```
#include <bits/stdc++.h>
using namespace std;

#define int long long
#define ull unsigned long long
#define ld long double
#define pb push_back
#define mp make_pair
#define pf push_front
#define all(v) (v).begin(), (v).end()
#define rall(v) (v).rbegin(), (v).rend()
#define sz(x) (int)(x).size()
#define ff first
#define ss second

const int MOD = 1e9 + 7;
const int INF = 1e18;
const ld PI = 3.14159265358979323846;
const ld EPS = 1e-9;
const int N = 3e5 + 9;

// Utility Functions—-----------------------------------
template <typename T>
void read(vector<T> &v)
{
    for (auto &x : v)
        cin >> x;
}

template <typename T>
void show(const T &container)
{
    for (const auto &x : container)
        cout << x << " ";
    cout << "\n";
}

// Depth-First Search (DFS)-------------------------
template <typename T>
void dfs(T node, const vector<vector<T>> &adj,
vector<bool> &vis)
{
    vis[node] = true;
    for (const T &child : adj[node])
    {
        if (!vis[child])
        {
            dfs(child, adj, vis);
        }
    }
}

// Breadth-First Search (BFS)-----------------------
template <typename T>
void bfs(const vector<vector<T>> &adj, T
source)
{
    int N = adj.size();
    queue<T> q;
    vector<bool> vis(N, false);
    vector<int> d(N, -1);

    q.push(source);
    vis[source] = true;
    d[source] = 0;

    while (!q.empty())
    {
        T v = q.front();
        q.pop();
        for (T u : adj[v])
        {
            if (!vis[u])
            {
                vis[u] = true;
                q.push(u);
                d[u] = d[v] + 1;
            }
        }
    }
}

// Binary Indexed Tree (BIT)-------------------------
vector<int> bit(N), arr(N);
int n;

void update(int idx, int value)
{
    for (; idx <= n; idx += idx & -idx)
        bit[idx] += value;
}

int prefixSum(int idx)
{
    int sum = 0;
    for (; idx > 0; idx -= idx & -idx)
```

```
        sum += bit[idx];
    return sum;
}

int query(int l, int r)
{
    return prefixSum(r) - prefixSum(l - 1);
}

// Binary Exponentiation—----------------------------
template <typename T>
T binExp(T a, T b, T m)
{
    T res = 1;
    a %= m;
    while (b > 0)
    {
        if (b & 1)
            res = res * a % m;
        a = a * a % m;
        b >>= 1;
    }
    return res;
}

// Binary Search—-------------------------------------
template <typename T>
int binarySearch(const vector<T> &arr, T key)
{
    int l = -1, r = arr.size();
    while (r - l > 1)
    {
        int m = (l + r) / 2;
        if (key < arr[m])
        {
            r = m;
        }
        else
        {
            l = m;
        }
    }
    return l;
}

// Bit Manipulation Utilities—-----------------------
template <typename T>
T checkKthBit(T x, int k)
{
    return (x >> k) & 1;
}

template <typename T>
void printOnBits(T x)
{
    for (int k = 0; k < sizeof(T) * 8; k++)
    {
        if (checkKthBit(x, k))
        {
            cout << k << ' ';
        }
    }
    cout << '\n';
}

template <typename T>
int countOnBits(T x)
{
    int ans = 0;
    for (int k = 0; k < sizeof(T) * 8; k++)
    {
        if (checkKthBit(x, k))
        {
            ans++;
        }
    }
    return ans;
}

template <typename T>
bool isEven(T x)
{
    return !(x & 1);
}

template <typename T>
T setKthBit(T x, int k)
{
    return x | (1 << k);
}

template <typename T>
T unsetKthBit(T x, int k)
{
    return x & (~(1 << k));
}
```

```
template <typename T>
T toggleKthBit(T x, int k)
{
    return x ^ (1 << k);
}

template <typename T>
bool isPowerOf2(T x)
{
    return x > 0 && (x & (x - 1)) == 0;
}

// Dijkstra's Algorithm—-------------------------------
const int MAXN = 1e5 + 5;
