==========
文件夹: class153_TreeDifference
==========

[Markdown 文件]
==========
文件: README_CF519E.md
==========
Codeforces 519E. A and B and Lecture Rooms

题目描述

给定一棵有 `n` 个节点的树（节点编号从 1 到 n），树由 `edges` 数组表示。有多个查询，每个查询给出两个节点 `u` 和 `v`，要求找到树上到 `u` 和 `v` 距离相等的节点数量。

算法思路

核心思想
1. **LCA 计算**：使用倍增法快速计算两个节点的最近公共祖先
2. **距离计算**：通过深度计算节点间距离
3. **中点分析**：根据距离的奇偶性判断是否存在满足条件的节点
4. **子树大小统计**：利用子树大小计算满足条件的节点数量

具体步骤

1. **预处理阶段**：
   - 构建树的邻接表表示
   - 使用 DFS 预处理每个节点的深度、父节点和子树大小
   - 使用倍增法预处理父节点数组，用于快速计算 LCA

2. **查询处理**：
   - 如果 `u == v`，所有节点都满足条件，返回 `n`
   - 计算 `u` 和 `v` 之间的距离 `dist = depth[u] + depth[v] - 2 * depth[lca]`
   - 如果距离为奇数，没有满足条件的节点，返回 `0`
   - 如果距离为偶数：
     - 当 `u` 和 `v` 在同一深度时，中点在 LCA 处
     - 当 `u` 和 `v` 在不同深度时，找到路径的中点

时间复杂度
- 预处理：O(n log n)
- 每个查询：O(log n)
- 总复杂度：O(n log n + q log n)，其中 q 是查询数量

空间复杂度
- O(n log n) 用于存储倍增数组
- O(n) 用于存储深度和子树大小信息

关键点分析

1. **距离计算**：树上两点距离 = depth[u] + depth[v] - 2 * depth[lca]
2. **中点位置**：
   - 当深度相同时，中点在 LCA 处
   - 当深度不同时，中点位于较深节点向上移动 dist/2 步的位置
3. **节点计数**：
   - 深度相同时：n - size[uMid] - size[vMid]
   - 深度不同时：size[mid] - size[prev]

代码实现

提供了三种语言的实现：
- **Java**：使用邻接表和倍增法
- **C++**：使用 vector 和数学函数
- **Python**：使用列表和数学模块

测试用例

```
输入:
n = 4
edges = [[1,2], [1,3], [2,4]]
queries = [[1,2], [2,3], [3,4], [2,4]]

输出: [2, 1, 1, 1]
```

解释：
- 查询(1,2)：节点 3 和 4 到 1 和 2 的距离相等
- 查询(2,3)：只有节点 1 满足条件
- 查询(3,4)：只有节点 1 满足条件
- 查询(2,4)：只有节点 1 满足条件

应用场景
- 网络拓扑分析
- 社交网络距离计算
- 路径规划中的等距点查找

==========

文件: README_CF739B.md
==========
Codeforces 739B Alyona and a tree

题目描述

给定一棵有根树，根节点为 1，每个节点都有一个权值 a[i]，每条边也有权值。对于每个节点 v，找到有多少个祖先 u，满足 dist(u,v) <= a[v]。其中 dist(u,v)表示从 u 到 v 路径上所有边权值的和。

题目链接

[Codeforces 739B Alyona and a tree](https://codeforces.com/contest/739/problem/B)

输入格式

- 第一行包含一个整数 n，表示节点的数量。
- 第二行包含 n 个整数 a[1], a[2], ..., a[n]，表示每个节点的权值。
- 接下来 n-1 行，每行包含两个整数 p[i]和 w[i]，表示节点 i 的父节点是 p[i]，连接它们的边权值为 w[i]。

输出格式

输出 n 个整数，第 i 个整数表示节点 i 有多少个满足条件的祖先。

样例输入

```
5
1 2 3 4 5
1 1
2 1
3 1
4 1
```

样例输出

```
0 1 2 3 4
```

数据范围

- 1 <= n <= 200000
- 1 <= a[i] <= 10^9
- 1 <= w[i] <= 10^9

解题思路

这是一道结合了二分查找和树上差分思想的题目。我们需要对每个节点找到满足条件的祖先数量。

算法分析

1. **问题转化**：
   - 对于每个节点 v，我们需要找到所有祖先 u，满足 dist(u,v) <= a[v]。
   - 由于树的性质，祖先节点到当前节点的距离是单调的，可以使用二分查找。

2. **倍增法优化**：
   - 使用倍增法预处理祖先信息，可以在 O(log n)时间内找到满足条件的最远祖先。
   - 预处理时间复杂度 O(n log n)。

3. **树上差分**：
   - 找到每个节点的最远满足条件的祖先后，对路径进行差分标记。
   - 最后通过一次 DFS 计算差分数组的前缀和，得到最终结果。

4. **实现要点**：
   - 使用 DFS 序对节点进行处理
   - 使用倍增数组预处理祖先信息
   - 使用差分数组统计答案

时间复杂度

- 预处理阶段：O(N log N)（倍增法预处理）
- 查询阶段：O(N log N)（每个节点二分查找）
- 计算结果：O(N)（DFS 遍历）
- 总体复杂度：O(N log N)

空间复杂度

- 树存储：O(N)
- 倍增数组：O(N log N)
- 差分数组：O(N)
- 总体复杂度：O(N log N)

代码实现

Java 版本

```java
// 详细实现见 Code10_AlyonaAndTree.java
```

C++版本

```cpp
// 详细实现见 Code10_AlyonaAndTree.cpp
```

Python 版本

```python
# 详细实现见 Code10_AlyonaAndTree.py
```

算法要点

1. **倍增法**：
   - 预处理祖先信息，stjump[u][p]表示节点 u 的第 2^p 个祖先
   - stsum[u][p]表示节点 u 到其第 2^p 个祖先路径上的边权和

2. **二分查找**：
   - 利用距离的单调性，使用倍增法进行二分查找
   - 找到满足条件的最远祖先节点

3. **树上差分**：
   - 对于每个节点 v，找到最远满足条件的祖先 u
   - 在差分数组上标记：diff[u]--, diff[v]++
   - 最后通过 DFS 计算前缀和得到答案

常见错误与注意事项

1. **倍增数组初始化错误**：
   - stjump[u][0]应为节点 u 的父节点
   - stsum[u][0]应为节点 u 到其父节点的边权

2. **二分查找错误**：
   - 边界条件处理错误
   - 距离计算错误

3. **差分标记错误**：
   - 标记位置错误
   - 忘记处理根节点的特殊情况

4. **数据类型溢出**：
   - 距离可能超过 int 范围，应使用 long long

扩展应用

1. **树上路径查询**：
   - 处理树上路径权值和查询问题
   - 结合树链剖分或点分治

2. **动态树问题**：
   - 处理动态加点/删点的树上问题
   - 结合 LCT 或 ETT 等数据结构

3. **树上统计问题**：
   - 统计满足特定条件的节点对数量
   - 结合莫队算法或虚树技术

相关题目

1. **Codeforces 191C Fools and Roads** - 树上差分
2. **洛谷 P3128 [USACO15DEC] Max Flow P** - 树上点差分
3. **LOJ 10131 暗的连锁** - 树上边差分
4. **AtCoder ABC187 E - Through Path** - 树上差分

总结

Alyona and a tree 是一道经典的树上算法题，结合了倍增法、二分查找和树上差分等多个知识点。通过这道题目，可以深入理解：

1. 倍增法在树上问题中的应用
2. 如何利用单调性进行二分查找优化
3. 树上差分思想在统计问题中的应用
4. 如何处理树上路径权值和查询问题

掌握这些知识点对于解决更复杂的树上问题具有重要意义。

==========

文件: README_LeetCode2096.md
==========
LeetCode 2096. 从二叉树一个节点到另一个节点的方向

题目描述

给定一棵二叉树的根节点 `root`，以及两个整数 `startValue` 和 `destValue`。找到从值为 `startValue` 的节点到值为 `destValue` 的节点的最短路径方向。

路径方向由字符组成：
- ``U``：向上移动到父节点
- ``L``：向左移动到左子节点
- ``R``：向右移动到右子节点

算法思路

核心思想
1. **LCA 计算**：找到起点和目标节点的最近公共祖先
2. **路径重建**：分别构建从起点到 LCA 和从 LCA 到目标的路径
3. **方向转换**：将起点到 LCA 的路径转换为'U'方向

具体步骤

1. **寻找 LCA**：
   - 使用递归方法找到起点和目标节点的最近公共祖先
   - 如果当前节点是起点或目标节点，或者左右子树分别包含起点和目标节点，则当前节点是 LCA

2. **构建路径**：
   - 从 LCA 到起点：记录路径方向（L 或 R）
   - 从 LCA 到目标：记录路径方向（L 或 R）

3. **方向转换**：
   - 起点到 LCA 的路径全部转换为'U'方向
   - LCA 到目标的路径保持原方向
   - 拼接两部分路径得到最终结果

时间复杂度
- 寻找 LCA：O(n)
- 构建路径：O(n)
- 总复杂度：O(n)

空间复杂度
- O(h) 用于递归栈，其中 h 是树的高度

- O(n) 在最坏情况下（树退化为链表）

关键点分析

1. **LCA 的重要性**：通过 LCA 可以避免重复遍历路径
2. **路径方向**：从子节点到父节点的方向都是'U'
3. **路径拼接**：起点→LCA→目标的路径是最短路径

