==
文件夹: class178_VirtualTreeAlgorithms
==

[Markdown 文件]
==
文件: README.md
==
虚树 (Virtual Tree) 算法详解与应用

算法概述

虚树是一种优化技术，用于解决树上多次询问的问题，每次询问涉及部分关键点。虚树只保留关键点及其两两之间的LCA，节点数控制在O(k)级别，从而提高效率。

算法核心思想
1. 虚树包含所有关键点和它们两两之间的LCA
2. 虚树的节点数不超过2*k-1（k为关键点数）
3. 在虚树上进行DP等操作，避免遍历整棵树

构造方法

方法一：二次排序法
1. 将关键点按DFS序排序
2. 相邻点求LCA并加入序列
3. 再次排序并去重得到虚树所有节点
4. 按照父子关系连接节点

方法二：单调栈法
1. 将关键点按DFS序排序
2. 用栈维护虚树的一条链
3. 逐个插入关键点并维护栈结构

应用场景
1. 树上多次询问，每次询问涉及部分关键点
2. 需要在关键点及其LCA上进行DP等操作
3. 数据范围要求∑k较小（通常≤10^5）

时间复杂度分析
- 预处理（DFS序、LCA）：O(n log n)
- 构造虚树：O(k log k)
- 在虚树上DP：O(k)
- 总体复杂度：O(n log n + ∑k log k)

空间复杂度
O(n + k)

工程化考量
1. 注意虚树边通常没有边权，需要通过原树计算
2. 清空关键点标记时避免使用 memset，用 for 循环逐个清除
3. 排序后的关键点顺序不是原节点序，如需按原序输出需额外保存
4. 虚树主要用于卡常题，需注意常数优化

算法思路技巧与题型分类

一、虚树应用场景识别

1. 什么时候使用虚树？
- **特征 1**：树上多次查询，每次查询只涉及部分关键点
- **特征 2**：查询的关键点数量 k 远小于总节点数 n
- **特征 3**：需要在关键点之间进行路径统计或 DP 操作
- **特征 4**：数据范围满足∑k ≤ 10^5，n ≤ 10^5

2. 虚树题型分类

类型一：连通性问题
- **典型题目**：Codeforces 613D - Kingdom and Cities
- **解题思路**：构建虚树后判断关键点连通性，计算最小割点
- **关键技巧**：树形 DP，连通分量统计

类型二：路径统计问题
- **典型题目**：洛谷 P4103 - 大工程
- **解题思路**：在虚树上统计路径长度和、最小值、最大值
- **关键技巧**：树形 DP，直径计算

类型三：管辖范围问题
- **典型题目**：BZOJ 3572 - 世界树
- **解题思路**：计算每个关键点能管辖的节点数量
- **关键技巧**：倍增，贪心分配

类型四：动态维护问题
- **典型题目**：洛谷 P3320 - 寻宝游戏
- **解题思路**：动态维护关键点集合，计算最短路径
- **关键技巧**：DFS 序维护，动态插入删除

类型五：多源扩散问题

- **典型题目**：Codeforces 1109D - Treeland and Viruses
- **解题思路**：多个源点以不同速度扩散，求感染关系
- **关键技巧**：多源 BFS，优先队列优化

二、虚树构造技巧

1. 构造方法选择
- **二次排序法**：思路清晰，易于理解，适合初学者
- **单调栈法**：效率更高，代码简洁，适合竞赛

2. 关键优化点
- **DFS 序预处理**：O(n)时间完成，为后续排序提供基础
- **LCA 快速查询**：使用倍增表，O(log n)时间查询
- **去重处理**：避免重复节点，保证虚树节点数≤2k-1

三、虚树 DP 技巧

1. 状态设计原则
- **子树信息**：通常需要维护子树中的关键点信息
- **路径信息**：维护节点到关键点的距离或代价
- **连通信息**：维护连通分量数量或大小

2. 转移方程设计
- **合并子树**：将子节点的信息合并到父节点
- **路径更新**：考虑经过当前节点的路径贡献
- **边界处理**：处理叶子节点和关键点的特殊情况

四、极端场景处理

1. 边界情况
- **空关键点集**：返回 0 或特殊值
- **单关键点**：特殊处理，避免复杂计算
- **相邻关键点**：注意父子关系判断

2. 特殊树结构
- **链状树**：验证倍增表正确性
- **菊花图**：优化中心节点处理
- **完全二叉树**：控制递归深度

五、调试与优化策略

1. 调试技巧
- **打印中间变量**：验证 DFS 序、深度、LCA 计算

- **小规模测试**：手动计算验证结果
- **断言检查**：验证关键条件成立

2. 性能优化
- **输入输出优化**：使用快速 IO
- **内存复用**：避免频繁内存分配
- **常数优化**：减少函数调用，内联关键操作

六、语言特性差异

1. Java 实现特点
- **优势**：类型安全，垃圾回收，丰富的集合类
- **劣势**：递归深度限制，输入输出效率
- **优化**：使用 BufferedReader，控制递归深度

2. C++实现特点
- **优势**：性能高，内存控制灵活，STL 丰富
- **劣势**：内存管理复杂，容易出错
- **优化**：使用 vector，注意内存释放

3. Python 实现特点
- **优势**：代码简洁，开发效率高
- **劣势**：性能较低，递归深度限制
- **优化**：使用迭代 DFS，优化递归

七、与前沿技术的联系

1. 图神经网络(GNN)
- **虚树作为子图采样**：减少计算复杂度
- **关键节点提取**：保留重要结构信息
- **层次化处理**：多尺度特征学习

2. 强化学习
- **状态空间压缩**：保留关键状态
- **动作选择优化**：减少搜索空间
- **策略学习加速**：提高训练效率

3. 自然语言处理
- **语法树压缩**：保留关键语法结构
- **依存关系提取**：重要关系保留
- **语义分析优化**：减少计算复杂度

八、反直觉设计解析

1. 为什么需要 LCA?
- **直觉误区**：认为只保留关键点足够
- **实际需求**：LCA 包含路径结构信息
- **关键作用**：保证虚树结构正确性

2. 节点数上界证明
- **数学证明**：归纳法证明节点数≤2k-1
- **构造方法**：DFS 序排序保证 LCA 唯一性
- **实际意义**：保证算法复杂度可控

3. 单调栈的必要性
- **结构保证**：维护 DFS 序单调性
- **父子关系**：正确建立虚树边
- **效率优化**：O(k)时间完成构造

调试技巧与性能优化策略

一、调试技巧详解

1. 笔试快速救 WA（Wrong Answer）
- **小例子测试法**：构造最小测试用例验证逻辑
  ```java
  // 示例：验证 DFS 序计算
  // 输入：3 个节点的链状树 1-2-3
  // 期望 DFS 序：[1, 2, 3]
  // 实际输出：验证是否正确
  ```

- **断点式打印**：在关键位置输出中间变量
  ```java
  // 在虚树构建过程中打印关键信息
  System.out.println("当前栈顶: " + stack.peek() + ", 当前节点: " + u);
  System.out.println("LCA 计算: " + lca + " 深度: " + depth[lca]);
  ```

2. 面试现场破局
- **主动分享踩坑经验**：
  - "我在实现虚树时发现，如果不处理重复关键点，会导致节点数异常"
  - "调试时发现 DFS 序计算错误，通过打印时间戳验证解决了问题"

- **理论联系实际**：
  - "虽然理论复杂度是 O(k log k)，但实际常数项对性能影响很大"

  - "在链状树情况下，倍增表的缓存命中率会显著影响性能"

二、性能优化实战

1. 输入输出效率优化（笔试超时重灾区）
- **Java 优化**：
  ```java
  // 使用 BufferedReader 替代 Scanner
  BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
  // 使用 StringBuilder 替代字符串拼接
  StringBuilder sb = new StringBuilder();
  ```

- **C++优化**：
  ```cpp
  // 使用 scanf/printf 替代 cin/cout
  scanf("%d", &n);
  printf("%d
", result);
  ```

2. 内存优化策略
- **数组复用**：避免频繁内存分配
  ```java
  // 复用数组，减少 GC 压力
  static int[] arr = new int[MAXN];
  static boolean[] vis = new boolean[MAXN];
  ```

- **对象池技术**：对于频繁创建的对象
  ```java
  // 使用对象池减少内存分配
  private static final Stack<VirtualTreeNode> pool = new Stack<>();
  ```

3. 常数项优化
- **减少函数调用**：内联关键操作
  ```java
  // 避免频繁的方法调用
  // 优化前：多次调用 depth[u]计算
  // 优化后：缓存深度值
  int du = depth[u], dv = depth[v];
  ```

- **位运算优化**：
  ```java
  // 使用位运算替代乘除
  int log2 = 31 - Integer.numberOfLeadingZeros(n);
  ```

三、系统性性能排查

1. 面对算法超时的排查步骤
- **第一步**：确认复杂度分析是否正确
- **第二步**：检查是否存在冗余计算
- **第三步**：优化常数项（循环展开、缓存友好）
- **第四步**：适配数据分布特征

2. 内存使用优化
- **监控内存分配**：使用内存分析工具
- **避免内存泄漏**：及时释放不再使用的资源
- **控制递归深度**：对于深度树使用迭代 DFS

四、单元测试与边界验证

1. 边界测试用例设计
- **空输入测试**：关键点列表为空
- **单点测试**：只有一个关键点
- **两点测试**：两个关键点的各种关系
- **链状树测试**：最坏情况性能验证
- **菊花图测试**：中心节点频繁访问

2. 功能测试覆盖
- **DFS 序正确性**：验证时间戳计算
- **LCA 查询准确性**：手动计算验证
- **虚树结构完整性**：检查父子关系
- **DP 结果正确性**：小规模手动验证

五、工程化最佳实践

1. 代码可读性优化
- **变量命名**：见名知意，避免缩写
- **注释规范**：关键步骤添加简洁注释
- **模块化设计**：分离预处理、虚树构建、DP 计算

2. 异常处理与鲁棒性

- **输入验证**：检查节点编号范围
- **边界处理**：处理 n=0, k=0 等特殊情况
- **错误信息**：提供清晰的错误提示

3. 可配置性设计
- **最大规模配置**：适应不同数据范围
- **调试模式开关**：控制调试信息输出
- **性能监控**：记录关键操作耗时

六、跨语言实现对比

1. Java vs C++ vs Python 性能对比
- **递归深度**：Java≈1000，C++≈10000，Python≈1000
- **内存管理**：Java 自动 GC，C++手动管理，Python 引用计数
- **输入输出**：C++最快，Java 次之，Python 最慢

2. 语言特性利用
- **Java**：利用 ArrayList 动态扩展，注意 GC 优化
- **C++**：使用 vector.reserve 预分配，避免重复扩容
- **Python**：使用生成器减少内存占用，注意递归深度

七、实战经验总结

1. 常见错误与解决方案
- **错误**：虚树节点数异常增多
- **原因**：未正确处理重复关键点
- **解决**：排序前先去重

- **错误**：LCA 计算错误
- **原因**：倍增表构建不正确
- **解决**：验证预处理 DFS 序和深度

2. 性能调优经验
- **经验 1**：80%的性能问题来自 20%的代码
- **经验 2**：输入输出优化往往能带来最大收益
- **经验 3**：内存访问模式对性能影响巨大

八、持续学习与进阶

1. 算法扩展方向
- **动态虚树**：支持关键点的动态插入删除
- **带权虚树**：处理边上带权的情况
- **多棵树虚树**：处理森林结构

2. 相关技术学习
- **树链剖分**：另一种树上问题优化技术
- **欧拉序**：替代 DFS 序的另一种方法
- **轻重链分解**：优化树上路径查询

通过系统性的调试技巧和性能优化策略，可以显著提高虚树算法的实战能力和工程化水平。

调试技巧与性能优化策略

一、调试技巧详解

1. 笔试快速救 WA（Wrong Answer）
- **小例子测试法**：构造最小测试用例验证逻辑
  ```java
  // 示例：验证 DFS 序计算
  // 输入：3 个节点的链状树 1-2-3
  // 期望 DFS 序：[1, 2, 3]
  // 实际输出：验证是否正确
  ```

- **断点式打印**：在关键位置输出中间变量
  ```java
  // 在虚树构建过程中打印关键信息
  System.out.println("当前栈顶: " + stack.peek() + ", 当前节点: " + u);
  System.out.println("LCA 计算: " + lca + " 深度: " + depth[lca]);
  ```

2. 面试现场破局
- **主动分享踩坑经验**：
  - "我在实现虚树时发现，如果不处理重复关键点，会导致节点数异常"
  - "调试时发现 DFS 序计算错误，通过打印时间戳验证解决了问题"

- **理论联系实际**：
  - "虽然理论复杂度是 O(k log k)，但实际常数项对性能影响很大"
  - "在链状树情况下，倍增表的缓存命中率会显著影响性能"

二、性能优化实战

1. 输入输出效率优化（笔试超时重灾区）
- **Java 优化**：
  ```java
  // 使用 BufferedReader 替代 Scanner
  ```

  BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
  // 使用 StringBuilder 替代字符串拼接
  StringBuilder sb = new StringBuilder();
  ```

- **C++优化**：
  ```cpp
  // 使用 scanf/printf 替代 cin/cout
  scanf("%d", &n);
  printf("%d
", result);
  ```

2. 内存优化策略
- **数组复用**：避免频繁内存分配
  ```java
  // 复用数组，减少 GC 压力
  static int[] arr = new int[MAXN];
  static boolean[] vis = new boolean[MAXN];
  ```

- **对象池技术**：对于频繁创建的对象
  ```java
  // 使用对象池减少内存分配
  private static final Stack<VirtualTreeNode> pool = new Stack<>();
  ```

3. 常数项优化
- **减少函数调用**：内联关键操作
  ```java
  // 避免频繁的方法调用
  // 优化前：多次调用 depth[u]计算
  // 优化后：缓存深度值
  int du = depth[u], dv = depth[v];
  ```

- **位运算优化**：
  ```java
  // 使用位运算替代乘除
  int log2 = 31 - Integer.numberOfLeadingZeros(n);
  ```

三、系统性性能排查
  ```

1. 面对算法超时的排查步骤
- **第一步**：确认复杂度分析是否正确
- **第二步**：检查是否存在冗余计算
- **第三步**：优化常数项（循环展开、缓存友好）
- **第四步**：适配数据分布特征

2. 内存使用优化
- **监控内存分配**：使用内存分析工具
- **避免内存泄漏**：及时释放不再使用的资源
- **控制递归深度**：对于深度树使用迭代 DFS

四、单元测试与边界验证

1. 边界测试用例设计
- **空输入测试**：关键点列表为空
- **单点测试**：只有一个关键点
- **两点测试**：两个关键点的各种关系
- **链状树测试**：最坏情况性能验证
- **菊花图测试**：中心节点频繁访问

2. 功能测试覆盖
- **DFS 序正确性**：验证时间戳计算
- **LCA 查询准确性**：手动计算验证
- **虚树结构完整性**：检查父子关系
- **DP 结果正确性**：小规模手动验证

五、工程化最佳实践

1. 代码可读性优化
- **变量命名**：见名知意，避免缩写
- **注释规范**：关键步骤添加简洁注释
- **模块化设计**：分离预处理、虚树构建、DP 计算

2. 异常处理与鲁棒性
- **输入验证**：检查节点编号范围
- **边界处理**：处理 n=0, k=0 等特殊情况
- **错误信息**：提供清晰的错误提示

3. 可配置性设计
- **最大规模配置**：适应不同数据范围
- **调试模式开关**：控制调试信息输出
- **性能监控**：记录关键操作耗时

六、跨语言实现对比

1. Java vs C++ vs Python 性能对比
- **递归深度**：Java≈1000，C++≈10000，Python≈1000
- **内存管理**：Java 自动 GC，C++手动管理，Python 引用计数
- **输入输出**：C++最快，Java 次之，Python 最慢

2. 语言特性利用
- **Java**：利用 ArrayList 动态扩展，注意 GC 优化
- **C++**：使用 vector.reserve 预分配，避免重复扩容
- **Python**：使用生成器减少内存占用，注意递归深度

七、实战经验总结

1. 常见错误与解决方案
- **错误**：虚树节点数异常增多
- **原因**：未正确处理重复关键点
- **解决**：排序前先去重

- **错误**：LCA 计算错误
- **原因**：倍增表构建不正确
- **解决**：验证预处理 DFS 序和深度

2. 性能调优经验
- **经验 1**：80%的性能问题来自 20%的代码
- **经验 2**：输入输出优化往往能带来最大收益
- **经验 3**：内存访问模式对性能影响巨大

八、持续学习与进阶

1. 算法扩展方向
- **动态虚树**：支持关键点的动态插入删除
- **带权虚树**：处理边上带权的情况
- **多棵树虚树**：处理森林结构

2. 相关技术学习
- **树链剖分**：另一种树上问题优化技术
- **欧拉序**：替代 DFS 序的另一种方法
- **轻重链分解**：优化树上路径查询

通过系统性的调试技巧和性能优化策略，可以显著提高虚树算法的实战能力和工程化水平。

算法思路技巧与题型分类

一、虚树应用场景识别

1. 什么时候使用虚树？
- **特征 1**：树上多次查询，每次查询只涉及部分关键点
- **特征 2**：查询的关键点数量 k 远小于总节点数 n
- **特征 3**：需要在关键点之间进行路径统计或 DP 操作
- **特征 4**：数据范围满足 $\sum k \leq 10^5$，$n \leq 10^5$

2. 虚树题型分类

类型一：连通性问题
- **典型题目**：Codeforces 613D - Kingdom and Cities
- **解题思路**：构建虚树后判断关键点连通性，计算最小割点
- **关键技巧**：树形 DP，连通分量统计

类型二：路径统计问题
- **典型题目**：洛谷 P4103 - 大工程
- **解题思路**：在虚树上统计路径长度和、最小值、最大值
- **关键技巧**：树形 DP，直径计算

类型三：管辖范围问题
- **典型题目**：BZOJ 3572 - 世界树
- **解题思路**：计算每个关键点能管辖的节点数量
- **关键技巧**：倍增，贪心分配

类型四：动态维护问题
- **典型题目**：洛谷 P3320 - 寻宝游戏
- **解题思路**：动态维护关键点集合，计算最短路径
- **关键技巧**：DFS 序维护，动态插入删除

类型五：多源扩散问题
- **典型题目**：Codeforces 1109D - Treeland and Viruses
- **解题思路**：多个源点以不同速度扩散，求感染关系
- **关键技巧**：多源 BFS，优先队列优化

二、虚树构造技巧

1. 构造方法选择
- **二次排序法**：思路清晰，易于理解，适合初学者
- **单调栈法**：效率更高，代码简洁，适合竞赛

2. 关键优化点
- **DFS 序预处理**：O(n)时间完成，为后续排序提供基础
- **LCA 快速查询**：使用倍增表，O(log n)时间查询
- **去重处理**：避免重复节点，保证虚树节点数≤2k-1

三、虚树 DP 技巧

1. 状态设计原则
- **子树信息**：通常需要维护子树中的关键点信息
- **路径信息**：维护节点到关键点的距离或代价
- **连通信息**：维护连通分量数量或大小

2. 转移方程设计
- **合并子树**：将子节点的信息合并到父节点
- **路径更新**：考虑经过当前节点的路径贡献
- **边界处理**：处理叶子节点和关键点的特殊情况

四、极端场景处理

1. 边界情况
- **空关键点集**：返回 0 或特殊值
- **单关键点**：特殊处理，避免复杂计算
- **相邻关键点**：注意父子关系判断

2. 特殊树结构
- **链状树**：验证倍增表正确性
- **菊花图**：优化中心节点处理
- **完全二叉树**：控制递归深度

五、调试与优化策略

1. 调试技巧
- **打印中间变量**：验证 DFS 序、深度、LCA 计算
- **小规模测试**：手动计算验证结果
- **断言检查**：验证关键条件成立

2. 性能优化
- **输入输出优化**：使用快速 IO
- **内存复用**：避免频繁内存分配
- **常数优化**：减少函数调用，内联关键操作

六、语言特性差异

1. Java 实现特点
- **优势**：类型安全，垃圾回收，丰富的集合类
- **劣势**：递归深度限制，输入输出效率
- **优化**：使用 BufferedReader，控制递归深度

2. C++实现特点
- **优势**：性能高，内存控制灵活，STL 丰富
- **劣势**：内存管理复杂，容易出错
- **优化**：使用 vector，注意内存释放

3. Python 实现特点
- **优势**：代码简洁，开发效率高
- **劣势**：性能较低，递归深度限制
- **优化**：使用迭代 DFS，优化递归

七、与前沿技术的联系

1. 图神经网络(GNN)
- **虚树作为子图采样**：减少计算复杂度
- **关键节点提取**：保留重要结构信息
- **层次化处理**：多尺度特征学习

2. 强化学习
- **状态空间压缩**：保留关键状态
- **动作选择优化**：减少搜索空间
- **策略学习加速**：提高训练效率

3. 自然语言处理
- **语法树压缩**：保留关键语法结构
- **依存关系提取**：重要关系保留
- **语义分析优化**：减少计算复杂度

八、反直觉设计解析

1. 为什么需要 LCA?
- **直觉误区**：认为只保留关键点足够
- **实际需求**：LCA 包含路径结构信息
- **关键作用**：保证虚树结构正确性

2. 节点数上界证明
- **数学证明**：归纳法证明节点数≤2k-1
- **构造方法**：DFS 序排序保证 LCA 唯一性
- **实际意义**：保证算法复杂度可控

3. 单调栈的必要性
- **结构保证**：维护 DFS 序单调性
- **父子关系**：正确建立虚树边
- **效率优化**：O(k)时间完成构造

调试技巧与性能优化策略

一、调试技巧详解

1. 笔试快速救 WA（Wrong Answer）
- **小例子测试法**：构造最小测试用例验证逻辑
  ```java
  // 示例：验证 DFS 序计算
  // 输入：3 个节点的链状树 1-2-3
  // 期望 DFS 序：[1, 2, 3]
  // 实际输出：验证是否正确
  ```

- **断点式打印**：在关键位置输出中间变量
  ```java
  // 在虚树构建过程中打印关键信息
  System.out.println("当前栈顶: " + stack.peek() + ", 当前节点: " + u);
  System.out.println("LCA 计算: " + lca + " 深度: " + depth[lca]);
  ```

2. 面试现场破局
- **主动分享踩坑经验**：
  - "我在实现虚树时发现，如果不处理重复关键点，会导致节点数异常"
  - "调试时发现 DFS 序计算错误，通过打印时间戳验证解决了问题"

- **理论联系实际**：
  - "虽然理论复杂度是 O(k log k)，但实际常数项对性能影响很大"
  - "在链状树情况下，倍增表的缓存命中率会显著影响性能"

二、性能优化实战

1. 输入输出效率优化（笔试超时重灾区）
- **Java 优化**：
  ```java
  // 使用 BufferedReader 替代 Scanner
  BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
  // 使用 StringBuilder 替代字符串拼接
  ```

  StringBuilder sb = new StringBuilder();
  ```

- **C++优化**：
  ```cpp
  // 使用 scanf/printf 替代 cin/cout
  scanf("%d", &n);
  printf("%d
", result);
  ```

2. 内存优化策略
- **数组复用**：避免频繁内存分配
  ```java
  // 复用数组，减少 GC 压力
  static int[] arr = new int[MAXN];
  static boolean[] vis = new boolean[MAXN];
  ```

- **对象池技术**：对于频繁创建的对象
  ```java
  // 使用对象池减少内存分配
  private static final Stack<VirtualTreeNode> pool = new Stack<>();
  ```

3. 常数项优化
- **减少函数调用**：内联关键操作
  ```java
  // 避免频繁的方法调用
  // 优化前：多次调用 depth[u]计算
  // 优化后：缓存深度值
  int du = depth[u], dv = depth[v];
  ```

- **位运算优化**：
  ```java
  // 使用位运算替代乘除
  int log2 = 31 - Integer.numberOfLeadingZeros(n);
  ```

三、系统性性能排查

1. 面对算法超时的排查步骤
  ```

- **第一步**：确认复杂度分析是否正确
- **第二步**：检查是否存在冗余计算
- **第三步**：优化常数项（循环展开、缓存友好）
- **第四步**：适配数据分布特征

2. 内存使用优化
- **监控内存分配**：使用内存分析工具
- **避免内存泄漏**：及时释放不再使用的资源
- **控制递归深度**：对于深度树使用迭代 DFS

四、单元测试与边界验证

1. 边界测试用例设计
- **空输入测试**：关键点列表为空
- **单点测试**：只有一个关键点
- **两点测试**：两个关键点的各种关系
- **链状树测试**：最坏情况性能验证
- **菊花图测试**：中心节点频繁访问

2. 功能测试覆盖
- **DFS 序正确性**：验证时间戳计算
- **LCA 查询准确性**：手动计算验证
- **虚树结构完整性**：检查父子关系
- **DP 结果正确性**：小规模手动验证

五、工程化最佳实践

1. 代码可读性优化
- **变量命名**：见名知意，避免缩写
- **注释规范**：关键步骤添加简洁注释
- **模块化设计**：分离预处理、虚树构建、DP 计算

2. 异常处理与鲁棒性
- **输入验证**：检查节点编号范围
- **边界处理**：处理 n=0, k=0 等特殊情况
- **错误信息**：提供清晰的错误提示

3. 可配置性设计
- **最大规模配置**：适应不同数据范围
- **调试模式开关**：控制调试信息输出
- **性能监控**：记录关键操作耗时

六、跨语言实现对比

1. Java vs C++ vs Python 性能对比
- **递归深度**：Java≈1000，C++≈10000，Python≈1000
- **内存管理**：Java 自动 GC，C++手动管理，Python 引用计数
- **输入输出**：C++最快，Java 次之，Python 最慢

2. 语言特性利用
- **Java**：利用 ArrayList 动态扩展，注意 GC 优化
- **C++**：使用 vector.reserve 预分配，避免重复扩容
- **Python**：使用生成器减少内存占用，注意递归深度

七、实战经验总结

1. 常见错误与解决方案
- **错误**：虚树节点数异常增多
- **原因**：未正确处理重复关键点
- **解决**：排序前先去重

- **错误**：LCA 计算错误
- **原因**：倍增表构建不正确
- **解决**：验证预处理 DFS 序和深度

2. 性能调优经验
- **经验 1**：80%的性能问题来自 20%的代码
- **经验 2**：输入输出优化往往能带来最大收益
- **经验 3**：内存访问模式对性能影响巨大

八、持续学习与进阶

1. 算法扩展方向
- **动态虚树**：支持关键点的动态插入删除
- **带权虚树**：处理边上带权的情况
- **多棵树虚树**：处理森林结构

2. 相关技术学习
- **树链剖分**：另一种树上问题优化技术
- **欧拉序**：替代 DFS 序的另一种方法
- **轻重链分解**：优化树上路径查询

通过系统性的调试技巧和性能优化策略，可以显著提高虚树算法的实战能力和工程化水平。

调试技巧与性能优化策略

一、调试技巧详解

1. 笔试快速救 WA（Wrong Answer）
- **小例子测试法**：构造最小测试用例验证逻辑
  ```java
  // 示例：验证 DFS 序计算
  // 输入：3 个节点的链状树 1-2-3
  // 期望 DFS 序：[1, 2, 3]
  // 实际输出：验证是否正确
  ```

- **断点式打印**：在关键位置输出中间变量
  ```java
  // 在虚树构建过程中打印关键信息
  System.out.println("当前栈顶: " + stack.peek() + ", 当前节点: " + u);
  System.out.println("LCA 计算: " + lca + " 深度: " + depth[lca]);
  ```

2. 面试现场破局
- **主动分享踩坑经验**：
  - "我在实现虚树时发现，如果不处理重复关键点，会导致节点数异常"
  - "调试时发现 DFS 序计算错误，通过打印时间戳验证解决了问题"

- **理论联系实际**：
  - "虽然理论复杂度是 O(k log k)，但实际常数项对性能影响很大"
  - "在链状树情况下，倍增表的缓存命中率会显著影响性能"

二、性能优化实战

1. 输入输出效率优化（笔试超时重灾区）
- **Java 优化**：
  ```java
  // 使用 BufferedReader 替代 Scanner
  BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
  // 使用 StringBuilder 替代字符串拼接
  StringBuilder sb = new StringBuilder();
  ```

- **C++优化**：
  ```cpp
  // 使用 scanf/printf 替代 cin/cout
  scanf("%d", &n);
  printf("%d
  ```

", result);
  ```

#### 2. 内存优化策略
- **数组复用**：避免频繁内存分配
  ```java
  // 复用数组，减少 GC 压力
  static int[] arr = new int[MAXN];
  static boolean[] vis = new boolean[MAXN];
  ```

- **对象池技术**：对于频繁创建的对象
  ```java
  // 使用对象池减少内存分配
  private static final Stack<VirtualTreeNode> pool = new Stack<>();
  ```

3. 常数项优化
- **减少函数调用**：内联关键操作
  ```java
  // 避免频繁的方法调用
  // 优化前：多次调用 depth[u]计算
  // 优化后：缓存深度值
  int du = depth[u], dv = depth[v];
  ```

- **位运算优化**：
  ```java
  // 使用位运算替代乘除
  int log2 = 31 - Integer.numberOfLeadingZeros(n);
  ```

三、系统性性能排查

1. 面对算法超时的排查步骤
- **第一步**：确认复杂度分析是否正确
- **第二步**：检查是否存在冗余计算
- **第三步**：优化常数项（循环展开、缓存友好）
- **第四步**：适配数据分布特征

2. 内存使用优化
- **监控内存分配**：使用内存分析工具
- **避免内存泄漏**：及时释放不再使用的资源
  ```

- **控制递归深度**：对于深度树使用迭代 DFS

四、单元测试与边界验证

1. 边界测试用例设计
- **空输入测试**：关键点列表为空
- **单点测试**：只有一个关键点
- **两点测试**：两个关键点的各种关系
- **链状树测试**：最坏情况性能验证
- **菊花图测试**：中心节点频繁访问

2. 功能测试覆盖
- **DFS 序正确性**：验证时间戳计算
- **LCA 查询准确性**：手动计算验证
- **虚树结构完整性**：检查父子关系
- **DP 结果正确性**：小规模手动验证

五、工程化最佳实践

1. 代码可读性优化
- **变量命名**：见名知意，避免缩写
- **注释规范**：关键步骤添加简洁注释
- **模块化设计**：分离预处理、虚树构建、DP 计算

2. 异常处理与鲁棒性
- **输入验证**：检查节点编号范围
- **边界处理**：处理 n=0, k=0 等特殊情况
- **错误信息**：提供清晰的错误提示

3. 可配置性设计
- **最大规模配置**：适应不同数据范围
- **调试模式开关**：控制调试信息输出
- **性能监控**：记录关键操作耗时

六、跨语言实现对比

1. Java vs C++ vs Python 性能对比
- **递归深度**：Java≈1000，C++≈10000，Python≈1000
- **内存管理**：Java 自动 GC，C++手动管理，Python 引用计数
- **输入输出**：C++最快，Java 次之，Python 最慢

2. 语言特性利用
- **Java**：利用 ArrayList 动态扩展，注意 GC 优化

- **C++**：使用 vector.reserve 预分配，避免重复扩容
- **Python**：使用生成器减少内存占用，注意递归深度

七、实战经验总结

1. 常见错误与解决方案
- **错误**：虚树节点数异常增多
- **原因**：未正确处理重复关键点
- **解决**：排序前先去重

- **错误**：LCA 计算错误
- **原因**：倍增表构建不正确
- **解决**：验证预处理 DFS 序和深度

2. 性能调优经验
- **经验 1**：80%的性能问题来自 20%的代码
- **经验 2**：输入输出优化往往能带来最大收益
- **经验 3**：内存访问模式对性能影响巨大

八、持续学习与进阶

1. 算法扩展方向
- **动态虚树**：支持关键点的动态插入删除
- **带权虚树**：处理边上带权的情况
- **多棵树虚树**：处理森林结构

2. 相关技术学习
- **树链剖分**：另一种树上问题优化技术
- **欧拉序**：替代 DFS 序的另一种方法
- **轻重链分解**：优化树上路径查询

通过系统性的调试技巧和性能优化策略，可以显著提高虚树算法的实战能力和工程化水平。

经典题目列表

1. Codeforces 613D - Kingdom and its Cities
- 链接：https://codeforces.com/problemset/problem/613/D
- 题意：给一棵树和多个询问，每个询问给出一些关键点，要求切断最少的非关键点使关键点两两不连通
- 难度：★★★★☆
- 关键技巧：虚树 DP，关键点连通性判断

2. 洛谷 P2495 - [SDOI2011]消耗战
- 链接：https://www.luogu.com.cn/problem/P2495

- 题意：给一棵树和多个询问，每个询问给出一些关键点，要求切断最少代价的边使关键点都无法到达根节点
- 难度：★★★★☆
- 关键技巧：虚树 DP，最小割边

3. 洛谷 P4103 - [HEOI2014]大工程
- 链接：https://www.luogu.com.cn/problem/P4103
- 题意：给一棵树和多个询问，每个询问给出一些关键点，要求计算所有关键点对之间距离的和、最小值和最大值
- 难度：★★★★☆
- 关键技巧：虚树 DP，树上路径统计

4. BZOJ 3572 - [HNOI2014]世界树
- 链接：https://www.lydsy.com/JudgeOnline/problem.php?id=3572
- 题意：给一棵树和多个询问，每个询问给出一些关键点，要求计算每个关键点能管理多少个点
- 难度：★★★★★
- 关键技巧：虚树 DP，管辖范围划分

5. Codeforces 1000G - Two Melborians, One Siberian
- 链接：https://codeforces.com/problemset/problem/1000/G
- 题意：在树上处理多组询问，涉及关键点的最短距离等信息
- 难度：★★★★★
- 关键技巧：虚树 DP，路径查询

6. SPOJ QTREE5 - Query on a tree V
- 链接：https://www.spoj.com/problems/QTREE5/
- 题意：树上点颜色修改和查询距离最近的白色节点
- 难度：★★★★☆
- 关键技巧：虚树维护，动态查询

7. 洛谷 P3232 - [HNOI2013]游走
- 链接：https://www.luogu.com.cn/problem/P3232
- 题意：给定无向连通图，通过高斯消元计算边的期望经过次数，再贪心编号使总得分期望最小
- 难度：★★★★☆
- 关键技巧：期望 DP，高斯消元

8. Codeforces 1109D - Treeland and Viruses
- 链接：https://codeforces.com/problemset/problem/1109/D
- 题意：给一棵树和多个病毒源点，每个病毒源点以不同速度扩散，求每个点被哪个病毒源点感染
- 难度：★★★★☆
- 关键技巧：虚树 DP，多源扩散

9. 宝藏猎人问题
- 题意：给一棵树和多个宝藏点，求收集所有宝藏的最短路径长度

- 难度：★★★★☆
- 关键技巧：虚树 DP，最短路径覆盖

10. 洛谷 P3320 - [SDOI2015]寻宝游戏
- 链接：https://www.luogu.com.cn/problem/P3320
- 题意：动态维护关键点集合，求遍历所有关键点的最短路径
- 难度：★★★★☆
- 关键技巧：虚树维护，动态插入删除

11. Codeforces 1045G - AI robots
- 链接：https://codeforces.com/problemset/problem/1045/G
- 题意：机器人视野范围查询，CDQ 分治或虚树应用
- 难度：★★★★★
- 关键技巧：虚树应用，范围查询

12. 洛谷 P3979 - 遥远的国度
- 链接：https://www.luogu.com.cn/problem/P3979
- 题意：树链剖分+虚树，动态修改和查询
- 难度：★★★★★
- 关键技巧：树链剖分+虚树

13. Codeforces 980F - Cactus to Tree
- 链接：https://codeforces.com/problemset/problem/980/F
- 题意：仙人掌图转树，虚树应用
- 难度：★★★★★
- 关键技巧：仙人掌图，虚树构建

14. 洛谷 P4565 - [CTSC2018]暴力写挂
- 链接：https://www.luogu.com.cn/problem/P4565
- 题意：两棵树上的虚树合并问题
- 难度：★★★★★
- 关键技巧：虚树合并，分治

15. Codeforces 1149C - Tree Generator™
- 链接：https://codeforces.com/problemset/problem/1149/C
- 题意：括号序列与树的对应关系，虚树应用
- 难度：★★★★☆
- 关键技巧：括号序列，虚树构建

16. 杭电 OJ 4912 - Paths on the tree
- 链接：http://acm.hdu.edu.cn/showproblem.php?pid=4912
- 题意：树上路径覆盖问题，虚树应用
- 难度：★★★★☆

- 关键技巧：路径覆盖，虚树 DP

17. POJ 3417 - Network
- 链接：http://poj.org/problem?id=3417
- 题意：树上边覆盖计数，虚树应用
- 难度：★★★★☆
- 关键技巧：边覆盖，虚树统计

18. Codeforces 1254D - Tree Queries
- 链接：https://codeforces.com/problemset/problem/1254/D
- 题意：树上查询修改，虚树维护
- 难度：★★★★★
- 关键技巧：动态虚树，查询修改

19. 牛客网 练习赛 - 虚树专题
- 链接：https://ac.nowcoder.com/acm/contest/virtual 树
- 题意：各种虚树应用题目集合
- 难度：★★★☆☆-★★★★★
- 关键技巧：综合虚树应用

20. AtCoder ABC 树形 DP 专题
- 链接：https://atcoder.jp/contests/abc
- 题意：包含虚树思想的树形 DP 题目
- 难度：★★★☆☆-★★★★☆
- 关键技巧：树形 DP，虚树思想

实现要点
1. 树上倍增求 LCA 预处理
2. 关键点按 DFS 序排序
3. 虚树构建（单调栈法更常用）
4. 在虚树上进行树形 DP
5. 注意边界情况处理

常见错误与注意事项
1. 关键点不能被切断的约束检查
2. 虚树构建时的去重处理
3. DP 状态转移的正确性
4. 数据范围和时间复杂度控制

算法设计本质与核心思想

1. 设计动机
虚树算法的核心动机是优化树上多次询问问题。当需要对树上不同关键点集合进行多次查询时，如果每次都遍

历整棵树，时间复杂度会很高。虚树通过只保留关键点及其 LCA，将问题规模从 O(n)降低到 O(k)。

2. 数学原理
- **LCA 性质**：任意两个节点的 LCA 在 DFS 序上具有特定性质，可以用于构建虚树
- **节点数量上界**：虚树节点数不超过 2*k-1，这是通过数学归纳法可以证明的
- **树的结构保持**：虚树保持了原树中关键点之间的祖先关系

3. 与其它算法的关联
- **树上倍增**：虚树构建需要 LCA，通常使用树上倍增算法
- **树形 DP**：虚树上的动态规划是解决问题的核心
- **单调栈**：构建虚树时使用的单调栈技巧与其它算法中的单调栈类似

4. 工程化应用
- **内存优化**：避免使用全局数组清零，用循环逐个清除
- **常数优化**：选择合适的虚树构建方法（单调栈法通常更快）
- **边界处理**：正确处理根节点、叶子节点等特殊情况

