DATASTRUCTURES

**BIT**

**const** **int** N=**1e5**+**10**;

**int** n, BIT[N];

**void** **UpdateIndex**(**int** idx,**int** value) // O(logn) - Updates value of a certain index

{

**for**(;idx<=n;idx+=idx&-idx)

BIT[idx]+=value;

}

**int** **GetAns**(**int** idx) // O(logn) - Gets answer in range [1:idx]

{

**int** sum = **0**;

**for**(;idx;idx-=(idx&-idx))

sum+=BIT[idx];

**return** sum;

}

**SEGMENT TREE**

// Size = 4\*N as there are children to the leaves

// The array a[N] is 1-based so as to correspond to the Build function

**const** **int** N=**1e5**+**10**;

**int** n,a[N]; // Don't redefine n

**int** seg[**4**\*N],lazy[**4**\*N];

**void** **propagate**(**int** ind, **int** l, **int** r) // Propagate contains l,r for summation values

{

seg[ind\***2**+**1**]|=lazy[ind];

seg[ind\***2**+**2**]|=lazy[ind];

lazy[ind\***2**+**1**]|=lazy[ind];

lazy[ind\***2**+**2**]|=lazy[ind];

lazy[ind]=**0**;

}

**void** **Build**(**int** ind=**0**, **int** l=**1**, **int** r=n) // O(nlogn)

{

**if**(l==r)

{

seg[ind]=a[l];

**return**;

}

**int** mid=l+(r-l)/**2**;

Build(ind\***2**+**1**,l,mid);

Build(ind\***2**+**2**,mid+**1**,r);

seg[ind]=seg[ind\***2**+**1**]|seg[ind\***2**+**2**];

}

**void** **UpdateIndex**(**int** qind,**int** qv, **int** ind=**0**, **int** l=**1**, **int** r=n) // O(logn)

{

**if**(qind<l || qind>r) **return**;

**if**(l==r)

{

seg[ind]|=qv;

**return**;

}

**int** mid=l+(r-l)/**2**;

UpdateIndex(qind,qv,ind\***2**+**1**,l,mid);

UpdateIndex(qind,qv,ind\***2**+**2**,mid+**1**,r);

seg[ind]=seg[ind\***2**+**1**]|seg[ind\***2**+**2**];

}

**void** **UpdateRange**(**int** qx, **int** qy, **int** qv, **int** ind=**0**, **int** l=**1**, **int** r=n) // O(logn) with lazy propagation

{

**if**(r<qx || l>qy) **return**;

**if**(l>=qx && r<=qy)

{

seg[ind]|=qv;

lazy[ind]|=qv;

**return**;

}

**int** mid=l+(r-l)/**2**;

propagate(ind, l, r);

UpdateRange(qx,qy,qv,ind\***2**+**1**,l,mid);

UpdateRange(qx,qy,qv,ind\***2**+**2**,mid+**1**,r);

seg[ind]=seg[ind\***2**+**1**]|seg[ind\***2**+**2**];

}

**int** **GetAns**(**int** qx, **int** qy, **int** ind=**0**, **int** l=**1**, **int** r=n, **bool** isRangeProblem=**0**) // O(logn)

{

**if**(r<qx || l>qy) **return** **0**;

**if**(l>=qx && r<=qy) **return** seg[ind];

**int** mid=l+(r-l)/**2**;

**if**(isRangeProblem) propagate(ind, l, r);

**int** r1=GetAns(qx,qy,ind\***2**+**1**,l,mid,isRangeProblem);

**int** r2=GetAns(qx,qy,ind\***2**+**2**,mid+**1**,r,isRangeProblem);

**return** r1|r2;

}

**SPARSE TABLE**

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

**const** **int** logN = **20**;

**int** n, a[N], ST[logN][N], LOG[N];

**void** **Build**() // O(nlogn) - Builds the sparse table (min sparse table)

{

LOG[**0**] = -**1**;

**for**(**int** i=**0**;i<n;i++) // Computes floor(log2(i)) for all values

ST[**0**][i] = i, LOG[i+**1**] = LOG[i] + !(i & (i+**1**));