代码实现

提供了三种语言的实现：
- **Java**：使用递归和 StringBuilder
- **C++**：使用递归和字符串操作
- **Python**：使用递归和列表操作

测试用例

测试用例 1
```
树结构：
      5
     / \\
    1   2
   /   / \\
  3   6   4

startValue = 3, destValue = 6
输出："UURL"
```

路径解释：
- 从 3 向上到 1：U
- 从 1 向上到 5：U
- 从 5 向右到 2：R
- 从 2 向左到 6：L

测试用例 2
```
树结构：
      2
     /
    1
```

startValue = 2, destValue = 1
输出: "L"
```

## 应用场景
- 二叉树导航系统
- 文件系统路径查找
- 游戏中的角色移动路径规划

==========

文件: README_LeetCode2646.md
==========
# LeetCode 2646. 最小化旅行的价格

## 题目描述

给定一棵有 `n` 个节点的树（节点编号从 0 到 n-1），每个节点有一个价格 `price[i]`。树由 `edges` 数组表示，其中 `edges[i] = [u, v]` 表示节点 u 和 v 之间有一条边。

有 `m` 个旅行路径，每个路径由 `trips[i] = [u, v]` 表示，表示从节点 u 旅行到节点 v。

你可以选择将某些节点的价格减半（即价格变为原来的一半，向下取整）。要求最小化所有旅行路径的总价格。

## 算法思路

### 核心思想
1. **树上差分统计**：使用树上差分算法统计每条边被经过的次数
2. **树形 DP 决策**：使用树形动态规划决定哪些节点应该减半

### 具体步骤

1. **构建图结构**：根据 edges 构建邻接表表示的树
2. **预处理 LCA**：使用倍增法预处理每个节点的父节点信息，用于快速计算 LCA
3. **树上差分统计**：
   - 对于每个旅行路径 (u, v)，找到它们的最近公共祖先 (LCA)
   - 在 u 和 v 处加 1，在 LCA 处减 2
   - 通过 DFS 统计每个节点被经过的总次数
4. **树形 DP 决策**：
   - `dp[u][0]`：节点 u 不减半的最小总价格
   - `dp[u][1]`：节点 u 减半的最小总价格
   - 状态转移：
     - 如果当前节点不减半，子节点可以减半或不减半
```

- 如果当前节点减半，子节点不能减半

时间复杂度
- 预处理 LCA：O(n log n)
- 树上差分：O(m log n)
- 树形 DP：O(n)
- 总复杂度：O((n + m) log n)

空间复杂度
- O(n log n) 用于存储倍增数组
- O(n) 用于存储差分数组和 DP 数组

代码实现

提供了三种语言的实现：
- **Java**：使用邻接表和倍增法
- **C++**：使用 vector 和队列
- **Python**：使用列表和双端队列

测试用例

测试用例 1
```
n = 4
edges = [[0,1],[1,2],[1,3]]
price = [2,2,10,6]
trips = [[0,3],[2,1],[2,3]]
输出: 23
```

测试用例 2
```
n = 2
edges = [[0,1]]
price = [2,2]
trips = [[0,0]]
输出: 1
```

关键点
1. 树上差分是统计路径覆盖次数的有效方法
2. 树形 DP 需要考虑相邻节点的约束关系
3. 减半操作只能进行一次，且相邻节点不能同时减半

4. 使用倍增法可以高效计算 LCA

应用场景
- 网络流量优化
- 资源分配问题
- 路径规划优化

==========

文件: README_LOJ10131.md
==========
LOJ 10131 暗的连锁 题目解析

题目描述

给定一棵包含 N 个节点的树（N-1 条边），以及 M 条额外的边（非树边）。每条非树边连接两个节点，与树边一起构成一个连通图。求有多少种方案，通过切断一条树边和一条非树边，使得图变得不连通。

算法思路

这是一道典型的树上边差分问题。解题的关键在于理解什么样的边组合被切断后会使图不连通：

1. **问题分析**：
   - 初始状态是一棵树，加上 M 条非树边后形成一个连通图
   - 要使图不连通，需要切断一条树边和一条非树边
   - 切断树边后，图分为两个连通分量
   - 如果切断的非树边连接这两个连通分量，则图重新连通
   - 因此，只有当非树边不连接这两个连通分量时，图才真正不连通

2. **核心思想**：
   - 对于每条树边，统计有多少条非树边跨越了这条树边连接的两个连通分量
   - 如果有 0 条非树边跨越，则切断该树边后，任意一条非树边都可以与其配对使图不连通（m 种方案）
   - 如果有 1 条非树边跨越，则只有切断该非树边才能使图不连通（1 种方案）
   - 如果有 2 条或以上非树边跨越，则无论切断哪条非树边，图都仍然连通（0 种方案）

3. **技术实现**：
   - 使用树上边差分统计每条树边被非树边跨越的次数
   - 对于每条连接 u 和 v 的非树边，会在树上形成一条 u 到 v 的路径
   - 这条路径上的所有树边都被该非树边跨越
   - 使用边差分技术：对 u 和 v 节点分别+1，对它们的 LCA 节点-2
   - 最后通过 DFS 计算每条边的覆盖次数

复杂度分析

时间复杂度
1. **预处理阶段**：
   - 构建树结构：O(N)
   - DFS 预处理深度和跳跃数组：O(N * log N)

2. **处理非树边**：
   - 对每条非树边求 LCA：O(M * log N)

3. **统计答案**：
   - DFS 计算边覆盖次数并统计答案：O(N)

总时间复杂度：O(N * log N + M * log N)

空间复杂度
- 存储树结构：O(N)
- 存储深度和跳跃数组：O(N * log N)
- 其他辅助数组：O(N)

总空间复杂度：O(N * log N)

解题技巧

1. 树上边差分
树上边差分是解决此类问题的关键技术：
```
对于连接 u 和 v 的路径，在 u 和 v 处+1，在 LCA(u,v)处-2
通过 DFS 累加子树和，可得到每条边的覆盖次数
```

2. 倍增法求 LCA
使用倍增法预处理节点的跳跃数组，可以在 O(log N)时间内求出任意两点的 LCA：
```java
// 预处理
stjump[u][0] = father[u]  // 直接父节点
stjump[u][i] = stjump[stjump[u][i-1]][i-1]  // 第 2^i 个祖先

// 查询 LCA
int lca(int a, int b) {
    // 调整深度
    if (deep[a] < deep[b]) swap(a, b);
    // 将 a 调整到与 b 同一深度
    for (int i = power; i >= 0; i--) {
```

```
        if (deep[stjump[a][i]] >= deep[b]) {
            a = stjump[a][i];
        }
    }
    // 特殊情况：一个节点是另一个的祖先
    if (a == b) return a;
    // 同时向上跳
    for (int i = power; i >= 0; i--) {
        if (stjump[a][i] != stjump[b][i]) {
            a = stjump[a][i];
            b = stjump[b][i];
        }
    }
    return stjump[a][0];  // 返回父节点
}
```

3. 链式前向星建图
使用链式前向星存储树结构，节省空间且效率高：
```java
int head[MAXN], next[MAXN<<1], to[MAXN<<1], cnt = 1;
void addEdge(int u, int v) {
    next[cnt] = head[u];
    to[cnt] = v;
    head[u] = cnt++;
}
```

代码实现要点

Java 实现要点
1. 使用 BufferedReader 和 StreamTokenizer 提高输入效率
2. 使用 Arrays.fill 初始化数组
3. 使用链式前向星存储图结构

C++实现要点
1. 使用 scanf/printf 提高输入输出效率
2. 使用 memset 初始化数组
3. 使用链式前向星存储图结构

Python 实现要点
1. 使用 sys.stdin.readline()提高输入效率
2. 使用列表推导式初始化二维数组

3. 使用非局部变量(nonlocal)在嵌套函数中修改外层变量

边界条件和异常处理

1. **空树情况**：题目保证 N>=1，无需特殊处理
2. **无非树边**：M=0 时，所有方案数为 0
3. **极端数据规模**：N 和 M 可达 10^5，需要考虑时间和空间复杂度
4. **重边和自环**：题目未明确说明，但通常不考虑

工程化考量

1. **输入输出优化**：
   - Java 使用 BufferedReader 和 StreamTokenizer
   - C++使用 scanf/printf 而非 cin/cout
   - Python 使用 sys.stdin.readline()

2. **内存管理**：
   - 合理设置数组大小，避免浪费空间
   - 及时释放不需要的资源

3. **可读性和维护性**：
   - 添加详细注释说明算法思路
   - 使用有意义的变量名
   - 模块化设计，分离不同功能

相关题目和扩展

1. **类似题目**：
   - POJ 3417 Network（几乎相同的题目）
   - 洛谷 P3128 [USACO15DEC]Max Flow P（树上点差分）
   - 洛谷 P3258 [JLOI2014]松鼠的新家（树上点差分）

2. **扩展应用**：
   - 树上点差分：对节点进行差分操作
   - 树链剖分：处理更复杂的树上路径问题
   - 树上启发式合并：处理子树信息合并问题

总结

LOJ 10131 暗的连锁是一道经典的树上边差分题目，考察了以下知识点：
1. 树的基本遍历和存储
2. 最近公共祖先(LCA)的倍增法实现
3. 树上差分（边差分）的原理和应用

