Preparation for contest

Basic Number Theory-1:

Prepared by: zzaman asad, Department of ICT, MBSTU

1.Modular arithmetic

- (a+b)%c=(a%c+b%c)%c
- (a∗b)%c=((a%c)∗(b%c))%c
- (a−b)%c=((a%c)−(b%c)+c)%c
- (a/b)%c=((a%c)∗(b^−1%c))%c

Note: In the last property, b^−1 is the multiplicative modulo inverse of b and c.

2. Modular exponentiation

Pow():

```
//O(n)
ll recursivePower(ll x,ll n)
{
    if(n==0)
        return 1;
    return x*recursivePower(x,n-1);
}
```

Bigmod():

```
O(log(n))
ll bigmod(ll a,ll b,ll m)
{
   if(b==0) return 1%m;
    ll x=M(a,b/2,m);
    x=(x*x)%m;
   if(b%2==1)
    x=(x*a)%m;
    return x;
}
```

3. Greatest Common Divisor (GCD)

Gcd():

```
O(log(max(A,B)))
ll GCD(ll A, ll B)
 {
    if(B==0)
        return A;
    else
        return GCD(B, A % B);
 }
```

4. Extended Euclidean algorithm

If GCD(A,B) has a special property so that it can always be represented in the form:

$$Ax+By = GCD(A,B)$$

The coefficients (x and y) of this equation will be used to find the modular multiplicative inverse. The coefficients can be zero, positive or negative in value.

This algorithm takes two inputs as A and B and returns GCD(A,B) and coefficients of the above equation as output.

Key idea:

$B*x_1+ (A \% B)*y_1=GCD(A,B)$ --------(1)

Here, $A\%B=A−B∗\lfloor A/B\rfloor$

$B*x_1+ (A - \lfloor A/B\rfloor*B)*y_1=GCD(A,B)$ -----(2)

$B*(x_1 - \lfloor A/B\rfloor*y_1)+A*y_1=GCD(A,B)$. ---(3)

From (1) and (3)

- $x=y_1$
- $y=x_1- \lfloor A/B\rfloor*y_1$

When is this algorithm used?

This algorithm is used when A and B are co-prime. In such cases, x becomes the multiplicative modulo inverse of A under modulo B, and y becomes the multiplicative modulo inverse of B under modulo A. This has been explained in detail in the ***Modular multiplicative inverse*** section.

```
/// extended euclid O(log(max(A,B)))
ll d, x, y;
void extendedEuclid(ll A,ll B) {
    if(B == 0) {
        d = A;
        x = 1;
        y = 0;
    }
    else {
        extendedEuclid(B, A%B);
        ll temp = x;
        x = y;
        y = temp - (A/B)*y;
    }
}
// for gcd print d and for coefficients
Print x,y
```

5. Modular multiplicative inverse

Multiplicative inverse? If $A*B=1$ you are required to find B such that it satisfies the equation. The solution is simple. The value of B is $1/A$ or A^-1. Here, B is the multiplicative inverse of A.

What is modular multiplicative inverse? If you have two numbers A and M, you are required to find B such it that satisfies the following equation:

$$(A*B)\%M = 1 \text{ or } A*B\equiv 1 \pmod M$$

Where,B is in the range [1,M-1] and modular multiplicative inverse of A under modulo M. **Inverse exists,** only when A and M are coprime or $GCD(A,M)=1$

Why is B in the range [1,M-1]?

$(A*B)\%M=((A\%M)*(B\%M))\%M$

Since we have B%M, the inverse must be in the range [0,M-1]. However, since 0 is invalid, the inverse must be in the range [1,M-1].

