Math

<table>
<tr><td>Check if a number is prime
O(sqrt(n))</td><td>

```
bool IsPrime(int n)
{
   if(n<=1) return false ;
   if(n==2) return true ;
   if(n%2==0) return false;
   for(int i=3 ; i*i<=n ; i +=2){
       if(n%i==0)
          return false ;
    }
    return true;
}
```

</td></tr>
<tr><td>Factorization
O(sqrt(n))</td><td>

```
vector<int>v; //Factors
for (int i = 1; i*i <= n; i++) {
    if (n%i == 0) {
        if (i != n / i) //2 different factors
            v.push_back(i), v.push_back(n / i);
        else
            v.push_back(i); //(perfect square)
    }
}
```

</td></tr>
<tr><td>Sieve
O(n.log(log(n)))</td><td>

```
bool primes[N + 5];
void Sieve(){
    primes[0] = primes[1] = 0;
    for (ll i = 2; i*i <= N + 3; i++) {
        if (primes[i]) {
            for (int j = i * 2; j <= N + 3; j += i)
                primes[j] = 0;
        }
    }
}
```

</td></tr>
<tr><td>Find the smallest prime factor of all number in range [2,N)
O(n.log(log(n)))</td><td>

```
int spf[N];
void Sieve() {
    spf[1] = 1;
    for (int i = 2; i < N; i++) spf[i] = i;
    for (int i = 4; i < N; i += 2) spf[i] = 2;
    for (int i = 3; i < N; i++) {
        if (spf[i] == i) {
            for (int j = i; j < N; j += i)
                if (spf[j] == j) spf[j] = i;
        }
    }
}
```

</td></tr>
<tr><td>Finding the prime factorization with powers
O(n.log(log(n)) + log(num))

I.e x = p1^m1 * p2^m2 * …. * pn^mn
Generate the spf and then do this.</td><td>

```
int power[N];
void GetFactors(int num)
{
    int cnt = 1, last = spf[num];
    while (num != 1) {
        num = num / spf[num];
        if (last == spf[num]) {
            cnt++;
            continue;
        }
        power[last] = cnt;
        cnt = 1;
        last = spf[num];
    }
}
```

</td></tr>
</table>

<table>
<tr><td>Generate all divisors of all numbers in range [1,n]
O(n.log(n))

Divs[i] contains all divs of number i</td><td><pre><code>vector<int> divs[N];
void generateDivisors(int n) {
    for (int i = 1; i <= n; ++i)
        for (int j = i; j <= n; j += i)
            divs[j].push_back(i);
}</code></pre></td></tr>
<tr><td>GCD & LCM
O(log(max(a,b))</td><td><pre><code>ll gcd(ll a, ll b) { return (b == 0 ? a : gcd(b, a % b)); }

ll lcm(ll a, ll b){ return ((a*b) / gcd(a, b)); }</code></pre></td></tr>
<tr><td>Extended Euclid
O(log(max(a,b)))

Returns the Bezout's coefficients of the smallest positive linear combination of a and b.
(i.e. GCD(a, b) = s.a + t.b)</td><td><pre><code>pair<int, int> extendedEuclid(int a, int b) {
    if (b == 0) return { 1, 0 };
    pair<int, int> p = extendedEuclid(b, a % b);
    int s = p.first, t = p.second;
    return { t, s - t * (a / b) };
}</code></pre></td></tr>
<tr><td>Fast power with modular exponentiation
O(log(exp))</td><td><pre><code>ll fastpower(ll base, ll exp)
{
    ll res = 1;
    while (exp > 0) {
        if (exp & 1) res = (res*base) % Mod;
        base = (base*base) % Mod;
        exp = exp >> 1;
    }
    return res;
}</code></pre></td></tr>
<tr><td>Mod of very large numbers
O(sz(num))
Mod rules
//(a * b) % m= ((a % m) * (b % m)) % m

//(a + b) % m= ((a % m) + (b % m)) % m</td><td><pre><code>int mod(string num, int MOD)
{
    int res = 0;
    for (int i = 0; i < num.size(); i++)
        res = (res*10 + num[i] - '0') %MOD;
    return res;
}</code></pre></td></tr>
<tr><td>Modular inverse
O(log(m))
Returns the modular inverse of the given number modulo m. (m is prime)
// (i.e. (a * mod_inverse(a)) == 1 (mod m)).
Calls the fast power function.</td><td><pre><code>int modInverse(int a, int m) {
    return power(a, m - 2, m);
}</code></pre></td></tr>
<tr><td>Combinations (nCr)
O(r)

Call the function as nCr(n, min(r, n-r)) for better performance</td><td><pre><code>int nCr(int n, int r) {
    if (n < r) return 0;
    if (r == 0) return 1;
    return n * nCr(n - 1, r - 1) / r;
}</code></pre></td></tr>
<tr><td>Combinations (nCr) using pascal’s triangle
O(n^2)

Comb[n][r] = nCr</td><td><pre><code>int comb[N][N];
void buildPT(int n) {
    for (int i = comb[0][0] = 1; i <= n; ++i)
        for (int j = comb[i][0] = 1; j <= i; ++j)
            comb[i][j] = (comb[i - 1][j] + comb[i - 1][j - 1]) % MOD;
}</code></pre></td></tr>
</table>

Derangements
O(n)
	<pre>ll fact(int n)
	{
	    ll ret = 1;
	    for (int i = n; i > 1; i--) ret = ret * 1LL * i;
	    return ret;
	}
	
	int pww(int p) { return p & 1 ? -1 : 1; }
	
	ll dearr(int n)
	{
	    ll tmp = fact(n), sum = 0;
	    for (int i = 0; i <= n; i++) sum += (pww(i)*tmp) / fact(i);
	    return sum;
	}</pre>