4. 链式前向星建图技巧
5. 算法复杂度分析

通过这道题目，可以深入理解树上差分的思想，掌握处理树上路径问题的一般方法，为解决更复杂的树上问题打下基础。

==========

文件: README_LOJ10132.md
==========
LOJ 10132 异象石

题目描述

在一个圆上有 n 个点，按顺时针编号为 0 到 n-1。有 m 次操作，每次操作会在两个点之间连一条弦。每次操作后，求所有弦将圆分割成多少个区域。

题目链接

[LOJ 10132 异象石](https://loj.ac/problem/10132)

输入格式

- 第一行包含一个整数 N，表示节点的数量。
- 接下来 N-1 行，每行包含两个整数 u 和 v，表示节点 u 和节点 v 之间有一条边。
- 接下来一行包含一个整数 M，表示操作的数量。
- 接下来 M 行，每行包含一个操作：
  - "+ x" 表示选中节点 x
  - "- x" 表示取消选中节点 x
  - "?" 表示查询当前选中节点在圆上的最小生成树的边权和

输出格式

对于每个"?"操作，输出一行一个整数，表示当前选中节点在圆上的最小生成树的边权和。

样例输入

```
6
1 2
1 3
1 4
2 5
```

```
2 6
7
+ 1
+ 2
+ 3
+ 5
?
- 2
?
```

样例输出

```
4
3
```

数据范围

- 1 <= N <= 100000
- 1 <= M <= 100000

解题思路

这是一道结合了虚树和树上差分思想的题目。我们需要维护圆上选中点的最小生成树边权和。

算法分析

1. **圆上的距离**：
   - 圆上两点间的距离是两点间顺时针和逆时针距离的较小值。
   - 可以通过将圆上的点按照顺时针顺序排列，构建一棵树来处理。

2. **虚树**：
   - 当选中的点数较多时，直接计算最小生成树会超时。
   - 使用虚树技术，只保留关键点构建一棵树，可以大大减少计算量。

3. **最小生成树**：
   - 在圆上，选中点的最小生成树可以通过按照顺时针顺序排序后，连接相邻点来构造。
   - 边权和等于相邻点间距离之和。

4. **实现要点**：
   - 使用 DFS 序对节点排序

   - 使用单调栈构建虚树
   - 计算虚树上边的权值和

时间复杂度

- 预处理阶段：O(N log N)（倍增法求 LCA）
- 每次操作：O(k log k)（k 为选中点数）
- 总体复杂度：O(N log N + M k log k)

空间复杂度

- 树存储：O(N)
- LCA 相关数组：O(N log N)
- 虚树相关：O(k)
- 总体复杂度：O(N log N)

代码实现

Java 版本

```java
// 详细实现见 Code09_AlienStone.java
```

C++版本

```cpp
// 详细实现见 Code09_AlienStone.cpp
```

Python 版本

```python
# 详细实现见 Code09_AlienStone.py
```

算法要点

1. **虚树构建**：
   - 按 DFS 序对选中点排序
   - 使用单调栈构建虚树
   - 虚树节点数不超过 2k（k 为选中点数）

2. **LCA 算法**：
   - 使用倍增法预处理 LCA，预处理时间复杂度 O(N log N)，查询时间复杂度 O(log N)。

3. **圆上距离计算**：
   - 圆上两点间距离 = min(|a-b|, n-|a-b|)

常见错误与注意事项

1. **虚树构建错误**：
   - 节点排序错误
   - 单调栈维护错误
   - LCA 计算错误

2. **边界处理错误**：
   - 选中点数为 0 或 1 的特殊情况
   - 根节点的特殊处理
   - 数组越界

3. **精度问题**：
   - 注意整数溢出
   - 使用 long long 类型存储结果

扩展应用

1. **树上差分**：
   - 处理树上路径修改问题
   - 点差分和边差分的应用

2. **虚树应用**：
   - 处理树上多点查询问题
   - 减少计算复杂度

3. **动态最小生成树**：
   - 处理动态加点/删点的最小生成树问题
   - 结合数据结构优化

相关题目

1. **LOJ 10131 暗的连锁** - 树上边差分
2. **洛谷 P3258 [JLOI2014] 松鼠的新家** - 树上点差分
3. **Codeforces 1017G The Tree** - 树链剖分线段树
4. **LibreOJ 10134 Dis** - 差分

总结

异象石是一道综合性的题目，结合了虚树、LCA、最小生成树等多个算法知识点。通过这道题目，可以深入理解：

1. 虚树的构建方法和应用场景
2. LCA 算法在树上问题中的应用
3. 如何处理动态点集的最小生成树问题
4. 树上差分思想的扩展应用

掌握这些知识点对于解决更复杂的树上问题具有重要意义。

==========

文件: README_P3258.md
==========
洛谷 P3258 [JLOI2014] 松鼠的新家

题目描述

松鼠家族的成员需要在树上移动，从一个节点到另一个节点。给定一棵包含 N 个节点的树，以及 N-1 次移动操作。每次移动操作表示从节点 a 移动到节点 b，经过的路径上的所有节点（包括起点和终点）都会被访问一次。求每个节点被访问的次数。

题目链接

[洛谷 P3258 [JLOI2014] 松鼠的新家](https://www.luogu.com.cn/problem/P3258)

输入格式

- 第一行包含一个整数 N，表示节点的数量。
- 第二行包含 N 个整数，表示访问节点的顺序。
- 接下来 N-1 行，每行包含两个整数 u 和 v，表示节点 u 和节点 v 之间有一条边。

输出格式

输出 N 行，第 i 行表示节点 i 被访问的次数。

样例输入

```
5
1 2 3 4 5
```

1 2
2 3
2 4
4 5
```

## 样例输出

```
1
2
1
2
1
```

数据范围

- 1 <= N <= 300000

解题思路

这是一道典型的树上点差分问题。我们需要统计树上每条路径经过的节点次数。

算法分析

1. **树上点差分**：
   - 对于树上从节点 u 到节点 v 的路径，我们需要将路径上所有节点的访问次数加 1。
   - 使用树上点差分技术，我们可以在 O(1)时间内完成这个操作：
     - diff[u] += 1
     - diff[v] += 1
     - diff[lca(u,v)] -= 1
     - diff[parent[lca(u,v)]] -= 1
   - 最后通过一次 DFS 遍历，计算每个节点的子树和，得到最终结果。

2. **LCA 计算**：
   - 使用倍增法预处理 LCA，时间复杂度为 O(N log N)预处理，O(log N)查询。

3. **特殊处理**：
   - 注意每次移动的终点也是下一次移动的起点，除了最后一次移动，所以需要对除最后一个节点外的所有节点的访问次数减 1。

时间复杂度
```

- 预处理阶段：O(N log N)（倍增法求LCA）
- 修改操作：O(1)每次
- 查询结果：O(N)（DFS遍历）
- 总体复杂度：O(N log N)

空间复杂度

- 树存储：O(N)
- LCA相关数组：O(N log N)
- 差分数组：O(N)
- 总体复杂度：O(N log N)

