Contents

0.1 Kick Off

```
#include <bits/stdc++.h>
using namespace std;
using ll = long long;
#define pb push_back
#define pi 2 * acos(0.0)
#define all(f) (f).begin(), (f).end()
#define rall(f) (f).rbegin(), (f).rend()
#define Files                           \
     freopen("input.txt", "r", stdin); \
     freopen("output.txt", "w", stdout);
#define Faster                          \
     ios_base::sync_with_stdio(false); \
     cin.tie(NULL);
#define fraction()                      \
     cout.unsetf(ios::floatfield); \
     cout.precision(6);            \
     cout.setf(ios::fixed, ios::floatfield);
```

0.2 PBDS

```
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>
using namespace __gnu_pbds;
template <typename T>
using o_set = tree<T, null_type, less<T>,
    rb_tree_tag, tree_order_statistics_node_update
    >;
//*os.find_by_order(idx)=>value of index(0 based)
//os.order_of_key(x)=>number of value less than x
```

0.3 Modular Functions

```
inline ll _norm(ll a, ll M) { a %= M; if (a < 0) a
    += M; return a; }
inline ll modAdd(ll a, ll b, ll M) { a = _norm(a,
    M), b = _norm(b, M); return (a + b) % M; }
inline ll modSub(ll a, ll b, ll M) { a = _norm(a,
    M), b = _norm(b, M); ll S = (a - b) % M; if (S
     < 0) S += M; return S; }
inline ll modMul(ll a, ll b, ll M) { a = _norm(a,
    M), b = _norm(b, M); return (a * b) % M; }
inline ll bigMod(ll a, ll b, ll M) { ll res = 1;
    while (b > 0) { if (b & 1LL) res = modMul(res,
     a, M); a = modMul(a, a, M); b >>= 1LL; }
    return res; }
inline ll fermats_inverse(ll a, ll M) { return
    bigMod(a, M - 2, M); } // (a, M) must be
    coprime
inline ll modDiv(ll a, ll b, ll M) { return modMul
    (a, fermats_inverse(b, M), M); }
```

0.4 Segmented Sieve

```
const ll inf = (ll)(sqrt(2147483647)) + 100;
vector<bool> mark(inf);
vector<int> primes;
void sieve() { } // at first precalculate simple
    sieve and generate primes
void seg_sieve(ll L, ll R){
     vector<bool> check(R - L + 1);
     for (ll i = 0; primes[i] <= (R + 1) / primes[
         i]; i++){
          ll base = (L / primes[i]) * primes[i];
          if (base < L)
               base += primes[i];
          for (ll j = base; j <= R; j += primes[i
              ])
               check[j - L] = 1;
          if (base == primes[i])
               check[base - L] = 0;
     }
     if (L == 1)
          check[0] = 1;
     for (int i = 0; i < R - L + 1; i++)
          if (!check[i]) cout << i + L  << ' ';

}
```

0.5 Smallest Prime Factor

```
const ll inf = 2e5 + 5;
vector<int> spf(inf);
void SPF() {
     spf[1] = 1;
     for (int i = 2; i < inf; i += 2)
          spf[i] = 2;
     for (int i = 3; i < inf; i += 2) {
          if (!spf[i]){
               spf[i] = i;
               if (i <= (inf + 1) / i)
                    for (int j = i * i; j < inf; j
                        += (i + i))
                         if (!(spf[j]))spf[j] = i;
          }
     }
```

```
}
```

0.6 Phi Generate

```
const ll inf = 2e5 + 5;
vector<int> phi(inf);
void phi_to_N() {
     for (int i = 1; i < inf; i++)
          phi[i] = i;
     for (int i = 2; i < inf; i++) {
          if (phi[i] == i)
               for (int j = i; j < inf; j += i)
                    phi[j] -= (phi[j] / i);
     }
}
```

0.7 Extended GCD

```
ll extentedGCD(ll a, ll b, ll &x, ll &y) {
// return gcd of (a,b) and change x,y as
    referrance. if gcd(a,b)=1 then x is equal a^-1
     MOD b.
     if (b == 0) {
          x = 1;
          y = 0;
          return a;
     }
     ll x1, y1;
     ll d = extentedGCD(b, a % b, x1, y1);
     x = y1;
     y = x1 - (a / b) * y1;
     return d;
}
```

