3 Number Theory

In this chapter we are going to talk about important mathematical concepts, theorems and tricks. Let's get started.

Greatest Common Divisor(GCD) using Euclid's Algorithm

The Greatest Common Divisor (GCD) of two integers (a, b) denoted by gcd(a,b), is defined as the largest positive integer d such that d | a and d | b where x | y implies that x divides y.

Example of GCD: gcd(4, 8) = 4, gcd(10, 5) = 5, gcd(20,12) = 4.

Gcd(A,B) = Gcd(B,A%B) // recurrence for gcd

Gcd(A,0) = A // base case

Proof : If a = bq + r.

Let d be any common divisor of a and b which implies d|a and d|b $\Rightarrow$ d|(a – bq) $\Rightarrow$ d|r.

Let e be any common divisor of b and r $\Rightarrow$ e|b , e|r $\Rightarrow$ e|(bq + r) $\Rightarrow$ e|a. hence any common divisor of a and b must also be a common divisor of r and any common divisor of b and r must also be a divisor of a $\Rightarrow$ d is a common divisor of a and b iff d is a common divisor of b and r.

Similarly, the LCM of two integers (a, b) denoted by lcm(a,b), is defined as the smallest positive integer l such that a |l and b | l. Example of LCM : lcm(4,8) = 8, lcm(10,5) = 10, lcm(20,12) = 60.

gcd(a,b) * lcm(a,b) = a*b

```
int gcd(int a, int b) {
 return (b == 0 ? a : gcd(b, a % b));
  }
int lcm(int a, int b) {
 return (a * (b / gcd(a, b)));
} // divide before multiply!
```

The GCD of more than 2 numbers, e.g. gcd(a,b,c) is equal to gcd(a,gcd(b,c)), etc, and similarly for LCM.

Both GCD and LCM algorithms run in $O(log(N))$, where n = max(a,b).

Extended Euclid's Algorithm

Extended Euclid's is used to find out the solution of equations of the form Ax + By = C , where C is a multiple of divisor of A and B. Extended Euclid's works in the same manner as the euclid's algorithm. Ax + By = 1 (we will find solutions of this equation let them be x' and y' given that gcd(a,b) is 1 then the solutions of equation Ax + By = k where k is a multiple of gcd(A,B) are given by k*x' and k*y'.

Ax + By = 1 . . . .(1)

Bx' + (A%B)y' = 1 . . . .(2) // using euclid's algo gcd(a,b) = gcd(b,a%b)

Compare coefficients of (1) and (2)

x = y'

y = x' – [A/B]y'

Hence we can recursively calculate x and y in the following manner:

```
void eeuclid(ll a, ll b){
      if(b==0){
            ex=1;ey=0;ed=a;
      }
      else{
            eeuclid(b,a%b);
            ll temp=ex;
            ex=ey;
            ey=temp-(a/b)*ey;
      }
}
```

Application of Extended Euclidean Algorithm :

1. To calculate multiplicative modulo inverse of a w.r.t. m.

Let's see :

$x \equiv a^{-1} (\text{mod } m)$

$a \cdot a^{-1} \equiv a \cdot x (\text{mod } m)$

$x \cdot a \equiv 1 (\text{mod } m)$

$\Rightarrow$ ax – 1 = qm

$\Rightarrow$ ax – qm = 1

This equation has solutions only if a and m are co-prime that is gcd(a,m) = 1.

We can calculate x and q using extended euclid's algorithm where x is the inverse of a modulo m.

Sieve of Eratosthenes

It is easy to find if some number (say N) is prime or not — you simply need to check if at least one number from numbers lower or equal sqrt(n) is divisor of N. This can be achieved by simple code:

```
boolean isPrime( int n )
{
 if ( n == 1 )
   return false; // by definition, 1 is not prime number
 if ( n == 2 )
   return true; // the only one even prime
 for ( int i = 2; i * i <= n; ++i )
   if ( n%i == 0 )
     return false;
  return true;
 }
```

So it takes sqrt(n) steps to check this. Of course you do not need to check all even numbers, so it can be "optimized" a bit:

```
boolean isPrime( int n )
{
   if ( n == 1 )
      return false; // by definition, 1 is not prime number
   if ( n == 2 )
      return true; // the only one even prime
   if ( n%2 == 0 )
      return false; // check if is even
   for ( int i = 3; i * i <= n; i += 2 ) // for each odd number
     if ( n%i == 0 )
       return false;
   return true;
}
```

So let say that it takes 0.5sqrt(n) steps*. That means it takes 50,000 steps to check that 10,000,000,000 is a prime.

Time Complexiety :

If we have to check numbers upto N, we have to check each number individually. So time complexity will be $O(Nsqrt(N))$.

Can we do better?

Ofcourse! we can use a sieve of numbers upto N. For all prime numbers <= sqrt(N), we can make their multiple non-prime i.e. if p is prime, 2p, 3p, ..., floor(n/p)*p will be non-prime.

