```cpp
#include <algorithm>
#include <cmath>
#include <cstring>
#include <functional>
#include <iostream>
#include <map>
#include <numeric>
#include <queue>
#include <random>
#include <set>
#include <stack>
#include <vector>
```

Table of Contents

```cpp
/**
 *    强连通分量缩点 (SCC)
 *    功能：Tarjan 算法用于求解图的强连通分量，将强连通分量缩点构建新的有向无环图
 *    链接：https://codeforces.com/contest/1835/submission/210147209
 *    日期：2023-06-18
**/
struct SCC {
    int n;                                  // 图中节点数
    std::vector<std::vector<int>> adj; // 邻接表，用于存储图的边
    std::vector<int> stk;              // 栈，用于 Tarjan 算法
    std::vector<int> dfn, low, bel; // dfn：节点访问时间，low：最小可回溯时间，bel：节点所属的 SCC 编号
    int cur, cnt;                      // cur：时间戳，cnt：SCC 的数量
    // 默认构造函数
    SCC() {}
    // 带参构造函数并初始化
    SCC(int n) {
        init(n);
    }
    // 初始化函数
```

```
    void init(int n) {
        this->n = n;
        adj.assign(n, {});              // 初始化邻接表
        dfn.assign(n, -1);              // 初始化访问时间
        low.resize(n);                  // 初始化最小回溯时间
        bel.assign(n, -1);              // 初始化节点所属的 SCC 编号
        stk.clear();                    // 清空栈
        cur = cnt = 0;                  // 重置时间戳和 SCC 计数器
    }
    // 添加有向边 u -> v
    void addEdge(int u, int v) {
        adj[u].push_back(v);
    }
    // 深度优先搜索 (DFS) 寻找强连通分量
    void dfs(int x) {
        dfn[x] = low[x] = cur++;   // 设定访问时间和初始最小回溯时间
        stk.push_back(x);               // 将节点压入栈中
        for (auto y : adj[x]) {
            if (dfn[y] == -1) {     // 未访问节点, 继续 DFS
                dfs(y);
                low[x] = std::min(low[x], low[y]); // 更新最小回溯时间
            } else if (bel[y] == -1) { // y 还在栈中, 说明是一个回边
                low[x] = std::min(low[x], dfn[y]);
            }
        }
        // 如果当前节点是 SCC 的根节点, 开始出栈形成 SCC
        if (dfn[x] == low[x]) {
            int y;
            do {
                y = stk.back();
                bel[y] = cnt;       // 将节点 y 标记为属于当前 SCC
                stk.pop_back();
            } while (y != x);
            cnt++;                          // 增加 SCC 计数
        }
    }
    // 计算强连通分量
    std::vector<int> work() {
        for (int i = 0; i < n; i++) {
            if (dfn[i] == -1) {     // 未访问的节点, 执行 DFS
                dfs(i);
            }
        }
        return bel;                         // 返回每个节点所属的 SCC
    }
};
/**
```

```
 *    2-SAT 问题的求解 (TwoSat)
 *    功能：判定并解决 2-SAT 问题（布尔公式可满足性问题）
 *    链接：https://atcoder.jp/contests/arc161/submissions/46031530
 *    日期：2023-09-29
**/
struct TwoSat {
    int n;                                          // 变量数量
    std::vector<std::vector<int>> e; // 邻接表, 存储隐含图
    std::vector<bool> ans;                  // 存储满足解的布尔值
    // 构造函数, 初始化邻接表和解数组
    TwoSat(int n) : n(n), e(2 * n), ans(n) {}
    // 添加一个子句 (u 或 v), 使用 f 和 g 表示 u 和 v 的真假值
    void addClause(int u, bool f, int v, bool g) {
        e[2 * u + !f].push_back(2 * v + g);
        e[2 * v + !g].push_back(2 * u + f);
    }
    bool satisfiable() {
        std::vector<int> id(2 * n, -1), dfn(2 * n, -1), low(2 * n, -1);
        std::vector<int> stk;
        int now = 0, cnt = 0;
        std::function<void(int)> tarjan = [&](int u) {
            stk.push_back(u);
            dfn[u] = low[u] = now++;
            for (auto v : e[u]) {
                if (dfn[v] == -1) {
                    tarjan(v);
                    low[u] = std::min(low[u], low[v]);
                } else if (id[v] == -1) {
                    low[u] = std::min(low[u], dfn[v]);
                }
            }
            if (dfn[u] == low[u]) {
                int v;
                do {
                    v = stk.back();
                    stk.pop_back();
                    id[v] = cnt;