Euler Totient function
O(n.log(log(n)))
 	
Calculates the number of coprimes of N starting from 1 to N-1
 	
phi[i*j] = phi[i] * phi[j]
	<pre>int phi[N];
	void EulerTotient()
	{ for(int i=0; i<N; ++i) phi[i] = i;
	    for(int i=2; i<N; ++i){
	        if(phi[i] == i){
	            for(int j=i; j<N; j+=i) phi[j] -= phi[j] / i;
	        }
	    }
	}</pre>

Mobius function
O(n.log(log(n)))
 	
$\mu(n) \equiv \begin{cases} 0 & \text{if } n \text{ has one or more repeated prime factors} \\ 1 & \text{if } n = 1 \\ (-1)^k & \text{if } n \text{ is a product of } k \text{ distinct primes,} \end{cases}$
	<pre>int mu[N];
	void mobius()
	{
	    mu[1] = 1;
	    for (int i = 1; i <= N; ++i)
	        for (int j = 2 * i; j <= N; j += i)
	            mu[j] -= mu[i];
	}</pre>

Is power of 2?
O(1)
	<pre>bool isPowerOfTwo (int x)
	{
	  return x && (!(x&(x-1)));
	}</pre>

Given a permutation, what is its index?	<pre>// Given a permutation, what is its index?
	int PermToIndex(vector<int> perm) {
	    int idx = 0;
	    int n = sz(perm);
	    for (int i = 0; i < n; ++i) {
	        // Remove first, and Renumber the remaining elements to remove gaps
	        idx += Fact[n-i-1] * perm[i];
	        for(int j = i+1; j < n; j++) perm[j] -= perm[j] > perm[i];
	    }
	    return idx;
	}</pre>

Given a permutation length, what is the ith permutation?	<pre>// Given a permutation length, what is the ith permutation?
	vector<int> nthPerm(int len, int nth) {
	    vector<int> identity(len+1), perm(len);
	    for(int i=1; i<=len; i++) identity[i] = i;
	    for (int i = len - 1; i >= 0; --i) {
	        int p = nth / Fact[i]; //Fact[i] = factorial(i)
	        perm[len - 1 - i] = identity[p];
	        identity.erase(identity.begin() + p);
	        nth %= Fact[i];
	    }
	    return perm;
	}</pre>

Divisibility:

for divisible (2 power n) -> the least n digit must be divisible by (2 power n)
for divisible (5 power n) -> the least n digit must be divisible by (5 power n)
for divisible 3 sum of digit must be divisible 3
for divisible 9 sum of digit must be divisible 9
for diviaible x (where x is not prime or prime power n) number must be divisible by all prime x1^n,x2^m…. Where x=(x1^n)*(x2^m)*...
for divisible 7 take the least digit , mul 2 then sub from the rest the result must be divisible 7
ex: 203 > 3*2=6 > 20-6=14 > 14 is divis 7 then the number is divisible 7 too

divisible 7	mul 2 > sub from the rest	**divisible 13**	mul 4 > add to the rest
divisible 11	mul 1 > sub from the rest	**divisible 19**	mul 2 > add to the rest
divisible 17	mul 5 > sub from the rest	**divisible 23**	mul 7 > add to the rest
divisible 27	mul 8 > sub from the rest	**divisible 29**	mul 3 > add to the rest

Mod rules:

(a+b)%n= (a%n+b%n)%n same for sub & mul
(a^x)%n=((a%n)^x)%n
(a^b)%n=((a^(b/2))%n*(a^(b/2))%n)%n if b%2==0
a%(2^n)=a&((2^n)-1) ex a%2=a&1

Omar's math notes:

* Permutations Application:
  - Say we have a permutation: 2 0 1 3
  - Applying a permutation on other, aka multiplication, means to map its values according to the permutation.
  So (2 0 1 3) MEANS: 0 -map-> 2   1 -map-> 0   2 -map-> 1   3 -map-> 3
    Then (2 3 1 0) * (2 0 1 3) = 1 3 0 2
- Properties: It's associative (like numbers multiplication) and NOT Commutative (like numbers subtraction).
* Permutation is set of disjoint cycles. (if you followed which value replace other, you create a cycle)
  Let Say we have permutation p: 3 2 1 0 4
    0 -> 3
    3 -> 0   // End of Even Cycle
    1 -> 2
    2 -> 1   // End of Even Cycle
    4 -> 4   // End of ODD Cycle, with one element

* Applying a permutation ONCE on a cycle, divide even cycle to 2 cycles of length cycleLen/2 and odd cycle remain same.
* A cycle of length N, if applied N times, it backs to is origin!
  (0 1 2 3) * (3 0 1 2)^1 = 3 0 1 2
  (0 1 2 3) * (3 0 1 2)^2 = 2 3 0 1   Notice, the rotation of the cycle
  (0 1 2 3) * (3 0 1 2)^3 = 1 2 3 0
  (0 1 2 3) * (3 0 1 2)^4 = 0 1 2 3   We backed again!

* We need **LCM** between cycles length, to know when we all back to original in same time.
* log(n!) = log(1) + log(2) + ... + log(n)
* Count of digits of n = 1 + (int)log10(n)
* Sum of first n terms of **arithmetic progression** with difference d between each two terms = (n/2) × (2 × a_1 + (n-1) × d)
* Sum of first n terms of **geometric progression** with ratio r between each two terms = a_1 × (1 - r^n) / (1 - r)
* If a number N = a^i × b^j × . . . × c^k, then the sum of divisors of N is (a^(i+1)−1)/(a−1) × (b^(j+1)−1)/(b−1) ×...× (c^(k+1)−1)/(c−1).
* To set the n_th bit in a number num with 1 : num |= (1 << n), with 0 : num &= ~(1<<n)

Data Structures.