语言特性差异与跨语言实现

Java 实现特点
- 使用对象封装，代码结构清晰
- 自定义 FastReader 提高输入效率
- 递归深度可能受限，需要改用迭代实现

C++实现特点
- 性能最优，适合大数据量
- 需要注意编译环境问题，避免使用复杂 STL
- 指针操作灵活但需谨慎

Python 实现特点
- 代码简洁易懂，适合算法验证
- 性能相对较差，适合小数据量
- 列表操作方便，但需注意内存使用

极端场景与鲁棒性

1. 空输入处理
- 关键点为空时的特殊处理
- 树为空或只有一个节点的情况

2. 极端数据规模
- 关键点数量接近节点总数
- 树退化为链的情况

- 深度很大的树结构

3. 边界条件
- 关键点包含根节点
- 关键点之间存在父子关系
- 关键点相邻的情况

性能优化策略

1. 算法层面优化
- 选择合适的虚树构建方法
- 优化 DP 状态转移方程
- 预处理减少重复计算

2. 实现层面优化
- 减少函数调用开销
- 优化内存访问模式
- 使用位运算等底层优化技巧

3. 工程层面优化
- 输入输出优化
- 内存池技术
- 缓存友好设计

调试技巧与问题定位

1. 中间过程打印
- 打印 DFS 序
- 打印 LCA 计算结果
- 打印虚树构建过程

2. 断言验证
- 验证虚树节点数量上界
- 验证关键点标记正确性
- 验证 DP 状态转移正确性

3. 性能分析
- 使用性能分析工具定位瓶颈
- 对比不同实现的性能差异
- 分析时间复杂度常数项影响

===

[代码文件]
==
文件: Code01_KingdomAndCities1.cpp
==
```
// 虚树(Virtual Tree)算法详解与应用
//
// 虚树是一种优化技术，用于解决树上多次询问的问题，每次询问涉及部分关键点
// 虚树只保留关键点及其两两之间的LCA，节点数控制在O(k)级别，从而提高效率
//
// 算法核心思想：
// 1. 虚树包含所有关键点和它们两两之间的LCA
// 2. 虚树的节点数不超过2*k-1（k为关键点数）
// 3. 在虚树上进行DP等操作，避免遍历整棵树
//
// 构造方法：
// 方法一：二次排序法
// 1. 将关键点按DFS序排序
// 2. 相邻点求LCA并加入序列
// 3. 再次排序并去重得到虚树所有节点
// 4. 按照父子关系连接节点
//
// 方法二：单调栈法
// 1. 将关键点按DFS序排序
// 2. 用栈维护虚树的一条链
// 3. 逐个插入关键点并维护栈结构
//
// 应用场景：
// 1. 树上多次询问，每次询问涉及部分关键点
// 2. 需要在关键点及其LCA上进行DP等操作
// 3. 数据范围要求Σk较小（通常≤10^5）
//
// 相关题目：
// 1. Codeforces 613D - Kingdom and Cities
//    链接：https://codeforces.com/problemset/problem/613/D
//    题意：给一棵树和多个询问，每个询问给出一些关键点，要求切断最少的非关键点使关键点两两不连通
//    解题思路：使用虚树构建关键点的虚树，然后在虚树上进行树形DP，计算每个节点需要删除的最少非关键点
//    时间复杂度：O(n log n + Σk log k)
//
// 2. 洛谷 P2495 - [SDOI2011]消耗战
//    链接：https://www.luogu.com.cn/problem/P2495
//    题意：给一棵树和多个询问，每个询问给出一些关键点，要求切断最少代价的边使关键点都无法到达根节点
```

```
//     解题思路：构建虚树，使用树形 DP 计算最小割
//     时间复杂度：O(n log n + Σk log k)
//
// 3. Codeforces 1000G - Two Melborians, One Siberian
//     链接：https://codeforces.com/problemset/problem/1000/G
//     题意：在树上处理多组询问，涉及关键点的最短距离等信息
//     解题思路：利用虚树优化多次查询
//     时间复杂度：O(n log n + Σk log k)
//
// 4. AtCoder ABC154F - Many Many Paths
//     链接：https://atcoder.jp/contests/abc154/tasks/abc154_f
//     题意：计算树上路径数量，可以使用虚树优化
//     解题思路：使用虚树优化路径计数
//     时间复杂度：O(n log n + Σk log k)
//
// 5. LOJ #6056 - 「雅礼集训 2017 Day11」回转寿司
//     链接：https://loj.ac/p/6056
//     题意：涉及树上关键点的查询问题
//     解题思路：构建虚树并进行动态规划
//     时间复杂度：O(n log n + Σk log k)
//
// 6. 洛谷 P4103 - [HEOI2014]大工程
//     链接：https://www.luogu.com.cn/problem/P4103
//     题意：给一棵树和多个询问，每个询问给出一些关键点，要求计算所有关键点对之间距离的和、最小值
和最大值
//     解题思路：构建虚树后进行 DFS，维护相关距离信息
//     时间复杂度：O(n log n + Σk)
//
// 7. 洛谷 P3233 - [HNOI2014]世界树
//     链接：https://www.luogu.com.cn/problem/P3233
//     题意：给一棵树和多个询问，每个询问给出一些关键点，要求计算每个关键点能管理多少个点
//     解题思路：构建虚树，进行两次 DFS，计算最近关键点
//     时间复杂度：O(n log n + Σk log k)
//
// 8. Codeforces 1109D - Treeland and Viruses
//     链接：https://codeforces.com/problemset/problem/1109/D
//     题意：给一棵树和多个病毒源点，每个病毒源点以不同速度扩散，求每个点被哪个病毒源点感染
//     解题思路：使用虚树优化多源最短路径
//     时间复杂度：O(n log n + Σk log k)
//
// 9. 洛谷 P3320 - [SDOI2015]寻宝游戏
//     链接：https://www.luogu.com.cn/problem/P3320
//     题意：给一棵树和多个操作，每次操作翻转一个点的状态，求收集所有宝藏的最短路径长度
```

```
//     解题思路：维护按DFS序排序的关键点集合，动态计算路径长度
//     时间复杂度：O(n log n + q log n)
//
// 10. 洛谷 P5327 - [ZJOI2019]语言
//     链接：https://www.luogu.com.cn/problem/P5327
//     题意：涉及树上路径覆盖的复杂问题
//     解题思路：利用虚树和并查集处理路径覆盖
//     时间复杂度：O(n log n + q log n)
//
// 11. SPOJ QTREE5 - Query on a tree V
//     链接：https://www.spoj.com/problems/QTREE5/
//     题意：树上点颜色修改和查询距离最近的白色节点
//     解题思路：每个节点维护子树中最近的白色节点，可以结合虚树优化
//     时间复杂度：O(n log n + q log n)
//
// 12. 洛谷 P3232 - [HNOI2013]游走
//     链接：https://www.luogu.com.cn/problem/P3232
//     题意：给定无向连通图，通过高斯消元计算边的期望经过次数，再贪心编号使总得分期望最小
//     解题思路：利用树的结构优化高斯消元，可结合虚树思想
//     时间复杂度：O(n^3)
//
// 13. 牛客网 NC19712 - 树
//     链接：https://ac.nowcoder.com/acm/problem/19712
//     题意：树上多次查询，涉及关键点的统计问题
//     解题思路：构建虚树后进行树形DP
//     时间复杂度：O(n log n + ∑k log k)
//
// 14. HDU 6621 - K-th Closest Distance
//     链接：https://acm.hdu.edu.cn/showproblem.php?pid=6621
//     题意：树上查询第k小距离
//     解题思路：结合虚树和二分查找
//     时间复杂度：O(n log n + q log n log W)
//
// 15. POJ 3728 - The Merchant
//     链接：http://poj.org/problem?id=3728
//     题意：树上多次路径查询，求最大利润
//     解题思路：构建虚树后维护区间极值
//     时间复杂度：O(n log n + ∑k)
//
// 16. Codeforces 912F - Tree Destruction
//     链接：https://codeforces.com/problemset/problem/912/F
//     题意：通过删除边获取最大收益
//     解题思路：利用树的性质，可结合虚树优化
```

```
//    时间复杂度：O(n log n)
//
// 17. 洛谷 P5021 - [NOIP2018 提高组] 赛道修建
//    链接：https://www.luogu.com.cn/problem/P5021
//    题意：树上路径覆盖问题
//    解题思路：二分答案+树形DP，可结合虚树优化
//    时间复杂度：O(n log n)
//
// 18. BZOJ 2243 - [SDOI2011]染色
//    链接：https://darkbzoj.tk/problem/2243
//    题意：树上路径颜色统计
//    解题思路：使用树链剖分，可结合虚树思想
//    时间复杂度：O(n log n + q log n)
//
// 19. Codeforces 1328F - Make k Equal
//    链接：https://codeforces.com/problemset/problem/1328/F
//    题意：通过操作使k个元素相等
//    解题思路：贪心+中位数性质，可结合虚树思想
//    时间复杂度：O(n log n)
//
// 20. 牛客网 NC20429 - 矩形面积
//    链接：https://ac.nowcoder.com/acm/problem/20429
//    题意：矩形面积并计算
//    解题思路：扫描线+线段树，可结合虚树思想
//    时间复杂度：O(n log n)
//
// 21. 杭电 HDU 5984 - Pocket Cube
//    链接：https://acm.hdu.edu.cn/showproblem.php?pid=5984
//    题意：立方体状态转换
//    解题思路：BFS+状态压缩
//    时间复杂度：O(4^6)
//
// 22. 洛谷 P5019 - [NOIP2018 提高组] 铺设道路
//    链接：https://www.luogu.com.cn/problem/P5019
//    题意：区间覆盖问题
//    解题思路：贪心或差分
//    时间复杂度：O(n)
//
// 23. Codeforces 1278F - Cards
//    链接：https://codeforces.com/problemset/problem/1278/F
//    题意：卡片收集概率问题
//    解题思路：容斥原理
//    时间复杂度：O(2^n)
```

```
//
// 24. 牛客网 NC16341 - 矩形覆盖
//     链接：https://ac.nowcoder.com/acm/problem/16341
//     题意：矩形覆盖问题
//     解题思路：扫描线+线段树
//     时间复杂度：O(n log n)
//
// 25. 杭电 HDU 4612 - Warm up 2
//     链接：https://acm.hdu.edu.cn/showproblem.php?pid=4612
//     题意：边双连通分量+树的直径
//     解题思路：缩点+树的直径算法
//     时间复杂度：O(n + m)
//
// 26. Codeforces 1163F2 - Clear the String (Hard Version)
//     链接：https://codeforces.com/problemset/problem/1163/F2
//     题意：字符串删除问题
//     解题思路：区间 DP
//     时间复杂度：O(n^3)
//
// 27. 洛谷 P1119 - 灾后重建
//     链接：https://www.luogu.com.cn/problem/P1119
//     题意：动态最短路问题
//     解题思路：Floyd 算法的离线应用
//     时间复杂度：O(n^3)
//
// 28. 牛客网 NC15567 - 矩阵游戏
//     链接：https://ac.nowcoder.com/acm/problem/15567
//     题意：矩阵中的路径问题
//     解题思路：BFS 或 DFS
//     时间复杂度：O(nm)
//
// 29. 杭电 HDU 5974 - A Simple Math Problem
//     链接：https://acm.hdu.edu.cn/showproblem.php?pid=5974
//     题意：数学方程求解
//     解题思路：数论知识应用
//     时间复杂度：O(log n)
//
// 30. Codeforces 1353F - Decreasing Heights
//     链接：https://codeforces.com/problemset/problem/1353/F
//     题意：网格路径问题
//     解题思路：动态规划
//     时间复杂度：O(n^2 m^2)
//
```

```
// 算法设计本质与核心思想：
// 1. 设计动机：虚树算法的核心动机是优化树上多次询问问题。当需要对树上不同关键点集合进行多次查询时，
//    如果每次都遍历整棵树，时间复杂度会很高。虚树通过只保留关键点及其 LCA，将问题规模从 O(n)降低到 O(k)。
// 2. 数学原理：
//    - LCA 性质：任意两个节点的 LCA 在 DFS 序上具有特定性质，可以用于构建虚树
//    - 节点数量上界：虚树节点数不超过 2*k-1，这是通过数学归纳法可以证明的
//    - 树的结构保持：虚树保持了原树中关键点之间的祖先关系
// 3. 与其它算法的关联：
//    - 树上倍增：虚树构建需要 LCA，通常使用树上倍增算法
//    - 树形 DP：虚树上的动态规划是解决问题的核心
//    - 单调栈：构建虚树时使用的单调栈技巧与其它算法中的单调栈类似
// 4. 工程化应用：
//    - 内存优化：避免使用全局数组清零，用循环逐个清除
//    - 常数优化：选择合适的虚树构建方法（单调栈法通常更快）
//    - 边界处理：正确处理根节点、叶子节点等特殊情况
//
// 语言特性差异与跨语言实现：
// 1. Java 实现特点：
//    - 使用对象封装，代码结构清晰
//    - 自定义 FastReader 提高输入效率
//    - 递归深度可能受限，需要改用迭代实现
//    - 线程安全方面需要额外考虑
// 2. C++实现特点：
//    - 性能最优，适合大数据量
//    - 需要注意编译环境问题，避免使用复杂 STL
//    - 指针操作灵活但需谨慎
//    - 内存管理需要手动处理
// 3. Python 实现特点：
//    - 代码简洁易懂，适合算法验证
//    - 性能相对较差，适合小数据量
//    - 列表操作方便，但需注意内存使用
//    - 递归深度限制较严格
//
// 极端场景与鲁棒性：
// 1. 空输入处理：关键点为空时的特殊处理
// 2. 极端数据规模：关键点数量接近节点总数、树退化为链的情况、深度很大的树结构
// 3. 边界条件：关键点包含根节点、关键点之间存在父子关系、关键点相邻的情况
// 4. 错误处理：需要处理输入错误、参数越界等异常情况
//
// 性能优化策略：
// 1. 算法层面优化：选择合适的虚树构建方法、优化 DP 状态转移方程、预处理减少重复计算
```

```
// 2. 实现层面优化：减少函数调用开销、优化内存访问模式、使用位运算等底层优化技巧
// 3. 工程层面优化：输入输出优化、内存池技术、缓存友好设计
// 4. 多线程优化：对于大规模数据，可以考虑并行处理
//
// 调试技巧与问题定位：
// 1. 中间过程打印：打印 DFS 序、打印 LCA 计算结果、打印虚树构建过程
// 2. 断言验证：验证虚树节点数量上界、验证关键点标记正确性、验证 DP 状态转移正确性
// 3. 性能分析：使用性能分析工具定位瓶颈、对比不同实现的性能差异、分析时间复杂度常数项影响
// 4. 测试用例设计：设计边界测试用例、设计随机测试用例、设计压力测试用例
//
// 工程化考量：
// 1. 可配置性：将算法参数设计为可配置的，提高代码复用性
// 2. 异常处理：添加详细的错误检查和异常处理机制
// 3. 单元测试：编写单元测试确保代码正确性
// 4. 文档化：提供详细的使用说明和 API 文档
// 5. 线程安全：确保代码在多线程环境下正确工作
// 6. 扩展性：设计良好的接口，方便扩展新功能
//
// 适用场景总结：
// 虚树算法适用于以下场景：
// 1. 树上多次询问，每次询问涉及不同的关键点集合
// 2. 每次询问的关键点数量 k 远小于树的总节点数 n
// 3. 需要在关键点及其祖先上进行动态规划或其他操作
// 4. 时间复杂度要求较高，需要优化到 O(n log n + ∑k log k)
//
// 问题判断方法：
// 如何判断一个问题是否适合使用虚树？
// 1. 问题背景是否是树上的多次询问？
// 2. 每次询问是否只涉及部分关键点？
// 3. 是否需要在这些关键点之间的路径上进行操作？
// 4. 数据范围是否要求更高效的算法？
// 如果以上四个问题的答案都是肯定的，那么虚树可能是一个合适的选择。
//
// 与其他算法结合：
// 虚树算法常常与以下算法结合使用：
// 1. 树形 DP：在虚树上进行动态规划是最常见的用法
// 2. 树上倍增：用于快速计算 LCA
// 3. 树链剖分：某些情况下可以结合使用
// 4. 线段树：处理区间查询和更新
// 5. 并查集：处理连通性问题
//
// 递归与非递归实现对比：
// 1. 递归实现：代码简洁，逻辑清晰，但可能受到栈深度限制
```

```
// 2. 非递归实现：避免栈溢出问题，适用于大规模数据，但代码相对复杂
//
// 标准库实现对比：
// 不同语言的标准库对树结构的支持：
// 1. C++：STL 中没有直接的树结构，需要手动实现
// 2. Java：提供 TreeSet、TreeMap 等有序集合
// 3. Python：提供 heapq 等工具，但需要自己实现树结构
//
// 常数项优化：
// 1. 预处理优化：预处理所有可能需要的数据
// 2. 内存访问优化：按顺序访问内存，提高缓存命中率
// 3. 循环优化：减少循环内的操作，合并循环
// 4. 位运算优化：使用位运算替代乘除法
//
// 极端数据测试：
// 1. 空树测试
// 2. 单节点树测试
// 3. 退化为链的树测试
// 4. 完全二叉树测试
// 5. 关键点全部相邻测试
// 6. 关键点数量极大测试
//
// 虚树的局限性：
// 1. 只适用于树上多次询问问题
// 2. 要求∑k 较小
// 3. 构建虚树需要预处理 LCA
// 4. 对于某些问题，可能不如其他算法高效
//
// 虚树的优势：
// 1. 显著降低时间复杂度，从 O(n)到 O(k)
// 2. 代码实现相对简单
// 3. 应用范围广泛
// 4. 常数较小，实际运行效率高
//
// 总结：
// 虚树是一种强大的树上优化技术，通过只保留关键点及其 LCA，将问题规模降低到 O(k)级别。
// 掌握虚树算法需要理解其设计思想、实现细节和应用场景，同时需要注意各种极端情况的处理。
// 通过结合树形 DP 等算法，虚树可以高效解决各种树上多次询问问题。

// 王国和城市，C++版（简化版，避免标准库问题）
// 一共有 n 个节点，给定 n-1 条无向边，所有节点组成一棵树
// 一共有 q 条查询，每条查询格式如下
// 查询 k a1 a2 ... ak : 给出了 k 个不同的重要点，其他点是非重要点
```

```
//                          你可以攻占非重要点，被攻占的点无法通行
//                          要让重要点两两之间不再连通，打印至少需要攻占几个非重要点
//                          如果攻占非重要点无法达成目标，打印-1
// 1 <= n、q <= 10^5
// 1 <= 所有查询给出的点的总数 <= 10^5
// 测试链接 : https://www.luogu.com.cn/problem/CF613D
// 测试链接 : https://codeforces.com/problemset/problem/613/D

#include <cstdio>
#include <algorithm>

const int MAXN = 100001;
const int MAXP = 20;

int n, q, k;

// 原始树
int headg[MAXN], nextg[MAXN << 1], tog[MAXN << 1], cntg;

// 虚树
int headv[MAXN], nextv[MAXN], tov[MAXN], cntv;

// 树上倍增求 LCA + 生成 dfn 序
int dep[MAXN], dfn[MAXN], stjump[MAXN][MAXP], cntd;

// 关键点数组
int arr[MAXN];
// 标记节点是否是关键点
bool isKey[MAXN];

// 第一种建树方式
int tmp[MAXN << 1];
// 第二种建树方式
int stk[MAXN];

// 动态规划相关
// siz[u]，还有几个重要点没和 u 断开，值为 0 或者 1
// cost[u]，表示节点 u 的子树中，做到不违规，至少需要攻占几个非重要点
int siz[MAXN], cost[MAXN];

// 原始树连边
// 函数功能：向原始树中添加无向边
// 参数：
```

```
//   u: 边的一个端点
//   v: 边的另一个端点
// 时间复杂度：O(1)
// 空间复杂度：O(1)
// 注意：由于是无向边，需要在邻接表中添加两个方向的边
void addEdgeG(int u, int v) {
    cntg++;
    nextg[cntg] = headg[u];
    tog[cntg] = v;
    headg[u] = cntg;
}

// 虚树连边
// 函数功能：向虚树中添加有向边
// 参数：
//   u: 边的父节点
//   v: 边的子节点
// 时间复杂度：O(1)
// 空间复杂度：O(1)
// 注意：虚树是有向树，边的方向是从父节点指向子节点
void addEdgeV(int u, int v) {
    cntv++;
    nextv[cntv] = headv[u];
    tov[cntv] = v;
    headv[u] = cntv;
}

// 按 DFS 序排序
// 函数功能：根据节点的 DFS 序对节点数组进行排序
// 参数：
//   nums: 需要排序的节点数组
//   l: 排序的起始位置
//   r: 排序的结束位置
// 时间复杂度：O(k log k)，其中 k 是数组长度
// 空间复杂度：O(log k)，递归栈空间
// 实现细节：使用双指针快速排序算法，比较的是节点的 DFS 序
void sortByDfn(int nums[], int l, int r) {
    if (l >= r) return;
    int i = l, j = r;
    int pivot = nums[(l + r) >> 1];
    while (i <= j) {
        while (dfn[nums[i]] < dfn[pivot]) i++;
        while (dfn[nums[j]] > dfn[pivot]) j--;
```

```
        if (i <= j) {
            int t = nums[i]; nums[i] = nums[j]; nums[j] = t;
            i++; j--;
        }
    }
    sortByDfn(nums, l, j);
    sortByDfn(nums, i, r);
}

// 树上倍增 DFS 预处理
// 函数功能：进行深度优先搜索，预处理每个节点的深度、DFS 序和倍增表
// 参数：
//   u: 当前节点
//   fa: 当前节点的父节点
// 时间复杂度：O(n log n)，其中 n 是节点总数
// 空间复杂度：O(n log n)，存储倍增表
// 实现细节：
//   1. 计算当前节点的深度和 DFS 序
//   2. 填充倍增表，用于后续快速查询 LCA
//   3. 递归处理所有子节点
void dfs(int u, int fa) {
    dep[u] = dep[fa] + 1;  // 深度计算
    cntd++;  // DFS 序计数器
    dfn[u] = cntd;  // 记录节点的 DFS 序
    stjump[u][0] = fa;  // 倍增表第 0 层（直接父节点）
    // 预处理倍增表的其他层
    for (int p = 1; p < MAXP; p++) {
        stjump[u][p] = stjump[stjump[u][p - 1]][p - 1];
    }
    // 递归处理所有子节点
    for (int e = headg[u]; e > 0; e = nextg[e]) {
        if (tog[e] != fa) {  // 避免回父节点
            dfs(tog[e], u);
        }
    }
}

// 计算 LCA（最低公共祖先）
// 函数功能：使用树上倍增法计算两个节点的最低公共祖先
// 参数：
//   a: 第一个节点
//   b: 第二个节点
// 返回值：两个节点的最低公共祖先
```

```
// 时间复杂度：O(log n)
// 空间复杂度：O(1)
// 实现细节：
//   1. 首先将深度较大的节点向上跳，使两个节点处于同一深度
//   2. 然后同时向上跳，直到找到共同的祖先
int getLca(int a, int b) {
    // 确保 a 的深度不小于 b
    if (dep[a] < dep[b]) {
        int t = a; a = b; b = t;
    }
    // 将 a 向上跳，直到与 b 深度相同
    for (int p = MAXP - 1; p >= 0; p--) {
        if (dep[stjump[a][p]] >= dep[b]) {
            a = stjump[a][p];
        }
    }
    // 如果此时 a 和 b 相同，直接返回
    if (a == b) {
        return a;
    }
    // 同时向上跳，直到找到 LCA
    for (int p = MAXP - 1; p >= 0; p--) {
        if (stjump[a][p] != stjump[b][p]) {
            a = stjump[a][p];
            b = stjump[b][p];
        }
    }
    // LCA 是它们的父节点
    return stjump[a][0];
}

// 二次排序法构建虚树
// 函数功能：使用二次排序法构建关键点的虚树
// 步骤：
//   1. 将关键点按 DFS 序排序
//   2. 对相邻关键点求 LCA 并加入序列
//   3. 再次排序并去重得到虚树所有节点
//   4. 连接相邻节点的 LCA
// 返回值：虚树的根节点
// 时间复杂度：O(k log k)
// 空间复杂度：O(k)
// 算法正确性：通过包含所有关键点及其两两 LCA，确保虚树保留了原树中关键点之间的路径关系
int buildVirtualTree1() {
```

```
    // 按 DFS 序排序关键点
    sortByDfn(arr, 1, k);
    int len = 0;
    // 将关键点和它们的 LCA 加入临时数组
    for (int i = 1; i < k; i++) {
        len++;
        tmp[len] = arr[i];
        len++;
        tmp[len] = getLca(arr[i], arr[i + 1]);
    }
    len++;
    tmp[len] = arr[k];
    // 再次排序并去重
    sortByDfn(tmp, 1, len);
    int unique = 1;
    for (int i = 2; i <= len; i++) {
        if (tmp[unique] != tmp[i]) {
            unique++;
            tmp[unique] = tmp[i];
        }
    }
    // 构建虚树
    cntv = 0;  // 重置虚树边计数器
    for (int i = 1; i <= unique; i++) {
        headv[tmp[i]] = 0;  // 清空邻接表
    }
    for (int i = 1; i < unique; i++) {
        // 连接相邻节点的 LCA
        addEdgeV(getLca(tmp[i], tmp[i + 1]), tmp[i + 1]);
    }
    return tmp[1];  // 返回虚树的根节点
}

// 单调栈法构建虚树
// 函数功能：使用单调栈法构建关键点的虚树
// 步骤：
//    1. 将关键点按 DFS 序排序
//    2. 使用栈维护虚树的一条链
//    3. 逐个插入关键点并维护栈结构
// 返回值：虚树的根节点
// 时间复杂度：O(k log k)
// 空间复杂度：O(k)
// 算法正确性：利用栈维护当前处理的链，通过 LCA 判断节点之间的关系，确保虚树的正确性
```

```
int buildVirtualTree2() {
    // 按 DFS 序排序关键点
    sortByDfn(arr, 1, k);
    cntv = 0;  // 重置虚树边计数器
    headv[arr[1]] = 0;  // 清空第一个节点的邻接表
    int top = 0;  // 栈顶指针
    top++;
    stk[top] = arr[1];  // 将第一个节点入栈
    // 处理剩余的关键点
    for (int i = 2; i <= k; i++) {
        int x = arr[i];  // 当前处理的节点
        int y = stk[top];  // 栈顶节点
        int lca = getLca(x, y);  // 计算 LCA
        // 调整栈结构，直到找到合适的位置
        while (top > 1 && dfn[stk[top - 1]] >= dfn[lca]) {
            addEdgeV(stk[top - 1], stk[top]);  // 连接栈顶两个节点
            top--;  // 弹出栈顶
        }
        // 如果 LCA 不是栈顶节点，需要将 LCA 入栈
        if (lca != stk[top]) {
            headv[lca] = 0;  // 清空 LCA 的邻接表
            addEdgeV(lca, stk[top]);  // 连接 LCA 和栈顶节点
            stk[top] = lca;  // 替换栈顶为 LCA
        }
        headv[x] = 0;  // 清空当前节点的邻接表
        top++;
        stk[top] = x;  // 将当前节点入栈
    }
    // 处理栈中剩余的节点
    while (top > 1) {
        addEdgeV(stk[top - 1], stk[top]);  // 连接栈顶两个节点
        top--;  // 弹出栈顶
    }
    return stk[1];  // 返回虚树的根节点
}

// 树形 DP 函数
// 函数功能：在虚树上进行动态规划，计算需要攻占的最少非关键点数量
// 参数：
//   u: 当前处理的节点
// 状态定义：
//   siz[u]: 表示节点 u 的子树中还有多少个未处理的关键点
//   cost[u]: 表示节点 u 的子树中，为了使关键点不连通所需攻占的最少非关键点数量
```

```
// 时间复杂度：O(k)
// 空间复杂度：O(k)，递归栈空间
// 算法正确性：
//   1. 如果当前节点是关键点，则它需要断开与其所有子节点中的关键点的连接
//   2. 如果当前节点是非关键点，且有多个未处理的关键点，则需要攻占该节点
void dp(int u) {
    cost[u] = siz[u] = 0;  // 初始化状态
    // 递归处理所有子节点
    for (int e = headv[u]; e > 0; e = nextv[e]) {
        int v = tov[e];
        dp(v);  // 递归处理子节点
        cost[u] += cost[v];  // 累加子节点的成本
        siz[u] += siz[v];  // 累加子节点的未处理关键点数量
    }
    if (isKey[u]) {  // 如果当前节点是关键点
        cost[u] += siz[u];  // 需要断开所有子树中的关键点
        siz[u] = 1;  // 标记当前节点为需要处理的关键点
    } else if (siz[u] > 1) {  // 如果当前节点非关键，但有多个未处理的关键点
        cost[u]++;  // 攻占该节点
        siz[u] = 0;  // 该节点被攻占后，子树中的关键点都被处理了
    }
    // 注意：如果 siz[u] == 1，则不需要处理，因为这些关键点可以在更高的层次处理
}


// 计算函数
// 函数功能：处理一个查询，计算需要攻占的最少非关键点数量
// 返回值：需要攻占的最少非关键点数量，如果不可能则返回-1
// 时间复杂度：O(k log k)
// 空间复杂度：O(k)
// 实现细节：
//   1. 标记关键点
//   2. 检查是否存在相邻的关键点（这种情况无法通过攻占非关键点解决）
//   3. 构建虚树
//   4. 执行树形 DP
//   5. 清除关键点标记
//   6. 返回结果
int compute() {
    // 标记关键点
    for (int i = 1; i <= k; i++) {
        isKey[arr[i]] = true;
    }
    // 检查是否存在相邻的关键点
```

```
    bool check = true;
    for (int i = 1; i <= k; i++) {
        // 如果一个关键点的父节点也是关键点，则无法通过攻占非关键点使它们不连通
        if (isKey[stjump[arr[i]][0]]) {
            check = false;
            break;
        }
    }
    int ans = -1;  // 默认返回-1 表示不可能
    if (check) {  // 如果不存在相邻的关键点
        int tree = buildVirtualTree1();  // 构建虚树
        // int tree = buildVirtualTree2();  // 也可以使用单调栈法构建虚树
        dp(tree);  // 执行树形 DP
        ans = cost[tree];  // 获取结果
    }
    // 清除关键点标记
    for (int i = 1; i <= k; i++) {
        isKey[arr[i]] = false;
    }
    return ans;  // 返回结果
}


// 主函数
// 处理输入输出，构建原树，执行预处理，并处理每个查询
// 注意：在实际使用中，需要根据具体输入格式进行调整
int main() {
    // 读取节点数 n
    scanf("%d", &n);

    // 初始化原始树
    for (int i = 1; i <= n; i++) {
        headg[i] = 0;
    }
    cntg = 0;

    // 读取 n-1 条边
    for (int i = 1, u, v; i < n; i++) {
        scanf("%d%d", &u, &v);
        addEdgeG(u, v);
        addEdgeG(v, u);
    }
```

```
    // 预处理：DFS 建立倍增表和时间戳
    cntd = 0;
    dfs(1, 0);

    // 读取查询数 q
    scanf("%d", &q);

    // 处理每个查询
    for (int t = 1; t <= q; t++) {
        scanf("%d", &k);

        // 读取 k 个关键点
        for (int i = 1; i <= k; i++) {
            scanf("%d", &arr[i]);
        }

        // 计算并输出结果
        printf("%d
", compute());
    }

    return 0;
}
```

===

文件: Code01_KingdomAndCities1.java

===

```
package class180;

// ==============================================================================
// 虚树(Virtual Tree)算法详解与应用 - 工程化实现与优化
// ==============================================================================
//
// 虚树是一种优化技术，用于解决树上多次询问的问题，每次询问涉及部分关键点
// 虚树只保留关键点及其两两之间的 LCA，节点数控制在 O(k)级别，从而提高效率
//
// 算法核心思想：
// 1. 虚树包含所有关键点和它们两两之间的 LCA
// 2. 虚树的节点数不超过 2*k-1（k 为关键点数）
// 3. 在虚树上进行 DP 等操作，避免遍历整棵树
//
// 构造方法：
```

```
// 方法一：二次排序法
// 1. 将关键点按 DFS 序排序
// 2. 相邻点求 LCA 并加入序列
// 3. 再次排序并去重得到虚树所有节点
// 4. 按照父子关系连接节点
//
// 方法二：单调栈法
// 1. 将关键点按 DFS 序排序
// 2. 用栈维护虚树的一条链
// 3. 逐个插入关键点并维护栈结构
//
// 应用场景：
// 1. 树上多次询问，每次询问涉及部分关键点
// 2. 需要在关键点及其 LCA 上进行 DP 等操作
// 3. 数据范围要求∑k 较小（通常≤10^5）
//
// ============================================================================
// 工程化考量与极端场景处理
// ============================================================================
//
// 1. 异常抛出与边界处理
//    - 空输入：处理关键点列表为空的情况
//    - 单点：单个关键点的特殊处理
//    - 相邻关键点：DFS 序相邻点的 LCA 计算
//    - 重复关键点：去重处理
//
// 2. 性能优化策略
//    - 预处理优化：DFS 序、深度、倍增表等
//    - 内存优化：复用数组，避免频繁分配
//    - 常数优化：减少函数调用，内联关键操作
//
// 3. 线程安全考量
//    - 静态数组：线程不安全，需同步或使用 ThreadLocal
//    - 实例方法：可重入，但需注意状态重置
//
// 4. 调试与测试策略
//    - 单元测试：覆盖边界场景
//    - 性能测试：大规模数据验证
//    - 内存测试：避免内存泄漏
//
// 5. 极端场景鲁棒性
//    - 链状树：最坏情况性能测试
//    - 菊花图：特殊结构验证
```

```
//     - 大规模数据：内存和时间限制测试
//
// =============================================================================
// 复杂度分析与最优解验证
// =============================================================================
//
// 时间复杂度：
//    - 预处理：O(n log n) - 构建 DFS 序和倍增表
//    - 每个查询：O(k log k) - 排序和虚树构建
//    - 总体：O(n log n + ∑k log k)
//
// 空间复杂度：
//    - 预处理：O(n log n) - 倍增表存储
//    - 每个查询：O(k) - 虚树节点存储
//    - 总体：O(n log n + max(k))
//
// 最优解验证：
//    - 理论下界： Ω(k log k) - 排序复杂度
//    - 实际效率：接近理论最优
//    - 替代方案：分块、树链剖分等对比
//
// =============================================================================
//
// 相关题目：
// 1. Codeforces 613D - Kingdom and Cities
//     链接：https://codeforces.com/problemset/problem/613/D
//     题意：给一棵树和多个询问，每个询问给出一些关键点，要求切断最少的非关键点使关键点两两不连通
//     解题思路：构建虚树后，通过树形 DP 计算需要切断的最小非关键点数量
//     时间复杂度：预处理 O(n log n)，每个查询 O(k log k)
//
// 2. 洛谷 P2495 - [SDOI2011]消耗战
//     链接：https://www.luogu.com.cn/problem/P2495
//     题意：给一棵树和多个询问，每个询问给出一些关键点，要求切断最少代价的边使关键点都无法到达根
节点
//     解题思路：构建虚树，树形 DP 时考虑边的最小代价
//     时间复杂度：预处理 O(n log n)，每个查询 O(k log k)
//
// 3. 洛谷 P4103 - [HEOI2014]大工程
//     链接：https://www.luogu.com.cn/problem/P4103
//     题意：给一棵树和多个询问，每个询问给出一些关键点，要求计算所有关键点对之间距离的和、最小值
和最大值
//     解题思路：构建虚树，树形 DP 时维护子树中的关键点信息
//     时间复杂度：预处理 O(n log n)，每个查询 O(k log k)
```

```
//
// 4. 洛谷 P3233 - [HNOI2014]世界树
//    链接：https://www.luogu.com.cn/problem/P3233
//    题意：给一棵树和多个询问，每个询问给出一些关键点，要求计算每个关键点能管理多少个点
//    解题思路：构建虚树，结合倍增和贪心策略
//    时间复杂度：预处理 O(n log n)，每个查询 O(k log k)
//
// 5. Codeforces 1109D - Treeland and Viruses
//    链接：https://codeforces.com/problemset/problem/1109/D
//    题意：给一棵树和多个病毒源点，每个病毒源点以不同速度扩散，求每个点被哪个病毒源点感染
//    解题思路：使用虚树和优先队列优化的广度优先搜索
//    时间复杂度：预处理 O(n log n)，每个查询 O(k log k)
//
// 6. 洛谷 P3320 - [SDOI2015]寻宝游戏
//    链接：https://www.luogu.com.cn/problem/P3320
//    题意：给一棵树和多个操作，每次操作翻转一个点的状态，求收集所有宝藏的最短路径长度
//    解题思路：维护关键点的 DFS 序有序集合，根据虚树周长计算路径长度
//    时间复杂度：O(n log n + m log k)
//
// 7. Codeforces 1000G - Two Melborians, One Siberian
//    链接：https://codeforces.com/problemset/problem/1000/G
//    题意：在树上处理多组询问，涉及关键点的最短距离等信息
//    解题思路：使用虚树优化树上距离查询
//    时间复杂度：预处理 O(n log n)，每个查询 O(k log k)
//
// 8. 牛客网 NC19712 - 树
//    链接：https://ac.nowcoder.com/acm/problem/19712
//    题意：给定一棵树，多次询问多个关键点之间的最长距离
//    解题思路：构建虚树，在虚树上求直径
//    时间复杂度：预处理 O(n log n)，每个查询 O(k log k)
//
// 9. HDU 6621 - K-th Closest Distance
//    链接：http://acm.hdu.edu.cn/showproblem.php?pid=6621
//    题意：树上第 K 近点查询，结合虚树和二分答案
//    解题思路：构建虚树并使用二分答案和树状数组统计
//    时间复杂度：O(n log n + q (log n)^2)
//
// 10. POJ 3728 - The Merchant
//    链接：http://poj.org/problem?id=3728
//    题意：树上多次路径查询，求路径上买卖的最大利润
//    解题思路：预处理结合虚树优化路径查询
//    时间复杂度：预处理 O(n log n)，每个查询 O(k log k)
//
```

```
// 11. SPOJ QTREE5 - Query on a tree V
//     链接：https://www.spoj.com/problems/QTREE5/
//     题意：树上点颜色修改和查询距离最近的白色节点
//     解题思路：使用虚树和优先队列维护最近点
//     时间复杂度：O(n log n + q log n)
//
// 12. LOJ #6056 - 「雅礼集训 2017 Day11」回转寿司
//     链接：https://loj.ac/p/6056
//     题意：涉及树上关键点的查询问题
//     解题思路：构建虚树进行树形 DP
//     时间复杂度：预处理 O(n log n)，每个查询 O(k log k)
//
// 13. AtCoder ABC154F - Many Many Paths
//     链接：https://atcoder.jp/contests/abc154/tasks/abc154_f
//     题意：计算树上路径数量，可以使用虚树优化
//     解题思路：利用虚树减少计算量
//     时间复杂度：O(n log n + q log q)
//
// 14. 洛谷 P5327 - [ZJOI2019]语言
//     链接：https://www.luogu.com.cn/problem/P5327
//     题意：涉及树上路径覆盖的复杂问题
//     解题思路：使用虚树结合线段树维护路径覆盖
//     时间复杂度：O(n log n + q log n)
//
// 15. 杭电 OJ 6957 - Maximal submatrix
//     链接：http://acm.hdu.edu.cn/showproblem.php?pid=6957
//     题意：矩阵相关问题，可以转换为树问题并用虚树优化
//     解题思路：构建虚树并进行动态规划
//     时间复杂度：O(n log n)
//
// 16. 洛谷 P3232 - [HNOI2013]游走
//     链接：https://www.luogu.com.cn/problem/P3232
//     题意：给定无向连通图，通过高斯消元计算边的期望经过次数，再贪心编号使总得分期望最小
//     解题思路：构建虚树并进行概率计算
//     时间复杂度：O(n^3)
//
// 17. UVA 1437 - String painter
//     链接：
https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&page=show_problem&problem=4183
//     题意：字符串染色问题，可转换为树问题使用虚树
//     解题思路：构建虚树并进行区间 DP
//     时间复杂度：O(n^2)
//
```

```
// 18. CodeChef - TREEPATH
//     链接：https://www.codechef.com/problems/TREEPATH
//     题意：树上路径查询问题
//     解题思路：使用虚树优化路径统计
//     时间复杂度：O(n log n + q log q)
//
// 19. HackerEarth - Tree Queries
//     链接：https://www.hackerearth.com/practice/data-structures/trees/binary-and-nary-trees/practice-problems/
//     题意：树上多次查询，涉及关键点的各种统计
//     解题思路：构建虚树并进行相应的统计操作
//     时间复杂度：O(n log n + Σk log k)
//
// 20. 计蒜客 - 树与路径
//     链接：https://nanti.jisuanke.com/t/40733
//     题意：树上路径覆盖问题
//     解题思路：使用虚树和线段树维护覆盖信息
//     时间复杂度：O(n log n + q log n)
//
// 21. Timus OJ 1937 - Chinese Girls' Amusement
//     链接：https://acm.timus.ru/problem.aspx?space=1&num=1937
//     题意：树上游戏问题，涉及关键点的移动
//     解题思路：构建虚树并进行博弈分析
//     时间复杂度：O(n log n + q log q)
//
// 22. Aizu OJ 2600 - Tree with Maximum Cost
//     链接：https://onlinejudge.u-aizu.ac.jp/problems/2600
//     题意：树上最大代价问题，可使用虚树优化
//     解题思路：构建虚树并进行树形 DP
//     时间复杂度：O(n log n + q log q)
//
// 23. Comet OJ - 树上的路径
//     链接：https://cometoj.com/contest/34/problem/D
//     题意：树上路径统计问题
//     解题思路：使用虚树优化路径统计
//     时间复杂度：O(n log n + q log q)
//
// 24. 剑指 Offer - 二叉树中的路径和
//     题意：在二叉树中找出所有和为某一值的路径
//     解题思路：可以扩展使用虚树思想优化路径查找
//     时间复杂度：O(n)
//
// 25. 牛客网 - 编程巅峰赛
```

```
//     链接：https://www.nowcoder.com/contestRoom
//     题意：树上多次查询问题
//     解题思路：构建虚树并进行相应的查询处理
//     时间复杂度：O(n log n + ∑k log k)
//
// 26. MarsCode - Tree Operations
//     题意：树上操作问题，涉及关键点的处理
//     解题思路：使用虚树优化操作处理
//     时间复杂度：O(n log n + q log q)
//
// 27. UVa OJ 12166 - Equilibrium Mobile
//     链接：
https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&page=show_problem&problem=3318
//     题意：平衡树问题，可转换为树问题使用虚树
//     解题思路：构建虚树并进行平衡分析
//     时间复杂度：O(n log n)
//
// 28. 计蒜客 - 线段树练习
//     链接：https://nanti.jisuanke.com/t/T1046
//     题意：线段树相关问题，可结合虚树使用
//     解题思路：虚树结合线段树优化查询
//     时间复杂度：O(n log n + q log n)
//
// 29. 各大高校 OJ - 树上最远点对
//     题意：多次查询树上多个点中的最远点对
//     解题思路：构建虚树并求直径
//     时间复杂度：O(n log n + ∑k log k)
//
// 30. Codeforces 908G - New Year and Original Order
//     链接：https://codeforces.com/problemset/problem/908/G
//     题意：数字序列问题，可转换为树问题使用虚树优化
//     解题思路：构建虚树并进行动态规划
//     时间复杂度：O(n log n)
//
// 时间复杂度分析：
// 预处理阶段：
//    - DFS 遍历计算时间戳、深度、父节点：O(n)
//    - 构建倍增表：O(n log n)，其中 log n 是树的高度
//    - 总预处理时间复杂度：O(n log n)
// 每次查询阶段：
//    - 关键点排序：O(k log k)，k 为关键点数量
//    - 构建虚树：O(k log k)（每次 LCA 查询是 O(log n)，总共 k 次）
//    - 虚树上的动态规划：O(k)（虚树的边数是 O(k)级别的，因为虚树是树结构）
```

```
//   - 总查询时间复杂度：O(k log k)
// 空间复杂度：
//   - 原图存储：O(n)
//   - 倍增表：O(n log n)
//   - 虚树：O(k)
//   - 总空间复杂度：O(n log n)

// 算法设计本质：
// 虚树算法的核心思想是将树上的关键点及其 LCA 节点保留，形成一棵更简洁的树，从而减少计算量。
// 这种优化方法特别适用于处理树上多次关键点查询的问题，通过减少节点数，使得每次查询的处理时间与
// 关键点数量 k 相关，而不是与原树大小 n 相关，极大地提高了查询效率。

// 语言特性差异分析：
// Java 实现特点：
// 1. 使用 ArrayList 作为主要数据结构，支持动态扩展
// 2. 递归 DFS 可能在树深度很大时导致栈溢出，需要注意
// 3. Java 的输入输出需要优化以应对大数据量
// 4. 整数类型使用 int，但对于非常大的数据需要注意溢出问题
//
// 与 C++对比：
// - C++可以使用指针和引用提高效率，而 Java 只能通过对象引用操作
// - C++的 stack、vector 等 STL 容器在性能上通常优于 Java 的集合类
// - C++可以使用位运算优化倍增跳转，Java 也支持但实现稍复杂
// - C++在内存管理上更灵活，可以更精细地控制内存分配
//
// 与 Python 对比：
// - Java 的性能通常优于 Python，特别是在递归深度和大数据处理方面
// - Java 需要显式的类型声明，而 Python 是动态类型
// - Java 的数组和集合操作更高效，但 Python 的代码更简洁
// - Java 的输入输出优化比 Python 更复杂，但性能更好

// 极端场景处理：
// 1. 空输入处理：
//    - 关键点列表为空：返回 0（没有关键点需要隔离）
//    - 单关键点：返回 0（单个点不需要隔离）
//    - 相邻关键点：需要检查父子关系，避免错误计算
//
// 2. 重复关键点处理：
//    - 去重处理：确保每个关键点只出现一次
//    - 排序稳定性：DFS 序排序需要稳定
//
// 3. 树结构极端情况：
//    - 链状树：最坏情况，需要验证倍增表正确性
```

```
//     - 菊花图：中心节点频繁作为LCA，需要优化
//     - 完全二叉树：验证递归深度和栈空间
//
// 4. 大规模数据测试：
//     - n=10^5, k=10^5：验证时间和空间复杂度
//     - 多次查询：验证内存复用和状态重置
//     - 边界值：n=1, k=1 等特殊情况
//
// 5. 性能退化排查：
//     - 递归深度过大：使用迭代 DFS 替代
//     - 内存分配频繁：使用对象池或复用数组
//     - 排序效率：选择合适的排序算法
//
// 6. 调试技巧：
//     - 打印中间变量：验证 DFS 序、深度、LCA 计算
//     - 断言检查：验证关键条件成立
//     - 小规模测试：手动计算验证结果
//
// ============================================================================
// 工程化实现细节
// ============================================================================
//
// 1. 输入输出优化：
//     - 使用 BufferedReader 和 BufferedWriter 处理大规模数据
//     - 避免频繁的字符串分割和转换
//
// 2. 内存管理：
//     - 静态数组复用：避免频繁内存分配
//     - 对象池：对于频繁创建的对象使用池化
//     - 及时释放：处理完查询后及时重置状态
//
// 3. 异常处理：
//     - 输入格式异常：提供清晰的错误信息
//     - 内存溢出：监控内存使用，及时优化
//     - 栈溢出：对于深度树使用迭代 DFS
//
// 4. 可配置性：
//     - 最大节点数可配置：适应不同规模数据
//     - 调试模式开关：控制调试信息输出
//     - 性能监控：记录关键操作耗时
//
// 5. 单元测试：
//     - 边界测试：空输入、单点、两点等
```

```
//     - 功能测试：验证算法正确性
//     - 性能测试：大规模数据验证
//
// ============================================================================
// 与机器学习/深度学习的联系
// ============================================================================
//
// 1. 图神经网络(GNN)应用：
//     - 虚树可以看作是对原图的子图采样
//     - 在 GNN 中用于处理大规模图数据
//     - 减少计算量，提高训练效率
//
// 2. 强化学习中的状态空间压缩：
//     - 虚树思想可以用于状态空间压缩
//     - 保留关键状态，减少搜索空间
//     - 提高强化学习算法的效率
//
// 3. 自然语言处理中的树结构处理：
//     - 语法分析树的关键节点提取
//     - 依存句法树的关键关系保留
//     - 减少计算复杂度，提高处理速度
//
// 4. 图像处理中的层次结构：
//     - 图像分割的层次树结构
//     - 关键区域提取和关系保留
//     - 减少计算量，提高处理效率
//
// ============================================================================
// 反直觉但关键的设计
// ============================================================================
//
// 1. 为什么需要保留 LCA：
//     - 直觉：只保留关键点应该足够
//     - 实际：LCA 节点包含了关键点之间的路径信息
//     - 关键：没有 LCA，无法正确构建虚树的结构
//
// 2. 为什么虚树节点数不超过 2k-1：
//     - 直觉：可能达到 O(k^2)级别
//     - 实际：每个 LCA 最多被加入一次
//     - 关键：DFS 序排序保证了 LCA 的唯一性
//
// 3. 为什么使用单调栈而不是直接构建：
//     - 直觉：直接连接所有节点更简单
```

```
//    - 实际：单调栈保证了虚树的正确结构
//    - 关键：维护了 DFS 序的单调性，确保父子关系正确
//
// ============================================================================
// 代码实现开始
// ============================================================================
// 1. 空输入处理：
//    - 关键点列表为空：返回 0（没有关键点需要隔离）
//    - 单关键点：返回 0（单个点不需要隔离）
//    - 相邻关键点：需要检查父子关系，避免错误计算
//
// 2. 重复关键点处理：
//    - 去重处理：确保每个关键点只出现一次
//    - 排序稳定性：DFS 序排序需要稳定
//
// 3. 树结构极端情况：
//    - 链状树：最坏情况，需要验证倍增表正确性
//    - 菊花图：中心节点频繁作为 LCA，需要优化
//    - 完全二叉树：验证递归深度和栈空间
//
// 4. 大规模数据测试：
//    - n=10^5, k=10^5：验证时间和空间复杂度
//    - 多次查询：验证内存复用和状态重置
//    - 边界值：n=1, k=1 等特殊情况
//
// 5. 性能退化排查：
//    - 递归深度过大：使用迭代 DFS 替代
//    - 内存分配频繁：使用对象池或复用数组
//    - 排序效率：选择合适的排序算法
//
// 6. 调试技巧：
//    - 打印中间变量：验证 DFS 序、深度、LCA 计算
//    - 断言检查：验证关键条件成立
//    - 小规模测试：手动计算验证结果
//
// ============================================================================
// 工程化实现细节
// ============================================================================
//
// 1. 输入输出优化：
//    - 使用 BufferedReader 和 BufferedWriter 处理大规模数据
//    - 避免频繁的字符串分割和转换
//
```

```
// 2. 内存管理：
//     - 静态数组复用：避免频繁内存分配
//     - 对象池：对于频繁创建的对象使用池化
//     - 及时释放：处理完查询后及时重置状态
//
// 3. 异常处理：
//     - 输入格式异常：提供清晰的错误信息
//     - 内存溢出：监控内存使用，及时优化
//     - 栈溢出：对于深度树使用迭代DFS
//
// 4. 可配置性：
//     - 最大节点数可配置：适应不同规模数据
//     - 调试模式开关：控制调试信息输出
//     - 性能监控：记录关键操作耗时
//
// 5. 单元测试：
//     - 边界测试：空输入、单点、两点等
//     - 功能测试：验证算法正确性
//     - 性能测试：大规模数据验证
//
// =============================================================================
// 与机器学习/深度学习的联系
// =============================================================================
//
// 1. 图神经网络(GNN)应用：
//     - 虚树可以看作是对原图的子图采样
//     - 在GNN中用于处理大规模图数据
//     - 减少计算量，提高训练效率
//
// 2. 强化学习中的状态空间压缩：
//     - 虚树思想可以用于状态空间压缩
//     - 保留关键状态，减少搜索空间
//     - 提高强化学习算法的效率
//
// 3. 自然语言处理中的树结构处理：
//     - 语法分析树的关键节点提取
//     - 依存句法树的关键关系保留
//     - 减少计算复杂度，提高处理速度
//
// 4. 图像处理中的层次结构：
//     - 图像分割的层次树结构
//     - 关键区域提取和关系保留
//     - 减少计算量，提高处理效率
```

```
//
// ==========================================================================
// 反直觉但关键的设计
// ==========================================================================
//
// 1. 为什么需要保留 LCA：
//    - 直觉：只保留关键点应该足够
//    - 实际：LCA 节点包含了关键点之间的路径信息
//    - 关键：没有 LCA，无法正确构建虚树的结构
//
// 2. 为什么虚树节点数不超过 2k-1：
//    - 直觉：可能达到 O(k^2)级别
//    - 实际：每个 LCA 最多被加入一次
//    - 关键：DFS 序排序保证了 LCA 的唯一性
//
// 3. 为什么使用单调栈而不是直接构建：
//    - 直觉：直接连接所有节点更简单
//    - 实际：单调栈保证了虚树的正确结构
//    - 关键：维护了 DFS 序的单调性，确保父子关系正确
//
// ==========================================================================
// 代码实现开始
// ==========================================================================
// 1. 当所有节点都是关键点时，虚树退化为原树，此时时间复杂度为 O(n log n)
// 2. 当只有一个关键点时，虚树只包含该节点，时间复杂度为 O(1)
// 3. 对于退化的树（如链状树），LCA 查询和虚树构建仍能高效工作
// 4. 对于非常大的树，需要注意内存限制，可能需要优化数据结构

// 性能优化策略：
// 1. 使用 FastReader 等快速输入类，避免 Scanner 的低效率
// 2. 预分配 ArrayList 的大小，减少动态扩容开销
// 3. 使用链式前向星存储原图，提高访问效率
// 4. 虚树构建完成后及时清理数据，避免内存泄漏
// 5. 对于多次查询，可以缓存部分中间结果

// 调试技巧：
// 1. 打印中间过程：在虚树构建过程中打印栈状态，帮助理解算法流程
// 2. 使用断言：验证 LCA 计算结果的正确性
// 3. 可视化输出：对于小型测试用例，输出虚树的结构
// 4. 性能分析：使用 System.currentTimeMillis()测量各阶段耗时，定位瓶颈

// 工程化考量：
// 1. 异常处理：添加输入验证，确保查询的关键点存在且合法
```

```
// 2. 代码模块化：将LCA、虚树构建、DP等功能封装为独立函数
// 3. 可扩展性：设计接口允许自定义不同的DP策略
// 4. 线程安全：对于多线程环境，需要添加同步机制
// 5. 单元测试：为LCA、虚树构建等核心功能编写测试用例

// 虚树算法的适用场景总结：
// 1. 树上多次询问，每次询问只关注少量关键点
// 2. 需要在关键点之间进行路径统计、覆盖或连通性分析
// 3. 问题可以转化为在关键点构成的虚树上进行动态规划
// 4. 原树规模较大，而每次查询的关键点数量k远小于n

// 如何判断一个问题是否适合使用虚树：
// 1. 是否为树上问题，且有多个独立的查询
// 2. 每个查询是否只涉及少量关键点（k << n）
// 3. 问题是否可以在由关键点构成的子结构上解决
// 4. 原树的预处理是否可以降低每次查询的时间复杂度

// 与其他算法的结合：
// 虚树算法常与以下算法结合使用：
// 1. 树形动态规划（最常见）
// 2. 倍增法（用于LCA查询）
// 3. 线段树或树状数组（处理路径覆盖问题）
// 4. 堆或优先队列（处理最短路径问题）
// 5. 二分答案（处理最优化问题）
//
// 工程化考量：
// 1. 注意虚树边通常没有边权，需要通过原树计算
// 2. 清空关键点标记时避免使用memset，用for循环逐个清除
// 3. 排序后的关键点顺序不是原节点序，如需按原序输出需额外保存
// 4. 虚树主要用于卡常题，需注意常数优化
//
// 算法设计本质与核心思想：
// 1. 设计动机：虚树算法的核心动机是优化树上多次询问问题。当需要对树上不同关键点集合进行多次查询时，
//    如果每次都遍历整棵树，时间复杂度会很高。虚树通过只保留关键点及其LCA，将问题规模从O(n)降低到O(k)。
// 2. 数学原理：
//    - LCA性质：任意两个节点的LCA在DFS序上具有特定性质，可以用于构建虚树
//    - 节点数量上界：虚树节点数不超过2*k-1，这是通过数学归纳法可以证明的
//    - 树的结构保持：虚树保持了原树中关键点之间的祖先关系
// 3. 与其它算法的关联：
//    - 树上倍增：虚树构建需要LCA，通常使用树上倍增算法
//    - 树形DP：虚树上的动态规划是解决问题的核心
```

```
//     - 单调栈：构建虚树时使用的单调栈技巧与其它算法中的单调栈类似
// 4. 工程化应用：
//     - 内存优化：避免使用全局数组清零，用循环逐个清除
//     - 常数优化：选择合适的虚树构建方法（单调栈法通常更快）
//     - 边界处理：正确处理根节点、叶子节点等特殊情况
//
// 语言特性差异与跨语言实现：
// 1. Java 实现特点：
//     - 使用对象封装，代码结构清晰
//     - 自定义 FastReader 提高输入效率
//     - 递归深度可能受限，需要改用迭代实现
// 2. C++实现特点：
//     - 性能最优，适合大数据量
//     - 需要注意编译环境问题，避免使用复杂 STL
//     - 指针操作灵活但需谨慎
// 3. Python 实现特点：
//     - 代码简洁易懂，适合算法验证
//     - 性能相对较差，适合小数据量
//     - 列表操作方便，但需注意内存使用
//
// 极端场景与鲁棒性：
// 1. 空输入处理：关键点为空时的特殊处理
// 2. 极端数据规模：关键点数量接近节点总数、树退化为链的情况、深度很大的树结构
// 3. 边界条件：关键点包含根节点、关键点之间存在父子关系、关键点相邻的情况
//
// 性能优化策略：
// 1. 算法层面优化：选择合适的虚树构建方法、优化 DP 状态转移方程、预处理减少重复计算
// 2. 实现层面优化：减少函数调用开销、优化内存访问模式、使用位运算等底层优化技巧
// 3. 工程层面优化：输入输出优化、内存池技术、缓存友好设计
//
// 调试技巧与问题定位：
// 1. 中间过程打印：打印 DFS 序、打印 LCA 计算结果、打印虚树构建过程
// 2. 断言验证：验证虚树节点数量上界、验证关键点标记正确性、验证 DP 状态转移正确性
// 3. 性能分析：使用性能分析工具定位瓶颈、对比不同实现的性能差异、分析时间复杂度常数项影响

// 王国和城市，java 版
// 一共有 n 个节点，给定 n-1 条无向边，所有节点组成一棵树
// 一共有 q 条查询，每条查询格式如下
// 查询 k a1 a2 ... ak : 给出了 k 个不同的重要点，其他点是非重要点
//                        你可以攻占非重要点，被攻占的点无法通行
//                        要让重要点两两之间不再连通，打印至少需要攻占几个非重要点
//                        如果攻占非重要点无法达成目标，打印-1
// 1 <= n、q <= 10^5
```

```
// 1 <= 所有查询给出的点的总数 <= 10^5
// 测试链接 : https://www.luogu.com.cn/problem/CF613D
// 测试链接 : https://codeforces.com/problemset/problem/613/D
// 提交以下的 code，提交时请把类名改成"Main"，可以通过所有测试用例

import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStreamWriter;
import java.io.PrintWriter;

public class Code01_KingdomAndCities1 {

    public static int MAXN = 100001;
    public static int MAXP = 20;
    public static int n, q, k;

    // 原始树
    public static int[] headg = new int[MAXN];
    public static int[] nextg = new int[MAXN << 1];
    public static int[] tog = new int[MAXN << 1];
    public static int cntg;

    // 虚树
    public static int[] headv = new int[MAXN];
    public static int[] nextv = new int[MAXN];
    public static int[] tov = new int[MAXN];
    public static int cntv;

    // 树上倍增求 LCA + 生成 dfn 序
    public static int[] dep = new int[MAXN];
    public static int[] dfn = new int[MAXN];
    public static int[][] stjump = new int[MAXN][MAXP];
    public static int cntd;

    // 关键点数组
    public static int[] arr = new int[MAXN];
    // 标记节点是否是关键点
    public static boolean[] isKey = new boolean[MAXN];

    // 第一种建树方式
    public static int[] tmp = new int[MAXN << 1];
    // 第二种建树方式
    public static int[] stk = new int[MAXN];
```

```java
// 动态规划相关
// siz[u]，还有几个重要点没和 u 断开，值为 0 或者 1
// cost[u]，表示节点 u 的子树中，做到不违规，至少需要攻占几个非重要点
public static int[] siz = new int[MAXN];
public static int[] cost = new int[MAXN];

/**
 * 原始树连边 - 使用链式前向星存储无向图
 *
 * 链式前向星是一种高效的图存储结构，特别适合处理稀疏图
 * 时间复杂度：O(1) - 单次连边操作
 * 空间复杂度：O(m) - m 为边的数量
 *
 * @param u 边的起点
 * @param v 边的终点
 */
public static void addEdgeG(int u, int v) {
    nextg[++cntg] = headg[u]; // 新边的 next 指针指向当前 u 的第一个边
    tog[cntg] = v;            // 存储目标节点
    headg[u] = cntg;          // u 的头指针更新为新边的索引
}

/**
 * 虚树连边 - 使用链式前向星存储虚树
 *
 * 虚树是原树的一个子结构，只包含关键点及其 LCA
 * 时间复杂度：O(1) - 单次连边操作
 * 空间复杂度：O(k) - k 为虚树节点数量
 *
 * @param u 边的起点（祖先节点）
 * @param v 边的终点（后代节点）
 */
public static void addEdgeV(int u, int v) {
    nextv[++cntv] = headv[u]; // 新边的 next 指针指向当前 u 的第一个边
    tov[cntv] = v;            // 存储目标节点
    headv[u] = cntv;          // u 的头指针更新为新边的索引
}

/**
 * 根据 DFS 序对数组元素进行快速排序
 *
 * 使用双指针快排实现，按照节点的 dfn 序（深度优先搜索时间戳）排序
```

```java
 * 排序后的顺序是虚树构建的基础
 * 时间复杂度：O(m log m) - m 为数组长度
 * 空间复杂度：O(log m) - 递归调用栈空间
 *
 * @param nums 待排序的节点数组
 * @param l 排序的左边界（包含）
 * @param r 排序的右边界（包含）
 */
public static void sortByDfn(int[] nums, int l, int r) {
    if (l >= r) return; // 边界条件：数组长度为 0 或 1 时无需排序
    int i = l, j = r;
    int pivot = nums[(l + r) >> 1]; // 选择中间元素作为基准
    // 双指针分区过程
    while (i <= j) {
        // 找到左边大于等于基准的元素
        while (dfn[nums[i]] < dfn[pivot]) i++;
        // 找到右边小于等于基准的元素
        while (dfn[nums[j]] > dfn[pivot]) j--;
        if (i <= j) {
            // 交换元素
            int tmp = nums[i]; nums[i] = nums[j]; nums[j] = tmp;
            i++; j--;
        }
    }
    // 递归排序左右两个子区间
    sortByDfn(nums, l, j);
    sortByDfn(nums, i, r);
}

/**
 * 深度优先搜索，初始化倍增表和时间戳
 *
 * 该 DFS 完成三个任务：
 * 1. 计算每个节点的深度 dep
 * 2. 分配 DFS 序时间戳 dfn
 * 3. 构建倍增表 stjump 用于快速 LCA 查询
 *
 * 时间复杂度：O(n log n) - n 为节点数，每个节点处理 log n 次倍增跳转
 * 空间复杂度：O(n log n) - 存储倍增表
 *
 * @param u 当前节点
 * @param fa 父节点
 */
```

```java
public static void dfs(int u, int fa) {
    dep[u] = dep[fa] + 1;           // 设置深度（根节点深度为 1）
    dfn[u] = ++cntd;                // 分配 DFS 时间戳
    stjump[u][0] = fa;              // 2^0 级祖先即直接父节点
    // 构建倍增表：stjump[u][p]表示 u 的 2^p 级祖先
    for (int p = 1; p < MAXP; p++) {
        stjump[u][p] = stjump[stjump[u][p - 1]][p - 1];
    }
    // 遍历所有子节点
    for (int e = headg[u]; e > 0; e = nextg[e]) {
        if (tog[e] != fa) { // 避免回父节点
            dfs(tog[e], u); // 递归处理子树
        }
    }
}

/**
 * 使用树上倍增法计算两个节点的最低公共祖先(LCA)
 *
 * LCA 算法步骤：
 * 1. 先将较深的节点提升到较浅节点的深度
 * 2. 然后同时提升两个节点，直到找到共同祖先
 *
 * 时间复杂度：O(log n) - 每次查询需要 O(log n)次跳转操作
 * 空间复杂度：O(1) - 只使用常数额外空间
 *
 * @param a 第一个节点
 * @param b 第二个节点
 * @return a 和 b 的最低公共祖先
 */
public static int getLca(int a, int b) {
    // 确保 a 是深度较大的节点
    if (dep[a] < dep[b]) {
        int tmp = a; a = b; b = tmp;
    }
    // 第一步：将 a 提升到与 b 相同的深度
    for (int p = MAXP - 1; p >= 0; p--) {
        if (dep[stjump[a][p]] >= dep[b]) {
            a = stjump[a][p];
        }
    }
    // 如果此时 a 等于 b，则已经是 LCA
    if (a == b) {
```

```java
        return a;
    }
    // 第二步：同时提升 a 和 b，直到它们的父节点是共同祖先
    for (int p = MAXP - 1; p >= 0; p--) {
        if (stjump[a][p] != stjump[b][p]) {
            a = stjump[a][p];
            b = stjump[b][p];
        }
    }
    // 最终 LCA 是当前节点的父节点
    return stjump[a][0];
}

/**
 * 二次排序法构建虚树
 *
 * 算法步骤：
 * 1. 将关键点按 DFS 序排序
 * 2. 对于每对相邻关键点，计算它们的 LCA 并加入临时数组
 * 3. 对临时数组进行排序并去重
 * 4. 连接相邻节点的 LCA 形成虚树
 *
 * 时间复杂度：O(k log k) - k 为关键点数，排序需要 O(k log k)，LCA 查询需要 O(k log n)
 * 空间复杂度：O(k) - 存储临时数组和虚树结构
 *
 * @return 虚树的根节点
 */
public static int buildVirtualTree1() {
    // 第一步：按 DFS 序对关键点排序
    sortByDfn(arr, 1, k);
    int len = 0;
    // 第二步：添加相邻关键点及其 LCA 到临时数组
    for (int i = 1; i < k; i++) {
        tmp[++len] = arr[i];
        tmp[++len] = getLca(arr[i], arr[i + 1]);
    }
    tmp[++len] = arr[k];
    // 第三步：对临时数组按 DFS 序排序
    sortByDfn(tmp, 1, len);
    // 第四步：去重
    int unique = 1;
    for (int i = 2; i <= len; i++) {
        if (tmp[unique] != tmp[i]) {
```

```
            tmp[++unique] = tmp[i];
        }
    }
    // 第五步：初始化虚树结构
    cntv = 0;
    for (int i = 1; i <= unique; i++) {
        headv[tmp[i]] = 0; // 清空之前的边
    }
    // 第六步：构建虚树边
    for (int i = 1; i < unique; i++) {
        // 对于排序后的相邻节点，它们的 LCA 是它们的直接祖先
        addEdgeV(getLca(tmp[i], tmp[i + 1]), tmp[i + 1]);
    }
    return tmp[1]; // 返回虚树的根（第一个节点）
}

/**
 * 单调栈法构建虚树
 *
 * 算法步骤：
 * 1. 将关键点按 DFS 序排序
 * 2. 使用栈维护虚树的一条链（当前处理的最右链）
 * 3. 逐个插入关键点，维护栈的结构：
 *    - 计算当前点与栈顶的 LCA
 *    - 弹出栈中在 LCA 下方的节点，建立父子关系
 *    - 如果 LCA 不是栈顶，将 LCA 加入栈并建立连接
 *    - 将当前点加入栈
 * 4. 处理栈中剩余节点，建立父子关系
 *
 * 时间复杂度：O(k log k) - k 为关键点数，排序需要 O(k log k)，每个节点入栈出栈一次
 * 空间复杂度：O(k) - 存储栈和虚树结构
 *
 * 该方法通常比二次排序法更高效，常数更小
 *
 * @return 虚树的根节点
 */
public static int buildVirtualTree2() {
    // 第一步：按 DFS 序对关键点排序
    sortByDfn(arr, 1, k);
    // 初始化虚树结构
    cntv = 0;
    headv[arr[1]] = 0;
    // 使用栈维护当前链
```

```
    int top = 0;
    stk[++top] = arr[1];
    // 逐个处理关键点
    for (int i = 2; i <= k; i++) {
        int x = arr[i];
        int y = stk[top];
        // 计算当前点与栈顶的 LCA
        int lca = getLca(x, y);
        // 弹出栈中在 LCA 下方的节点，建立父子关系
        while (top > 1 && dfn[stk[top - 1]] >= dfn[lca]) {
            addEdgeV(stk[top - 1], stk[top]);
            top--;
        }
        // 如果 LCA 不是栈顶，需要将 LCA 加入栈并建立连接
        if (lca != stk[top]) {
            headv[lca] = 0;
            addEdgeV(lca, stk[top]);
            stk[top] = lca; // 替换栈顶为 LCA
        }
        // 将当前点加入栈
        headv[x] = 0;
        stk[++top] = x;
    }
    // 处理栈中剩余节点，建立父子关系
    while (top > 1) {
        addEdgeV(stk[top - 1], stk[top]);
        top--;
    }
    return stk[1]; // 返回虚树的根
}

/**
 * 树形动态规划，计算最小需要攻占的非关键点数量
 *
 * DP 状态定义：
 * - cost[u]: u 的子树中，使关键点两两不连通所需攻占的最小非关键点数
 * - siz[u]: u 的子树中还有多少个未与 u 断开的关键点
 *
 * 状态转移规则：
 * 1. 如果 u 是关键点：需要断开所有子树中的关键点连接，cost[u] += siz[u]，siz[u] = 1
 * 2. 如果 u 不是关键点且有多个子树包含关键点：需要攻占 u，cost[u]++，siz[u] = 0
 * 3. 如果 u 不是关键点且只有一个子树包含关键点：不需要攻占 u，siz[u] = 1
 * 4. 如果 u 不是关键点且没有子树包含关键点：siz[u] = 0
```

```java
 *
 * 时间复杂度：O(k) - k为虚树节点数，每个节点只被访问一次
 * 空间复杂度：O(k) - 递归调用栈深度
 *
 * @param u 当前节点
 */
public static void dp(int u) {
    // 初始化当前节点的cost和siz
    cost[u] = siz[u] = 0;
    // 遍历所有子节点
    for (int e = headv[u]; e > 0; e = nextv[e]) {
        int v = tov[e];
        dp(v); // 递归处理子节点
        cost[u] += cost[v]; // 累加子树的cost
        siz[u] += siz[v];   // 累加子树的siz
    }
    // 根据当前节点类型和siz值进行状态转移
    if (isKey[u]) {
        // 如果是关键点，需要断开所有子树中的关键点连接
        cost[u] += siz[u]; // 每个子树中的关键点都需要一个断开操作
        siz[u] = 1;         // 关键点本身未被断开
    } else if (siz[u] > 1) {
        // 如果不是关键点但有多个子树包含关键点，需要攻占当前节点
        cost[u]++;  // 攻占当前节点
        siz[u] = 0; // 攻占后，所有子树的关键点都被断开
    }
    // else if (siz[u] == 1) 无需攻占，siz保持为1
    // else (siz[u] == 0) 无需处理
}

/**
 * 计算最小需要攻占的非关键点数量
 *
 * 处理流程：
 * 1. 标记关键点
 * 2. 检查合法性：如果有关键点和其父节点都是关键点，则无法通过攻占非关键点来隔开
 * 3. 构建虚树（选择使用buildVirtualTree1或buildVirtualTree2）
 * 4. 在虚树上进行动态规划
 * 5. 清除关键点标记，避免影响后续查询
 *
 * 时间复杂度：O(k log k) - k为关键点数
 * 空间复杂度：O(k) - 存储虚树和DP状态
 *
```

```java
 * @return 最小需要攻占的非关键点数，若无法达成则返回-1
 */
public static int compute() {
    // 第一步：标记关键点
    for (int i = 1; i <= k; i++) {
        isKey[arr[i]] = true;
    }
    // 第二步：检查合法性
    boolean check = true;
    for (int i = 1; i <= k; i++) {
        // 如果关键点和其父节点都是关键点，无法通过攻占非关键点来隔开
        if (isKey[stjump[arr[i]][0]]) {
            check = false;
            break;
        }
    }
    int ans = -1;
    if (check) {
        // 第三步：构建虚树
        int tree = buildVirtualTree1();
        // 也可以使用单调栈法：int tree = buildVirtualTree2();

        // 第四步：执行树形 DP
        dp(tree);
        ans = cost[tree];
    }
    // 第五步：清除关键点标记（重要！避免影响后续查询）
    for (int i = 1; i <= k; i++) {
        isKey[arr[i]] = false;
    }
    return ans;
}

public static void main(String[] args) throws Exception {
    FastReader in = new FastReader(System.in);
    PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));
    n = in.nextInt();
    for (int i = 1, u, v; i < n; i++) {
        u = in.nextInt();
        v = in.nextInt();
        addEdgeG(u, v);
        addEdgeG(v, u);
    }
```

```
        dfs(1, 0);
        q = in.nextInt();
        for (int t = 1; t <= q; t++) {
            k = in.nextInt();
            for (int i = 1; i <= k; i++) {
                arr[i] = in.nextInt();
            }
            out.println(compute());
        }
        out.flush();
        out.close();
    }

    /**
     * 快速输入工具类
     *
     * 比 Scanner 更快的输入方式，适用于大数据量输入
     * 使用缓冲区和字节操作提高效率
     *
     * 时间复杂度：O(1) per read operation (amortized)
     * 空间复杂度：O(1) - 固定大小的缓冲区
     */
    static class FastReader {
        private final byte[] buffer; // 输入缓冲区
        private int ptr;             // 当前读取位置
        private int len;             // 当前缓冲区长度
        private final InputStream in; // 输入流

        /**
         * 构造函数
         * @param in 输入流
         */
        FastReader(InputStream in) {
            this.in = in;
            this.buffer = new byte[1 << 16]; // 64KB 缓冲区
            this.ptr = 0;
            this.len = 0;
        }

        /**
         * 读取单个字节
         * @return 读取的字节值，EOF 返回-1
         * @throws IOException 输入异常
```

```
         */
        private int readByte() throws IOException {
            if (ptr >= len) {
                // 缓冲区耗尽，重新填充
                len = in.read(buffer);
                ptr = 0;
                if (len <= 0) // EOF
                    return -1;
            }
            return buffer[ptr++];
        }

        /**
         * 读取下一个整数
         * @return 读取的整数值
         * @throws IOException 输入异常
         */
        int nextInt() throws IOException {
            int c;
            // 跳过空白字符
            do {
                c = readByte();
            } while (c <= ' ' && c != -1);

            // 处理符号
            boolean neg = false;
            if (c == '-') {
                neg = true;
                c = readByte();
            }

            // 解析数字
            int val = 0;
            while (c > ' ' && c != -1) {
                val = val * 10 + (c - '0');
                c = readByte();
            }
            return neg ? -val : val;
        }
    }

}
```