                } while (v != u);
                ++cnt;
            }
        };
        for (int i = 0; i < 2 * n; ++i) if (dfn[i] == -1) tarjan(i);
        for (int i = 0; i < n; ++i) {
            if (id[2 * i] == id[2 * i + 1]) return false;
            ans[i] = id[2 * i] > id[2 * i + 1];
        }
        return true;
```

```
    }
    std::vector<bool> answer() { return ans; }
};
/**
 *   扩展欧几里得算法 (exgcd)
 *   功能：计算 ax + by = gcd(a, b) 的整数解 (x, y)
 *   链接：https://codeforces.com/contest/1993/submission/275110715
 *   日期：2024-08-07
**/
i64 exgcd(i64 a, i64 b, i64 &x, i64 &y) {
    if (b == 0) {              // 基础情况：b 为 0 时
        x = 1;                 // 设 x = 1，y = 0，此时 a*x + b*y = a
        y = 0;
        return a;              // 返回 gcd(a, 0) = a
    }
    i64 g = exgcd(b, a % b, y, x); // 递归调用，求 gcd(b, a % b) 的解
    y -= a / b * x;                 // 根据递推公式更新 y
    return g;                       // 返回 gcd(a, b)
}
/**
 *   线性同余方程解法
 *   功能：求解 ax + b ≡ 0 (mod m) 的整数解
 *   参数：a，b 和模数 m
**/
std::pair<i64, i64> sol(i64 a, i64 b, i64 m) {
    assert(m > 0);             // 确保模数 m > 0
    b *= -1;                   // 转化方程：ax ≡ -b (mod m)
    i64 x, y;
    i64 g = exgcd(a, m, x, y); // 求解扩展欧几里得方程得到 x 和 y
    if (g < 0) {               // 如果 g < 0，转换为正数
        g *= -1;
        x *= -1;
        y *= -1;
    }
    if (b % g != 0) {          // 如果 b 不能整除 gcd(a, m)，则无解
        return {-1, -1};
    }
    x = x * (b / g) % (m / g); // 求解方程，简化 x
    if (x < 0) {               // 保证 x 为非负
        x += m / g;
    }
    return {x, m / g};         // 返回 x 和模数 m/g
}
/**
 *   扩展欧几里得算法 (exgcd) 简化版本
 *   功能：计算 gcd(a, b) 和方程的解 (x, y) 满足 ax + by = gcd(a, b)
 *   链接：https://qoj.ac/submission/165983
```

```
 *    日期：2023-09-05
**/
std::array<i64, 3> exgcd(i64 a, i64 b) {
    if (!b) {                    // 基础情况
        return {a, 1, 0};   // 返回 gcd(a, b), x = 1, y = 0
    }
    auto [g, x, y] = exgcd(b, a % b); // 递归调用，解 gcd(b, a % b)
    return {g, y, x - a / b * y};        // 返回 g, x, y 更新
}
/**
 *    马拉车算法 (Manacher, with string)
 *    功能：求解字符串中所有回文子串的长度，能够在 O(n) 时间内完成。
 *    链接：https://qoj.ac/submission/380047
 *    日期：2024-04-06 和 2024-04-09 (模板)
**/
std::vector<int> manacher(std::string s) {
    // 为了处理偶数长度的回文串，将原始字符串转换为带有分隔符 '#' 的新字符串
    std::string t = "#";
    for (auto c : s) {
        t += c;
        t += '#';
    }
    int n = t.size();
    std::vector<int> r(n); // r[i] 表示以 i 为中心的最长回文半径 (包含分隔符)
    for (int i = 0, j = 0; i < n; i++) {
        // 如果 i 在以 j 为中心的回文串范围内，初始化 r[i]
        if (2 * j - i >= 0 && j + r[j] > i) {
            r[i] = std::min(r[2 * j - i], j + r[j] - i);
        }
        // 尝试扩展 r[i]，即找到以 i 为中心的最大回文长度
        while (i - r[i] >= 0 && i + r[i] < n && t[i - r[i]] == t[i + r[i]]) {
            r[i] += 1;
        }
        // 更新中心位置 j，以保证 j + r[j] 尽可能大
        if (i + r[i] > j + r[j]) {
            j = i;
        }
    }
    return r; // 返回半径数组 r，其中 r[i] - 1 是原始字符串中回文串的实际长度
}
/**
 *    前缀函数 (KMP)
 *    功能：计算字符串的前缀函数，用于模式匹配，能有效预处理字符串中的重复前缀。
 *    链接：https://qoj.ac/submission/253754
 *    日期：2023-11-17 和 2024-04-09 (模板)
**/
std::vector<int> kmp(std::string s) {
```

```
    int n = s.size();
    std::vector<int> f(n + 1); // f[i] 表示 s[0:i-1] 的最长相同前后缀长度
    for (int i = 1, j = 0; i < n; i++) {
        // 如果当前字符不匹配，则回退到上一个可能的匹配位置
        while (j && s[i] != s[j]) {
            j = f[j];
        }
        // 如果字符匹配，匹配长度 j+1
        j += (s[i] == s[j]);
        f[i + 1] = j; // 更新前缀函数值
    }
    return f; // 返回前缀函数数组，f[i+1] 是前 i+1 个字符的最长相同前后缀
}
/**
 *   状压 RMQ（Range Minimum Query, RMQ）
 *   功能：在静态数组上快速求解任意区间的最小值，使用状态压缩优化，适合高效查询场景。
 *   链接：https://atcoder.jp/contests/joi2022ho/submissions/39351739