Sieve code :

```
void primes(int *p)
{
  for(int i = 2;i<=1000000;i++)
    p[i] = 1;
  for(int i = 2;i<=1000000;i++)
  {
     if(p[i])
       {
           for(int j = 2*i;j<=1000000;j+=i)
          {
            p[j] = 0;
          }
       }
  }
   p[1] = 0;
   p[0] = 0;
  return;
}
```

Can we still do better?

Yeah sure! Here we don't need to check for even numbers. Instead of starting the non-prime loop from 2p we can start from p^2.

Optimised code :

```
void primes(bool *p)
{
   for(int i = 3;i<=1000000;i += 2)
   {
     if(p[i])
     {
         for(int j = i*i;j <= 1000000; j += i)
        {
          p[j] = 0;
```

```
        }
    }
}
p[1] = 0;
p[0] = 0;
return;
}
```

T = O(NloglogN)

Hence, we have significantly reduced our complexity from N*sqrt(N) to approx linear time.

Optimizations!

we know that all the numbers which are even are non prime except 2.Hence we can mark only odd numbers as non prime in our sieve and jump to odd numbers always.

Code :

```
#define lim 10000000
vector <bool> mark(lim+2,1);
vector <ll> primes;
void sieve() // we need primes upto 10^8
{
      //ll times = 0;
      for(ll i=3;i<=lim;i+=2)
      {
            //times++;
            if(mark[i] == 1)
            {
            for(ll j=i*i ; j <= lim ;j += 2*i) //skip to odd numbers as i*i is odd
                  {
                        mark[j] = 0;
                  }
            }
      }

      primes.pb(2);
      for(ll i=3;i<=lim;i+=2)
      {
            if(mark[i])
                  primes.pb(i);
      }
}
```

Factorization of a Number using this Sieve :

```
vector <ll> factorize(ll m)
{
      vector <ll> factors;
      factors.clear();
      ll i = 0;
      ll p = primes[i];
      while(p*p <= m)
      {
            if(m%p == 0)
            {
                  factors.pb(p);
                  while(m%p == 0)
                        m = m/p;
            }
            i++;
            p = primes[i];
      }
      if(m!=1)
            factors.pb(m);
      return factors;
}
```

Segmented Sieve

We use this sieve when array of size N does not fit in memory and we want to compute prime numbers between a range *l* and *r.* Example : *l* = 10^8, r = 10^9.

```
void sieve()
{
 for(int i = 0;i<=1000000;i++)
   p[i] = 1;
 for(int i = 2;i<=1000000;i++)
 {
   if(p[i])
   {
      for(int j = 2*i;j<=1000000;j+=i)
        p[j] = 0;
```

```
   }
 }
// for(int i=2;i<=20;i++)cout<<i<<" "<<p[i]<<endl;
}
int segmented_sieve(long long a,long long b)
{
 sieve();
 bool pp[b-a+1];
 memset(pp,1,sizeof(pp));
 for(long long i = 2;i*i<=b;i++)
 {
   for(long long j = a;j<=b;j++)
   {
    if(p[i])
    {
     if(j == i)
      continue;
     if(j % i == 0)
       pp[j-a] = 0;
    }
   }
  }
  int res = 1;
  for(long long i = a;i<b;i++)
     res += pp[i-a];
return res;
}
```

Division

Let a and b be integers. We say a divides b, denoted by a|b, if there exists an integer c such that b= ac.

Linear Diophantine Equations

A Diophantine equation is a polynomial equation, usually in two or more unknowns, such that only the integral solutions are required. An Integral solution is a solution such that all the unknown variables take only integer values. Given three integers a, b, c representing a linear equation of the form : ax + by = c. Determine if the equation has a solution such that x and y are both integral values.

General solution (Infinitely many solutions)

(x, y) = (xo + b/d *t, yo — a/d *t)

We can use *Extended Euclidean Method* above to find the xo,yo.

Chinese Remainder Theorem

Typical problems of the form "Find a number which when divided by 2 leaves remainder 1, when divided by 3 leaves remainder 2, when divided by 7 leaves remainder 5" etc can be reformulated into a system of linear congruences and then can be solved using Chinese Remainder theorem.