Sparse Table

- The data has to be immutable (doesn't change often).
- If the data changed, the whole table has to be recomputed.
- Sparse table only works with duplicate invariant functions like max, min, gcd, lcm.
- Total complexity n.log(n).
- Query complexity O(1).
- This example solves an RMQ problem.

```
const int N = 100100, M = 20;

int n, a[N], ST[M][N], LOG[N];

// Computes the floor of the log of integer from 1 to n.
// After calling this function, LOG[i] will be equals to floor(log2(i)).
// O(n)
void computeLog() {
    LOG[0] = -1;
    for (int i = 1; i <= n; ++i) {
        LOG[i] = LOG[i - 1] + !(i & (i - 1));
    }
}

// Builds sparse table for computing min/max/gcd/lcm/..etc
// for any contiguous sub-segment of the array.
// This is an example of sparse table computing the minimum value.
// O(n.log(n))
void buildST() {
    computeLog();

    for (int i = 0; i < n; ++i) {
        ST[0][i] = i;
    }

    for (int j = 1; (1 << j) <= n; ++j) {
        for (int i = 0; (i + (1 << j)) <= n; ++i) {
            int x = ST[j - 1][i];
            int y = ST[j - 1][i + (1 << (j - 1))];

            ST[j][i] = (a[x] <= a[y] ? x : y);
        }
    }
}

// Queries the sparse table for the value of the interval from l to r.
// O(1)
int query(int l, int r) {
    int g = LOG[r - l + 1];

    int x = ST[g][l];
    int y = ST[g][r - (1 << g) + 1];

    return (a[x] <= a[y] ? x : y);
}
```

Monotonic queue

- Solves sliding window problems.
- Usually done after a Binary Search on the window size and it tries all possible windows on that size to find a suitable answer.
- The data has to change monotonically, so either non-increasing or non-decreasing.
- Complexity of finding min, max is O(1)
- First we fill the queue with size-1 elements in the window, then push the sz_th element, query min or max and then pop the first element and repeat.

```
struct node {
    int val, min, max;
    node(int v, int m, int ma) :val(v), min(m), max(ma) {}
};
struct monotonicQ {
    stack<node> st1, st2;
    void mPush(int x) {
        node tmp(x, x, x);
        if (!st1.empty()) {
            tmp.min = min(tmp.min, st1.top().min);
            tmp.max = max(tmp.max, st1.top().max);
        }
        st1.push(tmp);
    }
    void mPop() {
        if (!st2.empty()) {
            st2.pop();
        }
        else {
            node tmp = st1.top();
            tmp.min = tmp.val;
            tmp.max = tmp.val;
            st2.push(tmp);
            st1.pop();
            while (!st1.empty()) {
                tmp = st1.top();
                st1.pop();
                tmp.min = min(tmp.val, st2.top().min);
                tmp.max = max(tmp.val, st2.top().max);
                st2.push(tmp);
            }
            st2.pop();
        }
    }
    int getMin() {
        int res = 2e9;
        if (!st1.empty())res = min(res, st1.top().min);
        if (!st2.empty())res = min(res, st2.top().min);
        return res;
    }

    int getMax() {
        int res = -1;
        if (!st1.empty())res = max(res, st1.top().max);
        if (!st2.empty())res = max(res, st2.top().max);
        return res;
    }
};
```

```
for (int i = 0; i < m - 1; i++) mq.mPush(arr[i]);
for (int i = m - 1; i < n; i++) {
    mq.mPush(arr[i]); int mn = mq.getMin(), mx = mq.getMax();
    //Do what the problem asks.
    mq.mPop();
}
```

Disjoint Union Set (DSU)

```
int par[N], szz[N];

void init() { for (int i = 0; i < N; i++) par[i] = i, szz[i] = 1; }

int find(int u) { return u == par[u] ? u : par[u] = find(par[u]); }

bool join(int u, int v)
{
    int paru = find(u), parv = find(v);
    if (paru != parv) {
        if (szz[paru] > szz[parv]) szz[paru] += szz[parv], par[parv] = paru;
        else szz[parv] += szz[paru], par[paru] = parv;
        return true;
    }
    return false;
}
```

Binary Indexed Tree (BIT)

```
const int N = (1 << 17);
int n, BIT[N];

// Updates the given index by the given value.
// O(log(n))
void update(int idx, int val) {
    while (idx <= n) {
        BIT[idx] += val;
        idx += idx & -idx;
    }
}

// Returns the sum of values from 1 to the given index.
// O(log(n))
int get(int idx) {
    int res = 0;
    while (idx > 0) {
        res += BIT[idx];
        idx -= idx & -idx;
    }
    return res;
}
```

