Table of Contents

[ **Berlekamp Massey – Pranto** 2](#_Toc5315465)

[ **Bridges Finding** 5](#_Toc5315466)

[ **Articulation Point Finding** 5](#_Toc5315467)

[ **DSU on Tree** 6](#_Toc5315468)

[ **UnionFind** 7](#_Toc5315469)

[ **Persistant/Rollback** 8](#_Toc5315470)

# **Berlekamp Massey – Pranto**

**#include** <bits/stdc++.h>

**using** **namespace** std;

/// ======================= Berlekamp Massey - Pranto =========================

/// Typedef

**typedef** **long** **long** ll;

**typedef** **unsigned** **long** **long** ull;

**typedef** vector<ll> vll;

**#define** FOR(i, x, y) **for**(**int** i=**int**(x); i<**int**(y); i++)

**#define** all(c) c.begin(), c.end()

**template** <**class** **T**> **inline** **T** **bigMod**(**T** p,**T** e,**T** M){ **T** ret = 1; **for**(; e > 0; e >>= 1){ **if**(e & 1) ret = (ret \* p) % M; p = (p \* p) % M; } **return** (**T**) ret;}

**template** <**class** **T**> **inline** **T** **modInverse**(**T** a,**T** M){**return** bigMod(a,M-2,M);}

**#define** mType **int**

**#define** matrix vector<vector<mType>>

**int** mod;

**inline** matrix **Identity\_Matrix**(**const** **int** row, **const** **int** col){

matrix I(row,vector<mType>(col,0));

**for**(**int** i = 0; i < row; i++) I[i][i] = 1;

**return** I;

}

**inline** matrix **operator \***(**const** matrix &a, **const** matrix &b){

**int** ra = a.size(), ca = a[0].size();

**int** rb = b.size(), cb = b[0].size();

**if**(ca != rb) **return** Identity\_Matrix(ra,cb);

\_\_int128 sum;

matrix res(ra,vector<mType>(cb));

**for**(**int** i = 0; i < ra; i++){

**for**(**int** j = 0; j < cb; j++){

sum = 0;

**for**(**int** k = 0; k < ca; k++) sum += 1LL\*a[i][k]\*b[k][j];

res[i][j] = sum%mod;

}

}

**return** res;

}

matrix I;

matrix **operator ^**(**const** matrix &base, **const** **long** **long** power){

**if**(power == 0) **return** I;

**if**(power == 1) **return** base;

matrix full = base^(power>>1); // half

full = full\*full; // full = half\*half

**if**(power&1) full = full\*base; // if(power is odd)

**return** full;

}

ll **convolution**(vll \_first, vll \_second, ll modulo){

ll result = 0, len = \_first.size();

FOR(i,0,len) result = (result + \_first[i] \* \_second[i]) % modulo;

**return** result;

}

vll **berlekamp\_massey**(vll base\_sequence, ll modulo){

**int** n = base\_sequence.size();

vll U(n + 1, 0),V(n + 1, 0),W;

U[0] = V[0] = 1;

reverse(all(base\_sequence));

vll \_first,\_second;

ll l = 0, m = 1, b = 1, deg = 0;

FOR(i,0,n){

ll d = base\_sequence[n - 1 - i];

**if**(l){

\_first.assign(V.begin()+1,V.begin()+1+l);

\_second.assign(base\_sequence.begin()+n-i,base\_sequence.begin()+n-i+l);

d = (d + convolution(\_first, \_second, modulo)) % modulo;

}

**if**(!d){ m += 1; **continue**;}

**if**(l \* 2 <= i) W.assign(V.begin(), V.begin()+l+1);

ll x = (modInverse(b,modulo) \* (modulo - d) % modulo + modulo) % modulo;

FOR(j,0,deg+1) V[m + j] = (V[m + j] + x \* U[j]) % modulo;

**if**(l \* 2 <= i){

swap(W,U);

deg = U.size()-1;

b = d , m = 1, l = i - l + 1;

}

**else** m += 1;

}

vll tmp(max(0LL,l-(ll)V.size()+1), 0);