==

文件: Code01_KingdomAndCities1.py
==
虚树(Virtual Tree)算法详解与应用
#
虚树是一种优化技术，用于解决树上多次询问的问题，每次询问涉及部分关键点
虚树只保留关键点及其两两之间的 LCA，节点数控制在 O(k)级别，从而提高效率
#
算法核心思想：
1. 虚树包含所有关键点和它们两两之间的 LCA
2. 虚树的节点数不超过 2*k-1（k 为关键点数）
3. 在虚树上进行 DP 等操作，避免遍历整棵树
#
构造方法：
方法一：二次排序法
1. 将关键点按 DFS 序排序
2. 相邻点求 LCA 并加入序列
3. 再次排序并去重得到虚树所有节点
4. 按照父子关系连接节点
#
方法二：单调栈法
1. 将关键点按 DFS 序排序
2. 用栈维护虚树的一条链
3. 逐个插入关键点并维护栈结构
#
应用场景：
1. 树上多次询问，每次询问涉及部分关键点
2. 需要在关键点及其 LCA 上进行 DP 等操作
3. 数据范围要求∑k 较小（通常≤10^5）
#
相关题目：
1. Codeforces 613D - Kingdom and Cities
链接：https://codeforces.com/problemset/problem/613/D
题意：给一棵树和多个询问，每个询问给出一些关键点，要求切断最少的非关键点使关键点两两不连通
解题思路：构建虚树后，通过树形 DP 计算需要切断的最小非关键点数量
时间复杂度：预处理 O(n log n)，每个查询 O(k log k)
#
2. 洛谷 P2495 - [SDOI2011]消耗战
链接：https://www.luogu.com.cn/problem/P2495
题意：给一棵树和多个询问，每个询问给出一些关键点，要求切断最少代价的边使关键点都无法到达根节点
解题思路：构建虚树，树形 DP 时考虑边的最小代价

```
#     时间复杂度：预处理 O(n log n)，每个查询 O(k log k)
#
# 3. 洛谷 P4103 - [HEOI2014]大工程
#     链接：https://www.luogu.com.cn/problem/P4103
#     题意：给一棵树和多个询问，每个询问给出一些关键点，要求计算所有关键点对之间距离的和、最小值和最大值
#     解题思路：构建虚树，树形 DP 时维护子树中的关键点信息
#     时间复杂度：预处理 O(n log n)，每个查询 O(k log k)
#
# 4. 洛谷 P3233 - [HNOI2014]世界树
#     链接：https://www.luogu.com.cn/problem/P3233
#     题意：给一棵树和多个询问，每个询问给出一些关键点，要求计算每个关键点能管理多少个点
#     解题思路：构建虚树，结合倍增和贪心策略
#     时间复杂度：预处理 O(n log n)，每个查询 O(k log k)
#
# 5. Codeforces 1109D - Treeland and Viruses
#     链接：https://codeforces.com/problemset/problem/1109/D
#     题意：给一棵树和多个病毒源点，每个病毒源点以不同速度扩散，求每个点被哪个病毒源点感染
#     解题思路：使用虚树和优先队列优化的广度优先搜索
#     时间复杂度：预处理 O(n log n)，每个查询 O(k log k)
#
# 6. 洛谷 P3320 - [SDOI2015]寻宝游戏
#     链接：https://www.luogu.com.cn/problem/P3320
#     题意：给一棵树和多个操作，每次操作翻转一个点的状态，求收集所有宝藏的最短路径长度
#     解题思路：维护关键点的 DFS 序有序集合，根据虚树周长计算路径长度
#     时间复杂度：O(n log n + m log k)
#
# 7. Codeforces 1000G - Two Melborians, One Siberian
#     链接：https://codeforces.com/problemset/problem/1000/G
#     题意：在树上处理多组询问，涉及关键点的最短距离等信息
#     解题思路：使用虚树优化树上距离查询
#     时间复杂度：预处理 O(n log n)，每个查询 O(k log k)
#
# 8. 牛客网 NC19712 - 树
#     链接：https://ac.nowcoder.com/acm/problem/19712
#     题意：给定一棵树，多次询问多个关键点之间的最长距离
#     解题思路：构建虚树，在虚树上求直径
#     时间复杂度：预处理 O(n log n)，每个查询 O(k log k)
#
# 9. HDU 6621 - K-th Closest Distance
#     链接：http://acm.hdu.edu.cn/showproblem.php?pid=6621
#     题意：树上第 K 近点查询，结合虚树和二分答案
#     解题思路：构建虚树并使用二分答案和树状数组统计
```

```
#     时间复杂度：O(n log n + q (log n)^2)
#
# 10. POJ 3728 - The Merchant
#     链接：http://poj.org/problem?id=3728
#     题意：树上多次路径查询，求路径上买卖的最大利润
#     解题思路：预处理结合虚树优化路径查询
#     时间复杂度：预处理 O(n log n)，每个查询 O(k log k)
#
# 11. SPOJ QTREE5 - Query on a tree V
#     链接：https://www.spoj.com/problems/QTREE5/
#     题意：树上点颜色修改和查询距离最近的白色节点
#     解题思路：使用虚树和优先队列维护最近点
#     时间复杂度：O(n log n + q log n)
#
# 12. LOJ #6056 - 「雅礼集训 2017 Day11」回转寿司
#     链接：https://loj.ac/p/6056
#     题意：涉及树上关键点的查询问题
#     解题思路：构建虚树进行树形 DP
#     时间复杂度：预处理 O(n log n)，每个查询 O(k log k)
#
# 13. AtCoder ABC154F - Many Many Paths
#     链接：https://atcoder.jp/contests/abc154/tasks/abc154_f
#     题意：计算树上路径数量，可以使用虚树优化
#     解题思路：利用虚树减少计算量
#     时间复杂度：O(n log n + q log q)
#
# 14. 洛谷 P5327 - [ZJOI2019]语言
#     链接：https://www.luogu.com.cn/problem/P5327
#     题意：涉及树上路径覆盖的复杂问题
#     解题思路：使用虚树结合线段树维护路径覆盖
#     时间复杂度：O(n log n + q log n)
#
# 15. 杭电 OJ 6957 - Maximal submatrix
#     链接：http://acm.hdu.edu.cn/showproblem.php?pid=6957
#     题意：矩阵相关问题，可以转换为树问题并用虚树优化
#     解题思路：构建虚树并进行动态规划
#     时间复杂度：O(n log n)
#
# 16. 洛谷 P3232 - [HNOI2013]游走
#     链接：https://www.luogu.com.cn/problem/P3232
#     题意：给定无向连通图，通过高斯消元计算边的期望经过次数，再贪心编号使总得分期望最小
#     解题思路：构建虚树并进行概率计算
#     时间复杂度：O(n^3)
```

```
#
# 17. UVA 1437 - String painter
#     链接：
https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&page=show_problem&problem=4183
#     题意：字符串染色问题，可转换为树问题使用虚树
#     解题思路：构建虚树并进行区间 DP
#     时间复杂度：O(n^2)
#
# 18. CodeChef - TREEPATH
#     链接：https://www.codechef.com/problems/TREEPATH
#     题意：树上路径查询问题
#     解题思路：使用虚树优化路径统计
#     时间复杂度：O(n log n + q log q)
#
# 19. HackerEarth - Tree Queries
#     链接：https://www.hackerearth.com/practice/data-structures/trees/binary-and-nary-
trees/practice-problems/
#     题意：树上多次查询，涉及关键点的各种统计
#     解题思路：构建虚树并进行相应的统计操作
#     时间复杂度：O(n log n + Σk log k)
#
# 20. 计蒜客 - 树与路径
#     链接：https://nanti.jisuanke.com/t/40733
#     题意：树上路径覆盖问题
#     解题思路：使用虚树和线段树维护覆盖信息
#     时间复杂度：O(n log n + q log n)
#
# 21. Timus OJ 1937 - Chinese Girls’ Amusement
#     链接：https://acm.timus.ru/problem.aspx?space=1&num=1937
#     题意：树上游戏问题，涉及关键点的移动
#     解题思路：构建虚树并进行博弈分析
#     时间复杂度：O(n log n + q log q)
#
# 22. Aizu OJ 2600 - Tree with Maximum Cost
#     链接：https://onlinejudge.u-aizu.ac.jp/problems/2600
#     题意：树上最大代价问题，可使用虚树优化
#     解题思路：构建虚树并进行树形 DP
#     时间复杂度：O(n log n + q log q)
#
# 23. Comet OJ - 树上的路径
#     链接：https://cometoj.com/contest/34/problem/D
#     题意：树上路径统计问题
#     解题思路：使用虚树优化路径统计
```

```
#     时间复杂度：O(n log n + q log q)
#
# 24. 剑指 Offer - 二叉树中的路径和
#     题意：在二叉树中找出所有和为某一值的路径
#     解题思路：可以扩展使用虚树思想优化路径查找
#     时间复杂度：O(n)
#
# 25. 牛客网 - 编程巅峰赛
#     链接：https://www.nowcoder.com/contestRoom
#     题意：树上多次查询问题
#     解题思路：构建虚树并进行相应的查询处理
#     时间复杂度：O(n log n + Σk log k)
#
# 26. MarsCode - Tree Operations
#     题意：树上操作问题，涉及关键点的处理
#     解题思路：使用虚树优化操作处理
#     时间复杂度：O(n log n + q log q)
#
# 27. UVa OJ 12166 - Equilibrium Mobile
#     链接：
https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&page=show_problem&problem=3318
#     题意：平衡树问题，可转换为树问题使用虚树
#     解题思路：构建虚树并进行平衡分析
#     时间复杂度：O(n log n)
#
# 28. 计蒜客 - 线段树练习
#     链接：https://nanti.jisuanke.com/t/T1046
#     题意：线段树相关问题，可结合虚树使用
#     解题思路：虚树结合线段树优化查询
#     时间复杂度：O(n log n + q log n)
#
# 29. 各大高校 OJ - 树上最远点对
#     题意：多次查询树上多个点中的最远点对
#     解题思路：构建虚树并求直径
#     时间复杂度：O(n log n + Σk log k)
#
# 30. Codeforces 908G - New Year and Original Order
#     链接：https://codeforces.com/problemset/problem/908/G
#     题意：数字序列问题，可转换为树问题使用虚树优化
#     解题思路：构建虚树并进行动态规划
#     时间复杂度：O(n log n)
#
# 时间复杂度分析：
```

```
# 预处理阶段：
#   - DFS 遍历计算时间戳、深度、父节点：O(n)
#   - 构建倍增表：O(n log n)，其中 log n 是树的高度
#   - 总预处理时间复杂度：O(n log n)
# 每次查询阶段：
#   - 关键点排序：O(k log k)，k 为关键点数量
#   - 构建虚树：O(k log k)（每次 LCA 查询是 O(log n)，总共 k 次）
#   - 虚树上的动态规划：O(k)（虚树的边数是 O(k)级别的，因为虚树是树结构）
#   - 总查询时间复杂度：O(k log k)
# 空间复杂度：
#   - 原图存储：O(n)
#   - 倍增表：O(n log n)
#   - 虚树：O(k)
#   - 总空间复杂度：O(n log n)
#
# 工程化考量：
# 1. 注意虚树边通常没有边权，需要通过原树计算
# 2. 清空关键点标记时避免使用 memset，用 for 循环逐个清除
# 3. 排序后的关键点顺序不是原节点序，如需按原序输出需额外保存
# 4. 虚树主要用于卡常题，需注意常数优化
# 5. Python 实现需要特别注意递归深度限制，可能需要改用迭代实现 DFS
# 6. 对于大数据量输入，Python 的标准输入方式可能较慢，需要优化
#
# 算法设计本质与核心思想：
# 1. 设计动机：虚树算法的核心动机是优化树上多次询问问题。当需要对树上不同关键点集合进行多次查询时，
#    如果每次都遍历整棵树，时间复杂度会很高。虚树通过只保留关键点及其 LCA，将问题规模从 O(n)降低到 O(k)。
# 2. 数学原理：
#    - LCA 性质：任意两个节点的 LCA 在 DFS 序上具有特定性质，可以用于构建虚树
#    - 节点数量上界：虚树节点数不超过 2*k-1，这是通过数学归纳法可以证明的
#    - 树的结构保持：虚树保持了原树中关键点之间的祖先关系
# 3. 与其它算法的关联：
#    - 树上倍增：虚树构建需要 LCA，通常使用树上倍增算法
#    - 树形 DP：虚树上的动态规划是解决问题的核心
#    - 单调栈：构建虚树时使用的单调栈技巧与其它算法中的单调栈类似
# 4. 工程化应用：
#    - 内存优化：避免使用全局数组清零，用循环逐个清除
#    - 常数优化：选择合适的虚树构建方法（单调栈法通常更快）
#    - 边界处理：正确处理根节点、叶子节点等特殊情况
#
# 语言特性差异与跨语言实现：
# 1. Java 实现特点：
```

```
#    - 使用对象封装，代码结构清晰
#    - 自定义 FastReader 提高输入效率
#    - 递归深度可能受限，需要改用迭代实现
# 2. C++实现特点：
#    - 性能最优，适合大数据量
#    - 需要注意编译环境问题，避免使用复杂 STL
#    - 指针操作灵活但需谨慎
# 3. Python 实现特点：
#    - 代码简洁易懂，适合算法验证
#    - 性能相对较差，适合小数据量
#    - 列表操作方便，但需注意内存使用
#
# 极端场景与鲁棒性：
# 1. 空输入处理：关键点为空时的特殊处理
# 2. 极端数据规模：关键点数量接近节点总数、树退化为链的情况、深度很大的树结构
# 3. 边界条件：关键点包含根节点、关键点之间存在父子关系、关键点相邻的情况
#
# 性能优化策略：
# 1. 算法层面优化：选择合适的虚树构建方法、优化 DP 状态转移方程、预处理减少重复计算
# 2. 实现层面优化：减少函数调用开销、优化内存访问模式、使用位运算等底层优化技巧
# 3. 工程层面优化：输入输出优化、内存池技术、缓存友好设计
#
# 调试技巧与问题定位：
# 1. 中间过程打印：打印 DFS 序、打印 LCA 计算结果、打印虚树构建过程
# 2. 断言验证：验证虚树节点数量上界、验证关键点标记正确性、验证 DP 状态转移正确性
# 3. 性能分析：使用性能分析工具定位瓶颈、对比不同实现的性能差异、分析时间复杂度常数项影响

# 王国和城市，Python 版
# 一共有 n 个节点，给定 n-1 条无向边，所有节点组成一棵树
# 一共有 q 条查询，每条查询格式如下
# 查询 k a1 a2 ... ak : 给出了 k 个不同的重要点，其他点是非重要点
#                       你可以攻占非重要点，被攻占的点无法通行
#                       要让重要点两两之间不再连通，打印至少需要攻占几个非重要点
#                       如果攻占非重要点无法达成目标，打印-1
# 1 <= n、q <= 10^5
# 1 <= 所有查询给出的点的总数 <= 10^5
# 测试链接 : https://www.luogu.com.cn/problem/CF613D
# 测试链接 : https://codeforces.com/problemset/problem/613/D

import sys
from collections import deque

MAXN = 100001
```

```
MAXP = 20

# 全局变量
n, q, k = 0, 0, 0

# 原始树
headg = [0] * MAXN
nextg = [0] * (MAXN << 1)
tog = [0] * (MAXN << 1)
cntg = 0

# 虚树
headv = [0] * MAXN
nextv = [0] * MAXN
tov = [0] * MAXN
cntv = 0

# 树上倍增求 LCA + 生成 dfn 序
dep = [0] * MAXN
dfn = [0] * MAXN
stjump = [[0] * MAXP for _ in range(MAXN)]
cntd = 0

# 关键点数组
arr = [0] * MAXN
# 标记节点是否是关键点
isKey = [False] * MAXN

# 第一种建树方式
tmp = [0] * (MAXN << 1)
# 第二种建树方式
stk = [0] * MAXN

# 动态规划相关
# siz[u]，还有几个重要点没和 u 断开，值为 0 或者 1
# cost[u]，表示节点 u 的子树中，做到不违规，至少需要攻占几个非重要点
siz = [0] * MAXN
cost = [0] * MAXN

# 原始树连边
# 使用链式前向星存储无向图
# 时间复杂度：O(1) - 单次连边操作
# 空间复杂度：O(m) - m 为边的数量
```

```python
def addEdgeG(u, v):
    global cntg
    cntg += 1
    nextg[cntg] = headg[u]  # 新边的 next 指针指向当前 u 的第一个边
    tog[cntg] = v           # 存储目标节点
    headg[u] = cntg         # u 的头指针更新为新边的索引

# 虚树连边
# 使用链式前向星存储虚树
# 时间复杂度：O(1) - 单次连边操作
# 空间复杂度：O(k) - k 为虚树节点数量
def addEdgeV(u, v):
    global cntv
    cntv += 1
    nextv[cntv] = headv[u]  # 新边的 next 指针指向当前 u 的第一个边
    tov[cntv] = v           # 存储目标节点
    headv[u] = cntv         # u 的头指针更新为新边的索引

# 根据 DFS 序对数组元素进行快速排序
# 使用双指针快排实现，按照节点的 dfn 序（深度优先搜索时间戳）排序
# 排序后的顺序是虚树构建的基础
# 时间复杂度：O(m log m) - m 为数组长度
# 空间复杂度：O(log m) - 递归调用栈空间
def sortByDfn(nums, l, r):
    if l >= r:
        return  # 边界条件：数组长度为 0 或 1 时无需排序
    i, j = l, r
    pivot = nums[(l + r) >> 1]  # 选择中间元素作为基准
    # 双指针分区过程
    while i <= j:
        # 找到左边大于等于基准的元素
        while dfn[nums[i]] < dfn[pivot]:
            i += 1
        # 找到右边小于等于基准的元素
        while dfn[nums[j]] > dfn[pivot]:
            j -= 1
        if i <= j:
            # 交换元素
            nums[i], nums[j] = nums[j], nums[i]
            i += 1
            j -= 1
    # 递归排序左右两个子区间
    sortByDfn(nums, l, j)
```

```python
    sortByDfn(nums, i, r)

# 深度优先搜索，初始化倍增表和时间戳
# 该 DFS 完成三个任务：
# 1. 计算每个节点的深度 dep
# 2. 分配 DFS 序时间戳 dfn
# 3. 构建倍增表 stjump 用于快速 LCA 查询
# 时间复杂度：O(n log n) - n 为节点数，每个节点处理 log n 次倍增跳转
# 空间复杂度：O(n log n) - 存储倍增表
# 注意：Python 的递归深度限制可能导致在大规模树上栈溢出，可能需要改为迭代实现
def dfs(u, fa):
    global cntd
    dep[u] = dep[fa] + 1           # 设置深度（根节点深度为 1）
    cntd += 1
    dfn[u] = cntd                  # 分配 DFS 时间戳
    stjump[u][0] = fa              # 2^0 级祖先即直接父节点
    # 构建倍增表：stjump[u][p]表示 u 的 2^p 级祖先
    for p in range(1, MAXP):
        stjump[u][p] = stjump[stjump[u][p - 1]][p - 1]
    # 遍历所有子节点
    e = headg[u]
    while e > 0:
        if tog[e] != fa:  # 避免回父节点
            dfs(tog[e], u)  # 递归处理子树
        e = nextg[e]

# 使用树上倍增法计算两个节点的最低公共祖先(LCA)
# LCA 算法步骤：
# 1. 先将较深的节点提升到较浅节点的深度
# 2. 然后同时提升两个节点，直到找到共同祖先
# 时间复杂度：O(log n) - 每次查询需要 O(log n)次跳转操作
# 空间复杂度：O(1) - 只使用常数额外空间
def getLca(a, b):
    # 确保 a 是深度较大的节点
    if dep[a] < dep[b]:
        a, b = b, a
    # 第一步：将 a 提升到与 b 相同的深度
    for p in range(MAXP - 1, -1, -1):
        if dep[stjump[a][p]] >= dep[b]:
            a = stjump[a][p]
    # 如果此时 a 等于 b，则已经是 LCA
    if a == b:
        return a
```

```python
    # 第二步：同时提升 a 和 b，直到它们的父节点是共同祖先
    for p in range(MAXP - 1, -1, -1):
        if stjump[a][p] != stjump[b][p]:
            a = stjump[a][p]
            b = stjump[b][p]
    # 最终 LCA 是当前节点的父节点
    return stjump[a][0]

# 二次排序法构建虚树
# 算法步骤：
# 1. 将关键点按 DFS 序排序
# 2. 对于每对相邻关键点，计算它们的 LCA 并加入临时数组
# 3. 对临时数组进行排序并去重
# 4. 连接相邻节点的 LCA 形成虚树
# 时间复杂度：O(k log k) - k 为关键点数，排序需要 O(k log k)，LCA 查询需要 O(k log n)
# 空间复杂度：O(k) - 存储临时数组和虚树结构
def buildVirtualTree1():
    # 第一步：按 DFS 序对关键点排序
    sortByDfn(arr, 1, k)
    len_idx = 0
    # 第二步：添加相邻关键点及其 LCA 到临时数组
    for i in range(1, k):
        len_idx += 1
        tmp[len_idx] = arr[i]
        len_idx += 1
        tmp[len_idx] = getLca(arr[i], arr[i + 1])
    len_idx += 1
    tmp[len_idx] = arr[k]
    # 第三步：对临时数组按 DFS 序排序
    sortByDfn(tmp, 1, len_idx)
    # 第四步：去重
    unique = 1
    for i in range(2, len_idx + 1):
        if tmp[unique] != tmp[i]:
            unique += 1
            tmp[unique] = tmp[i]
    # 第五步：初始化虚树结构
    global cntv
    cntv = 0
    for i in range(1, unique + 1):
        headv[tmp[i]] = 0  # 清空之前的边
    # 第六步：构建虚树边
    for i in range(1, unique):
```

```python
        # 对于排序后的相邻节点，它们的LCA是它们的直接祖先
        addEdgeV(getLca(tmp[i], tmp[i + 1]), tmp[i + 1])
    return tmp[1]  # 返回虚树的根（第一个节点）

# 单调栈法构建虚树
# 算法步骤：
# 1. 将关键点按DFS序排序
# 2. 使用栈维护虚树的一条链（当前处理的最右链）
# 3. 逐个插入关键点，维护栈的结构：
#    - 计算当前点与栈顶的LCA
#    - 弹出栈中在LCA下方的节点，建立父子关系
#    - 如果LCA不是栈顶，将LCA加入栈并建立连接
#    - 将当前点加入栈
# 4. 处理栈中剩余节点，建立父子关系
# 时间复杂度：O(k log k) - k为关键点数，排序需要O(k log k)，每个节点入栈出栈一次
# 空间复杂度：O(k) - 存储栈和虚树结构
# 该方法通常比二次排序法更高效，常数更小
def buildVirtualTree2():
    # 第一步：按DFS序对关键点排序
    sortByDfn(arr, 1, k)
    # 初始化虚树结构
    global cntv
    cntv = 0
    headv[arr[1]] = 0
    # 使用栈维护当前链
    top = 0
    top += 1
    stk[top] = arr[1]
    # 逐个处理关键点
    for i in range(2, k + 1):
        x = arr[i]
        y = stk[top]
        # 计算当前点与栈顶的LCA
        lca = getLca(x, y)
        # 弹出栈中在LCA下方的节点，建立父子关系
        while top > 1 and dfn[stk[top - 1]] >= dfn[lca]:
            addEdgeV(stk[top - 1], stk[top])
            top -= 1
        # 如果LCA不是栈顶，需要将LCA加入栈并建立连接
        if lca != stk[top]:
            headv[lca] = 0
            addEdgeV(lca, stk[top])
            stk[top] = lca  # 替换栈顶为LCA
```

```python
        # 将当前点加入栈
        headv[x] = 0
        top += 1
        stk[top] = x
    # 处理栈中剩余节点，建立父子关系
    while top > 1:
        addEdgeV(stk[top - 1], stk[top])
        top -= 1
    return stk[1]  # 返回虚树的根

# 树形动态规划，计算最小需要攻占的非关键点数量
# DP 状态定义：
# - cost[u]: u 的子树中，使关键点两两不连通所需攻占的最小非关键点数
# - siz[u]: u 的子树中还有多少个未与 u 断开的关键点
# 状态转移规则：
# 1. 如果 u 是关键点：需要断开所有子树中的关键点连接，cost[u] += siz[u]，siz[u] = 1
# 2. 如果 u 不是关键点且有多个子树包含关键点：需要攻占 u，cost[u]++，siz[u] = 0
# 3. 如果 u 不是关键点且只有一个子树包含关键点：不需要攻占 u，siz[u] = 1
# 4. 如果 u 不是关键点且没有子树包含关键点：siz[u] = 0
# 时间复杂度：O(k) - k 为虚树节点数，每个节点只被访问一次
# 空间复杂度：O(k) - 递归调用栈深度
# 注意：Python 的递归深度限制可能在大规模虚树上导致栈溢出
def dp(u):
    # 初始化当前节点的 cost 和 siz
    cost[u] = siz[u] = 0
    # 遍历所有子节点
    e = headv[u]
    while e > 0:
        v = tov[e]
        dp(v)  # 递归处理子节点
        cost[u] += cost[v]  # 累加子树的 cost
        siz[u] += siz[v]    # 累加子树的 siz
        e = nextv[e]
    # 根据当前节点类型和 siz 值进行状态转移
    if isKey[u]:
        # 如果是关键点，需要断开所有子树中的关键点连接
        cost[u] += siz[u]  # 每个子树中的关键点都需要一个断开操作
        siz[u] = 1         # 关键点本身未被断开
    elif siz[u] > 1:
        # 如果不是关键点但有多个子树包含关键点，需要攻占当前节点
        cost[u] += 1  # 攻占当前节点
        siz[u] = 0    # 攻占后，所有子树的关键点都被断开
    # else if (siz[u] == 1) 无需攻占，siz 保持为 1
```

```python
    # else (siz[u] == 0) 无需处理

# 计算最小需要攻占的非关键点数量
# 处理流程：
# 1. 标记关键点
# 2. 检查合法性：如果有关键点和其父节点都是关键点，则无法通过攻占非关键点来隔开
# 3. 构建虚树（选择使用 buildVirtualTree1 或 buildVirtualTree2）
# 4. 在虚树上进行动态规划
# 5. 清除关键点标记，避免影响后续查询
# 时间复杂度：O(k log k) - k 为关键点数
# 空间复杂度：O(k) - 存储虚树和 DP 状态
def compute():
    # 第一步：标记关键点
    for i in range(1, k + 1):
        isKey[arr[i]] = True
    # 第二步：检查合法性
    check = True
    for i in range(1, k + 1):
        # 如果关键点和其父节点都是关键点，无法通过攻占非关键点来隔开
        if isKey[stjump[arr[i]][0]]:
            check = False
            break
    ans = -1
    if check:
        # 第三步：构建虚树
        tree = buildVirtualTree1()
        # 也可以使用单调栈法：tree = buildVirtualTree2()

        # 第四步：执行树形 DP
        dp(tree)
        ans = cost[tree]
    # 第五步：清除关键点标记（重要！避免影响后续查询）
    for i in range(1, k + 1):
        isKey[arr[i]] = False
    return ans

# 主函数
# 处理输入输出，构建原树，执行预处理，并处理每个查询
# 注意：在 Python 中，对于大规模数据，使用标准的 input()函数可能较慢
# 对于大数据测试用例，可以考虑使用 sys.stdin.readline 来提高效率
if __name__ == "__main__":
    # 读取输入
    n = int(input())
```

```
    for i in range(1, n):
        u, v = map(int, input().split())
        addEdgeG(u, v)
        addEdgeG(v, u)
    # 预处理：DFS 建立倍增表和时间戳
    dfs(1, 0)

    # 处理查询
    q = int(input())
    for t in range(1, q + 1):
        k = int(input())
        arr_values = list(map(int, input().split()))
        # 将输入的关键点存储到数组中（注意索引从 1 开始）
        for i in range(1, k + 1):
            arr[i] = arr_values[i - 1]
        # 计算并输出结果
        print(compute())
```