Segment Tree

- Segment tree solves query range problems in O(n.log(n)).
- Determining what goes into the segment tree node is essential.
- Sometimes we don't need a build function if the data is not ready
- The build and update functions are usually identical.
- If we are using a special struct for the node, then the getAns function should also return that node struct.
- If the update is being done on a range of elements, then we need to use lazy propagation.

```
ll seg[4 * N], lazy[4*N];
void prop(int idx, int l, int r)
{
    if (lazy[idx] == 0) return;
    int mid = (r + l) / 2;
    seg[2 * idx] += (mid - l) * 1LL * lazy[idx];
    seg[2 * idx + 1] += (r - mid) * 1LL * lazy[idx];
    lazy[2 * idx] += lazy[idx];
    lazy[2 * idx + 1] += lazy[idx];
    lazy[idx] = 0;
}

void update(int x, int y, int l, int r, int idx, int v)
{
    prop(idx, l, r);
    if (l >= y || r <= x) return;
    if (l >= x && r <= y) {
        seg[idx] += (r - l)*1LL*v;
        lazy[idx] += v;
        return;
    }
    int mid = (r + l) / 2;
    update(x, y, l, mid, 2 * idx, v);
    update(x, y, mid, r, 2 * idx + 1, v);
    seg[idx] = seg[2 * idx] + seg[2 * idx + 1];
}

ll getAns(int x, int y, int l, int r, int idx)
{
    prop(idx, l, r);
    if (l >= y || r <= x) return 0;
    if (l >= x && r <= y) return seg[idx];
    int mid = (r + l) / 2;
    return getAns(x, y, l, mid, 2 * idx) + getAns(x, y, mid, r, 2 * idx + 1);
}
```

Mo's Algorithm (SQRT decomposition)

- The queries has to be independent (can be solved offline).
- Every problem just change the insert and remove functions.
- This example find the count of distinct numbers in range [l,r].
- Complexity: O((N+Q).sqrt(N)).

```
int a[N], cnt[N], ans[N];
int curL, curR, curAns, blockSize;

// Mo's query struct
struct query {
    int l, r, i;

    bool operator<(const query& rhs) const {
        if (l / blockSize != rhs.l / blockSize) {
            return l < rhs.l;
        }

        return r < rhs.r;
    }
} queries[Q];

// Inserts the given index into our current answer
void insert(int k) {
    curAns += (++cnt[a[k]] == 1);
}

// Removes the given index from our current answer
void remove(int k) {
    curAns -= (--cnt[a[k]] == 0);
}

// Solves all queries according to Mo's algorithm.
void solveMO() {
    // Set Mo's algorithms variables
    blockSize = sqrt(n) + 1;
    curL = 0, curR = -1, curAns = 0;

    // Sort queries
    sort(queries, queries + m);

    // Solve each query and save its answer
    for (int i = 0; i < m; ++i) {
        query& q = queries[i];

        while (curL < q.l) remove(curL++);
        while (curL > q.l) insert(--curL);
        while (curR < q.r) insert(++curR);
        while (curR > q.r) remove(curR--);

        ans[q.i] = curAns;
    }
}
```

Graphs

DFS (Edge Classification)

```
vector<int> start, finish;
bool anyCycle = 0;
int timer = 0;
void dfs_EdgeClassification(int node)
{
    start[node] = timer++;
    lop(i, adj[node]){
        int child = adj[node][i];
        if (start[child] == -1)    // Not visited Before. Treed Edge
            dfs_EdgeClassification(child);
        else {
            if(finish[child] == -1)// then this is ancestor that called us and waiting us to
             finish. Then Cycle. Back Edge
                anyCycle = 1;
            else if(start[node] < start[child])  ; //you are my descendant (Forward Edge)
            else   ;    // Cross Edge
        }
    }
    finish[node] = timer++;
}
```

Floyd Applications (cont.)

```
bool isNegativeCycle() {
    lop(i, n)
        if(adj[i][i] < 0)
            return true;
    return false;
}

bool anyEffectiveCycle(int src, int dest){
    lop(i, n)
        if(adj[i][i] < 0 && adj[src][i] < OO && adj[i][dest] < OO)
            return true;
    return false;    // there is a finite path although cycles if any
}

int graphDiameter(){    // longest path among all shortest ones
    floyd();
    int mx = 0;
    lop(i, n) lop(j, n) if(adj[i][j] < OO)
        mx = max(mx, adj[i][j]);
    return mx;
}

vector< vector<int> > SCC(){
    vector<int> comp(n, -1);
    int cnt = 0;
    lop(i, n) if(comp[i] == -1) {
        comp[i] = cnt++;
        lop(j, n) if(adj[i][j] < OO && adj[j][i] < OO)    // transitive closure is enoug
            comp[j] = comp[i];
    }
    vector< vector<int> > compGraph(cnt, vector<int>(cnt));
    lop(i, n) lop(j, n)    if(adj[i][j] < OO)
        compGraph[ comp[i] ][ comp[j] ] = 1;    // DAG
    return compGraph;
}
```

Floyd

```
void floyd(){
    //OO is greater than max path (nodes * edges).
    lop(k, n) lop(i, n) lop(j, n)
        if(adj[i][k] < OO && adj[k][j] < OO) //remove this line IFF 2*OO fit in the data type.
            adj[i][j] = min(adj[i][j], adj[i][k]+adj[k][j]);
}

//Define path arr, initialize it to -1, which means direct
//If path (i, j) has intermediate node k, then path[i][j] = k; means path from i to j pass with k
void build_path(int src, int dest)
{
    if( path[src][dest] == -1 )    //So this is the last way
    {
        // print
        return;
    }
    build_path( src, path[src][dest]);
    build_path( path[src][dest], dest);
}

// Other way is through previous of path: prev[i][j]: last node before j from i to j
// Initialize prev[i][j] = i
// If path (i, j) has intermediate node k, then prev[i][j] = prev[k][j];
void printPath(int i, int j){
    if (i != j)
        printPath(i, prev[i][j]);
    printf(" %d", j);
}
```