0.8 Important Formulas

$$\text{Sum of Divisors} = \prod_{i=1}^{k}\left(\frac{p_i^{e_i+1}-1}{p_i-1}\right)$$

$$\text{Number of Divisors} = \prod_{i=1}^{k}(e_i+1)$$

$$\text{Product of Divisors} \quad = n^{d(n)/2}$$

$$\text{Euler Totient } \phi(n) = n \times \prod_{i=1}^{k}\left(1-\frac{1}{p_i}\right)$$

$$\text{GCD Sum } g(n) = \prod_{i=1}^{k}\left((e_i+1)p_i^{e_i} - e_i p_i^{e_i-1}\right)$$

$$\text{Sum of Coprimes } f(n) = \frac{\phi(n)}{2}n$$

$$x^n = x^{\phi(m)+[n \mod \phi(m)]} \mod m; \text{ n>= log2 m}$$

0.9 Segment Tree Lazy Propagation

```
// lazy segTree code explanetion
// be careful about data types and SegTree node
    type
#define lc (node << 1)
#define rc ((node << 1) + 1)
const int inf = 3e5;
int n, q;
ll arr[inf], segTree[4 * inf], lazy[4 * inf];
void build(int node, int b, int e) {
     if (b == e) {
          segTree[node] = arr[b];
          lazy[node] = 0;
          return;
     }
     int mid = (b + e) >> 1;
     build(lc, b, mid);
     build(rc, mid + 1, e);
     segTree[node] = min(segTree[lc], segTree[rc])
         ; // this 'merge' may need change per
         problem (min,max,sum,gcd etc..)
}

void prop(int node, int b, int e) {
// to propagate the pendings
     if (lazy[node] != 0) {
          // very important thing, be careful
          // if the update is [set, replace] then
              "segTree[node] = lazy[node]" and
              same assigning goes for childs
          // if the update is [add, increase] then
               "segTree[node] += lazy[node]" and
              same assigning goes for childs
          // segTree[node] will be multiplied by
              its size in case of sum but not for
              min,max
          // node size = (e - b + 1) /// end -
              begin + 1
          segTree[node] += lazy[node];
          if (b != e) // toward childs
               lazy[lc] += lazy[node], lazy[rc] +=
                    lazy[node];
          lazy[node] = 0;
     }
}
ll dummy = 1e18;
ll qry(int node, int b, int e, int L, int R) {
     prop(node, b, e); // careful
     if (b > R or e < L)
          return dummy; // check per problem
     if (b >= L and e <= R)
          return segTree[node];
     int mid = (b + e) >> 1;
     return min((qry(lc, b, mid, L, R)), (qry(rc,
         mid + 1, e, L, R))); // this 'merge' may
         need change per problem (min,max,sum,gcd
         etc..)
}
void upd(int node, int b, int e, int L, int R, ll
    val) {
     prop(node, b, e);
     if (b > R or e < L)
          return;
```

```
    if (b >= L and e <= R) {
        // very important thing, be careful
        // if the update is [set, replace] then
            "segTree[node] = val" and same
            assigning goes for childs
        // if the update is [add, increase] then
             "segTree[node] += val" and same
            assigning goes for childs
        // segTree[node] will be multiplied by
            its size in case of sum but not for
            min,max
        // node size = (e - b + 1) /// end -
            begin + 1
        segTree[node] += val;
        if (b != e)
            lazy[lc] += val, lazy[rc] += val;
        return;
    }
    int mid = (b + e) >> 1;
    upd(lc, b, mid, L, R, val);
    upd(rc, mid + 1, e, L, R, val);
    segTree[node] = min(segTree[lc], segTree[rc])
        ; // this 'merge' may need change per
        problem (min,max,sum,gcd etc..)
}
// need to build --- build(1, 1, n)
```

0.10 Sparse Table

```
 const int N = 1e5 + 9;
int t[N][18], a[N];
void build(int n) {
  for(int i = 1; i <= n; ++i) t[i][0] = a[i];
  for(int k = 1; k < 18; ++k) {
    for(int i = 1; i + (1 << k) - 1 <= n; ++i) {
      t[i][k] = min(t[i][k - 1], t[i + (1 << (k -
          1))][k - 1]);
    }
  }
}
int query(int l, int r) {
  int k = 31 - __builtin_clz(r - l + 1);
  return min(t[l][k], t[r - (1 << k) + 1][k]);
}
```