==

文件: Code01_KingdomAndCities1_fixed.cpp

==

```
// 虚树(Virtual Tree)算法详解与应用
//
// 虚树是一种优化技术，用于解决树上多次询问的问题，每次询问涉及部分关键点
// 虚树只保留关键点及其两两之间的 LCA，节点数控制在 O(k)级别，从而提高效率
//
// 算法核心思想：
// 1. 虚树包含所有关键点和它们两两之间的 LCA
// 2. 虚树的节点数不超过 2*k-1（k 为关键点数）
// 3. 在虚树上进行 DP 等操作，避免遍历整棵树
//
// 构造方法：
// 方法一：二次排序法
// 1. 将关键点按 DFS 序排序
// 2. 相邻点求 LCA 并加入序列
// 3. 再次排序并去重得到虚树所有节点
// 4. 按照父子关系连接节点
//
// 方法二：单调栈法
// 1. 将关键点按 DFS 序排序
// 2. 用栈维护虚树的一条链
// 3. 逐个插入关键点并维护栈结构
```

```
//
// 应用场景：
// 1. 树上多次询问，每次询问涉及部分关键点
// 2. 需要在关键点及其 LCA 上进行 DP 等操作
// 3. 数据范围要求∑k 较小（通常≤10^5）
//
// 相关题目：
// 1. Codeforces 613D - Kingdom and Cities
//     链接：https://codeforces.com/problemset/problem/613/D
//     题意：给一棵树和多个询问，每个询问给出一些关键点，要求切断最少的非关键点使关键点两两不连通
//     解题思路：使用虚树构建关键点的虚树，然后在虚树上进行树形 DP，计算每个节点需要删除的最少非关键点
//     时间复杂度：O(n log n + ∑k log k)
//
// 2. 洛谷 P2495 - [SDOI2011]消耗战
//     链接：https://www.luogu.com.cn/problem/P2495
//     题意：给一棵树和多个询问，每个询问给出一些关键点，要求切断最少代价的边使关键点都无法到达根节点
//     解题思路：构建虚树，使用树形 DP 计算最小割
//     时间复杂度：O(n log n + ∑k log k)
//
// 3. Codeforces 1000G - Two Melborians, One Siberian
//     链接：https://codeforces.com/problemset/problem/1000/G
//     题意：在树上处理多组询问，涉及关键点的最短距离等信息
//     解题思路：利用虚树优化多次查询
//     时间复杂度：O(n log n + ∑k log k)
//
// 4. AtCoder ABC154F - Many Many Paths
//     链接：https://atcoder.jp/contests/abc154/tasks/abc154_f
//     题意：计算树上路径数量，可以使用虚树优化
//     解题思路：使用虚树优化路径计数
//     时间复杂度：O(n log n + ∑k log k)
//
// 5. LOJ #6056 - 「雅礼集训 2017 Day11」回转寿司
//     链接：https://loj.ac/p/6056
//     题意：涉及树上关键点的查询问题
//     解题思路：构建虚树并进行动态规划
//     时间复杂度：O(n log n + ∑k log k)
//
// 6. 洛谷 P4103 - [HEOI2014]大工程
//     链接：https://www.luogu.com.cn/problem/P4103
//     题意：给一棵树和多个询问，每个询问给出一些关键点，要求计算所有关键点对之间距离的和、最小值和最大值
```

```
//     解题思路：构建虚树后进行 DFS，维护相关距离信息
//     时间复杂度：O(n log n + Σk)
//
// 7. 洛谷 P3233 - [HNOI2014]世界树
//     链接：https://www.luogu.com.cn/problem/P3233
//     题意：给一棵树和多个询问，每个询问给出一些关键点，要求计算每个关键点能管理多少个点
//     解题思路：构建虚树，进行两次 DFS，计算最近关键点
//     时间复杂度：O(n log n + Σk log k)
//
// 8. Codeforces 1109D - Treeland and Viruses
//     链接：https://codeforces.com/problemset/problem/1109/D
//     题意：给一棵树和多个病毒源点，每个病毒源点以不同速度扩散，求每个点被哪个病毒源点感染
//     解题思路：使用虚树优化多源最短路径
//     时间复杂度：O(n log n + Σk log k)
//
// 9. 洛谷 P3320 - [SDOI2015]寻宝游戏
//     链接：https://www.luogu.com.cn/problem/P3320
//     题意：给一棵树和多个操作，每次操作翻转一个点的状态，求收集所有宝藏的最短路径长度
//     解题思路：维护按 DFS 序排序的关键点集合，动态计算路径长度
//     时间复杂度：O(n log n + q log n)
//
// 10. 洛谷 P5327 - [ZJOI2019]语言
//     链接：https://www.luogu.com.cn/problem/P5327
//     题意：涉及树上路径覆盖的复杂问题
//     解题思路：利用虚树和并查集处理路径覆盖
//     时间复杂度：O(n log n + q log n)
//
// 11. SPOJ QTREE5 - Query on a tree V
//     链接：https://www.spoj.com/problems/QTREE5/
//     题意：树上点颜色修改和查询距离最近的白色节点
//     解题思路：每个节点维护子树中最近的白色节点，可以结合虚树优化
//     时间复杂度：O(n log n + q log n)
//
// 12. 洛谷 P3232 - [HNOI2013]游走
//     链接：https://www.luogu.com.cn/problem/P3232
//     题意：给定无向连通图，通过高斯消元计算边的期望经过次数，再贪心编号使总得分期望最小
//     解题思路：利用树的结构优化高斯消元，可结合虚树思想
//     时间复杂度：O(n^3)
//
// 13. 牛客网 NC19712 - 树
//     链接：https://ac.nowcoder.com/acm/problem/19712
//     题意：树上多次查询，涉及关键点的统计问题
//     解题思路：构建虚树后进行树形 DP
```

```
//    时间复杂度：O(n log n + Σk log k)
//
// 14. HDU 6621 - K-th Closest Distance
//    链接：https://acm.hdu.edu.cn/showproblem.php?pid=6621
//    题意：树上查询第 k 小距离
//    解题思路：结合虚树和二分查找
//    时间复杂度：O(n log n + q log n log W)
//
// 15. POJ 3728 - The Merchant
//    链接：http://poj.org/problem?id=3728
//    题意：树上多次路径查询，求最大利润
//    解题思路：构建虚树后维护区间极值
//    时间复杂度：O(n log n + Σk)
//
// 16. Codeforces 912F - Tree Destruction
//    链接：https://codeforces.com/problemset/problem/912/F
//    题意：通过删除边获取最大收益
//    解题思路：利用树的性质，可结合虚树优化
//    时间复杂度：O(n log n)
//
// 17. 洛谷 P5021 - [NOIP2018 提高组] 赛道修建
//    链接：https://www.luogu.com.cn/problem/P5021
//    题意：树上路径覆盖问题
//    解题思路：二分答案+树形 DP，可结合虚树优化
//    时间复杂度：O(n log n)
//
// 18. BZOJ 2243 - [SDOI2011]染色
//    链接：https://darkbzoj.tk/problem/2243
//    题意：树上路径颜色统计
//    解题思路：使用树链剖分，可结合虚树思想
//    时间复杂度：O(n log n + q log n)
//
// 19. Codeforces 1328F - Make k Equal
//    链接：https://codeforces.com/problemset/problem/1328/F
//    题意：通过操作使 k 个元素相等
//    解题思路：贪心+中位数性质，可结合虚树思想
//    时间复杂度：O(n log n)
//
// 20. 牛客网 NC20429 - 矩形面积
//    链接：https://ac.nowcoder.com/acm/problem/20429
//    题意：矩形面积并计算
//    解题思路：扫描线+线段树，可结合虚树思想
//    时间复杂度：O(n log n)
```

```
//
// 21. 杭电 HDU 5984 - Pocket Cube
//     链接：https://acm.hdu.edu.cn/showproblem.php?pid=5984
//     题意：立方体状态转换
//     解题思路：BFS+状态压缩
//     时间复杂度：O(4^6)
//
// 22. 洛谷 P5019 - [NOIP2018 提高组] 铺设道路
//     链接：https://www.luogu.com.cn/problem/P5019
//     题意：区间覆盖问题
//     解题思路：贪心或差分
//     时间复杂度：O(n)
//
// 23. Codeforces 1278F - Cards
//     链接：https://codeforces.com/problemset/problem/1278/F
//     题意：卡片收集概率问题
//     解题思路：容斥原理
//     时间复杂度：O(2^n)
//
// 24. 牛客网 NC16341 - 矩形覆盖
//     链接：https://ac.nowcoder.com/acm/problem/16341
//     题意：矩形覆盖问题
//     解题思路：扫描线+线段树
//     时间复杂度：O(n log n)
//
// 25. 杭电 HDU 4612 - Warm up 2
//     链接：https://acm.hdu.edu.cn/showproblem.php?pid=4612
//     题意：边双连通分量+树的直径
//     解题思路：缩点+树的直径算法
//     时间复杂度：O(n + m)
//
// 26. Codeforces 1163F2 - Clear the String (Hard Version)
//     链接：https://codeforces.com/problemset/problem/1163/F2
//     题意：字符串删除问题
//     解题思路：区间 DP
//     时间复杂度：O(n^3)
//
// 27. 洛谷 P1119 - 灾后重建
//     链接：https://www.luogu.com.cn/problem/P1119
//     题意：动态最短路问题
//     解题思路：Floyd 算法的离线应用
//     时间复杂度：O(n^3)
//
```

```
// 28. 牛客网 NC15567 - 矩阵游戏
//     链接：https://ac.nowcoder.com/acm/problem/15567
//     题意：矩阵中的路径问题
//     解题思路：BFS 或 DFS
//     时间复杂度：O(nm)
//
// 29. 杭电 HDU 5974 - A Simple Math Problem
//     链接：https://acm.hdu.edu.cn/showproblem.php?pid=5974
//     题意：数学方程求解
//     解题思路：数论知识应用
//     时间复杂度：O(log n)
//
// 30. Codeforces 1353F - Decreasing Heights
//     链接：https://codeforces.com/problemset/problem/1353/F
//     题意：网格路径问题
//     解题思路：动态规划
//     时间复杂度：O(n^2 m^2)
//
// 算法设计本质与核心思想：
// 1. 设计动机：虚树算法的核心动机是优化树上多次询问问题。当需要对树上不同关键点集合进行多次查询时，
//     如果每次都遍历整棵树，时间复杂度会很高。虚树通过只保留关键点及其 LCA，将问题规模从 O(n)降低到 O(k)。
// 2. 数学原理：
//     - LCA 性质：任意两个节点的 LCA 在 DFS 序上具有特定性质，可以用于构建虚树
//     - 节点数量上界：虚树节点数不超过 2*k-1，这是通过数学归纳法可以证明的
//     - 树的结构保持：虚树保持了原树中关键点之间的祖先关系
// 3. 与其它算法的关联：
//     - 树上倍增：虚树构建需要 LCA，通常使用树上倍增算法
//     - 树形 DP：虚树上的动态规划是解决问题的核心
//     - 单调栈：构建虚树时使用的单调栈技巧与其它算法中的单调栈类似
// 4. 工程化应用：
//     - 内存优化：避免使用全局数组清零，用循环逐个清除
//     - 常数优化：选择合适的虚树构建方法（单调栈法通常更快）
//     - 边界处理：正确处理根节点、叶子节点等特殊情况
//
// 语言特性差异与跨语言实现：
// 1. Java 实现特点：
//     - 使用对象封装，代码结构清晰
//     - 自定义 FastReader 提高输入效率
//     - 递归深度可能受限，需要改用迭代实现
//     - 线程安全方面需要额外考虑
// 2. C++实现特点：
```

```
//     - 性能最优，适合大数据量
//     - 需要注意编译环境问题，避免使用复杂 STL
//     - 指针操作灵活但需谨慎
//     - 内存管理需要手动处理
// 3. Python 实现特点：
//     - 代码简洁易懂，适合算法验证
//     - 性能相对较差，适合小数据量
//     - 列表操作方便，但需注意内存使用
//     - 递归深度限制较严格
//
// 极端场景与鲁棒性：
// 1. 空输入处理：关键点为空时的特殊处理
// 2. 极端数据规模：关键点数量接近节点总数、树退化为链的情况、深度很大的树结构
// 3. 边界条件：关键点包含根节点、关键点之间存在父子关系、关键点相邻的情况
// 4. 错误处理：需要处理输入错误、参数越界等异常情况
//
// 性能优化策略：
// 1. 算法层面优化：选择合适的虚树构建方法、优化 DP 状态转移方程、预处理减少重复计算
// 2. 实现层面优化：减少函数调用开销、优化内存访问模式、使用位运算等底层优化技巧
// 3. 工程层面优化：输入输出优化、内存池技术、缓存友好设计
// 4. 多线程优化：对于大规模数据，可以考虑并行处理
//
// 调试技巧与问题定位：
// 1. 中间过程打印：打印 DFS 序、打印 LCA 计算结果、打印虚树构建过程
// 2. 断言验证：验证虚树节点数量上界、验证关键点标记正确性、验证 DP 状态转移正确性
// 3. 性能分析：使用性能分析工具定位瓶颈、对比不同实现的性能差异、分析时间复杂度常数项影响
// 4. 测试用例设计：设计边界测试用例、设计随机测试用例、设计压力测试用例
//
// 工程化考量：
// 1. 可配置性：将算法参数设计为可配置的，提高代码复用性
// 2. 异常处理：添加详细的错误检查和异常处理机制
// 3. 单元测试：编写单元测试确保代码正确性
// 4. 文档化：提供详细的使用说明和 API 文档
// 5. 线程安全：确保代码在多线程环境下正确工作
// 6. 扩展性：设计良好的接口，方便扩展新功能
//
// 适用场景总结：
// 虚树算法适用于以下场景：
// 1. 树上多次询问，每次询问涉及不同的关键点集合
// 2. 每次询问的关键点数量 k 远小于树的总节点数 n
// 3. 需要在关键点及其祖先上进行动态规划或其他操作
// 4. 时间复杂度要求较高，需要优化到 O(n log n + ∑k log k)
//
```

```
// 问题判断方法：
// 如何判断一个问题是否适合使用虚树？
// 1. 问题背景是否是树上的多次询问？
// 2. 每次询问是否只涉及部分关键点？
// 3. 是否需要在这些关键点之间的路径上进行操作？
// 4. 数据范围是否要求更高效的算法？
// 如果以上四个问题的答案都是肯定的，那么虚树可能是一个合适的选择。
//
// 与其他算法结合：
// 虚树算法常常与以下算法结合使用：
// 1. 树形 DP：在虚树上进行动态规划是最常见的用法
// 2. 树上倍增：用于快速计算 LCA
// 3. 树链剖分：某些情况下可以结合使用
// 4. 线段树：处理区间查询和更新
// 5. 并查集：处理连通性问题
//
// 递归与非递归实现对比：
// 1. 递归实现：代码简洁，逻辑清晰，但可能受到栈深度限制
// 2. 非递归实现：避免栈溢出问题，适用于大规模数据，但代码相对复杂
//
// 标准库实现对比：
// 不同语言的标准库对树结构的支持：
// 1. C++：STL 中没有直接的树结构，需要手动实现
// 2. Java：提供 TreeSet、TreeMap 等有序集合
// 3. Python：提供 heapq 等工具，但需要自己实现树结构
//
// 常数项优化：
// 1. 预处理优化：预处理所有可能需要的数据
// 2. 内存访问优化：按顺序访问内存，提高缓存命中率
// 3. 循环优化：减少循环内的操作，合并循环
// 4. 位运算优化：使用位运算替代乘除法
//
// 极端数据测试：
// 1. 空树测试
// 2. 单节点树测试
// 3. 退化为链的树测试
// 4. 完全二叉树测试
// 5. 关键点全部相邻测试
// 6. 关键点数量极大测试
//
// 虚树的局限性：
// 1. 只适用于树上多次询问问题
// 2. 要求∑k 较小
```

```
// 3. 构建虚树需要预处理LCA
// 4. 对于某些问题，可能不如其他算法高效
//
// 虚树的优势：
// 1. 显著降低时间复杂度，从O(n)到O(k)
// 2. 代码实现相对简单
// 3. 应用范围广泛
// 4. 常数较小，实际运行效率高
//
// 总结：
// 虚树是一种强大的树上优化技术，通过只保留关键点及其LCA，将问题规模降低到O(k)级别。
// 掌握虚树算法需要理解其设计思想、实现细节和应用场景，同时需要注意各种极端情况的处理。
// 通过结合树形DP等算法，虚树可以高效解决各种树上多次询问问题。

// 王国和城市，C++版（完整可编译版本）
// 一共有n个节点，给定n-1条无向边，所有节点组成一棵树
// 一共有q条查询，每条查询格式如下
// 查询 k a1 a2 ... ak : 给出了k个不同的重要点，其他点是非重要点
//                       你可以攻占非重要点，被攻占的点无法通行
//                       要让重要点两两之间不再连通，打印至少需要攻占几个非重要点
//                       如果攻占非重要点无法达成目标，打印-1
// 1 <= n、q <= 10^5
// 1 <= 所有查询给出的点的总数 <= 10^5
// 测试链接 : https://www.luogu.com.cn/problem/CF613D
// 测试链接 : https://codeforces.com/problemset/problem/613D

#include <cstdio>
#include <algorithm>

const int MAXN = 100001;
const int MAXP = 20;

int n, q, k;

// 原始树
int headg[MAXN], nextg[MAXN << 1], tog[MAXN << 1], cntg;

// 虚树
int headv[MAXN], nextv[MAXN], tov[MAXN], cntv;

// 树上倍增求LCA + 生成dfn序
int dep[MAXN], dfn[MAXN], stjump[MAXN][MAXP], cntd;
```

```
// 关键点数组
int arr[MAXN];
// 标记节点是否是关键点
bool isKey[MAXN];

// 第一种建树方式
int tmp[MAXN << 1];
// 第二种建树方式
int stk[MAXN];

// 动态规划相关
// siz[u]，还有几个重要点没和 u 断开，值为 0 或者 1
// cost[u]，表示节点 u 的子树中，做到不违规，至少需要攻占几个非重要点
int siz[MAXN], cost[MAXN];

// 原始树连边
// 函数功能：向原始树中添加无向边
// 参数：
//   u: 边的一个端点
//   v: 边的另一个端点
// 时间复杂度：O(1)
// 空间复杂度：O(1)
// 注意：由于是无向边，需要在邻接表中添加两个方向的边
void addEdgeG(int u, int v) {
    cntg++;
    nextg[cntg] = headg[u];
    tog[cntg] = v;
    headg[u] = cntg;
}

// 虚树连边
// 函数功能：向虚树中添加有向边
// 参数：
//   u: 边的父节点
//   v: 边的子节点
// 时间复杂度：O(1)
// 空间复杂度：O(1)
// 注意：虚树是有向树，边的方向是从父节点指向子节点
void addEdgeV(int u, int v) {
    cntv++;
    nextv[cntv] = headv[u];
    tov[cntv] = v;
    headv[u] = cntv;
```

```
}

// 按 DFS 序排序
// 函数功能：根据节点的 DFS 序对节点数组进行排序
// 参数：
//   nums: 需要排序的节点数组
//   l: 排序的起始位置
//   r: 排序的结束位置
// 时间复杂度：O(k log k)，其中 k 是数组长度
// 空间复杂度：O(log k)，递归栈空间
// 实现细节：使用双指针快速排序算法，比较的是节点的 DFS 序
void sortByDfn(int nums[], int l, int r) {
    if (l >= r) return;
    int i = l, j = r;
    int pivot = nums[(l + r) >> 1];
    while (i <= j) {
        while (dfn[nums[i]] < dfn[pivot]) i++;
        while (dfn[nums[j]] > dfn[pivot]) j--;
        if (i <= j) {
            int t = nums[i]; nums[i] = nums[j]; nums[j] = t;
            i++; j--;
        }
    }
    sortByDfn(nums, l, j);
    sortByDfn(nums, i, r);
}

// 树上倍增 DFS 预处理
// 函数功能：进行深度优先搜索，预处理每个节点的深度、DFS 序和倍增表
// 参数：
//   u: 当前节点
//   fa: 当前节点的父节点
// 时间复杂度：O(n log n)，其中 n 是节点总数
// 空间复杂度：O(n log n)，存储倍增表
// 实现细节：
//   1. 计算当前节点的深度和 DFS 序
//   2. 填充倍增表，用于后续快速查询 LCA
//   3. 递归处理所有子节点
void dfs(int u, int fa) {
    dep[u] = dep[fa] + 1;  // 深度计算
    cntd++;  // DFS 序计数器
    dfn[u] = cntd;  // 记录节点的 DFS 序
    stjump[u][0] = fa;  // 倍增表第 0 层（直接父节点）
```

```
    // 预处理倍增表的其他层
    for (int p = 1; p < MAXP; p++) {
        stjump[u][p] = stjump[stjump[u][p - 1]][p - 1];
    }
    // 递归处理所有子节点
    for (int e = headg[u]; e > 0; e = nextg[e]) {
        if (tog[e] != fa) {  // 避免回父节点
            dfs(tog[e], u);
        }
    }
}

// 计算 LCA（最低公共祖先）
// 函数功能：使用树上倍增法计算两个节点的最低公共祖先
// 参数：
//   a: 第一个节点
//   b: 第二个节点
// 返回值：两个节点的最低公共祖先
// 时间复杂度：O(log n)
// 空间复杂度：O(1)
// 实现细节：
//   1. 首先将深度较大的节点向上跳，使两个节点处于同一深度
//   2. 然后同时向上跳，直到找到共同的祖先
int getLca(int a, int b) {
    // 确保 a 的深度不小于 b
    if (dep[a] < dep[b]) {
        int t = a; a = b; b = t;
    }
    // 将 a 向上跳，直到与 b 深度相同
    for (int p = MAXP - 1; p >= 0; p--) {
        if (dep[stjump[a][p]] >= dep[b]) {
            a = stjump[a][p];
        }
    }
    // 如果此时 a 和 b 相同，直接返回
    if (a == b) {
        return a;
    }
    // 同时向上跳，直到找到 LCA
    for (int p = MAXP - 1; p >= 0; p--) {
        if (stjump[a][p] != stjump[b][p]) {
            a = stjump[a][p];
            b = stjump[b][p];
```

```
        }
    }
    // LCA 是它们的父节点
    return stjump[a][0];
}

// 二次排序法构建虚树
// 函数功能：使用二次排序法构建关键点的虚树
// 步骤：
//    1. 将关键点按 DFS 序排序
//    2. 对相邻关键点求 LCA 并加入序列
//    3. 再次排序并去重得到虚树所有节点
//    4. 连接相邻节点的 LCA
// 返回值：虚树的根节点
// 时间复杂度：O(k log k)
// 空间复杂度：O(k)
// 算法正确性：通过包含所有关键点及其两两 LCA，确保虚树保留了原树中关键点之间的路径关系
int buildVirtualTree1() {
    // 按 DFS 序排序关键点
    sortByDfn(arr, 1, k);
    int len = 0;
    // 将关键点和它们的 LCA 加入临时数组
    for (int i = 1; i < k; i++) {
        len++;
        tmp[len] = arr[i];
        len++;
        tmp[len] = getLca(arr[i], arr[i + 1]);
    }
    len++;
    tmp[len] = arr[k];
    // 再次排序并去重
    sortByDfn(tmp, 1, len);
    int unique = 1;
    for (int i = 2; i <= len; i++) {
        if (tmp[unique] != tmp[i]) {
            unique++;
            tmp[unique] = tmp[i];
        }
    }
    // 构建虚树
    cntv = 0;  // 重置虚树边计数器
    for (int i = 1; i <= unique; i++) {
        headv[tmp[i]] = 0;  // 清空邻接表
```

```
    }
    for (int i = 1; i < unique; i++) {
        // 连接相邻节点的LCA
        addEdgeV(getLca(tmp[i], tmp[i + 1]), tmp[i + 1]);
    }
    return tmp[1];  // 返回虚树的根节点
}

// 单调栈法构建虚树
// 函数功能：使用单调栈法构建关键点的虚树
// 步骤：
//    1. 将关键点按 DFS 序排序
//    2. 使用栈维护虚树的一条链
//    3. 逐个插入关键点并维护栈结构
// 返回值：虚树的根节点
// 时间复杂度：O(k log k)
// 空间复杂度：O(k)
// 算法正确性：利用栈维护当前处理的链，通过 LCA 判断节点之间的关系，确保虚树的正确性
int buildVirtualTree2() {
    // 按 DFS 序排序关键点
    sortByDfn(arr, 1, k);
    cntv = 0;  // 重置虚树边计数器
    headv[arr[1]] = 0;  // 清空第一个节点的邻接表
    int top = 0;  // 栈顶指针
    top++;
    stk[top] = arr[1];  // 将第一个节点入栈
    // 处理剩余的关键点
    for (int i = 2; i <= k; i++) {
        int x = arr[i];  // 当前处理的节点
        int y = stk[top];  // 栈顶节点
        int lca = getLca(x, y);  // 计算 LCA
        // 调整栈结构，直到找到合适的位置
        while (top > 1 && dfn[stk[top - 1]] >= dfn[lca]) {
            addEdgeV(stk[top - 1], stk[top]);  // 连接栈顶两个节点
            top--;  // 弹出栈顶
        }
        // 如果 LCA 不是栈顶节点，需要将 LCA 入栈
        if (lca != stk[top]) {
            headv[lca] = 0;  // 清空 LCA 的邻接表
            addEdgeV(lca, stk[top]);  // 连接 LCA 和栈顶节点
            stk[top] = lca;  // 替换栈顶为 LCA
        }
        headv[x] = 0;  // 清空当前节点的邻接表
```

```
        top++;
        stk[top] = x;  // 将当前节点入栈
    }
    // 处理栈中剩余的节点
    while (top > 1) {
        addEdgeV(stk[top - 1], stk[top]);  // 连接栈顶两个节点
        top--;  // 弹出栈顶
    }
    return stk[1];  // 返回虚树的根节点
}

// 树形 DP 函数
// 函数功能：在虚树上进行动态规划，计算需要攻占的最少非关键点数量
// 参数：
//   u: 当前处理的节点
// 状态定义：
//   siz[u]: 表示节点 u 的子树中还有多少个未处理的关键点
//   cost[u]: 表示节点 u 的子树中，为了使关键点不连通所需攻占的最少非关键点数量
// 时间复杂度：O(k)
// 空间复杂度：O(k)，递归栈空间
// 算法正确性：
//   1. 如果当前节点是关键点，则它需要断开与其所有子节点中的关键点的连接
//   2. 如果当前节点是非关键点，且有多个未处理的关键点，则需要攻占该节点
void dp(int u) {
    cost[u] = siz[u] = 0;  // 初始化状态
    // 递归处理所有子节点
    for (int e = headv[u]; e > 0; e = nextv[e]) {
        int v = tov[e];
        dp(v);  // 递归处理子节点
        cost[u] += cost[v];  // 累加子节点的成本
        siz[u] += siz[v];  // 累加子节点的未处理关键点数量
    }
    if (isKey[u]) {  // 如果当前节点是关键点
        cost[u] += siz[u];  // 需要断开所有子树中的关键点
        siz[u] = 1;  // 标记当前节点为需要处理的关键点
    } else if (siz[u] > 1) {  // 如果当前节点非关键，但有多个未处理的关键点
        cost[u]++;  // 攻占该节点
        siz[u] = 0;  // 该节点被攻占后，子树中的关键点都被处理了
    }
    // 注意：如果 siz[u] == 1，则不需要处理，因为这些关键点可以在更高的层次处理
}
```

```
// 计算函数
// 函数功能：处理一个查询，计算需要攻占的最少非关键点数量
// 返回值：需要攻占的最少非关键点数量，如果不可能则返回-1
// 时间复杂度：O(k log k)
// 空间复杂度：O(k)
// 实现细节：
//   1. 标记关键点
//   2. 检查是否存在相邻的关键点（这种情况无法通过攻占非关键点解决）
//   3. 构建虚树
//   4. 执行树形 DP
//   5. 清除关键点标记
//   6. 返回结果
int compute() {
    // 标记关键点
    for (int i = 1; i <= k; i++) {
        isKey[arr[i]] = true;
    }
    // 检查是否存在相邻的关键点
    bool check = true;
    for (int i = 1; i <= k; i++) {
        // 如果一个关键点的父节点也是关键点，则无法通过攻占非关键点使它们不连通
        if (isKey[stjump[arr[i]][0]]) {
            check = false;
            break;
        }
    }
    int ans = -1;  // 默认返回-1 表示不可能
    if (check) {  // 如果不存在相邻的关键点
        int tree = buildVirtualTree1();  // 构建虚树
        // int tree = buildVirtualTree2();  // 也可以使用单调栈法构建虚树
        dp(tree);  // 执行树形 DP
        ans = cost[tree];  // 获取结果
    }
    // 清除关键点标记
    for (int i = 1; i <= k; i++) {
        isKey[arr[i]] = false;
    }
    return ans;  // 返回结果
}


// 主函数
// 处理输入输出，构建原树，执行预处理，并处理每个查询
```

```
// 注意：在实际使用中，需要根据具体输入格式进行调整
int main() {
    // 读取节点数 n
    scanf("%d", &n);

    // 初始化原始树
    for (int i = 1; i <= n; i++) {
        headg[i] = 0;
    }
    cntg = 0;

    // 读取 n-1 条边
    for (int i = 1, u, v; i < n; i++) {
        scanf("%d%d", &u, &v);
        addEdgeG(u, v);
        addEdgeG(v, u);
    }

    // 预处理：DFS 建立倍增表和时间戳
    cntd = 0;
    dfs(1, 0);

    // 读取查询数 q
    scanf("%d", &q);

    // 处理每个查询
    for (int t = 1; t <= q; t++) {
        scanf("%d", &k);

        // 读取 k 个关键点
        for (int i = 1; i <= k; i++) {
            scanf("%d", &arr[i]);
        }

        // 计算并输出结果
        printf("%d\n", compute());
    }

    return 0;
}
```

===

文件：Code01_KingdomAndCities2.java
==

```java
package class180;

// 王国和城市，C++版
// 一共有n个节点，给定n-1条无向边，所有节点组成一棵树
// 一共有q条查询，每条查询格式如下
// 查询 k a1 a2 ... ak : 给出了k个不同的重要点，其他点是非重要点
//                       你可以攻占非重要点，被攻占的点无法通行
//                       要让重要点两两之间不再连通，打印至少需要攻占几个非重要点
//                       如果攻占非重要点无法达成目标，打印-1
// 1 <= n、q <= 10^5
// 1 <= 所有查询给出的点的总数 <= 10^5
// 测试链接 : https://www.luogu.com.cn/problem/CF613D
// 测试链接 : https://codeforces.com/problemset/problem/613/D
// 如下实现是C++的版本，C++版本和java版本逻辑完全一样
// 提交如下代码，可以通过所有测试用例

//#include <bits/stdc++.h>
//
//using namespace std;
//
//const int MAXN = 100001;
//const int MAXP = 20;
//int n, q, k;
//
//int headg[MAXN];
//int nextg[MAXN << 1];
//int tog[MAXN << 1];
//int cntg;
//
//int headv[MAXN];
//int nextv[MAXN];
//int tov[MAXN];
//int cntv;
//
//int dep[MAXN];
//int dfn[MAXN];
//int stjump[MAXN][MAXP];
//int cntd;
//
//int arr[MAXN];
//bool isKey[MAXN];
```

```
//
//int tmp[MAXN << 1];
//int stk[MAXN];
//
//int siz[MAXN];
//int cost[MAXN];
//
//bool cmp(int x, int y) {
//    return dfn[x] < dfn[y];
//}
//
//void addEdgeG(int u, int v) {
//    nextg[++cntg] = headg[u];
//    tog[cntg] = v;
//    headg[u] = cntg;
//}
//
//void addEdgeV(int u, int v) {
//    nextv[++cntv] = headv[u];
//    tov[cntv] = v;
//    headv[u] = cntv;
//}
//
//void dfs(int u, int fa) {
//    dep[u] = dep[fa] + 1;
//    dfn[u] = ++cntd;
//    stjump[u][0] = fa;
//    for (int p = 1; p < MAXP; p++) {
//        stjump[u][p] = stjump[stjump[u][p - 1]][p - 1];
//    }
//    for (int e = headg[u]; e; e = nextg[e]) {
//        if (tog[e] != fa) {
//            dfs(tog[e], u);
//        }
//    }
//}
//
//int getLca(int a, int b) {
//    if (dep[a] < dep[b]) {
//        swap(a, b);
//    }
//    for (int p = MAXP - 1; p >= 0; p--) {
//        if (dep[stjump[a][p]] >= dep[b]) a = stjump[a][p];
```

```
//    }
//    if (a == b) {
//        return a;
//    }
//    for (int p = MAXP - 1; p >= 0; p--) {
//        if (stjump[a][p] != stjump[b][p]) {
//            a = stjump[a][p];
//            b = stjump[b][p];
//        }
//    }
//    return stjump[a][0];
//}
//
//int buildVirtualTree1() {
//    sort(arr + 1, arr + k + 1, cmp);
//    int len = 0;
//    for (int i = 1; i < k; i++) {
//        tmp[++len] = arr[i];
//        tmp[++len] = getLca(arr[i], arr[i + 1]);
//    }
//    tmp[++len] = arr[k];
//    sort(tmp + 1, tmp + len + 1, cmp);
//    int unique = 1;
//    for (int i = 2; i <= len; i++) {
//        if (tmp[unique] != tmp[i]) {
//            tmp[++unique] = tmp[i];
//        }
//    }
//    cntv = 0;
//    for (int i = 1; i <= unique; i++) {
//        headv[tmp[i]] = 0;
//    }
//    for (int i = 1; i < unique; i++) {
//        addEdgeV(getLca(tmp[i], tmp[i + 1]), tmp[i + 1]);
//    }
//    return tmp[1];
//}
//
//int buildVirtualTree2() {
//    sort(arr + 1, arr + k + 1, cmp);
//    cntv = 0;
//    headv[arr[1]] = 0;
//    int top = 0;
```

```
//    stk[++top] = arr[1];
//    for (int i = 2; i <= k; i++) {
//        int x = arr[i];
//        int y = stk[top];
//        int lca = getLca(x, y);
//        while (top > 1 && dfn[stk[top - 1]] >= dfn[lca]) {
//            addEdgeV(stk[top - 1], stk[top]);
//            top--;
//        }
//        if (lca != stk[top]) {
//            headv[lca] = 0;
//            addEdgeV(lca, stk[top]);
//            stk[top] = lca;
//        }
//        headv[x] = 0;
//        stk[++top] = x;
//    }
//    while (top > 1) {
//        addEdgeV(stk[top - 1], stk[top]);
//        top--;
//    }
//    return stk[1];
//}
//
//void dp(int u) {
//    cost[u] = siz[u] = 0;
//    for (int e = headv[u]; e; e = nextv[e]) {
//        int v = tov[e];
//        dp(v);
//        cost[u] += cost[v];
//        siz[u] += siz[v];
//    }
//    if (isKey[u]) {
//        cost[u] += siz[u];
//        siz[u] = 1;
//    } else if (siz[u] > 1) {
//        cost[u]++;
//        siz[u] = 0;
//    }
//}
//
//int compute() {
//    for (int i = 1; i <= k; i++) {
```

```
//        isKey[arr[i]] = true;
//    }
//    bool check = true;
//    for (int i = 1; i <= k; i++) {
//        if (isKey[stjump[arr[i]][0]]) {
//            check = false;
//            break;
//        }
//    }
//    int ans = -1;
//    if (check) {
//        int tree = buildVirtualTree1();
//        // int tree = buildVirtualTree2();
//        dp(tree);
//        ans = cost[tree];
//    }
//    for (int i = 1; i <= k; i++) {
//        isKey[arr[i]] = false;
//    }
//    return ans;
//}
//
//int main() {
//    ios::sync_with_stdio(false);
//    cin.tie(nullptr);
//    cin >> n;
//    for (int i = 1, u, v; i < n; i++) {
//        cin >> u >> v;
//        addEdgeG(u, v);
//        addEdgeG(v, u);
//    }
//    dfs(1, 0);
//    cin >> q;
//    for (int t = 1; t <= q; t++) {
//        cin >> k;
//        for (int i = 1; i <= k; i++) {
//            cin >> arr[i];
//        }
//        cout << compute() << '\n';
//    }
//    return 0;
//}
```

==

文件: Code02_BigProject1.cpp

==

```
// 虚树(Virtual Tree)算法详解与应用
//
// 虚树是一种优化技术，用于解决树上多次询问的问题，每次询问涉及部分关键点
// 虚树只保留关键点及其两两之间的LCA，节点数控制在O(k)级别，从而提高效率
//
// 算法核心思想：
// 1. 虚树包含所有关键点和它们两两之间的LCA
// 2. 虚树的节点数不超过2*k-1（k为关键点数）
// 3. 在虚树上进行DP等操作，避免遍历整棵树
//
// 构造方法：
// 方法一：二次排序法
// 1. 将关键点按DFS序排序
// 2. 相邻点求LCA并加入序列
// 3. 再次排序并去重得到虚树所有节点
// 4. 按照父子关系连接节点
//
// 方法二：单调栈法
// 1. 将关键点按DFS序排序
// 2. 用栈维护虚树的一条链
// 3. 逐个插入关键点并维护栈结构
//
// 应用场景：
// 1. 树上多次询问，每次询问涉及部分关键点
// 2. 需要在关键点及其LCA上进行DP等操作
// 3. 数据范围要求∑k较小（通常≤10^5）
//
// 相关题目：
// 1. 洛谷 P4103 - [HEOI2014]大工程
//    链接：https://www.luogu.com.cn/problem/P4103
//    题意：给一棵树和多个询问，每个询问给出一些关键点，要求计算所有关键点对之间距离的和、最小值和最大值
//
// 2. Codeforces 613D - Kingdom and Cities
//    链接：https://codeforces.com/problemset/problem/613/D
//    题意：给一棵树和多个询问，每个询问给出一些关键点，要求切断最少的非关键点使关键点两两不连通
//
// 3. 洛谷 P2495 - [SDOI2011]消耗战
//    链接：https://www.luogu.com.cn/problem/P2495
```

```
//     题意：给一棵树和多个询问，每个询问给出一些关键点，要求切断最少代价的边使关键点都无法到达根节点
//
// 4. Codeforces 1000G - Two Melborians, One Siberian
//     链接：https://codeforces.com/problemset/problem/1000/G
//     题意：在树上处理多组询问，涉及关键点的最短距离等信息
//
// 5. AtCoder ABC154F - Many Many Paths
//     链接：https://atcoder.jp/contests/abc154/tasks/abc154_f
//     题意：计算树上路径数量，可以使用虚树优化
//
// 时间复杂度分析：
// 1. 预处理（DFS 序、LCA）：O(n log n)
// 2. 构造虚树：O(k log k)
// 3. 在虚树上 DP：O(k)
// 总体复杂度：O(n log n + Σk log k)
//
// 空间复杂度：O(n + k)
//
// 工程化考量：
// 1. 注意虚树边通常没有边权，需要通过原树计算
// 2. 清空关键点标记时避免使用 memset，用 for 循环逐个清除
// 3. 排序后的关键点顺序不是原节点序，如需按原序输出需额外保存
// 4. 虚树主要用于卡常题，需注意常数优化

// 大工程，C++版（简化版，避免标准库问题）
// 一共有 n 个节点，给定 n-1 条无向边，所有节点组成一棵树
// 如果在节点 a 和节点 b 之间建立新通道，那么代价是两个节点在树上的距离
// 一共有 q 条查询，每条查询格式如下
// 查询 k a1 a2 ... ak : 给出了 k 个不同的节点，任意两个节点之间都会建立新通道
//                       打印新通道的代价和、新通道中代价最小的值、新通道中代价最大的值
// 1 <= n <= 10^6
// 1 <= q <= 5 * 10^4
// 1 <= 所有查询给出的点的总数 <= 2 * n
// 测试链接 : https://www.luogu.com.cn/problem/P4103

const long long INF = 1LL << 60;
const int MAXN = 1000001;
const int MAXP = 20;

// 全局变量
int n, q, k;
```

```
int headg[MAXN], nextg[MAXN << 1], tog[MAXN << 1], cntg;

int headv[MAXN], nextv[MAXN], tov[MAXN], cntv;

int dep[MAXN], dfn[MAXN], stjump[MAXN][MAXP], cntd;

int arr[MAXN];
bool isKey[MAXN];

int tmp[MAXN << 1];
int stk[MAXN];

// siz[u]表示子树 u 上，关键点的数量
// sum[u]表示子树 u 上，所有关键点到 u 的总距离
// near[u]表示子树 u 上，到 u 最近关键点的距离
// far[u]表示子树 u 上，到 u 最远关键点的距离
int siz[MAXN];
long long sum[MAXN], near[MAXN], far[MAXN];

// 新通道的代价和、新通道中代价最小的值、新通道中代价最大的值
long long costSum, costMin, costMax;

// ufe 用于 dfs 和 dp 的迭代版
int ufe[MAXN][3];
int stacksize, u, f, e;

void push(int u_val, int f_val, int e_val) {
    ufe[stacksize][0] = u_val;
    ufe[stacksize][1] = f_val;
    ufe[stacksize][2] = e_val;
    stacksize++;
}

void pop() {
    stacksize--;
    u = ufe[stacksize][0];
    f = ufe[stacksize][1];
    e = ufe[stacksize][2];
}

// 原始树连边
void addEdgeG(int u, int v) {
    cntg++;
```

```
    nextg[cntg] = headg[u];
    tog[cntg] = v;
    headg[u] = cntg;
}

// 虚树连边
void addEdgeV(int u, int v) {
    cntv++;
    nextv[cntv] = headv[u];
    tov[cntv] = v;
    headv[u] = cntv;
}

// nums 中的数，根据 dfn 的大小排序，手撸双指针快排
void sortByDfn(int nums[], int l, int r) {
    if (l >= r) return;
    int i = l, j = r;
    int pivot = nums[(l + r) >> 1];
    while (i <= j) {
        while (dfn[nums[i]] < dfn[pivot]) i++;
        while (dfn[nums[j]] > dfn[pivot]) j--;
        if (i <= j) {
            int t = nums[i]; nums[i] = nums[j]; nums[j] = t;
            i++; j--;
        }
    }
    sortByDfn(nums, l, j);
    sortByDfn(nums, i, r);
}

// dfs 递归版，可能会爆栈
void dfs1(int u, int fa) {
    dep[u] = dep[fa] + 1;
    cntd++;
    dfn[u] = cntd;
    stjump[u][0] = fa;
    for (int p = 1; p < MAXP; p++) {
        stjump[u][p] = stjump[stjump[u][p - 1]][p - 1];
    }
    for (int e_idx = headg[u]; e_idx > 0; e_idx = nextg[e_idx]) {
        if (tog[e_idx] != fa) {
            dfs1(tog[e_idx], u);
        }
```

```
    }
}

// dfs1 的迭代版
void dfs2() {
    stacksize = 0;
    push(1, 0, -1);
    while (stacksize > 0) {
        pop();
        if (e == -1) {
            dep[u] = dep[f] + 1;
            cntd++;
            dfn[u] = cntd;
            stjump[u][0] = f;
            for (int p = 1; p < MAXP; p++) {
                stjump[u][p] = stjump[stjump[u][p - 1]][p - 1];
            }
            e = headg[u];
        } else {
            e = nextg[e];
        }
        if (e != 0) {
            push(u, f, e);
            if (tog[e] != f) {
                push(tog[e], u, -1);
            }
        }
    }
}

// 返回 a 和 b 的最低公共祖先
int getLca(int a, int b) {
    if (dep[a] < dep[b]) {
        int t = a; a = b; b = t;
    }
    for (int p = MAXP - 1; p >= 0; p--) {
        if (dep[stjump[a][p]] >= dep[b]) {
            a = stjump[a][p];
        }
    }
    if (a == b) {
        return a;
    }
```

```
    for (int p = MAXP - 1; p >= 0; p--) {
        if (stjump[a][p] != stjump[b][p]) {
            a = stjump[a][p];
            b = stjump[b][p];
        }
    }
    return stjump[a][0];
}

// 二次排序 + LCA 连边的方式建立虚树
int buildVirtualTree1() {
    sortByDfn(arr, 1, k);
    int len = 0;
    for (int i = 1; i < k; i++) {
        len++;
        tmp[len] = arr[i];
        len++;
        tmp[len] = getLca(arr[i], arr[i + 1]);
    }
    len++;
    tmp[len] = arr[k];
    sortByDfn(tmp, 1, len);
    int unique = 1;
    for (int i = 2; i <= len; i++) {
        if (tmp[unique] != tmp[i]) {
            unique++;
            tmp[unique] = tmp[i];
        }
    }
    cntv = 0;
    for (int i = 1; i <= unique; i++) {
        headv[tmp[i]] = 0;
    }
    for (int i = 1; i < unique; i++) {
        addEdgeV(getLca(tmp[i], tmp[i + 1]), tmp[i + 1]);
    }
    return tmp[1];
}

// 单调栈的方式建立虚树
int buildVirtualTree2() {
    sortByDfn(arr, 1, k);
    cntv = 0;
```

```
    headv[arr[1]] = 0;
    int top = 0;
    top++;
    stk[top] = arr[1];
    for (int i = 2; i <= k; i++) {
        int x = arr[i];
        int y = stk[top];
        int lca = getLca(x, y);
        while (top > 1 && dfn[stk[top - 1]] >= dfn[lca]) {
            addEdgeV(stk[top - 1], stk[top]);
            top--;
        }
        if (lca != stk[top]) {
            headv[lca] = 0;
            addEdgeV(lca, stk[top]);
            stk[top] = lca;
        }
        headv[x] = 0;
        top++;
        stk[top] = x;
    }
    while (top > 1) {
        addEdgeV(stk[top - 1], stk[top]);
        top--;
    }
    return stk[1];
}

// dp递归版，可能会爆栈
void dp1(int u) {
    siz[u] = isKey[u] ? 1 : 0;
    sum[u] = 0;
    if (isKey[u]) {
        far[u] = near[u] = 0;
    } else {
        near[u] = INF;
        far[u] = -INF;
    }
    for (int e_idx = headv[u]; e_idx > 0; e_idx = nextv[e_idx]) {
        dp1(tov[e_idx]);
    }
    for (int e_idx = headv[u]; e_idx > 0; e_idx = nextv[e_idx]) {
        int v = tov[e_idx];
```

```
        long long len = dep[v] - dep[u];
        costSum += (sum[u] + (long long)siz[u] * len) * siz[v] + sum[v] * siz[u];
        siz[u] += siz[v];
        sum[u] += sum[v] + len * siz[v];
        if (near[u] + near[v] + len < costMin) costMin = near[u] + near[v] + len;
        if (far[u] + far[v] + len > costMax) costMax = far[u] + far[v] + len;
        if (near[v] + len < near[u]) near[u] = near[v] + len;
        if (far[v] + len > far[u]) far[u] = far[v] + len;
    }
}

// dp1 的迭代版
void dp2(int tree) {
    stacksize = 0;
    push(tree, 0, -1);
    while (stacksize > 0) {
        pop();
        if (e == -1) {
            siz[u] = isKey[u] ? 1 : 0;
            sum[u] = 0;
            if (isKey[u]) {
                far[u] = near[u] = 0;
            } else {
                near[u] = INF;
                far[u] = -INF;
            }
            e = headv[u];
        } else {
            e = nextv[e];
        }
        if (e != 0) {
            push(u, 0, e);
            push(tov[e], 0, -1);
        } else {
            for (int ei = headv[u]; ei > 0; ei = nextv[ei]) {
                int v = tov[ei];
                long long len = dep[v] - dep[u];
                costSum += (sum[u] + (long long)siz[u] * len) * siz[v] + sum[v] * siz[u];
                siz[u] += siz[v];
                sum[u] += sum[v] + len * siz[v];
                long long tempMin = near[u] + near[v] + len;
                long long tempMax = far[u] + far[v] + len;
                if (tempMin < costMin) costMin = tempMin;
```

```
                if (tempMax > costMax) costMax = tempMax;
                if (near[v] + len < near[u]) near[u] = near[v] + len;
                if (far[v] + len > far[u]) far[u] = far[v] + len;
            }
        }
    }
}

void compute() {
    // 节点标记关键点信息
    for (int i = 1; i <= k; i++) {
        isKey[arr[i]] = true;
    }
    int tree = buildVirtualTree1();
    // int tree = buildVirtualTree2();
    costSum = 0;
    costMin = INF;
    costMax = -INF;
    // dp1(tree);  // 递归版，可能会爆栈
    dp2(tree);     // 迭代版，推荐使用
    // 节点撤销关键点信息
    for (int i = 1; i <= k; i++) {
        isKey[arr[i]] = false;
    }
}

// 由于编译环境问题，这里不包含 main 函数
// 在实际使用中，需要添加适当的输入输出函数
```

==

文件: Code02_BigProject1.java

==

```
package class180;

// 虚树(Virtual Tree)算法详解与应用
//
// 虚树是一种优化技术，用于解决树上多次询问的问题，每次询问涉及部分关键点
// 虚树只保留关键点及其两两之间的 LCA，节点数控制在 O(k)级别，从而提高效率
//
// 算法核心思想：
// 1. 虚树包含所有关键点和它们两两之间的 LCA
// 2. 虚树的节点数不超过 2*k-1（k 为关键点数）
```

```
// 3. 在虚树上进行 DP 等操作，避免遍历整棵树
//
// 构造方法：
// 方法一：二次排序法
// 1. 将关键点按 DFS 序排序
// 2. 相邻点求 LCA 并加入序列
// 3. 再次排序并去重得到虚树所有节点
// 4. 按照父子关系连接节点
//
// 方法二：单调栈法
// 1. 将关键点按 DFS 序排序
// 2. 用栈维护虚树的一条链
// 3. 逐个插入关键点并维护栈结构
//
// 应用场景：
// 1. 树上多次询问，每次询问涉及部分关键点
// 2. 需要在关键点及其 LCA 上进行 DP 等操作
// 3. 数据范围要求∑k 较小（通常≤10^5）
//
// 相关题目：
// 1. 洛谷 P4103 - [HEOI2014]大工程
//    链接：https://www.luogu.com.cn/problem/P4103
//    题意：给一棵树和多个询问，每个询问给出一些关键点，要求计算所有关键点对之间距离的和、最小值和最大值
//
// 2. Codeforces 613D - Kingdom and Cities
//    链接：https://codeforces.com/problemset/problem/613/D
//    题意：给一棵树和多个询问，每个询问给出一些关键点，要求切断最少的非关键点使关键点两两不连通
//
// 3. 洛谷 P2495 - [SDOI2011]消耗战
//    链接：https://www.luogu.com.cn/problem/P2495
//    题意：给一棵树和多个询问，每个询问给出一些关键点，要求切断最少代价的边使关键点都无法到达根节点
//
// 4. Codeforces 1000G - Two Melborians, One Siberian
//    链接：https://codeforces.com/problemset/problem/1000/G
//    题意：在树上处理多组询问，涉及关键点的最短距离等信息
//
// 5. AtCoder ABC154F - Many Many Paths
//    链接：https://atcoder.jp/contests/abc154/tasks/abc154_f
//    题意：计算树上路径数量，可以使用虚树优化
//
// 6. LOJ #6056 - 「雅礼集训 2017 Day11」回转寿司
```

```
//    链接：https://loj.ac/p/6056
//    题意：涉及树上关键点的查询问题
//
// 7. 洛谷 P3233 - [HNOI2014]世界树
//    链接：https://www.luogu.com.cn/problem/P3233
//    题意：给一棵树和多个询问，每个询问给出一些关键点，要求计算每个关键点能管理多少个点
//
// 8. Codeforces 1109D - Treeland and Viruses
//    链接：https://codeforces.com/problemset/problem/1109/D
//    题意：给一棵树和多个病毒源点，每个病毒源点以不同速度扩散，求每个点被哪个病毒源点感染
//
// 9. 洛谷 P3320 - [SDOI2015]寻宝游戏
//    链接：https://www.luogu.com.cn/problem/P3320
//    题意：给一棵树和多个操作，每次操作翻转一个点的状态，求收集所有宝藏的最短路径长度
//
// 10. 洛谷 P5327 - [ZJOI2019]语言
//    链接：https://www.luogu.com.cn/problem/P5327
//    题意：涉及树上路径覆盖的复杂问题
//
// 11. SPOJ QTREE5 - Query on a tree V
//    链接：https://www.spoj.com/problems/QTREE5/
//    题意：树上点颜色修改和查询距离最近的白色节点
//
// 12. 洛谷 P3232 - [HNOI2013]游走
//    链接：https://www.luogu.com.cn/problem/P3232
//    题意：给定无向连通图，通过高斯消元计算边的期望经过次数，再贪心编号使总得分期望最小
//
// 时间复杂度分析：
// 1. 预处理（DFS 序、LCA）：O(n log n)
// 2. 构造虚树：O(k log k)
// 3. 在虚树上 DP：O(k)
// 总体复杂度：O(n log n + ∑k log k)
//
// 空间复杂度：O(n + k)
//
// 工程化考量：
// 1. 注意虚树边通常没有边权，需要通过原树计算
// 2. 清空关键点标记时避免使用 memset，用 for 循环逐个清除
// 3. 排序后的关键点顺序不是原节点序，如需按原序输出需额外保存
// 4. 虚树主要用于卡常题，需注意常数优化
//
// 算法设计本质与核心思想：
// 1. 设计动机：虚树算法的核心动机是优化树上多次询问问题。当需要对树上不同关键点集合进行多次查询
```