vector<pair<int, int>> g[MAXN];

vector<int> dijkstra(int src, vector<int> &cnt)
{
     priority_queue<pair<int, int>, vector<pair<int,
int>>, greater<>> pq;
    vector<int> dist(n + 1, INF);
    vector<bool> vis(n + 1, false);
    cnt.assign(n + 1, 0);

    dist[src] = 0;
    cnt[src] = 1;
    pq.push({0, src});

    while (!pq.empty())
    {
        auto [curDist, u] = pq.top();
        pq.pop();

        if (vis[u])
            continue;
        vis[u] = true;

        for (auto &edge : g[u])
        {
            int v = edge.first;
            int w = edge.second;
            if (dist[u] + w < dist[v])
            {
                dist[v] = dist[u] + w;
                pq.push({dist[v], v});
                cnt[v] = cnt[u];
            }
            else if (dist[u] + w == dist[v])
            {
                cnt[v] = (cnt[v] + cnt[u]) % MOD;
            }
        }
    }
    return dist;
}

// Disjoint Set Union (DSU)-------------------------
vector<int> par(N), sz(N), rnk(N, 0);

void make_set(int v)
{
    par[v] = v;
    rnk[v] = 0;
    sz[v] = 1;
}

int find_set(int v)
{
    if (v == par[v])
        return v;
    return par[v] = find_set(par[v]);
}

void union_sets(int a, int b)
{
    a = find_set(a);
    b = find_set(b);
    if (a != b)
    {
        if (sz[a] < sz[b])
            swap(a, b);
        par[b] = a;
        sz[a] += sz[b];
    }
}

// Floyd-Warshall Algorithm—-----------------------
void floydWarshall(vector<vector<int>> &d)
{
    int n = d.size();
    const int INF = 1e9;

    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < n; j++)
```

```
        {
            if (d[i][j] == -1)
                d[i][j] = INF;
            if (i == j)
                d[i][j] = 0;
        }
    }

    for (int k = 0; k < n; k++)
    {
        for (int i = 0; i < n; i++)
        {
            for (int j = 0; j < n; j++)
            {
                if (d[i][k] < INF && d[k][j] < INF)
                {
                    d[i][j] = min(d[i][j], d[i][k] + d[k][j]);
                }
            }
        }
    }

    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < n; j++)
        {
            if (d[i][j] == INF)
                d[i][j] = -1;
        }
    }
}

// Generate All Subsets of an Array—----------
template <typename T>
vector<vector<T>> generateSubsets(vector<T> &arr)
{
    int n = arr.size();
    int totalSubsets = 1 << n;
    vector<vector<T>> subsets;

        for (int mask = 0; mask < totalSubsets; mask++)
    {
        vector<T> subset;
        for (int i = 0; i < n; i++)
        {
            if (mask & (1 << i))
            {
                subset.push_back(arr[i]);
            }
        }
        subsets.push_back(subset);
    }
    return subsets;
}

// KMP String Matching Algorithm—----------
vector<int> build_lps(const string &p)
{
    int m = p.size();
    vector<int> lps(m, 0);
    for (int i = 1, j = 0; i < m; i++)
    {
        while (j > 0 && p[i] != p[j])
        {
            j = lps[j - 1];
        }
        if (p[i] == p[j])
        {
            j++;
        }
        lps[i] = j;
    }
    return lps;
}

vector<int> kmp(const string &s, const string &p)
{
    int n = s.size(), m = p.size();
    vector<int> ans;
    vector<int> lps = build_lps(p);
    for (int i = 0, j = 0; i < n; i++)
    {
        while (j > 0 && s[i] != p[j])
        {
            j = lps[j - 1];
        }
        if (s[i] == p[j])
        {
            j++;
        }
        if (j == m)
        {
            ans.push_back(i - m + 1);
```

```
            j = lps[j - 1];
        }
    }
    return ans;
}

// Binary Exponentiation—----------------------------