0.11 Hashing

```
// at first, must attach the modular functions in
    source code
const int PB1 = 137, PB2 = 347, PM1 = 1e9 + 7, PM2
     = 1e9 + 9, inf = 3e5;
// pair for double hashing
vector<pair<int, int>> pw(inf), inv_pw(inf);
void precalc_Powers() // every powers of Base %
    Mod
{
    pw[0] = {1, 1};
    for (int i = 1; i < inf; i++)
    {
        pw[i].first = modMul(pw[i - 1].first,
            PB1, PM1);
        pw[i].second = modMul(pw[i - 1].second,
            PB2, PM2);
    }
    int inv_base1 = fermats_inverse(PB1, PM1);
    int inv_base2 = fermats_inverse(PB2, PM2);
    inv_pw[0] = {1, 1};
    for (int i = 1; i < inf; i++)
    {
        inv_pw[i].first = modMul(inv_pw[i - 1].
            first, inv_base1, PM1);
        inv_pw[i].second = modMul(inv_pw[i - 1].
            second, inv_base2, PM2);
    }
}

pair<int, int> F_Hash(const string &st1) //
    forward string Hash
{
    pair<int, int> res = {0, 0};
    int sz = st1.size();
    for (int i = 0; i < sz; i++)
    {
        res.first = modAdd(res.first, modMul((
            st1[i] - 'a' + 1), pw[i].first, PM1)
            , PM1);
        res.second = modAdd(res.second, modMul((
            st1[i] - 'a' + 1), pw[i].second, PM2
            ), PM2);
    }
    return res;
}

vector<pair<int, int>> pref_Hash(inf);
void Build(const string &str1) // Building Prefix
{
    int sz = str1.size();
    for (int i = 0; i < sz; i++)
    {
        pref_Hash[i].first = modAdd((i == 0) ? 0
             : pref_Hash[i - 1].first, modMul((
            str1[i] - 'a' + 1), pw[i].first, PM1
            ), PM1);
        pref_Hash[i].second = modAdd((i == 0) ?
            0 : pref_Hash[i - 1].second, modMul
            ((str1[i] - 'a' + 1), pw[i].second,
            PM2), PM2);
    }
}

pair<int, int> get_Hash(int L, int R) // Hash of a
     substring
{
    pair<int, int> res = {0, 0};
    res.first = modSub(pref_Hash[R].first, (L ==
        0) ? 0 : pref_Hash[L - 1].first, PM1);
    res.first = modMul(res.first, inv_pw[L].first
        , PM1);
    res.second = modSub(pref_Hash[R].second, (L
        == 0) ? 0 : pref_Hash[L - 1].second, PM2)
        ;
    res.second = modMul(res.second, inv_pw[L].
        second, PM2);
    return res;
}
```

```
void Solve()
{
	string s;
	cin>>s;
	Build(s);

}
```