**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); // Changes according to the operation

}

}

**int** **GetAns**(**int** l, **int** r) // O(1) - Queries for the values from l to 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); // Changes according to the operation

}

**TRIE**

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

**const** **int** logN = **20**;

**int** n, a[N], ST[logN][N], LOG[N];

**void** **Build**() // O(nlogn) - Builds the sparse table (min sparse table)

{

LOG[**0**] = -**1**;

**for**(**int** i=**0**;i<n;i++) // Computes floor(log2(i)) for all values

ST[**0**][i] = i, LOG[i+**1**] = LOG[i] + !(i & (i+**1**));

**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); // Changes according to the operation

}

}

**int** **GetAns**(**int** l, **int** r) // O(1) - Queries for the values from l to 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); // Changes according to the operation

}

DYNAMIC PROGRAMMING

**LCS**

vector<**int**> vec1,vec2;

**int** dp[**5005**][**5005**];

**int** **LCS**(**int** i, **int** j)

{

**if**(i == vec1.size() || j == vec2.size()) **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));

}

**LIS O(N^2)**

**const** **int** N = **1e5**+**10**;

**int** dp[N];

vector<**int**> a,path;

**int** **LIS**()

{

**for**(**int** i=**0**;i<N;i++) dp[i] = **1**;

**for**(**int** i=**1**;i<a.size();i++)

**for**(**int** j=**0**;j<i;j++)

**if**(a[i] > a[j] && dp[i] < dp[j]+**1**)

dp[i] = dp[j] + **1**;

**int** ans = **0**, mx = **0**, idx = **0**;

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

**if**(dp[i] > mx)

mx = dp[i],idx=i;

ans = mx;

**for**(**int** i=idx;i>=**0**;i--)

**if**(dp[i] == mx)

path.pb(a[i]),mx--;

reverse(path.begin(),path.end());

**return** ans;

}

**LIS O(NlogN)**

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

**int** n,a[N];

**int** **getLIS**()

{

**if**(n<**1**) **return** **0**;

**int** len = **0**;

vector<**int**> LIS(n,oo);

**for**(**int** i=**0**;i<n;i++)

{

**int** idx = lower\_bound(LIS.begin(),LIS.end(),a[i]) - LIS.begin();

LIS[idx] = a[i];

len = max(len,idx);

}

**return** len+**1**;

}

GRAPHS

**DSU**

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

**int** par[N];

**int** num[N];

**for**(**int** i=**0**;i<N;i++) // Initialization in main

{

par[i] = i;

num[i] = **1**;

}

**int** FindSet(**int** n) // Use FindSet instead of par

{

**if**(par[n] == n)

**return** n;

**return** par[n] = FindSet(par[n]);

}

**void** Union(**int** x,**int** y)

{

x = FindSet(x);

y = FindSet(y);

**if**(x==y)

**return**;

par[x]=y;

num[y] += num[x]; // num holds number of elements in a group

}

**FLOYD**

// Shortest path between all pairs of nodes, negative edges allowed, no negative cycles

**int** n; // Number of nodes

**int** adj[n][n]; // adj[u][v] = cost of edge from u to v

**void** **init**()

{

**for**(**int** i=**0**;i<n;i++)

**for**(**int** j=**0**;j<n;j++)

adj[i][j] = (i==j? **0** : oo); // Sets adj[u][u] = 0, adj[u][u] = oo if no edge exists

}

**void** **floyd**()

{

**for**(**int** i=**0**;i<n;i++)

**for**(**int** j=**0**;j<n;j++)

adj[i][j] = oo;

**for**(**int** k=**0**;k<n;k++)

**for**(**int** i=**0**;i<n;i++)

**for**(**int** j=**0**;j<n;j++)

**if**(adj[i][k] + adj[k][j] < adj[i][j])

adj[i][j] = adj[i][k] + adj[k][j];

}

**bool** **checkNegativeCycle**()

{

**bool** ret = false;

**for**(**int** i=**0**;i<n;i++)

ret = ret || (adj[i][i] < **0**);

**return** ret;

}

**IS BIPARTITE**

// Shortest path between all pairs of nodes, negative edges allowed, no negative cycles