Approach 1 (naive approach)

```
///O(M)
ll modInverse(ll A,ll M)
{
    A=A%M;
    for(ll B=1;B<M;B++)
        if((A*B)%M)==1)
            return B;
}
```

Approach 2(when A,M are Co-prime)

If A and M are coprime or GCD(A,M)=1

$A*x + M*y=1$

In the extended Euclidean algorithm, x is the modular multiplicative inverse of A under modulo M. Therefore, the answer is x. You can use the extended Euclidean algorithm to find the multiplicative inverse.

```
///O(log(max(A,M)))
ll d,x,y;
ll modInverse(int A, int M)
{
  extendedEuclid(A,M);

//goto  extendedeuclid function

  return (x%M+M)%M;//x may be negative
}
```

Approach 3 (used only when M is prime)

This approach uses Fermat's Little Theorem.

$$A\text{^}(M-1)\equiv 1 \pmod M$$

By multiplying with A^-1 both side ,the equation can be rewritten as follows:

$$A\text{^}{-1}\equiv A\text{^}(M-2) \pmod M \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots(1)$$

So, The formula for A^-1 is in the form of exponents. Therefore, **bigmod()** can be used to determine the result.

```
///O(log(M))
ll modInverse(ll A,ll M)
{
    return bigmod(A,M-2,M);
   ///bigmod() function
}
```

__**Where use:**__

Modular inverse is used to solve $(A\text{^}B/C)\%M$,As follows:

$(A\text{^}B/C)\ \%\ M = (A\text{^}B*C\text{^}{-1})\ \%\ M=$
$(bigmod(A,B,M) * (modInverse(C,M)))\%M$

6.Sieve of Eratosthenes:

- **COD(Count the number of Divisor):**

O(√N)

```
void checkprime(ll N)
{
  ll count = 0;
  for(ll i = 2;i * i <=N;++i )
   {
     if( N % i == 0)
       {
         if( i * i == N ) count++;
         else //i<sqrt(N) and (N/i)> sqrt(N)
             count += 2;
       }
   }
}
//if count >0 then N is composite else prime
```

- **General Sieve:**

```
O(Nloglog(N))
void sieve(ll N)
{
   bool isPrime[N+1];
   for(ll i = 0; i <= N;++i)
   isPrime[i] = true;
   isPrime[0] = false;
   isPrime[1] = false;
   for(ll i = 2; i * i <= N; ++i)
   {
     if(isPrime[i] == true)
     {
//Mark all the multiples of i as composite
numbers
       for(ll j = i * i; j <= N ;j += i)
        isPrime[j] = false;
     }
   }
}
```

- **Prime Generator:** O(Nloglog(N))

```
ll a[m+7],l=0;
bool p[m+7];
void sieve(ll m)
{


    memset(p,true,sizeof(p));
    p[0]=p[1]=false;
    for(ll i=2; i<m; i++)
    {
        if(p[i])
        {
            a[l++]=i;
            for(ll j=i*i; j<m; j+=i)
                p[j]=false;
        }
    }

}
```

- **Finding All divisor(Including Non prime and Prime)**

```
O(Nlog(N))
vll v[1000001];
void div(ll n)
{
    for(ll i=1;i<=n;i++)
    {
        for(ll j=i;j<=n;j+=i)
            v[j].pb(i);
    }
}
///Print 0 to v[i].size() to print all
divisor of i
```

- **Finding Prime Factorization :**

$O(\sqrt{\boxed{N}})$

```
void factor(ll n)
{
    ll b=n;
        vector<ll>fact;
        vector<int>pow;
        for(ll i=2;i*i<=n;i++)
        {
            int p=0;
            while(n%i==0)
            {
                ++p;
                n/=i;
            }
            if(p>1)
            {
                fact.push_back(i);
                pow.push_back(p);
            }
        }
        if(n>1)
        {
            fact.push_back(n);
            pow.push_back(1);
        }
}
```

Here, The factorization of $N =$ $p_1^{q_1} * p_2^{q_2} * \ldots * p_k^{q_k}$ where p1,p2, , ,pk are the prime factors of N and q1,q2...qk are the powers of the respective prime factors.

