**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

745

We can write,

45 = 1x25 + 0x24 + 1x23 + 1x22 + 0x21 + 1x20

⇒ 745 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 an , we recursively call on an/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 ≤ 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

𝜱(n) = n\*(1 - 1/p1)\*(1 - 1/p2)\*(1 - 1/p3)...\*(1 - 1/pk)

where p1, p2, p3, .. , pk are distinct prime factors of n.

Derivation

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

𝜱(A\*B) = 𝜱(A)\*𝜱(B)

We can write

n = p1a \* p2b \* p3c...pkk

𝜱(n) = 𝜱(p1a \* p2b \* p3c...pkk)

Since gcd(p1a,p2b) = 1

𝜱(n) = 𝜱(p1a) \* 𝜱(p2b) \* 𝜱(p3c) \* ... \* 𝜱(pkk)

Let us analyze 𝜱(pa)

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

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

*pa = p + (x-1)\*p*

*x = pa-1*

Therefore, the number of numbers that are coprime with pa are

pa - pa-1

pa (1 - 1/p)

Substituting this in above equation of 𝜱(n), we get

𝜱(n) = p1a(1-1/p1) \* p2b(1-1/p2) \* p3c(1-1/p3) \* ... \*pkk(1-1/pk)

𝜱(n) = p1a \* p2b \* p3c \* pkk \* (1-1/p1) \*(1-1/p2) \*(1-1/p3) \* ... \*(1-1/pk)

𝜱(n) = n \* (1-1/p1) \*(1-1/p2) \*(1-1/p3) \* ... \*(1-1/pk)

since n = p1a \* p2b \* p3c \*...\* pkk

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 108 , 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;

}