Number Theory 2

Binary Exponentiation

Many times , our answer is out of range of datatype int. To avoid this we use modulo operation to overcome this problem. Some of the properties of modulo operation are:

$$(a + b)\%m = (a\%m) + (b\%m)$$
$$(a * b)\%m = (a\%m) * (b\%m)$$
$$(a - b)\%m = (a\%m) - (b\%m)$$
$$(a/b)\%m = (a\%m) * (b^{-1}\%m)$$

Iterative

We can write any decimal number into a binary number. Let us take an example

$$7^{45}$$

We can write,

$$45 = 1\text{x}2^5 + 0\text{x}2^4 + 1\text{x}2^3 + 1\text{x}2^2 + 0\text{x}2^1 + 1\text{x}2^0$$

$\Rightarrow$ 7^{45} can be calculated easily.

Code

```
int pr(int a, int n)
{
   int ans=1;
   while(n)
   {
       if(n&1)
         ans = (ans*a)%MOD;
       a = (a*a)%MOD;

       n >>= 1;
   }
   return ans;
}
```

Recursive

To calculate a^n, we recursively call on $a^{n/2}$ and multiply them. Its base case is returning 1 when n = 0.

Code

```
int power(int a, int n){
   if(n == 0)
       return 1;

   if(n == 1){
       return a;
   }

   if(n&1)
       return ((a*power(a,n/2)%MOD)*power(a,n/2))%MOD;

   return (power(a,n/2)*power(a,n/2))%MOD;
}
```

Euler Totient Function

For a positive integer n , totient function is represented as Φ(n). It is defined as the number of integers m such that

$$1 \leq m < n$$
$$gcd(m, n) = 1$$

In simple words number of numbers from 1 to n-1 which are coprime with n. Its formula is given by

$$\Phi(n) = n*(1 - 1/p_1)*(1 - 1/p2)*(1 - 1/p_3)...*(1 - 1/p_k)$$

where $p_1, p_2, p_3, .., p_k$ are distinct prime factors of n.

Derivation

If A and B are coprime or gcd(A,B) = 1, then

$$\Phi(A*B) = \Phi(A)*\Phi(B)$$

We can write

$$n = p_1^a * p_2^b * p_3^c ... p_k^k$$
$$\Phi(n) = \Phi(p_1^a * p_2^b * p_3^c ... p_k^k)$$

Since $gcd(p_1^a, p_2^b) = 1$

$$\Phi(n) = \Phi(p_1^a) * \Phi(p_2^b) * \Phi(p_3^c) * ... * \Phi(p_k^k)$$

Let us analyze $\Phi(p^a)$

Numbers from 1 to p^a which are not coprime with p^a are p, 2p, 3p... p^a.

This is an AP with common difference p and first term p. Using the formula of nth term of AP, we get

$$p^a = p + (x-1)*p$$
$$x = p^{a-1}$$

Therefore, the number of numbers that are coprime with p^a are

$$p^a - p^{a-1}$$

$$p^a(1 - 1/p)$$

Substituting this in above equation of $\Phi(n)$, we get

$$\Phi(n) = p_1^a(1-1/p_1) * p_2^b(1-1/p_2) * p_3^c(1-1/p_3) * \ldots *p_k^k(1-1/p_k)$$
$$\Phi(n) = p_1^a * p_2^b * p_3^c * p_k^k * (1-1/p_1) *(1-1/p_2) *(1-1/p_3) * \ldots *(1-1/p_k)$$
$$\Phi(n) = n * (1-1/p_1) *(1-1/p_2) *(1-1/p_3) * \ldots *(1-1/p_k)$$
$$\text{since } n = p_1^a * p_2^b * p_3^c *\ldots* p_k^k$$

Implementing totient function

1. Declare an array a[] of size n+1.
2. Initialize the array with a[i] = i.
3. Iterate from 2 to n and check if(a[i] == i), if yes, that means it is a prime number because it is not touched by previous numbers during their iteration. Change it to a[i]-1 and multiply all its multiples with (1 − 1/ a[i]).
4. You have your array with totient values ready.

Code

```
// tot array is ready with tot values upto 10^6..
const int size = 1e6+2;
int tot[size];
void totient()
{
   for(int i=0; i<size; i++)
     tot[i] = i;

   for(int i=2; i<size; i++)
   {
       if(tot[i] == i)
       {
           tot[i] -= 1;
           for(int j=2*i; j<size; j+=i)
             tot[j] -= tot[j]/i;
       }
   }
}
```

Segmented Sieve

When the value of n is very large but the r-l range is less than 10^8 , then we use segmented sieve in place of Sieve of Eratosthenes.

Code

```
// segmented sieve implementation
vector<bool> segmentedSieve(int l, int r)
{
    vector<int> primes;
    int rootR = sqrtl(r);
    vector<bool> isprime(rootR+1, 1);
    isprime[0] = isprime[1] = 0;

    for(int i=2; i<rootR+1; i++)
    {
        if(isprime[i])
        {
            primes.pb(i);
            for(int j=i*i; j<=rootR; j+=i)
                isprime[j] = 0;
        }
    }

    vector<bool> requiredSieve(r-l+1, 1);
    for(int currPrime : primes)
    {
        for(int j=max(currPrime*currPrime, (l+currPrime-1)/currPrime *
currPrime); j<=r; j+=currPrime)
        {
            requiredSieve[j-l] = 0;
        }
    }
    if(l==1)
        requiredSieve[0] = 0;

    return requiredSieve;
}
```