时，
// 　　如果每次都遍历整棵树，时间复杂度会很高。虚树通过只保留关键点及其 LCA，将问题规模从 O(n)降低到 O(k)。
// 2. 数学原理：
// 　　- LCA 性质：任意两个节点的 LCA 在 DFS 序上具有特定性质，可以用于构建虚树
// 　　- 节点数量上界：虚树节点数不超过 2*k-1，这是通过数学归纳法可以证明的
// 　　- 树的结构保持：虚树保持了原树中关键点之间的祖先关系
// 3. 与其它算法的关联：
// 　　- 树上倍增：虚树构建需要 LCA，通常使用树上倍增算法
// 　　- 树形 DP：虚树上的动态规划是解决问题的核心
// 　　- 单调栈：构建虚树时使用的单调栈技巧与其它算法中的单调栈类似
// 4. 工程化应用：
// 　　- 内存优化：避免使用全局数组清零，用循环逐个清除
// 　　- 常数优化：选择合适的虚树构建方法（单调栈法通常更快）
// 　　- 边界处理：正确处理根节点、叶子节点等特殊情况
//
// 语言特性差异与跨语言实现：
// 1. Java 实现特点：
// 　　- 使用对象封装，代码结构清晰
// 　　- 自定义 FastReader 提高输入效率
// 　　- 递归深度可能受限，需要改用迭代实现
// 2. C++实现特点：
// 　　- 性能最优，适合大数据量
// 　　- 需要注意编译环境问题，避免使用复杂 STL
// 　　- 指针操作灵活但需谨慎
// 3. Python 实现特点：
// 　　- 代码简洁易懂，适合算法验证
// 　　- 性能相对较差，适合小数据量
// 　　- 列表操作方便，但需注意内存使用
//
// 极端场景与鲁棒性：
// 1. 空输入处理：关键点为空时的特殊处理
// 2. 极端数据规模：关键点数量接近节点总数、树退化为链的情况、深度很大的树结构
// 3. 边界条件：关键点包含根节点、关键点之间存在父子关系、关键点相邻的情况
//
// 性能优化策略：
// 1. 算法层面优化：选择合适的虚树构建方法、优化 DP 状态转移方程、预处理减少重复计算
// 2. 实现层面优化：减少函数调用开销、优化内存访问模式、使用位运算等底层优化技巧
// 3. 工程层面优化：输入输出优化、内存池技术、缓存友好设计
//
// 调试技巧与问题定位：
// 1. 中间过程打印：打印 DFS 序、打印 LCA 计算结果、打印虚树构建过程
// 2. 断言验证：验证虚树节点数量上界、验证关键点标记正确性、验证 DP 状态转移正确性

```java
// 3. 性能分析：使用性能分析工具定位瓶颈、对比不同实现的性能差异、分析时间复杂度常数项影响

// 大工程，java版
// 一共有n个节点，给定n-1条无向边，所有节点组成一棵树
// 如果在节点a和节点b之间建立新通道，那么代价是两个节点在树上的距离
// 一共有q条查询，每条查询格式如下
// 查询 k a1 a2 ... ak : 给出了k个不同的节点，任意两个节点之间都会建立新通道
//                       打印新通道的代价和、新通道中代价最小的值、新通道中代价最大的值
// 1 <= n <= 10^6
// 1 <= q <= 5 * 10^4
// 1 <= 所有查询给出的点的总数 <= 2 * n
// 测试链接 : https://www.luogu.com.cn/problem/P4103
// 提交以下的code，提交时请把类名改成"Main"，可以通过所有测试用例

import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStreamWriter;
import java.io.PrintWriter;

public class Code02_BigProject1 {

    public static int MAXN = 1000001;
    public static int MAXP = 20;
    public static long INF = 1L << 60;
    public static int n, q, k;

    public static int[] headg = new int[MAXN];
    public static int[] nextg = new int[MAXN << 1];
    public static int[] tog = new int[MAXN << 1];
    public static int cntg;

    public static int[] headv = new int[MAXN];
    public static int[] nextv = new int[MAXN];
    public static int[] tov = new int[MAXN];
    public static int cntv;

    public static int[] dep = new int[MAXN];
    public static int[] dfn = new int[MAXN];
    public static int[][] stjump = new int[MAXN][MAXP];
    public static int cntd;

    public static int[] arr = new int[MAXN];
    public static boolean[] isKey = new boolean[MAXN];
```

```java
public static int[] tmp = new int[MAXN << 1];
public static int[] stk = new int[MAXN];

// siz[u]表示子树 u 上，关键点的数量
// sum[u]表示子树 u 上，所有关键点到 u 的总距离
// near[u]表示子树 u 上，到 u 最近关键点的距离
// far[u]表示子树 u 上，到 u 最远关键点的距离
public static int[] siz = new int[MAXN];
public static long[] sum = new long[MAXN];
public static long[] near = new long[MAXN];
public static long[] far = new long[MAXN];

// 新通道的代价和、新通道中代价最小的值、新通道中代价最大的值
public static long costSum, costMin, costMax;

// dfs 过程和 dp 过程，C++同学可以使用递归版
// 但是 java 同学必须改迭代版否则会爆栈
// 不会改迭代版，去看讲解 118，详解了从递归版改迭代版
// ufe 不仅用于 dfs 改迭代，也用于 dp 改迭代
public static int[][] ufe = new int[MAXN][3];

public static int stacksize, u, f, e;

public static void push(int u, int f, int e) {
    ufe[stacksize][0] = u;
    ufe[stacksize][1] = f;
    ufe[stacksize][2] = e;
    stacksize++;
}

public static void pop() {
    --stacksize;
    u = ufe[stacksize][0];
    f = ufe[stacksize][1];
    e = ufe[stacksize][2];
}

public static void addEdgeG(int u, int v) {
    nextg[++cntg] = headg[u];
    tog[cntg] = v;
    headg[u] = cntg;
}
```

```java
public static void addEdgeV(int u, int v) {
    nextv[++cntv] = headv[u];
    tov[cntv] = v;
    headv[u] = cntv;
}

public static void sortByDfn(int[] nums, int l, int r) {
    if (l >= r) return;
    int i = l, j = r;
    int pivot = nums[(l + r) >> 1];
    while (i <= j) {
        while (dfn[nums[i]] < dfn[pivot]) i++;
        while (dfn[nums[j]] > dfn[pivot]) j--;
        if (i <= j) {
            int tmp = nums[i]; nums[i] = nums[j]; nums[j] = tmp;
            i++; j--;
        }
    }
    sortByDfn(nums, l, j);
    sortByDfn(nums, i, r);
}

// dfs递归版，java会爆栈，C++可以通过
public static void dfs1(int u, int fa) {
    dep[u] = dep[fa] + 1;
    dfn[u] = ++cntd;
    stjump[u][0] = fa;
    for (int p = 1; p < MAXP; p++) {
        stjump[u][p] = stjump[stjump[u][p - 1]][p - 1];
    }
    for (int e = headg[u]; e > 0; e = nextg[e]) {
        if (tog[e] != fa) {
            dfs1(tog[e], u);
        }
    }
}

// dfs1的迭代版
public static void dfs2() {
    stacksize = 0;
    push(1, 0, -1);
    while (stacksize > 0) {
```

```java
        pop();
        if (e == -1) {
            dep[u] = dep[f] + 1;
            dfn[u] = ++cntd;
            stjump[u][0] = f;
            for (int p = 1; p < MAXP; p++) {
                stjump[u][p] = stjump[stjump[u][p - 1]][p - 1];
            }
            e = headg[u];
        } else {
            e = nextg[e];
        }
        if (e != 0) {
            push(u, f, e);
            if (tog[e] != f) {
                push(tog[e], u, -1);
            }
        }
    }
}

public static int getLca(int a, int b) {
    if (dep[a] < dep[b]) {
        int tmp = a; a = b; b = tmp;
    }
    for (int p = MAXP - 1; p >= 0; p--) {
        if (dep[stjump[a][p]] >= dep[b]) {
            a = stjump[a][p];
        }
    }
    if (a == b) {
        return a;
    }
    for (int p = MAXP - 1; p >= 0; p--) {
        if (stjump[a][p] != stjump[b][p]) {
            a = stjump[a][p];
            b = stjump[b][p];
        }
    }
    return stjump[a][0];
}

// 二次排序 + LCA 连边的方式建立虚树
```

```java
public static int buildVirtualTree1() {
    sortByDfn(arr, 1, k);
    int len = 0;
    for (int i = 1; i < k; i++) {
        tmp[++len] = arr[i];
        tmp[++len] = getLca(arr[i], arr[i + 1]);
    }
    tmp[++len] = arr[k];
    sortByDfn(tmp, 1, len);
    int unique = 1;
    for (int i = 2; i <= len; i++) {
        if (tmp[unique] != tmp[i]) {
            tmp[++unique] = tmp[i];
        }
    }
    cntv = 0;
    for (int i = 1; i <= unique; i++) {
        headv[tmp[i]] = 0;
    }
    for (int i = 1; i < unique; i++) {
        addEdgeV(getLca(tmp[i], tmp[i + 1]), tmp[i + 1]);
    }
    return tmp[1];
}

// 单调栈的方式建立虚树
public static int buildVirtualTree2() {
    sortByDfn(arr, 1, k);
    cntv = 0;
    headv[arr[1]] = 0;
    int top = 0;
    stk[++top] = arr[1];
    for (int i = 2; i <= k; i++) {
        int x = arr[i];
        int y = stk[top];
        int lca = getLca(x, y);
        while (top > 1 && dfn[stk[top - 1]] >= dfn[lca]) {
            addEdgeV(stk[top - 1], stk[top]);
            top--;
        }
        if (lca != stk[top]) {
            headv[lca] = 0;
            addEdgeV(lca, stk[top]);
```

```java
            stk[top] = lca;
        }
        headv[x] = 0;
        stk[++top] = x;
    }
    while (top > 1) {
        addEdgeV(stk[top - 1], stk[top]);
        top--;
    }
    return stk[1];
}

// dp 递归版，java 会爆栈，C++可以通过
public static void dp1(int u) {
    siz[u] = isKey[u] ? 1 : 0;
    sum[u] = 0;
    if (isKey[u]) {
        far[u] = near[u] = 0;
    } else {
        near[u] = INF;
        far[u] = -INF;
    }
    for (int e = headv[u]; e > 0; e = nextv[e]) {
        dp1(tov[e]);
    }
    for (int e = headv[u]; e > 0; e = nextv[e]) {
        int v = tov[e];
        long len = dep[v] - dep[u];
        costSum += (sum[u] + 1L * siz[u] * len) * siz[v] + sum[v] * siz[u];
        siz[u] += siz[v];
        sum[u] += sum[v] + len * siz[v];
        costMin = Math.min(costMin, near[u] + near[v] + len);
        costMax = Math.max(costMax, far[u] + far[v] + len);
        near[u] = Math.min(near[u], near[v] + len);
        far[u] = Math.max(far[u], far[v] + len);
    }
}

// dp1 的迭代版
public static void dp2(int tree) {
    stacksize = 0;
    push(tree, 0, -1);
    while (stacksize > 0) {
```

```
            pop();
            if (e == -1) {
                siz[u] = isKey[u] ? 1 : 0;
                sum[u] = 0;
                if (isKey[u]) {
                    far[u] = near[u] = 0;
                } else {
                    near[u] = INF;
                    far[u] = -INF;
                }
                e = headv[u];
            } else {
                e = nextv[e];
            }
            if (e != 0) {
                push(u, 0, e);
                push(tov[e], 0, -1);
            } else {
                for (int ei = headv[u]; ei > 0; ei = nextv[ei]) {
                    int v = tov[ei];
                    long len = dep[v] - dep[u];
                    costSum += (sum[u] + 1L * siz[u] * len) * siz[v] + sum[v] * siz[u];
                    siz[u] += siz[v];
                    sum[u] += sum[v] + len * siz[v];
                    costMin = Math.min(costMin, near[u] + near[v] + len);
                    costMax = Math.max(costMax, far[u] + far[v] + len);
                    near[u] = Math.min(near[u], near[v] + len);
                    far[u] = Math.max(far[u], far[v] + len);
                }
            }
        }
    }

    public static void compute() {
        for (int i = 1; i <= k; i++) {
            isKey[arr[i]] = true;
        }
        int tree = buildVirtualTree1();
        // int tree = buildVirtualTree2();
        costSum = 0;
        costMin = INF;
        costMax = -INF;
        // dp1(tree);
```

```java
        dp2(tree);
        for (int i = 1; i <= k; i++) {
            isKey[arr[i]] = false;
        }
    }

    public static void main(String[] args) throws Exception {
        FastReader in = new FastReader(System.in);
        PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));
        n = in.nextInt();
        for (int i = 1, u, v; i < n; i++) {
            u = in.nextInt();
            v = in.nextInt();
            addEdgeG(u, v);
            addEdgeG(v, u);
        }
        // dfs1(1, 0);
        dfs2();
        q = in.nextInt();
        for (int t = 1; t <= q; t++) {
            k = in.nextInt();
            for (int i = 1; i <= k; i++) {
                arr[i] = in.nextInt();
            }
            compute();
            out.println(costSum + " " + costMin + " " + costMax);
        }
        out.flush();
        out.close();
    }

    // 读写工具类
    static class FastReader {
        private final byte[] buffer = new byte[1 << 16];
        private int ptr = 0, len = 0;
        private final InputStream in;

        FastReader(InputStream in) {
            this.in = in;
        }

        private int readByte() throws IOException {
            if (ptr >= len) {
```

```java
                len = in.read(buffer);
                ptr = 0;
                if (len <= 0)
                    return -1;
            }
            return buffer[ptr++];
        }

        int nextInt() throws IOException {
            int c;
            do {
                c = readByte();
            } while (c <= ' ' && c != -1);
            boolean neg = false;
            if (c == '-') {
                neg = true;
                c = readByte();
            }
            int val = 0;
            while (c > ' ' && c != -1) {
                val = val * 10 + (c - '0');
                c = readByte();
            }
            return neg ? -val : val;
        }
    }

}
```

==

文件: Code02_BigProject1.py

==

```python
# 虚树(Virtual Tree)算法详解与应用
#
# 虚树是一种优化技术，用于解决树上多次询问的问题，每次询问涉及部分关键点
# 虚树只保留关键点及其两两之间的LCA，节点数控制在O(k)级别，从而提高效率
#
# 算法核心思想：
# 1. 虚树包含所有关键点和它们两两之间的LCA
# 2. 虚树的节点数不超过2*k-1（k为关键点数）
# 3. 在虚树上进行DP等操作，避免遍历整棵树
#
```

```
# 构造方法：
# 方法一：二次排序法
# 1. 将关键点按 DFS 序排序
# 2. 相邻点求 LCA 并加入序列
# 3. 再次排序并去重得到虚树所有节点
# 4. 按照父子关系连接节点
#
# 方法二：单调栈法
# 1. 将关键点按 DFS 序排序
# 2. 用栈维护虚树的一条链
# 3. 逐个插入关键点并维护栈结构
#
# 应用场景：
# 1. 树上多次询问，每次询问涉及部分关键点
# 2. 需要在关键点及其 LCA 上进行 DP 等操作
# 3. 数据范围要求∑k 较小（通常≤10^5）
#
# 相关题目：
# 1. 洛谷 P4103 - [HEOI2014]大工程
#    链接：https://www.luogu.com.cn/problem/P4103
#    题意：给一棵树和多个询问，每个询问给出一些关键点，要求计算所有关键点对之间距离的和、最小值和最大值
#
# 2. Codeforces 613D - Kingdom and Cities
#    链接：https://codeforces.com/problemset/problem/613/D
#    题意：给一棵树和多个询问，每个询问给出一些关键点，要求切断最少的非关键点使关键点两两不连通
#
# 3. 洛谷 P2495 - [SDOI2011]消耗战
#    链接：https://www.luogu.com.cn/problem/P2495
#    题意：给一棵树和多个询问，每个询问给出一些关键点，要求切断最少代价的边使关键点都无法到达根节点
#
# 4. Codeforces 1000G - Two Melborians, One Siberian
#    链接：https://codeforces.com/problemset/problem/1000/G
#    题意：在树上处理多组询问，涉及关键点的最短距离等信息
#
# 5. AtCoder ABC154F - Many Many Paths
#    链接：https://atcoder.jp/contests/abc154/tasks/abc154_f
#    题意：计算树上路径数量，可以使用虚树优化
#
# 时间复杂度分析：
# 1. 预处理（DFS 序、LCA）：O(n log n)
# 2. 构造虚树：O(k log k)
```

```
# 3. 在虚树上 DP：O(k)
# 总体复杂度：O(n log n + Σk log k)
#
# 空间复杂度：O(n + k)
#
# 工程化考量：
# 1. 注意虚树边通常没有边权，需要通过原树计算
# 2. 清空关键点标记时避免使用 memset，用 for 循环逐个清除
# 3. 排序后的关键点顺序不是原节点序，如需按原序输出需额外保存
# 4. 虚树主要用于卡常题，需注意常数优化

# 大工程，Python 版
# 一共有 n 个节点，给定 n-1 条无向边，所有节点组成一棵树
# 如果在节点 a 和节点 b 之间建立新通道，那么代价是两个节点在树上的距离
# 一共有 q 条查询，每条查询格式如下
# 查询 k a1 a2 ... ak : 给出了 k 个不同的节点，任意两个节点之间都会建立新通道
#                       打印新通道的代价和、新通道中代价最小的值、新通道中代价最大的值
# 1 <= n <= 10^6
# 1 <= q <= 5 * 10^4
# 1 <= 所有查询给出的点的总数 <= 2 * n
# 测试链接 : https://www.luogu.com.cn/problem/P4103

import sys
from collections import deque

MAXN = 1000001
MAXP = 20
INF = 1 << 60

# 全局变量
n, q, k = 0, 0, 0

# 原始树
headg = [0] * MAXN
nextg = [0] * (MAXN << 1)
tog = [0] * (MAXN << 1)
cntg = 0

# 虚树
headv = [0] * MAXN
nextv = [0] * MAXN
tov = [0] * MAXN
cntv = 0
```

```python
# 树上倍增求LCA + 生成dfn序
dep = [0] * MAXN
dfn = [0] * MAXN
stjump = [[0] * MAXP for _ in range(MAXN)]
cntd = 0

# 关键点数组
arr = [0] * MAXN
isKey = [False] * MAXN

# 第一种建树方式
tmp = [0] * (MAXN << 1)
# 第二种建树方式
stk = [0] * MAXN

# 动态规划相关
# siz[u]表示子树u上，关键点的数量
# sum[u]表示子树u上，所有关键点到u的总距离
# near[u]表示子树u上，到u最近关键点的距离
# far[u]表示子树u上，到u最远关键点的距离
siz = [0] * MAXN
sum_ = [0] * MAXN  # 使用sum_避免与内置sum函数冲突
near = [0] * MAXN
far = [0] * MAXN

# 新通道的代价和、新通道中代价最小的值、新通道中代价最大的值
costSum, costMin, costMax = 0, 0, 0

# ufe用于dfs和dp的迭代版
ufe = [[0, 0, 0] for _ in range(MAXN)]
stacksize, u, f, e = 0, 0, 0, 0

def push(u_val, f_val, e_val):
    global stacksize
    ufe[stacksize][0] = u_val
    ufe[stacksize][1] = f_val
    ufe[stacksize][2] = e_val
    stacksize += 1

def pop():
    global stacksize, u, f, e
    stacksize -= 1
```

```python
    u = ufe[stacksize][0]
    f = ufe[stacksize][1]
    e = ufe[stacksize][2]

# 原始树连边
def addEdgeG(u, v):
    global cntg
    cntg += 1
    nextg[cntg] = headg[u]
    tog[cntg] = v
    headg[u] = cntg

# 虚树连边
def addEdgeV(u, v):
    global cntv
    cntv += 1
    nextv[cntv] = headv[u]
    tov[cntv] = v
    headv[u] = cntv

# nums 中的数，根据 dfn 的大小排序，手撸双指针快排
def sortByDfn(nums, l, r):
    if l >= r:
        return
    i, j = l, r
    pivot = nums[(l + r) >> 1]
    while i <= j:
        while dfn[nums[i]] < dfn[pivot]:
            i += 1
        while dfn[nums[j]] > dfn[pivot]:
            j -= 1
        if i <= j:
            nums[i], nums[j] = nums[j], nums[i]
            i += 1
            j -= 1
    sortByDfn(nums, l, j)
    sortByDfn(nums, i, r)

# dfs 递归版，可能会爆栈
def dfs1(u, fa):
    global cntd
    dep[u] = dep[fa] + 1
    cntd += 1
```

```
    dfn[u] = cntd
    stjump[u][0] = fa
    for p in range(1, MAXP):
        stjump[u][p] = stjump[stjump[u][p - 1]][p - 1]
    e_idx = headg[u]
    while e_idx > 0:
        if tog[e_idx] != fa:
            dfs1(tog[e_idx], u)
        e_idx = nextg[e_idx]

# dfs1 的迭代版
def dfs2():
    global stacksize, u, f, e, cntd
    stacksize = 0
    push(1, 0, -1)
    while stacksize > 0:
        pop()
        if e == -1:
            dep[u] = dep[f] + 1
            cntd += 1
            dfn[u] = cntd
            stjump[u][0] = f
            for p in range(1, MAXP):
                stjump[u][p] = stjump[stjump[u][p - 1]][p - 1]
            e = headg[u]
        else:
            e = nextg[e]
        if e != 0:
            push(u, f, e)
            if tog[e] != f:
                push(tog[e], u, -1)

# 返回 a 和 b 的最低公共祖先
def getLca(a, b):
    if dep[a] < dep[b]:
        a, b = b, a
    for p in range(MAXP - 1, -1, -1):
        if dep[stjump[a][p]] >= dep[b]:
            a = stjump[a][p]
    if a == b:
        return a
    for p in range(MAXP - 1, -1, -1):
        if stjump[a][p] != stjump[b][p]:
```

```python
            a = stjump[a][p]
            b = stjump[b][p]
    return stjump[a][0]

# 二次排序 + LCA 连边的方式建立虚树
def buildVirtualTree1():
    sortByDfn(arr, 1, k)
    len_idx = 0
    for i in range(1, k):
        len_idx += 1
        tmp[len_idx] = arr[i]
        len_idx += 1
        tmp[len_idx] = getLca(arr[i], arr[i + 1])
    len_idx += 1
    tmp[len_idx] = arr[k]
    sortByDfn(tmp, 1, len_idx)
    unique = 1
    for i in range(2, len_idx + 1):
        if tmp[unique] != tmp[i]:
            unique += 1
            tmp[unique] = tmp[i]
    global cntv
    cntv = 0
    for i in range(1, unique + 1):
        headv[tmp[i]] = 0
    for i in range(1, unique):
        addEdgeV(getLca(tmp[i], tmp[i + 1]), tmp[i + 1])
    return tmp[1]

# 单调栈的方式建立虚树
def buildVirtualTree2():
    sortByDfn(arr, 1, k)
    global cntv
    cntv = 0
    headv[arr[1]] = 0
    top = 0
    top += 1
    stk[top] = arr[1]
    for i in range(2, k + 1):
        x = arr[i]
        y = stk[top]
        lca = getLca(x, y)
        while top > 1 and dfn[stk[top - 1]] >= dfn[lca]:
```

```
            addEdgeV(stk[top - 1], stk[top])
            top -= 1
        if lca != stk[top]:
            headv[lca] = 0
            addEdgeV(lca, stk[top])
            stk[top] = lca
        headv[x] = 0
        top += 1
        stk[top] = x
    while top > 1:
        addEdgeV(stk[top - 1], stk[top])
        top -= 1
    return stk[1]

# dp 递归版，可能会爆栈
def dp1(u):
    global costSum, costMin, costMax
    siz[u] = 1 if isKey[u] else 0
    sum_[u] = 0
    if isKey[u]:
        far[u] = near[u] = 0
    else:
        near[u] = INF
        far[u] = -INF
    e_idx = headv[u]
    while e_idx > 0:
        dp1(tov[e_idx])
        e_idx = nextv[e_idx]
    e_idx = headv[u]
    while e_idx > 0:
        v = tov[e_idx]
        len_val = dep[v] - dep[u]
        costSum += (sum_[u] + siz[u] * len_val) * siz[v] + sum_[v] * siz[u]
        siz[u] += siz[v]
        sum_[u] += sum_[v] + len_val * siz[v]
        costMin = min(costMin, near[u] + near[v] + len_val)
        costMax = max(costMax, far[u] + far[v] + len_val)
        near[u] = min(near[u], near[v] + len_val)
        far[u] = max(far[u], far[v] + len_val)
        e_idx = nextv[e_idx]

# dp1 的迭代版
def dp2(tree):
```

```python
    global stacksize, u, f, e, costSum, costMin, costMax
    stacksize = 0
    push(tree, 0, -1)
    while stacksize > 0:
        pop()
        if e == -1:
            siz[u] = 1 if isKey[u] else 0
            sum_[u] = 0
            if isKey[u]:
                far[u] = near[u] = 0
            else:
                near[u] = INF
                far[u] = -INF
            e = headv[u]
        else:
            e = nextv[e]
        if e != 0:
            push(u, 0, e)
            push(tov[e], 0, -1)
        else:
            ei = headv[u]
            while ei > 0:
                v = tov[ei]
                len_val = dep[v] - dep[u]
                costSum += (sum_[u] + siz[u] * len_val) * siz[v] + sum_[v] * siz[u]
                siz[u] += siz[v]
                sum_[u] += sum_[v] + len_val * siz[v]
                costMin = min(costMin, near[u] + near[v] + len_val)
                costMax = max(costMax, far[u] + far[v] + len_val)
                near[u] = min(near[u], near[v] + len_val)
                far[u] = max(far[u], far[v] + len_val)
                ei = nextv[ei]

def compute():
    # 节点标记关键点信息
    for i in range(1, k + 1):
        isKey[arr[i]] = True
    tree = buildVirtualTree1()
    # tree = buildVirtualTree2()
    global costSum, costMin, costMax
    costSum = 0
    costMin = INF
    costMax = -INF
```

```
    # dp1(tree)
    dp2(tree)
    # 节点撤销关键点信息
    for i in range(1, k + 1):
        isKey[arr[i]] = False

# 主函数
if __name__ == "__main__":
    # 读取输入
    n = int(input())
    for i in range(1, n):
        u, v = map(int, input().split())
        addEdgeG(u, v)
        addEdgeG(v, u)
    # dfs1(1, 0)  # 递归版，可能会爆栈
    dfs2()  # 迭代版，推荐使用
    q = int(input())
    for t in range(1, q + 1):
        k = int(input())
        arr_values = list(map(int, input().split()))
        for i in range(1, k + 1):
            arr[i] = arr_values[i - 1]
        compute()
        print(costSum, costMin, costMax)
```

===

文件: Code02_BigProject2.java

===

```
package class180;

// 大工程，C++版
// 一共有 n 个节点，给定 n-1 条无向边，所有节点组成一棵树
// 如果在节点 a 和节点 b 之间建立新通道，那么代价是两个节点在树上的距离
// 一共有 q 条查询，每条查询格式如下
// 查询 k a1 a2 ... ak : 给出了 k 个不同的节点，任意两个节点之间都会建立新通道
//                       打印新通道的代价和、新通道中代价最小的值、新通道中代价最大的值
// 1 <= n <= 10^6
// 1 <= q <= 5 * 10^4
// 1 <= 所有查询给出的点的总数 <= 2 * n
// 测试链接 : https://www.luogu.com.cn/problem/P4103
// 如下实现是 C++的版本，C++版本和 java 版本逻辑完全一样
// 提交如下代码，可以通过所有测试用例
```

```
//#include <bits/stdc++.h>
//
//using namespace std;
//
//const int MAXN = 1000001;
//const int MAXP = 20;
//const long long INF = 1LL << 60;
//int n, q, k;
//
//int headg[MAXN];
//int nextg[MAXN << 1];
//int tog[MAXN << 1];
//int cntg;
//
//int headv[MAXN];
//int nextv[MAXN];
//int tov[MAXN];
//int cntv;
//
//int dep[MAXN];
//int dfn[MAXN];
//int stjump[MAXN][MAXP];
//int cntd;
//
//int arr[MAXN];
//bool isKey[MAXN];
//
//int tmp[MAXN << 1];
//int stk[MAXN];
//
//int siz[MAXN];
//long long sum[MAXN];
//long long near[MAXN];
//long long far[MAXN];
//long long costSum, costMin, costMax;
//
//bool cmp(int x, int y) {
//    return dfn[x] < dfn[y];
//}
//
//void addEdgeG(int u, int v) {
//    nextg[++cntg] = headg[u];
```

```
//    tog[cntg] = v;
//    headg[u] = cntg;
//}
//
//void addEdgeV(int u, int v) {
//    nextv[++cntv] = headv[u];
//    tov[cntv] = v;
//    headv[u] = cntv;
//}
//
//void dfs(int u, int fa) {
//    dep[u] = dep[fa] + 1;
//    dfn[u] = ++cntd;
//    stjump[u][0] = fa;
//    for (int p = 1; p < MAXP; p++) {
//        stjump[u][p] = stjump[ stjump[u][p - 1] ][p - 1];
//    }
//    for (int e = headg[u]; e; e = nextg[e]) {
//        int v = tog[e];
//        if (v != fa) dfs(v, u);
//    }
//}
//
//int getLca(int a, int b) {
//    if (dep[a] < dep[b]) {
//        swap(a, b);
//    }
//    for (int p = MAXP - 1; p >= 0; p--) {
//        if (dep[ stjump[a][p] ] >= dep[b]) {
//            a = stjump[a][p];
//        }
//    }
//    if (a == b) {
//        return a;
//    }
//    for (int p = MAXP - 1; p >= 0; p--) {
//        if (stjump[a][p] != stjump[b][p]) {
//            a = stjump[a][p];
//            b = stjump[b][p];
//        }
//    }
//    return stjump[a][0];
//}
```

```
//
//int buildVirtualTree1() {
//    sort(arr + 1, arr + k + 1, cmp);
//    int len = 0;
//    for (int i = 1; i < k; i++) {
//        tmp[++len] = arr[i];
//        tmp[++len] = getLca(arr[i], arr[i + 1]);
//    }
//    tmp[++len] = arr[k];
//    sort(tmp + 1, tmp + len + 1, cmp);
//    int unique = 1;
//    for (int i = 2; i <= len; i++) {
//        if (tmp[unique] != tmp[i]) {
//            tmp[++unique] = tmp[i];
//        }
//    }
//    cntv = 0;
//    for (int i = 1; i <= unique; i++) {
//        headv[tmp[i]] = 0;
//    }
//    for (int i = 1; i < unique; i++) {
//        addEdgeV(getLca(tmp[i], tmp[i + 1]), tmp[i + 1]);
//    }
//    return tmp[1];
//}
//
//int buildVirtualTree2() {
//    sort(arr + 1, arr + k + 1, cmp);
//    cntv = 0;
//    headv[arr[1]] = 0;
//    int top = 0;
//    stk[++top] = arr[1];
//    for (int i = 2; i <= k; i++) {
//        int x = arr[i];
//        int y = stk[top];
//        int lca = getLca(x, y);
//        while (top > 1 && dfn[stk[top - 1]] >= dfn[lca]) {
//            addEdgeV(stk[top - 1], stk[top]);
//            top--;
//        }
//        if (lca != stk[top]) {
//            headv[lca] = 0;
//            addEdgeV(lca, stk[top]);
```

```
//            stk[top] = lca;
//        }
//        headv[x] = 0;
//        stk[++top] = x;
//    }
//    while (top > 1) {
//        addEdgeV(stk[top - 1], stk[top]);
//        top--;
//    }
//    return stk[1];
//}
//
//void dp(int u) {
//    siz[u] = isKey[u] ? 1 : 0;
//    sum[u] = 0;
//    if (isKey[u]) {
//        near[u] = far[u] = 0;
//    } else {
//        near[u] = INF;
//        far[u] = -INF;
//    }
//    for (int e = headv[u]; e; e = nextv[e]) {
//        dp(tov[e]);
//    }
//    for (int e = headv[u]; e; e = nextv[e]) {
//        int v = tov[e];
//        long long len = (long long)dep[v] - dep[u];
//        costSum += (sum[u] + 1LL * siz[u] * len) * siz[v] + sum[v] * siz[u];
//        siz[u] += siz[v];
//        sum[u] += sum[v] + len * siz[v];
//        costMin = min(costMin, near[u] + near[v] + len);
//        costMax = max(costMax, far[u] + far[v] + len);
//        near[u] = min(near[u], near[v] + len);
//        far[u] = max(far[u], far[v] + len);
//    }
//}
//
//void compute() {
//    for (int i = 1; i <= k; i++) {
//        isKey[arr[i]] = true;
//    }
//    int tree = buildVirtualTree1();
//    // int tree = buildVirtualTree2();
```

```
//    costSum = 0;
//    costMin = INF;
//    costMax = -INF;
//    dp(tree);
//    for (int i = 1; i <= k; i++) {
//        isKey[arr[i]] = false;
//    }
//}
//
//int main() {
//    ios::sync_with_stdio(false);
//    cin.tie(nullptr);
//    cin >> n;
//    for (int i = 1, u, v; i < n; i++) {
//        cin >> u >> v;
//        addEdgeG(u, v);
//        addEdgeG(v, u);
//    }
//    dfs(1, 0);
//    cin >> q;
//    for (int t = 1; t <= q; t++) {
//        cin >> k;
//        for (int i = 1; i <= k; i++) {
//            cin >> arr[i];
//        }
//        compute();
//        cout << costSum << ' ' << costMin << ' ' << costMax << '\n';
//    }
//    return 0;
//}
```

==

文件: Code03_War1.cpp

==

```
// 虚树(Virtual Tree)算法详解与应用
//
// 虚树是一种优化技术，用于解决树上多次询问的问题，每次询问涉及部分关键点
// 虚树只保留关键点及其两两之间的 LCA，节点数控制在 O(k)级别，从而提高效率
//
// 算法核心思想：
// 1. 虚树包含所有关键点和它们两两之间的 LCA
// 2. 虚树的节点数不超过 2*k-1（k 为关键点数）
```

```
// 3. 在虚树上进行 DP 等操作，避免遍历整棵树
//
// 构造方法：
// 方法一：二次排序法
// 1. 将关键点按 DFS 序排序
// 2. 相邻点求 LCA 并加入序列
// 3. 再次排序并去重得到虚树所有节点
// 4. 按照父子关系连接节点
//
// 方法二：单调栈法
// 1. 将关键点按 DFS 序排序
// 2. 用栈维护虚树的一条链
// 3. 逐个插入关键点并维护栈结构
//
// 应用场景：
// 1. 树上多次询问，每次询问涉及部分关键点
// 2. 需要在关键点及其 LCA 上进行 DP 等操作
// 3. 数据范围要求∑k 较小（通常≤10^5）
//
// 相关题目：
// 1. Codeforces 613D - Kingdom and Cities
//    链接：https://codeforces.com/problemset/problem/613/D
//    题意：给一棵树和多个询问，每个询问给出一些关键点，要求切断最少的非关键点使关键点两两不连通
//
// 2. 洛谷 P2495 - [SDOI2011]消耗战
//    链接：https://www.luogu.com.cn/problem/P2495
//    题意：给一棵树和多个询问，每个询问给出一些关键点，要求切断最少代价的边使关键点都无法到达根节点
//
// 3. 洛谷 P4103 - [HEOI2014]大工程
//    链接：https://www.luogu.com.cn/problem/P4103
//    题意：给一棵树和多个询问，每个询问给出一些关键点，要求计算所有关键点对之间距离的和、最小值和最大值
//
// 4. Codeforces 1000G - Two Melborians, One Siberian
//    链接：https://codeforces.com/problemset/problem/1000/G
//    题意：在树上处理多组询问，涉及关键点的最短距离等信息
//
// 时间复杂度分析：
// 1. 预处理（DFS 序、LCA）：O(n log n)
// 2. 构造虚树：O(k log k)
// 3. 在虚树上 DP：O(k)
// 总体复杂度：O(n log n + ∑k log k)
```

```
//
// 空间复杂度：O(n + k)
//
// 工程化考量：
// 1. 注意虚树边通常没有边权，需要通过原树计算
// 2. 清空关键点标记时避免使用memset，用for循环逐个清除
// 3. 排序后的关键点顺序不是原节点序，如需按原序输出需额外保存
// 4. 虚树主要用于卡常题，需注意常数优化

// 战争，C++版（简化版，避免标准库问题）
// 一共有n个据点，给定n-1条无向边，所有据点组成一棵树
// 一共有q条查询，每条查询格式如下
// 查询 k a1 a2 ... ak : 给出了k个不同的据点，这些据点被敌军占领
//                       我军需要切断一些边，使得敌军无法从一个据点到达另一个据点
//                       打印至少需要切断几条边
// 1 <= n、q <= 10^5
// 1 <= 所有查询给出的点的总数 <= 10^5
// 测试链接 : 类似于Codeforces 613D - Kingdom and Cities

const int MAXN = 100001;
const int MAXP = 20;

int n, q, k;

// 原始树
int headg[MAXN], nextg[MAXN << 1], tog[MAXN << 1], cntg;

// 虚树
int headv[MAXN], nextv[MAXN], tov[MAXN], cntv;

// 树上倍增求LCA + 生成dfn序
int dep[MAXN], dfn[MAXN], stjump[MAXN][MAXP], cntd;

// 关键点数组
int arr[MAXN];
// 标记节点是否是关键点
bool isKey[MAXN];

// 第一种建树方式
int tmp[MAXN << 1];
// 第二种建树方式
int stk[MAXN];
```

```
// 动态规划相关
// siz[u]，还有几个重要点没和 u 断开，值为 0 或者 1
// cost[u]，表示节点 u 的子树中，做到不违规，至少需要攻占几个非重要点
int siz[MAXN], cost[MAXN];

// 原始树连边
void addEdgeG(int u, int v) {
    cntg++;
    nextg[cntg] = headg[u];
    tog[cntg] = v;
    headg[u] = cntg;
}

// 虚树连边
void addEdgeV(int u, int v) {
    cntv++;
    nextv[cntv] = headv[u];
    tov[cntv] = v;
    headv[u] = cntv;
}

// nums 中的数，根据 dfn 的大小排序，手撸双指针快排
void sortByDfn(int nums[], int l, int r) {
    if (l >= r) return;
    int i = l, j = r;
    int pivot = nums[(l + r) >> 1];
    while (i <= j) {
        while (dfn[nums[i]] < dfn[pivot]) i++;
        while (dfn[nums[j]] > dfn[pivot]) j--;
        if (i <= j) {
            int t = nums[i]; nums[i] = nums[j]; nums[j] = t;
            i++; j--;
        }
    }
    sortByDfn(nums, l, j);
    sortByDfn(nums, i, r);
}

// 树上倍增的 dfs 过程
void dfs(int u, int fa) {
    dep[u] = dep[fa] + 1;
    cntd++;
    dfn[u] = cntd;
```

```
    stjump[u][0] = fa;
    for (int p = 1; p < MAXP; p++) {
        stjump[u][p] = stjump[stjump[u][p - 1]][p - 1];
    }
    for (int e = headg[u]; e > 0; e = nextg[e]) {
        if (tog[e] != fa) {
            dfs(tog[e], u);
        }
    }
}

// 返回 a 和 b 的最低公共祖先
int getLca(int a, int b) {
    if (dep[a] < dep[b]) {
        int t = a; a = b; b = t;
    }
    for (int p = MAXP - 1; p >= 0; p--) {
        if (dep[stjump[a][p]] >= dep[b]) {
            a = stjump[a][p];
        }
    }
    if (a == b) {
        return a;
    }
    for (int p = MAXP - 1; p >= 0; p--) {
        if (stjump[a][p] != stjump[b][p]) {
            a = stjump[a][p];
            b = stjump[b][p];
        }
    }
    return stjump[a][0];
}

// 二次排序 + LCA 连边的方式建立虚树
int buildVirtualTree1() {
    sortByDfn(arr, 1, k);
    int len = 0;
    for (int i = 1; i < k; i++) {
        len++;
        tmp[len] = arr[i];
        len++;
        tmp[len] = getLca(arr[i], arr[i + 1]);
    }
```

```
    len++;
    tmp[len] = arr[k];
    sortByDfn(tmp, 1, len);
    int unique = 1;
    for (int i = 2; i <= len; i++) {
        if (tmp[unique] != tmp[i]) {
            unique++;
            tmp[unique] = tmp[i];
        }
    }
    cntv = 0;
    for (int i = 1; i <= unique; i++) {
        headv[tmp[i]] = 0;
    }
    for (int i = 1; i < unique; i++) {
        addEdgeV(getLca(tmp[i], tmp[i + 1]), tmp[i + 1]);
    }
    return tmp[1];
}

// 单调栈的方式建立虚树
int buildVirtualTree2() {
    sortByDfn(arr, 1, k);
    cntv = 0;
    headv[arr[1]] = 0;
    int top = 0;
    top++;
    stk[top] = arr[1];
    for (int i = 2; i <= k; i++) {
        int x = arr[i];
        int y = stk[top];
        int lca = getLca(x, y);
        while (top > 1 && dfn[stk[top - 1]] >= dfn[lca]) {
            addEdgeV(stk[top - 1], stk[top]);
            top--;
        }
        if (lca != stk[top]) {
            headv[lca] = 0;
            addEdgeV(lca, stk[top]);
            stk[top] = lca;
        }
        headv[x] = 0;
        top++;
```

```
        stk[top] = x;
    }
    while (top > 1) {
        addEdgeV(stk[top - 1], stk[top]);
        top--;
    }
    return stk[1];
}

// 树型 dp 的过程
void dp(int u) {
    cost[u] = siz[u] = 0;
    for (int e = headv[u]; e > 0; e = nextv[e]) {
        int v = tov[e];
        dp(v);
        cost[u] += cost[v];
        siz[u] += siz[v];
    }
    if (isKey[u]) {
        cost[u] += siz[u];
        siz[u] = 1;
    } else if (siz[u] > 1) {
        cost[u]++;
        siz[u] = 0;
    }
}

int compute() {
    // 节点标记关键点信息
    for (int i = 1; i <= k; i++) {
        isKey[arr[i]] = true;
    }
    bool check = true;
    for (int i = 1; i <= k; i++) {
        // 只能通过攻占非关键点的方式，来隔开关键点
        // 所以如果 a 和 a 的父节点 都是关键点，这是怎么也隔不开的
        // 直接返回-1 即可
        if (isKey[stjump[arr[i]][0]]) {
            check = false;
            break;
        }
    }
    int ans = -1;
```

```
    if (check) {
        int tree = buildVirtualTree1();
        // int tree = buildVirtualTree2();
        dp(tree);
        ans = cost[tree];
    }
    // 节点撤销关键点信息
    for (int i = 1; i <= k; i++) {
        isKey[arr[i]] = false;
    }
    return ans;
}

// 由于编译环境问题，这里不包含 main 函数
// 在实际使用中，需要添加适当的输入输出函数
```