For example, the above problem can be expressed as a system of three linear congruences:

$$x \equiv 1 \pmod 2, \quad x \equiv 2 \bmod 3, \quad x \equiv 5 \bmod 7.$$

x % num[0] = rem[0],

x % num[1] = rem[1],

.....

x % num[k-1] = rem[k-1]

A Naive Approach is to find x is to start with 1 and one by one increment it and check if dividing it with given elements in num[] produces corresponding remainders in rem[]. Once we find such a x, we return it

Chinese remainder theorem

$$x = \sum_{0 \le i \le n-1} (rem[i]*pp[i]*inv[i])) \ \% \ prod$$

rem[i] is given array of remainders

prod is product of all given numbers

prod = num[0] * num[1] * ... * num[k-1]

pp[i] is product of all but num[i]

pp[i] = prod / num[i]

inv[i] = Modular Multiplicative Inverse of pp[i] with respect to num[i]

Code :

```
ll chinese_remainder_theorem(vector <ll> num,vector <ll> rem)
{
      // find pp vector
      vector <ll> pp; // product of all num array except num[i]
      pp.clear();
```

```
        ll prod = 1ll;
        for(ll  i=0;i<num.size();++i)
              prod *= num[i];
        for(ll  i=0;i<num.size();++i)
              pp.pb(prod/num[i]);
        // find inv[] vector
        // inv[i] is modular inverse of pp[i] with respect to num[i]
        vector <ll> inv;
        inv.clear();
        for(ll  i=0;i<pp.size();++i)
              inv.pb(modular_inverse(pp[i],num[i]-2,num[i]));
        // (a^-1)%m when m is prime is (a^(m-2))%m using fermat's
        // now use the sum formula
        ll ans = 0ll;
        for(ll  i=0;i<pp.size();++i)
        {
              ans = ans%prod + ( ((rem[i]*pp[i])%prod)*(inv[i])%prod )%prod;
              ans %= prod;
        }
        return ans;
}
```

Euler Phi Function

Euler's Phi function (also known as totient function, denoted by φ) is a function on natural numbers that gives the count of positive integers coprime with the corresponding natural number.
Thus , $\varphi(8) = 4$, $\varphi(9) = 6$

The value $\varphi(n)$ can be obtained by Euler's formula :

Let $n = p_1^{a_1} * p_2^{a_2} * \ldots \ldots \ldots * p_k^{a_k}$ be the prime factorization of n. Then

$$\varphi(n) = n * \left(1 - \frac{1}{p_1}\right) * \left(1 - \frac{1}{p_2}\right) * \ldots * \left(1 - \frac{1}{p_k}\right)$$

Code :

```
int phi[] = new int[n+1];
for(int i=2; i <= n; i++)
 phi[i] = i; //phi[1] is 0
```

```
for(int i=2; i <= n; i++)
  if( phi[i] == i )
    for(int j=i; j <= n; j += i )
        phi[j] = (phi[j]/i)*(i-1);
```

Properties :

1. If P is prime then $\varphi(p^k) = (p - 1)\, p^{(k-1)}$.
2. φ function is multiplicative, i.e. if (a,b) = 1 then $\varphi(ab) = \varphi(a)\varphi(b)$.
3. Let $d_1, d_2, \ldots d_k$ be all divisors of n (including n). Then $\varphi(d1) + \varphi(d2) + \ldots + \varphi(dk) = n$

   For Example: The divisors of 18 are 1,2,3,6,9 and 18.

   Observe that $\varphi(1) + \varphi(2) + \varphi(3) + \varphi(6) + \varphi(9) + \varphi(18) = 1 + 1 + 2 + 2 + 6 + 6 = 18$

4. Number of divisors of $n = p_1^{a1} * p_2^{a2} * \ldots\ldots * p_n^{an}$:

   $$d(n) = (a_1 + 1) * (a_2 + 1) * \ldots\ldots * (a_n + 1)$$

5. Sum of divisors:

   $$S(n) = \frac{p_1^{a_1 - 1}}{p_1 - 1} * \frac{p_2^{a_2 - 1}}{p_2 - 1} * \ldots * \frac{p_n^{a_n - 1}}{p_n - 1}$$

Wilson's Theorem

If p is a prime, then (p — 1)! = –1 (mod p)

Problem : DCEPC11B (SPOJ)

Hint :

This can be solved using Wisdom theorem

1. If n > = p ans would be 0
2. Else we have to use Wilson's theorem

   $$(p-1)! \equiv -1 (\text{mod } p)$$

   1*2*3* . . . . . *(n – 1)* (n)* . . . . . *(p – 1) ≡ –1 (mod p)

   n! (n + 1)* . . . . . *(p – 1) ≡ –1 (mod p)

   n! ≡ –1*[(n + 1)* . . . . .(p – 2)*(p – 1)^ –1 (mod p)

Lucas Theorem

In number theory, Lucas's theorem expresses the remainder of division of the binomial coefficient ${}^{m}C_{n}$ by a prime number p in terms of base p expansions of integers m and n.

Formulation :

$$\binom{m}{n} \equiv \prod_{i=0}^{k} \binom{m_i}{n_i} \pmod{p},$$

where $m = m_k p^k + m_{k-1} p^{k-1} + ... + m_1 p + m_0$ and $n = n_k p^k + n_{k-1} p^{k-1} + ... + n_1 p + n_0$

Problem : Compute ${}^{n}C_{r}$ % p.

Given three numbers n, r and p, compute the above value of ${}^{n}C_{r}$ % p.