Floyd Applications

```
void TransitiveClosure(){
    // assume matrix is 0 for disconnect, 1 is connect
    // we only care if a path exist or not, not a shortest path value
    lop(k, n) lop(i, n) lop(j, n)
        adj[i][j] |= (adj[i][k] & adj[k][j]);
}

void minimax(){
    // find path such that max value on road is minimum
    lop(k, n) lop(i, n) lop(j, n)
        adj[i][j] = min(adj[i][j], max(adj[i][k], adj[k][j]) );
}

void maximin(){
    // find path such that min value on road is maximum
    lop(k, n) lop(i, n) lop(j, n)
        adj[i][j] = max(adj[i][j], min(adj[i][k], adj[k][j]) );
}

void longestPathDAG(){
    lop(k, n) lop(i, n) lop(j, n)
        adj[i][j] = max(adj[i][j], max(adj[i][k], adj[k][j]) );
}

void countPaths(){
    lop(k, n) lop(i, n) lop(j, n)
        adj[i][j] += adj[i][k] * adj[k][j];
}
```

Dijkstra	BFS (Get the path)

Dijkstra

```
struct edge{
    int from, to, w;
    edge(int from, int to, int w): from(from), to(to), w(w) {}
    bool operator < (const edge & e) const {
        return w > e.w;
    }
};

int Dijkstra(vector< vector< edge > > adjList, int src, int dest = -1){  // O(E logV)
    int n = sz(adjList);
    vi dist(n, OO), prev(n, -1);
    dist[src] = 0;
    priority_queue<edge> q;
    q.push( edge(-1, src, 0) );
    while( !q.empty() ) {
        edge e = q.top();    q.pop();
        if(e.w > dist[e.to])    continue;    // some other state reached better
        prev[ e.to ] = e.from;
        lop(j, adjList[e.to]) {
            edge ne = adjList[e.to][j];
            if( dist[ne.to] > dist[ne.from] + ne.w ) {
                ne.w = dist[ne.to] = dist[ne.from] + ne.w, q.push( ne );
            }
        }
    }
    return dest == -1 ? -1 : dist[dest];
}
```

BFS (Get the path)

```
vector<int> BFSPath(int s, int d, vector<vector<int> > & adjList) {
    vector<int> len(sz(adjList), OO);
    vector<int> par(sz(adjList), -1);
    queue<int> q;
    q.push(s), len[s] = 0;
    int dep = 0, cur = s, sz = 1;
    bool ok = true;
    for ( ; ok && !q.empty(); ++dep, sz = q.size()) {
        while (ok && sz--) {
            cur = q.front(), q.pop();
            lop(i, adjList[cur]) if (len[adjList[cur][i]] == OO){
                q.push(adjList[cur][i]);
                len[adjList[cur][i]] = dep+1, par[ adjList[cur][i] ] = cur;
                if(adjList[cur][i] == d){   // we found target no need to continue
                    ok = false;
                    break;
                }
            }
        }
    }
    vector<int> path;
    while(d != -1) path.push_back(d), d = par[d];
    reverse( all(path) );

    return path;
}
```

Bellman-Ford

```cpp
vi buildPath(vi prev, int src) {
    vi path;    // make sure to test case self edge. E.g. 2 --> 2
    for (int i = src; i > -1 && sz(path) <= sz(prev); i = prev[i])
        path.push_back(i);
    reverse( all(path) );
    return path;
}

bool BellmanPrcoessing(vector<edge> & edgeList, int n, vi &dist, vi &prev, vi &pos) {
    if(sz(edgeList) == 0)    return false;
    for (int it = 0, r = 0; it < n+1; ++it, r = 0) {
        for (int j = 0; j < sz(edgeList) ; ++j) {
            edge ne = edgeList[j];
            if(dist[ne.from] >= OO || ne.w >= OO)    continue;
            if( dist[ne.to] > dist[ne.from] + ne.w ) {
                dist[ne.to] = dist[ne.from] + ne.w;
                prev[ ne.to ] = ne.from, pos[ ne.to ] = j, r++;
                if(it == n)        return true;
            }
        }
        if(!r)    break;
    }
    return false;
}

pair<int, bool> BellmanFord(vector<edge> & edgeList, int n, int src, int dest){    // O(NE)
    vi dist(n, OO), prev(n, -1), reachCycle(n), path, pos(n);
    // To use pos: edgeList[pos[path[i]]].w
    dist[src] = 0;
    bool cycle = BellmanPrcoessing(edgeList, n, dist, prev, pos);
    if(cycle) {
        vi odist = dist;
        BellmanPrcoessing(edgeList, n, dist, prev, pos);
        for (int i = 0; i < n; ++i)    // find all nodes that AFFECTED by negative cycle
            reachCycle[i] = (odist[i] != dist[i]);
    } else
        path = buildPath(prev, dest);

    return make_pair(dist[dest], cycle);
}
```

BFS
(Length of shortest path from s to all other nodes)

```cpp
vector<int> BFS(int s, vector<vector<int> > & adjList) {
    vector<int> len(sz(adjList), OO);
    queue< pair<int, int> > q;
    q.push(MP(s, 0)), len[s] = 0;
    int cur, dep;
    while(!q.empty()) {
        pair<int, int> p = q.front();    q.pop();
        cur = p.first, dep = p.second;
        lop(i, adjList[cur]) if (len[adjList[cur][i]] == OO)
            q.push(MP(adjList[cur][i], dep+1)), len[adjList[cur][i]] = dep+1;
    }
    return len;    //cur is the furthest node from s with depth dep
}
```