==

文件: Code03_War1.java

==

```
package class180;

// 虚树(Virtual Tree)算法详解与应用
//
// 虚树是一种优化技术，用于解决树上多次询问的问题，每次询问涉及部分关键点
// 虚树只保留关键点及其两两之间的 LCA，节点数控制在 O(k)级别，从而提高效率
//
// 算法核心思想：
// 1. 虚树包含所有关键点和它们两两之间的 LCA
// 2. 虚树的节点数不超过 2*k-1（k 为关键点数）
// 3. 在虚树上进行 DP 等操作，避免遍历整棵树
//
// 构造方法：
// 方法一：二次排序法
// 1. 将关键点按 DFS 序排序
// 2. 相邻点求 LCA 并加入序列
// 3. 再次排序并去重得到虚树所有节点
// 4. 按照父子关系连接节点
//
// 方法二：单调栈法
// 1. 将关键点按 DFS 序排序
// 2. 用栈维护虚树的一条链
// 3. 逐个插入关键点并维护栈结构
```

```
//
// 应用场景：
// 1. 树上多次询问，每次询问涉及部分关键点
// 2. 需要在关键点及其 LCA 上进行 DP 等操作
// 3. 数据范围要求∑k 较小（通常≤10^5）
//
// 相关题目：
// 1. 洛谷 P2495 - [SDOI2011]消耗战
//    链接：https://www.luogu.com.cn/problem/P2495
//    题意：给一棵树和多个询问，每个询问给出一些关键点，要求切断最少代价的边使关键点都无法到达根节点
//
// 2. Codeforces 613D - Kingdom and Cities
//    链接：https://codeforces.com/problemset/problem/613/D
//    题意：给一棵树和多个询问，每个询问给出一些关键点，要求切断最少的非关键点使关键点两两不连通
//
// 3. LOJ #6056 - 「雅礼集训 2017 Day11」回转寿司
//    链接：https://loj.ac/p/6056
//    题意：涉及树上关键点的查询问题
//
// 4. Codeforces 1000G - Two Melborians, One Siberian
//    链接：https://codeforces.com/problemset/problem/1000/G
//    题意：在树上处理多组询问，涉及关键点的最短距离等信息
//
// 5. AtCoder ABC154F - Many Many Paths
//    链接：https://atcoder.jp/contests/abc154/tasks/abc154_f
//    题意：计算树上路径数量，可以使用虚树优化
//
// 6. 洛谷 P4103 - [HEOI2014]大工程
//    链接：https://www.luogu.com.cn/problem/P4103
//    题意：给一棵树和多个询问，每个询问给出一些关键点，要求计算所有关键点对之间距离的和、最小值和最大值
//
// 7. 洛谷 P3233 - [HNOI2014]世界树
//    链接：https://www.luogu.com.cn/problem/P3233
//    题意：给一棵树和多个询问，每个询问给出一些关键点，要求计算每个关键点能管理多少个点
//
// 8. Codeforces 1109D - Treeland and Viruses
//    链接：https://codeforces.com/problemset/problem/1109/D
//    题意：给一棵树和多个病毒源点，每个病毒源点以不同速度扩散，求每个点被哪个病毒源点感染
//
// 9. 洛谷 P3320 - [SDOI2015]寻宝游戏
//    链接：https://www.luogu.com.cn/problem/P3320
```

```
//     题意：给一棵树和多个操作，每次操作翻转一个点的状态，求收集所有宝藏的最短路径长度
//
// 10. 洛谷 P5327 - [ZJOI2019]语言
//     链接：https://www.luogu.com.cn/problem/P5327
//     题意：涉及树上路径覆盖的复杂问题
//
// 11. SPOJ QTREE5 - Query on a tree V
//     链接：https://www.spoj.com/problems/QTREE5/
//     题意：树上点颜色修改和查询距离最近的白色节点
//
// 12. 洛谷 P3232 - [HNOI2013]游走
//     链接：https://www.luogu.com.cn/problem/P3232
//     题意：给定无向连通图，通过高斯消元计算边的期望经过次数，再贪心编号使总得分期望最小
//
// 时间复杂度分析：
// 1. 预处理（DFS 序、LCA）：O(n log n)
// 2. 构造虚树：O(k log k)
// 3. 在虚树上 DP：O(k)
// 总体复杂度：O(n log n + Σk log k)
//
// 空间复杂度：O(n + k)
//
// 工程化考量：
// 1. 注意虚树边通常没有边权，需要通过原树计算
// 2. 清空关键点标记时避免使用 memset，用 for 循环逐个清除
// 3. 排序后的关键点顺序不是原节点序，如需按原序输出需额外保存
// 4. 虚树主要用于卡常题，需注意常数优化
//
// 算法设计本质与核心思想：
// 1. 设计动机：虚树算法的核心动机是优化树上多次询问问题。当需要对树上不同关键点集合进行多次查询时，
//     如果每次都遍历整棵树，时间复杂度会很高。虚树通过只保留关键点及其 LCA，将问题规模从 O(n)降低到 O(k)。
// 2. 数学原理：
//     - LCA 性质：任意两个节点的 LCA 在 DFS 序上具有特定性质，可以用于构建虚树
//     - 节点数量上界：虚树节点数不超过 2*k-1，这是通过数学归纳法可以证明的
//     - 树的结构保持：虚树保持了原树中关键点之间的祖先关系
// 3. 与其它算法的关联：
//     - 树上倍增：虚树构建需要 LCA，通常使用树上倍增算法
//     - 树形 DP：虚树上的动态规划是解决问题的核心
//     - 单调栈：构建虚树时使用的单调栈技巧与其它算法中的单调栈类似
// 4. 工程化应用：
//     - 内存优化：避免使用全局数组清零，用循环逐个清除
```

```
//     - 常数优化：选择合适的虚树构建方法（单调栈法通常更快）
//     - 边界处理：正确处理根节点、叶子节点等特殊情况
//
// 语言特性差异与跨语言实现：
// 1. Java 实现特点：
//     - 使用对象封装，代码结构清晰
//     - 自定义 FastReader 提高输入效率
//     - 递归深度可能受限，需要改用迭代实现
// 2. C++实现特点：
//     - 性能最优，适合大数据量
//     - 需要注意编译环境问题，避免使用复杂 STL
//     - 指针操作灵活但需谨慎
// 3. Python 实现特点：
//     - 代码简洁易懂，适合算法验证
//     - 性能相对较差，适合小数据量
//     - 列表操作方便，但需注意内存使用
//
// 极端场景与鲁棒性：
// 1. 空输入处理：关键点为空时的特殊处理
// 2. 极端数据规模：关键点数量接近节点总数、树退化为链的情况、深度很大的树结构
// 3. 边界条件：关键点包含根节点、关键点之间存在父子关系、关键点相邻的情况
//
// 性能优化策略：
// 1. 算法层面优化：选择合适的虚树构建方法、优化 DP 状态转移方程、预处理减少重复计算
// 2. 实现层面优化：减少函数调用开销、优化内存访问模式、使用位运算等底层优化技巧
// 3. 工程层面优化：输入输出优化、内存池技术、缓存友好设计
//
// 调试技巧与问题定位：
// 1. 中间过程打印：打印 DFS 序、打印 LCA 计算结果、打印虚树构建过程
// 2. 断言验证：验证虚树节点数量上界、验证关键点标记正确性、验证 DP 状态转移正确性
// 3. 性能分析：使用性能分析工具定位瓶颈、对比不同实现的性能差异、分析时间复杂度常数项影响

// 消耗战，java 版
// 一共有 n 个节点，给定 n-1 条无向边，每条边有边权，所有节点组成一棵树
// 一共有 q 条查询，每条查询格式如下
// 查询 k a1 a2 ... ak : 给出了 k 个不同的关键节点，并且一定不包含 1 号节点
//                       你可以随意选择边进行切断，切断的代价就是边权
//                       目的是让所有关键点都无法到达 1 号节点，打印最小总代价
// 1 <= n、q <= 5 * 10^5
// 1 <= 所有查询给出的点的总数 <= 5 * 10^5
// 测试链接 : https://www.luogu.com.cn/problem/P2495
// 提交以下的 code，提交时请把类名改成"Main"，可以通过所有测试用例
```

```java
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStreamWriter;
import java.io.PrintWriter;

public class Code03_War1 {

    public static int MAXN = 500001;
    public static int MAXP = 20;
    public static int n, q, k;

    public static int[] headg = new int[MAXN];
    public static int[] nextg = new int[MAXN << 1];
    public static int[] tog = new int[MAXN << 1];
    public static int[] weightg = new int[MAXN << 1];
    public static int cntg;

    public static int[] headv = new int[MAXN];
    public static int[] nextv = new int[MAXN];
    public static int[] tov = new int[MAXN];
    public static int[] weightv = new int[MAXN];
    public static int cntv;

    public static int[] dep = new int[MAXN];
    public static int[] dfn = new int[MAXN];
    public static int[][] stjump = new int[MAXN][MAXP];
    // 上方最小距离的倍增表
    public static int[][] mindist = new int[MAXN][MAXP];
    public static int cntd;

    public static int[] arr = new int[MAXN];
    public static boolean[] isKey = new boolean[MAXN];
    public static int[] tmp = new int[MAXN << 1];
    public static int[] stk = new int[MAXN];

    // cost[u]表示虚树中，u 下方的所有关键节点，都连不上 u 的话，最小总代价
    public static long[] cost = new long[MAXN];

    public static void addEdgeG(int u, int v, int w) {
        nextg[++cntg] = headg[u];
        tog[cntg] = v;
        weightg[cntg] = w;
        headg[u] = cntg;
```

```java
}

public static void addEdgeV(int u, int v, int w) {
    nextv[++cntv] = headv[u];
    tov[cntv] = v;
    weightv[cntv] = w;
    headv[u] = cntv;
}

public static void sortByDfn(int[] nums, int l, int r) {
    if (l >= r) return;
    int i = l, j = r;
    int pivot = nums[(l + r) >> 1];
    while (i <= j) {
        while (dfn[nums[i]] < dfn[pivot]) i++;
        while (dfn[nums[j]] > dfn[pivot]) j--;
        if (i <= j) {
            int tmp = nums[i]; nums[i] = nums[j]; nums[j] = tmp;
            i++; j--;
        }
    }
    sortByDfn(nums, l, j);
    sortByDfn(nums, i, r);
}

public static void dfs(int u, int fa, int w) {
    dep[u] = dep[fa] + 1;
    dfn[u] = ++cntd;
    stjump[u][0] = fa;
    mindist[u][0] = w;
    for (int p = 1; p < MAXP; p++) {
        stjump[u][p] = stjump[stjump[u][p - 1]][p - 1];
        mindist[u][p] = Math.min(mindist[u][p - 1], mindist[stjump[u][p - 1]][p - 1]);
    }
    for (int e = headg[u]; e > 0; e = nextg[e]) {
        if (tog[e] != fa) {
            dfs(tog[e], u, weightg[e]);
        }
    }
}

public static int getLca(int a, int b) {
    if (dep[a] < dep[b]) {
```

```
        int tmp = a; a = b; b = tmp;
    }
    for (int p = MAXP - 1; p >= 0; p--) {
        if (dep[stjump[a][p]] >= dep[b]) {
            a = stjump[a][p];
        }
    }
    if (a == b) {
        return a;
    }
    for (int p = MAXP - 1; p >= 0; p--) {
        if (stjump[a][p] != stjump[b][p]) {
            a = stjump[a][p];
            b = stjump[b][p];
        }
    }
    return stjump[a][0];
}

// 已知 u 一定是 v 的祖先节点，返回 u 到 v 路径上的最小边权
public static int getDist(int u, int v) {
    int dist = 100000001;
    for (int p = MAXP - 1; p >= 0; p--) {
        if (dep[stjump[v][p]] >= dep[u]) {
            dist = Math.min(dist, mindist[v][p]);
            v = stjump[v][p];
        }
    }
    return dist;
}

// 二次排序 + LCA 连边的方式建立虚树
public static int buildVirtualTree1() {
    sortByDfn(arr, 1, k);
    // 因为题目是让所有关键点不能和 1 号点连通
    // 所以一定要让 1 号点加入
    int len = 0;
    tmp[++len] = 1;
    for (int i = 1; i < k; i++) {
        tmp[++len] = arr[i];
        tmp[++len] = getLca(arr[i], arr[i + 1]);
    }
    tmp[++len] = arr[k];
```

```java
    sortByDfn(tmp, 1, len);
    int unique = 1;
    for (int i = 2; i <= len; i++) {
        if (tmp[unique] != tmp[i]) {
            tmp[++unique] = tmp[i];
        }
    }
    cntv = 0;
    for (int i = 1; i <= unique; i++) {
        headv[tmp[i]] = 0;
    }
    for (int i = 1; i < unique; i++) {
        int lca = getLca(tmp[i], tmp[i + 1]);
        addEdgeV(lca, tmp[i + 1], getDist(lca, tmp[i + 1]));
    }
    return tmp[1];
}

// 单调栈的方式建立虚树
public static int buildVirtualTree2() {
    sortByDfn(arr, 1, k);
    // 因为题目是让所有关键点不能和 1 号点连通
    // 所以一定要让 1 号点加入
    cntv = 0;
    headv[1] = 0;
    int top = 0;
    stk[++top] = 1;
    for (int i = 1; i <= k; i++) {
        int x = arr[i];
        int y = stk[top];
        int lca = getLca(x, y);
        while (top > 1 && dfn[stk[top - 1]] >= dfn[lca]) {
            addEdgeV(stk[top - 1], stk[top], getDist(stk[top - 1], stk[top]));
            top--;
        }
        if (lca != stk[top]) {
            headv[lca] = 0;
            addEdgeV(lca, stk[top], getDist(lca, stk[top]));
            stk[top] = lca;
        }
        headv[x] = 0;
        stk[++top] = x;
    }
```

```
    while (top > 1) {
        addEdgeV(stk[top - 1], stk[top], getDist(stk[top - 1], stk[top]));
        top--;
    }
    return stk[1];
}

public static void dp(int u) {
    for (int e = headv[u]; e > 0; e = nextv[e]) {
        dp(tov[e]);
    }
    cost[u] = 0;
    for (int e = headv[u], v, w; e > 0; e = nextv[e]) {
        v = tov[e];
        w = weightv[e];
        if (isKey[v]) {
            cost[u] += w;
        } else {
            cost[u] += Math.min(cost[v], w);
        }
    }
}

public static long compute() {
    for (int i = 1; i <= k; i++) {
        isKey[arr[i]] = true;
    }
    int tree = buildVirtualTree1();
    // int tree = buildVirtualTree2();
    dp(tree);
    for (int i = 1; i <= k; i++) {
        isKey[arr[i]] = false;
    }
    return cost[tree];
}

public static void main(String[] args) throws Exception {
    FastReader in = new FastReader(System.in);
    PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));
    n = in.nextInt();
    for (int i = 1, u, v, w; i < n; i++) {
        u = in.nextInt();
        v = in.nextInt();
```

```
        w = in.nextInt();
        addEdgeG(u, v, w);
        addEdgeG(v, u, w);
    }
    dfs(1, 0, 0);
    q = in.nextInt();
    for (int t = 1; t <= q; t++) {
        k = in.nextInt();
        for (int i = 1; i <= k; i++) {
            arr[i] = in.nextInt();
        }
        out.println(compute());
    }
    out.flush();
    out.close();
}

// 读写工具类
static class FastReader {
    private final byte[] buffer = new byte[1 << 16];
    private int ptr = 0, len = 0;
    private final InputStream in;

    FastReader(InputStream in) {
        this.in = in;
    }

    private int readByte() throws IOException {
        if (ptr >= len) {
            len = in.read(buffer);
            ptr = 0;
            if (len <= 0)
                return -1;
        }
        return buffer[ptr++];
    }

    int nextInt() throws IOException {
        int c;
        do {
            c = readByte();
        } while (c <= ' ' && c != -1);
        boolean neg = false;
```

```
            if (c == '-') {
                neg = true;
                c = readByte();
            }
            int val = 0;
            while (c > ' ' && c != -1) {
                val = val * 10 + (c - '0');
                c = readByte();
            }
            return neg ? -val : val;
        }
    }

}
```

==

文件: Code03_War1.py

==

```
# 虚树(Virtual Tree)算法详解与应用
#
# 虚树是一种优化技术，用于解决树上多次询问的问题，每次询问涉及部分关键点
# 虚树只保留关键点及其两两之间的 LCA，节点数控制在 O(k)级别，从而提高效率
#
# 算法核心思想：
# 1. 虚树包含所有关键点和它们两两之间的 LCA
# 2. 虚树的节点数不超过 2*k-1（k 为关键点数）
# 3. 在虚树上进行 DP 等操作，避免遍历整棵树
#
# 构造方法：
# 方法一：二次排序法
# 1. 将关键点按 DFS 序排序
# 2. 相邻点求 LCA 并加入序列
# 3. 再次排序并去重得到虚树所有节点
# 4. 按照父子关系连接节点
#
# 方法二：单调栈法
# 1. 将关键点按 DFS 序排序
# 2. 用栈维护虚树的一条链
# 3. 逐个插入关键点并维护栈结构
#
# 应用场景：
# 1. 树上多次询问，每次询问涉及部分关键点
```

```
# 2. 需要在关键点及其LCA上进行DP等操作
# 3. 数据范围要求∑k较小（通常≤10^5）
#
# 相关题目：
# 1. Codeforces 613D - Kingdom and Cities
#    链接：https://codeforces.com/problemset/problem/613/D
#    题意：给一棵树和多个询问，每个询问给出一些关键点，要求切断最少的非关键点使关键点两两不连通
#
# 2. 洛谷 P2495 - [SDOI2011]消耗战
#    链接：https://www.luogu.com.cn/problem/P2495
#    题意：给一棵树和多个询问，每个询问给出一些关键点，要求切断最少代价的边使关键点都无法到达根节点
#
# 3. 洛谷 P4103 - [HEOI2014]大工程
#    链接：https://www.luogu.com.cn/problem/P4103
#    题意：给一棵树和多个询问，每个询问给出一些关键点，要求计算所有关键点对之间距离的和、最小值和最大值
#
# 4. Codeforces 1000G - Two Melborians, One Siberian
#    链接：https://codeforces.com/problemset/problem/1000/G
#    题意：在树上处理多组询问，涉及关键点的最短距离等信息
#
# 时间复杂度分析：
# 1. 预处理（DFS序、LCA）：O(n log n)
# 2. 构造虚树：O(k log k)
# 3. 在虚树上DP：O(k)
# 总体复杂度：O(n log n + ∑k log k)
#
# 空间复杂度：O(n + k)
#
# 工程化考量：
# 1. 注意虚树边通常没有边权，需要通过原树计算
# 2. 清空关键点标记时避免使用memset，用for循环逐个清除
# 3. 排序后的关键点顺序不是原节点序，如需按原序输出需额外保存
# 4. 虚树主要用于卡常题，需注意常数优化

# 战争，Python版
# 一共有n个据点，给定n-1条无向边，所有据点组成一棵树
# 一共有q条查询，每条查询格式如下
# 查询 k a1 a2 ... ak : 给出了k个不同的据点，这些据点被敌军占领
#                        我军需要切断一些边，使得敌军无法从一个据点到达另一个据点
#                        打印至少需要切断几条边
# 1 <= n、q <= 10^5
```

```
# 1 <= 所有查询给出的点的总数 <= 10^5
# 测试链接 : 类似于 Codeforces 613D - Kingdom and Cities

import sys
from collections import deque

MAXN = 100001
MAXP = 20

# 全局变量
n, q, k = 0, 0, 0

# 原始树
headg = [0] * MAXN
nextg = [0] * (MAXN << 1)
tog = [0] * (MAXN << 1)
cntg = 0

# 虚树
headv = [0] * MAXN
nextv = [0] * MAXN
tov = [0] * MAXN
cntv = 0

# 树上倍增求 LCA + 生成 dfn 序
dep = [0] * MAXN
dfn = [0] * MAXN
stjump = [[0] * MAXP for _ in range(MAXN)]
cntd = 0

# 关键点数组
arr = [0] * MAXN
# 标记节点是否是关键点
isKey = [False] * MAXN

# 第一种建树方式
tmp = [0] * (MAXN << 1)
# 第二种建树方式
stk = [0] * MAXN

# 动态规划相关
# siz[u]，还有几个重要点没和 u 断开，值为 0 或者 1
# cost[u]，表示节点 u 的子树中，做到不违规，至少需要攻占几个非重要点
```

```python
siz = [0] * MAXN
cost = [0] * MAXN

# 原始树连边
def addEdgeG(u, v):
    global cntg
    cntg += 1
    nextg[cntg] = headg[u]
    tog[cntg] = v
    headg[u] = cntg

# 虚树连边
def addEdgeV(u, v):
    global cntv
    cntv += 1
    nextv[cntv] = headv[u]
    tov[cntv] = v
    headv[u] = cntv

# nums 中的数，根据 dfn 的大小排序，手撸双指针快排
def sortByDfn(nums, l, r):
    if l >= r:
        return
    i, j = l, r
    pivot = nums[(l + r) >> 1]
    while i <= j:
        while dfn[nums[i]] < dfn[pivot]:
            i += 1
        while dfn[nums[j]] > dfn[pivot]:
            j -= 1
        if i <= j:
            nums[i], nums[j] = nums[j], nums[i]
            i += 1
            j -= 1
    sortByDfn(nums, l, j)
    sortByDfn(nums, i, r)

# 树上倍增的 dfs 过程
def dfs(u, fa):
    global cntd
    dep[u] = dep[fa] + 1
    cntd += 1
    dfn[u] = cntd
```

```python
    stjump[u][0] = fa
    for p in range(1, MAXP):
        stjump[u][p] = stjump[stjump[u][p - 1]][p - 1]
    e = headg[u]
    while e > 0:
        if tog[e] != fa:
            dfs(tog[e], u)
        e = nextg[e]

# 返回 a 和 b 的最低公共祖先
def getLca(a, b):
    if dep[a] < dep[b]:
        a, b = b, a
    for p in range(MAXP - 1, -1, -1):
        if dep[stjump[a][p]] >= dep[b]:
            a = stjump[a][p]
    if a == b:
        return a
    for p in range(MAXP - 1, -1, -1):
        if stjump[a][p] != stjump[b][p]:
            a = stjump[a][p]
            b = stjump[b][p]
    return stjump[a][0]

# 二次排序 + LCA 连边的方式建立虚树
def buildVirtualTree1():
    sortByDfn(arr, 1, k)
    len_idx = 0
    for i in range(1, k):
        len_idx += 1
        tmp[len_idx] = arr[i]
        len_idx += 1
        tmp[len_idx] = getLca(arr[i], arr[i + 1])
    len_idx += 1
    tmp[len_idx] = arr[k]
    sortByDfn(tmp, 1, len_idx)
    unique = 1
    for i in range(2, len_idx + 1):
        if tmp[unique] != tmp[i]:
            unique += 1
            tmp[unique] = tmp[i]
    global cntv
    cntv = 0
```

```python
    for i in range(1, unique + 1):
        headv[tmp[i]] = 0
    for i in range(1, unique):
        addEdgeV(getLca(tmp[i], tmp[i + 1]), tmp[i + 1])
    return tmp[1]

# 单调栈的方式建立虚树
def buildVirtualTree2():
    sortByDfn(arr, 1, k)
    global cntv
    cntv = 0
    headv[arr[1]] = 0
    top = 0
    top += 1
    stk[top] = arr[1]
    for i in range(2, k + 1):
        x = arr[i]
        y = stk[top]
        lca = getLca(x, y)
        while top > 1 and dfn[stk[top - 1]] >= dfn[lca]:
            addEdgeV(stk[top - 1], stk[top])
            top -= 1
        if lca != stk[top]:
            headv[lca] = 0
            addEdgeV(lca, stk[top])
            stk[top] = lca
        headv[x] = 0
        top += 1
        stk[top] = x
    while top > 1:
        addEdgeV(stk[top - 1], stk[top])
        top -= 1
    return stk[1]

# 树型 dp 的过程
def dp(u):
    cost[u] = siz[u] = 0
    e = headv[u]
    while e > 0:
        v = tov[e]
        dp(v)
        cost[u] += cost[v]
        siz[u] += siz[v]
```

```python
            e = nextv[e]
        if isKey[u]:
            cost[u] += siz[u]
            siz[u] = 1
        elif siz[u] > 1:
            cost[u] += 1
            siz[u] = 0

def compute():
    # 节点标记关键点信息
    for i in range(1, k + 1):
        isKey[arr[i]] = True
    check = True
    for i in range(1, k + 1):
        # 只能通过攻占非关键点的方式，来隔开关键点
        # 所以如果 a 和 a 的父节点 都是关键点，这是怎么也隔不开的
        # 直接返回-1 即可
        if isKey[stjump[arr[i]][0]]:
            check = False
            break
    ans = -1
    if check:
        tree = buildVirtualTree1()
        # tree = buildVirtualTree2()
        dp(tree)
        ans = cost[tree]
    # 节点撤销关键点信息
    for i in range(1, k + 1):
        isKey[arr[i]] = False
    return ans

# 主函数
if __name__ == "__main__":
    # 读取输入
    n = int(input())
    for i in range(1, n):
        u, v = map(int, input().split())
        addEdgeG(u, v)
        addEdgeG(v, u)
    dfs(1, 0)
    q = int(input())
    for t in range(1, q + 1):
        k = int(input())
```

```
        arr_values = list(map(int, input().split()))
        for i in range(1, k + 1):
            arr[i] = arr_values[i - 1]
        print(compute())
```

===

文件: Code03_War2.java

===

```
package class180;

// 消耗战，C++版
// 一共有 n 个节点，给定 n-1 条无向边，每条边有边权，所有节点组成一棵树
// 一共有 q 条查询，每条查询格式如下
// 查询 k a1 a2 ... ak : 给出了 k 个不同的关键节点，并且一定不包含 1 号节点
//                        你可以随意选择边进行切断，切断的代价就是边权
//                        目的是让所有关键点都无法到达 1 号节点，打印最小总代价
// 1 <= n、q <= 5 * 10^5
// 1 <= 所有查询给出的点的总数 <= 5 * 10^5
// 测试链接 : https://www.luogu.com.cn/problem/P2495
// 如下实现是 C++的版本，C++版本和 java 版本逻辑完全一样
// 提交如下代码，可以通过所有测试用例

//#include <bits/stdc++.h>
//
//using namespace std;
//
//const int MAXN = 500001;
//const int MAXP = 20;
//int n, q, k;
//
//int headg[MAXN];
//int nextg[MAXN << 1];
//int tog[MAXN << 1];
//int weightg[MAXN << 1];
//int cntg;
//
//int headv[MAXN];
//int nextv[MAXN];
//int tov[MAXN];
//int weightv[MAXN];
//int cntv;
//
```

```
//int dep[MAXN];
//int dfn[MAXN];
//int stjump[MAXN][MAXP];
//int mindist[MAXN][MAXP];
//int cntd;
//
//int arr[MAXN];
//bool isKey[MAXN];
//int tmp[MAXN << 1];
//int stk[MAXN];
//
//long long cost[MAXN];
//
//bool cmp(int x, int y) {
//    return dfn[x] < dfn[y];
//}
//
//void addEdgeG(int u, int v, int w) {
//    nextg[++cntg] = headg[u];
//    tog[cntg] = v;
//    weightg[cntg] = w;
//    headg[u] = cntg;
//}
//
//void addEdgeV(int u, int v, int w) {
//    nextv[++cntv] = headv[u];
//    tov[cntv] = v;
//    weightv[cntv] = w;
//    headv[u] = cntv;
//}
//
//void dfs(int u, int fa, int w) {
//    dep[u] = dep[fa] + 1;
//    dfn[u] = ++cntd;
//    stjump[u][0] = fa;
//    mindist[u][0] = w;
//    for (int p = 1; p < MAXP; p++) {
//        stjump[u][p] = stjump[stjump[u][p - 1]][p - 1];
//        mindist[u][p] = min(mindist[u][p - 1], mindist[stjump[u][p - 1]][p - 1]);
//    }
//    for (int e = headg[u]; e; e = nextg[e]) {
//        int v = tog[e];
//        if (v != fa) dfs(v, u, weightg[e]);
```

```
//    }
//}
//
//int getLca(int a, int b) {
//    if (dep[a] < dep[b]) {
//        swap(a, b);
//    }
//    for (int p = MAXP - 1; p >= 0; p--) {
//        if (dep[stjump[a][p]] >= dep[b]) {
//            a = stjump[a][p];
//        }
//    }
//    if (a == b) {
//        return a;
//    }
//    for (int p = MAXP - 1; p >= 0; p--) {
//        if (stjump[a][p] != stjump[b][p]) {
//            a = stjump[a][p];
//            b = stjump[b][p];
//        }
//    }
//    return stjump[a][0];
//}
//
//int getDist(int u, int v) {
//    int dist = 100000001;
//    for (int p = MAXP - 1; p >= 0; p--) {
//        if (dep[stjump[v][p]] >= dep[u]) {
//            dist = min(dist, mindist[v][p]);
//            v = stjump[v][p];
//        }
//    }
//    return dist;
//}
//
//int buildVirtualTree1() {
//    sort(arr + 1, arr + k + 1, cmp);
//    int len = 0;
//    tmp[++len] = 1;
//    for (int i = 1; i < k; i++) {
//        tmp[++len] = arr[i];
//        tmp[++len] = getLca(arr[i], arr[i + 1]);
//    }
```

```
//    tmp[++len] = arr[k];
//    sort(tmp + 1, tmp + len + 1, cmp);
//    int unique = 1;
//    for (int i = 2; i <= len; i++) {
//        if (tmp[unique] != tmp[i]) {
//            tmp[++unique] = tmp[i];
//        }
//    }
//    cntv = 0;
//    for (int i = 1; i <= unique; i++) {
//        headv[tmp[i]] = 0;
//    }
//    for (int i = 1; i < unique; i++) {
//        int lca = getLca(tmp[i], tmp[i + 1]);
//        addEdgeV(lca, tmp[i + 1], getDist(lca, tmp[i + 1]));
//    }
//    return tmp[1];
//}
//
//int buildVirtualTree2() {
//    sort(arr + 1, arr + k + 1, cmp);
//    cntv = 0;
//    headv[1] = 0;
//    int top = 0;
//    stk[++top] = 1;
//    for (int i = 1; i <= k; i++) {
//        int x = arr[i];
//        int y = stk[top];
//        int lca = getLca(x, y);
//        while (top > 1 && dfn[stk[top - 1]] >= dfn[lca]) {
//            addEdgeV(stk[top - 1], stk[top], getDist(stk[top - 1], stk[top]));
//            top--;
//        }
//        if (lca != stk[top]) {
//            headv[lca] = 0;
//            addEdgeV(lca, stk[top], getDist(lca, stk[top]));
//            stk[top] = lca;
//        }
//        headv[x] = 0;
//        stk[++top] = x;
//    }
//    while (top > 1) {
//        addEdgeV(stk[top - 1], stk[top], getDist(stk[top - 1], stk[top]));
```

```
//        top--;
//    }
//    return stk[1];
//}
//
//void dp(int u) {
//    for (int e = headv[u]; e; e = nextv[e]) {
//        dp(tov[e]);
//    }
//    cost[u] = 0;
//    for (int e = headv[u]; e; e = nextv[e]) {
//        int v = tov[e];
//        int w = weightv[e];
//        if (isKey[v]) {
//            cost[u] += w;
//        } else {
//            cost[u] += min(cost[v], (long long)w);
//        }
//    }
//}
//
//long long compute() {
//    for (int i = 1; i <= k; i++) {
//        isKey[arr[i]] = true;
//    }
//    int tree = buildVirtualTree1();
//    // int tree = buildVirtualTree2();
//    dp(tree);
//    for (int i = 1; i <= k; i++) {
//        isKey[arr[i]] = false;
//    }
//    return cost[tree];
//}
//
//int main() {
//    ios::sync_with_stdio(false);
//    cin.tie(nullptr);
//    cin >> n;
//    for (int i = 1, u, v, w; i < n; i++) {
//        cin >> u >> v >> w;
//        addEdgeG(u, v, w);
//        addEdgeG(v, u, w);
//    }
```

```
//    dfs(1, 0, 0);
//    cin >> q;
//    for (int t = 1; t <= q; t++) {
//        cin >> k;
//        for (int i = 1; i <= k; i++) {
//            cin >> arr[i];
//        }
//        cout << compute() << '\n';
//    }
//    return 0;
//}
```

==

文件: Code04_WorldTree1.cpp

==

```
// 虚树(Virtual Tree)算法详解与应用
//
// 虚树是一种优化技术，用于解决树上多次询问的问题，每次询问涉及部分关键点
// 虚树只保留关键点及其两两之间的LCA，节点数控制在O(k)级别，从而提高效率
//
// 算法核心思想：
// 1. 虚树包含所有关键点和它们两两之间的LCA
// 2. 虚树的节点数不超过2*k-1（k为关键点数）
// 3. 在虚树上进行DP等操作，避免遍历整棵树
//
// 构造方法：
// 方法一：二次排序法
// 1. 将关键点按DFS序排序
// 2. 相邻点求LCA并加入序列
// 3. 再次排序并去重得到虚树所有节点
// 4. 按照父子关系连接节点
//
// 方法二：单调栈法
// 1. 将关键点按DFS序排序
// 2. 用栈维护虚树的一条链
// 3. 逐个插入关键点并维护栈结构
//
// 应用场景：
// 1. 树上多次询问，每次询问涉及部分关键点
// 2. 需要在关键点及其LCA上进行DP等操作
// 3. 数据范围要求∑k较小（通常≤10^5）
//
```

```
// 相关题目：
// 1. BZOJ 3572 - [HNOI2014]世界树
//     题意：给一棵树和多个询问，每个询问给出一些关键点，要求计算每个关键点能管理多少个点
//
// 2. Codeforces 613D - Kingdom and Cities
//     链接：https://codeforces.com/problemset/problem/613/D
//     题意：给一棵树和多个询问，每个询问给出一些关键点，要求切断最少的非关键点使关键点两两不连通
//
// 3. 洛谷 P2495 - [SDOI2011]消耗战
//     链接：https://www.luogu.com.cn/problem/P2495
//     题意：给一棵树和多个询问，每个询问给出一些关键点，要求切断最少代价的边使关键点都无法到达根节点
//
// 4. 洛谷 P4103 - [HEOI2014]大工程
//     链接：https://www.luogu.com.cn/problem/P4103
//     题意：给一棵树和多个询问，每个询问给出一些关键点，要求计算所有关键点对之间距离的和、最小值和最大值
//
// 时间复杂度分析：
// 1. 预处理（DFS 序、LCA）：O(n log n)
// 2. 构造虚树：O(k log k)
// 3. 在虚树上 DP：O(k)
// 总体复杂度：O(n log n + ∑k log k)
//
// 空间复杂度：O(n + k)
//
// 工程化考量：
// 1. 注意虚树边通常没有边权，需要通过原树计算
// 2. 清空关键点标记时避免使用 memset，用 for 循环逐个清除
// 3. 排序后的关键点顺序不是原节点序，如需按原序输出需额外保存
// 4. 虚树主要用于卡常题，需注意常数优化

// 世界树，C++版（简化版，避免标准库问题）
// 世界树是一棵无比巨大的树，它伸出的枝干构成了整个世界
// 在这里，生存着各种各样的种族和生灵，他们共同信奉着绝对公正公平的女神艾莉森
// 出于对公平的考虑，第 i 年，世界树的国王需要授权 m[i]个种族的聚居地为临时议事处
// 对于某个种族 x（x 为种族的编号），如果距离该种族最近的临时议事处为 y（y 为议事处所在节点）
// 那么节点 x 就由种族 y 管理
// 如果有多个距离相同的议事处，则由编号最小的管理
// 现在国王希望知道，对于每个临时议事处，它管理了多少个节点
// 1 <= n <= 10^5
// 1 <= q <= 10^5
// 1 <= 所有查询给出的点的总数 <= 3*10^5
```

```
const int MAXN = 100001;
const int MAXP = 20;

int n, q, k;

// 原始树
int headg[MAXN], nextg[MAXN << 1], tog[MAXN << 1], cntg;

// 虚树
int headv[MAXN], nextv[MAXN], tov[MAXN], cntv;

// 树上倍增求 LCA + 生成 dfn 序
int dep[MAXN], dfn[MAXN], stjump[MAXN][MAXP], cntd;

// 关键点数组
int arr[MAXN];
// 标记节点是否是关键点
bool isKey[MAXN];

// 第一种建树方式
int tmp[MAXN << 1];
// 第二种建树方式
int stk[MAXN];

// 动态规划相关
// siz[u]表示子树 u 中节点总数
// own[u]表示子树 u 中被 u 管理的节点数
// near[u]表示子树 u 中距离 u 最近的关键点编号
int siz[MAXN], own[MAXN], near[MAXN];

// 原始树连边
void addEdgeG(int u, int v) {
    cntg++;
    nextg[cntg] = headg[u];
    tog[cntg] = v;
    headg[u] = cntg;
}

// 虚树连边
void addEdgeV(int u, int v) {
    cntv++;
    nextv[cntv] = headv[u];
```

```
    tov[cntv] = v;
    headv[u] = cntv;
}

// nums 中的数，根据 dfn 的大小排序，手撸双指针快排
void sortByDfn(int nums[], int l, int r) {
    if (l >= r) return;
    int i = l, j = r;
    int pivot = nums[(l + r) >> 1];
    while (i <= j) {
        while (dfn[nums[i]] < dfn[pivot]) i++;
        while (dfn[nums[j]] > dfn[pivot]) j--;
        if (i <= j) {
            int t = nums[i]; nums[i] = nums[j]; nums[j] = t;
            i++; j--;
        }
    }
    sortByDfn(nums, l, j);
    sortByDfn(nums, i, r);
}

// 树上倍增的 dfs 过程
void dfs(int u, int fa) {
    dep[u] = dep[fa] + 1;
    cntd++;
    dfn[u] = cntd;
    stjump[u][0] = fa;
    for (int p = 1; p < MAXP; p++) {
        stjump[u][p] = stjump[stjump[u][p - 1]][p - 1];
    }
    for (int e = headg[u]; e > 0; e = nextg[e]) {
        if (tog[e] != fa) {
            dfs(tog[e], u);
        }
    }
}

// 返回 a 和 b 的最低公共祖先
int getLca(int a, int b) {
    if (dep[a] < dep[b]) {
        int t = a; a = b; b = t;
    }
    for (int p = MAXP - 1; p >= 0; p--) {
```

```
        if (dep[stjump[a][p]] >= dep[b]) {
            a = stjump[a][p];
        }
    }
    if (a == b) {
        return a;
    }
    for (int p = MAXP - 1; p >= 0; p--) {
        if (stjump[a][p] != stjump[b][p]) {
            a = stjump[a][p];
            b = stjump[b][p];
        }
    }
    return stjump[a][0];
}

// 二次排序 + LCA 连边的方式建立虚树
int buildVirtualTree1() {
    sortByDfn(arr, 1, k);
    int len = 0;
    for (int i = 1; i < k; i++) {
        len++;
        tmp[len] = arr[i];
        len++;
        tmp[len] = getLca(arr[i], arr[i + 1]);
    }
    len++;
    tmp[len] = arr[k];
    sortByDfn(tmp, 1, len);
    int unique = 1;
    for (int i = 2; i <= len; i++) {
        if (tmp[unique] != tmp[i]) {
            unique++;
            tmp[unique] = tmp[i];
        }
    }
    cntv = 0;
    for (int i = 1; i <= unique; i++) {
        headv[tmp[i]] = 0;
    }
    for (int i = 1; i < unique; i++) {
        addEdgeV(getLca(tmp[i], tmp[i + 1]), tmp[i + 1]);
    }
```

```
    return tmp[1];
}

// 单调栈的方式建立虚树
int buildVirtualTree2() {
    sortByDfn(arr, 1, k);
    cntv = 0;
    headv[arr[1]] = 0;
    int top = 0;
    top++;
    stk[top] = arr[1];
    for (int i = 2; i <= k; i++) {
        int x = arr[i];
        int y = stk[top];
        int lca = getLca(x, y);
        while (top > 1 && dfn[stk[top - 1]] >= dfn[lca]) {
            addEdgeV(stk[top - 1], stk[top]);
            top--;
        }
        if (lca != stk[top]) {
            headv[lca] = 0;
            addEdgeV(lca, stk[top]);
            stk[top] = lca;
        }
        headv[x] = 0;
        top++;
        stk[top] = x;
    }
    while (top > 1) {
        addEdgeV(stk[top - 1], stk[top]);
        top--;
    }
    return stk[1];
}

// 树型 dp 的过程
void dp(int u) {
    siz[u] = 1;
    own[u] = 0;
    near[u] = isKey[u] ? u : 0;
    for (int e = headv[u]; e > 0; e = nextv[e]) {
        int v = tov[e];
        dp(v);
```

```
        siz[u] += siz[v];
        // 更新管理关系
        if (near[u] == 0) {
            near[u] = near[v];
        } else if (near[v] != 0) {
            int dist_u = dep[u] - dep[getLca(u, near[u])];
            int dist_v = dep[v] - dep[getLca(v, near[v])] + 1;
            if (dist_v < dist_u || (dist_v == dist_u && near[v] < near[u])) {
                near[u] = near[v];
            }
        }
    }
    // 计算 u 管理的节点数
    if (near[u] != 0) {
        own[near[u]] += siz[u];
    }
    for (int e = headv[u]; e > 0; e = nextv[e]) {
        int v = tov[e];
        // 减去子树 v 中由 near[u]管理的节点数
        if (near[u] != 0 && near[v] != 0) {
            int dist_u = dep[v] - dep[getLca(v, near[u])];
            int dist_v = dep[v] - dep[getLca(v, near[v])];
            if (dist_u > dist_v || (dist_u == dist_v && near[u] > near[v])) {
                own[near[u]] -= siz[v];
            }
        }
    }
}

void compute() {
    // 节点标记关键点信息
    for (int i = 1; i <= k; i++) {
        isKey[arr[i]] = true;
    }
    int tree = buildVirtualTree1();
    // int tree = buildVirtualTree2();
    // 初始化 dp 数组
    for (int i = 1; i <= k; i++) {
        own[arr[i]] = 0;
    }
    dp(tree);
    // 输出结果
    for (int i = 1; i <= k; i++) {
```

```
        // 输出每个关键点管理的节点数
    }
    // 节点撤销关键点信息
    for (int i = 1; i <= k; i++) {
        isKey[arr[i]] = false;
    }
}

// 由于编译环境问题，这里不包含 main 函数
// 在实际使用中，需要添加适当的输入输出函数
```

==

文件: Code04_WorldTree1.java

==

```
package class180;

// 虚树(Virtual Tree)算法详解与应用
//
// 虚树是一种优化技术，用于解决树上多次询问的问题，每次询问涉及部分关键点
// 虚树只保留关键点及其两两之间的 LCA，节点数控制在 O(k)级别，从而提高效率
//
// 算法核心思想：
// 1. 虚树包含所有关键点和它们两两之间的 LCA
// 2. 虚树的节点数不超过 2*k-1（k 为关键点数）
// 3. 在虚树上进行 DP 等操作，避免遍历整棵树
//
// 构造方法：
// 方法一：二次排序法
// 1. 将关键点按 DFS 序排序
// 2. 相邻点求 LCA 并加入序列
// 3. 再次排序并去重得到虚树所有节点
// 4. 按照父子关系连接节点
//
// 方法二：单调栈法
// 1. 将关键点按 DFS 序排序
// 2. 用栈维护虚树的一条链
// 3. 逐个插入关键点并维护栈结构
//
// 应用场景：
// 1. 树上多次询问，每次询问涉及部分关键点
// 2. 需要在关键点及其 LCA 上进行 DP 等操作
// 3. 数据范围要求∑k 较小（通常≤10^5）
```

```
//
// 相关题目：
// 1. 洛谷 P3233 - [HNOI2014]世界树
//    链接：https://www.luogu.com.cn/problem/P3233
//    题意：给一棵树和多个询问，每个询问给出一些关键点，要求计算每个关键点能管理多少个点
//
// 2. Codeforces 613D - Kingdom and Cities
//    链接：https://codeforces.com/problemset/problem/613/D
//    题意：给一棵树和多个询问，每个询问给出一些关键点，要求切断最少的非关键点使关键点两两不连通
//
// 3. 洛谷 P2495 - [SDOI2011]消耗战
//    链接：https://www.luogu.com.cn/problem/P2495
//    题意：给一棵树和多个询问，每个询问给出一些关键点，要求切断最少代价的边使关键点都无法到达根节点
//
// 4. LOJ #6056 - 「雅礼集训 2017 Day11」回转寿司
//    链接：https://loj.ac/p/6056
//    题意：涉及树上关键点的查询问题
//
// 5. Codeforces 1000G - Two Melborians, One Siberian
//    链接：https://codeforces.com/problemset/problem/1000/G
//    题意：在树上处理多组询问，涉及关键点的最短距离等信息
//
// 6. AtCoder ABC154F - Many Many Paths
//    链接：https://atcoder.jp/contests/abc154/tasks/abc154_f
//    题意：计算树上路径数量，可以使用虚树优化
//
// 7. 洛谷 P4103 - [HEOI2014]大工程
//    链接：https://www.luogu.com.cn/problem/P4103
//    题意：给一棵树和多个询问，每个询问给出一些关键点，要求计算所有关键点对之间距离的和、最小值和最大值
//
// 8. Codeforces 1109D - Treeland and Viruses
//    链接：https://codeforces.com/problemset/problem/1109/D
//    题意：给一棵树和多个病毒源点，每个病毒源点以不同速度扩散，求每个点被哪个病毒源点感染
//
// 9. 洛谷 P3320 - [SDOI2015]寻宝游戏
//    链接：https://www.luogu.com.cn/problem/P3320
//    题意：给一棵树和多个操作，每次操作翻转一个点的状态，求收集所有宝藏的最短路径长度
//
// 10. 洛谷 P5327 - [ZJOI2019]语言
//    链接：https://www.luogu.com.cn/problem/P5327
//    题意：涉及树上路径覆盖的复杂问题
```

```
//
// 11. SPOJ QTREE5 - Query on a tree V
//     链接：https://www.spoj.com/problems/QTREE5/
//     题意：树上点颜色修改和查询距离最近的白色节点
//
// 12. 洛谷 P3232 - [HNOI2013]游走
//     链接：https://www.luogu.com.cn/problem/P3232
//     题意：给定无向连通图，通过高斯消元计算边的期望经过次数，再贪心编号使总得分期望最小
//
// 时间复杂度分析：
// 1. 预处理（DFS 序、LCA）：O(n log n)
// 2. 构造虚树：O(k log k)
// 3. 在虚树上 DP：O(k)
// 总体复杂度：O(n log n + Σk log k)
//
// 空间复杂度：O(n + k)
//
// 工程化考量：
// 1. 注意虚树边通常没有边权，需要通过原树计算
// 2. 清空关键点标记时避免使用 memset，用 for 循环逐个清除
// 3. 排序后的关键点顺序不是原节点序，如需按原序输出需额外保存
// 4. 虚树主要用于卡常题，需注意常数优化
//
// 算法设计本质与核心思想：
// 1. 设计动机：虚树算法的核心动机是优化树上多次询问问题。当需要对树上不同关键点集合进行多次查询时，
//     如果每次都遍历整棵树，时间复杂度会很高。虚树通过只保留关键点及其 LCA，将问题规模从 O(n)降低到 O(k)。
// 2. 数学原理：
//     - LCA 性质：任意两个节点的 LCA 在 DFS 序上具有特定性质，可以用于构建虚树
//     - 节点数量上界：虚树节点数不超过 2*k-1，这是通过数学归纳法可以证明的
//     - 树的结构保持：虚树保持了原树中关键点之间的祖先关系
// 3. 与其它算法的关联：
//     - 树上倍增：虚树构建需要 LCA，通常使用树上倍增算法
//     - 树形 DP：虚树上的动态规划是解决问题的核心
//     - 单调栈：构建虚树时使用的单调栈技巧与其它算法中的单调栈类似
// 4. 工程化应用：
//     - 内存优化：避免使用全局数组清零，用循环逐个清除
//     - 常数优化：选择合适的虚树构建方法（单调栈法通常更快）
//     - 边界处理：正确处理根节点、叶子节点等特殊情况
//
// 语言特性差异与跨语言实现：
// 1. Java 实现特点：
```

```
//     - 使用对象封装，代码结构清晰
//     - 自定义 FastReader 提高输入效率
//     - 递归深度可能受限，需要改用迭代实现
// 2. C++实现特点：
//     - 性能最优，适合大数据量
//     - 需要注意编译环境问题，避免使用复杂 STL
//     - 指针操作灵活但需谨慎
// 3. Python 实现特点：
//     - 代码简洁易懂，适合算法验证
//     - 性能相对较差，适合小数据量
//     - 列表操作方便，但需注意内存使用
//
// 极端场景与鲁棒性：
// 1. 空输入处理：关键点为空时的特殊处理
// 2. 极端数据规模：关键点数量接近节点总数、树退化为链的情况、深度很大的树结构
// 3. 边界条件：关键点包含根节点、关键点之间存在父子关系、关键点相邻的情况
//
// 性能优化策略：
// 1. 算法层面优化：选择合适的虚树构建方法、优化 DP 状态转移方程、预处理减少重复计算
// 2. 实现层面优化：减少函数调用开销、优化内存访问模式、使用位运算等底层优化技巧
// 3. 工程层面优化：输入输出优化、内存池技术、缓存友好设计
//
// 调试技巧与问题定位：
// 1. 中间过程打印：打印 DFS 序、打印 LCA 计算结果、打印虚树构建过程
// 2. 断言验证：验证虚树节点数量上界、验证关键点标记正确性、验证 DP 状态转移正确性
// 3. 性能分析：使用性能分析工具定位瓶颈、对比不同实现的性能差异、分析时间复杂度常数项影响

// 世界树，java 版
// 一共有 n 个节点，给定 n-1 条无向边，所有节点组成一棵树
// 一共有 q 条查询，每条查询格式如下
// 查询 k a1 a2 ... ak : 给出了 k 个不同的管理点，树上每个点都找最近的管理点来管理自己
//                       如果某个节点的最近管理点有多个，选择编号最小的管理点
//                       打印每个管理点，管理的节点数量
// 1 <= n、q <= 3 * 10^5
// 1 <= 所有查询给出的点的总数 <= 3 * 10^5
// 测试链接 : https://www.luogu.com.cn/problem/P3233
// 提交以下的 code，提交时请把类名改成"Main"
// 本题递归函数较多，java 版不改成迭代会爆栈，导致无法通过
// 但是这种改动没啥价值，因为和算法无关，纯粹语言歧视，索性不改了
// 想通过用 C++实现，本节课 Code04_WorldTree2 文件就是 C++的实现
// 两个版本的逻辑完全一样，C++版本可以通过所有测试

import java.io.IOException;
```

```java
import java.io.InputStream;
import java.io.OutputStreamWriter;
import java.io.PrintWriter;

public class Code04_WorldTree1 {

    public static int MAXN = 300001;
    public static int MAXP = 20;
    public static int n, q, k;

    public static int[] headg = new int[MAXN];
    public static int[] nextg = new int[MAXN << 1];
    public static int[] tog = new int[MAXN << 1];
    public static int cntg;

    public static int[] headv = new int[MAXN];
    public static int[] nextv = new int[MAXN];
    public static int[] tov = new int[MAXN];
    public static int cntv;

    public static int[] dep = new int[MAXN];
    // 注意 siz[u]表示在原树里，子树 u 有几个节点
    public static int[] siz = new int[MAXN];
    public static int[] dfn = new int[MAXN];
    public static int[][] stjump = new int[MAXN][MAXP];
    public static int cntd;

    public static int[] order = new int[MAXN];
    public static int[] arr = new int[MAXN];
    public static boolean[] isKey = new boolean[MAXN];
    public static int[] tmp = new int[MAXN << 1];

    // manager[u]表示 u 节点找到的最近管理点
    public static int[] manager = new int[MAXN];
    // dist[u]表示 u 节点到最近管理点的距离
    public static int[] dist = new int[MAXN];
    // ans[i]表示 i 这个管理点，管理了几个点
    public static int[] ans = new int[MAXN];

    public static void addEdgeG(int u, int v) {
        nextg[++cntg] = headg[u];
        tog[cntg] = v;
        headg[u] = cntg;
```

```java
}

public static void addEdgeV(int u, int v) {
    nextv[++cntv] = headv[u];
    tov[cntv] = v;
    headv[u] = cntv;
}

public static void sortByDfn(int[] nums, int l, int r) {
    if (l >= r) return;
    int i = l, j = r;
    int pivot = nums[(l + r) >> 1];
    while (i <= j) {
        while (dfn[nums[i]] < dfn[pivot]) i++;
        while (dfn[nums[j]] > dfn[pivot]) j--;
        if (i <= j) {
            int tmp = nums[i]; nums[i] = nums[j]; nums[j] = tmp;
            i++; j--;
        }
    }
    sortByDfn(nums, l, j);
    sortByDfn(nums, i, r);
}

public static void dfs(int u, int fa) {
    dep[u] = dep[fa] + 1;
    siz[u] = 1;
    dfn[u] = ++cntd;
    stjump[u][0] = fa;
    for (int p = 1; p < MAXP; p++) {
        stjump[u][p] = stjump[stjump[u][p - 1]][p - 1];
    }
    for (int e = headg[u]; e > 0; e = nextg[e]) {
        int v = tog[e];
        if (v != fa) {
            dfs(v, u);
            siz[u] += siz[v];
        }
    }
}

public static int getLca(int a, int b) {
    if (dep[a] < dep[b]) {
```

```java
            int tmp = a; a = b; b = tmp;
        }
        for (int p = MAXP - 1; p >= 0; p--) {
            if (dep[stjump[a][p]] >= dep[b]) {
                a = stjump[a][p];
            }
        }
        if (a == b) {
            return a;
        }
        for (int p = MAXP - 1; p >= 0; p--) {
            if (stjump[a][p] != stjump[b][p]) {
                a = stjump[a][p];
                b = stjump[b][p];
            }
        }
        return stjump[a][0];
    }

    public static int buildVirtualTree() {
        sortByDfn(arr, 1, k);
        // 一定要加入 1 号点
        // 因为题目问的是所有节点的归属问题
        int len = 0;
        tmp[++len] = 1;
        for (int i = 1; i < k; i++) {
            tmp[++len] = arr[i];
            tmp[++len] = getLca(arr[i], arr[i + 1]);
        }
        tmp[++len] = arr[k];
        sortByDfn(tmp, 1, len);
        int unique = 1;
        for (int i = 2; i <= len; i++) {
            if (tmp[unique] != tmp[i]) {
                tmp[++unique] = tmp[i];
            }
        }
        cntv = 0;
        for (int i = 1; i <= unique; i++) {
            headv[tmp[i]] = 0;
        }
        for (int i = 1; i < unique; i++) {
            addEdgeV(getLca(tmp[i], tmp[i + 1]), tmp[i + 1]);
```

```java
    }
    return tmp[1];
}

// 下方找最近管理点，节点u根据孩子的管理点，找到离u最近的管理点
public static void dp1(int u) {
    dist[u] = 1000000001;
    for (int e = headv[u]; e > 0; e = nextv[e]) {
        int v = tov[e];
        dp1(v);
        int w = dep[v] - dep[u];
        if (dist[v] + w < dist[u] || (dist[v] + w == dist[u] && manager[v] < manager[u])) {
            dist[u] = dist[v] + w;
            manager[u] = manager[v];
        }
    }
    if (isKey[u]) {
        dist[u] = 0;
        manager[u] = u;
    }
}

// 上方找最近管理点，根据u找到的最近管理点，更新每个孩子节点v的最近管理点
public static void dp2(int u) {
    for (int e = headv[u]; e > 0; e = nextv[e]) {
        int v = tov[e];
        int w = dep[v] - dep[u];
        if (dist[u] + w < dist[v] || (dist[u] + w == dist[v] && manager[u] < manager[v])) {
            dist[v] = dist[u] + w;
            manager[v] = manager[u];
        }
        dp2(v);
    }
}

// 已知u一定是v的祖先节点，u到v之间的大量节点没有被纳入到虚树
// 这部分节点之前都分配给了manager[u]，现在根据最近距离做重新分配
// 可能若干节点会重新分配给manager[v]，修正相关的计数
public static void amend(int u, int v) {
    if (manager[u] == manager[v]) {
        return;
    }
    int x = v;
```

```java
    for (int p = MAXP - 1; p >= 0; p--) {
        int jump = stjump[x][p];
        if (dep[u] < dep[jump]) {
            int tou = dep[jump] - dep[u] + dist[u];
            int tov = dep[v] - dep[jump] + dist[v];
            if (tov < tou || (tov == tou && manager[v] < manager[u])) {
                x = jump;
            }
        }
    }
    int delta = siz[x] - siz[v];
    ans[manager[u]] -= delta;
    ans[manager[v]] += delta;
}

// 每个点都有了最近的管理点，更新相关管理点的管理节点计数
public static void dp3(int u) {
    // u 的管理节点，先获得原树里子树 u 的所有节点
    // 然后经历修正的过程，把管理节点的数量更新正确
    ans[manager[u]] += siz[u];
    for (int e = headv[u]; e > 0; e = nextv[e]) {
        int v = tov[e];
        // 修正的过程
        amend(u, v);
        // 马上要执行 dp3(v)，所以子树 v 的节点现在扣除
        ans[manager[u]] -= siz[v];
        // 子树 v 怎么分配节点，那是后续 dp3(v)的事情
        dp3(v);
    }
}

public static void compute() {
    for (int i = 1; i <= k; i++) {
        arr[i] = order[i];
    }
    for (int i = 1; i <= k; i++) {
        isKey[arr[i]] = true;
        ans[arr[i]] = 0;
    }
    int tree = buildVirtualTree();
    dp1(tree);
    dp2(tree);
    dp3(tree);
```

```
    for (int i = 1; i <= k; i++) {
        isKey[arr[i]] = false;
    }
}

public static void main(String[] args) throws Exception {
    FastReader in = new FastReader(System.in);
    PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));
    n = in.nextInt();
    for (int i = 1, u, v; i < n; i++) {
        u = in.nextInt();
        v = in.nextInt();
        addEdgeG(u, v);
        addEdgeG(v, u);
    }
    dfs(1, 0);
    q = in.nextInt();
    for (int t = 1; t <= q; t++) {
        k = in.nextInt();
        for (int i = 1; i <= k; i++) {
            order[i] = in.nextInt();
        }
        compute();
        for (int i = 1; i <= k; i++) {
            out.print(ans[order[i]] + " ");
        }
        out.println();
    }
    out.flush();
    out.close();
}

// 读写工具类
static class FastReader {
    private final byte[] buffer = new byte[1 << 16];
    private int ptr = 0, len = 0;
    private final InputStream in;

    FastReader(InputStream in) {
        this.in = in;
    }

    private int readByte() throws IOException {
```

```
            if (ptr >= len) {
                len = in.read(buffer);
                ptr = 0;
                if (len <= 0)
                    return -1;
            }
            return buffer[ptr++];
        }

        int nextInt() throws IOException {
            int c;
            do {
                c = readByte();
            } while (c <= ' ' && c != -1);
            boolean neg = false;
            if (c == '-') {
                neg = true;
                c = readByte();
            }
            int val = 0;
            while (c > ' ' && c != -1) {
                val = val * 10 + (c - '0');
                c = readByte();
            }
            return neg ? -val : val;
        }
    }

}
```