**int** n; // Number of nodes

**int** adj[n][n]; // adj[u][v] = cost of edge from u to v

**void** **init**()

{

**for**(**int** i=**0**;i<n;i++)

**for**(**int** j=**0**;j<n;j++)

adj[i][j] = (i==j? **0** : oo); // Sets adj[u][u] = 0, adj[u][u] = oo if no edge exists

}

**void** **floyd**()

{

**for**(**int** i=**0**;i<n;i++)

**for**(**int** j=**0**;j<n;j++)

adj[i][j] = oo;

**for**(**int** k=**0**;k<n;k++)

**for**(**int** i=**0**;i<n;i++)

**for**(**int** j=**0**;j<n;j++)

**if**(adj[i][k] + adj[k][j] < adj[i][j])

adj[i][j] = adj[i][k] + adj[k][j];

}

**bool** **checkNegativeCycle**()

{

**bool** ret = false;

**for**(**int** i=**0**;i<n;i++)

ret = ret || (adj[i][i] < **0**);

**return** ret;

}

**KRUSKALL MST**

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

vector< pair< **int**, pair<**int**,**int**> > edges; // {cost, {u,v}}

**int** par[N];

**int** num[N];

**for**(**int** i=**0**;i<N;i++) // Initialization in main

{

par[i] = i;

num[i] = **1**;

}

**int** FindSet(**int** n) // Use FindSet instead of par

{

**if**(par[n] == n)

**return** n;

**return** par[n] = find\_set(par[n]);

}

**void** Union(**int** x,**int** y)

{

x = find\_set(x);

y = find\_set(y);

**if**(x==y)

**return**;

par[x]=y;

num[y] += num[x]; // num holds number of elements in a group

}

**int** getMST() // Returns summation of edges of Minimum Spanning Tree and builds the tree in DSU

{

**int** sum = **0**;

sort(edges.begin(),edges.end());

//reverse(edges.begin(), edges().end()); // Uncomment for Max. spanning tree

**for**(**auto** e : edges)

{

**int** u = e.s.f; **int** v = e.s.s;

**if**(FindSet(u) == FindSet(v))

**continue**;

Union(u,v);

sum+=e.f;

}

**return** sum;

}

**LCA O(1)**

**int** **const** N = **1e5**+**10**;

**int** **const** nN = **2**\*N;

**int** id,ID[N],rID[N],First[N],ST[nN][logN],lg[nN];

vector<**int**>vec;

**void** **dfs**(**int** cur,**int** par)

{

ID[cur] = id++; // Creating ur own ID by euler tour.

rID[ID[cur]] = cur; // reverseID to get back to the original node name.

vec.push\_back(ID[cur]); //Push ur tour.

**for**(**auto** x:adj[cur])

**if**(x!=par)

dfs(x,cur),vec.push\_back(ID[cur]);

}

**inline** **void** **Build**()

{

**int** cnt = **0**,two = **1**;

**while**(two<nN) lg[two] = cnt++,two\*=**2**;

**for**(**int** i=**3**;i<nN;++i) **if**(!lg[i]) lg[i] = lg[i-**1**]; //Create an array to get the log of any number [nearest log].

//the previous part should be done only one time! Put it before the test cases.

**int** nN = sz(vec);

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

**for**(**int** i=**0**;i<nN;++i) // Fill the "First" array. remeber this array contains the idx of the first appearence of an id.

ST[i][**0**] = vec[i],(First[vec[i]]==-**1**? First[vec[i]]=i:**0**);

// Bulding the sparse table.

**for**(**int** i=**1**;(**1**<<i) <= nN;++i)

**for**(**int** j=**0**;j+(**1**<<i)-**1**<nN;++j)

ST[j][i] = min(ST[j][i-**1**],ST[j+(**1**<<(i-**1**))][i-**1**]);

}

**inline** **int** **LCA**(**int** a,**int** b)

{

**int** L,R,idx,d,lca;

a = ID[a]; b = ID[b]; // Go to ur domain.