Using Lucas Theorem ${}^{n}C_{r}$ % p

Lucas theorem basically suggests that the value of ${}^{n}C_{r}$ can be computed by multiplying results of ${}^{n_i}C_{r_i}$ where n_i and r_i are individually same-positioned digits in base p representations of n and r respectively. The idea is to one by one compute ${}^{n_i}C_{r_i}$ for individual digits n_i and r_i in base p.

Code :

```
#include<bits/stdc++.h>
using namespace std;
int Cal_nCr_mod_p(int n, int r, int p)
{


   int C[r+1];
   memset(C, 0, sizeof(C));
   C[0] = 1;
   for (int i = 1; i <= n; i++)
   {
       for (int j = min(i, r); j > 0; j–)
            C[j] = (C[j] + C[j-1])%p;
   }
   return C[r];
}
```

```
int LucasApproach(int n, int r, int p)
{
    if(r==0)
    return 1;
    else
    {
         int n_i = n%p, r_i = r%p;
          int result = (LucasApproach(n/p, r/p, p)*Cal_nCr_mod_p(n_i,r_i,p))%p;
         return result;
    }
}
int main()
{
    int n,r,p;
    cin>>n>>r>>p;
    int result = LucasApproach(n,r,p);
    cout<<"nCr mod p is "<<result;
}
```

Fermat's little Theorem

Fermat's little theorem states that if p is a prime number, then for any integer a, the number $a^p - a$ is an integer multiple of p. In the notation of modular arithmetic, this is expressed as

$$a^p \equiv a \pmod p.$$

For example, if $a = 2$ and $p = 7$, $2^7 = 128$, and $128 - 2 = 7 \times 18$ is an integer multiple of 7.

If a is not divisible by p, Fermat's little theorem is equivalent to the statement that $a^{p-1} - 1$ is an integer multiple of p, or in symbols

$$a^{p-1} = 1 \pmod p$$

For example, if $a = 2$ and $p = 7$ then $2^6 = 64$ and $64 - 1 = 63$ is thus a multiple of 7.

Problem based on Fermat's Theorem (Light's New Car) :

Statement : Given A and B, we have to find (A power B) % (10^9 + 7) where A, B<=10^100000

First let's deal with the base A. Now what we have to find is A%(10^9 + 7). This is beacause, let's suppose B = n (where n is an integer). A can be expressed as

$A = [a * (mod) + b]$ (where mod = $10^9 + 7$, a and b are integers). This implies that

$(A \uparrow n)\ \%\ mod = [(a * (mod) + b) \uparrow n] \%mod = \{[(a * (mod) + b) \uparrow \%\ mod] * [(a * (mod) + b) \uparrow \%mod] *$

$\ldots\ldots\ldots$ n times $\%\ mod = b \uparrow n$ (where b is nothing but A % mod)

Using modulo properties :

$(a * b)\ \%m = ((a\%m)\%\ (a + b)\%m = (a\%m + b\%m)\%m$

Now A% ($10^9 + 7$) can be found by iterating over the string A and generating an integer from it but at the same time taking it's modulo with ($10^9 + 7$) to prevent overflow.

Now let's deal with the power B. We can use the concept of fermat's little theorem as

$x^{p-1}\%p = 1$ (where p is a prime number)

B can be presented as B = a*(p – 1) + b (where a and b are integers and p = ($10^9 + 7$))

Hence $A^B \%p = A^{(a*(p-1)+b)} \%p = \{[A^{a*(p-1)}]\%p * [A^b]\%p\}\ \%p = (A^b)\%p$

Here b is nothing but B %(p – 1)

Using Fermat's little theorem and modulo properties

Now B%(p – 1) can found in the similar way as A % ($10^9 + 7$)

Finally

Let x = A %($10^9 + 7$), n = B %($10^9 + 7 - 1$)

Required answer would be x%($10^9 + 7$) which can be found out easily by fast modulo exponentiation in O(logn).

Miller - Rabin Primality Test

The Miller-Rabin primality test or Rabin-Miller primality test is a primality test: an algorithm which determines whether a given number is prime.

This method is a probabilistic method to find Prime number.

Input #1 : n > 3, an odd integer to be tested for primality;

Input #2 : k, a parameter that determines the accuracy of the test

Output: Composite if n is composite, otherwise probably prime

write n – 1 as $2^r.d$ with d odd by factoring powers of 2 from n – 1

WitnessLoop : repeat k times : pick a random integer *a* in the range [2, n – 2]

x ← a^d mod n

if x = 1 or y = n – 1 then

continue WitnessLoop

repeat r – 1 times:

$x \leftarrow x^2 \bmod n$

if x = 1 then

return composite

if x = n – 1 then

continue WitnessLoop

return composite

return probably prime

Example :

Input: n = 13, k = 2

1. Computed d and r such that d*2r = n – 1,
   d = 3, r = 2.
2. Call millerTest k times.

1st Iteration:

1. Pick a random number 'a' in range [2, n – 2]
   Suppose a = 4
2. Compute: x = pow(a, d) % n
   x = 43 % 13 = 12
3. Since x = (n – 1), return prime.

