Number Theory - 1

73

Number Theory | Primes | CodeMonk

Introduction:

Problems in competitive programming which involve Mathematics are are usually about number theory, or geometry. If you know number theory, that increases your ammo heavily in solving a lot of tougher problems, and helps you in getting a strong hold on a lot of other problems, too.

Problems in competitive programming require insight, so just knowing some topics is not enough at all. All of the problems requires more or less math tough. For instance, solving large systems of equations and approximating solutions to differential equations.

Modulo:
Modulo operation gives the remainder after division, when one number is divided by another. It is denoted by % sign.

Example:
We have two numbers 5 and 2, then 5%2 is 1 as when 5 is divided by 2, it leaves 1 as remainder.

Properties:
Some of the properties of modulo are:
(a+b)%c = (a%c + b%c)%c.
(a * b)%c = ((a%c) * (b%c))%c.

Example:
Let's say a = 5, b = 3, c = 2.
Then:
1) (5+3)%2 = 8%2 = 0.
Similarly (5%2 + 3%2)%2 = (1 + 1)%2 = 0.
2) (5 * 3)%2 = 15%2 = 1.
Similarly ((5%2) * (3%2))%2 = (1 * 1)%2 = 1.

Greatest Common Divisor
Greatest Common Divisor (GCD) of two or more numbers is the largest positive number that divides all the numbers which are being taken into consideration.
For example: GCD of 6, 10 is 2 since 2 is the largest positive number that divides both 6 and 10.

Naive Approach:
We can traverse over all the numbers from min(A, B) to 1 and check if the current number divides both A and B or not. If it does, then it will be the GCD of A and B.

```
int GCD(int A, int B) {
    int m = min(A, B), gcd;
    for(int i = m; i > 0; --i)
        if(A % i == 0 && B % i == 0) {
            gcd = i;
```

```
            return gcd;
        }
}
```

Time Complexity: Time complexity of this function is O(min(A, B)).

Euclid's Algorithm:

Idea behind Euclid's Algorithm is GCD(A, B) = GCD(B, A % B). The algorithm will recurse until A % B = 0.

```
int GCD(int A, int B) {
    if(B==0)
        return A;
    else
        return GCD(B, A % B);
}
```

Let us take an example.
Let A = 16, B = 10.
GCD(16, 10) = GCD(10, 16 % 10) = GCD(10, 6)
GCD(10, 6) = GCD(6, 10 % 6) = GCD(6, 4)
GCD(6, 4) = GCD(4, 6 % 4) = GCD(4, 2)
GCD(4, 2) = GCD(2, 4 % 2) = GCD(2, 0)

Since B = 0 so GCD(2, 0) will return 2.

Time Complexity: The time complexity of Euclid's Algorithm is **O(log(max(A, B)))**.

Extended Euclid Algorithm:

This is the extended form of Euclid's Algorithm explained above. GCD(A,B) has a special property that it can always be represented in the form of an equation, i.e., Ax + By = GCD(A, B).

This algorithm gives us the coefficients (x and y) of this equation which will be later useful in finding the Modular Multiplicative Inverse. These coefficients can be zero or negative in their value. This algorithm takes two inputs as A and B and returns GCD(A, B) and coefficients of the above equations as output.

Implementation:

```
#include < iostream >

int d, x, y;
void extendedEuclid(int A, int B) {
    if(B == 0) {
        d = A;
        x = 1;
        y = 0;
    }
    else {
        extendedEuclid(B, A%B);
        int temp = x;
        x = y;
        y = temp - (A/B)*y;
    }
}
```

```
int main( ) {
extendedEuclid(16, 10);
cout << ”The GCD of 16 and 10 is ” << d << endl;
cout << ”Coefficient x and y are: ”<< x <<  “and  “ << y << endl;
return 0;
}
```

Output:

```
The GCD of 16 and 10 is 2
Coefficient x and y are: 2 and -3
```

Initially, Extended Euclid Algorithm will run as Euclid Algorithm until we get GCD(A, B) or until B gets 0 and then it will assign x = 1 and y = 0. As B = 0 and GCD(A, B) is A in the current condition, so equation Ax + By = GCD(A, B) will be changed to A * 1 + 0 * 0 = A.

So the values of d, x, y in the whole process of extendedEuclid( ) function are:

1) d=2, x = 1, y = 0.
2) d=2, x = 0 , y = 1 - (4/2) * 0 = 1.
3) d=2, x = 1 , y = 0 - (6/4) * 1 = -1.
4) d=2, x = -1 , y = 1 - (10/6) * -1 = 2.
5) d=2 , x= 2, y = -1 - (16/10) * 2 = -3.