template <typename T>
T binExp(T a, T b, T m)
{
    T res = 1;
    a %= m;
    while (b > 0)
    {
        if (b & 1)
            res = res * a % m;
        a = a * a % m;
        b >>= 1;
    }
    return res;
}

// GCD Calculation—----------------------------------
template <typename T>
T gcd(T a, T b)
{
    return b == 0 ? a : gcd(b, a % b);
}

// Modular Inverse
template <typename T>
void modInverse(T a, T m)
{
    if (gcd(a, m) != 1)
    {
        cout << "Inverse doesn't exist\n";
    }
    else
    {
        cout << binExp(a, m - 2, m) << "\n";
    }
}

// Segment Tree —-------------------------------------
int arr[N], t[4 * N];
void build(int n, int st, int ed)
{
    if (st == ed)
    {
        t[n] = arr[st];
        return;
    }
    int mid = (st + ed) >> 1;
    int lc = n << 1, rc = lc | 1;
    build(lc, st, mid);
    build(rc, mid + 1, ed);
    t[n] = max(t[lc], t[rc]);
}

void update(int n, int st, int ed, int ind, int val)
{
    if (st > ind || ed < ind)
        return;
    if (st == ed && st == ind)
    {
        t[n] = val;
        return;
    }
    int mid = (st + ed) >> 1;
    int lc = n << 1, rc = lc | 1;
    update(lc, st, mid, ind, val);
    update(rc, mid + 1, ed, ind, val);
    t[n] = max(t[lc], t[rc]);
}

int query(int n, int st, int ed, int l, int r)
{
    if (st > r || ed < l)
        return -INF;
    if (st >= l && ed <= r)
        return t[n];
    int mid = (st + ed) >> 1;
    int lc = n << 1, rc = lc | 1;
    int maxL = query(lc, st, mid, l, r);
    int maxR = query(rc, mid + 1, ed, l, r);
    return max(maxL, maxR);
}

// Segment Tree(Lazy)--------------------------------
int arr[N], t[4 * N], lazy[4 * N];

void build(int n, int st, int ed)
{
    if (st == ed)
    {
        t[n] = arr[st];
```

```
        return;
    }
    int mid = (st + ed) >> 1;
    int lc = n << 1, rc = lc | 1;
    build(lc, st, mid);
    build(rc, mid + 1, ed);
    t[n] = t[lc] + t[rc];
}

void apply_lazy(int n, int st, int ed)
{
    if (lazy[n] != 0)
    {
        t[n] += (ed - st + 1) * lazy[n];
        if (st != ed)
        {
            lazy[n << 1] += lazy[n];
            lazy[(n << 1) | 1] += lazy[n];
        }
        lazy[n] = 0;
    }
}

void update(int n, int st, int ed, int l, int r, int val)
{
    apply_lazy(n, st, ed);
    if (st > r || ed < l)
        return;
    if (st >= l && ed <= r)
    {
        lazy[n] += val;
        apply_lazy(n, st, ed);
        return;
    }
    int mid = (st + ed) >> 1;
    int lc = n << 1, rc = lc | 1;
    update(lc, st, mid, l, r, val);
    update(rc, mid + 1, ed, l, r, val);
    t[n] = t[lc] + t[rc];
}

int query(int n, int st, int ed, int l, int r)
{
    apply_lazy(n, st, ed);
    if (st > r || ed < l)
        return 0;
    if (st >= l && ed <= r)
        return t[n];
    int mid = (st + ed) >> 1;
    int lc = n << 1, rc = lc | 1;
    int leftQuery = query(lc, st, mid, l, r);
    int rightQuery = query(rc, mid + 1, ed, l, r);
    return leftQuery + rightQuery;
}

// Sieve of Eratosthenes—----------------------------
vector<bool> is_prime(N + 1, true);
void sieve()
{
    is_prime[0] = is_prime[1] = false;
    for (int i = 2; i * i <= N; i++)
    {
        if (is_prime[i])
        {
            for (int j = i * i; j <= N; j += i)
            {
                is_prime[j] = false;
            }
        }
    }
}

// Square Root Decomposition—----------------------
vector<int> seg, arr;
int n, len;
void buildSqrtDecomposition()
{
    len = sqrt(n) + 1;
    seg.assign(len, 0);
    for (int i = 0; i < n; ++i)
        seg[i / len] += arr[i];
}

int querySqrt(int l, int r)
{
    int sum = 0;
    for (int i = l; i <= r;)
    {