V.erase(V.begin()+l+1,V.end());

V.erase(V.begin());

V.insert(V.end(), all(tmp));

**return** V;

}

**int** **solve\_linear\_recurrence**(vll base\_sequence, ll nth\_term, ll modulo){

**int** n = base\_sequence.size();

**if**(nth\_term < n) **return** base\_sequence[nth\_term];

vll recurrence = berlekamp\_massey(base\_sequence,modulo);

// for(auto i : recurrence) cout << i << endl;

**int** l = recurrence.size();

I = Identity\_Matrix(l,l);

matrix M(l, vector<mType>(l, 0));

mod = modulo;

FOR(i,0,l){

M[0][i] = modulo - recurrence[i];

**if**(i) M[i][i - 1] = 1;

}

ll result = 0;

M = M ^ (nth\_term - n + 1);

FOR(i,0,l) result += M[0][i] \* base\_sequence[n - i - 1];

**return** result % modulo;

}

**int** **solve**(ll n, ll c, ll k, ll modulo){

ll s = 0, a = 0, b = 1;

vll fib(400,0);

fib[1] = 1;

FOR(i,2,400) fib[i] = (fib[i-1]+fib[i-2])%modulo;

vll base\_sequence(30);

FOR(i,0,30) base\_sequence[i] = bigMod(fib[i\*c],k,modulo);

FOR(i,1,30) base\_sequence[i] = (base\_sequence[i]+base\_sequence[i-1])%modulo;

**return** solve\_linear\_recurrence(base\_sequence,n,modulo);

}

**int** **main**(){

**#ifdef** VAMP

**int** start\_s=clock();

freopen("//home//pranto//Desktop//input.txt","r",stdin);

freopen("//home//pranto//Desktop//output.txt","w",stdout);

**#endif**

//fastIO;

//srand((rand()-time(NULL))%MOD);

ll n, c, k, modulo = 1e9+7;

**int** t,cs = 0; cin >> t;

**while**(t--){

cin >> n >> c >> k;

**printf**("Case %d: %d\n", ++cs, solve(n,c,k,modulo));

}

**#ifdef** VAMP

**int** end\_s=clock();

printf("\nTime :: %.2lf\n", (**double**)(end\_s-start\_s)/CLOCKS\_PER\_SEC);

**#endif**

**return** 0;

}

# **Bridges Finding**

// ============== Bridges Finding ====================

**const** **int** maxn = 1e5+5;

vector<**int**> g[maxn]; // adjacency list

**int** N; // Number of nodes in the graph

**bool** visited[maxn];

**int** Time; // current time of dfs

**int** discoverTime[maxn]; // first discovering time of node i

**int** min\_discoverTime[maxn]; // minimum first discovering Time of the nodes of the descendants of node i in the dfs tree

vector<pair<**int**,**int**>> bridge; // List of bridge Edges

**void** **dfs**(**int** node, **int** parent = -1){

visited[node] = 1;

discoverTime[node] = min\_discoverTime[node] = Time++; // all discovering Time is unique

**for**(**auto** &next : g[node]) **if**(next != parent){

**if**(visited[next]){ // This is a back Edge. If descendants of node "next" is connected to ancestor of current node, then

min\_discoverTime[node] = min(min\_discoverTime[node], discoverTime[next]); // discoverTime[next] <= discoverTime[node];

}

**else**{

dfs(next,node);

min\_discoverTime[node] = min(min\_discoverTime[node], min\_discoverTime[next]);

**if**(discoverTime[node] < min\_discoverTime[next]){ // if min\_discoverTime[node] < discoverTime[next] then definitely descendants of

bridge.emplace\_back(min(node,next),max(node,next)); // "next" is not connected to ancestor of current node in the dfs tree

}

}

}

}

**void** **Find\_Bridge**(){

Time = 0;

bridge.clear();

memset(visited, 0, **sizeof**(visited));

memset(discoverTime, -1, **sizeof**(discoverTime));

memset(min\_discoverTime, -1, **sizeof**(min\_discoverTime));