代码实现

Java版本

```java
// 详细实现见Code08_SquirrelNewHome.java
```

C++版本

```cpp
// 详细实现见Code08_SquirrelNewHome.cpp
```

Python版本

```python
# 详细实现见Code08_SquirrelNewHome.py
```

算法要点

1. **树上点差分**：
   - 点差分用于处理对路径上所有点的修改操作。
   - 对于路径u到v：
     ```
     diff[u] += d
     diff[v] += d
     diff[lca(u,v)] -= d
     diff[father[lca(u,v)]] -= d
     ```

```

2. **LCA 算法**：
   - 使用倍增法预处理 LCA，预处理时间复杂度 O(N log N)，查询时间复杂度 O(log N)。

3. **DFS 遍历**：
   - 通过 DFS 遍历计算子树和，得到最终结果。

## 常见错误与注意事项

1. **LCA 计算错误**：
   - 倍增数组初始化错误
   - 深度比较错误
   - 跳跃过程错误

2. **差分标记错误**：
   - 点差分和边差分混淆
   - 标记位置错误
   - 系数错误

3. **边界处理错误**：
   - 根节点的特殊处理
   - LCA 为端点的特殊情况未考虑
   - 数组越界

4. **特殊处理**：
   - 每次移动的终点也是下一次移动的起点，需要特殊处理

## 扩展应用

1. **树上边差分**：
   - 处理对路径上所有边的修改操作。
   - 对于路径 u 到 v：
     ```
     diff[u] += d
     diff[v] += d
     diff[lca(u,v)] -= 2 * d
     ```

2. **树链剖分 + 线段树**：
   - 对于更复杂的操作，可以结合树链剖分和线段树：
     - 支持区间修改和区间查询
     - 更灵活的操作支持
     ```

    - 但代码复杂度较高

3. **树上启发式合并**：
   - 对于子树信息合并问题：
     - 时间复杂度优化到O(N log N)
     - 适用于特定类型的树上问题

相关题目

1. **洛谷 P3128 [USACO15DEC] Max Flow P** - 树上点差分模板题
2. **LOJ 10131 暗的连锁** - 树上边差分
3. **POJ 3417 Network** - 树上边差分
4. **洛谷 P2680 运输计划** - 树上边差分+二分

总结

树上点差分是处理树上路径修改问题的重要算法技巧，通过将路径修改转化为端点标记，大大优化了时间复杂度。掌握树上点差分需要：

1. 深入理解差分思想
2. 熟练掌握LCA算法
3. 灵活应用点差分
4. 注意实现细节和边界条件

通过系统学习和大量练习，可以熟练掌握这一重要算法技巧，为解决更复杂的树上问题打下坚实基础。

==========

文件: README_POJ3263.md
==========
POJ 3263 Tallest Cow 题目解析

题目描述

有N头牛站成一行，两头牛能够相互看见，当且仅当它们中间的牛身高都比它们矮。已知最高的牛是第P头，身高为H，还知道R对关系，每对关系表示两头牛可以相互看见。求每头牛的身高最大可能是多少。

算法思路

这是一道经典的差分数组应用题，虽然不是树上差分，但体现了差分思想的核心：将区间修改转化为端点标记。

问题分析

1. 初始时，所有牛的身高都可以设为最高身高 H（题目要求最大可能身高）
2. 对于每对可以相互看见的牛(a,b)，它们之间的牛都必须比它们矮
3. 因此，区间(min(a,b)+1, max(a,b)-1)内的牛身高都要减 1
4. 最终每头牛的身高 = H + 相对身高变化值

核心思想

使用差分数组优化区间修改操作：
- 对于区间[l,r]内的元素都减去 d，可以在差分数组中：
  - diff[l] -= d
  - diff[r+1] += d
- 最后通过前缀和还原原数组

在本题中：
- 对于每对关系(a,b)，在区间(a+1, b-1)内的牛身高减 1
- 即在差分数组中：
  - diff[a+1] -= 1
  - diff[b] += 1

特殊处理

1. **去重处理**：可能存在重复的关系，需要避免重复处理
2. **边界处理**：确保 a < b，便于统一处理
3. **区间有效性**：只有当|a-b| > 1 时，中间才有牛需要处理

复杂度分析

时间复杂度
1. **输入处理**：O(R)，需要处理 R 对关系
2. **差分操作**：O(1)每次，总共 O(R)
3. **前缀和计算**：O(N)，计算每头牛的最终身高
4. **去重检查**：O(log R)每次（使用 set），总共 O(R log R)

总时间复杂度：O(R log R + N)

空间复杂度
- 差分数组：O(N)
- 去重集合：O(R)
- 其他辅助变量：O(1)

总空间复杂度：O(N + R)

解题技巧

1. 差分数组应用
差分数组是处理区间修改的经典技巧：
```java
// 区间[l,r]都加上 d
diff[l] += d;
diff[r+1] -= d;

// 还原原数组（前缀和）
int value = 0;
for (int i = 1; i <= n; i++) {
    value += diff[i];
    result[i] = value;
}
```

2. 去重技巧
使用 set 或者哈希表避免重复处理相同关系：
```java
// Java 中使用 Set<Long>存储配对关系
long pair = (long) Math.min(a, b) * (n + 1) + Math.max(a, b);
if (seen.contains(pair)) continue;
seen.add(pair);