 *   日期：2023-03-02、2023-09-04 和 2024-08-07
**/
template<class T, class Cmp = std::less<T>>
struct RMQ {
    const Cmp cmp = Cmp();                          // 比较函数，默认为小于操作
    static constexpr unsigned B = 64;           // 每个块的大小（用于状态压缩）
    using u64 = unsigned long long;
    int n;                                          // 数组大小
    std::vector<std::vector<T>> a;              // 稀疏表，存储每个区间块的最小值
    std::vector<T> pre, suf, ini;             // pre：前缀最小值，suf：后缀最小值，ini：初始值数
组
    std::vector<u64> stk;                          // 用于状态压缩的栈
    // 默认构造函数
    RMQ() {}
    // 带参构造函数，调用初始化函数
    RMQ(const std::vector<T> &v) {
        init(v);
    }
    // 初始化函数，使用稀疏表和状态压缩来预处理最小值
    void init(const std::vector<T> &v) {
        n = v.size();
        pre = suf = ini = v;
        stk.resize(n);
        if (!n) {
            return;
        }
        const int M = (n - 1) / B + 1;          // 计算需要的块数
        const int lg = std::__lg(M);            // 计算块数的对数用于稀疏表深度
        a.assign(lg + 1, std::vector<T>(M));
        // 初始化每个块的最小值
```

```
        for (int i = 0; i < M; i++) {
            a[0][i] = v[i * B];
            for (int j = 1; j < B && i * B + j < n; j++) {
                a[0][i] = std::min(a[0][i], v[i * B + j], cmp);
            }
        }
        // 计算前缀最小值和后缀最小值
        for (int i = 1; i < n; i++) {
            if (i % B) {
                pre[i] = std::min(pre[i], pre[i - 1], cmp);
            }
        }
        for (int i = n - 2; i >= 0; i--) {
            if (i % B != B - 1) {
                suf[i] = std::min(suf[i], suf[i + 1], cmp);
            }
        }
        // 构建稀疏表
        for (int j = 0; j < lg; j++) {
            for (int i = 0; i + (2 << j) <= M; i++) {
                a[j + 1][i] = std::min(a[j][i], a[j][i + (1 << j)], cmp);
            }
        }
        // 状态压缩部分
        for (int i = 0; i < M; i++) {
            const int l = i * B;
            const int r = std::min(1U * n, l + B);
            u64 s = 0;
            for (int j = l; j < r; j++) {
                while (s && cmp(v[j], v[std::__lg(s) + l])) {
                    s ^= 1ULL << std::__lg(s);
                }
                s |= 1ULL << (j - l);
                stk[j] = s;
            }
        }
    }
    // 查询区间 [l, r) 的最小值
    T operator()(int l, int r) {
        if (l / B != (r - 1) / B) {
            T ans = std::min(suf[l], pre[r - 1], cmp);
            l = l / B + 1;
            r = r / B;
            if (l < r) {
                int k = std::__lg(r - l);
                ans = std::min({ans, a[k][l], a[k][r - (1 << k)]}, cmp);
            }
```

```
                return ans;
            } else {
                int x = B * (l / B);
                return ini[__builtin_ctzll(stk[r - 1] >> (l - x)) + l];
            }
        }
};
/**
 *   树状数组 （Fenwick Tree 新版）
 *   功能：高效支持前缀和与区间和查询，适合动态更新的场景。
 *   链接：https://codeforces.com/contest/1915/submission/239262801
 *   日期：2023-12-28
**/
template <typename T>
struct Fenwick {
    int n;                          // 数组大小
    std::vector<T> a;               // 存储树状数组的值
    // 构造函数，初始化树状数组
    Fenwick(int n_ = 0) {
        init(n_);
    }
    // 初始化函数，分配数组大小并清零
    void init(int n_) {
        n = n_;
        a.assign(n, T{});           // 将 a 初始化为大小为 n 的数组，默认值为 T{}
    }
    // 单点更新：在索引 x 位置增加值 v
    void add(int x, const T &v) {
        for (int i = x + 1; i <= n; i += i & -i) {
            a[i - 1] = a[i - 1] + v; // 更新树状数组
        }
    }
    // 计算前缀和 [0, x) 的和
    T sum(int x) {
        T ans{};
        for (int i = x; i > 0; i -= i & -i) {
            ans = ans + a[i - 1];
        }
        return ans;
    }
    // 计算区间和 [l, r) 的和
    T rangeSum(int l, int r) {
        return sum(r) - sum(l);
    }