**for**(**int** i = 0; i < N; i++){

**if**(!visited[i]){

dfs(i);

}

}

}

# **Articulation Point Finding**

// ===================== Articulation Point Finding ========================

**const** **int** maxn = 1e5+5;

vector<**int**> g[maxn]; // adjacency list

**int** N; // Number of nodes in the graph

**bool** visited[maxn];

**int** Time; // current time of dfs

**int** discoverTime[maxn]; // first discovering time of node i

**int** min\_discoverTime[maxn]; // minimum first discovering Time of the nodes of the descendants of node i in the dfs tree

vector<**int**> cutPoint; // List of Articulation Points

**void** **dfs**(**int** node, **int** parent = -1){

visited[node] = 1;

discoverTime[node] = min\_discoverTime[node] = Time++; // all discovering Time is unique

**int** child = 0;

**bool** isCutPoint = **false**;

**for**(**auto** &next : g[node]) **if**(next != parent){

**if**(visited[next]){ // This is a back Edge. If descendants of node "next" is connected to ancestor of current node, then

min\_discoverTime[node] = min(min\_discoverTime[node], discoverTime[next]); // discoverTime[next] <= discoverTime[node];

}

**else**{

dfs(next,node);

min\_discoverTime[node] = min(min\_discoverTime[node], min\_discoverTime[next]);

**if**(discoverTime[node] <= min\_discoverTime[next] && ~parent){ // if min\_discoverTime[node] < discoverTime[next] then definitely descendants of

isCutPoint = **true**; // "next" is not connected to ancestor of current node in the dfs tree

}

child++;

}

}

**if**(isCutPoint || (parent == -1 && child > 1)){ // if current node is root and it has more than one child in dfs tree

cutPoint.push\_back(node); // then it is also a an articulation point

}

}

**void** **Find\_cutPoint**(){

Time = 0;

cutPoint.clear();

memset(visited, 0, **sizeof**(visited));

memset(discoverTime, -1, **sizeof**(discoverTime));

memset(min\_discoverTime, -1, **sizeof**(min\_discoverTime));

**for**(**int** i = 0; i < N; i++){

**if**(!visited[i]){

dfs(i);

}

}

**for**(**auto** p : cutPoint) cout << p << " "; cout << endl;

}

# **DSU on Tree**

**const** **int** maxn = 1e6+8; // maximum number of element

std::vector<**int**> g[maxn]; // Adjacency list

**char** ch[maxn]; // color of the ith node

**int** sz[maxn]; // size of the ith subtree

**int** bigChild[maxn]; // index if the ith nodes bigchild/heavychild

**int** lev[maxn]; // level of the ith node

**void** **getSize**(**int** v, **int** p, **int** lv){ // pre-calculates the values of sz, bigChild and lev

sz[v] = 1, bigChild[v] = -1, lev[v] = lv;

**int** mx = -1;

**for**(**auto** u : g[v])

**if**(u != p){

getSize(u,v,lv+1);

sz[v] += sz[u];

**if**(mx < sz[u]) mx = sz[u], bigChild[v] = u;

}

}

vector< pair<**int**,**int**> > qry[maxn]; // queries for the ith node

**bool** big[maxn]; // flag to if ith node is currently a bigChild or not

**bool** ans[maxn]; // // answer for the ith query

**int** cnt[26][maxn]; // cnt[i][j] = count of color i at jth level

**void** **add**(**int** v, **int** p, **int** x){ // adds x to the all nodes of the subtree v

cnt[ ch[v]-'a' ][ lev[v] ] += x;

**for**(**auto** u : g[v])

**if**(u != p && !big[u]) // don't calculate for bigChild as we kept the changes made by this child

add(u,v,x);