// Python 中使用 set 存储元组
pair = (min(a, b), max(a, b))
if pair in seen: continue
seen.add(pair)
```

3. 边界处理
统一处理方式，确保 a < b：
```java
if (a > b) {
    int temp = a;
    a = b;
    b = temp;
}
```

代码实现要点

Java 实现要点
1. 使用 BufferedReader 和 StreamTokenizer 提高输入效率
2. 使用 HashSet 去重
3. 使用 long 类型避免配对哈希冲突

C++实现要点
1. 使用 scanf/printf 提高输入输出效率
2. 使用 set 去重
3. 注意 long long 类型的使用

Python 实现要点
1. 使用 sys.stdin.readline()提高输入效率
2. 使用 set 存储元组实现去重
3. 利用 Python 的多元赋值简化交换操作

边界条件和异常处理

1. **空关系**：R=0 时，所有牛身高都为 H
2. **相邻牛**：|a-b|=1 时，中间没有牛需要处理
3. **重复关系**：需要去重处理
4. **相同牛**：a=b 时，属于无效输入，但题目保证不会出现

工程化考量

1. **输入输出优化**：
   - Java 使用 BufferedReader 和 StreamTokenizer
   - C++使用 scanf/printf 而非 cin/cout
   - Python 使用 sys.stdin.readline()

2. **内存管理**：
   - 合理设置数组大小，避免浪费空间
   - 及时释放不需要的资源

3. **可读性和维护性**：
   - 添加详细注释说明算法思路
   - 使用有意义的变量名
   - 模块化设计，分离不同功能

相关题目和扩展

1. **类似题目**：
   - 洛谷 P2879 [USACO07JAN]区间统计 Tallest Cow（同一题目）
   - 一维差分数组模板题

   - 二维差分数组应用题

2. **扩展应用**：
   - 树上差分：将差分思想应用到树结构上
   - 二维差分：处理矩阵区间修改问题
   - 离散化+差分：处理值域较大的区间修改问题

总结

POJ 3263 Tallest Cow 是一道经典的差分数组应用题，虽然不涉及树结构，但体现了差分思想的核心：将区间修改转化为端点标记，从而将 O(N)的区间操作优化为 O(1)的端点操作。

通过这道题目，可以掌握以下知识点：
1. 差分数组的基本原理和应用
2. 区间修改问题的优化方法
3. 去重处理技巧
4. 输入输出优化方法

这为学习更复杂的树上差分奠定了基础。

==========

文件: SUMMARY.md
==========
树上差分算法学习总结

概述

树上差分是一种在树结构上应用差分思想的算法技巧，用于高效处理树上路径的区间修改问题。它是普通差分数组在树结构上的扩展应用，可以将 O(N)的路径修改操作优化为 O(1)的端点操作。

核心思想

差分思想回顾
普通差分数组通过在区间端点打标记的方式，将区间修改操作转化为端点操作：
- 对区间[l,r]都加上 d：diff[l] += d, diff[r+1] -= d
- 通过前缀和还原原数组

树上差分扩展
在树上，路径修改操作也可以通过类似的思想优化：
- 对树上从 u 到 v 的路径进行操作
- 在 u 和 v 处打标记，在 LCA(u,v)处消除影响
- 通过 DFS 遍历计算子树和得到最终结果

分类与应用

1. 树上点差分
处理对路径上所有点的修改操作。

操作方法：
对于路径 u 到 v：
```
diff[u] += d
diff[v] += d
diff[lca(u,v)] -= d
diff[father[lca(u,v)]] -= d
```

应用场景：
- 松鼠的新家（洛谷 P3258）
- 运输计划（洛谷 P2680）

2. 树上边差分
处理对路径上所有边的修改操作。

操作方法：
对于路径 u 到 v：
```
diff[u] += d
diff[v] += d
diff[lca(u,v)] -= 2 * d
```

应用场景：
- Max Flow（洛谷 P3128）
- 暗的连锁（LOJ 10131）
- Network（POJ 3417）

关键技术

1. 最近公共祖先(LCA)
树上差分通常需要配合 LCA 算法使用，用于找到路径的转折点。

常用实现方法：
- 倍增法：预处理 O(N log N)，查询 O(log N)
- Tarjan 离线算法：预处理 O(N + M)，查询 O(1)

- 树链剖分：预处理 O(N)，查询 O(log N)

2. 树的存储结构
通常使用链式前向星存储树结构，便于遍历。

3. DFS 遍历
通过 DFS 遍历计算子树和，得到最终结果。

算法复杂度

时间复杂度
- 预处理阶段：O(N log N)（倍增法求 LCA）
- 修改操作：O(1)每次
- 查询结果：O(N)（DFS 遍历）
- 总体复杂度：O(N log N + M log N + N) = O((N + M) log N)

空间复杂度
- 树存储：O(N)
- LCA 相关数组：O(N log N)
- 差分数组：O(N)
- 总体复杂度：O(N log N)

实现要点

1. 数据结构选择
- 使用链式前向星存储树结构
- 使用二维数组存储倍增信息
- 使用一维数组存储差分值

2. 边界条件处理
- 根节点的特殊处理
- LCA 为端点的特殊情况
- 重边和自环的处理

3. 输入输出优化
- 对于大数据量，使用快速输入输出
- Java 使用 BufferedReader 和 StreamTokenizer
- C++使用 scanf/printf
- Python 使用 sys.stdin.readline()

经典题目分析

1. LOJ 10131 暗的连锁

题目类型：树上边差分
核心思想：统计每条树边被非树边跨越的次数
解题技巧：边差分 + LCA + DFS 统计
相关文件：Code06_DarkLock1.java, Code06_DarkLock1.cpp, Code06_DarkLock1.py

2. POJ 3263 Tallest Cow
题目类型：线性差分（非树上差分）
核心思想：区间修改优化
解题技巧：差分数组 + 去重处理
相关文件：Code07_TallestCow.java, Code07_TallestCow.cpp, Code07_TallestCow.py, README_POJ3263.md

3. 洛谷 P3128 Max Flow
题目类型：树上点差分
核心思想：统计路径经过次数
解题技巧：点差分 + LCA + DFS 统计
相关文件：Code01_MaxFlow1.java, Code01_MaxFlow2.java

4. 洛谷 P3258 松鼠的新家
题目类型：树上点差分
核心思想：统计路径经过次数
解题技巧：点差分 + LCA + DFS 统计
相关文件：Code08_SquirrelNewHome.java, Code08_SquirrelNewHome.cpp, Code08_SquirrelNewHome.py, README_P3258.md

5. LOJ 10132 异象石
题目类型：虚树 + 树上差分
核心思想：动态维护圆上点集的最小生成树
解题技巧：虚树构建 + LCA + 差分统计
相关文件：Code09_AlienStone.java, Code09_AlienStone.cpp, Code09_AlienStone.py, README_LOJ10132.md

6. Codeforces 739B Alyona and a tree
题目类型：二分查找 + 树上差分
核心思想：统计满足距离条件的祖先节点数量
解题技巧：倍增法 + 二分查找 + 差分统计
相关文件：Code10_AlyonaAndTree.java, Code10_AlyonaAndTree.cpp, Code10_AlyonaAndTree.py, README_CF739B.md

7. 洛谷 P2680 运输计划
题目类型：树上边差分 + 二分答案
核心思想：通过二分最长路径，使用边差分统计需要修改的边数
解题技巧：边差分 + 二分答案 + LCA

相关文件：Code05_TransportPlan1.java, Code05_TransportPlan2.java, Code05_TransportPlan3.java

8. POJ 3417 Network
题目类型：树上边差分
核心思想：统计每条树边被非树边覆盖的次数
解题技巧：边差分 + LCA + DFS统计
相关文件：Code04_Network.java

9. AtCoder ABC187E. Through Path
题目类型：树上点差分 + DFS
核心思想：处理树上路径修改操作
解题技巧：点差分 + DFS遍历统计
相关文件：Code11_AtCoderABC187E.java, Code11_AtCoderABC187E.cpp, Code11_AtCoderABC187E.py

10. Codeforces 191C. Fools and Roads
题目类型：树上边差分
核心思想：统计每条边被经过的次数
解题技巧：边差分 + LCA + DFS统计
相关文件：Code12_Codeforces191C.java, Code12_Codeforces191C.cpp, Code12_Codeforces191C.py

11. LeetCode 1483. 树节点的第 K 个祖先
题目类型：树上倍增法应用
核心思想：预处理每个节点的2^k级祖先
解题技巧：倍增数组预处理 + 二进制拆分
相关文件：Code13_LeetCode1483.java, Code13_LeetCode1483.cpp, Code13_LeetCode1483.py

12. LeetCode 235. 二叉搜索树的最近公共祖先
题目类型：LCA基础应用
核心思想：利用二叉搜索树的性质快速找到LCA
解题技巧：根据值的大小关系遍历树
相关文件：Code14_LeetCode235.java, Code14_LeetCode235.cpp, Code14_LeetCode235.py

13. LeetCode 2646. 最小化旅行的价格
题目类型：树上差分 + 树形DP
核心思想：统计路径覆盖次数并决策节点减半
解题技巧：树上差分 + LCA + 树形DP
相关文件：Code15_LeetCode2646.java, Code15_LeetCode2646.cpp, Code15_LeetCode2646.py, README_LeetCode2646.md

14. Codeforces 519E. A and B and Lecture Rooms
题目类型：LCA + 树上距离分析
核心思想：找到到两个节点距离相等的节点数量
解题技巧：LCA + 距离计算 + 子树大小统计

相关文件：Code16_Codeforces519E.java, Code16_Codeforces519E.cpp, Code16_Codeforces519E.py, README_CF519E.md

15. LeetCode 2096. 从二叉树一个节点到另一个节点的方向
题目类型：二叉树路径方向
核心思想：找到最短路径并转换为方向字符串
解题技巧：LCA + 路径重建 + 方向转换
相关文件：Code17_LeetCode2096.java, Code17_LeetCode2096.cpp, Code17_LeetCode2096.py, README_LeetCode2096.md

16. 洛谷 P3379 最近公共祖先
题目类型：LCA 模板题
核心思想：倍增法求最近公共祖先
解题技巧：深度优先搜索预处理 + 倍增跳跃

17. HDU 2874 Connections between cities
题目类型：树上路径查询
核心思想：利用 LCA 计算两点间距离
解题技巧：深度数组 + 距离数组 + LCA

扩展应用

1. 树链剖分 + 线段树
对于更复杂的操作，可以结合树链剖分和线段树：
- 支持区间修改和区间查询
- 更灵活的操作支持
- 但代码复杂度较高

2. 树上启发式合并
对于子树信息合并问题：
- 时间复杂度优化到 O(N log N)
- 适用于特定类型的树上问题

3. 点分治
对于树上路径统计问题：
- 通过点分治将树上问题转化为序列问题
- 结合容斥原理处理

4. 动态树（Link-Cut Tree）
对于动态树问题：
- 支持动态连接和断开边
- 维护树的连通性和路径信息
- 时间复杂度 O(log N) per operation

5. 虚树
对于树上的关键点处理：
- 只保留必要的节点构建虚树
- 大幅降低问题规模
- 与树上差分结合处理大规模数据

6. 与机器学习的结合
- 树结构在图神经网络中的应用
- 差分思想在梯度下降中的体现
- 树模型（决策树、随机森林）中的路径分析

学习建议

1. 掌握基础
- 先掌握差分数组的基本概念和线性应用
- 理解树的遍历方式（DFS、BFS）及其特性
- 学习 LCA 算法的多种实现（倍增法、Tarjan 离线算法等）
- 熟悉链式前向星等树的存储结构

2. 循序渐进
- 从简单的树上差分问题开始（如洛谷 P3128）
- 学习点差分与边差分的区别与应用场景
- 尝试结合其他算法的复杂问题（如二分答案、虚树等）

3. 实战练习
- 多做相关题目巩固理解，跨平台练习
- 尝试用不同语言实现同一算法，理解语言特性差异
- 对经典题目进行变形思考，提升应变能力

4. 深入理解
- 分析算法的时间和空间复杂度
- 思考算法的优化空间
- 理解为什么这个算法是最优解

常见错误与注意事项

1. LCA 计算错误
- 倍增数组初始化错误（特别是根节点的处理）
- 跳跃过程中的逻辑错误（未正确处理节点深度）
- 边界情况处理不当（如节点是另一个节点的祖先）

2. 差分标记错误

- 点差分与边差分的标记方式混淆
- 忘记处理 LCA 节点或 LCA 的父节点
- 标记后未正确传播（DFS 回溯时未累加子节点信息）

3. 边界条件处理错误
- 根节点的处理（父节点不存在）
- 空树或单节点树的特殊情况
- 大数据规模下的栈溢出问题（递归深度过深）

4. 输入输出效率问题
- 对于大规模数据，未使用快速 IO
- 数据读取顺序错误导致树的构建失败

5. 内存管理问题
- 数组大小不足导致越界
- 重复内存分配导致性能下降

工程化考量

1. 代码健壮性
- 输入验证：检查输入数据的合法性
- 异常处理：妥善处理各种边界情况
- 防御性编程：避免空指针、数组越界等问题

2. 性能优化
- 输入输出优化：使用 BufferedReader/Scanner 等高效 IO
- 递归深度控制：对于深树考虑使用非递归 DFS
- 内存复用：避免频繁的内存分配和释放

3. 代码可读性与可维护性
- 变量命名规范：使用有意义的变量名
- 代码模块化：将不同功能拆分为独立函数
- 详细注释：说明算法原理、实现细节和优化点

4. 跨语言实现考量
- 语言特性差异：递归深度限制、栈大小等
- 数据类型选择：根据数据规模选择合适的整数类型
- 内存管理差异：手动管理 vs 自动垃圾回收

与其他领域的联系

1. 与机器学习的结合
- 树结构在图神经网络中的应用

- 差分思想在梯度下降中的体现
- 树模型中的路径分析和特征提取

2. 与大数据处理的结合
- 在大规模树结构中进行高效的路径查询
- 分布式环境下的树上算法实现
- 流处理中的增量树更新

3. 与软件工程的联系
- 版本控制系统中的差异比较（类似差分思想）
- 树结构在软件工程中的应用（DOM 树、依赖树等）
- 性能监控中的增量分析

总结

树上差分是一种高效的树上路径修改与查询算法，通过将路径操作转化为点标记操作，大幅提升算法效率。它在解决树上路径覆盖、统计等问题上具有显著优势，时间复杂度通常为 O(N log N)，空间复杂度为 O(N log N)，是处理树上问题的重要工具。

掌握树上差分算法需要深入理解其核心思想、实现细节和应用场景，并能够灵活运用到各种复杂问题中。同时，还需要关注算法的工程化实现，包括代码健壮性、性能优化、可读性和可维护性等方面。

在实际应用中，树上差分常与 LCA、二分答案、虚树等技术结合使用，可以解决众多复杂的树上问题，是算法竞赛和软件工程中的重要工具。

==========

[代码文件]
==========
文件: Code01_MaxFlow1.java
==========
```
package class122;

/**
 * 树上点差分模版(递归版)
 *
 * 题目来源：洛谷 P3128 [USACO15DEC] Max Flow P
 * 题目链接：https://www.luogu.com.cn/problem/P3128
 *
 * 题目描述：
 * 给定一棵包含 N 个节点的树，以及 K 次操作。每次操作需要将一条路径上的所有点的点权加 1。
 * 所有操作完成后，返回树上点权的最大值。
 *
```