- **NOD(Number of Prime Divisor):**

NOD $=$ $(q_1+1)*(q_2+1)*\ldots*(q_k+1)$

- **SOD(Sum of Prime Divisor):**

$$SOD = \frac{p_1^{q1+1}-1}{p_1-1} * \ldots\ldots * \frac{p_k^{qk+1}-1}{p_k-1}$$

OR ,

$$SOD = (p_1^0+p_1^1+\ldots+p_1^{q1}) * (p_2^0+p_2^1+\ldots+p_2^{q2}) ** (p_k^0+p_k^1+\ldots+p_k^{qk})$$

- **Segment Sieve:**

```
O(sqrt(N))
l= lower range and r =higher range
bool isPrime[r - l + 1]; //filled by true
for (ll i=2;i*i<=r;++i)
{
 for(ll j=max(i*i,(l+(i-1))/i*i);j<=r;j+=i)
        isPrime[j - l] = false;
}
for (ll i = max(l, 2); i <= r; ++i) {
    if (isPrime[i - l])
    {
        //then i is prime
    }
}
```

6.Primality Test:

$O(\sqrt{\boxed{N}})$

```
bool check(ll n)
{
    if(n==2)return true;
    if(n%2==0)return false;
    for(ll i=3; i<=sqrt(n); i+=2)
        if(n%i==0)return false;
    return true;
}
```

- **Fermat Primality Testing:**

This method is a probabilistic method

If n is a prime number, then for every a, 1 <= a < n,

$a^{n-1} \equiv 1 \pmod{n}$

OR

$a^{n-1} \% n = 1$

Example: Since 5 is prime, $2^4 \equiv 1 \pmod 5)$
$3^4 \equiv 1 \pmod 5$ and $4^4 \equiv 1 \pmod 5$

Since 7 is prime, $2^6 \equiv 1 \pmod 7$,
$3^6 \equiv 1 \pmod 7$, $4^6 \equiv 1 \pmod 7$
$5^6 \equiv 1 \pmod 7$ and $6^6 \equiv 1 \pmod 7$

Algorithm Of Fermat Prime test

// Higher value of k indicates probability of correct

// results for composite inputs become higher. For prime

// inputs, result is always correct

1) Repeat following k times:

a) Pick a randomly in the range [2, n - 2]

b) If $a^{n-1} \not\equiv 1 \pmod n$, then return false

2) Return true [probably prime].

```
///O(k)
bool fermat(ll n)
{
    ll k=1000000;
    if (n <= 1 || n == 4)   return false;
    if (n <= 3) return true;
    while (k--)
    {
        ll a = 2+ rand()%(n-4);
        if (bigmod(a, n-1, n) != 1)
        ///Just call big mod() algorithm
            return false;
    }

    return true;
}
```

- **Miller Rabin Primality Test:** This method is a probabilistic method and better than Fermat test

bool isPrime(int n, int k)

1) Handle base cases for n < 3
2) If n is even, return false.
3) Find an odd number d such that n-1 can be written as $d*2^r$. Note that since n is odd, (n-1) must be even and r must be greater than 0.
4) Do following k times
    if (millerTest(n, d) == false)
        return false
5) Return true.

bool millerTest(int n, int d)

1) Pick a random number 'a' in range [2, n-2]
2) Compute: x = pow(a, d) % n
3) If x == 1 or x == n-1, return true.

// Below loop mainly runs 'r-1' times.