0.12 BFS Overview

```
//"Path Sequence"
const int inf = 3e5;
vector<int> adj[inf];
vector<bool> vis(inf);
vector<int> lev(inf), par(inf);
void bfs(int src)
{ // par vector stores parent of a node
	queue<int> q;
	q.push(src);
	vis[src] = true;
	lev[src] = 0;
	par[src] = -1;
	while (!q.empty()) {
		int at = q.front();
		q.pop();
		for (auto u : adj[at]) {
			if (!vis[u]) {
				q.push(u);
				vis[u] = true;
				lev[u] = lev[at] + 1;
				par[u] = at;
			}
		}
	}
}

int src = 1, target;
cin >> target;
bfs(src);
vector<int> s_path;
s_path.push_back(target);
while (par[target] != -1){
	s_path.push_back(par[target]);
	target = par[target];
}
reverse(s_path.begin(), s_path.end());
for (auto u : s_path)
	cout << u << ' ';
cout << '\n';

// Bi partite coloring by BFS

const int inf = 3e5;
vector<int> adj[inf];
vector<bool> vis(inf);
vector<int> colr(inf);
bool bfs(int src, int c)
{ // bipartite checking by BFS
	queue<int> q;
	q.push(src);
	vis[src] = true;
	colr[src] = 1;
	while (!q.empty()) {
		int at = q.front();
		q.pop();
		for (auto u : adj[at]) {
			if (!vis[u]) {
				q.push(u);
				colr[u] = (colr[at] ^ 1);
				vis[u] = true;
			}
			else if (colr[at] == colr[u])
				return false;
		}
	}
	return true;
}

// MULTIPLE source BFS
// push all sources into the queue at first
// BFS on 2-D Grid
// Direction arrays
// first four is for 4 directions
int dx[] = {1, -1, 0, 0, 1, -1, -1, 1};
int dy[] = {0, 0, -1, 1, -1, 1, -1, 1};
// knights move
int kdx[] = {1, 1, -1, -1, 2, 2, -2, -2};
int kdy[] = {2, -2, 2, -2, 1, -1, 1, -1};
void bfs(pair<int, int> src)
{
	memset(vis, 0, sizeof vis);
	queue<pair<int, int>> q;
	q.push({src.first, src.second});
	vis[src.first][src.second] = 1;
	while (!q.empty()) {
		pair<int, int> at = q.front();
		q.pop();
		for (int i = 0; i < 4; i++)  {
			int nx = at.first + dx[i];
			int ny = at.second + dy[i];
			if (nx >= 1 and nx <= row and ny >=
			  1 and ny <= col and (!vis[nx][
			ny]) and (arr[nx][ny] != '#'))
			  {
				q.push({nx, ny});
				vis[nx][ny] = 1;
			}
		}
	}
}

// 0-1 BFS

const int inf = 3e5 + 100;
vector<pair<int, int>> adj[inf];
int cost[inf];
int node;
void bfs_01(int src) {
	for (int i = 0; i < node + 10; i++)
		cost[i] = INT_MAX; // initializing with
		   infinity
	deque<int> dq;
	dq.push_front(src);
	cost[src] = 0;
	while (!dq.empty()) {
		int at = dq.front();
		dq.pop_front();
```

```
        for (auto u : adj[at])  {
            int child = u.first;
            int wt = u.second;
            if (cost[at] + wt < cost[child]) {
                cost[child] = cost[at] + wt;
                if (wt == 1)
                    dq.push_back(child);
                else
                    dq.push_front(child);
            }
        }
    }
}

// Dijkstra Allgorithm (SSSP in weighted graph)
const int inf = 3e5;
int node, edge;
vector<pair<int, int>> adj[inf];
ll dist[inf];
void dijkstra(int src) {
    for (int i = 0; i < inf; i++)
        dist[i] = 2e18;
    priority_queue<pair<ll, int>, vector<pair<ll,
        int>>, greater<pair<ll, int>>> pq;
    pq.push({0LL, src});
    dist[src] = 0;
    while (!pq.empty()) {
        pair<ll, int> at = pq.top();
        ll curDis = at.first;
        int curNode = at.second;
        pq.pop();
        if (dist[curNode] < curDis)
            continue;
        for (auto u : adj[curNode]) {
            int child = u.first;
            ll wt = u.second;
            if (curDis + wt < dist[child]) {
                dist[child] = curDis + wt;
                pq.push({dist[child], child});
            }
        }
    }
}

// ## a sequence is valid in BFS Traversal or not

// If a node X is visited before a node Y, then it
   's safe to assume X appears before Y in every
   adjacency list.
// So initially we can sort each list, using as
   comparator the positions in the Given Sequence
   .
// Then we can just run a BFS and check if we
   visit the nodes in the given order.

int node;
vector<int> adj[inf];
bool vis[inf];
vector<int> bfs_path;
int pos[inf];
void bfs(int src)
{
    queue<int> q;
    q.push(src);
    vis[src] = 1;
    bfs_path.push_back(src);
    while (!q.empty())
    {
        int at = q.front();
        q.pop();
        for (auto u : adj[at]) {
            if (!vis[u]) {
                q.push(u);
                vis[u] = 1;
                bfs_path.push_back(u);
            }
        }
    }
}
bool cmp(int a, int b) {
    if (pos[a] < pos[b])
        return true;
    return false;
}

void Solve()
{
    cin >> node;
    for (int i = 1; i < node; i++) {
        int u, v;
        cin >> u >> v;
        adj[u].push_back(v);
        adj[v].push_back(u);
    }

    vector<int> given;
    for (int i = 1; i <= node; i++) {
        int x;
        cin >> x;
        given.push_back(x);
        pos[x] = i;
    }

    for (int i = 1; i <= node; i++) // sort
       adjacency list according to input order
        sort(adj[i].begin(), adj[i].end(), cmp);

    bfs(1);

    if (bfs_path == given)
        cout << "Yes\n";
    else
        cout << "No\n";
}
```