    // 查找使得前缀和不超过 k 的最大索引
    int select(const T &k) {
        int x = 0;
```

```
        T cur{};
        for (int i = 1 << std::__lg(n); i; i /= 2) {
            if (x + i <= n && cur + a[x + i - 1] <= k) {
                x += i;
                cur = cur + a[x - 1];
            }
        }
        return x;
    }
};
/**
 *   并查集（Disjoint Set Union, DSU）
 *   功能：维护元素集合并支持合并操作与连接查询，用于动态连通性问题。
 *   链接：https://ac.nowcoder.com/acm/contest/view-submission?submissionId=63239142
 *   日期：2023-08-04
**/
struct DSU {
    std::vector<int> f, siz;        // f：父节点数组，siz：每个集合的大小
    // 默认构造函数
    DSU() {}
    // 带参构造函数，初始化并查集
    DSU(int n) {
        init(n);
    }
    // 初始化函数，n 个元素初始状态为各自独立的集合
    void init(int n) {
        f.resize(n);
        std::iota(f.begin(), f.end(), 0); // 初始化父节点为自身
        siz.assign(n, 1);                 // 初始每个集合的大小为 1
    }
    // 路径压缩的查找操作，返回 x 所属的根节点
    int find(int x) {
        while (x != f[x]) {
            x = f[x] = f[f[x]];           // 路径压缩
        }
        return x;
    }
    // 判断两个元素是否在同一集合
    bool same(int x, int y) {
        return find(x) == find(y);
    }
    // 合并 x 和 y 所在的集合，按大小合并，返回合并是否成功
    bool merge(int x, int y) {
        x = find(x);
        y = find(y);
        if (x == y) {
            return false;
```

```
		}
		siz[x] += siz[y];				// 合并集合大小
		f[y] = x;					// 将 y 的根连接到 x
		return true;
	}
	// 获取 x 所在集合的大小
	int size(int x) {
		return siz[find(x)];
	}
};
/**
 *	最大流（MaxFlow 新版）
 *	功能：用于计算图中的最大流量，基于分层网络的增广路径算法实现（Dinic's Algorithm）。
 *	链接：https://ac.nowcoder.com/acm/contest/view-submission?submissionId=62915815
 *	日期：2023-07-21
**/
constexpr int inf = 1E9; // 表示无穷大
template<class T>
struct MaxFlow {
	struct _Edge {
		int to;				// 边的终点
		T cap;				// 边的容量
		_Edge(int to, T cap) : to(to), cap(cap) {}
	};
	int n;					// 节点数量
	std::vector<_Edge> e;			// 存储所有边
	std::vector<std::vector<int>> g; // 邻接表，每个节点的出边索引
	std::vector<int> cur, h;		// cur：当前弧优化；h：分层网络中的高度
	// 构造函数
	MaxFlow() {}
	MaxFlow(int n) {
		init(n);
	}
	// 初始化函数
	void init(int n) {
		this->n = n;
		e.clear();
		g.assign(n, {});
		cur.resize(n);
		h.resize(n);
	}
	// 构建分层网络的 BFS 函数
	bool bfs(int s, int t) {
		h.assign(n, -1);
		std::queue<int> que;
		h[s] = 0;
		que.push(s);
```

```
    while (!que.empty()) {
        const int u = que.front();
        que.pop();
        for (int i : g[u]) {
            auto [v, c] = e[i];
            if (c > 0 && h[v] == -1) {
                h[v] = h[u] + 1;
                if (v == t) {
                    return true;          // 如果找到增广路径则返回 true
                }
                que.push(v);
            }
        }
    }
    return false;                         // 无增广路径
}
// 增广路径的 DFS 函数
T dfs(int u, int t, T f) {
    if (u == t) {
        return f;                         // 到达汇点，返回当前流量
    }
    auto r = f;
    for (int &i = cur[u]; i < int(g[u].size()); ++i) {
        const int j = g[u][i];
        auto [v, c] = e[j];
        if (c > 0 && h[v] == h[u] + 1) {
            auto a = dfs(v, t, std::min(r, c));
            e[j].cap -= a;
            e[j ^ 1].cap += a;            // 更新反向边容量
            r -= a;
            if (r == 0) {
                return f;
            }
        }
    }
    return f - r;
}
// 添加边函数，建立有向图，支持双向边
void addEdge(int u, int v, T c) {
    g[u].push_back(e.size());
    e.emplace_back(v, c);
    g[v].push_back(e.size());
    e.emplace_back(u, 0);                 // 反向边初始容量为 0
}
// 主函数，返回从 s 到 t 的最大流
T flow(int s, int t) {
    T ans = 0;
```

```
        while (bfs(s, t)) {                        // 当存在增广路径时，继续增广
            cur.assign(n, 0);
            ans += dfs(s, t, std::numeric_limits<T>::max());
        }
        return ans;
    }
    // 获取最小割，返回哪些节点属于源点所在集合