L = First[a], R =First[b];

**if**(L>R) swap(L,R);

d = R-L+**1**;

d = lg[d];

idx = R - (**1**<<d) + **1**;

lca = min(ST[L][d],ST[idx][d]);

lca = rID[lca];

a = rID[a]; b = rID[b]; //Back to main domain.

**return** lca;

}

**LCA O(logN)**

**const** **int** N = **1e5**+**10**;

**const** **int** logN = **20**;

**int** ST[logN][N],depth[N];

**void** **dfs**(**int** cur, **int** par)

{

ST[**0**][cur] = par;

**for**(**int** i=**1**;i<logN;++i)

ST[i][cur] = ST[i-**1**][ST[i-**1**][cur]];

**for** (**auto** x:adj[cur])

**if**(x != par)

depth[x]=depth[cur]+**1**,dfs(x, cur);

}

**int** **KthAncestor**(**int** u, **int** k)

{

**if**(!k) **return** u;

**int** d = depth[u] - k; // Get the depth of the wanted node.

**for** (**int** j = logN-**1**; j >= **0**; --j)

{

**int** nu = ST[j][u];

**if** (!nu) **continue**;

**if** (depth[nu] == d) **return** nu;

**else** **if** (depth[nu] > d && u) u = nu;

}

**return** **0**;

}

**int** **LCA**(**int** a, **int** b)

{

**if** (depth[a] > depth[b]) a = KthAncestor(a, depth[a] - depth[b]); // make them at the same depth.

**else** **if**(depth[b] > depth[a]) b = KthAncestor(b, depth[b] - depth[a]);

**if** (a == b) **return** a;

**for** (**int** j = logN-**1**; j >= **0**; --j)

**if** (ST[j][b] != ST[j][a])

a = ST[j][a], b = ST[j][b];

**return** ST[**0**][a];

}

**LONGEST WEIGHTED PATH**

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

vector<pair<**int**,**int**> > adj[N];

ll **dfs**(**int** u=**0**, **int** par=-**1**)

{

ll ret = **0**;

**for**(**int** i=**0**;i<adj[u].size();i++)

**if**(adj[u][i].f != par)

ret = max(ret, dfs(adj[u][i].F,u) + adj[u][i].s);

**return** ret;

}

**TARJAN SCC**

vector<vector<**int**> > SCCs /\* The components itself\*/;

#define comps SCCs

vector<**int**> compIndex /\* for each node, what is the index of the

component this node inside\*/

,ind, lowLink;

stack<**int**> st;

vector<**bool**> inst;

vector<vector<**int**> > adj /\*The intial graph\*/;

**int** idx = **0**; //must be intialized by zero;

**void** **tarjanSCC**(**int** i) {

lowLink[i] = ind[i] = idx++;

st.push(i);

inst[i] = true;

**for** (**int** j = **0**; j < adj[i].size(); j++) {

**int** k = adj[i][j];

**if** (ind[k] == -**1**) {

tarjanSCC(k);

lowLink[i] = min(lowLink[i], lowLink[k]);

} **else** **if** (inst[k]) {

lowLink[i] = min(lowLink[i], lowLink[k]);

}

}

**if** (lowLink[i] == ind[i]) {

vector<**int**> comp;

**int** n = -**1**;

**while** (n != i) {

n = st.top();

st.pop();

comp.push\_back(n);

inst[n] = **0**;

compIndex[n] = comps.size();

}

comps.push\_back(comp);

}

}

**void** **SCC**() {

comps.clear();

compIndex.resize(adj.size());

ind.clear();

ind.resize(adj.size(), -**1**);

lowLink.resize(adj.size());

inst.resize(adj.size());

idx = **0**; //must be intialized by zero;

**for** (**int** i = **0**; i < adj.size(); i++)

**if** (ind[i] == -**1**)

tarjanSCC(i);

}

**int** cntSrc /\*the number of source components\*/,

cntSnk /\*the number of sink components\*/;

vector<vector<**int**> > cmpAdj /\*The new graph between components\*/;

vector<**int**> inDeg, outDeg /\*the in degree and out degree for each

component\*/;