        if (i % len == 0 && i + len - 1 <= r)
        {
            sum += seg[i / len];
            i += len;
        }
        else
        {
            sum += arr[i];
```

```
            ++i;
        }
    }
    return sum;
}

void updateSqrt(int index, int value)
{
    seg[index / len] += value - arr[index];
    arr[index] = value;
}

// Global Variables—----------------------------------
int arr[N], t[4 * N];
vector<int> adj[N];
int st[N], en[N], dfsarr[N];
int clk = 0;

// Segment Tree and DFS Helpers
void dfs(int u, int par)
{
    clk++;
    st[u] = clk;
    dfsarr[clk] = u;
    for (auto v : adj[u])
    {
        if (v == par)
            continue;
        dfs(v, u);
    }
    en[u] = clk;
}

void build(int n, int st, int ed)
{
    if (st == ed)
    {
        t[n] = arr[dfsarr[st]];
        return;
    }
    int mid = (st + ed) >> 1;
    int lc = n << 1, rc = lc | 1;
    build(lc, st, mid);
    build(rc, mid + 1, ed);
    t[n] = t[lc] + t[rc];
}

void update(int n, int st, int ed, int ind, int val)
{
    if (st > ind || ed < ind)
        return;
    if (st == ed && st == ind)
    {
        t[n] = val;
        return;
    }
    int mid = (st + ed) >> 1;
    int lc = n << 1, rc = lc | 1;
    update(lc, st, mid, ind, val);
    update(rc, mid + 1, ed, ind, val);
    t[n] = t[lc] + t[rc];
}

int query(int n, int st, int ed, int l, int r)
{
    if (st > r || ed < l)
        return 0;
    if (st >= l && ed <= r)
        return t[n];
    int mid = (st + ed) >> 1;
    int lc = n << 1, rc = lc | 1;
     return query(lc, st, mid, l, r) + query(rc, mid +
1, ed, l, r);
}

// Ternary Search—-----------------------------------
double f(double x)
{
    // Define your function here
    return;
}

double ternary_search(double l, double r)
{
    while (r - l > EPS)
    {
        double m1 = l + (r - l) / 3;
        double m2 = r - (r - l) / 3;
        if (f(m1) < f(m2))
            l = m1;
        else
            r = m2;
    }
    return l;
}
```

```
// Topological Sort using DFS—----------------------
vector<int> vis, topsort;

bool dfs_topo(int u)
{
    vis[u] = 1;
    for (int v : adj[u])
    {
        if (vis[v] == 2)
            continue;
        if (vis[v] == 1)
            return false;
        if (!dfs_topo(v))
            return false;
    }
    vis[u] = 2;
    topsort.push_back(u);
    return true;
}

void solve_topo_dfs()
{
    int n, m;
    cin >> n >> m;
    vis.assign(n + 1, 0);
    topsort.clear();
    for (int i = 1; i <= n; i++)
        adj[i].clear();

    for (int i = 0; i < m; i++)
    {
        int u, v;
        cin >> u >> v;
        adj[u].emplace_back(v);
    }

    for (int i = 1; i <= n; i++)
    {
        if (vis[i] == 0)
        {
            if (!dfs_topo(i))
            {
                cout << "IMPOSSIBLE\n";
                return;
            }
        }
    }
    reverse(topsort.begin(), topsort.end());
    for (int it : topsort)
    {
        cout << it << " ";
    }
    cout << "\n";
}

// Topological Sort using BFS—----------------------
vector<int> adj[N];
vector<int> indeg;
vector<int> topsort;

void bfs_topo(int n)
{
    priority_queue<int, vector<int>, greater<int>>
pq;

    for (int i = 1; i <= n; i++)
    {
        if (indeg[i] == 0)
            pq.push(i);
    }

    while (!pq.empty())
    {
        int node = pq.top();
        pq.pop();
        topsort.push_back(node);
        for (auto it : adj[node])
        {
            indeg[it]--;
            if (indeg[it] == 0)
                pq.push(it);
        }
    }
}

void solve_topo_bfs()
{
    int n, m;
    cin >> n >> m;
    indeg.assign(n + 1, 0);
    topsort.clear();
    for (int i = 1; i <= n; i++)
        adj[i].clear();