    std::vector<bool> minCut() {
        std::vector<bool> c(n);
        for (int i = 0; i < n; i++) {
            c[i] = (h[i] != -1);
        }
        return c;
    }
    // 存储边的结构体（包含流量信息）
    struct Edge {
        int from;
        int to;
        T cap;
        T flow;
    };
    // 获取边的详细信息
    std::vector<Edge> edges() {
        std::vector<Edge> a;
        for (int i = 0; i < e.size(); i += 2) {
            Edge x;
            x.from = e[i + 1].to;
            x.to = e[i].to;
            x.cap = e[i].cap + e[i + 1].cap;
            x.flow = e[i + 1].cap;
            a.push_back(x);
        }
        return a;
    }
};
/**
 *   费用流（MinCostFlow 新版）
 *   功能：用于计算最小费用最大流，适用于带费用的流网络问题。
 *   链接：https://qoj.ac/submission/244680
 *   日期：2023-11-09
**/
template<class T>
struct MinCostFlow {
    struct _Edge {
        int to;                              // 边的终点
        T cap;                               // 边的容量
        T cost;                              // 边的费用
```

```
        _Edge(int to_, T cap_, T cost_) : to(to_), cap(cap_), cost(cost_) {}
    };
    int n;                                  // 节点数量
    std::vector<_Edge> e;                   // 存储所有边
    std::vector<std::vector<int>> g; // 邻接表，每个节点的出边索引
    std::vector<T> h, dis;                  // h：势能；dis：最短距离
    std::vector<int> pre;                   // 前驱边
    // 使用 Dijkstra 算法构建最短路，判断是否有增广路径
    bool dijkstra(int s, int t) {
        dis.assign(n, std::numeric_limits<T>::max());
        pre.assign(n, -1);
        std::priority_queue<std::pair<T, int>, std::vector<std::pair<T, int>>, std::greater<std::pair<T,
int>>> que;
        dis[s] = 0;
        que.emplace(0, s);
        while (!que.empty()) {
            T d = que.top().first;
            int u = que.top().second;
            que.pop();
            if (dis[u] != d) {
                continue;
            }
            for (int i : g[u]) {
                int v = e[i].to;
                T cap = e[i].cap;
                T cost = e[i].cost;
                if (cap > 0 && dis[v] > d + h[u] - h[v] + cost) {
                    dis[v] = d + h[u] - h[v] + cost;
                    pre[v] = i;
                    que.emplace(dis[v], v);
                }
            }
        }
        return dis[t] != std::numeric_limits<T>::max();
    }
    // 默认构造函数
    MinCostFlow() {}
    // 带参构造函数
    MinCostFlow(int n_) {
        init(n_);
    }
    // 初始化函数
    void init(int n_) {
        n = n_;
        e.clear();
        g.assign(n, {});
    }
```

```
// 添加边函数，建立有向图，包含容量和费用
void addEdge(int u, int v, T cap, T cost) {
    g[u].push_back(e.size());
    e.emplace_back(v, cap, cost);
    g[v].push_back(e.size());
    e.emplace_back(u, 0, -cost); // 反向边容量为 0，费用为负
}
// 主函数，返回从 s 到 t 的最大流和最小费用
std::pair<T, T> flow(int s, int t) {
    T flow = 0;
    T cost = 0;
    h.assign(n, 0);
    while (dijkstra(s, t)) {            // 当存在增广路径时，继续增广
        for (int i = 0; i < n; ++i) {
            h[i] += dis[i];
        }
        T aug = std::numeric_limits<int>::max();
        for (int i = t; i != s; i = e[pre[i] ^ 1].to) {
            aug = std::min(aug, e[pre[i]].cap);
        }
        for (int i = t; i != s; i = e[pre[i] ^ 1].to) {
            e[pre[i]].cap -= aug;
            e[pre[i] ^ 1].cap += aug;
        }
        flow += aug;
        cost += aug * h[t];
    }
    return std::make_pair(flow, cost);
}
// 存储边的结构体（包含流量和费用信息）
struct Edge {
    int from;
    int to;
    T cap;
    T cost;
    T flow;
};
// 获取边的详细信息
std::vector<Edge> edges() {
    std::vector<Edge> a;
    for (int i = 0; i < e.size(); i += 2) {
        Edge x;
        x.from = e[i + 1].to;
        x.to = e[i].to;
        x.cap = e[i].cap + e[i + 1].cap;
        x.cost = e[i].cost;