0.13 DFS Overview

```
const int inf = 3e5;
vector<int> adj[inf];
vector<bool> vis(inf);
void dfs(int src)
{
    vis[src] = true;
    for (auto u : adj[src])
        if (!vis[u])
            dfs(u);
}
```

```
}
// Connected Component size (component size is
    going to be stored in comp_sz)
const int inf = 3e5;
vector<int> adj[inf];
vector<bool> vis(inf);
int comp_sz = 0;
void dfs(int src)
{
     vis[src] = true;
     comp_sz++;
     for (auto u : adj[src])
          if (!vis[u])
               dfs(u);
}

const int inf = 3e5;
vector<int> adj[inf];
vector<bool> vis(inf), colr(inf);
bool dfs(int src, int c)
{ // bi-coloring by DFS
     vis[src] = true;
     colr[src] = c;
     int tmp;
     if (c == 1)
          tmp = 0;
     else
          tmp = 1;
     for (auto u : adj[src]) {
          if (!vis[u]) {
               if (dfs(u, tmp) == false)
                    return false;
          }
          else if (colr[src] == colr[u])
               return false;
     }
     return true;
}

// better
const int inf = 3e5;
vector<int> adj[inf];
vector<bool> vis(inf), colr(inf);
bool dfs(int src, int c)
{ // bi-coloring by DFS (Better Version)
     vis[src] = true;
     colr[src] = c;
     for (auto u : adj[src])
     {
          if (!vis[u]) {
               if (dfs(u, (c ^ 1)) == false) //
                   xor alternates value
                    return false;
          }
          else if (colr[src] == colr[u])
               return false;
     }
     return true;
}
// Direction arrays

// first four is for 4 directions
int dx[] = {1, -1, 0, 0, 1, -1, -1, 1};
int dy[] = {0, 0, -1, 1, -1, 1, -1, 1};
```

```
// knights move
int kdx[] = {1, 1, -1, -1, 2, 2, -2, -2};
int kdy[] = {2, -2, 2, -2, 1, -1, 1, -1};

// DFS on 2-D Grid (By problem solving)
int row, col;
char arr[row][col];
bool vis[row][col];
void dfs(pair<int, int> src)
{
     vis[src.first][src.second] = 1;
     for (int i = 0; i < 4; i++) {
          int nx = src.first + dx[i], ny = src.
              second + dy[i];
          if (nx >= 1 and nx <= row and ny >= 1
              and ny <= col and (!vis[nx][ny]) and
               (arr[nx][ny] != '#'))
               dfs({nx, ny});
     }
}

// Cycle Detection by DFS
bool vis[inf];
bool isCycle(int src, int par)
{
     vis[src] = 1;
     for (auto u : adj[src]) {
          if (!vis[u])  {
               if (isCycle(u, src) == true)
                    return true;
          }
          else if (u != par)
               return true;
     }
     return false;
}
//
    ---------------------------------------------------

// cycle findinding and print cycle (any one e
    cycle only)
const int inf = 2e5;
int node, edge;
vector<int> adj[inf];
vector<int> _cycle;
bool vis[inf];
bool dfs(int src, int par)
{
     vis[src] = 1;
     _cycle.push_back(src);
     for (auto u : adj[src])
     {
          if (!vis[u]) {
               if (dfs(u, src) == true)
                    return true;
          }
          else if (u != par) {
               _cycle.push_back(u);
               return true;
          }
     }
     _cycle.pop_back();
     return false;
}
```

```
// if this function return true then the _cycle
    vector should be printed from the last ,until
    it finds the last node again

// cycle detection in directed graph by DFS

bool vis[inf];
int IN[inf], OUT[inf];
int timer = 1;
void dfs(int src)
{ // IN TIME and OUT TIME
     vis[src] = 1;
     IN[src] = timer++;
     for (auto u : adj[src])
          if (!vis[u])
               dfs(u);
     OUT[src] = timer++;
}

// ## problem
// "Given N(1e5) nodes and Q queries. In each
    Query, given 2 nodes, find whether one node
    lies in the subree of another node."

//     "FACT - if node X is in the Subtree of node
     Y" -
//     IN TIME of X > IN TIME of Y
//     and
//     OUT TIME of X < OUT TIME of Y

// store Subtree size of all nodes using DFS

// subtree size of Root node = 1 + Subtree size of
     it's Childs
bool vis[inf];
int subtree_size[inf];
int dfs_SubtreeSize(int src)
{
     vis[src] = 1;
     int cur_sz = 1;
     for (auto u : adj[src])
          if (!vis[u])
               cur_sz += dfs_SubtreeSize(u);
     return subtree_size[src] = cur_sz;
}

// ## a sequence is valid in DFS Traversal or not
// If a node X is visited before a node Y, then it
    's safe to assume X appears before Y in every
    adjacency list.
// So initially we can sort each list, using as
    comparator the positions in the Given Sequence
    .
// Then we can just run a BFS and check if we
    visit the nodes in the given order.

using ll = long long;
const int inf = 1e5 + 100;
int node, edge;
vector<int> adj[inf];
bool vis[inf];
vector<int> dfs_path, given;
int pos[inf];
```

```
void dfs(int src)
{
     vis[src] = 1;
     dfs_path.push_back(src); // dfs path visited
     for (auto u : adj[src])
          if (!vis[u])
               dfs(u);
}
bool cmp(int a, int b)
{
     if (pos[a] < pos[b])
          return true;
     return false;
}
void Solve()
{
     cin >> node >> edge;
     for (int i = 1; i <= node; i++)
     {
          int x;
          cin >> x;
          given.push_back(x);
          pos[x] = i; // storing the node order of
               input (positions)
     }
     while (edge--)
     {
          int u, v;
          cin >> u >> v;
          adj[u].push_back(v);
          adj[v].push_back(u);
     }
     for (int i = 1; i <= node; i++) // sort
        adjacency list according to input order
          sort(adj[i].begin(), adj[i].end(), cmp);
     dfs(1);
     if (given == dfs_path)
          cout << 1 << '\n';
     else
          cout << 0 << '\n';
}
```