    for (int i = 0; i < m; i++)
```

```
    {
        int u, v;
        cin >> u >> v;
        adj[u].emplace_back(v);
        indeg[v]++;
    }

    bfs_topo(n);

    if ((int)topsort.size() < n)
    {
        cout << "IMPOSSIBLE\n";
    }
    else
    {
        for (int it : topsort)
        {
            cout << it << " ";
        }
        cout << "\n";
    }
}

// Trie—----------------------------------------------------
struct Node
{
    Node *links[26] = {nullptr};
    int cntEndWith = 0;
    int cntPrefix = 0;

    bool containsKey(char ch)
    {
        return links[ch - 'a'] != nullptr;
    }

    void put(char ch, Node *node)
    {
        links[ch - 'a'] = node;
    }

    Node *get(char ch)
    {
        return links[ch - 'a'];
    }

    void increaseEnd()
    {
        cntEndWith++;
    }

    void increasePrefix()
    {
        cntPrefix++;
    }

    void deleteEnd()
    {
        cntEndWith--;
    }

    void reducePrefix()
    {
        cntPrefix--;
    }
};

Node *root = new Node();
void deleteTrie(Node *node)
{
    for (int i = 0; i < 26; ++i)
    {
        if (node->links[i])
        {
            deleteTrie(node->links[i]);
        }
    }
    delete node;
}

void insert(const string &word)
{
    Node *node = root;
    for (char ch : word)
    {
        if (!node->containsKey(ch))
        {
            node->put(ch, new Node());
        }
        node = node->get(ch);
        node->increasePrefix();
    }
    node->increaseEnd();
}

bool search(const string &word)
{
```

```
    Node *node = root;
    for (char ch : word)
    {
        if (!node->containsKey(ch))
        {
            return false;
        }
        node = node->get(ch);
    }
    return node->cntEndWith > 0;
}

bool startsWith(const string &prefix)
{
    Node *node = root;
    for (char ch : prefix)
    {
        if (!node->containsKey(ch))
        {
            return false;
        }
        node = node->get(ch);
    }
    return true;
}

int countWordsEqualTo(const string &word)
{
    Node *node = root;
    for (char ch : word)
    {
        if (!node->containsKey(ch))
        {
            return 0;
        }
        node = node->get(ch);
    }
    return node->cntEndWith;
}

int countWordsStartingWith(const string &prefix)
{
    Node *node = root;
    for (char ch : prefix)
    {
        if (!node->containsKey(ch))
        {
            return 0;
        }
        node = node->get(ch);
    }
    return node->cntPrefix;
}

void erase(const string &word)
{
    Node *node = root;
    for (char ch : word)
    {
        if (node->containsKey(ch))
        {
            node = node->get(ch);
            node->reducePrefix();
        }
        else
        {
            return;
        }
    }
    node->deleteEnd();
}

//Cycle detection algorithm—------------------------
const int N = 1e5 + 5;
vector<int> adj[N];
int vis[N], col[N], pr[N];

void dfs(int node, int par) {
    if (vis[node] == 2) {
        return;
    }
    if (vis[node] == 1) {
        col[node] = 1;
        int x = par;
        while (x != node) {
            col[x] = 1;
            x = pr[x];
        }
        return;
    }
    pr[node] = par;
    vis[node] = 1;
    for (auto child : adj[node]) {
        if (pr[node] == child) {
            continue;
```

```
        }
        dfs(child, node);
    }
    vis[node] = 2;
}

// Extended Euclid Algorithm----------------------
int gcd(int a, int b, int &x, int &y)
{
    if (b == 0)
    {
        x = 1;
        y = 0;
        return a;
    }
    int x1, y1;
    int d = gcd(b, a % b, x1, y1);
    x = y1;
    y = x1 - y1 * (a / b);
    return d;
}

// Shift solution for linear Diophantine equations