**void** **computeNewGraph**() {

outDeg.clear();

outDeg.resize(comps.size());

inDeg.clear();

inDeg.resize(comps.size());

cntSrc = cntSnk = comps.size();

cmpAdj.clear();

cmpAdj.resize(comps.size());

**for** (**int** i = **0**; i < adj.size(); i++) {

**for** (**int** j = **0**; j < adj[i].size(); j++) {

**int** k = adj[i][j];

**if** (compIndex[k] != compIndex[i]) {

cmpAdj[compIndex[i]].push\_back(compIndex[k]);

**if** (!(inDeg[compIndex[k]]++))

cntSrc--;

**if** (!(outDeg[compIndex[i]]++))

cntSnk--;

}

}

}

}

**TREE DIAMETER**

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

**int** sp[N];

**int** **bfs**(**int** u)

{

queue<**int**> q;

q.push(u);

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

sp[u] = **0**;

**while**(!q.empty())

{

u = q.front();

q.pop();

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

**if**(sp[v] == -**1**)

sp[v] = sp[u] + **1**,q.push(v);

}

**return** u;

}

**int** **calcTreeDiameter**(**int** root)

{

**int** u = bfs(root);

**int** v = bfs(u);

**return** sp[v];

}

**BELLMAN FORD**

// Shortest path from source to all nodes, negative edges allowed, no negative cycle

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

vector<pair<**int**,**int**> > adj[N];

**int** n,par[N]; // n is number of nodes

**bool** **bellmanFord**(**int** source)

{

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

memset(dis, **0x3F**, **sizeof**(dis));

dis[source] = **0**;

**bool** updated = **1**;

**for**(**int** k=**0**;k<n && updated;k++)

{

updated = **0**;

**for**(**int** u = **1**; u<=n;u++)

{

**for**(**auto**& e : edges[u])

{

**int** v = e.first;

**int** c = e.second;

**if**(dis[v] > dis[u] + c)

{

dis[v] = dis[u] + c;

par[v] = u;

updated = **1**;

}

}

}

}

**return** updated; // Whether a negative cycle exist or not

}

**BFS**

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

vector<**int**> adj[N];

**int** sp[N];

**void** **bfs**()

{

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

queue<**int**> q;

q.push(s); //pushes start

sp[s] = **0**;

**while**(!q.empty()){

**int** u = q.front();

q.pop();

**for**(**int** i=**0**;i<adj[u].size();i++){

**int** v = adj[u][i];

**if**(sp[v] == -**1**){

q.push(v);

sp[v] = sp[u] + **1**;

}

}

}

}

**DFS**

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

vector<**int**> adj[N];

**int** visited[N];

**void** **dfs**(**int** u){

visited[u] = **1**;

**for**(**int** i=**0**;i<adj[u].size();i++){

**int** v = adj[u][i];

**if**(!visited[v])

dfs(v);

}

}

**DIJKSTRA**

// Dijkstra SP for weighted graph + path build, No negative edges

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

vector< pair<**int**,**int**> > adj[N]; // adj[u] = {v,cost}

vector<**int**> path;

**int** sp[N]; **int** par[N];

**void** **build**(**int** u) {

**while** (u != -**1**) {

path.pb(u);

u = par[u];

}

reverse(path.begin(),path.end()); //The path is reversed, fix it

}

**int** **Dijkstra**(**int** source, **int** destination)

{

**int** u,v,c;

**for** (**int** i = **0**; i < n; ++i) // initialization

sp[i] = oo, par[i] = -**1**;

priority\_queue< pair<**int**,**int**> , vector<pair<**int**,**int**> >, greater<pair<**int**,**int**> > > pq;

pq.push( { **0**, source });

sp[source] = **0**;

**while** (!pq.empty()) {

u = pq.top().second;

c = pq.top().first;

pq.pop();

**if** (sp[u] < c)

**continue**;

**for** (**auto** edge : adj[u]) {

**int** to = edge.first;

**int** cost = edge.second;

**if** (c + cost < sp[to]) {

sp[to] = c + cost;

pq.push( { sp[to], to });

//assign (u) as parent of (to) to rebuild the path

par[to] = u;