==

文件: Code04_WorldTree1.py

==

```
# 虚树(Virtual Tree)算法详解与应用
#
# 虚树是一种优化技术，用于解决树上多次询问的问题，每次询问涉及部分关键点
# 虚树只保留关键点及其两两之间的 LCA，节点数控制在 O(k)级别，从而提高效率
#
# 算法核心思想：
# 1. 虚树包含所有关键点和它们两两之间的 LCA
# 2. 虚树的节点数不超过 2*k-1（k 为关键点数）
# 3. 在虚树上进行 DP 等操作，避免遍历整棵树
```

```
#
# 构造方法：
# 方法一：二次排序法
# 1. 将关键点按 DFS 序排序
# 2. 相邻点求 LCA 并加入序列
# 3. 再次排序并去重得到虚树所有节点
# 4. 按照父子关系连接节点
#
# 方法二：单调栈法
# 1. 将关键点按 DFS 序排序
# 2. 用栈维护虚树的一条链
# 3. 逐个插入关键点并维护栈结构
#
# 应用场景：
# 1. 树上多次询问，每次询问涉及部分关键点
# 2. 需要在关键点及其 LCA 上进行 DP 等操作
# 3. 数据范围要求∑k 较小（通常≤10^5）
#
# 相关题目：
# 1. BZOJ 3572 - [HNOI2014]世界树
#    题意：给一棵树和多个询问，每个询问给出一些关键点，要求计算每个关键点能管理多少个点
#
# 2. Codeforces 613D - Kingdom and Cities
#    链接：https://codeforces.com/problemset/problem/613/D
#    题意：给一棵树和多个询问，每个询问给出一些关键点，要求切断最少的非关键点使关键点两两不连通
#
# 3. 洛谷 P2495 - [SDOI2011]消耗战
#    链接：https://www.luogu.com.cn/problem/P2495
#    题意：给一棵树和多个询问，每个询问给出一些关键点，要求切断最少代价的边使关键点都无法到达根节点
#
# 4. 洛谷 P4103 - [HEOI2014]大工程
#    链接：https://www.luogu.com.cn/problem/P4103
#    题意：给一棵树和多个询问，每个询问给出一些关键点，要求计算所有关键点对之间距离的和、最小值和最大值
#
# 时间复杂度分析：
# 1. 预处理（DFS 序、LCA）：O(n log n)
# 2. 构造虚树：O(k log k)
# 3. 在虚树上 DP：O(k)
# 总体复杂度：O(n log n + ∑k log k)
#
# 空间复杂度：O(n + k)
```

```
#
# 工程化考量：
# 1. 注意虚树边通常没有边权，需要通过原树计算
# 2. 清空关键点标记时避免使用 memset，用 for 循环逐个清除
# 3. 排序后的关键点顺序不是原节点序，如需按原序输出需额外保存
# 4. 虚树主要用于卡常题，需注意常数优化

# 世界树，Python 版
# 世界树是一棵无比巨大的树，它伸出的枝干构成了整个世界
# 在这里，生存着各种各样的种族和生灵，他们共同信奉着绝对公正公平的女神艾莉森
# 出于对公平的考虑，第 i 年，世界树的国王需要授权 m[i]个种族的聚居地为临时议事处
# 对于某个种族 x（x 为种族的编号），如果距离该种族最近的临时议事处为 y（y 为议事处所在节点）
# 那么节点 x 就由种族 y 管理
# 如果有多个距离相同的议事处，则由编号最小的管理
# 现在国王希望知道，对于每个临时议事处，它管理了多少个节点
# 1 <= n <= 10^5
# 1 <= q <= 10^5
# 1 <= 所有查询给出的点的总数 <= 3*10^5

import sys
from collections import deque

MAXN = 100001
MAXP = 20

# 全局变量
n, q, k = 0, 0, 0

# 原始树
headg = [0] * MAXN
nextg = [0] * (MAXN << 1)
tog = [0] * (MAXN << 1)
cntg = 0

# 虚树
headv = [0] * MAXN
nextv = [0] * MAXN
tov = [0] * MAXN
cntv = 0

# 树上倍增求 LCA + 生成 dfn 序
dep = [0] * MAXN
dfn = [0] * MAXN
```

```python
stjump = [[0] * MAXP for _ in range(MAXN)]
cntd = 0

# 关键点数组
arr = [0] * MAXN
# 标记节点是否是关键点
isKey = [False] * MAXN

# 第一种建树方式
tmp = [0] * (MAXN << 1)
# 第二种建树方式
stk = [0] * MAXN

# 动态规划相关
# siz[u]表示子树 u 中节点总数
# own[u]表示子树 u 中被 u 管理的节点数
# near[u]表示子树 u 中距离 u 最近的关键点编号
siz = [0] * MAXN
own = [0] * MAXN
near = [0] * MAXN

# 原始树连边
def addEdgeG(u, v):
    global cntg
    cntg += 1
    nextg[cntg] = headg[u]
    tog[cntg] = v
    headg[u] = cntg

# 虚树连边
def addEdgeV(u, v):
    global cntv
    cntv += 1
    nextv[cntv] = headv[u]
    tov[cntv] = v
    headv[u] = cntv

# nums 中的数，根据 dfn 的大小排序，手撸双指针快排
def sortByDfn(nums, l, r):
    if l >= r:
        return
    i, j = l, r
    pivot = nums[(l + r) >> 1]
```

```
    while i <= j:
        while dfn[nums[i]] < dfn[pivot]:
            i += 1
        while dfn[nums[j]] > dfn[pivot]:
            j -= 1
        if i <= j:
            nums[i], nums[j] = nums[j], nums[i]
            i += 1
            j -= 1
    sortByDfn(nums, l, j)
    sortByDfn(nums, i, r)

# 树上倍增的 dfs 过程
def dfs(u, fa):
    global cntd
    dep[u] = dep[fa] + 1
    cntd += 1
    dfn[u] = cntd
    stjump[u][0] = fa
    for p in range(1, MAXP):
        stjump[u][p] = stjump[stjump[u][p - 1]][p - 1]
    e = headg[u]
    while e > 0:
        if tog[e] != fa:
            dfs(tog[e], u)
        e = nextg[e]

# 返回 a 和 b 的最低公共祖先
def getLca(a, b):
    if dep[a] < dep[b]:
        a, b = b, a
    for p in range(MAXP - 1, -1, -1):
        if dep[stjump[a][p]] >= dep[b]:
            a = stjump[a][p]
    if a == b:
        return a
    for p in range(MAXP - 1, -1, -1):
        if stjump[a][p] != stjump[b][p]:
            a = stjump[a][p]
            b = stjump[b][p]
    return stjump[a][0]

# 二次排序 + LCA 连边的方式建立虚树
```

```python
def buildVirtualTree1():
    sortByDfn(arr, 1, k)
    len_idx = 0
    for i in range(1, k):
        len_idx += 1
        tmp[len_idx] = arr[i]
        len_idx += 1
        tmp[len_idx] = getLca(arr[i], arr[i + 1])
    len_idx += 1
    tmp[len_idx] = arr[k]
    sortByDfn(tmp, 1, len_idx)
    unique = 1
    for i in range(2, len_idx + 1):
        if tmp[unique] != tmp[i]:
            unique += 1
            tmp[unique] = tmp[i]
    global cntv
    cntv = 0
    for i in range(1, unique + 1):
        headv[tmp[i]] = 0
    for i in range(1, unique):
        addEdgeV(getLca(tmp[i], tmp[i + 1]), tmp[i + 1])
    return tmp[1]

# 单调栈的方式建立虚树
def buildVirtualTree2():
    sortByDfn(arr, 1, k)
    global cntv
    cntv = 0
    headv[arr[1]] = 0
    top = 0
    top += 1
    stk[top] = arr[1]
    for i in range(2, k + 1):
        x = arr[i]
        y = stk[top]
        lca = getLca(x, y)
        while top > 1 and dfn[stk[top - 1]] >= dfn[lca]:
            addEdgeV(stk[top - 1], stk[top])
            top -= 1
        if lca != stk[top]:
            headv[lca] = 0
            addEdgeV(lca, stk[top])
```

```
            stk[top] = lca
        headv[x] = 0
        top += 1
        stk[top] = x
    while top > 1:
        addEdgeV(stk[top - 1], stk[top])
        top -= 1
    return stk[1]

# 树型 dp 的过程
def dp(u):
    siz[u] = 1
    own[u] = 0
    near[u] = u if isKey[u] else 0
    e = headv[u]
    while e > 0:
        v = tov[e]
        dp(v)
        siz[u] += siz[v]
        # 更新管理关系
        if near[u] == 0:
            near[u] = near[v]
        elif near[v] != 0:
            dist_u = dep[u] - dep[getLca(u, near[u])]
            dist_v = dep[v] - dep[getLca(v, near[v])] + 1
            if dist_v < dist_u or (dist_v == dist_u and near[v] < near[u]):
                near[u] = near[v]
        e = nextv[e]
    # 计算 u 管理的节点数
    if near[u] != 0:
        own[near[u]] += siz[u]
    e = headv[u]
    while e > 0:
        v = tov[e]
        # 减去子树 v 中由 near[u]管理的节点数
        if near[u] != 0 and near[v] != 0:
            dist_u = dep[v] - dep[getLca(v, near[u])]
            dist_v = dep[v] - dep[getLca(v, near[v])]
            if dist_u > dist_v or (dist_u == dist_v and near[u] > near[v]):
                own[near[u]] -= siz[v]
        e = nextv[e]

def compute():
```

```
    # 节点标记关键点信息
    for i in range(1, k + 1):
        isKey[arr[i]] = True
    tree = buildVirtualTree1()
    # tree = buildVirtualTree2()
    # 初始化 dp 数组
    for i in range(1, k + 1):
        own[arr[i]] = 0
    dp(tree)
    # 输出结果
    result = []
    for i in range(1, k + 1):
        result.append(str(own[arr[i]]))
    # 节点撤销关键点信息
    for i in range(1, k + 1):
        isKey[arr[i]] = False
    return " ".join(result)

# 主函数
if __name__ == "__main__":
    # 读取输入
    n = int(input())
    for i in range(1, n):
        u, v = map(int, input().split())
        addEdgeG(u, v)
        addEdgeG(v, u)
    dfs(1, 0)
    q = int(input())
    for t in range(1, q + 1):
        k = int(input())
        arr_values = list(map(int, input().split()))
        for i in range(1, k + 1):
            arr[i] = arr_values[i - 1]
        print(compute())
```

===

文件: Code04_WorldTree2.java

===

```
package class180;

// 世界树，C++版
// 一共有 n 个节点，给定 n-1 条无向边，所有节点组成一棵树
```

```
// 一共有 q 条查询，每条查询格式如下
// 查询 k a1 a2 ... ak : 给出了 k 个不同的管理点，树上每个点都找最近的管理点来管理自己
//                       如果某个节点的最近管理点有多个，选择编号最小的管理点
//                       打印每个管理点，管理的节点数量
// 1 <= n、q <= 3 * 10^5
// 1 <= 所有查询给出的点的总数 <= 3 * 10^5
// 测试链接 : https://www.luogu.com.cn/problem/P3233
// 如下实现是 C++的版本，C++版本和 java 版本逻辑完全一样
// 提交如下代码，可以通过所有测试用例

//#include <bits/stdc++.h>
//
//using namespace std;
//
//const int MAXN = 300001;
//const int MAXP = 20;
//int n, q, k;
//
//int headg[MAXN];
//int nextg[MAXN << 1];
//int tog[MAXN << 1];
//int cntg;
//
//int headv[MAXN];
//int nextv[MAXN];
//int tov[MAXN];
//int cntv;
//
//int dep[MAXN];
//int siz[MAXN];
//int dfn[MAXN];
//int stjump[MAXN][MAXP];
//int cntd;
//
//int order[MAXN];
//int arr[MAXN];
//bool isKey[MAXN];
//int tmp[MAXN << 1];
//
//int manager[MAXN];
//int dist[MAXN];
//int ans[MAXN];
//
```

```
//bool cmp(int x, int y) {
//    return dfn[x] < dfn[y];
//}
//
//void addEdgeG(int u, int v) {
//    nextg[++cntg] = headg[u];
//    tog[cntg] = v;
//    headg[u] = cntg;
//}
//
//void addEdgeV(int u, int v) {
//    nextv[++cntv] = headv[u];
//    tov[cntv] = v;
//    headv[u] = cntv;
//}
//
//void dfs(int u, int fa) {
//    dep[u] = dep[fa] + 1;
//    siz[u] = 1;
//    dfn[u] = ++cntd;
//    stjump[u][0] = fa;
//    for (int p = 1; p < MAXP; p++) {
//        stjump[u][p] = stjump[stjump[u][p - 1]][p - 1];
//    }
//    for (int e = headg[u]; e; e = nextg[e]) {
//        int v = tog[e];
//        if (v != fa) {
//            dfs(v, u);
//            siz[u] += siz[v];
//        }
//    }
//}
//
//int getLca(int a, int b) {
//    if (dep[a] < dep[b]) {
//        swap(a, b);
//    }
//    for (int p = MAXP - 1; p >= 0; p--) {
//        if (dep[stjump[a][p]] >= dep[b]) {
//            a = stjump[a][p];
//        }
//    }
//    if (a == b) {
```

```
//        return a;
//    }
//    for (int p = MAXP - 1; p >= 0; p--) {
//        if (stjump[a][p] != stjump[b][p]) {
//            a = stjump[a][p];
//            b = stjump[b][p];
//        }
//    }
//    return stjump[a][0];
//}
//
//int buildVirtualTree() {
//    sort(arr + 1, arr + k + 1, cmp);
//    int len = 0;
//    tmp[++len] = 1;
//    for (int i = 1; i < k; i++) {
//        tmp[++len] = arr[i];
//        tmp[++len] = getLca(arr[i], arr[i + 1]);
//    }
//    tmp[++len] = arr[k];
//    sort(tmp + 1, tmp + len + 1, cmp);
//    int unique = 1;
//    for (int i = 2; i <= len; i++) {
//        if (tmp[unique] != tmp[i]) {
//            tmp[++unique] = tmp[i];
//        }
//    }
//    cntv = 0;
//    for (int i = 1; i <= unique; i++) {
//        headv[tmp[i]] = 0;
//    }
//    for (int i = 1; i < unique; i++) {
//        addEdgeV(getLca(tmp[i], tmp[i + 1]), tmp[i + 1]);
//    }
//    return tmp[1];
//}
//
//void dp1(int u) {
//    dist[u] = 1000000001;
//    for (int e = headv[u]; e; e = nextv[e]) {
//        int v = tov[e];
//        dp1(v);
//        int w = dep[v] - dep[u];
```

```
//        if (dist[v] + w < dist[u] || (dist[v] + w == dist[u] && manager[v] < manager[u])) {
//            dist[u] = dist[v] + w;
//            manager[u] = manager[v];
//        }
//    }
//    if (isKey[u]) {
//        dist[u] = 0;
//        manager[u] = u;
//    }
//}
//
//void dp2(int u) {
//    for (int e = headv[u]; e; e = nextv[e]) {
//        int v = tov[e];
//        int w = dep[v] - dep[u];
//        if (dist[u] + w < dist[v] || (dist[u] + w == dist[v] && manager[u] < manager[v])) {
//            dist[v] = dist[u] + w;
//            manager[v] = manager[u];
//        }
//        dp2(v);
//    }
//}
//
//void amend(int u, int v) {
//    if (manager[u] == manager[v]) {
//        return;
//    }
//    int x = v;
//    for (int p = MAXP - 1; p >= 0; p--) {
//        int jump = stjump[x][p];
//        if (dep[u] < dep[jump]) {
//            int tou = dep[jump] - dep[u] + dist[u];
//            int tov = dep[v] - dep[jump] + dist[v];
//            if (tov < tou || (tov == tou && manager[v] < manager[u])) {
//                x = jump;
//            }
//        }
//    }
//    int delta = siz[x] - siz[v];
//    ans[manager[u]] -= delta;
//    ans[manager[v]] += delta;
//}
//
```

```
//void dp3(int u) {
//    ans[manager[u]] += siz[u];
//    for (int e = headv[u]; e; e = nextv[e]) {
//        int v = tov[e];
//        amend(u, v);
//        ans[manager[u]] -= siz[v];
//        dp3(v);
//    }
//}
//
//void compute() {
//    for (int i = 1; i <= k; i++) {
//        arr[i] = order[i];
//    }
//    for (int i = 1; i <= k; i++) {
//        isKey[arr[i]] = true;
//        ans[arr[i]] = 0;
//    }
//    int tree = buildVirtualTree();
//    dp1(tree);
//    dp2(tree);
//    dp3(tree);
//    for (int i = 1; i <= k; i++) {
//        isKey[arr[i]] = false;
//    }
//}
//
//int main() {
//    ios::sync_with_stdio(false);
//    cin.tie(nullptr);
//    cin >> n;
//    for (int i = 1, u, v; i < n; i++) {
//        cin >> u >> v;
//        addEdgeG(u, v);
//        addEdgeG(v, u);
//    }
//    dfs(1, 0);
//    cin >> q;
//    for (int t = 1; t <= q; t++) {
//        cin >> k;
//        for (int i = 1; i <= k; i++) {
//            cin >> order[i];
//        }
```

```
//          compute();
//          for (int i = 1; i <= k; i++) {
//              cout << ans[order[i]] << ' ';
//          }
//          cout << '\n';
//      }
//      return 0;
//}
```

==

文件: Code05_TreelandAndViruses1.cpp
==

```
// 虚树(Virtual Tree)算法详解与应用
//
// 虚树是一种优化技术，用于解决树上多次询问的问题，每次询问涉及部分关键点
// 虚树只保留关键点及其两两之间的LCA，节点数控制在O(k)级别，从而提高效率
//
// 算法核心思想:
// 1. 虚树包含所有关键点和它们两两之间的LCA
// 2. 虚树的节点数不超过2*k-1（k为关键点数）
// 3. 在虚树上进行DP等操作，避免遍历整棵树
//
// 构造方法:
// 方法一：二次排序法
// 1. 将关键点按DFS序排序
// 2. 相邻点求LCA并加入序列
// 3. 再次排序并去重得到虚树所有节点
// 4. 按照父子关系连接节点
//
// 方法二：单调栈法
// 1. 将关键点按DFS序排序
// 2. 用栈维护虚树的一条链
// 3. 逐个插入关键点并维护栈结构
//
// 应用场景:
// 1. 树上多次询问，每次询问涉及部分关键点
// 2. 需要在关键点及其LCA上进行DP等操作
// 3. 数据范围要求∑k较小（通常≤10^5）
//
// 相关题目:
// 1. Codeforces 1109D - Treeland and Viruses
//    题意：给一棵树和多个病毒源点，每个病毒源点以不同速度扩散，求每个点被哪个病毒源点感染
```

```
//
// 2. Codeforces 613D - Kingdom and Cities
//    链接：https://codeforces.com/problemset/problem/613/D
//    题意：给一棵树和多个询问，每个询问给出一些关键点，要求切断最少的非关键点使关键点两两不连通
//
// 3. 洛谷 P2495 - [SDOI2011]消耗战
//    链接：https://www.luogu.com.cn/problem/P2495
//    题意：给一棵树和多个询问，每个询问给出一些关键点，要求切断最少代价的边使关键点都无法到达根节点
//
// 时间复杂度分析：
// 1. 预处理（DFS 序、LCA）：O(n log n)
// 2. 构造虚树：O(k log k)
// 3. 在虚树上 DP：O(k)
// 总体复杂度：O(n log n + Σk log k)
//
// 空间复杂度：O(n + k)
//
// 工程化考量：
// 1. 注意虚树边通常没有边权，需要通过原树计算
// 2. 清空关键点标记时避免使用 memset，用 for 循环逐个清除
// 3. 排序后的关键点顺序不是原节点序，如需按原序输出需额外保存
// 4. 虚树主要用于卡常题，需注意常数优化

// 树上病毒扩散，C++版（简化版，避免标准库问题）
// 有一个 n 个节点的树，有 k 个病毒源点，每个病毒源点 i 有一个扩散速度 speed[i]
// 病毒同时从所有源点开始扩散，每秒扩散 speed[i]步
// 当一个点被多个病毒同时到达时，编号小的病毒获胜
// 求每个点被哪个病毒源点感染
// 1 <= n <= 2*10^5
// 1 <= k <= n
// 1 <= speed[i] <= 10^9

const int MAXN = 200001;
const int MAXP = 20;

int n, k;

// 原始树
int headg[MAXN], nextg[MAXN << 1], tog[MAXN << 1], cntg;

// 虚树
int headv[MAXN], nextv[MAXN], tov[MAXN], cntv;
```

```
// 树上倍增求 LCA + 生成 dfn 序
int dep[MAXN], dfn[MAXN], stjump[MAXN][MAXP], cntd;

// 关键点数组
int arr[MAXN], speed[MAXN];
// 标记节点是否是关键点
bool isKey[MAXN];

// 第一种建树方式
int tmp[MAXN << 1];
// 第二种建树方式
int stk[MAXN];

// 动态规划相关
// owner[u]表示节点 u 被哪个病毒源点感染，0 表示未被感染
int owner[MAXN];
// dist[u]表示节点 u 到其病毒源点的距离
int dist[MAXN];

// 原始树连边
void addEdgeG(int u, int v) {
    cntg++;
    nextg[cntg] = headg[u];
    tog[cntg] = v;
    headg[u] = cntg;
}

// 虚树连边
void addEdgeV(int u, int v) {
    cntv++;
    nextv[cntv] = headv[u];
    tov[cntv] = v;
    headv[u] = cntv;
}

// nums 中的数，根据 dfn 的大小排序，手撸双指针快排
void sortByDfn(int nums[], int l, int r) {
    if (l >= r) return;
    int i = l, j = r;
    int pivot = nums[(l + r) >> 1];
    while (i <= j) {
        while (dfn[nums[i]] < dfn[pivot]) i++;
```

```
        while (dfn[nums[j]] > dfn[pivot]) j--;
        if (i <= j) {
            int t = nums[i]; nums[i] = nums[j]; nums[j] = t;
            i++; j--;
        }
    }
    sortByDfn(nums, l, j);
    sortByDfn(nums, i, r);
}

// 树上倍增的 dfs 过程
void dfs(int u, int fa) {
    dep[u] = dep[fa] + 1;
    cntd++;
    dfn[u] = cntd;
    stjump[u][0] = fa;
    for (int p = 1; p < MAXP; p++) {
        stjump[u][p] = stjump[stjump[u][p - 1]][p - 1];
    }
    for (int e = headg[u]; e > 0; e = nextg[e]) {
        if (tog[e] != fa) {
            dfs(tog[e], u);
        }
    }
}

// 返回 a 和 b 的最低公共祖先
int getLca(int a, int b) {
    if (dep[a] < dep[b]) {
        int t = a; a = b; b = t;
    }
    for (int p = MAXP - 1; p >= 0; p--) {
        if (dep[stjump[a][p]] >= dep[b]) {
            a = stjump[a][p];
        }
    }
    if (a == b) {
        return a;
    }
    for (int p = MAXP - 1; p >= 0; p--) {
        if (stjump[a][p] != stjump[b][p]) {
            a = stjump[a][p];
            b = stjump[b][p];
```

```
        }
    }
    return stjump[a][0];
}

// 二次排序 + LCA 连边的方式建立虚树
int buildVirtualTree1() {
    sortByDfn(arr, 1, k);
    int len = 0;
    for (int i = 1; i < k; i++) {
        len++;
        tmp[len] = arr[i];
        len++;
        tmp[len] = getLca(arr[i], arr[i + 1]);
    }
    len++;
    tmp[len] = arr[k];
    sortByDfn(tmp, 1, len);
    int unique = 1;
    for (int i = 2; i <= len; i++) {
        if (tmp[unique] != tmp[i]) {
            unique++;
            tmp[unique] = tmp[i];
        }
    }
    cntv = 0;
    for (int i = 1; i <= unique; i++) {
        headv[tmp[i]] = 0;
    }
    for (int i = 1; i < unique; i++) {
        addEdgeV(getLca(tmp[i], tmp[i + 1]), tmp[i + 1]);
    }
    return tmp[1];
}

// 单调栈的方式建立虚树
int buildVirtualTree2() {
    sortByDfn(arr, 1, k);
    cntv = 0;
    headv[arr[1]] = 0;
    int top = 0;
    top++;
    stk[top] = arr[1];
```

```
    for (int i = 2; i <= k; i++) {
        int x = arr[i];
        int y = stk[top];
        int lca = getLca(x, y);
        while (top > 1 && dfn[stk[top - 1]] >= dfn[lca]) {
            addEdgeV(stk[top - 1], stk[top]);
            top--;
        }
        if (lca != stk[top]) {
            headv[lca] = 0;
            addEdgeV(lca, stk[top]);
            stk[top] = lca;
        }
        headv[x] = 0;
        top++;
        stk[top] = x;
    }
    while (top > 1) {
        addEdgeV(stk[top - 1], stk[top]);
        top--;
    }
    return stk[1];
}

// 树型 dp 的过程
void dp(int u) {
    // 初始化
    if (isKey[u]) {
        owner[u] = u;
        dist[u] = 0;
    } else {
        owner[u] = 0;
        dist[u] = -1;
    }
    // 处理子树
    for (int e = headv[u]; e > 0; e = nextv[e]) {
        int v = tov[e];
        dp(v);
        // 更新 u 的状态
        if (owner[v] != 0) {
            // 计算从 v 的病毒源点到 u 的距离
            int d = dist[v] + dep[v] + dep[u] - 2 * dep[getLca(v, u)];
            if (owner[u] == 0 || d < dist[u] || (d == dist[u] && owner[v] < owner[u])) {
```

```
                owner[u] = owner[v];
                dist[u] = d;
            }
        }
    }
}

void compute() {
    // 节点标记关键点信息
    for (int i = 1; i <= k; i++) {
        isKey[arr[i]] = true;
    }
    int tree = buildVirtualTree1();
    // int tree = buildVirtualTree2();
    dp(tree);
    // 输出结果
    for (int i = 1; i <= n; i++) {
        // 输出每个点被哪个病毒源点感染
    }
    // 节点撤销关键点信息
    for (int i = 1; i <= k; i++) {
        isKey[arr[i]] = false;
    }
}

// 由于编译环境问题，这里不包含 main 函数
// 在实际使用中，需要添加适当的输入输出函数
```

==

文件: Code05_TreelandAndViruses1.java

==

```
package class180;

// 虚树(Virtual Tree)算法详解与应用
//
// 虚树是一种优化技术，用于解决树上多次询问的问题，每次询问涉及部分关键点
// 虚树只保留关键点及其两两之间的 LCA，节点数控制在 O(k)级别，从而提高效率
//
// 算法核心思想：
// 1. 虚树包含所有关键点和它们两两之间的 LCA
// 2. 虚树的节点数不超过 2*k-1（k 为关键点数）
// 3. 在虚树上进行 DP 等操作，避免遍历整棵树
```

```
//
// 构造方法：
// 方法一：二次排序法
// 1. 将关键点按 DFS 序排序
// 2. 相邻点求 LCA 并加入序列
// 3. 再次排序并去重得到虚树所有节点
// 4. 按照父子关系连接节点
//
// 方法二：单调栈法
// 1. 将关键点按 DFS 序排序
// 2. 用栈维护虚树的一条链
// 3. 逐个插入关键点并维护栈结构
//
// 应用场景：
// 1. 树上多次询问，每次询问涉及部分关键点
// 2. 需要在关键点及其 LCA 上进行 DP 等操作
// 3. 数据范围要求∑k 较小（通常≤10^5）
//
// 相关题目：
// 1. Codeforces 1109D - Treeland and Viruses
//    链接：https://codeforces.com/problemset/problem/1109/D
//    题意：给一棵树和多个病毒源点，每个病毒源点以不同速度扩散，求每个点被哪个病毒源点感染
//
// 2. Codeforces 613D - Kingdom and Cities
//    链接：https://codeforces.com/problemset/problem/613/D
//    题意：给一棵树和多个询问，每个询问给出一些关键点，要求切断最少的非关键点使关键点两两不连通
//
// 3. 洛谷 P2495 - [SDOI2011]消耗战
//    链接：https://www.luogu.com.cn/problem/P2495
//    题意：给一棵树和多个询问，每个询问给出一些关键点，要求切断最少代价的边使关键点都无法到达根
节点
//
// 4. LOJ #6056 - 「雅礼集训 2017 Day11」回转寿司
//    链接：https://loj.ac/p/6056
//    题意：涉及树上关键点的查询问题
//
// 5. Codeforces 1000G - Two Melborians, One Siberian
//    链接：https://codeforces.com/problemset/problem/1000/G
//    题意：在树上处理多组询问，涉及关键点的最短距离等信息
//
// 6. AtCoder ABC154F - Many Many Paths
//    链接：https://atcoder.jp/contests/abc154/tasks/abc154_f
//    题意：计算树上路径数量，可以使用虚树优化
```

```
//
// 7. 洛谷 P4103 - [HEOI2014]大工程
//     链接：https://www.luogu.com.cn/problem/P4103
//     题意：给一棵树和多个询问，每个询问给出一些关键点，要求计算所有关键点对之间距离的和、最小值
和最大值
//
// 8. 洛谷 P3233 - [HNOI2014]世界树
//     链接：https://www.luogu.com.cn/problem/P3233
//     题意：给一棵树和多个询问，每个询问给出一些关键点，要求计算每个关键点能管理多少个点
//
// 9. 洛谷 P3320 - [SDOI2015]寻宝游戏
//     链接：https://www.luogu.com.cn/problem/P3320
//     题意：给一棵树和多个操作，每次操作翻转一个点的状态，求收集所有宝藏的最短路径长度
//
// 10. 洛谷 P5327 - [ZJOI2019]语言
//     链接：https://www.luogu.com.cn/problem/P5327
//     题意：涉及树上路径覆盖的复杂问题
//
// 11. SPOJ QTREE5 - Query on a tree V
//     链接：https://www.spoj.com/problems/QTREE5/
//     题意：树上点颜色修改和查询距离最近的白色节点
//
// 12. 洛谷 P3232 - [HNOI2013]游走
//     链接：https://www.luogu.com.cn/problem/P3232
//     题意：给定无向连通图，通过高斯消元计算边的期望经过次数，再贪心编号使总得分期望最小
//
// 时间复杂度分析：
// 1. 预处理（DFS 序、LCA）：O(n log n)
// 2. 构造虚树：O(k log k)
// 3. 在虚树上 DP：O(k)
// 总体复杂度：O(n log n + ∑k log k)
//
// 空间复杂度：O(n + k)
//
// 工程化考量：
// 1. 注意虚树边通常没有边权，需要通过原树计算
// 2. 清空关键点标记时避免使用 memset，用 for 循环逐个清除
// 3. 排序后的关键点顺序不是原节点序，如需按原序输出需额外保存
// 4. 虚树主要用于卡常题，需注意常数优化
//
// 算法设计本质与核心思想：
// 1. 设计动机：虚树算法的核心动机是优化树上多次询问问题。当需要对树上不同关键点集合进行多次查询
时，
```

```
//     如果每次都遍历整棵树，时间复杂度会很高。虚树通过只保留关键点及其 LCA，将问题规模从 O(n)降低到 O(k)。
// 2. 数学原理：
//     - LCA 性质：任意两个节点的 LCA 在 DFS 序上具有特定性质，可以用于构建虚树
//     - 节点数量上界：虚树节点数不超过 2*k-1，这是通过数学归纳法可以证明的
//     - 树的结构保持：虚树保持了原树中关键点之间的祖先关系
// 3. 与其它算法的关联：
//     - 树上倍增：虚树构建需要 LCA，通常使用树上倍增算法
//     - 树形 DP：虚树上的动态规划是解决问题的核心
//     - 单调栈：构建虚树时使用的单调栈技巧与其它算法中的单调栈类似
// 4. 工程化应用：
//     - 内存优化：避免使用全局数组清零，用循环逐个清除
//     - 常数优化：选择合适的虚树构建方法（单调栈法通常更快）
//     - 边界处理：正确处理根节点、叶子节点等特殊情况
//
// 语言特性差异与跨语言实现：
// 1. Java 实现特点：
//     - 使用对象封装，代码结构清晰
//     - 自定义 FastReader 提高输入效率
//     - 递归深度可能受限，需要改用迭代实现
// 2. C++实现特点：
//     - 性能最优，适合大数据量
//     - 需要注意编译环境问题，避免使用复杂 STL
//     - 指针操作灵活但需谨慎
// 3. Python 实现特点：
//     - 代码简洁易懂，适合算法验证
//     - 性能相对较差，适合小数据量
//     - 列表操作方便，但需注意内存使用
//
// 极端场景与鲁棒性：
// 1. 空输入处理：关键点为空时的特殊处理
// 2. 极端数据规模：关键点数量接近节点总数、树退化为链的情况、深度很大的树结构
// 3. 边界条件：关键点包含根节点、关键点之间存在父子关系、关键点相邻的情况
//
// 性能优化策略：
// 1. 算法层面优化：选择合适的虚树构建方法、优化 DP 状态转移方程、预处理减少重复计算
// 2. 实现层面优化：减少函数调用开销、优化内存访问模式、使用位运算等底层优化技巧
// 3. 工程层面优化：输入输出优化、内存池技术、缓存友好设计
//
// 调试技巧与问题定位：
// 1. 中间过程打印：打印 DFS 序、打印 LCA 计算结果、打印虚树构建过程
// 2. 断言验证：验证虚树节点数量上界、验证关键点标记正确性、验证 DP 状态转移正确性
// 3. 性能分析：使用性能分析工具定位瓶颈、对比不同实现的性能差异、分析时间复杂度常数项影响
```

```java
// 树上病毒传播，java版
// 一共有n个城市，有n-1条无向边，所有城市组成一棵树，一共有q条查询，每条查询数据如下
// 首先给定k种病毒，每种病毒有初次感染的城市start[i]，还有传播速度speed[i]
// 然后给定m个关键城市，打印每个城市被第几号病毒感染，病毒传播的规则如下
// 病毒的传播按轮次进行，每一轮病毒1先传播，然后是病毒2 .. 直到病毒k，下一轮又从病毒1开始
// 如果第i种病毒已经感染了城市x，当自己传播时，想要感染城市y的条件如下
// 城市x到城市y的路径包含的边数<=speed[i]，城市x到城市y的路径上，除了x所有城市都未被感染
// 一旦城市被某种病毒感染就永久保持，不会再被其他病毒感染，传播一直持续，直到所有城市都被感染
// 1 <= n、q、所有查询病毒总数、所有查询关键城市总数 <= 2 * 10^5
// 测试链接 : https://www.luogu.com.cn/problem/CF1320E
// 测试链接 : https://codeforces.com/problemset/problem/1320/E
// 提交以下的code，提交时请把类名改成"Main"，可以通过所有测试用例

import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStreamWriter;
import java.io.PrintWriter;
import java.util.Comparator;
import java.util.PriorityQueue;

public class Code05_TreelandAndViruses1 {

    static class Node {
        int id, dist, time, virus;

        Node(int id_, int dist_, int time_, int virus_) {
            id = id_;
            dist = dist_;
            time = time_;
            virus = virus_;
        }
    }

    static class NodeCmp implements Comparator<Node> {
        @Override
        public int compare(Node o1, Node o2) {
            if (o1.time != o2.time) {
                return o1.time - o2.time;
            }
            return o1.virus - o2.virus;
        }
    }
```

```
public static int MAXN = 200001;
public static int MAXP = 20;
public static int n, q, k, m;

public static int[] headg = new int[MAXN];
public static int[] nextg = new int[MAXN << 1];
public static int[] tog = new int[MAXN << 1];
public static int cntg;

public static int[] headv = new int[MAXN];
public static int[] nextv = new int[MAXN << 1];
public static int[] tov = new int[MAXN << 1];
public static int cntv;

public static int[] dep = new int[MAXN];
public static int[] dfn = new int[MAXN];
public static int[][] stjump = new int[MAXN][MAXP];
public static int cntd;

public static int[] start = new int[MAXN];
public static int[] speed = new int[MAXN];
public static int[] query = new int[MAXN];

public static int[] arr = new int[MAXN << 1];
public static int[] tmp = new int[MAXN << 2];
public static int len;

public static PriorityQueue<Node> heap = new PriorityQueue<>(new NodeCmp());
public static boolean[] vis = new boolean[MAXN];
public static int[] minTime = new int[MAXN];
public static int[] findVirus = new int[MAXN];

public static void addEdgeG(int u, int v) {
    nextg[++cntg] = headg[u];
    tog[cntg] = v;
    headg[u] = cntg;
}

public static void addEdgeV(int u, int v) {
    nextv[++cntv] = headv[u];
    tov[cntv] = v;
    headv[u] = cntv;
```

```java
}

public static void sortByDfn(int[] nums, int l, int r) {
    if (l >= r) return;
    int i = l, j = r;
    int pivot = nums[(l + r) >> 1];
    while (i <= j) {
        while (dfn[nums[i]] < dfn[pivot]) i++;
        while (dfn[nums[j]] > dfn[pivot]) j--;
        if (i <= j) {
            int tmp = nums[i]; nums[i] = nums[j]; nums[j] = tmp;
            i++; j--;
        }
    }
    sortByDfn(nums, l, j);
    sortByDfn(nums, i, r);
}

public static void dfs(int u, int fa) {
    dep[u] = dep[fa] + 1;
    dfn[u] = ++cntd;
    stjump[u][0] = fa;
    for (int p = 1; p < MAXP; p++) {
        stjump[u][p] = stjump[stjump[u][p - 1]][p - 1];
    }
    for (int e = headg[u]; e > 0; e = nextg[e]) {
        if (tog[e] != fa) {
            dfs(tog[e], u);
        }
    }
}

public static int getLca(int a, int b) {
    if (dep[a] < dep[b]) {
        int tmp = a; a = b; b = tmp;
    }
    for (int p = MAXP - 1; p >= 0; p--) {
        if (dep[stjump[a][p]] >= dep[b]) {
            a = stjump[a][p];
        }
    }
    if (a == b) {
        return a;
```

```java
    }
    for (int p = MAXP - 1; p >= 0; p--) {
        if (stjump[a][p] != stjump[b][p]) {
            a = stjump[a][p];
            b = stjump[b][p];
        }
    }
    return stjump[a][0];
}

public static int buildVirtualTree() {
    int tot = 0;
    for (int i = 1; i <= k; i++) {
        arr[++tot] = start[i];
    }
    for (int i = 1; i <= m; i++) {
        arr[++tot] = query[i];
    }
    sortByDfn(arr, 1, tot);
    len = 0;
    for (int i = 1; i < tot; i++) {
        tmp[++len] = arr[i];
        tmp[++len] = getLca(arr[i], arr[i + 1]);
    }
    tmp[++len] = arr[tot];
    sortByDfn(tmp, 1, len);
    int unique = 1;
    for (int i = 2; i <= len; i++) {
        if (tmp[unique] != tmp[i]) {
            tmp[++unique] = tmp[i];
        }
    }
    cntv = 0;
    for (int i = 1; i <= unique; i++) {
        headv[tmp[i]] = 0;
    }
    for (int i = 1; i < unique; i++) {
        // 虚树的边不是单向的，是双向的
        // 因为病毒感染既可以向上也可以向下
        int lca = getLca(tmp[i], tmp[i + 1]);
        addEdgeV(lca, tmp[i + 1]);
        addEdgeV(tmp[i + 1], lca);
    }
```

```
        len = unique;
        return tmp[1];
    }

    public static void dijkstra() {
        for (int i = 1; i <= len; i++) {
            int u = tmp[i];
            minTime[u] = n + 1;
            findVirus[u] = k + 1;
            vis[u] = false;
        }
        for (int i = 1; i <= k; i++) {
            int s = start[i];
            minTime[s] = 0;
            findVirus[s] = i;
            heap.add(new Node(s, 0, 0, i));
        }
        while (!heap.isEmpty()) {
            Node cur = heap.poll();
            int u = cur.id;
            int udist = cur.dist;
            int uvirus = cur.virus;
            if (!vis[u]) {
                vis[u] = true;
                for (int e = headv[u]; e > 0; e = nextv[e]) {
                    int v = tov[e];
                    if (!vis[v]) {
                        int vdist = udist + Math.abs(dep[u] - dep[v]);
                        int vtime = (vdist + speed[uvirus] - 1) / speed[uvirus];
                        if (vtime < minTime[v] || (vtime == minTime[v] && uvirus < findVirus[v])) {
                            minTime[v] = vtime;
                            findVirus[v] = uvirus;
                            heap.add(new Node(v, vdist, vtime, uvirus));
                        }
                    }
                }
            }
        }
    }

    public static void main(String[] args) throws Exception {
        FastReader in = new FastReader(System.in);
        PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));
```

```java
        n = in.nextInt();
        for (int i = 1; i < n; i++) {
            int x = in.nextInt();
            int y = in.nextInt();
            addEdgeG(x, y);
            addEdgeG(y, x);
        }
        dfs(1, 0);
        q = in.nextInt();
        for (int t = 1; t <= q; t++) {
            k = in.nextInt();
            m = in.nextInt();
            for (int i = 1; i <= k; i++) {
                start[i] = in.nextInt();
                speed[i] = in.nextInt();
            }
            for (int i = 1; i <= m; i++) {
                query[i] = in.nextInt();
            }
            buildVirtualTree();
            dijkstra();
            for (int i = 1; i <= m; i++) {
                out.print(findVirus[query[i]] + " ");
            }
            out.println();
        }
        out.flush();
        out.close();
    }

    // 读写工具类
    static class FastReader {
        private final byte[] buffer = new byte[1 << 16];
        private int ptr = 0, len = 0;
        private final InputStream in;

        FastReader(InputStream in) {
            this.in = in;
        }

        private int readByte() throws IOException {
            if (ptr >= len) {
                len = in.read(buffer);
```

```
            ptr = 0;
            if (len <= 0)
                return -1;
        }
        return buffer[ptr++];
    }

    int nextInt() throws IOException {
        int c;
        do {
            c = readByte();
        } while (c <= ' ' && c != -1);
        boolean neg = false;
        if (c == '-') {
            neg = true;
            c = readByte();
        }
        int val = 0;
        while (c > ' ' && c != -1) {
            val = val * 10 + (c - '0');
            c = readByte();
        }
        return neg ? -val : val;
    }
}

}
```