```
 * 算法原理：树上点差分
 * 树上点差分是普通差分数组在树结构上的扩展应用，用于高效处理树上路径的区间修改问题。
 * 对于每次路径 u 到 v 的操作：
 * 1. diff[u] += 1
 * 2. diff[v] += 1
 * 3. diff[lca(u,v)] -= 1
 * 4. diff[father[lca(u,v)]] -= 1
 * 最后通过一次 DFS 回溯累加子节点的差分标记，得到每个节点的最终点权。
 *
 * 时间复杂度分析：
 * - 预处理 LCA：O(N log N)
 * - 差分标记：O(K log N)，其中 K 是操作次数，每次需要计算 LCA
 * - DFS 回溯统计：O(N)
 * 总时间复杂度：O((N + K) log N)
 *
 * 空间复杂度分析：
 * - 树的存储：O(N)
 * - LCA 倍增数组：O(N log N)
 * - 差分数组：O(N)
 * 总空间复杂度：O(N log N)
 *
 * 工程化考量：
 * 1. 使用链式前向星存储树结构，提高空间效率和遍历速度
 * 2. 使用 StreamTokenizer 进行高效输入，处理大量数据时性能优于 Scanner
 * 3. 使用 PrintWriter 进行高效输出，支持缓冲
 * 4. 采用静态成员变量减少对象创建，在算法竞赛中常用此技巧
 *
 * 最优解分析：
 * 树上点差分是解决此类问题的最优解，通过 O(1)的操作标记每条路径的影响范围，
 * 避免了暴力遍历每条路径上的所有节点，时间复杂度比暴力方法的 O(K*N)有极大提升。
 */
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;
import java.io.PrintWriter;
import java.io.StreamTokenizer;
import java.util.Arrays;

public class Code01_MaxFlow1 {

    /**
     * 最大节点数量
```

```
 * 题目中节点数量范围：2 <= N <= 50000
 */
public static int MAXN = 50001;

/**
 * 倍增数组的层数限制
 * 2^16 = 65536 > 50000，足够处理题目中的最大节点数
 */
public static int LIMIT = 16;

/**
 * 最大幂次，根据节点数量动态计算
 */
public static int power;

/**
 * 计算 log2(n)的整数部分
 *
 * @param n 输入整数
 * @return log2(n)的整数部分
 */
public static int log2(int n) {
    int ans = 0;
    while ((1 << ans) <= (n >> 1)) {
        ans++;
    }
    return ans;
}

/**
 * 差分数组
 * num[i]表示节点 i 被路径覆盖的次数的差分值
 */
public static int[] num = new int[MAXN];

/**
 * 链式前向星存储树结构
 * head[u]: 节点 u 的第一个边的索引
 * next[e]: 边 e 的下一个边的索引
 * to[e]: 边 e 指向的节点
 * cnt: 当前可用的边索引
 */
public static int[] head = new int[MAXN];
```

```
public static int[] next = new int[MAXN << 1];
public static int[] to = new int[MAXN << 1];
public static int cnt;

/**
 * LCA 相关数组
 * deep[u]: 节点 u 的深度
 * stjump[u][p]: 节点 u 的 2^p 级祖先
 */
public static int[] deep = new int[MAXN];
public static int[][] stjump = new int[MAXN][LIMIT];

/**
 * 初始化算法所需的数据结构
 * 设置数组初始值，准备处理新的测试用例
 *
 * @param n 节点数量
 */
public static void build(int n) {
    power = log2(n);
    // 初始化差分数组，从 1 开始因为节点编号从 1 开始
    Arrays.fill(num, 1, n + 1, 0);
    // 边索引从 1 开始，0 表示没有边
    cnt = 1;
    // 初始化链式前向星的 head 数组
    Arrays.fill(head, 1, n + 1, 0);
}

/**
 * 向链式前向星中添加一条无向边
 *
 * @param u 边的一个端点
 * @param v 边的另一个端点
 */
public static void addEdge(int u, int v) {
    // 添加 u 到 v 的边
    next[cnt] = head[u];
    to[cnt] = v;
    head[u] = cnt++;
}

/**
 * 第一次 DFS，预处理每个节点的深度和倍增跳跃数组
```

```
     *
     * @param u 当前处理的节点
     * @param f 当前节点的父节点
     *
     * 时间复杂度：O(N log N)
     * 空间复杂度：O(log N) - 递归调用栈深度
     */
    public static void dfs1(int u, int f) {
        // 设置当前节点的深度，父节点深度+1
        deep[u] = deep[f] + 1;
        // 设置当前节点的直接父节点（2^0 级祖先）
        stjump[u][0] = f;

        // 预处理倍增数组
        // 利用动态规划思想：u 的 2^p 级祖先 = u 的 2^(p-1)级祖先的 2^(p-1)级祖先
        for (int p = 1; p <= power; p++) {
            stjump[u][p] = stjump[stjump[u][p - 1]][p - 1];
        }

        // 递归处理所有子节点
        for (int e = head[u]; e != 0; e = next[e]) {
            if (to[e] != f) {
                dfs1(to[e], u);
            }
        }
    }

    /**
     * 使用倍增法计算两个节点的最近公共祖先
     *
     * @param a 第一个节点
     * @param b 第二个节点
     * @return a 和 b 的最近公共祖先
     *
     * 时间复杂度：O(log N)
     * 空间复杂度：O(1)
     */
    public static int lca(int a, int b) {
        // 确保 a 的深度不小于 b
        if (deep[a] < deep[b]) {
            int tmp = a;
            a = b;
            b = tmp;
```

```
    }

    // 将 a 向上跳到与 b 同一深度
    for (int p = power; p >= 0; p--) {
        if (deep[stjump[a][p]] >= deep[b]) {
            a = stjump[a][p];
        }
    }

    // 如果此时 a==b，则找到了 LCA
    if (a == b) {
        return a;
    }

    // 继续向上跳，直到找到 LCA
    for (int p = power; p >= 0; p--) {
        if (stjump[a][p] != stjump[b][p]) {
            a = stjump[a][p];
            b = stjump[b][p];
        }
    }

    // 返回它们的父节点
    return stjump[a][0];
}

/**
 * 第二次 DFS，计算每个节点被覆盖的次数，并找出最大值
 *
 * @param u 当前处理的节点
 * @param f 当前节点的父节点
 *
 * 时间复杂度：O(N)
 * 空间复杂度：O(log N) - 递归调用栈深度
 *
 * 算法逻辑：
 * 1. 先递归处理所有子节点
 * 2. 累加子节点的覆盖次数到当前节点
 * 3. 当前节点的最终覆盖次数即为答案
 */
public static void dfs2(int u, int f) {
    // 递归处理所有子节点
    for (int e = head[u], v; e != 0; e = next[e]) {
```

```
            v = to[e];
            if (v != f) {
                dfs2(v, u);
            }
        }

        // 将子节点的覆盖次数累加到父节点，完成差分标记的传播
        for (int e = head[u], v; e != 0; e = next[e]) {
            v = to[e];
            if (v != f) {
                num[u] += num[v];
            }
        }
    }

    /**
     * 主函数，处理输入输出并调用相应的算法函数
     *
     * @param args 命令行参数
     * @throws IOException 输入输出异常
     */
    public static void main(String[] args) throws IOException {
        // 使用高效的输入方式
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        StreamTokenizer in = new StreamTokenizer(br);
        // 使用高效的输出方式
        PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));

        // 读取节点数和操作数
        in.nextToken();
        int n = (int) in.nval;
        in.nextToken();
        int m = (int) in.nval;

        // 初始化数据结构
        build(n);

        // 构建树结构，添加 n-1 条树边
        for (int i = 1, u, v; i < n; i++) {
            in.nextToken();
            u = (int) in.nval;
            in.nextToken();
            v = (int) in.nval;
```

```
    addEdge(u, v);
    addEdge(v, u);  // 无向图，添加反向边
}

// 预处理 LCA 所需的数据
// 以节点 1 为根节点进行 DFS
dfs1(1, 0);

// 处理所有操作，执行树上点差分操作
for (int i = 1, u, v, lca_node, lca_father; i <= m; i++) {
    in.nextToken();
    u = (int) in.nval;
    in.nextToken();
    v = (int) in.nval;
    // 计算 u 和 v 的 LCA
    lca_node = lca(u, v);
    // 计算 LCA 的父节点
    lca_father = stjump[lca_node][0];

    /**
     * 树上点差分核心操作
     * 对于路径(u,v)，它在原树上会形成一条路径
     * 通过点差分，我们标记路径上的所有节点
     * 1. 在 u 处+1
     * 2. 在 v 处+1
     * 3. 在 LCA 处-1
     * 4. 在 LCA 的父节点处-1
     */
    num[u]++;
    num[v]++;
    num[lca_node]--;
    num[lca_father]--;
}

// 计算每个节点的最终覆盖次数
dfs2(1, 0);

// 找出最大的覆盖次数
int max = 0;
for (int i = 1; i <= n; i++) {
    max = Math.max(max, num[i]);
}
```

```
        // 输出结果
        out.println(max);
        out.flush();
        out.close();
        br.close();
    }

}
```

==========

文件: Code01_MaxFlow2.java

==========

```
package class122;