IInd Iteration:

1. Pick a random number 'a' in range [2 + n – 2]
   Suppose a = 5
2. Compute: x = pow(a, d) % n
   x = 53 % 13 = 8
3. x neither 1 nor 12
4. Do following (r – 1) = 1 times
   (a) x = (x * x) % 13 = (8 * 8) % 13 = 12 **(b)** Since x = (n – 1), return true.

Since both iterations return true, we return prime.

POWPOW2, Spoj

Problem :

Given three integers a, b, n, 1d á a,b,nd á 10^{5}

$a^{(b(f(n))} \bmod 1000000007$, where $f(n) = ({}^nC_0^2 + {}^nC_1^2 + \ldots\ldots + {}^nC_n^2)$

Dealing with f(n) :

The function f complicates the expression, but we can notice that f(n) = $^{2n}C_n$. It's easy to find proofs online, e.g. here, so I'll skip that.

Reducing the Exponents :

$b^{(2n, n)}$ is a huge number and we need to reduce it to a more tractable number.

Euler's theorem states that if a and m are coprime, then $a^{\varphi(m)} = 1 (\mathrm{mod}\, m)$, where $\varphi(m)$ is Euler's totient function. This is useful because $a^y \equiv a^{(y \bmod \varphi(m))} (\mathrm{mod}\, m)$.

The repeated $\varphi(m)$factors in the exponent will yield a bunch of 1s).

m = 10^9 + 7 which is a prime number, so $\varphi(m) = m - 1 = 10^9 + 6 = 2 \times 500000003$

So, we have $a^{y \bmod 100000006} \bmod 1000000007$.

The main difficulty of this problem is that our y is also an exponential, y = $b^{(2n, n)}$. In order to find the result, we need first to calculate $b^{(2n, n)} \bmod 1000000006$.

Finding $b^{(2n, n)} \bmod 1000000006$ when b is odd

Suppose b is odd. Then, we can apply Euler's theorem because and 1,000,000,006 are coprime (recall that $b \leq 10^5$ so the 500000003 factor will always be coprime with b).

$$b^{(2n, n)} \equiv b^{(2n, n) \bmod \varphi(1000000006)} \bmod 1000000006$$

$$\varphi(1000000006) = \varphi(2) \times (500000003) = (2-1) \times (500000003 - 1) = 500000002)$$

$$500000002 = 2 \times 41^2 \times 148721500000002 = 2 \times 41^2 \times 148721$$

So, we need to find (2n, n)mod 500000002 which is not prime. Therefore, we need to use another tool: the Chinese Remainder Theorem (CRT). We can calculate

(2n, n)mod 2

(2n, n)mod 41^2

(2n, n)mod 148721

and use CRT to get the result modulo 500000002.

Finding $b^{(2n, n)} \bmod 10000_0000_6$ $b^{(2n, n)}$ when b is even

Unfortunately, if b is even, b and 1000000006 are not coprime.

Therefore, we need CRT again. Our modulus is the product of two primes: 2 and 500,000,003. So, we shall find and and use CRT to get the result modulo 1,000,000,006.

Note that when b is even the result modulo 2 is always 0. So, we only need to calculate the result modulo 500000003 and $\varphi(500000003) = \varphi(1000000006)$, so this part is equal to the case when b is odd. The only difference is using CRT.

Adding everything together

After finding y = $b^{(2n, n)}$ mod 1000000006, we can calculate a y mod 1000000007 normally to get the final result.