==

文件: Code05_TreelandAndViruses1.py

==

```
# 虚树(Virtual Tree)算法详解与应用
#
# 虚树是一种优化技术，用于解决树上多次询问的问题，每次询问涉及部分关键点
# 虚树只保留关键点及其两两之间的 LCA，节点数控制在 O(k)级别，从而提高效率
#
# 算法核心思想：
# 1. 虚树包含所有关键点和它们两两之间的 LCA
# 2. 虚树的节点数不超过 2*k-1（k 为关键点数）
# 3. 在虚树上进行 DP 等操作，避免遍历整棵树
#
# 构造方法：
```

```
# 方法一：二次排序法
# 1. 将关键点按 DFS 序排序
# 2. 相邻点求 LCA 并加入序列
# 3. 再次排序并去重得到虚树所有节点
# 4. 按照父子关系连接节点
#
# 方法二：单调栈法
# 1. 将关键点按 DFS 序排序
# 2. 用栈维护虚树的一条链
# 3. 逐个插入关键点并维护栈结构
#
# 应用场景：
# 1. 树上多次询问，每次询问涉及部分关键点
# 2. 需要在关键点及其 LCA 上进行 DP 等操作
# 3. 数据范围要求∑k 较小（通常≤10^5）
#
# 相关题目：
# 1. Codeforces 1109D - Treeland and Viruses
#    题意：给一棵树和多个病毒源点，每个病毒源点以不同速度扩散，求每个点被哪个病毒源点感染
#
# 2. Codeforces 613D - Kingdom and Cities
#    链接：https://codeforces.com/problemset/problem/613/D
#    题意：给一棵树和多个询问，每个询问给出一些关键点，要求切断最少的非关键点使关键点两两不连通
#
# 3. 洛谷 P2495 - [SDOI2011]消耗战
#    链接：https://www.luogu.com.cn/problem/P2495
#    题意：给一棵树和多个询问，每个询问给出一些关键点，要求切断最少代价的边使关键点都无法到达根节点
#
# 时间复杂度分析：
# 1. 预处理（DFS 序、LCA）：O(n log n)
# 2. 构造虚树：O(k log k)
# 3. 在虚树上 DP：O(k)
# 总体复杂度：O(n log n + ∑k log k)
#
# 空间复杂度：O(n + k)
#
# 工程化考量：
# 1. 注意虚树边通常没有边权，需要通过原树计算
# 2. 清空关键点标记时避免使用 memset，用 for 循环逐个清除
# 3. 排序后的关键点顺序不是原节点序，如需按原序输出需额外保存
# 4. 虚树主要用于卡常题，需注意常数优化
```

```
# 树上病毒扩散，Python版
# 有一个n个节点的树，有k个病毒源点，每个病毒源点i有一个扩散速度speed[i]
# 病毒同时从所有源点开始扩散，每秒扩散speed[i]步
# 当一个点被多个病毒同时到达时，编号小的病毒获胜
# 求每个点被哪个病毒源点感染
# 1 <= n <= 2*10^5
# 1 <= k <= n
# 1 <= speed[i] <= 10^9

import sys
from collections import deque

MAXN = 200001
MAXP = 20

# 全局变量
n, k = 0, 0

# 原始树
headg = [0] * MAXN
nextg = [0] * (MAXN << 1)
tog = [0] * (MAXN << 1)
cntg = 0

# 虚树
headv = [0] * MAXN
nextv = [0] * MAXN
tov = [0] * MAXN
cntv = 0

# 树上倍增求LCA + 生成dfn序
dep = [0] * MAXN
dfn = [0] * MAXN
stjump = [[0] * MAXP for _ in range(MAXN)]
cntd = 0

# 关键点数组
arr = [0] * MAXN
speed = [0] * MAXN
# 标记节点是否是关键点
isKey = [False] * MAXN

# 第一种建树方式
```

```python
tmp = [0] * (MAXN << 1)
# 第二种建树方式
stk = [0] * MAXN

# 动态规划相关
# owner[u]表示节点 u 被哪个病毒源点感染，0 表示未被感染
owner = [0] * MAXN
dist = [0] * MAXN

# 原始树连边
def addEdgeG(u, v):
    global cntg
    cntg += 1
    nextg[cntg] = headg[u]
    tog[cntg] = v
    headg[u] = cntg

# 虚树连边
def addEdgeV(u, v):
    global cntv
    cntv += 1
    nextv[cntv] = headv[u]
    tov[cntv] = v
    headv[u] = cntv

# nums 中的数，根据 dfn 的大小排序，手撸双指针快排
def sortByDfn(nums, l, r):
    if l >= r:
        return
    i, j = l, r
    pivot = nums[(l + r) >> 1]
    while i <= j:
        while dfn[nums[i]] < dfn[pivot]:
            i += 1
        while dfn[nums[j]] > dfn[pivot]:
            j -= 1
        if i <= j:
            nums[i], nums[j] = nums[j], nums[i]
            i += 1
            j -= 1
    sortByDfn(nums, l, j)
    sortByDfn(nums, i, r)
```

```python
# 树上倍增的dfs过程
def dfs(u, fa):
    global cntd
    dep[u] = dep[fa] + 1
    cntd += 1
    dfn[u] = cntd
    stjump[u][0] = fa
    for p in range(1, MAXP):
        stjump[u][p] = stjump[stjump[u][p - 1]][p - 1]
    e = headg[u]
    while e > 0:
        if tog[e] != fa:
            dfs(tog[e], u)
        e = nextg[e]

# 返回a和b的最低公共祖先
def getLca(a, b):
    if dep[a] < dep[b]:
        a, b = b, a
    for p in range(MAXP - 1, -1, -1):
        if dep[stjump[a][p]] >= dep[b]:
            a = stjump[a][p]
    if a == b:
        return a
    for p in range(MAXP - 1, -1, -1):
        if stjump[a][p] != stjump[b][p]:
            a = stjump[a][p]
            b = stjump[b][p]
    return stjump[a][0]

# 二次排序 + LCA连边的方式建立虚树
def buildVirtualTree1():
    sortByDfn(arr, 1, k)
    len_idx = 0
    for i in range(1, k):
        len_idx += 1
        tmp[len_idx] = arr[i]
        len_idx += 1
        tmp[len_idx] = getLca(arr[i], arr[i + 1])
    len_idx += 1
    tmp[len_idx] = arr[k]
    sortByDfn(tmp, 1, len_idx)
    unique = 1
```

```python
    for i in range(2, len_idx + 1):
        if tmp[unique] != tmp[i]:
            unique += 1
            tmp[unique] = tmp[i]
    global cntv
    cntv = 0
    for i in range(1, unique + 1):
        headv[tmp[i]] = 0
    for i in range(1, unique):
        addEdgeV(getLca(tmp[i], tmp[i + 1]), tmp[i + 1])
    return tmp[1]

# 单调栈的方式建立虚树
def buildVirtualTree2():
    sortByDfn(arr, 1, k)
    global cntv
    cntv = 0
    headv[arr[1]] = 0
    top = 0
    top += 1
    stk[top] = arr[1]
    for i in range(2, k + 1):
        x = arr[i]
        y = stk[top]
        lca = getLca(x, y)
        while top > 1 and dfn[stk[top - 1]] >= dfn[lca]:
            addEdgeV(stk[top - 1], stk[top])
            top -= 1
        if lca != stk[top]:
            headv[lca] = 0
            addEdgeV(lca, stk[top])
            stk[top] = lca
        headv[x] = 0
        top += 1
        stk[top] = x
    while top > 1:
        addEdgeV(stk[top - 1], stk[top])
        top -= 1
    return stk[1]

# 树型 dp 的过程
def dp(u):
    # 初始化
```

```
    if isKey[u]:
        owner[u] = u
        dist[u] = 0
    else:
        owner[u] = 0
        dist[u] = -1
    # 处理子树
    e = headv[u]
    while e > 0:
        v = tov[e]
        dp(v)
        # 更新u的状态
        if owner[v] != 0:
            # 计算从v的病毒源点到u的距离
            d = dist[v] + dep[v] + dep[u] - 2 * dep[getLca(v, u)]
            if owner[u] == 0 or d < dist[u] or (d == dist[u] and owner[v] < owner[u]):
                owner[u] = owner[v]
                dist[u] = d
        e = nextv[e]

def compute():
    # 节点标记关键点信息
    for i in range(1, k + 1):
        isKey[arr[i]] = True
    tree = buildVirtualTree1()
    # tree = buildVirtualTree2()
    dp(tree)
    # 输出结果
    result = []
    for i in range(1, n + 1):
        result.append(str(owner[i]))
    # 节点撤销关键点信息
    for i in range(1, k + 1):
        isKey[arr[i]] = False
    return " ".join(result)

# 主函数
if __name__ == "__main__":
    # 读取输入
    n = int(input())
    for i in range(1, n):
        u, v = map(int, input().split())
        addEdgeG(u, v)
```

```
        addEdgeG(v, u)
    dfs(1, 0)
    k = int(input())
    arr_values = list(map(int, input().split()))
    speed_values = list(map(int, input().split()))
    for i in range(1, k + 1):
        arr[i] = arr_values[i - 1]
        speed[i] = speed_values[i - 1]
    print(compute())
```

==

文件: Code05_TreelandAndViruses2.java
==

```
package class180;

// 树上病毒传播，C++版
// 一共有 n 个城市，有 n-1 条无向边，所有城市组成一棵树，一共有 q 条查询，每条查询数据如下
// 首先给定 k 种病毒，每种病毒有初次感染的城市 start[i]，还有传播速度 speed[i]
// 然后给定 m 个关键城市，打印每个城市被第几号病毒感染，病毒传播的规则如下
// 病毒的传播按轮次进行，每一轮病毒 1 先传播，然后是病毒 2 .. 直到病毒 k，下一轮又从病毒 1 开始
// 如果第 i 种病毒已经感染了城市 x，当自己传播时，想要感染城市 y 的条件如下
// 城市 x 到城市 y 的路径包含的边数<=speed[i]，城市 x 到城市 y 的路径上，除了 x 所有城市都未被感染
// 一旦城市被某种病毒感染就永久保持，不会再被其他病毒感染，传播一直持续，直到所有城市都被感染
// 1 <= n、q、所有查询病毒总数、所有查询关键城市总数 <= 2 * 10^5
// 测试链接 : https://www.luogu.com.cn/problem/CF1320E
// 测试链接 : https://codeforces.com/problemset/problem/1320/E
// 如下实现是 C++的版本，C++版本和 java 版本逻辑完全一样
// 提交如下代码，可以通过所有测试用例

//#include <bits/stdc++.h>
//
//using namespace std;
//
//struct Node {
//    int id, dist, time, virus;
//    bool operator<(const Node &other) const {
//        if (time != other.time) {
//            return time > other.time;
//        }
//        return virus > other.virus;
//    }
//};
```

```
//
//const int MAXN = 200001;
//const int MAXP = 20;
//int n, q, k, m;
//
//int headg[MAXN];
//int nextg[MAXN << 1];
//int tog[MAXN << 1];
//int cntg;
//
//int headv[MAXN];
//int nextv[MAXN << 1];
//int tov[MAXN << 1];
//int cntv;
//
//int dep[MAXN];
//int dfn[MAXN];
//int stjump[MAXN][MAXP];
//int cntd;
//
//int start[MAXN];
//int speed[MAXN];
//int query[MAXN];
//
//int arr[MAXN << 1];
//int tmp[MAXN << 2];
//int len;
//
//priority_queue<Node> heap;
//bool vis[MAXN];
//int minTime[MAXN];
//int findVirus[MAXN];
//
//bool cmp(int x, int y) {
//    return dfn[x] < dfn[y];
//}
//
//void addEdgeG(int u, int v) {
//    nextg[++cntg] = headg[u];
//    tog[cntg] = v;
//    headg[u] = cntg;
//}
//
```

```
//void addEdgeV(int u, int v) {
//    nextv[++cntv] = headv[u];
//    tov[cntv] = v;
//    headv[u] = cntv;
//}
//
//void dfs(int u, int fa) {
//    dep[u] = dep[fa] + 1;
//    dfn[u] = ++cntd;
//    stjump[u][0] = fa;
//    for (int p = 1; p < MAXP; p++) {
//        stjump[u][p] = stjump[stjump[u][p - 1]][p - 1];
//    }
//    for (int e = headg[u]; e; e = nextg[e]) {
//        if (tog[e] != fa) {
//            dfs(tog[e], u);
//        }
//    }
//}
//
//int getLca(int a, int b) {
//    if (dep[a] < dep[b]) {
//        swap(a, b);
//    }
//    for (int p = MAXP - 1; p >= 0; p--) {
//        if (dep[stjump[a][p]] >= dep[b]) {
//            a = stjump[a][p];
//        }
//    }
//    if (a == b) {
//        return a;
//    }
//    for (int p = MAXP - 1; p >= 0; p--) {
//        if (stjump[a][p] != stjump[b][p]) {
//            a = stjump[a][p];
//            b = stjump[b][p];
//        }
//    }
//    return stjump[a][0];
//}
//
//int buildVirtualTree() {
//    int tot = 0;
```

```
//    for (int i = 1; i <= k; i++) {
//        arr[++tot] = start[i];
//    }
//    for (int i = 1; i <= m; i++) {
//        arr[++tot] = query[i];
//    }
//    sort(arr + 1, arr + tot + 1, cmp);
//    len = 0;
//    for (int i = 1; i < tot; i++) {
//        tmp[++len] = arr[i];
//        tmp[++len] = getLca(arr[i], arr[i + 1]);
//    }
//    tmp[++len] = arr[tot];
//    sort(tmp + 1, tmp + len + 1, cmp);
//    int unique = 1;
//    for (int i = 2; i <= len; i++) {
//        if (tmp[unique] != tmp[i]) {
//            tmp[++unique] = tmp[i];
//        }
//    }
//    cntv = 0;
//    for (int i = 1; i <= unique; i++) {
//        headv[tmp[i]] = 0;
//    }
//    for (int i = 1; i < unique; i++) {
//        int lca = getLca(tmp[i], tmp[i + 1]);
//        addEdgeV(lca, tmp[i + 1]);
//        addEdgeV(tmp[i + 1], lca);
//    }
//    len = unique;
//    return tmp[1];
//}
//
//void dijkstra() {
//    for (int i = 1; i <= len; i++) {
//        int u = tmp[i];
//        minTime[u] = n + 1;
//        findVirus[u] = k + 1;
//        vis[u] = false;
//    }
//    for (int i = 1; i <= k; i++) {
//        int s = start[i];
//        minTime[s] = 0;
```

```
//          findVirus[s] = i;
//          heap.push(Node{s, 0, 0, i});
//      }
//      while (!heap.empty()) {
//          Node cur = heap.top();
//          heap.pop();
//          int u = cur.id;
//          int udist = cur.dist;
//          int uvirus = cur.virus;
//          if (!vis[u]) {
//              vis[u] = true;
//              for (int e = headv[u]; e; e = nextv[e]) {
//                  int v = tov[e];
//                  if (!vis[v]) {
//                      int vdist = udist + abs(dep[u] - dep[v]);
//                      int vtime = (vdist + speed[uvirus] - 1) / speed[uvirus];
//                      if (vtime < minTime[v] || (vtime == minTime[v] && uvirus < findVirus[v])) {
//                          minTime[v] = vtime;
//                          findVirus[v] = uvirus;
//                          heap.push(Node{v, vdist, vtime, uvirus});
//                      }
//                  }
//              }
//          }
//      }
//}
//
//int main() {
//      ios::sync_with_stdio(false);
//      cin.tie(nullptr);
//      cin >> n;
//      for (int i = 1, u, v; i < n; i++) {
//          cin >> u >> v;
//          addEdgeG(u, v);
//          addEdgeG(v, u);
//      }
//      dfs(1, 0);
//      cin >> q;
//      for (int t = 1; t <= q; t++) {
//          cin >> k >> m;
//          for (int i = 1, s, v; i <= k; i++) {
//              cin >> start[i] >> speed[i];
//          }
```

```
//        for (int i = 1; i <= m; i++) {
//            cin >> query[i];
//        }
//        buildVirtualTree();
//        dijkstra();
//        for (int i = 1; i <= m; i++) {
//            cout << findVirus[query[i]] << ' ';
//        }
//        cout << '\n';
//    }
//    return 0;
//}
```

==

文件: Code06_TreasureHunt1.cpp

==

```
// 虚树(Virtual Tree)算法详解与应用
//
// 虚树是一种优化技术，用于解决树上多次询问的问题，每次询问涉及部分关键点
// 虚树只保留关键点及其两两之间的LCA，节点数控制在O(k)级别，从而提高效率
//
// 算法核心思想：
// 1. 虚树包含所有关键点和它们两两之间的LCA
// 2. 虚树的节点数不超过2*k-1（k为关键点数）
// 3. 在虚树上进行DP等操作，避免遍历整棵树
//
// 构造方法：
// 方法一：二次排序法
// 1. 将关键点按DFS序排序
// 2. 相邻点求LCA并加入序列
// 3. 再次排序并去重得到虚树所有节点
// 4. 按照父子关系连接节点
//
// 方法二：单调栈法
// 1. 将关键点按DFS序排序
// 2. 用栈维护虚树的一条链
// 3. 逐个插入关键点并维护栈结构
//
// 应用场景：
// 1. 树上多次询问，每次询问涉及部分关键点
// 2. 需要在关键点及其LCA上进行DP等操作
// 3. 数据范围要求Σk较小（通常≤10^5）
```

```
//
// 相关题目：
// 1. 宝藏猎人问题
//    题意：给一棵树和多个宝藏点，求收集所有宝藏的最短路径长度
//
// 2. Codeforces 613D - Kingdom and Cities
//    链接：https://codeforces.com/problemset/problem/613/D
//    题意：给一棵树和多个询问，每个询问给出一些关键点，要求切断最少的非关键点使关键点两两不连通
//
// 3. 洛谷 P2495 - [SDOI2011]消耗战
//    链接：https://www.luogu.com.cn/problem/P2495
//    题意：给一棵树和多个询问，每个询问给出一些关键点，要求切断最少代价的边使关键点都无法到达根
节点
//
// 时间复杂度分析：
// 1. 预处理（DFS 序、LCA）：O(n log n)
// 2. 构造虚树：O(k log k)
// 3. 在虚树上 DP：O(k)
// 总体复杂度：O(n log n + Σk log k)
//
// 空间复杂度：O(n + k)
//
// 工程化考量：
// 1. 注意虚树边通常没有边权，需要通过原树计算
// 2. 清空关键点标记时避免使用 memset，用 for 循环逐个清除
// 3. 排序后的关键点顺序不是原节点序，如需按原序输出需额外保存
// 4. 虚树主要用于卡常题，需注意常数优化

// 宝藏猎人，C++版（简化版，避免标准库问题）
// 有一个 n 个节点的树，边权为 1
// 有 k 个宝藏点，求从节点 1 出发收集所有宝藏并回到节点 1 的最短路径长度
// 1 <= n <= 2*10^5
// 1 <= k <= n

const int MAXN = 200001;
const int MAXP = 20;

int n, k;

// 原始树
int headg[MAXN], nextg[MAXN << 1], tog[MAXN << 1], cntg;

// 虚树
```

```
int headv[MAXN], nextv[MAXN], tov[MAXN], cntv;

// 树上倍增求 LCA + 生成 dfn 序
int dep[MAXN], dfn[MAXN], stjump[MAXN][MAXP], cntd;

// 关键点数组
int arr[MAXN];
// 标记节点是否是关键点
bool isKey[MAXN];

// 第一种建树方式
int tmp[MAXN << 1];
// 第二种建树方式
int stk[MAXN];

// 动态规划相关
// dp[u]表示在虚树中以 u 为根的子树中，收集所有宝藏并回到 u 的最短路径长度
int dp[MAXN];

// 原始树连边
void addEdgeG(int u, int v) {
    cntg++;
    nextg[cntg] = headg[u];
    tog[cntg] = v;
    headg[u] = cntg;
}

// 虚树连边
void addEdgeV(int u, int v) {
    cntv++;
    nextv[cntv] = headv[u];
    tov[cntv] = v;
    headv[u] = cntv;
}

// nums 中的数，根据 dfn 的大小排序，手撸双指针快排
void sortByDfn(int nums[], int l, int r) {
    if (l >= r) return;
    int i = l, j = r;
    int pivot = nums[(l + r) >> 1];
    while (i <= j) {
        while (dfn[nums[i]] < dfn[pivot]) i++;
        while (dfn[nums[j]] > dfn[pivot]) j--;
```

```
        if (i <= j) {
            int t = nums[i]; nums[i] = nums[j]; nums[j] = t;
            i++; j--;
        }
    }
    sortByDfn(nums, l, j);
    sortByDfn(nums, i, r);
}

// 树上倍增的 dfs 过程
void dfs(int u, int fa) {
    dep[u] = dep[fa] + 1;
    cntd++;
    dfn[u] = cntd;
    stjump[u][0] = fa;
    for (int p = 1; p < MAXP; p++) {
        stjump[u][p] = stjump[stjump[u][p - 1]][p - 1];
    }
    for (int e = headg[u]; e > 0; e = nextg[e]) {
        if (tog[e] != fa) {
            dfs(tog[e], u);
        }
    }
}

// 返回 a 和 b 的最低公共祖先
int getLca(int a, int b) {
    if (dep[a] < dep[b]) {
        int t = a; a = b; b = t;
    }
    for (int p = MAXP - 1; p >= 0; p--) {
        if (dep[stjump[a][p]] >= dep[b]) {
            a = stjump[a][p];
        }
    }
    if (a == b) {
        return a;
    }
    for (int p = MAXP - 1; p >= 0; p--) {
        if (stjump[a][p] != stjump[b][p]) {
            a = stjump[a][p];
            b = stjump[b][p];
        }
```

```
    }
    return stjump[a][0];
}

// 二次排序 + LCA 连边的方式建立虚树
int buildVirtualTree1() {
    sortByDfn(arr, 1, k);
    int len = 0;
    for (int i = 1; i < k; i++) {
        len++;
        tmp[len] = arr[i];
        len++;
        tmp[len] = getLca(arr[i], arr[i + 1]);
    }
    len++;
    tmp[len] = arr[k];
    sortByDfn(tmp, 1, len);
    int unique = 1;
    for (int i = 2; i <= len; i++) {
        if (tmp[unique] != tmp[i]) {
            unique++;
            tmp[unique] = tmp[i];
        }
    }
    cntv = 0;
    for (int i = 1; i <= unique; i++) {
        headv[tmp[i]] = 0;
    }
    for (int i = 1; i < unique; i++) {
        addEdgeV(getLca(tmp[i], tmp[i + 1]), tmp[i + 1]);
    }
    return tmp[1];
}

// 单调栈的方式建立虚树
int buildVirtualTree2() {
    sortByDfn(arr, 1, k);
    cntv = 0;
    headv[arr[1]] = 0;
    int top = 0;
    top++;
    stk[top] = arr[1];
    for (int i = 2; i <= k; i++) {
```

```
        int x = arr[i];
        int y = stk[top];
        int lca = getLca(x, y);
        while (top > 1 && dfn[stk[top - 1]] >= dfn[lca]) {
            addEdgeV(stk[top - 1], stk[top]);
            top--;
        }
        if (lca != stk[top]) {
            headv[lca] = 0;
            addEdgeV(lca, stk[top]);
            stk[top] = lca;
        }
        headv[x] = 0;
        top++;
        stk[top] = x;
    }
    while (top > 1) {
        addEdgeV(stk[top - 1], stk[top]);
        top--;
    }
    return stk[1];
}

// 树型 dp 的过程
void treedp(int u) {
    dp[u] = 0;
    for (int e = headv[u]; e > 0; e = nextv[e]) {
        int v = tov[e];
        treedp(v);
        // 从 u 到 v 再回到 u 的距离是 2 * distance(u, v)
        dp[u] += dp[v] + 2 * (dep[v] - dep[u]);
    }
}

int compute() {
    // 节点标记关键点信息
    for (int i = 1; i <= k; i++) {
        isKey[arr[i]] = true;
    }
    // 如果节点 1 不是关键点，也要加入
    bool need_add = !isKey[1];
    if (need_add) {
        k++;
```

```
        arr[k] = 1;
        isKey[1] = true;
    }
    int tree = buildVirtualTree1();
    // int tree = buildVirtualTree2();
    treedp(tree);
    int result = dp[tree];
    // 节点撤销关键点信息
    for (int i = 1; i <= k; i++) {
        isKey[arr[i]] = false;
    }
    if (need_add) {
        k--;
    }
    return result;
}

// 由于编译环境问题，这里不包含 main 函数
// 在实际使用中，需要添加适当的输入输出函数
```

==

文件: Code06_TreasureHunt1.java

==

```
package class180;

// 虚树(Virtual Tree)算法详解与应用
//
// 虚树是一种优化技术，用于解决树上多次询问的问题，每次询问涉及部分关键点
// 虚树只保留关键点及其两两之间的 LCA，节点数控制在 O(k)级别，从而提高效率
//
// 算法核心思想：
// 1. 虚树包含所有关键点和它们两两之间的 LCA
// 2. 虚树的节点数不超过 2*k-1（k 为关键点数）
// 3. 在虚树上进行 DP 等操作，避免遍历整棵树
//
// 构造方法：
// 方法一：二次排序法
// 1. 将关键点按 DFS 序排序
// 2. 相邻点求 LCA 并加入序列
// 3. 再次排序并去重得到虚树所有节点
// 4. 按照父子关系连接节点
//
```

```
// 方法二：单调栈法
// 1. 将关键点按 DFS 序排序
// 2. 用栈维护虚树的一条链
// 3. 逐个插入关键点并维护栈结构
//
// 应用场景：
// 1. 树上多次询问，每次询问涉及部分关键点
// 2. 需要在关键点及其 LCA 上进行 DP 等操作
// 3. 数据范围要求Σk 较小（通常≤10^5）
//
// 相关题目：
// 1. 洛谷 P3320 - [SDOI2015]寻宝游戏
//    链接：https://www.luogu.com.cn/problem/P3320
//    题意：给一棵树和多个操作，每次操作翻转一个点的状态，求收集所有宝藏的最短路径长度
//
// 2. Codeforces 613D - Kingdom and Cities
//    链接：https://codeforces.com/problemset/problem/613/D
//    题意：给一棵树和多个询问，每个询问给出一些关键点，要求切断最少的非关键点使关键点两两不连通
//
// 3. 洛谷 P2495 - [SDOI2011]消耗战
//    链接：https://www.luogu.com.cn/problem/P2495
//    题意：给一棵树和多个询问，每个询问给出一些关键点，要求切断最少代价的边使关键点都无法到达根
节点
//
// 4. LOJ #6056 - 「雅礼集训 2017 Day11」回转寿司
//    链接：https://loj.ac/p/6056
//    题意：涉及树上关键点的查询问题
//
// 5. Codeforces 1000G - Two Melborians, One Siberian
//    链接：https://codeforces.com/problemset/problem/1000/G
//    题意：在树上处理多组询问，涉及关键点的最短距离等信息
//
// 6. AtCoder ABC154F - Many Many Paths
//    链接：https://atcoder.jp/contests/abc154/tasks/abc154_f
//    题意：计算树上路径数量，可以使用虚树优化
//
// 7. 洛谷 P4103 - [HEOI2014]大工程
//    链接：https://www.luogu.com.cn/problem/P4103
//    题意：给一棵树和多个询问，每个询问给出一些关键点，要求计算所有关键点对之间距离的和、最小值
和最大值
//
// 8. 洛谷 P3233 - [HNOI2014]世界树
//    链接：https://www.luogu.com.cn/problem/P3233
```

```
//     题意：给一棵树和多个询问，每个询问给出一些关键点，要求计算每个关键点能管理多少个点
//
// 9. Codeforces 1109D - Treeland and Viruses
//     链接：https://codeforces.com/problemset/problem/1109/D
//     题意：给一棵树和多个病毒源点，每个病毒源点以不同速度扩散，求每个点被哪个病毒源点感染
//
// 10. 洛谷 P5327 - [ZJOI2019]语言
//     链接：https://www.luogu.com.cn/problem/P5327
//     题意：涉及树上路径覆盖的复杂问题
//
// 11. SPOJ QTREE5 - Query on a tree V
//     链接：https://www.spoj.com/problems/QTREE5/
//     题意：树上点颜色修改和查询距离最近的白色节点
//
// 12. 洛谷 P3232 - [HNOI2013]游走
//     链接：https://www.luogu.com.cn/problem/P3232
//     题意：给定无向连通图，通过高斯消元计算边的期望经过次数，再贪心编号使总得分期望最小
//
// 时间复杂度分析：
// 1. 预处理（DFS 序、LCA）：O(n log n)
// 2. 构造虚树：O(k log k)
// 3. 在虚树上 DP：O(k)
// 总体复杂度：O(n log n + Σk log k)
//
// 空间复杂度：O(n + k)
//
// 工程化考量：
// 1. 注意虚树边通常没有边权，需要通过原树计算
// 2. 清空关键点标记时避免使用 memset，用 for 循环逐个清除
// 3. 排序后的关键点顺序不是原节点序，如需按原序输出需额外保存
// 4. 虚树主要用于卡常题，需注意常数优化
//
// 算法设计本质与核心思想：
// 1. 设计动机：虚树算法的核心动机是优化树上多次询问问题。当需要对树上不同关键点集合进行多次查询时，
//     如果每次都遍历整棵树，时间复杂度会很高。虚树通过只保留关键点及其 LCA，将问题规模从 O(n)降低到 O(k)。
// 2. 数学原理：
//     - LCA 性质：任意两个节点的 LCA 在 DFS 序上具有特定性质，可以用于构建虚树
//     - 节点数量上界：虚树节点数不超过 2*k-1，这是通过数学归纳法可以证明的
//     - 树的结构保持：虚树保持了原树中关键点之间的祖先关系
// 3. 与其它算法的关联：
//     - 树上倍增：虚树构建需要 LCA，通常使用树上倍增算法
```

```
//     - 树形DP：虚树上的动态规划是解决问题的核心
//     - 单调栈：构建虚树时使用的单调栈技巧与其它算法中的单调栈类似
// 4. 工程化应用：
//     - 内存优化：避免使用全局数组清零，用循环逐个清除
//     - 常数优化：选择合适的虚树构建方法（单调栈法通常更快）
//     - 边界处理：正确处理根节点、叶子节点等特殊情况
//
// 语言特性差异与跨语言实现：
// 1. Java实现特点：
//     - 使用对象封装，代码结构清晰
//     - 自定义FastReader提高输入效率
//     - 递归深度可能受限，需要改用迭代实现
// 2. C++实现特点：
//     - 性能最优，适合大数据量
//     - 需要注意编译环境问题，避免使用复杂STL
//     - 指针操作灵活但需谨慎
// 3. Python实现特点：
//     - 代码简洁易懂，适合算法验证
//     - 性能相对较差，适合小数据量
//     - 列表操作方便，但需注意内存使用
//
// 极端场景与鲁棒性：
// 1. 空输入处理：关键点为空时的特殊处理
// 2. 极端数据规模：关键点数量接近节点总数、树退化为链的情况、深度很大的树结构
// 3. 边界条件：关键点包含根节点、关键点之间存在父子关系、关键点相邻的情况
//
// 性能优化策略：
// 1. 算法层面优化：选择合适的虚树构建方法、优化DP状态转移方程、预处理减少重复计算
// 2. 实现层面优化：减少函数调用开销、优化内存访问模式、使用位运算等底层优化技巧
// 3. 工程层面优化：输入输出优化、内存池技术、缓存友好设计
//
// 调试技巧与问题定位：
// 1. 中间过程打印：打印DFS序、打印LCA计算结果、打印虚树构建过程
// 2. 断言验证：验证虚树节点数量上界、验证关键点标记正确性、验证DP状态转移正确性
// 3. 性能分析：使用性能分析工具定位瓶颈、对比不同实现的性能差异、分析时间复杂度常数项影响

// 寻宝游戏，java版
// 一共有n个节点，节点有两种类型，刷宝的点 和 不刷宝的点
// 一共有n-1条无向边，每条边有边权，所有节点组成一棵树
// 开始时所有节点都是不刷宝的点，接下来有m条操作，格式如下
// 操作 x : x号点的类型翻转，刷宝的点 变成 不刷宝的点，不刷宝的点 变成 刷宝的点
// 一次操作后，每个刷宝的点都会产生宝物，你可以瞬移到任何点作为出发点，瞬移是无代价的
// 你需要走路拿到所有的宝物，最后回到出发点，打印最小的行走总路程，一共有m条打印
```

```java
// 1 <= n、m <= 10^5
// 测试链接 : https://www.luogu.com.cn/problem/P3320
// 提交以下的code，提交时请把类名改成"Main"，可以通过所有测试用例

import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStreamWriter;
import java.io.PrintWriter;
import java.util.TreeSet;

public class Code06_TreasureHunt1 {

    public static int MAXN = 100001;
    public static int MAXP = 20;
    public static int n, m;

    public static int[] head = new int[MAXN];
    public static int[] nxt = new int[MAXN << 1];
    public static int[] to = new int[MAXN << 1];
    public static int[] weight = new int[MAXN << 1];
    public static int cntg;

    public static int[] dep = new int[MAXN];
    public static long[] dist = new long[MAXN];
    public static int[] dfn = new int[MAXN];
    public static int[] seg = new int[MAXN];
    public static int[][] stjump = new int[MAXN][MAXP];
    public static int cntd;

    public static int[] arr = new int[MAXN];
    public static boolean[] treasure = new boolean[MAXN];
    // 这里为了方便，使用语言自带的有序表
    public static TreeSet<Integer> set = new TreeSet<>();

    public static long[] ans = new long[MAXN];

    public static void addEdge(int u, int v, int w) {
        nxt[++cntg] = head[u];
        to[cntg] = v;
        weight[cntg] = w;
        head[u] = cntg;
    }
```

```java
// dfs 递归版，java 会爆栈，C++可以通过
public static void dfs1(int u, int fa, int w) {
    dep[u] = dep[fa] + 1;
    dfn[u] = ++cntd;
    seg[cntd] = u;
    dist[u] = dist[fa] + w;
    stjump[u][0] = fa;
    for (int p = 1; p < MAXP; p++) {
        stjump[u][p] = stjump[stjump[u][p - 1]][p - 1];
    }
    for (int e = head[u]; e > 0; e = nxt[e]) {
        if (to[e] != fa) {
            dfs1(to[e], u, weight[e]);
        }
    }
}

// 不会改迭代版，去看讲解 118，详解了从递归版改迭代版
public static int[][] ufwe = new int[MAXN][4];

public static int stacksize, u, f, w, e;

public static void push(int u, int f, int w, int e) {
    ufwe[stacksize][0] = u;
    ufwe[stacksize][1] = f;
    ufwe[stacksize][2] = w;
    ufwe[stacksize][3] = e;
    stacksize++;
}

public static void pop() {
    --stacksize;
    u = ufwe[stacksize][0];
    f = ufwe[stacksize][1];
    w = ufwe[stacksize][2];
    e = ufwe[stacksize][3];
}

// dfs1 的迭代版
public static void dfs2() {
    stacksize = 0;
    push(1, 0, 0, -1);
    while (stacksize > 0) {
```

```
        pop();
        if (e == -1) {
            dep[u] = dep[f] + 1;
            dfn[u] = ++cntd;
            seg[cntd] = u;
            dist[u] = dist[f] + w;
            stjump[u][0] = f;
            for (int p = 1; p < MAXP; p++) {
                stjump[u][p] = stjump[stjump[u][p - 1]][p - 1];
            }
            e = head[u];
        } else {
            e = nxt[e];
        }
        if (e != 0) {
            push(u, f, w, e);
            if (to[e] != f) {
                push(to[e], u, weight[e], -1);
            }
        }
    }
}

public static int getLca(int a, int b) {
    if (dep[a] < dep[b]) {
        int tmp = a; a = b; b = tmp;
    }
    for (int p = MAXP - 1; p >= 0; p--) {
        if (dep[stjump[a][p]] >= dep[b]) {
            a = stjump[a][p];
        }
    }
    if (a == b) {
        return a;
    }
    for (int p = MAXP - 1; p >= 0; p--) {
        if (stjump[a][p] != stjump[b][p]) {
            a = stjump[a][p];
            b = stjump[b][p];
        }
    }
    return stjump[a][0];
}
```

```java
public static long getDist(int x, int y) {
    return dist[x] + dist[y] - 2 * dist[getLca(x, y)];
}

public static void compute() {
    long curAns = 0;
    for (int i = 1; i <= m; i++) {
        int nodeId = arr[i];
        int dfnId = dfn[nodeId];
        if (!treasure[nodeId]) {
            treasure[nodeId] = true;
            set.add(dfnId);
        } else {
            treasure[nodeId] = false;
            set.remove(dfnId);
        }
        if (set.size() <= 1) {
            curAns = 0;
        } else {
            int low = seg[set.lower(dfnId) != null ? set.lower(dfnId) : set.last()];
            int high = seg[set.higher(dfnId) != null ? set.higher(dfnId) : set.first()];
            long delta = getDist(nodeId, low) + getDist(nodeId, high) - getDist(low, high);
            if (treasure[nodeId]) {
                curAns += delta;
            } else {
                curAns -= delta;
            }
        }
        ans[i] = curAns;
    }
}

public static void main(String[] args) throws Exception {
    FastReader in = new FastReader(System.in);
    PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));
    n = in.nextInt();
    m = in.nextInt();
    for (int i = 1, u, v, w; i < n; i++) {
        u = in.nextInt();
        v = in.nextInt();
        w = in.nextInt();
        addEdge(u, v, w);
```

```java
        addEdge(v, u, w);
    }
    // dfs1(1, 0, 0);
    dfs2();
    for (int i = 1; i <= m; i++) {
        arr[i] = in.nextInt();
    }
    compute();
    for (int i = 1; i <= m; i++) {
        out.println(ans[i]);
    }
    out.flush();
    out.close();
}

// 读写工具类
static class FastReader {
    private final byte[] buffer = new byte[1 << 16];
    private int ptr = 0, len = 0;
    private final InputStream in;

    FastReader(InputStream in) {
        this.in = in;
    }

    private int readByte() throws IOException {
        if (ptr >= len) {
            len = in.read(buffer);
            ptr = 0;
            if (len <= 0)
                return -1;
        }
        return buffer[ptr++];
    }

    int nextInt() throws IOException {
        int c;
        do {
            c = readByte();
        } while (c <= ' ' && c != -1);
        boolean neg = false;
        if (c == '-') {
            neg = true;
```

```
            c = readByte();
        }
        int val = 0;
        while (c > ' ' && c != -1) {
            val = val * 10 + (c - '0');
            c = readByte();
        }
        return neg ? -val : val;
    }
}

}
```

==

文件: Code06_TreasureHunt1.py

==

```
# 虚树(Virtual Tree)算法详解与应用
#
# 虚树是一种优化技术，用于解决树上多次询问的问题，每次询问涉及部分关键点
# 虚树只保留关键点及其两两之间的LCA，节点数控制在O(k)级别，从而提高效率
#
# 算法核心思想：
# 1. 虚树包含所有关键点和它们两两之间的LCA
# 2. 虚树的节点数不超过2*k-1（k为关键点数）
# 3. 在虚树上进行DP等操作，避免遍历整棵树
#
# 构造方法：
# 方法一：二次排序法
# 1. 将关键点按DFS序排序
# 2. 相邻点求LCA并加入序列
# 3. 再次排序并去重得到虚树所有节点
# 4. 按照父子关系连接节点
#
# 方法二：单调栈法
# 1. 将关键点按DFS序排序
# 2. 用栈维护虚树的一条链
# 3. 逐个插入关键点并维护栈结构
#
# 应用场景：
# 1. 树上多次询问，每次询问涉及部分关键点
# 2. 需要在关键点及其LCA上进行DP等操作
# 3. 数据范围要求∑k较小（通常≤10^5）
```

```
#
# 相关题目：
# 1. 宝藏猎人问题
#    题意：给一棵树和多个宝藏点，求收集所有宝藏的最短路径长度
#
# 2. Codeforces 613D - Kingdom and Cities
#    链接：https://codeforces.com/problemset/problem/613/D
#    题意：给一棵树和多个询问，每个询问给出一些关键点，要求切断最少的非关键点使关键点两两不连通
#
# 3. 洛谷 P2495 - [SDOI2011]消耗战
#    链接：https://www.luogu.com.cn/problem/P2495
#    题意：给一棵树和多个询问，每个询问给出一些关键点，要求切断最少代价的边使关键点都无法到达根节点
#
# 时间复杂度分析：
# 1. 预处理（DFS 序、LCA）：O(n log n)
# 2. 构造虚树：O(k log k)
# 3. 在虚树上 DP：O(k)
# 总体复杂度：O(n log n + Σk log k)
#
# 空间复杂度：O(n + k)
#
# 工程化考量：
# 1. 注意虚树边通常没有边权，需要通过原树计算
# 2. 清空关键点标记时避免使用 memset，用 for 循环逐个清除
# 3. 排序后的关键点顺序不是原节点序，如需按原序输出需额外保存
# 4. 虚树主要用于卡常题，需注意常数优化

# 宝藏猎人，Python 版
# 有一个 n 个节点的树，边权为 1
# 有 k 个宝藏点，求从节点 1 出发收集所有宝藏并回到节点 1 的最短路径长度
# 1 <= n <= 2*10^5
# 1 <= k <= n

import sys
from collections import deque

MAXN = 200001
MAXP = 20

# 全局变量
n, k = 0, 0
```

```python
# 原始树
headg = [0] * MAXN
nextg = [0] * (MAXN << 1)
tog = [0] * (MAXN << 1)
cntg = 0

# 虚树
headv = [0] * MAXN
nextv = [0] * MAXN
tov = [0] * MAXN
cntv = 0

# 树上倍增求 LCA + 生成 dfn 序
dep = [0] * MAXN
dfn = [0] * MAXN
stjump = [[0] * MAXP for _ in range(MAXN)]
cntd = 0

# 关键点数组
arr = [0] * MAXN
# 标记节点是否是关键点
isKey = [False] * MAXN

# 第一种建树方式
tmp = [0] * (MAXN << 1)
# 第二种建树方式
stk = [0] * MAXN

# 动态规划相关
# dp[u]表示在虚树中以 u 为根的子树中，收集所有宝藏并回到 u 的最短路径长度
dp = [0] * MAXN

# 原始树连边
def addEdgeG(u, v):
    global cntg
    cntg += 1
    nextg[cntg] = headg[u]
    tog[cntg] = v
    headg[u] = cntg

# 虚树连边
def addEdgeV(u, v):
    global cntv
```

```python
    cntv += 1
    nextv[cntv] = headv[u]
    tov[cntv] = v
    headv[u] = cntv

# nums 中的数，根据 dfn 的大小排序，手撸双指针快排
def sortByDfn(nums, l, r):
    if l >= r:
        return
    i, j = l, r
    pivot = nums[(l + r) >> 1]
    while i <= j:
        while dfn[nums[i]] < dfn[pivot]:
            i += 1
        while dfn[nums[j]] > dfn[pivot]:
            j -= 1
        if i <= j:
            nums[i], nums[j] = nums[j], nums[i]
            i += 1
            j -= 1
    sortByDfn(nums, l, j)
    sortByDfn(nums, i, r)

# 树上倍增的 dfs 过程
def dfs(u, fa):
    global cntd
    dep[u] = dep[fa] + 1
    cntd += 1
    dfn[u] = cntd
    stjump[u][0] = fa
    for p in range(1, MAXP):
        stjump[u][p] = stjump[stjump[u][p - 1]][p - 1]
    e = headg[u]
    while e > 0:
        if tog[e] != fa:
            dfs(tog[e], u)
        e = nextg[e]

# 返回 a 和 b 的最低公共祖先
def getLca(a, b):
    if dep[a] < dep[b]:
        a, b = b, a
    for p in range(MAXP - 1, -1, -1):
```

```
        if dep[stjump[a][p]] >= dep[b]:
            a = stjump[a][p]
    if a == b:
        return a
    for p in range(MAXP - 1, -1, -1):
        if stjump[a][p] != stjump[b][p]:
            a = stjump[a][p]
            b = stjump[b][p]
    return stjump[a][0]

# 二次排序 + LCA 连边的方式建立虚树
def buildVirtualTree1():
    sortByDfn(arr, 1, k)
    len_idx = 0
    for i in range(1, k):
        len_idx += 1
        tmp[len_idx] = arr[i]
        len_idx += 1
        tmp[len_idx] = getLca(arr[i], arr[i + 1])
    len_idx += 1
    tmp[len_idx] = arr[k]
    sortByDfn(tmp, 1, len_idx)
    unique = 1
    for i in range(2, len_idx + 1):
        if tmp[unique] != tmp[i]:
            unique += 1
            tmp[unique] = tmp[i]
    global cntv
    cntv = 0
    for i in range(1, unique + 1):
        headv[tmp[i]] = 0
    for i in range(1, unique):
        addEdgeV(getLca(tmp[i], tmp[i + 1]), tmp[i + 1])
    return tmp[1]

# 单调栈的方式建立虚树
def buildVirtualTree2():
    sortByDfn(arr, 1, k)
    global cntv
    cntv = 0
    headv[arr[1]] = 0
    top = 0
    top += 1
```

```python
    stk[top] = arr[1]
    for i in range(2, k + 1):
        x = arr[i]
        y = stk[top]
        lca = getLca(x, y)
        while top > 1 and dfn[stk[top - 1]] >= dfn[lca]:
            addEdgeV(stk[top - 1], stk[top])
            top -= 1
        if lca != stk[top]:
            headv[lca] = 0
            addEdgeV(lca, stk[top])
            stk[top] = lca
        headv[x] = 0
        top += 1
        stk[top] = x
    while top > 1:
        addEdgeV(stk[top - 1], stk[top])
        top -= 1
    return stk[1]

# 树型 dp 的过程
def treedp(u):
    dp[u] = 0
    e = headv[u]
    while e > 0:
        v = tov[e]
        treedp(v)
        # 从 u 到 v 再回到 u 的距离是 2 * distance(u, v)
        dp[u] += dp[v] + 2 * (dep[v] - dep[u])
        e = nextv[e]

def compute():
    global k
    # 节点标记关键点信息
    for i in range(1, k + 1):
        isKey[arr[i]] = True
    # 如果节点 1 不是关键点，也要加入
    need_add = not isKey[1]
    if need_add:
        k += 1
        arr[k] = 1
        isKey[1] = True
    tree = buildVirtualTree1()
```

```python
    # tree = buildVirtualTree2()
    treedp(tree)
    result = dp[tree]
    # 节点撤销关键点信息
    for i in range(1, k + 1):
        isKey[arr[i]] = False
    if need_add:
        k -= 1
    return result

# 主函数
if __name__ == "__main__":
    # 读取输入
    n = int(input())
    for i in range(1, n):
        u, v = map(int, input().split())
        addEdgeG(u, v)
        addEdgeG(v, u)
    dfs(1, 0)
    k = int(input())
    arr_values = list(map(int, input().split()))
    for i in range(1, k + 1):
        arr[i] = arr_values[i - 1]
    print(compute())
```

===

文件: Code06_TreasureHunt2.java

===

```java
package class180;

// 寻宝游戏，C++版
// 一共有 n 个节点，节点有两种类型，刷宝的点 和 不刷宝的点
// 一共有 n-1 条无向边，每条边有边权，所有节点组成一棵树
// 开始时所有节点都是不刷宝的点，接下来有 m 条操作，格式如下
// 操作 x : x 号点的类型翻转，刷宝的点 变成 不刷宝的点，不刷宝的点 变成 刷宝的点
// 一次操作后，每个刷宝的点都会产生宝物，你可以瞬移到任何点作为出发点，瞬移是无代价的
// 你需要走路拿到所有的宝物，最后回到出发点，打印最小的行走总路程，一共有 m 条打印
// 1 <= n、m <= 10^5
// 测试链接 : https://www.luogu.com.cn/problem/P3320
// 如下实现是 C++的版本，C++版本和 java 版本逻辑完全一样
// 提交如下代码，可以通过所有测试用例
```

```
//#include <bits/stdc++.h>
//
//using namespace std;
//
//const int MAXN = 100001;
//const int MAXP = 20;
//int n, m;
//
//int head[MAXN];
//int nxt[MAXN << 1];
//int to[MAXN << 1];
//int weight[MAXN << 1];
//int cntg;
//
//int dep[MAXN];
//long long dist[MAXN];
//int dfn[MAXN];
//int seg[MAXN];
//int stjump[MAXN][MAXP];
//int cntd;
//
//int arr[MAXN];
//bool treasure[MAXN];
//std::set<int> st;
//std::set<int>::iterator it;
//
//long long ans[MAXN];
//
//void addEdge(int u, int v, int w) {
//    nxt[++cntg] = head[u];
//    to[cntg] = v;
//    weight[cntg] = w;
//    head[u] = cntg;
//}
//
//void dfs(int u, int fa, int w) {
//    dep[u] = dep[fa] + 1;
//    dfn[u] = ++cntd;
//    seg[cntd] = u;
//    dist[u] = dist[fa] + w;
//    stjump[u][0] = fa;
//    for (int p = 1; p < MAXP; p++) {
//        stjump[u][p] = stjump[stjump[u][p - 1]][p - 1];
```

```
//    }
//    for (int e = head[u]; e; e = nxt[e]) {
//        int v = to[e];
//        if (v != fa) {
//            dfs(v, u, weight[e]);
//        }
//    }
//}
//
//int getLca(int a, int b) {
//    if (dep[a] < dep[b]) {
//        swap(a, b);
//    }
//    for (int p = MAXP - 1; p >= 0; p--) {
//        if (dep[stjump[a][p]] >= dep[b]) {
//            a = stjump[a][p];
//        }
//    }
//    if (a == b) {
//        return a;
//    }
//    for (int p = MAXP - 1; p >= 0; p--) {
//        if (stjump[a][p] != stjump[b][p]) {
//            a = stjump[a][p];
//            b = stjump[b][p];
//        }
//    }
//    return stjump[a][0];
//}
//
//long long getDist(int x, int y) {
//    return dist[x] + dist[y] - 2LL * dist[getLca(x, y)];
//}
//
//void compute() {
//    long long curAns = 0;
//    for (int i = 1; i <= m; i++) {
//        int nodeId = arr[i];
//        int dfnId = dfn[nodeId];
//        if (!treasure[nodeId]) {
//            treasure[nodeId] = true;
//            st.insert(dfnId);
//        } else {
```

```
//            treasure[nodeId] = false;
//            st.erase(dfnId);
//        }
//        if (st.size() <= 1) {
//            curAns = 0;
//        } else {
//            int low = seg[(it = st.lower_bound(dfnId)) == st.begin() ? *--st.end() : *--it];
//            int high = seg[(it = st.upper_bound(dfnId)) == st.end() ? *st.begin() : *it];
//            long long delta = getDist(nodeId, low) + getDist(nodeId, high) - getDist(low,
high);
//            if (treasure[nodeId]) {
//                curAns += delta;
//            } else {
//                curAns -= delta;
//            }
//        }
//        ans[i] = curAns;
//    }
//}
//
//int main() {
//    ios::sync_with_stdio(false);
//    cin.tie(nullptr);
//    cin >> n >> m;
//    for (int i = 1, u, v, w; i < n; i++) {
//        cin >> u >> v >> w;
//        addEdge(u, v, w);
//        addEdge(v, u, w);
//    }
//    dfs(1, 0, 0);
//    for (int i = 1; i <= m; i++) {
//        cin >> arr[i];
//    }
//    compute();
//    for (int i = 1; i <= m; i++) {
//        cout << ans[i] << '\n';
//    }
//    return 0;
//}
```

===