/**
 * 树上点差分模版(迭代版)
 *
 * 题目来源：洛谷 P3128 [USACO15DEC] Max Flow P
 * 题目链接：https://www.luogu.com.cn/problem/P3128
 *
 * 题目描述：
 * 给定一棵包含 N 个节点的树，以及 K 次操作。每次操作需要将一条路径上的所有点的点权加 1。
 * 所有操作完成后，返回树上点权的最大值。
 *
 * 算法原理：树上点差分（迭代版）
 * 树上点差分是普通差分数组在树结构上的扩展应用，用于高效处理树上路径的区间修改问题。
 * 对于每次路径 u 到 v 的操作：
 * 1. diff[u] += 1
 * 2. diff[v] += 1
 * 3. diff[lca(u,v)] -= 1
 * 4. diff[father[lca(u,v)]] -= 1
 * 最后通过一次 DFS 回溯累加子节点的差分标记，得到每个节点的最终点权。
 *
 * 时间复杂度分析：
 * - 预处理 LCA：O(N log N)
 * - 差分标记：O(K log N)，其中 K 是操作次数，每次需要计算 LCA
 * - DFS 回溯统计：O(N)
 * 总时间复杂度：O((N + K) log N)
 *
 * 空间复杂度分析：
 * - 树的存储：O(N)
 * - LCA 倍增数组：O(N log N)
```

```
 * - 差分数组：O(N)
 * - 迭代 DFS 栈：O(N)
 * 总空间复杂度：O(N log N)
 *
 * 工程化考量：
 * 1. 使用链式前向星存储树结构，提高空间效率和遍历速度
 * 2. 使用迭代 DFS 替代递归 DFS，避免在大数据量下栈溢出问题
 * 3. 使用 StreamTokenizer 进行高效输入，处理大量数据时性能优于 Scanner
 * 4. 使用 PrintWriter 进行高效输出，支持缓冲
 * 5. 采用静态成员变量减少对象创建，在算法竞赛中常用此技巧
 *
 * 最优解分析：
 * 树上点差分是解决此类问题的最优解，通过 O(1)的操作标记每条路径的影响范围，
 * 避免了暴力遍历每条路径上的所有节点，时间复杂度比暴力方法的 O(K*N)有极大提升。
 * 迭代版相比递归版具有更好的栈安全性，适合处理大规模数据。
 */
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;
import java.io.PrintWriter;
import java.io.StreamTokenizer;
import java.util.Arrays;

public class Code01_MaxFlow2 {

    /**
     * 最大节点数量
     * 题目中节点数量范围：2 <= N <= 50000
     */
    public static int MAXN = 50001;

    /**
     * 倍增数组的层数限制
     * 2^16 = 65536 > 50000，足够处理题目中的最大节点数
     */
    public static int LIMIT = 16;

    /**
     * 最大幂次，根据节点数量动态计算
     */
    public static int power;
```

```
/**
 * 计算 log2(n)的整数部分
 *
 * @param n 输入整数
 * @return log2(n)的整数部分
 */
public static int log2(int n) {
    int ans = 0;
    while ((1 << ans) <= (n >> 1)) {
        ans++;
    }
    return ans;
}

/**
 * 差分数组
 * num[i]表示节点 i 被路径覆盖的次数的差分值
 */
public static int[] num = new int[MAXN];

/**
 * 链式前向星存储树结构
 * head[u]: 节点 u 的第一个边的索引
 * next[e]: 边 e 的下一个边的索引
 * to[e]: 边 e 指向的节点
 * cnt: 当前可用的边索引
 */
public static int[] head = new int[MAXN];
public static int[] next = new int[MAXN << 1];
public static int[] to = new int[MAXN << 1];
public static int cnt;

/**
 * LCA 相关数组
 * deep[u]: 节点 u 的深度
 * stjump[u][p]: 节点 u 的 2^p 级祖先
 */
public static int[] deep = new int[MAXN];
public static int[][] stjump = new int[MAXN][LIMIT];

/**
 * 初始化算法所需的数据结构
 * 设置数组初始值，准备处理新的测试用例
```

```
     *
     * @param n 节点数量
     */
    public static void build(int n) {
        power = log2(n);
        // 初始化差分数组，从 1 开始因为节点编号从 1 开始
        Arrays.fill(num, 1, n + 1, 0);
        // 边索引从 1 开始，0 表示没有边
        cnt = 1;
        // 初始化链式前向星的 head 数组
        Arrays.fill(head, 1, n + 1, 0);
    }

    /**
     * 向链式前向星中添加一条无向边
     *
     * @param u 边的一个端点
     * @param v 边的另一个端点
     */
    public static void addEdge(int u, int v) {
        // 添加 u 到 v 的边
        next[cnt] = head[u];
        to[cnt] = v;
        head[u] = cnt++;
    }

    /**
     * 用于迭代 DFS 的栈元素结构
     * ufe[i][0]: 当前节点
     * ufe[i][1]: 当前节点的父节点
     * ufe[i][2]: 当前处理的边索引
     */
    public static int[][] ufe = new int[MAXN][3];

    /**
     * 迭代 DFS 的栈大小和当前处理的节点、父节点、边索引
     */
    public static int stackSize, u, f, e;

    /**
     * 将节点信息压入栈中
     *
     * @param u 当前节点
```

```
     * @param f 当前节点的父节点
     * @param e 当前处理的边索引
     */
    public static void push(int u, int f, int e) {
        ufe[stackSize][0] = u;
        ufe[stackSize][1] = f;
        ufe[stackSize][2] = e;
        stackSize++;
    }

    /**
     * 从栈中弹出节点信息
     */
    public static void pop() {
        --stackSize;
        u = ufe[stackSize][0];
        f = ufe[stackSize][1];
        e = ufe[stackSize][2];
    }

    /**
     * 第一次 DFS 的迭代版，预处理每个节点的深度和倍增跳跃数组
     *
     * @param root 根节点
     *
     * 时间复杂度：O(N log N)
     * 空间复杂度：O(N) - 迭代栈空间
     */
    public static void dfs1(int root) {
        // 初始化栈
        stackSize = 0;
        // 将根节点压入栈，e=-1 表示第一次访问该节点
        push(root, 0, -1);

        // 当栈不为空时继续处理
        while (stackSize > 0) {
            // 弹出栈顶元素
            pop();

            // 如果是第一次访问该节点
            if (e == -1) {
                // 设置当前节点的深度，父节点深度+1
                deep[u] = deep[f] + 1;
```

```
            // 设置当前节点的直接父节点（2^0 级祖先）
            stjump[u][0] = f;

            // 预处理倍增数组
            // 利用动态规划思想：u 的 2^p 级祖先 = u 的 2^(p-1)级祖先的 2^(p-1)级祖先
            for (int p = 1; p <= power; p++) {
                stjump[u][p] = stjump[stjump[u][p - 1]][p - 1];
            }

            // 设置当前节点的第一个邻接边
            e = head[u];
        } else {
            // 如果不是第一次访问该节点，处理下一条邻接边
            e = next[e];
        }

        // 如果还有未处理的邻接边
        if (e != 0) {
            // 将当前节点状态重新压入栈
            push(u, f, e);
            // 如果邻接边指向的节点不是父节点
            if (to[e] != f) {
                // 将邻接边指向的节点压入栈，e=-1 表示第一次访问
                push(to[e], u, -1);
            }
        }
    }
}

/**
 * 使用倍增法计算两个节点的最近公共祖先
 *
 * @param a 第一个节点
 * @param b 第二个节点
 * @return a 和 b 的最近公共祖先
 *
 * 时间复杂度：O(log N)
 * 空间复杂度：O(1)
 */
public static int lca(int a, int b) {
    // 确保 a 的深度不小于 b
    if (deep[a] < deep[b]) {
        int tmp = a;
```

```
        a = b;
        b = tmp;
    }

    // 将 a 向上跳到与 b 同一深度
    for (int p = power; p >= 0; p--) {
        if (deep[stjump[a][p]] >= deep[b]) {
            a = stjump[a][p];
        }
    }

    // 如果此时 a==b，则找到了 LCA
    if (a == b) {
        return a;
    }

    // 继续向上跳，直到找到 LCA
    for (int p = power; p >= 0; p--) {
        if (stjump[a][p] != stjump[b][p]) {
            a = stjump[a][p];
            b = stjump[b][p];
        }
    }

    // 返回它们的父节点
    return stjump[a][0];
}

/**
 * 第二次 DFS 的迭代版，计算每个节点被覆盖的次数，并找出最大值
 *
 * @param root 根节点
 *
 * 时间复杂度：O(N)
 * 空间复杂度：O(N) - 迭代栈空间
 *
 * 算法逻辑：
 * 1. 使用栈模拟递归 DFS 过程
 * 2. 先递归处理所有子节点
 * 3. 累加子节点的覆盖次数到当前节点
 * 4. 当前节点的最终覆盖次数即为答案
 */
public static void dfs2(int root) {
```

```
    // 初始化栈
    stackSize = 0;
    // 将根节点压入栈，e=-1 表示第一次访问该节点
    push(root, 0, -1);

    // 当栈不为空时继续处理
    while (stackSize > 0) {
        // 弹出栈顶元素
        pop();

        // 如果是第一次访问该节点
        if (e == -1) {
            // 设置当前节点的第一个邻接边
            e = head[u];
        } else {
            // 如果不是第一次访问该节点，处理下一条邻接边
            e = next[e];
        }

        // 如果还有未处理的邻接边
        if (e != 0) {
            // 将当前节点状态重新压入栈
            push(u, f, e);
            // 如果邻接边指向的节点不是父节点
            if (to[e] != f) {
                // 将邻接边指向的节点压入栈，e=-1 表示第一次访问
                push(to[e], u, -1);
            }
        } else {
            // 如果所有邻接边都已处理完，累加子节点的覆盖次数
            for (int e = head[u], v; e != 0; e = next[e]) {
                v = to[e];
                if (v != f) {
                    num[u] += num[v];
                }
            }
        }
    }
}