Code :

```
#include<bits/stdc++.h>
#define ll long long int
int t;
ll a, b, n;
ll  fact[200005];
ll md = 1000000007;
long long int c_pow(ll i, ll j, ll mod)
 {
   if (j == 0)
     return 1;
   ll d;
  d = c_pow(i, j / (long long)2, mod);
  if (j % 2 == 0)
    return (d*d) % mod;
  else
      return ((d*d) % mod * i) % mod;
 }
ll InverseEuler(ll n, ll MOD)
{
  return c_pow(n, MOD - 2,MOD);
}
ll  fact_14[1700][1700];
ll  fact_B[150000];
ll min1(ll a, ll b)
{
 return a > b ? b : a;
}
void  calc_fact()
{
 fact[0] = fact[1] = 1;
 ll tmd = 148721;
  for (int i = 2; i < 200003; ++i)
  {
    fact[i] = (fact[i - 1] * i);
    if (fact[i] >= (tmd))
```

```
      fact[i] %= (tmd);
  }
 }
ll  fact_41[200005];
ll  fact_41_p[200005];
void  do_func()
{
 fact_41[0] = 1;
 fact_41_p[0] = 0;
 for (int i = 1; i < 200003; ++i)
 {
   ll y = i;
   fact_41_p[i] = fact_41_p[i - 1];
   while (y % 41 == 0)
   {
     y = y / 41;
     fact_41_p[i]++;
   }
   fact_41[i] = (y*fact_41[i - 1]) % 1681;
  }
}
ll  fact_2[200005];
void  do_func2()
{
  fact_2[0] = 1;
  for (int i = 1; i < 200005; ++i)
   {
    fact_2[i] = (i*fact_2[i - 1]) % 2;
   }
}
ll get_3rd(ll n, ll r, ll MOD)
{
   ll ans = (InverseEuler(fact[r], MOD)*InverseEuler(fact[n - r], MOD)) % MOD;
  ans = (fact[n] * ans) % MOD;
  return ans;
}
ll inverse2(ll m1, ll p1)
```

```
 {
   ll i = 1;
   while (1)
   {
    if ((m1*i) % p1 == 1)
     return i;
    i++;
   }
 }
ll chinese_remainder_2(ll n1, ll n2, ll n3)
 {
   ll p1 = 2, p2 = 1681, p3 = 148721;
   ll m1, m2, m3;
   ll i1, i2, i3;
   ll m;
   ll ans;
   m = p1*p2*p3;
   m1 = m / p1; m2 = m / p2; m3 = m / p3;
    i1 = InverseEuler(m1, p1); i2 = inverse2(m2, p2); i3 = InverseEuler(m3, p3);
   //printf("i1 = %lld i2 = %lld\n",i1,i2);
   ans = (n1*m1*i1) % m + (n2*m2*i2) % m + (n3*m3*i3) % m;
   ans = ans%m;
   return ans;
    //printf("%d\n",ans);
 }
int main()
 {
  ios_base::sync_with_stdio(false);
  cin.tie(NULL);
  calc_fact();
  do_func();
  do_func2();
  cin >> t;
  while (t--)
  {
   cin >> a >> b >> n;
    if (a == 0 && b == 0)
    {
```

```
      cout << "1\n";
        continue;
    }

    if (b == 0)
    {
      cout << "1\n";
      continue;
    }
   ll a1 = (n == 0) ? 1 : 0;
   ll a2 = (fact_41[2 * n] * inverse2(fact_41[n],1681)) % 1681;
   a2 = (a2 * inverse2(fact_41[n], 1681)) % 1681;
   a2 = (a2 * c_pow(41, fact_41_p[2 * n] - 2 * fact_41_p[n], 1681)) % 1681;
   ll a3 = get_3rd(2 * n, n, 148721);
   //cout << a1 << " " << a2 << " " << a3 << "\n";
   ll ans = chinese_remainder_2(a1, a2, a3);
   if (ans == 0)ans = 500000002;
   ll y1 = c_pow(b, ans, md - 1);
   cout << y1 << "\n";
   ll z = c_pow(a, y1, md);
   cout << z << "\n";
  }
 return 0;
}
```

Best Method for ${}^{n}C_{r}$

We want to compute C(n,r)%p where p is prime and N,R <=10^8 :

```
#include<iostream>
using namespace std;
#include<vector>
/* This function calculates (a^b)%MOD */
long long pow(int a, int b, int MOD)
{
 long long x=1,y=a;
 while(b > 0)
  {
   if(b%2 == 1)
```

```
    {
       x=(x*y);
      if(x>MOD) x%=MOD;
    }
   y = (y*y);
    if(y>MOD) y%=MOD;
   b /= 2;
  }
  return x;
}
/* Modular Multiplicative Inverse
Using Euler's Theorem
a^(phi(m)) = 1 (mod m)
a^(-1) = a^(m-2) (mod m) */
long long InverseEuler(int n, int MOD)
{
 return pow(n,MOD-2,MOD);
}
long long C(int n, int r, int MOD)
{
 vector<long long> f(n + 1,1);


 for (int i=2; i<=n;i++)
    f[i]= (f[i-1]*i) % MOD;
   return (f[n]*((InverseEuler(f[r], MOD) * InverseEuler(f[n-r],MOD))% MOD)) %
MOD;
 }
int main()
{
 int n,r,p;
 while (~scanf("%d%d%d",&n,&r,&p))
 {
   printf("%lld\n",C(n,r,p));
 }
}
```

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.

Code :

```
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
the current value of a
    in  ans. */
          if(b&1) {          //rightmost digit of b in binary form is 1.
           ans = ans*a ;
            ans = ans%c;     //at each iteration if value of ans exceeds then
reduce it to modulo c.
       }
  a = a*a;                  /
  a %= c;                   //at each iteration if value of a exceeds then reduce
it to modulo c.
  b >>= 1;                  //Trim the right-most digit of b in binary form.
}
```

Questions :

1. **Find sum of divisors of all the numbers from 1 to n . n <= 10^5.**
   (SPOJ DIVSUM)
   Here n is relatively small so we can precompute all the divisor sum using sieve like approach which runs in O(NLOGN).