}

}

}

build(destination); // Path is stored in path

**return** sp[destination];

}

MATH

**PRIME FACTORIZATION**

**const** **int** MAX = **1e6**+**10**;

**int** spf[MAX], power[MAX]

**void** Sieve() // O(nlogn)

{

spf[**1**] = **1**;

**for** (**int** i = **2**; i < MAX; i++) spf[i] = i;

**for** (**int** i = **4**; i < MAX; i += **2**) spf[i] = **2**;

**for** (**int** i = **3**; i < MAX; i++)

**if** (spf[i] == i)

**for** (**int** j = i; j < MAX; j += i)

**if** (spf[j] == j)

spf[j] = i;

}

**void** GetFactors(**int** num)

{

**int** cnt = **1**, last = spf[num], ans = **1**;

**while** (num != **1**) {

num = num / spf[num];

**if** (last == spf[num]) {

cnt++;

**continue**;

}

power[last] = cnt;

cnt = **1**;

last = spf[num];

}

}

**nCr WITH MOD**

ll mod = **1e9**+**7**;

**const** **int** N = **1e5**+**10**;

ll fact[N];

**void** **ComputeFactorial**() // Don't forget to compute the factorial

{

fact[**0**] = **1**;

**for**(ll i=**1**;i<N;i++)

fact[i] = (fact[i-**1**]%mod \* i%mod)%mod;

}

ll **power**(ll x, ll y)

{

ll res = **1**;

x = x % mod;

**while** (y > **0**)

{

**if** (y & **1**)

res = (res\*x) % mod;

y = y>>**1**;

x = (x\*x) % mod;

}

**return** res;

}

ll **modInverse**(ll n)

{

**return** power(n, mod-**2**);

}

ll **nCr**(ll n, ll r)

{

**if** (r==**0**)

**return** **1**;

**return** (fact[n]\* modInverse(fact[r]) % mod \*

modInverse(fact[n-r]) % mod) % mod;

}

**SIEVE**

**const** **int** MAX = **1e6**+**10**;

vector<**bool**> prime(MAX, true);

**void** **Sieve**() // O(nlog(logn))

{

prime[**0**] = prime[**1**] = **0**;

**for** (**int** i = **2**; i\*i <= MAX; i++)

**if** (prime[i])

**for** (**int** j = i \* **2**; j <= MAX; j += i)

prime[j] = **0**;

}

**SMALLEST PRIME FACTOR**

**const** **int** MAX = **1e6**+**10**;

**int** spf[MAX];

**void** **Sieve**() // O(nlogn)

{

spf[**1**] = **1**;

**for** (**int** i = **2**; i < MAX; i++) spf[i] = i;

**for** (**int** i = **4**; i < MAX; i += **2**) spf[i] = **2**;

**for** (**int** i = **3**; i < MAX; i++)

**if** (spf[i] == i)

**for** (**int** j = i; j < MAX; j += i)

**if** (spf[j] == j)

spf[j] = i;

}

**EULER PHI**

// Theorem: a^phi(b) % b = 1, if a,b are coprimes (gcd(a,b) == 1)

ll **Phi**(ll n) // O(sqrt(n)), returns Euler Phi of n

{

ll result = n;

**for**(**int** i=**2**;i\*i<=n;i++)

{

**if**(n%i == **0**)

result -= result/i;

**while**(n%i==**0**)

n/=i;

}

**if**(n>**1**)

result -=result/n;

**return** result;

}

ll phi[N];

**void** **EulerTotient**() // O(nlog(logn)), generates all Euler Phi's to N

{

**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;

}

**EXTENDED EUCLID**

// O( log(max(a,b)) )

// Returns Bezout's coeffs of the smallest

// linear combination of s.a + t.b = gcd(a,b)

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)};

}

**FASTPOWER WITH MOD**

ll **power**(ll base, ll exp)

{

ll ans = **1**;

base%=mod;

**while**(exp>**0**)

{

**if**(exp&**1**) ans = (ans\*base)%mod;

exp>>=**1**;

base = (base\*base)%mod;