}

**void** **dfs**(**int** v, **int** p, **int** keep){ // calculates answer for the node v

**for**(**auto** u : g[v])// 1st calculate for all child except bigChild and don't keep the changes

**if**(u != p && u != bigChild[v]) dfs(u,v,0);

**if**(bigChild[v] != -1)

dfs(bigChild[v],v,1), big[ bigChild[v] ] = 1;// calculate for the bigChild and keep the changes

add(v,p,1); // add value 1 to all nodes of the subtree v

**for**(**auto** d : qry[v]){ // value addition/update is node, now calulate the answer for this node

**int** odd = 0, even = 0;

**for**(**int** i = 0; i < 26; i++) cnt[i][d.first]&1 ? odd++ : even++;

ans[d.second] = (odd <= 1); // store the answer

}

**if**(bigChild[v] != -1) big[ bigChild[v] ] = 0;

**if**(keep == 0) add(v,p,-1); // as keep = 0 remove the changes made by this node

}

# **UnionFind**

**class** UnionFind {

**private**:

**static** **const** **int** *maxn* = 1e5+5;

**int** n; // totall number of nodes

**int** group; // totall disconnected group;

**int** sz[*maxn*]; // size of a network

**int** par[*maxn*]; // parent of a neode

**public**:

**void** **Init**(**int** n){

**this**->n = group = n;

**for**(**int** i = 0; i < n; i++) par[i] = i, sz[i] = 1;

}

**int** **find**(**int** p){ **return** p == par[p] ? p : par[p] = find(par[p]);} // finds parent of p

**int** **connected**(**int** p, **int** q){ **return** find(p) == find(q);} // checks p and q is connected or not

**int** **groupCnt**(){ **return** group;} // gives number of disconnected group

**int** **totall**(){ **return** n;} // gives totall number of nodes in network

**int** **size**(**int** p){ **return** sz[ find(p) ];} // gives the size of the group of p

**void** **merge**(**int** p, **int** q){ // merges the group of p and q

**int** par1 = find(p);

**int** par2 = find(q);

**if**(par1 == par2) **return**;

**else** group--;

**if**(sz[par1] > sz[par2]) par[par2] = par1, sz[par1] += sz[par2];

**else** par[par1] = par2, sz[par2] += sz[par1];

}

}uf;

## **Persistant/Rollback**

**const** **int** maxn = 3e5+5, K = 3e5; // shift

**class** UnionFind {

**private**:

**int** n; // totall number of nodes

**int** szX[maxn\*2]; // size of a network

**int** szY[maxn\*2]; // size of a network

**int** par[maxn\*2]; // parent of a neode

**int** value[maxn\*80];

**int** \*address[maxn\*80]; // need for undo/rollback operation

**int** timer;

**public**:

**void** **Init**(**int** n){

timer = 0;

**this**->n = n;

**for**(**int** i = 0; i < n; i++) par[i] = i, par[i+K] = i+K, szX[i] = szY[i+K] = 1;

}

**int** **find**(**int** p){ **return** p == par[p] ? p : find(par[p]);} // finds parent of p

**void** **change**(**int** &from, **int** to){

address[timer] = &from;

value[timer] = from;

from = to;

timer++;

}

**void** **rollback**(){

timer--;

\*address[timer] = value[timer];

}

**int** **currState**(){ **return** timer;}

**void** **merge**(**int** p, **int** q){ // merges the group of p and q

**int** x = find(p);

**int** y = find(q);

**if**(x == y) **return**;

**if**(szX[x]+szY[x] < szX[y]+szY[y]) swap(x,y);

ans -= 1LL\*szX[x]\*szY[x];

ans -= 1LL\*szX[y]\*szY[y];

change(par[y], x);

change(szX[x], szX[x]+szX[y]);

change(szY[x], szY[x]+szY[y]);

ans += 1LL\*szX[x]\*szY[x];

}

}uf;

# **Extended\_Euclid**

**template**<**typename** **T**> **inline** **T** **Extended\_Euclid**(**T** a, **T** b, **T** &x, **T** &y){

**T** r2 = a, r1 = b;

**T** X[2] = {1,0};

**T** Y[2] = {0,1};

**T** r,q;

**do**{

q = r2/r1, r = r2-r1\*q;

X[0] = X[0]-X[1]\*q; swap(X[0],X[1]);