        x.flow = e[i + 1].cap;
```

```
			a.push_back(x);
		}
		return a;
	}
};
/**	离线 LCA 算法（LCA Offline）	**/
class LCA {
	int numOfQuery;
	std::vector<std::vector<int>> adj;
	std::vector<std::vector<std::pair<int,int>>> query;
public :
	LCA (int n){
		numOfQuery = 0;
		adj.resize(n);
		query.resize(n);
	}
	void addEdge (int x,int y) {
		adj[x].push_back(y);
		adj[y].push_back(x);
	}
	void addQuery(int x,int y,int z){
		query[x].push_back({y,z});
		query[y].push_back({x,z});
		numOfQuery++;
	}
	std::vector<int> Tarjan (int s){
		std::vector<int> ans(numOfQuery);
		std::vector<int> p(adj.size());
		std::vector<bool> vis(adj.size());
		std::iota(p.begin(),p.end(),0);
		auto find = [&](auto self,int x) -> int {
			return p[x] == x ? x : p[x] = self(self,p[x]);
		};
		auto dfs = [&](auto self,int u) -> void {
			vis[u] = 1;
			for(auto t : adj[u]){
				if(vis[t])continue;
				self(self,t);
				p[t] = u;
			}
			for(auto [x,y] : query[u]){
				if(vis[x]){
					ans[y] = find(find,x);
				}
			}
		};
		dfs(dfs,s);
```

```
        return ans;
    }
};
int main(){
    int n;
    std::cin>>n;
    std::vector<std::vector<int>> adj(n);
    for(int i = 0;i < n - 1;i++){
        int x,y;
        std::cin>>x>>y;
        adj[x].push_back(y);
    }
    return 0;
}
```

```
/**     树哈希算法（Tree Hashing）    **/
//
// Created by William on 2024/10/30.
//
using ull = unsigned long long;
const ull mask = std::mt19937_64(time(nullptr))();
ull shift(ull x) {
    x ^= mask;
    x ^= x << 13;
    x ^= x >> 7;
    x ^= x << 17;
    x ^= mask;
    return x;
}
std::pair<int,int> getroot(std::vector<std::vector<int>>& adj){
    std::pair<int,int> res = {-1,-1};
    int mn = adj.size();
    std::vector<int> sz(adj.size());
    std::function<void(int,int)> dfs = [&](int u,int v) -> void {
        sz[u] = 1;
        int mx = 0;
        for(auto t : adj[u]){
            if(t == v)continue;
            dfs(t,u);
            sz[u] += sz[t];
            mx = std::max(sz[t],mx);
        }
        mx = std::max(mx,(int)adj.size() - sz[u]);
        if(mx < mn){
            res = {u,-1};
            mn = mx;
        }
        else if(mx == mn){
```

```
            res = {res.first,u};
        }
    };
    dfs(0,-1);
//    for(int i = 0;i < adj.size();i++){
//        std::cout<<sz[i]<<" \n"[i == adj.size() - 1];
//    }
    return res;
}
int main(){
    std::ios::sync_with_stdio(false);
    std::cin.tie(nullptr);
    std::cout.tie(nullptr);
    int n;
    std::map<ull,int> map;
    std::cin>>n;
    for(int i = 0;i < n;i++){
        int m;
        std::cin>>m;
        std::vector<std::vector<int>> adj(m);
        for(int j = 0;j < m;j++){
            int p;
            std::cin>>p;
            p--;
            if(p >= 0){
                adj[p].push_back(j);
                adj[j].push_back(p);
            }
        }
        auto [rt1,rt2] = getroot(adj);
        std::function<ull(int,int)> cal = [&](int u,int v) -> ull {
            ull res = 1;
            for(auto t : adj[u]){
                if(t == v)continue;
                res += shift(cal(t,u));
            }
            return res;
        };
        ull hash = 0;
        if(rt1 != -1){
            hash = cal(rt1,-1);
        }
        if(rt2 != -1){
            hash = std::max(hash,cal(rt2,-1));
        }
//        std::cout<<rt1<<" "<<rt2<<"\n";
//        std::cout<<"hash:"<<hash<<"\n";
```

```
            if(map.count(hash) == 0){
                map[hash] = i + 1;
            }
            std::cout<<map[hash]<<"\n";
        }
        return 0;
}
/**     扫描线算法（Sweep Line）
 *      功能：解决多边形并集面积、重叠区间等二维平面问题的经典算法。
 *      日期：2023-11-08
**/
//
//  main.cpp
//  sweep line
//
//  Created by William on 2023/11/8.
