}
```

No algorithm is complete without some practice. Here are links to a few practice problems that you should try.

- [10104 - Euclid Problem](#)
- [GYM - \(J\) Once Upon A Time](#)
- [UVA - 12775 - Gift Dilemma](#)

Multiplicative Modulo Inverse

Let's try to understand this problem with the help of an example,

Given: $(A * B) \% m = 1$, $A = 5$, $m = 17$

To find: B (multiplicative modulo inverse)

Steps:

$$(5 * 1) \% 17 = 5$$

$$(5 * 2) \% 17 = 10$$

$$(5 * 3) \% 17 = 15$$

$$(5 * 4) \% 17 = 3$$

$$(5 * 5) \% 17 = 8$$

$$(5 * 6) \% 17 = 13$$

$$(5 * 7) \% 17 = 1$$

Solution: This problem can be solved using the extended euclid algorithm.

$$(A * B) \% m = 1$$

Since $A * B \equiv 1 (\% m)$

$$(A * B - 1) \% m = 0$$

$(A * B - 1) = m * q$ where q is an integer.

$$(A * B) - m * q = 1$$

$$(A * B) + m * (-q) = 1$$

$$(A * B) + m * (Q) = 1$$

Now, according to **linear diophantine equations**,

B will exist only **if GCD (A, M) divides 1**.

$$\text{i.e., GCD (A, M) = 1}$$

=> A, M should be co-prime.

Code:

```
Triplet extendedEuclid(int a, int b){  
  
    // base case  
    if(b == 0){  
        Triplet ans;  
        ans.gcd = a;  
        ans.x = 1;  
        ans.y = 0;  
        return ans;  
    }  
  
    // extended euclid algorithm  
    Triplet smallAns = extendedEuclid(b, a%b);  
    Triplet ans;  
    ans.gcd = smallAns.gcd;  
    ans.x = smallAns.y;  
    ans.y = smallAns.x - (a/b)*smallAns.y;  
    return ans;  
}  
  
int mmInverse(int a, int m){  
    Triplet ans = extendedEuclid(a, m);
```

```
    return ans.x;  
}
```