Kruskal's Algorithm

```cpp
struct edge {
    int from, to, weight;

    edge() {}
    edge(int f, int t, int w) : from(f), to(t), weight(w) {}

    bool operator<(const edge& rhs) const {
        return (weight < rhs.weight);
    }
};
int n, m;//number of nodes,number of edges
int par[N];
vector<edge> edges;
// Returns the total weight of the minimum spanning tree of the given weighted graph.
// O(n.log(n))
int kruskalMST() {
    // Initializes an n-sets.
    for (int i = 1; i <= n; ++i)
        par[i] = i;
    sort(edges.begin(), edges.end());

    int MST = 0;
    for (auto& e : edges) {
        if (unionSets(e.from, e.to))
            MST += e.weight;
    }
    return MST; }
```

Topological sort (BFS)

```cpp
void topologicalSort()
{
    // Create a vector to store indegrees of all
    vector<int> in_degree(V, 0);
    for (int i=0;i<V;i++)
    {
        for(int j=0;j<sz(adj[i]);j++) indegree[adj[i][j]]++;
    }
    // Create an queue and enqueue all vertices with indegree 0
    queue<int> q;
    for (int i = 0; i < V; i++) if (in_degree[i] == 0) q.push(i);
    int cnt = 0; vector <int> top_order;
    while (!q.empty())
    {
        int u = q.front(); q.pop();
        top_order.push_back(u);
        for (int i=0;i<sz(adj[u]);i++)
            if (--in_degree[adj[u][i]] == 0)
                q.push(adj[u][i]);
        cnt++;
    }
    if (cnt != V) cout << "There exists a cycle in the graph\n";
}
```

Find tree diameter	Detect a cycle using colors

<pre>int dist = 0, nd = 0;
void dfs(int node, int len, int col)
{
    if (len > dist) dist = len, nd = node;
    vis[node] = col;
    for (int i = 0; i < sz(adjlist[node]); i++)
        if (vis[adjlist[node][i]] != col)
            dfs(adjlist[node][i], len + 1, col);
}
 	
dist = -1;
dfs(start, 0, 1);
dist = -1;
dfs(nd, 0, 2);</pre>
	<pre>int cycle = 0;
	void dfs(int u)
	{
	    if (c) return;
	    vis[u] = 1;
	    for (int i = 0; i < sz(adj[x]); i++) {
	        int v = adj[x][i];
	        if (vis[v] == 1) { c = 1; return; }
	        if (vis[v] == 0) dfs(v);
	    }
	    vis[u] = 2;
	}</pre>

Get the edges of any (one) cycle in the graph

```
int f, vis[N]; stack<int>st; vi cycle;
//0: not visited, 1: being processed, 2: done
void getCycle(int u)
{
    if (f) return;
    vis[u] = 1;
    st.push(u);
    for (int i = 0; i < sz(adj[u]); i++) {
        int v = adj[u][i];
        if (f) return;
        if (vis[v] == 1) {
            f = 1; int b = u;
            cycle.PB(v);
            while (b != v) {
                b = st.top();
                st.pop();
                cycle.PB(b);
            }
            reverse(all(cycle));
            return;
        }
        else if (vis[v] == 0) getCycle(v);
    }
    vis[u] = 2;
    if(!st.empty()) st.pop();
}
```

Find the length of the longest path in a weighted graph

```
ll dfs(int u, int par)
{
    ll ret = 0;
    for (int i = 0; i < sz(adjlist[u]); i++)
        if (adjlist[u][i].F != par)
            ret = max(ret, dfs(adjlist[u][i].F, u) + adjlist[u][i].S);
    return ret;
}
```

Dynamic Programming

Longest Common Subsequence O(N^2)

```
int LCS(int i, int j)
{
    if (i == sz(vec1) || j == sz(vec2)) return 0;
    if (dp[i][j] != -1) return dp[i][j];
    if (vec1[i] == vec2[j])
        return dp[i][j] = LCS(i + 1, j + 1) + 1;
    return dp[i][j] = max(LCS(i, j + 1), LCS(i + 1, j));
}
```

Longest Increasing Subsequence O(N^2)

```
void LIS()
{
    for (int i = 0; i < 1100; i++) dp[i] = 1;
    for (int i = 1; i < sz(vec); i++) {
        for (int j = 0; j < i; j++) {
            if (vec[i].F.S > vec[j].F.S && dp[i] < dp[j] + 1)
                dp[i] = dp[j] + 1;
        }
    }
    int mx = 0, idx = 0;
    for (int i = 0; i < 1100; i++) {
        if (dp[i] > mx) {
            mx = dp[i];
            idx = i;
        }
    }
    printf("%d\n", mx); vi path;
    for (int i = idx; i >= 0; i--) {
        if (dp[i] == mx){
            path.PB(vec[i].S);
            mx--;
        }
    }
    for (int i = sz(path) - 1; i >= 0; i--) printf("%d\n", path[i]);
}
```

Longest Increasing Subsequence O(n.log(n))

```
int n, a[N];
int getLIS() {
    int len = 0;
    vector<int> LIS(n, INT_MAX);
    for (int i = 0; i < n; ++i) {
        // To get the length of the longest non decreasing subsequence
        // replace function "lower_bound" with "upper_bound"
        int idx = lower_bound(LIS.begin(), LIS.end(), a[i]) - LIS.begin();
        LIS[idx] = a[i];
        len = max(len, idx);
    }
    return len + 1;
}
```

Geometry:

- Area of a **sector** (like pizza slice) with angle *TH* (in radians) in circle with radius *r* = (TH / 2) * r * r.
- Area of **segment** with angle *TH* (in radians) in circle with radius *r* = ((TH - sin(TH)) / 2) * r * r.
- **Arc** length (of a **sector** or **segment** with angle *TH* in circle with radius *r*) = TH * r.
- To find the intersection point between two lines (A1x + B1y = C1, A2x + B2y = C2):
    - Get the **det**. of matrix formed by those two lines : det = A1 * B2 - A2 * B1.
    - If det == 0: no intersection. (Two lines are parallel)
    - Else: intersection point = ((B2*C1 - B1*C2) / det, (A2*C1 - A1*C2) / det).
- Sum of interior angles in any **n-sides polygon** = (n-2) * 180.
- Maximum cuts in a circle = 0.5(n^2 + n + 2) where n is the number of line cuts.
- **Compare Double Numbers**: int dcmp(ld d1,ld d2){ return fabs(d1-d2)<=EPS?0:d1>d2?1:-1; }
- **Regular Polygon** formulas:
    - n = number of sides, s = length of a side, r (apothem or radius of inscribed circle) = 0.5*s*cot(180/n), R (radius of circumcircle) = 0.5*s*csc(180/n)
    - Area = 0.5*n*s*r = 0.25*n*s*s*cot(180/n) = n*r*r*tan(180/n) = 0.5*n*R*R*sin(360/n).
- **Shoelace formula:**
    - Notes: i= [1,n] , x_(n+1) = x_1 , x_0 = x_n, if clockwise => you need absolute value.
    - Area of **polygon** = 0.5*SUM(x_i*(y_(i+1) - y_(i-1))) = 0.5*SUM(y_i*(x_(i+1) - x_(i-1))) = 0.5*SUM(x_i * y_(i+1) - x_(i+1) * y_i) = 0.5*SUM((x_(i+1) + x_i) * (y_(i+1) - y_i)).

- A **lattice point** in the plane is any point that has integer coordinates.
- **Pick's Theorem:** Let **P** be a polygon in the plane whose vertices have integer coordinates. Then the area of **P** can be determined just by counting the lattice points on the interior and boundary of the polygon!
  So the area is given by Area(**P**) = **i** + (**B** / 2) - 1.
  Where **i** is the number of interior lattice points, and **B** is the number of boundary lattice points.
- Given one solution of the **Diophantine Equation [ax + by = g]** (x0,y0**),** all possible solutions can be obtained with this relation: x = x0 + k*(b/g), y = y0 - k*(a/g). For k = 1,2,3,....

Finding the discrete root (Given a prime **n** and **a,k** , find all **x** for which **pow(x,k) ≡ a (mod n)**.)

```
int main() {

    int n, k, a;
    cin >> n >> k >> a;
    if (a == 0) {
        puts ("1\n0");
        return 0;
    }

    int g = generator (n);

    int sq = (int) sqrt (n + .0) + 1;
    vector < pair<int,int> > dec (sq);
    for (int i=1; i<=sq; ++i)
        dec[i-1] = make_pair (powmod (g, int (i * sq * 1ll * k % (n - 1)), n), i);
    sort (dec.begin(), dec.end());
    int any_ans = -1;
    for (int i=0; i<sq; ++i) {
        int my = int (powmod (g, int (i * 1ll * k % (n - 1)), n) * 1ll * a % n);
        vector < pair<int,int> >::iterator it =
            lower_bound (dec.begin(), dec.end(), make_pair (my, 0));
        if (it != dec.end() && it->first == my) {
            any_ans = it->second * sq - i;
            break;
        }
    }
    if (any_ans == -1) {
        puts ("0");
        return 0;
    }

    int delta = (n-1) / gcd (k, n-1);
    vector<int> ans;
    for (int cur=any_ans%delta; cur<n-1; cur+=delta)
        ans.push_back (powmod (g, cur, n));
    sort (ans.begin(), ans.end());
    printf ("%d\n", ans.size());
    for (size_t i=0; i<ans.size(); ++i)
        printf ("%d ", ans[i]);

}
```

Finding the no. of solutions in a given interval

```
void shift_solution (int & x, int & y, int a, int b, int cnt) {
    x += cnt * b;
    y -= cnt * a;
}

int find_all_solutions (int a, int b, int c, int minx, int maxx, int miny, int maxy) {
    int x, y, g;
    if (! find_any_solution (a, b, c, x, y, g))
        return 0;
    a /= g;  b /= g;

    int sign_a = a>0 ? +1 : -1;
    int sign_b = b>0 ? +1 : -1;

    shift_solution (x, y, a, b, (minx - x) / b);
    if (x < minx)
        shift_solution (x, y, a, b, sign_b);
    if (x > maxx)
        return 0;
    int lx1 = x;

    shift_solution (x, y, a, b, (maxx - x) / b);
    if (x > maxx)
        shift_solution (x, y, a, b, -sign_b);
    int rx1 = x;

    shift_solution (x, y, a, b, - (miny - y) / a);
    if (y < miny)
        shift_solution (x, y, a, b, -sign_a);
    if (y > maxy)
        return 0;
    int lx2 = x;

    shift_solution (x, y, a, b, - (maxy - y) / a);
    if (y > maxy)
        shift_solution (x, y, a, b, sign_a);
    int rx2 = x;

    if (lx2 > rx2)
        swap (lx2, rx2);
    int lx = max (lx1, lx2);
    int rx = min (rx1, rx2);

    if (lx > rx) return 0;
    return (rx - lx) / abs(b) + 1;
}
```

Finding modular inverse for every number modulo m (m is prime)

```
inv[1] = 1;
for(int i = 2; i < m; ++i)
    inv[i] = (m - (m/i) * inv[m%i] % m) % m;
```

For **Discrete Algorithm**:

- Instead of **map**, we can also use hash table **unordered_map** which has complexity O(1) for inserting and searching. And when the value of **m** is small enough, we can also get rid of **map**, and use a regular array to store and lookup values of **f1 (i.e. vals).**
- If we need to return all possible solutions, we need to change **map<int,int>** to, say, **map<int, vector<int> >**.