0.14 Bitwise Go Through

```
#define LSB_set(x) __builtin_ffs(x)
#define leading_zero(x) __builtin_clz(x) // for
    integer [*add ll for long long*]
#define trailing_zero(x) __builtin_ctz(x)
#define count_set_bits(x) __builtin_popcount(x) //
     count set bits

inline bool check_Bit(ll n, int i) { return (n &
    (1LL << i)); }
inline ll set_Bit(ll n, int i) { return (n | (1LL
    << i)); }
inline ll reset_Bit(ll n, int i) { return (n &
    (~(1LL << i))); }
inline ll flip_Bit(ll n, int i) { return (n ^ (1LL
     << i)); }

bool isPowerOfTwo(ll n) { return n && !(n & (n -
    1)); }
```

```
// if((n&1)==0) then n is even [*last bit 0*]
// if((n&1)==1) then n is odd [*last bit 1*]
// (n>>k) is equal (n/(2^k))
// (n<<k) is equal (n*(2^k))
// (n MOD 2^k) equals to (n&((1<<k)-1)) (*keeps
    last k bits as usual*)

int computeXOR(int n)
{ // xor from 1 to n
     if (n % 4 == 0)
          return n;
     if (n % 4 == 1)
          return 1;
     if (n % 4 == 2)
          return n + 1;
     else
          return 0;
}

//  Equal Sum and XOR
// Problem: Given a positive integer n, find count
     of positive integers i such that 0 <= i <= n
    and (n+i = n XOR i)
// Instead of Brute force method, we can directly
    find it by a mathematical trick
// Let x be the number of unset bits in the number
     n.
// Answer = 2^x
// hamming distance(a,b) : Number of set bits in (
    a^b)

int clear_last_i_bit(int n, int i)
{ // Clear last i Bit
     int mask = (-1 << i);
     n = (n & mask);
     return n;
}
int clear_bits_in_range(int n, int i, int j)
{ // Clear Bits In Range
     int a = (-1 << j + 1);
     int b = (i << i - 1);
     int mask = (a | b);
     n = (n & mask);
     return n;
}
void all_Subsets(vector<int> v, int N)
{ // bit masking
     for (int mask = 0; mask < (1 << N); ++mask)
     {
          for (int i = 0; i < N; ++i)
               if (mask & (1 << i))
                    cout << v[i] << ' ';
          cout << endl;
     }
}
long largest_power(long N)
{ // largets power of 2 less or equal N
     // changing all right side bits to 1.
     N = N | (N >> 1);
     N = N | (N >> 2);
     N = N | (N >> 4);
     N = N | (N >> 8);
     // here for 16 bit number
     // as now the number is 2 * x-1, where x is
        required answer, so adding 1 and dividing
         it by2.
     return (N + 1) >> 1;
}

// a|b = a XOR b + a&b
// a XOR (a&b) = (a|b) XOR b
// (a&b) XOR (a|b) = a XOR b
// a+b = a|b + a&b
// a+b = a XOR b + 2(a&b)

// all pair xor
void all_pair_xor()
{
     cin >> n;
     vector<ll> v(n);
     for (int i = 0; i < n; i++)
          cin >> v[i];
     ll sum = 0;
     for (ll i = 0; i < 25; i++)
     {
          ll zc = 0, oc = 0;
          ll idsum = 0;
          for (int j = 0; j < n; j++)
          {
               if (v[j] % 2 == 0)
                    zc++;
               else
                    oc++;
               v[j] /= 2;
          }

          idsum = 1LL * oc * zc * (1LL << i);
          sum += idsum;
     }
     cout << sum << '\n';
}
```