void shift_solution(int &x, int &y, int a, int b, int
cnt)
{
    x += b * cnt;
    y -= a * cnt;
}

//nCr—-----------------------------------------------
int fact[N], invfact[N];
// Precomputes factorials and modular inverses
void precompute()
{
    fact[0] = invfact[0] = 1;
    for (int i = 1; i < N; i++)
    {
        fact[i] = fact[i - 1] * i % M;
    }
    invfact[N - 1] = binExp(fact[N - 1], M - 2, M);
    for (int i = N - 2; i >= 0; i--)
    {
        invfact[i] = invfact[i + 1] * (i + 1) % M;
    }
}
// Computes nCr % M
int nCr(int n, int r)
{
    if (r < 0 || n < 0 || r > n)
        return 0;
     return fact[n] * invfact[r] % M * invfact[n - r] %
M;
}

// Matrix Exponentiation—----------------------------
vector<vector<int>>multiply(
const vector<vector<int>>&a,
const vector<vector<int>> &b)
{
    int size = a.size();
    vector<vector<int>>
    ans(size, vector<int>(size, 0));
    for (int i = 0; i < size; i++)
    {
        for (int j = 0; j < size; j++)
        {
            for (int k = 0; k < size; k++)
            {
                ans[i][j] = (ans[i][j] + (a[i][k] * b[k][j])
                                % M) % M;
            }
        }
    }
    return ans;
}

vector<vector<int>>
matpow(vector<vector<int>> a, int exp)
{
    int size = a.size();
    vector<vector<int>>
   ans(size, vector<int>(size, 0));

    for (int i = 0; i < size; i++)  ans[i][i] = 1;
    while (exp > 0)
    {
        if (exp & 1)
            ans = multiply(ans, a);
            a = multiply(a, a);
            exp >>= 1;
    }
    return ans;
}
```

```
// Function to initialize and calculate matrix exponentiation
vector<vector<int>> calmatpow(int n, int size)
{
    vector<vector<int>>
a(size, vector<int>(size, 0));
    for (int i = 0; i < size; i++)  a[0][i] = 1;
    for (int i = 1; i < size; i++)  a[i][i - 1] = 1;
    return matpow(a, n);
}

// Lowest Common Ancestor (LCA) Template—----------------------------------------
const int LOG = 20; // Maxi depth for binary lifting
vector<int> adj[N];
vector<vector<int>> anc(N, vector<int>(LOG, -1));
vector<int> depth(N, 0);

// DFS to initialize ancestor table and depth array
void dfs(int u, int par)
{
    if (par != -1)
    {
        depth[u] = depth[par] + 1;
        anc[u][0] = par;
        for (int i = 1; i < LOG; i++)
        {
            int v = anc[u][i - 1];
            if (v == -1)
                break;
            anc[u][i] = anc[v][i - 1];
        }
    }
    for (int v : adj[u])
    {
        if (v == par)
            continue;
        dfs(v, u);
    }
}

// Function to find the k-th ancestor of a node
int getAncestor(int u, int k)
{
    for (int i = 0; i < LOG; i++)
    {
        if (k & (1 << i))
        {
            u = anc[u][i];
            if (u == -1)
                return -1;
        }
    }
    return u;
}

// Function to find the LCA of two nodes
int lca(int u, int v)
{
    if (depth[u] > depth[v])
        u = getAncestor(u, depth[u] - depth[v]);
    if (depth[v] > depth[u])
        v = getAncestor(v, depth[v] - depth[u]);

    if (u == v)
        return u;

    for (int i = LOG - 1; i >= 0; i--)
    {
        if (anc[u][i] != anc[v][i])
        {
            u = anc[u][i];
            v = anc[v][i];
        }
    }
    return anc[u][0];
}

// Example usage function
void solve()
{
    int n, m;
    cin >> n >> m;
    for (int i = 1; i < n; i++)
    {
        int boss;
        cin >> boss;
        adj[i].push_back(boss);
        adj[boss].push_back(i);
    }
    dfs(0, -1);

    for (int i = 0; i < m; i++)
```

```
    {
        int u, v;
        cin >> u >> v;
        cout << lca(u, v) << "\n";
    }
}
```