Code :

```
ll  sum[5000005];
void  sieve()
{
     F(i,1,5000002)
     {
           for(ll  j=i;j<=5000002;j+=i) // every j has i as a divisor
           {
                 sum[j] += i;
```

```
            }
      }
      // subtract number itself from sum if proper divisors are required
      F(i,1,5000002)
            sum[i] -= i;
}
```

2. Find sum of divisors of a number. n <= 10^16.
(SPOJ-DIVSUM2)

Explanation :

Here our previous approach will fail since we cannot create such large array. Hence we have to factorize n in the form of p1^k1 * p2^k2…

Now we can get the sum of divisors using the formula mentioned in this booklet above.

Code :

```
/*input
3
2
10
10000000000000000
*/
#include <bits/stdc++.h>
#include<stdio.h>
using namespace std;
#define F(i,a,b) for(ll i = a; i <= b; i++)
#define RF(i,a,b) for(ll i = a; i >= b; i–)
#define pii pair<ll,ll>
#define PI 3.14159265358979323846264338327950288
#define ll long long
#define ff first
#define ss second
#define pb(x) push_back(x)
#define mp(x,y) make_pair(x,y)
#define debug(x) cout << #x << “ = “ << x << endl
#define INF 1000000009
#define mod 1000000007
#define S(x) scanf(“%d”,&x)
#define S2(x,y) scanf(“%d%d”,&x,&y)
```

```
#define P(x) printf("%d\n",x)
#define all(v) v.begin(),v.end()
#define lim 100000002
vector <bool> mark(lim+2,1);
vector <ll> primes;
void sieve() // we need primes upto 10^8
{
	//ll times = 0;
	for(ll i=3;i<=lim;i+=2)
	{
		//times++;
		if(mark[i] == 1)
		{
			for(ll j=i*i ; j <= lim ;j += 2*i)
			{
		//		times++;
				mark[j] = 0;
			}
		}
	}
	//debug(times);
	primes.pb(2);
	for(ll i=3;i<=lim;i+=2)
	{
		if(mark[i])
			primes.pb(i);
	}
}
ll power(ll a,ll b) // find a to the power b
{
	ll ans = 1ll;
	while(b > 0)
	{
		if(b&1)
			ans = ans*a;
		a = a*a;
		b /= 2;
	}
	return ans;
```

```
}
ll factorize(ll n) // find multiplication of all (p^(k+1)-1)/(p-1) where k is
the power of p in n
{
      // exhaust powers of 2 first
      ll c = 0;
      while(n%2 == 0)
      {
            c++;
            n = n/2;
      }
      ll i=0 , ans = 1ll;
      if(c>0)
            ans = (power(2ll,c+1) - 1);
      ll times = 0;
      ll p = primes[0];
      while(p*p <= n)
      {
            //times++;
            if(n%p == 0) // if p is a prime factor of n
            {
                  ll cnt = 0; // find power of p
                  while(n%p == 0)
                  {
                        //times++;
                        n /= p;
                        cnt++;
                  }
                  // update ans;
                  ll numerator = power(p,cnt+1) - 1;
                  //debug(numerator);
                  ll denom = p - 1;
                  ll curans = numerator/denom;
                  ans = ans * (curans);
            }
            //debug(n);
            if(n == 1)
```

```
                  break;
            i++;
            p = primes[i];
      }
      //debug(times);
      if(n != 1)
            ans = ans * (n+1);
      return ans;
}
int main()
{
      std::ios::sync_with_stdio(false);
      sieve();
      //cout<<primes.size(); //5761455 primes less than 10^8
      ll t;
      cin>>t;
      while(t--)
      {
            ll n;
            cin>>n;
            ll ans = factorize(n);
            ans -= n;
            cout<<ans<<endl;
      }
      return 0;
}
```

3. Find the value of 1! * 2! * 3*... N! modulo P where P = 109546051211.

FACTMUL SPOJ, Chinese Remainder Thm

Explanation :

The naive approach for calculating this value under modulo p will fail since (a*b)%p will overflow coz p is itself large.

Here is the trick :-

P = p1*p2 where p1 = 186583 p2 = 587117

Let a = 1! * 2! * 3*... N!

x1 = a%p1

x2 = a%p2

This is a set of equations satisfying the CRT criteria hence we can calculate the value of a using CRT.