Y[0] = Y[0]-Y[1]\*q; swap(Y[0],Y[1]);

r2 = r; swap(r1,r2);

}

**while**(r);

x = X[0], y = Y[0];

**return** r2;

}

**template**<**typename** **T**> **inline** **bool** **Linear\_Diophantine**(**T** A, **T** B, **T** C, **T** &x, **T** &y){

**if**(!A && !B){ x = 1, y = 1; **return** !C;}

**if**(!A){ x = 1, y = C/B; **return** C%B==0;}

**if**(!B){ x = C/A, y = 1; **return** C%A==0;}

**T** g = \_\_gcd(A,B); **if**(C%g) **return** 0;

**T** a = A/g, b = B/g, c = C/g;

**if**(g < 0) a \*= -1, b \*= -1, c \*= -1;

Extended\_Euclid(a,b,x,y);

x \*= c, y \*= c; // (x1,y1) = (x+kb/g,y-ka/g);

**return** 1;

}

**template**<**typename** **T**> **inline** **T** **CountInRange**(**T** a, **T** b, **T** c, **T** x1, **T** x2, **T** y1, **T** y2){

**T** x,y;

**if**(a < 0) swap(x1,x2), a = -a, x1 = -x1, x2 = -x2;

**if**(b < 0) swap(y1,y2), b = -b, y1 = -y1, y2 = -y2;

**bool** have\_solution = Linear\_Diophantine(a,b,c,x,y);

**T** cnt = 0;

**if**(have\_solution){

**if**(!a && !b) cnt = (!c)\*(x2-x1+1)\*(y2-y1+1);

**else** **if**(!a) cnt = (c%b==0 && y1 <= c/b && c/b <= y2)\*(x2-x1+1);

**else** **if**(!b) cnt = (c%a==0 && x1 <= c/a && c/a <= x2)\*(y2-y1+1);

**else**

{

**T** g = \_\_gcd(a,b);

**double** ga = a/g, gb = b/g;

**double** lx = (x1-x)/gb, hx = (x2-x)/gb;

**double** ly = (y-y2)/ga, hy = (y-y1)/ga;

**T** h = min(floor(hx),floor(hy));

**T** l = max(ceil(lx),ceil(ly));

**if**(l <= h) cnt = h-l+1;

}

}

**return** cnt;

}

**template**<**typename** **T**> **inline** pair<**T**,**T**> **Chinese\_Reminder\_Theorem**(**const** vector<**T**> &m, **const** vector<**T**> &r, **bool** coprime = 0){ // coprime = 1 if mods are pariwise coprime

**int** n = m.size();

**T** r1 = r[0], m1 = m[0],m2,r2,p,q,mod,g = 1;

**for**(**int** i = 1; i < n; i++){

m2 = m[i], r2 = r[i];

**if**(!coprime){

g = \_\_gcd(m1,m2);

**if**(r1%g != r2%g) **return** {-1,-1};

}

Extended\_Euclid(m1/g,m2/g,p,q);

mod = m1/g\*m2; // LCM(m1,m2);

r1 = ((\_\_int128)r1\*(m2/g)%mod\*q%mod + (\_\_int128)r2\*(m1/g)%mod\*p%mod)%mod; // Becarefull about overflow, in case use \_\_int128

m1 = mod;

}

**return** {(r1+m1)%m1, m1};

}

# **Fenwick Tree**

**#define** MAX 100010

**struct** fenwick{

**int** n, tree[MAX];

**fenwick**(){}

**fenwick**(**int** m){

n = m;

**for** (**int** i = 1; i <= n; i++) tree[i] = 0;

}

**inline** **void** **update**(**int** pos, **int** value) {

**while** (pos <= n) tree[pos] += value, pos += pos&(-pos);

}

**inline** **int** **query**(**int** pos){

**int** res = 0;

**while** (pos) res += tree[pos], pos ^= pos&(-pos);

**return** res;

}

**inline** **int** **query**(**int** l, **int** r){

**if** (l > r) **return** 0;

**return** (query(r) - query(--l));

}

}