//
struct Sweep_Line {
public :
        struct Node {
            int l,r;
            long long sum;
            int len;
        };
        std::vector<Node> tr;
        std::vector<int> X;
        struct line{
            int x1,x2,y,mark;
        };
        std::vector<line> lines;
        void addLine(int x1,int y1,int x2,int y2){
            X.push_back(x1);
            X.push_back(x2);
            if(y1 > y2)std::swap(y1,y2);
            lines.push_back({x1,x2,y1,1});
            lines.push_back({x1,x2,y2,-1});
        }
        void build(int u,int l,int r){
            tr[u].l = l,tr[u].r = r;
            tr[u].sum = tr[u].len = 0;
            if(l == r)return;
            int mid = l + r >> 1;
            build(u << 1,l,mid);
            build(u << 1 | 1,mid + 1,r);
        }
        void pushup(int u){
            if(tr[u].sum){
```

```
            tr[u].len = X[tr[u].r + 1] - X[tr[u].l];
        }
        else {
            tr[u].len = tr[u << 1].len + tr[u << 1 | 1].len;
        }
    }
    void edit(int u,int l,int r,int mark){
        if(X[tr[u].l] >= r || X[tr[u].r + 1] <= l)return;
        if(X[tr[u].l] >= l && X[tr[u].r + 1] <= r){
            tr[u].sum += mark;
            pushup(u);
            return;
        }
        edit(u << 1,l,r,mark);
        edit(u << 1 | 1,l,r,mark);
        pushup(u);
    }
    long long work(){
        sort(X.begin(),X.end());
        int tot = std::unique(X.begin(),X.end()) - X.begin() - 1;
        tr.resize(tot * 8);
        build(1,0,tot - 1);
        sort(lines.begin(),lines.end(),[](line a,line b){
            return a.y < b.y;
        });
        long long res = 0;
        for(int i = 0;i < (int)lines.size() - 1;i++){
            edit(1,lines[i].x1,lines[i].x2,lines[i].mark);
            res += 1LL * tr[1].len * (lines[i + 1].y - lines[i].y);
        }
        return res;
    }
};
int main()
{
    std::ios::sync_with_stdio(false);
    std::cin.tie(nullptr);
    std::cout.tie(nullptr);
    int n;
    std::cin >> n;
    Sweep_Line sl;
    for(int i = 0; i < n; i++){
        int x1, x2, y1, y2;
        std::cin >> x1 >> y1 >> x2 >> y2;
        sl.addLine(x1, y1, x2, y2);
    }
    std::cout << sl.work() << "\n";
```

```
	return 0;
}
/**	树链剖分（HLD）和线段树（Segment Tree）
 *	功能：在树上实现区间操作和查询，常用于路径加和、路径查询等场景。
 *	日期：2023-11-08
**/
int p;
struct HLD {
	int n;
	std::vector<int> sz,top,depth,parent,in,out,seq;
	std::vector<std::vector<int>> adj;
	int cur;
	HLD () {}
	HLD(int n) {
		this -> n = n;
		sz.resize(n);
		top.resize(n);
		depth.resize(n);
		parent.resize(n);
		in.resize(n);
		out.resize(n);
		seq.resize(n);
		adj.resize(n);
		cur = 0;
	}
	void addEdge(int u,int v){
		adj[u].push_back(v);
		adj[v].push_back(u);
	}
	void work(int root = 0){
		top[root] = root;
		depth[root] = 0;
		parent[root] = -1;
		dfs1(root);
		dfs2(root);
	}
	void dfs1(int u) {
		if(parent[u] != -1){
			adj[u].erase(std::find(adj[u].begin(), adj[u].end(),parent[u]));
		}
		sz[u] = 1;
		for(auto &v : adj[u]){
			depth[v] = depth[u] + 1;
			parent[v] = u;
			dfs1(v);
			sz[u] += sz[v];
			if(sz[v] > sz[adj[u][0]]){
```

```
                std::swap(v,adj[u][0]);
            }
        }
    }
    void dfs2(int u) {
        in[u] = cur++;
        seq[in[u]] = u;
        for(auto v : adj[u]){
            top[v] = v == adj[u][0] ? top[u] : v;
            dfs2(v);
        }
        out[u] = cur;
    }
    int lca(int u,int v){
        while(top[u] != top[v]){
            if(depth[top[u]] > depth[top[v]]){
                u = parent[top[u]];
            }
            else v = parent[top[v]];
        }
        return depth[u] < depth[v] ? u : v;
    }
    int dist(int u,int v){
        return depth[u] + depth[v] - 2 * depth[lca(u,v)];
    }
};
class Segment_Tree{