Code :

```
/*input
5
*/
#include <bits/stdc++.h>
#include<stdio.h>
using namespace std;
#define F(i,a,b) for(ll i = a; i <= b; i++)
#define RF(i,a,b) for(ll i = a; i >= b; i—)
#define pii pair<ll,ll>
#define PI 3.14159265358979323846264338327950288
#define ll long long
#define ff first
#define ss second
#define pb(x) push_back(x)
#define mp(x,y) make_pair(x,y)
#define debug(x) cout << #x << " = " << x << endl
#define INF 1000000009
#define mod 109546051211 // 186583*587117
#define S(x) scanf("%d",&x)
#define S2(x,y) scanf("%d%d",&x,&y)
#define P(x) printf("%d\n",x)
#define all(v) v.begin(),v.end()
ll power(ll a,ll b,ll m)
{
        ll ans = 1ll;
        while(b > 0)
        {
                if(b&1)
                        ans = ans*a;
                a = a*a;
                ans%=m;
                a%=m;
```

```
                b /= 2;
        }
        return ans;
}
int main()
{
        std::ios::sync_with_stdio(false);
        ll n;
        cin>>n;
        //use crt since MOD = p1*p2
        ll p1 = 186583ll;
        ll p2 = 587117ll;
        // find first value with respect to p1 and second value with respect to p2
        ll ans_p1 = 1ll , ans_p2=1ll , curfactorial_p1 = 1ll , curfactorial_p2 = 1ll;
        F(i,2,n)
        {
                curfactorial_p1 = curfactorial_p1*i;
                curfactorial_p1 %= p1;
                curfactorial_p2 = curfactorial_p2*i;
                curfactorial_p2 %= p2;
                ans_p1 = ans_p1*curfactorial_p1;
                ans_p1 %= p1;
                ans_p2 = ans_p2*curfactorial_p2;
                ans_p2 %= p2;
        }
        //debug(ans_p1);
        //debug(ans_p2);
        // num[0] = p1 , num[1] = p2
        // rem[0] = ans_p1 rem[1] = ans_p2
        // pp[0] = p2 pp[1] = p1
        // prod = p1*p2
        // inv[i] = mpdular multiplicative inverse of pp[i] with respect to num[i]
        // inv[0] = inverse of p2 w.r.t p1 , inv[1] = inverse of p1 w.r.t p2
        // first remainder is ans_p1 and second remainder is ans_p2
        // (x%p1) = ans_p1
        // (x%p2) = ans_p2 , we can combine these two to find x
```

```
        // x = rem[0]*inv[0]*pp[0] + rem[1]*inv[1]*pp[1]
        ll inv_zero = power(p2,p1-2ll,p1); // fermats
        ll inv_first = power(p1,p2-2ll,p2); // fermats
        ll ans = (((ans_p1*inv_zero)%mod)*(p2%mod))%mod +
(((ans_p2*inv_first)%mod)*p1%mod)%mod;
        ans %= mod;
        cout<<ans<<endl;
        return 0;
}
```

TRY YOURSELVES

http://www.spoj.com/problems/MAIN74/ //Find first few values and observe pattern
http://www.spoj.com/problems/DIVSUM/ // precomputation or multiplicative formula
http://www.spoj.com/problems/DIVSUM2/ // multiplicative formula
http://www.spoj.pl/problems/NDIVPHI/ // can be solved only using BIG INTEGER or in PYTHON
http://www.codechef.com/problems/THREEDIF // very simple
http://www.spoj.com/problems/LCPCP2/ // very simple
http://www.spoj.com/problems/GCD2/ // tricky
http://www.spoj.com/problems/FINDPRM/
http://www.spoj.com/problems/TDKPRIME/ // simple sieve and precompute
http://www.spoj.com/problems/TDPRIMES/ // same as TDKPRIME
http://www.spoj.com/problems/PRIME1/ //segmented sieve
http://www.spoj.com/problems/FACTMUL/ // CRT
http://www.spoj.com/problems/FACTCG2/ // factorization
http://www.spoj.com/problems/ALICESIE/ // formula
http://www.spoj.com/problems/AMR10C/ // factorization
http://www.spoj.com/problems/DCEPC11B/ // Wilson Theorem
http://www.codechef.com/problems/SPOTWO
http://www.spoj.com/problems/DCEPC13D/ // CRT + LUCAS + FERMAT
http://www.spoj.com/problems/CUBEFR/
http://www.spoj.com/problems/NFACTOR/
http://www.spoj.com/problems/CSQUARE/
http://www.spoj.com/problems/CPRIME/
http://www.spoj.com/problems/ANARC09C/
http://www.spoj.com/problems/GCDEX/ read this :-
https://discuss.codechef.com/questions/72953/a-dance-with-mobius-function

http://www.spoj.com/problems/AMR11E/ // easy sieve

https://www.hackerearth.com/challenge/competitive/code-monk-number-theory-i/problems/

https://www.hackerearth.com/challenge/competitive/code-monk-number-theory-ii/problems/

https://www.hackerearth.com/challenge/competitive/code-monk-number-theory-iii/problems/

Advanced Problem :- https://www.hackerrank.com/challenges/ncr/problem

SELF STUDY NOTES

SELF STUDY NOTES