0.15 Geometry basic

```
struct pt
{
     T x, y; // if coordinates are fraction the
        long double needed
     void read() { cin >> x >> y; }
     pt operator+(pt p) { return {x + p.x, y + p.y
        }; }
     pt operator-(pt p) { return {x - p.x, y - p.y
        }; }
     // data type of operand and point must be
        same for the next two
     pt operator*(T d) { return {x * d, y * d}; }
     pt operator/(T d) { return {x / d, y / d}; }
        // only for floating-point
     T norm2() { return x * x + y * y; }
};
// Picks Theorem
struct pt
{ // 2-D points
     ll x, y;
     void read() { cin >> x >> y; }
};
```

```cpp
int n;

ll cross(pt a, pt b) { return a.x * b.y - a.y * b.
    x; } // cross product

T areaPolygon(vector<pt> &v)
{ // shoelace Theorem (points must have sorted)
     T A = 0.0;
     for (int i = 0, n = v.size(); i < n; i++)
          A += T(cross(v[i], v[(i + 1) % n]));
     return fabs(A) * 0.5;
}

pair<ll, ll> picks_Thm(vector<pt> &v) // area =
    Interior + Boundary/2 -1
{
     ll B = v.size();
     for (int i = 0, n = v.size(); i < n; i++)
     {
          ll dx = abs(v[i].x - v[(i + 1) % n].x);
              // difference of X coordinate
          ll dy = abs(v[i].y - v[(i + 1) % n].y);
              // difference of Y coordinate
          B += ((__gcd(dx, dy)) - 1);
     }
     T A = 2.0 * areaPolygon(v); // doubled area
         to avoid fraction
     ll I = (A + 2 - B) / 2;
     return {B, I};
}

bool operator==(pt a, pt b) { return a.x == b.x &&
     a.y == b.y; }
bool operator!=(pt a, pt b) { return !(a == b); }

T sqr(pt p) { return p.x * p.x + p.y * p.y; }  //
    sqaured value
long double mag(pt p) { return sqrt(sqr(p)); } //
    magnitude
pt perp(pt p) { return {-p.y, p.x}; }          //
    perpendicular point

T dot(pt v, pt w) { return v.x * w.x + v.y * w.y;
    } // dot product
bool isPerp(pt v, pt w) { return dot(v, w) == 0; }
     // perpendicular or not, be careful about v,
     w
long double internal_angle(pt v, pt w)
                  // angle in radian between two
    vectors
{
     long double cosTheta = dot(v, w) / mag(v) /
         mag(w);
     return acos(max((T)-1.0, min((T)1.0, cosTheta
         )));
}

T cross(pt v, pt w) { return v.x * w.y - v.y * w.x
    ; }      // cross product
T orient(pt a, pt b, pt c) { return cross(b - a, c
     - a); } // It is positive if C is on the left
     side of AB, negative on the right side, and
    zero if C is on the line containing AB.
bool isParallel(pt v, pt w) { return cross(v, w)
    == 0; }   // parallel or not, be careful about
     v, w

bool inAngle(pt a, pt b, pt c, pt p) // to check
    if point P lies in the angle formed by lines
    AB and AC.
{
     assert(orient(a, b, c) != 0);
     if (orient(a, b, c) < 0)
          swap(b, c);
     return orient(a, b, p) >= 0 && orient(a, c, p
         ) <= 0;
}
bool inDisk(pt a, pt b, pt p) { return dot(a - p,
    b - p) <= 0; }
bool onSegment(pt a, pt b, pt p) { return orient(a
    , b, p) == 0 and inDisk(a, b, p); } //
    determine whether p is on(touch) segment ab or
     not.

bool above(pt a, pt p) { return p.y >= a.y; }

    // true if P at least as high as A
bool crossesRay(pt a, pt p, pt q) { return (above(
    a, q) - above(a, p)) * orient(a, p, q) > 0; }
    // check if [PQ] crosses ray from A

bool inPolygon(vector<pt> p, pt a, bool touch) //
    touch must be false while receiving
{                                                //
    if strictly inside return true, returns false
    when A is on the boundary or outside
     int numCrossings = 0;
     for (int i = 0, n = p.size(); i < n; i++)
     {
          if (onSegment(p[i], p[(i + 1) % n], a))
          {
               touch = true;
               return false;
          }
          numCrossings += crossesRay(a, p[i], p[(i
               + 1) % n]);
     }
     return (numCrossings & 1); // inside if odd
         number of crossings
}
T cosine_formula(T a, T b, T c) { return acos(((a
    * a + b * b - c * c) * 0.5) / (a * b)); }
T circle_circle_Area(pt s1, T r1, pt s2, T r2)
{
     T d = mag(s2 - s1);
     if (d >= r1 + r2)
          return 0.0;
     else if (d <= max(r1, r2) - min(r1, r2))
          return pi * min(r1, r2) * min(r1, r2);
     else
     {
          T ang1 = cosine_formula(d, r1, r2) *
              2.0, ang2 = cosine_formula(d, r2, r1
              ) * 2.0;
          T ar1 = r1 * r1 * ang1 * 0.5 - (0.5 *
              sin(ang1) * r1 * r1);
```

```
        T ar2 = r2 * r2 * ang2 * 0.5 - (0.5 * r2
            * r2 * sin(ang2));
        return ar1 + ar2;
    }
    return -1.0;
}

// convex hull
vector<pt> cvx_hull;
void CVX_hull(vector<pt> pnts)
{
    sort(pnts.begin(), pnts.end()); // sorted
        from the leftmost
    for (int rep = 0; rep < 2; rep++)
    {
        const int szz = cvx_hull.size();
        for (pt u : pnts)
        {
            while ((int)cvx_hull.size() - szz
                >= 2)
            {
                pt A = cvx_hull.end()[-2];
                pt B = cvx_hull.end()[-1];
                if (orient(A, u, B) >= 0) // B
                    is at left (good)
                    break;
                cvx_hull.pop_back();
            }
            cvx_hull.push_back(u);
        }
        cvx_hull.pop_back();
        reverse(pnts.begin(), pnts.end());
    }
```