Time Complexity: The time complexity of Extended Euclid’s Algorithm is **O(log(max(A, B)))**.

Prime Numbers

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

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

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

There are different methods to check if the number is prime or not.

Naive Approach:

We will traverse through all the numbers from 1 to N and count the number of divisors. If the number of divisors are equal to 2 then the given number is prime otherwise 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: Time complexity of this function is **O(N)** as we are traversing from 1 to N.

A slightly better approach:

Consider two positive numbers N and D such that N is divisible by D and D is less than the square root of N. Then, (N / D) must be greater than the square root of N. N is also divisible by (N / D). So, if there is a divisor of N that is less than square root of N then there will be a divisor of N that is greater than square root of N. Therefore, we just need to traverse till the square root of N.

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

Time complexity: Time complexity of this function is O(sqrt(N)) as we are traversing from 1 to sqrt(N).

Sieve of Eratosthenes:

The Sieve of Eratosthenes can be used to find all the prime numbers less than or equal to a given number N. It can also be used to find out if a number is prime or not, by just looking up at the sieve.

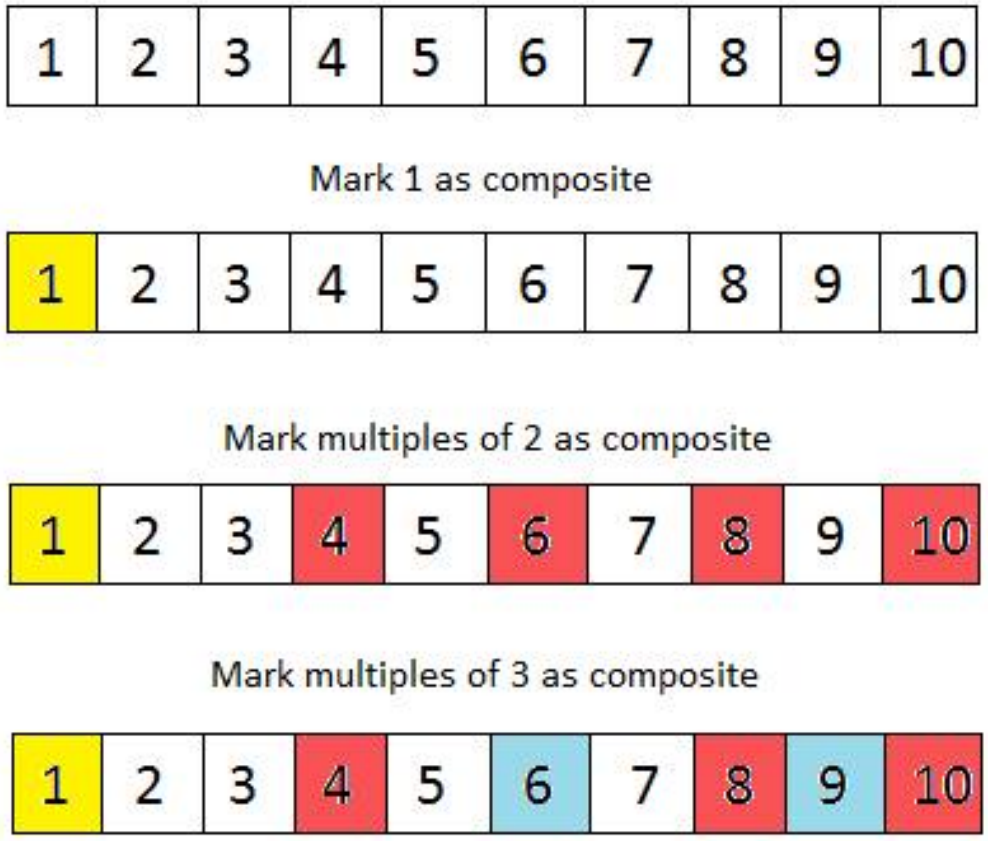
The basic idea behind the sieve of eratosthenes is that at each iteration, we pick one prime number and eliminate all the multiples of that prime number. After the elimination process ends, all the unmarked numbers which are left are prime!