Primitive Root (p is prime,else cal its **phi)**
* **g** is a **primitive_root_modulo_n** if and only if for any integer **a** such that **gcd(a,n)=1**, there exists an integer **k** such that: **pow(g,k) ≡ a (mod n)**.

```
int generator (int p) {
    vector<int> fact;
    int phi = p-1,   n = phi;
    for (int i=2; i*i<=n; ++i)
        if (n % i == 0) {
            fact.push_back (i);
            while (n % i == 0)
                n /= i;
        }
    if (n > 1)
        fact.push_back (n);

    for (int res=2; res<=p; ++res) {
        bool ok = true;
        for (size_t i=0; i<fact.size() && ok; ++i)
            ok &= powmod (res, phi / fact[i], p) != 1;
        if (ok)  return res;
    }
    return -1;
}
```

Gray Code

```cpp
int g (int n) {
    return n ^ (n >> 1);
}
```

Inverse Gray Code

```cpp
int rev_g (int g) {
  int n = 0;
  for (; g; g >>= 1)
    n ^= g;
  return n;
}
```

Iterating over all submasks of a given mask s

```cpp
for (int s=m; ; s=(s-1)&m) {
 ... you can use s ...
 if (s==0)  break;
}
```

Calculate (a*b)%m where a,b are big integers

```cpp
uint64_t mul_mod(uint64_t a, uint64_t b, uint64_t m)
{
  long double x;
  uint64_t c;
  int64_t r;
  if (a >= m) a %= m;
  if (b >= m) b %= m;
  x = a;
  c = x * b / m;
  r = (int64_t)(a * b - c * m) % (int64_t)m;
  return r < 0 ? r + m : r;
}
```

Discrete Logarithm

(find **x** such that **pow(a,x) ≡ b (mod m),** where **a,m** are relatively prime) **O(sqrt(m) * log(m))**

```cpp
int solve (int a, int b, int m) {
    int n = (int) sqrt (m + .0) + 1;

    int an = 1;
    for (int i=0; i<n; ++i)
        an = (an * a) % m;

    map<int,int> vals;
    for (int i=1, cur=an; i<=n; ++i) {
        if (!vals.count(cur))
            vals[cur] = i;
        cur = (cur * an) % m;
    }

    for (int i=0, cur=b; i<=n; ++i) {
        if (vals.count(cur)) {
            int ans = vals[cur] * n - i;
            if (ans < m)
                return ans;
        }
        cur = (cur * a) % m;
    }
    return -1;
}
```

BigInteger operations (Addition, Subtraction and Division by short integer)

```cpp
int carry = 0;
for (size_t i=0; i<max(a.size(),b.size()) || carry; ++i) {
    if (i == a.size())
        a.push_back (0);
    a[i] += carry + (i < b.size() ? b[i] : 0);
    carry = a[i] >= base;
    if (carry)  a[i] -= base;
}
```

```cpp
int carry = 0;
for (size_t i=0; i<b.size() || carry; ++i) {
    a[i] -= carry + (i < b.size() ? b[i] : 0);
    carry = a[i] < 0;
    if (carry)  a[i] += base;
}
while (a.size() > 1 && a.back() == 0)
    a.pop_back();
```

```cpp
int carry = 0;
for (int i=(int)a.size()-1; i>=0; --i) {
    long long cur = a[i] + carry * 1ll * base;
    a[i] = int (cur / b);
    carry = int (cur % b);
}
while (a.size() > 1 && a.back() == 0)
    a.pop_back();
```

BigInteger operations (Multiplication by short integer and big integer)

```cpp
int carry = 0;
for (size_t i=0; i<a.size() || carry; ++i) {
    if (i == a.size())
        a.push_back (0);
    long long cur = carry + a[i] * 1ll * b;
    a[i] = int (cur % base);
    carry = int (cur / base);
}
while (a.size() > 1 && a.back() == 0)
    a.pop_back();
```

```cpp
lnum c (a.size()+b.size());
for (size_t i=0; i<a.size(); ++i)
    for (int j=0, carry=0; j<(int)b.size() || carry; ++j) {
        long long cur = c[i+j] + a[i] * 1ll * (j < (int)b.size() ? b[j] : 0) + carry;
        c[i+j] = int (cur % base);
        carry = int (cur / base);
    }
while (c.size() > 1 && c.back() == 0)
    c.pop_back();
```

* **lnum** is typedef for **vector<int>.**

Factorial modulo p [O(p*log_p(n))]

```
int factmod(int n, int p) {
    int res = 1;
    while (n > 1) {
        res = (res * ((n/p) % 2 ?  p-1 : 1)) % p;
        for (int i = 2; i <= n%p; ++i)
            res = (res * i) % p;
        n /= p;
    }
    return res % p;
}
```

Finding power of factorial divisor

```
int fact_pow (int n, int k) {
    int res = 0;
    while (n) {
        n /= k;
        res += n;
    }
    return res;
}
```

If composite k, factorize it to pow(k1,p1)*..*pow(km,pm).

then get all a_i such that a_i = fact_pow(n,ki).

The answer is min(a_i / p_i) for i = 1,2,..,m

- (a^b)%m = ((a%m) ^ (b%(m-1))%m [only if m is prime]
- Count of set bits in **n = __builtin_popcount(n)**
- Count of trailing zeros in **n = __builtin_ctz(n)**
- Count of leading zeros in **n = __builtin_clz(n)**