}

**return** ans;

}

**IS PRIME**

**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;

}

**MATH MISC**

ll **gcd**(ll a, ll b) { **return** (b == **0** ? a : gcd(b, a % b)); } // O(log(max(a,b)))

ll **lcm**(ll a, ll b){ **return** ((a\*b) / gcd(a, b)); } // O(log(max(a,b)))

ll **modInverse**(ll a, ll m) // O(logm)

{

**return** power(a,m-**2**,m);

}

ll **nCr**(ll n, ll r) // O(r) - Call the function as nCr(n,min(r,n-r)) for better performance

{

**if**(n<r) **return** **0**;

**if**(r==**0**) **return** **1**;

**return** n\*nCr(n-**1**,r-**1**)/r;

}

// Factorization - O(sqrt(n))

vector<ll>factors;

**void** **GetFactors**(ll n){

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

**if** (n%i == **0**) {

**if** (i != n / i)

factors.push\_back(i), factors.push\_back(n / i);

**else**

factors.push\_back(i);

}

}

}

//Mod of very large numbers - O(n)

//rule: xy (mod a) ≡ ((x (mod a) \* y) (mod a))

ll **mod**(string num)

{

**int** res = **0**;

**for** (**int** i = **0**; i < num.size(); i++)

res = (res\***10** + num[i] - '0') %mod;

**return** res;

}

// Derangements - O(n)

ll **fact**(**int** n)

{

ll ret = **1**;

**for**(**int** i=n;i>**1**;i--) ret = ret\* **1LL** \* i;

**return** ret;

}

ll **pww**(ll p){**return** p&**1** ? -**1** : **1**;}

ll **Derangement**(ll n)

{

ll tmp = fact(n), sum=**0**;

**for**(**int** i=**0**;i<=n;i++) sum+= (pww(i) \* tmp) / fact(i);

**return** sum;

}

// isPowerOfTwo - O(1)

**bool** **isPowerOfTwo**(ll x) {**return** x && (!(x&(x-**1**)));}

**MATRIX POWER**

**const** **int** mod = **1e9**+**7**;

**const** **int** MatSize = **2**;

**struct** Matrix

{

**long** **long** mat[MatSize][MatSize];

};

Matrix **MatMul**(Matrix &a , Matrix &b)

{

Matrix res;

**for**(**int** i = **0** ; i<MatSize ; i++)

**for**(**int** j = **0** ; j<MatSize ; j++)

{

res.arr[i][j] = **0**;

**for**(**int** k = **0** ; k<MatSize ; k++)

{

**long** **long** sum = a.arr[i][k] \* b.arr[k][j];

sum%=mod;

res.arr[i][j] +=sum;

res.arr[i][j]%=mod;

}

}

**return** res;

}

Matrix **MatIdentity**()

{

Matrix res;

**for**(**int** i = **0** ; i<MatSize ; i++)

**for**(**int** j = **0** ; j<MatSize ; j++)

res.arr[i][j] = (i==j);

**return** res;

}

Matrix **MatPower**(Matrix a , **long** **long** x)

{

Matrix res = MatIdentity();

**while**(x)

{

**if**(x&**1**)

{

res = MatMul(res , a);

}

a = MatMul(a,a);

x>>=**1**;

}

**return** res;

}

**TO BINARY**

string **toBinary**(**int** num)

{

string tmp = "";

**if**(!num) tmp = "0";

**while**(num) tmp+=( (num&**1**)+'0'),num>>=**1**;

reverse(tmp.begin(),tmp.end());

**return** tmp;

}

**DIGIT SUM**

**int** **DigitSum**(ll x)

{

**int** ret = **0**;

**while**(x) ret+=(x%**10**),x/=**10**;

**return** ret;

}

// DigitSum(a+b) = DigitSum(DigitSum(a)+DigitSum(b))

// DigitSum(a-b) = DigitSum(DigitSum(a)-DigitSum(b))

// DigitSum(a\*b) = DigitSum(DigitSum(a)\*DigitSum(b))

**IS PALINDROME**

**bool** **Is\_palindrome**(string &p){

**int** s = **0** , e = p.size() - **1**;

**while**(s < e){

**if**(p[s] != p[e])**return** false;

s++ , e--;

}

**return** true;

}