Pseudo Code:
Mark all the numbers as prime numbers.
Traverse over all the prime numbers smaller than sqrt(N).
Mark the multiple of the prime numbers as 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;
        }
```

```
    }
}
```

The above code will compute all the prime numbers that are smaller than or equal to N.
Let us compute prime numbers where N = 10. First mark all the numbers as prime. Now mark 1 as composite. Then in each iteration we will select a prime and then mark its multiple as composite.



So the prime numbers are 2, 3, 5 and 7.

Time Complexity:

The inner loop runs for each element.
if i = 2, inner loop runs N / 2 times
if i = 3, inner loop runs N / 3 times
if i = 5, inner loop runs N / 5 times
.
.
.
Total complexity: N * (½ + ⅓ + □ + ...) = O(NloglogN)

Modular Exponentiation

Consider a problem in which you need to calculate a^b%c, where **%** is a modulo operator and b can be very large (i.e. in order of 10^{18}).

Naive Approach:

a^b%c can be rewritten as a * a * a * a *... upto b times. So in this approach we will multiply **a**, **b** times.

```
long long exponentiation(long long a, long long b, long long c) {
       long long ans = 1;
       for(int i = 1;i <= b;i++) {
           ans *= a;                              //multiplying a, b times.
           ans %= c;
       }
   return ans;
}
```

In each iteration the variable **ans** will get multiplied by a. Also, special care needs to be taken that this value never exceeds 'c' in any iteration. Hence we take modulo of 'ans' with 'c' in each iteration. i.e., ans = ans%c.
The basic property that we exploit in this is: (x*y) mod n = ((x mod n) * (y mod n)) mod n.

So in the above code we have to calculate (ansa)%c which is equal to ((ans%c)(a%c))%c .

Complexity: O(b)

Modular exponentiation:
We can now clearly see that this approach is very inefficient, and we need to come up with something better. We can take care of this problem in $O(\log_2 b)$ by using a technique called exponentiation by squaring. This uses only $O(\log_2 b)$ squarings and $O(\log_2 b)$ multiplications. This is a major improvement over the most naive method.

The process we would follow is:

```
ans=1                              //Final answer which will be displayed
while(b !=0 ) {
   /*Finding the right most digit of 'b' in binary form, if it is 1 , then multiply t
    in  ans. */
         if(b%2 == 1) {            //as if b%2 == 1,means last /rightmost digit of b i
           ans = ans*a ;
            ans = ans%c;       //at each iteration if value of ans exceeds then reduce
         }
 a = a*a;                  // This is explained below
 a %= c;                   //at each iteration if value of a exceeds then reduce it t
```

```
    b /= 2;                    //Trim the right-most digit of b in binary form.
}
```

To understand the above code : Let's say b = 45, then express 'b' in binary form. Note the positions of '1' in 'b' .
As in binary form b changes to $\{101101\}_2$

$a^b = a^{\{101101\}_2}$

As if we split the binary representation of b ,

$b = 1 * 2^5 + 0 * 2^4 + 1 * 2^3 + 1 * 2^2 + 0 * 2^1 + 1 * 2^0$.

Then $a^b = (1 * (a^{2^5}) * (0 * (a^{2^4})) * (1 * (a^{2^3})) * (1 * (a^{2^2})) * (0 * (a^{2^1})) * (1 * (a^{2^0}))$

Now this can be computed easily in 5 iterations.

In every iteration the value of 'a' is made equal to 'a * a'.
As,

$a * (2^0) = a$

$a * (2^1) = a^2 = a * a$

$a * (2^2) = a^4 = ((a * a) * (a * a))$

and so on..

So, at 1st iteration a changes to a * a, then a * a changes to (a * a) * (a * a) and so on. And also in each iteration, we multiply current value of a in ans, whenever we encounter a digit of b in binary form, which is equal to 1.

Processing:

1. Calculate $(5^{59})\%19$, where '%' stands for modulo operator.

express 59 in binary = $\{111011\}_2$

$5^{59} = 5^{\{111011\}_2} = 5^{2^5} * 5^{2^4} * 5^{2^3} * 5^{2^3} * 5^{2^1} * 5^{2^0}$

Let's compute its value in 6 iterations:

Let ans=1, a=5, b=59, c=19;

Iteration 1:
Since the rightmost digit of 'b'(111011) is 1:
ans=(ans * a)%c = (1 * 5)%19 = 5
a=(a * a)%c = (5 * 5)%19 = 6
b /= 2 (b = 11101)

Iteration 2:
Since the rightmost digit of 'b'(11101) is 1:
ans=(ans * a)%c = (5 * 6)%19 = 11
a=(a * a)%c = (6 * 6)%19 = 17
b /= 2 (b = 1110)

Iteration 3:
Since the rightmost digit of 'b'(1110) is 0:
We won't multiply anything to ans
a=(a * a)%c = (17 * 17)%19 = 4
b /= 2 (b = 111)

Iteration 4:
Since the rightmost digit of 'b'(111) is 1:

ans=(ans * a)%c = (11 * 4)%19 = 6
a=(a * a)%c = (4 * 4)%19 = 16
b /= 2 (b = 11)

Iteration 5:
Since rightmost digit of 'b'(11) is 1:
ans=(ans * a)%c = (6 * 16)%19 = 1
a=(a * a)%c = (16 * 16)%19 = 9
b /= 2 (b = 1)

Iteration 6:
Since rightmost digit of 'b'(1) is 1:
ans=(ans * a)%c = (1 * 9)%19 = 9
a=(a * a)%c = (9 * 9)%19 = 5
b /= 2 (b = 0) //break

Hence, the final answer is ans = 9,

therefore $(5^{59})\%19 = 9$

Time limit:
$O(\log_2(b))$

Memory limit:
O(1)

Complexity: O($\log_2(b)$) (number of digits present in binary notation of number 'b')

Implementation:

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

int modpowIter(int a, int b, int c) {
        int ans=1;
        while(b != 0) {
                if(b%2 == 1)
                        ans=(ans*a)%c;

                a=(a*a)%c;
                b /= 2;
        }
        return ans;
}
int main() {
        int a=5,b=59,c=19,ans,ans1;
        ans = modpowIter(a,b,c);
        cout << ans << endl;
}
```