4) Do following while d doesn't become n-1.
    a) x = (x*x) % n.
    b) If (x == 1) return false.
    c) If (x == n-1) return true.

```
bool millerTest(ll d,ll n)
{
    ll a = 2 + rand() % (n - 4);
    ll x = bigmod(a, d, n);
    if (x == 1 || x == n-1)
        return true;
    while (d != n-1)
    {
        x = (x * x) % n;
        d *= 2;
        if (x == 1)  return false;
        if (x == n-1)  return true;
    }
    return false;
}
bool isPrime(ll n)
{
    ll k=100;
    if (n <= 1 || n == 4) return false;
    if (n <= 3) return true;
    if(n%2==0) return false;
    ll d = n - 1;
    while (d % 2 == 0)
        d /= 2;
    for (int i = 0; i < k; i++)
        if (millerTest(d, n) == false)
            return false;
    return true;
}
```

- **Euler's Totient Function:**

Euler's Totient function **Φ(n) for an input n is count of numbers in {1, 2, 3, …, n} that are relatively prime to n or GCD(n, k) = 1.** The numbers whose GCD (Greatest Common Divisor) with n is 1.

Logic:

Here, The factorization of N = $p_1^{q_1} * p_2^{q_2} * \ldots * p_k^{q_k}$ where p1,p2, , ,pk are the prime factors of N and q1,q2...qk are the powers of the respective prime factors.

$$\begin{aligned}\varphi(n) &= \varphi\left(p_1^{k_1}\right)\varphi\left(p_2^{k_2}\right)\cdots\varphi\left(p_r^{k_r}\right) \\ &= p_1^{k_1}\left(1-\frac{1}{p_1}\right)p_2^{k_2}\left(1-\frac{1}{p_2}\right)\cdots p_r^{k_r}\left(1-\frac{1}{p_r}\right) \\ &= p_1^{k_1}p_2^{k_2}\cdots p_r^{k_r}\left(1-\frac{1}{p_1}\right)\left(1-\frac{1}{p_2}\right)\cdots\left(1-\frac{1}{p_r}\right) \\ &= n\left(1-\frac{1}{p_1}\right)\left(1-\frac{1}{p_2}\right)\cdots\left(1-\frac{1}{p_r}\right).\end{aligned}$$

O(√N) /// for finding **Φ(N)**

```
ll phi(ll n)
{
    ll result = n;
    for (ll i=2; i*i<=n;i++)
    {
        if (n % i == 0)
        {
            while (n % i == 0)
                n /= i;
            result -= result / i;
        }
    }
    if (n > 1)
        result -= result / n;
    return result;
}
```

find all Φ(n) from 1 to N

```
///O(log(Log(N))) //you have to done pre calculation
ll phi[1000006];
bool mark[1000006];
void computeTotient(ll n)
{
    for (ll i=1; i<=n; i++)
        phi[i] = i;
        mark[1]=1;
    for (ll i=2; i<=n; i++)
    {
        if (!mark[i])
        {
            for (ll j= i; j<=n; j += i)
            {
                mark[j]=1;
            phi[j] = phi[j]*(i-1)/i;
            }
        }
    }
}
```

Some Interesting Properties of Euler's Totient Function

1) For a prime number p, Φ(p)=(p-1). For example Φ(5) = 4

2) two numbers a and b, if gcd(a, b) = 1, then Φ(ab) = Φ(a) * Φ(b)

3) two prime numbers p and q, Φ(pq) = (p-1)*(q-1)

4) If p is a prime number, then $\Phi(p^k) = p^k - p^{k-1}$.

5) Sum of values of totient functions of all divisors of n is equal to n.

$$\sum_{d|n} \varphi(d) = n,$$

6) ***Euler's theorem*** :

The theorem states that if n and a are coprime (or relatively prime) positive integers, then

$a^{\Phi(n)} \equiv 1 \pmod{n}$

when n is prime say p, Euler's theorem turns into ***Fermat's little theorem***

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

Basic of Combinatorics:

Prepared by: zzaman asad, Department of ICT, MBSTU

- **Permutation :**

Permutations of choosing R distinct objects out of a collection of N objects can be calculated using the following formula (Way of arrangement matter):

$$^{N}P_{R} = \frac{N!}{(N-R)!}$$

```
P(n, r) = P(n-1, r) + r* P(n-1, r-1)
= n* (n-1)*(n-2)*…….*(n-r+1)
///O(r) corner case n<r or n=0, nCr=0
ll npr(ll n,ll r)
{
    ll p=1,m=1;
    fr(i,0,r) /// r times
    {
        m*=n;
        n--;
    }
    return m;
}
```