    struct Node{
        int l ,r;
        long long lazy = 0,sum = 0;
    };
    std::vector<Node> tr;
    void init(int u,int l,int r){
        tr[u].l = l;
        tr[u].r = r;
        if(l == r)return;
        int mid = l + r >> 1;
        init(u << 1,l,mid);
        init(u << 1 | 1,mid + 1,r);
    }
    void pushup(int u){
        tr[u].sum = (tr[u << 1].sum + tr[u << 1 | 1].sum) % p;
    }
    void pushdown(int u){
        tr[u << 1].lazy = (tr[u].lazy + tr[u << 1].lazy) % p;
        tr[u << 1].sum = (tr[u << 1].sum + (tr[u << 1].r - tr[u << 1].l + 1) * tr[u].lazy % p) % p;
        tr[u << 1 | 1].lazy = (tr[u << 1 | 1].lazy + tr[u].lazy) % p;
```

```
            tr[u << 1 | 1].sum = (tr[u << 1 | 1].sum + (tr[u << 1 | 1].r - tr[u << 1 | 1].l + 1) * tr[u].lazy %
p) % p;
            tr[u].lazy = 0;
        }
public :
        Segment_Tree(int n){
            tr.resize(n * 5);
            init(1,0,n);
        }
        void add(int l,int r,long long d,int u = 1){
            if(tr[u].l >= l && tr[u].r <= r){
                tr[u].sum = (tr[u].sum + (tr[u].r - tr[u].l + 1) * d % p) % p;
                tr[u].lazy = (tr[u].lazy + d) % p;
            }
            else {
                pushdown(u);
                int mid = tr[u].l + tr[u].r >> 1;
                if(l <= mid)add(l,r,d,u << 1);
                if(r > mid)add(l,r,d,u << 1 | 1);
                pushup(u);
            }
        }
        long long query(int l,int r,int u = 1){
            if(tr[u].l >= l && tr[u].r <= r){
                return tr[u].sum;
            }
            pushdown(u);
            int mid = tr[u].l + tr[u].r >> 1;
            long long res = 0;
            if(l <= mid)res = (res + query(l,r,u << 1)) % p;
            if(r > mid)res = (res + query(l,r,u << 1 | 1)) % p;
            return res;
        }
};
int main(){
        std::ios::sync_with_stdio(false);
        std::cin.tie(nullptr);
        std::cout.tie(nullptr);
        int n,m,r;
        std::cin >> n >> m >> r >> p;
        HLD t(n);
        std::vector<int> a(n);
        for(int i = 0; i < n; i++){
            std::cin >> a[i];
        }
        for(int i = 0; i < n - 1; i++){
            int x,y;
```

```
        std::cin >> x >> y;
        x--;
        y--;
        t.addEdge(x,y);
    }
    r--;
    t.work(r);
    Segment_Tree sg(n);
    for(int i = 0; i < n; i++){
        sg.add(i,i,a[t.seq[i]]);
    }
    while(m--){
        int op;
        std::cin >> op;
        if(op == 1){
            int x,y,z;
            std::cin >> x >> y >> z;
            x--;
            y--;
            while(t.top[x] != t.top[y]){
                if(t.depth[t.top[x]] < t.depth[t.top[y]])std::swap(x,y);
                sg.add(t.in[t.top[x]],t.in[x],z);
                x = t.parent[t.top[x]];
            }
            if(t.depth[x] > t.depth[y])std::swap(x,y);
            sg.add(t.in[x],t.in[y],z);
        }
        else if(op == 2){
            int x,y;
            std::cin >> x >> y;
            x--;
            y--;
            long long sum = 0;
            while(t.top[x] != t.top[y]){
                if(t.depth[t.top[x]] < t.depth[t.top[y]])std::swap(x,y);
                sum = (sum + sg.query(t.in[t.top[x]],t.in[x])) % p;
                x = t.parent[t.top[x]];
            }
            if(t.depth[x] > t.depth[y])std::swap(x,y);
            sum = (sum + sg.query(t.in[x],t.in[y])) % p;
            std::cout << sum << "\n";
        }
        else if(op == 3){
            int x,z;
            std::cin >> x >> z;
            x--;
            sg.add(t.in[x],t.out[x] - 1,z);
```

```
        }
        else {
            int x;
            std::cin >> x;
            x--;
            std::cout << sg.query(t.in[x],t.out[x] - 1) << "\n";
        }
    }
}
```