Output:

```
9
```

You can use another recursive approach, to find $(a^b)\%c$.

We know that a^b can be written as:

$a^b = a^{2^{(b/2)}}$ - If b is even, and b>0.

$a^b = (a) * a^{2^{(b-1/2)}}$ - If b is odd.

$a^b = 1$ - If b is 0.

```
int modRecursion(int a, int b, int c)
{
    if(b == 0)
    return 1;
    if(b == 1)
    return a%c;
    else if( b%2 == 0)                                          //if b is even
    {
        return modRecursion((a *a)%c,b/2,c);
    }
    else                                                              // if b is o
    {
        return (a*modRecursion((a*a%c),b/2,c))%c;
    }
}
```

Let's take a = 2 ,b = 5 , c = 5, so modRecusrion(2, 5, 5).

1) As b is odd, it will be 2 * modRecusriion(2 * 2, 2, 5) = modRecursion(4, 2, 5).
2) Now in modRecursion(4 , 2, 5) as b is even, it will move to modRecusrion(16%5, 1, 5) = modRecursion(1, 1, 5).
3) Now as in modRecursion(1, 1, 5), B =1 ,therefore it will return 1.
4) Therefore modRecusrion(4, 2, 5) will return 1 and as in first statement modRecusrion(2, 5, 5) will return 2 * modRecusrion(4, 2, 5) = 2 * 1 = 2.
So final answer will be 2 here.

Complexity: O($\log_2(b)$)

//Change the data types according to the problem statement

In many problems 'c' might be given as 10^9+7. Hence, data types of all variables must be kept "long long int".

Solve Problems

AUTHOR

Prateek Garg

Student at DIT University

Dehradun

7 notes

TRENDING NOTES

Number Theory - II
written by Tanmay Chaudhari

Matrix exponentiation
written by Mike Koltsov

Graph Theory - Part II
written by Pawel Kacprzak

Computational Geometry - I
written by Arjit Srivastava

Rendering Performance in Android - Overdraw
written by Vishnu Sosale

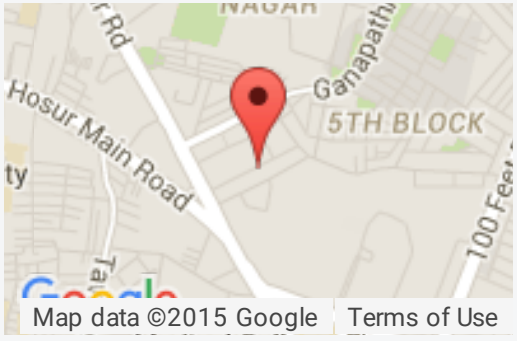
more ...

Updates & Releases
Team
Careers
In the Press
CodeTable
HackerEarth Academy
Developer Profile
Resume
Campus Ambassadors
Get Me Hired
Privacy
Terms of Service
Judge Environment
Solution Guide
Problem Setter Guide
Practice Problems
HackerEarth Challenges
College Challenges

RECRUIT

Developer Sourcing
Lateral Hiring
Campus Hiring
FAQs
Customers
Annual Report

REACH US



IIIrd Floor,
Salarpuria Business Center,
4th B Cross Road,
5th A Block,
Koramangala Industrial Layout,
Bangalore,
Karnataka 560095,
India.

contact@hackerear
+91-80-4155-4695
+1-650-461-4192