0.16 Basic DP Variations

```
// knapsack

const int inf = 110;
ll n, bag;
ll arr[110][3];
ll dp[110][100020];

ll f(ll id, ll now)
{
    if (id > n)
        return 0;
    if (dp[id][now] != -1)
        return dp[id][now];

    ll way1 = 0, way2 = 0;
    if (arr[id][1] <= bag - now)
        way1 = arr[id][2] + f(id + 1, now + arr[
            id][1]);
    way2 = f(id + 1, now);

    return dp[id][now] = max(way1, way2);
}

void Solve()
{
    memset(dp, -1, sizeof dp);
    cin >> n >> bag;
    for (int i = 1; i <= n; i++)
        cin >> arr[i][1] >> arr[i][2];
    cout << f(1, 0) << '\n';
}
```

0.17 Matrix Exponentiation

```
const int MXD = 10, mod = 1e9 + 7;
int dim;
ll n;
//
    ---------------------------------------------
    include mod functions ***
struct matrix
{
    ll Mat[MXD][MXD];
};

matrix matMul(matrix A, matrix B)
{ // multiplication in dim^3
    matrix prod;
    for (int row = 1; row <= dim; row++)
    {
        for (int col = 1; col <= dim; col++)
        {
            ll here = 0;
            for (int it = 1; it <= dim; it++)
                here = modAdd(here, modMul(A.
                    Mat[row][it], B.Mat[it][
                    col], mod), mod);
            prod.Mat[row][col] = here;
        }
    }
    return prod;
}

matrix define_Identity()
{ // identity matrix
    matrix Idt;
    for (int row = 1; row <= dim; row++)
        for (int col = 1; col <= dim; col++)
            Idt.Mat[row][col] = (row == col) ?
                1 : 0;
    return Idt;
}

matrix matExpo(matrix A, ll pw)
{ // like binary expo
    matrix res = define_Identity();
    while (pw > 0)
    {
        if (pw & 1)
            res = matMul(res, A);
        A = matMul(A, A);
        pw >>= 1LL;
    }
    return res;
}

matrix base_trans_product(matrix BS, matrix TR)
{
    matrix tmp;
    for (int i = 1; i <= 1; i++)
```

```
    {
        for (int j = 1; j <= dim; j++)
        {
            ll here = 0;
            for (int k = 1; k <= dim; k++)
                here = modAdd(here, modMul(BS.
                    Mat[i][k], TR.Mat[k][j],
                    mod), mod);
            tmp.Mat[i][j] = here;
        }
    }
    return tmp;
}

void process()
{
    matrix build;
    dim = 2;
    build.Mat[1][1] = 0, build.Mat[1][2] = 1;
    build.Mat[2][1] = 1, build.Mat[2][2] = 1;
    matrix nth_power = matExpo(build, n);
    matrix base_matrix;
    base_matrix.Mat[1][1] = 0, base_matrix.Mat
        [1][2] = 1;
    matrix res = base_trans_product(base_matrix,
        nth_power);

    cout << res.Mat[1][1] << '\n';
}

void Solve()
{
    cin >> n;
    process();
}

// in the end, it doesn't even matter
```

0.18 Linear Diophantine

```
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;
}
```