- **Combination:**

Combinations of choosing R distinct objects out of a collection of N objects can be calculated using the following formula (Way of arrangement does not matter):

$$^{N}C_{R} = \frac{N!}{(N-R)! \times R!}$$

So,

```
C(n, r) = C(n-1, r-1) + C(n-1, r)
C(n, 0) = C(n, n) = 1
= n*(n-1)*(n-2)*…….*(n-r+1)/(1*2*………*r)
```

```
///O(min(r, n-r))corner case n<r or n=0, nCr=0
ll ncr(ll n,ll r)
{
    ll p=1,m=1;
    r=min(n-r,r);
    fr(i,0,r) /// r times
    {
        m*=n;
        p*=(i+1);
        n--;

    }
    return m/p;
}
```

- Basic Combinatorics Rules:

Suppose there are two sets A and B , if there are **X** number of ways to choose one from **A** and **Y** number of ways to choose one from **B** ,

1. **The Rule of Product**: **X×Y** = number of ways to choose two elements, one from **A** and one from **B**.
2. **The Rule of Sum**: **X+Y** = number of ways to choose one element that can belong to either **A** or to **B**.

- **Permutations with repetition:** If N objects out of N_1 objects are of type 1, N_2 objects are of type 2, ..., N_k objects are of type k, then number of ways of arrangement of these N objects:

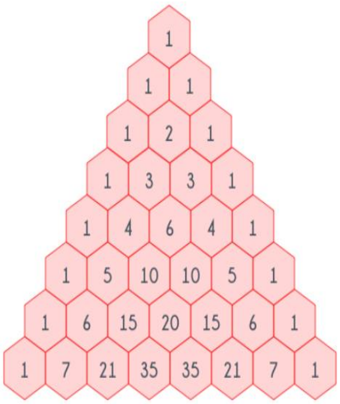
$$\frac{N!}{N_1!N_2!...N_k!}$$

- **Combinations with repetition:** If N elements out of which we want to choose K elements and it is allowed to choose one element more than once, then number of ways are given by:

$$^{N+K-1}C_{K} = \frac{(N+K-1)!}{(K)!(N-1)!}$$

- **Pascal Triangle:**

The i th row there are i elements, where $i \geq 1$ j th element of ith row is equal to ${}^{i-1}C_{j-1}$ where $1 \leq j \leq i$.



Here, The corner elements of each row are always equal to = 1 (${}^{i-1}C_0$ and ${}^{i-1}C_{i-1}$, $i \geq 1$). All the other (i,j) th elements of the triangle, (where $i \geq 3$ and $2 \leq j \leq i-1$) , are equal to the sum of $(i-1,j-1)$ th and $(i-1,j)$ th element.

```
///O(n*n)
```

```
ll dp[N+1][N+1];
void pascal_triangle()
{
    memset(dp,0,sizeof(dp));
    for(int i = 0;i<N;i++)
        dp[i][0]=dp[0][i]=1;

    for(int i = 1;i<N;i++)
    {
         for(int j = 1;j<N;j++)
         dp[i][j] = dp[i-1][j]+dp[i][j-1];
    }
    /// print dp[i-1][j-1]
}
```

Rules of Pascal Triangle:

- dp[i][j] denotes , ${}^{i+j}C_i$
- **Hockey Stick Rule:**

$$\sum_{i=0}^{r} {}^{n+i}C_i = \sum_{i=0}^{r} {}^{n+i}C_n = {}^{n+r+1}C_r = {}^{n+r+1}$$

- The sum of all the elements in ith row is equal to $2^{(i-1)}$, where $i \geq 1$.

- How many different ways are there to represent N as sum of K non-zero integers = ${}^{N-1}C_{K-1}$