/**
 * 主函数，处理输入输出并调用相应的算法函数
 *
```

```
 * @param args 命令行参数
 * @throws IOException 输入输出异常
 */
public static void main(String[] args) throws IOException {
    // 使用高效的输入方式
    BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
    StreamTokenizer in = new StreamTokenizer(br);
    // 使用高效的输出方式
    PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));

    // 读取节点数和操作数
    in.nextToken();
    int n = (int) in.nval;
    build(n);
    in.nextToken();
    int m = (int) in.nval;

    // 构建树结构，添加 n-1 条树边
    for (int i = 1, u, v; i < n; i++) {
        in.nextToken();
        u = (int) in.nval;
        in.nextToken();
        v = (int) in.nval;
        addEdge(u, v);
        addEdge(v, u);  // 无向图，添加反向边
    }

    // 预处理 LCA 所需的数据
    // 以节点 1 为根节点进行 DFS
    dfs1(1);

    // 处理所有操作，执行树上点差分操作
    for (int i = 1, u, v, lca_node, lca_father; i <= m; i++) {
        in.nextToken();
        u = (int) in.nval;
        in.nextToken();
        v = (int) in.nval;
        // 计算 u 和 v 的 LCA
        lca_node = lca(u, v);
        // 计算 LCA 的父节点
        lca_father = stjump[lca_node][0];

        /**
```

```
            * 树上点差分核心操作
            * 对于路径(u,v)，它在原树上会形成一条路径
            * 通过点差分，我们标记路径上的所有节点
            * 1. 在u处+1
            * 2. 在v处+1
            * 3. 在LCA处-1
            * 4. 在LCA的父节点处-1
            */
            num[u]++;
            num[v]++;
            num[lca_node]--;
            num[lca_father]--;
        }

        // 计算每个节点的最终覆盖次数
        dfs2(1);

        // 找出最大的覆盖次数
        int max = 0;
        for (int i = 1; i <= n; i++) {
            max = Math.max(max, num[i]);
        }

        // 输出结果
        out.println(max);
        out.flush();
        out.close();
        br.close();
    }

}
```

==========

文件: Code02_SquirrelHome1.java

==========

```
package class122;

/**
 * 松鼠的新家(递归版)
 *
 * 题目来源：洛谷 P3258 [JLOI2014] 松鼠的新家
 * 题目链接：https://www.luogu.com.cn/problem/P3258
```

*
* 题目描述：
* 松鼠的新家是一棵树，前几天刚刚装修了新家，新家有 n 个房间，并且有 n-1 根树枝连接，
* 每个房间都可以相互到达，且俩个房间之间的路线都是唯一的。
* 天哪，他居然真的住在"树"上。
* 松鼠想邀请他的好友来玩，恰巧他的好友也是一只松鼠，住在森林的深处。
* 毫无疑问，这只懒惰的松鼠是不愿意亲自到新家的，他要求松鼠从他的家中把糖果送过去。
* 松鼠的家在节点 a，好友的家在节点 b，松鼠必须按照顺序访问节点 c1,c2,...,cm。
* 为了保证送糖果的效率，他想一次性把所有糖果都送到，即按照顺序访问所有节点。
* 每次移动到相邻节点时，松鼠必须消耗一个糖果。
* 为了保证松鼠不会饿着肚子上路，节点 ci 必须提供一个糖果（除了最后一个节点 cm）。
* 给定树的结构和访问顺序，求每个节点最少需要准备多少糖果。
*
* 算法原理：树上点差分 + Tarjan 离线 LCA
* 这是一个树上点差分的典型应用，但与普通的树上点差分不同的是，
* 这里的路径是按顺序访问的，而不是简单的两点间路径。
*
* 解题思路：
* 1. 对于顺序访问的节点序列 c1,c2,...,cm，我们需要在路径 ci->ci+1 上增加 1 的点权
* 2. 但要注意，除了最后一个节点 cm，其他节点都需要消耗一个糖果
* 3. 使用 Tarjan 算法离线计算所有相邻节点对的 LCA
* 4. 使用树上点差分标记路径，然后通过 DFS 回溯计算每个节点的最终值
*
* 时间复杂度分析：
* - 建图：O(N)
* - Tarjan 离线 LCA：O(N + M)，其中 M 是查询数量
* - 树上点差分标记：O(M)
* - DFS 回溯统计：O(N)
* 总时间复杂度：O(N + M)
*
* 空间复杂度分析：
* - 树的存储：O(N)
* - 查询存储：O(M)
* - 并查集：O(N)
* - 差分数组：O(N)
* 总空间复杂度：O(N + M)
*
* 工程化考量：
* 1. 使用链式前向星存储树结构和查询，提高空间效率和遍历速度
* 2. 使用 Tarjan 离线算法计算 LCA，避免多次在线查询的开销
* 3. 使用 StreamTokenizer 进行高效输入，处理大量数据时性能优于 Scanner
* 4. 使用 PrintWriter 进行高效输出，支持缓冲
* 5. 采用静态成员变量减少对象创建，在算法竞赛中常用此技巧

```
 *
 * 最优解分析：
 * 本题结合了 Tarjan 离线 LCA 和树上点差分两种技术，是解决此类问题的最优解。
 * 相比于在线 LCA 查询的方法，离线算法可以将时间复杂度从 O(M log N)优化到 O(N + M)。
 */
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;
import java.io.PrintWriter;
import java.io.StreamTokenizer;
import java.util.Arrays;

public class Code02_SquirrelHome1 {

    /**
     * 最大节点数量
     * 题目中节点数量范围：1 <= N <= 300000
     */
    public static int MAXN = 300001;

    /**
     * 依次去往节点的顺序
     * travel[i]表示第 i 个访问的节点编号
     */
    public static int[] travel = new int[MAXN];

    /**
     * 每个节点需要分配多少糖果
     * num[i]表示节点 i 需要准备的糖果数量
     */
    public static int[] num = new int[MAXN];

    /**
     * 链式前向星建图 - 树边存储
     * headEdge[u]: 节点 u 的第一条树边索引
     * edgeNext[e]: 边 e 的下一条边索引
     * edgeTo[e]: 边 e 指向的节点
     * tcnt: 树边计数器
     */
    public static int[] headEdge = new int[MAXN];
    public static int[] edgeNext = new int[MAXN << 1];
    public static int[] edgeTo = new int[MAXN << 1];
```

```
public static int tcnt;

/**
 * 链式前向星建图 - 查询存储
 * headQuery[u]: 节点 u 的第一条查询索引
 * queryNext[e]: 查询 e 的下一条查询索引
 * queryTo[e]: 查询 e 的目标节点
 * queryIndex[e]: 查询 e 在原查询序列中的索引
 * qcnt: 查询计数器
 */
public static int[] headQuery = new int[MAXN];
public static int[] queryNext = new int[MAXN << 1];
public static int[] queryTo = new int[MAXN << 1];
public static int[] queryIndex = new int[MAXN << 1];
public static int qcnt;

/**
 * Tarjan 算法相关数组
 * visited[u]: 节点 u 是否已被访问
 * unionfind[u]: 节点 u 在并查集中的父节点
 * father[u]: 节点 u 在 DFS 过程中的父节点
 * ans[i]: 第 i 次查询的 LCA 结果
 */
public static boolean[] visited = new boolean[MAXN];
public static int[] unionfind = new int[MAXN];
public static int[] father = new int[MAXN];
public static int[] ans = new int[MAXN];

/**
 * 初始化算法所需的数据结构
 * 设置数组初始值，准备处理新的测试用例
 *
 * @param n 节点数量
 */
public static void build(int n) {
    // 初始化糖果数组
    Arrays.fill(num, 1, n + 1, 0);
    // 初始化边和查询计数器
    tcnt = qcnt = 1;
    // 初始化链式前向星的头数组
    Arrays.fill(headEdge, 1, n + 1, 0);
    Arrays.fill(headQuery, 1, n + 1, 0);
    // 初始化访问标记数组
```

```
    Arrays.fill(visited, 1, n + 1, false);
    // 初始化并查集，每个节点初始时是自己的代表元素
    for (int i = 1; i <= n; i++) {
        unionfind[i] = i;
    }
}

/**
 * 向链式前向星中添加一条无向树边
 *
 * @param u 边的一个端点
 * @param v 边的另一个端点
 */
public static void addEdge(int u, int v) {
    // 添加 u 到 v 的边
    edgeNext[tcnt] = headEdge[u];
    edgeTo[tcnt] = v;
    headEdge[u] = tcnt++;
}

/**
 * 向链式前向星中添加一条查询
 *
 * @param u 查询的起始节点
 * @param v 查询的目标节点
 * @param i 查询在原查询序列中的索引
 */
public static void addQuery(int u, int v, int i) {
    // 添加 u 到 v 的查询
    queryNext[qcnt] = headQuery[u];
    queryTo[qcnt] = v;
    queryIndex[qcnt] = i;
    headQuery[u] = qcnt++;
}

/**
 * 并查集的查找操作（带路径压缩）
 *
 * @param i 要查找的节点
 * @return 节点 i 所在集合的代表元素
 */
public static int find(int i) {
    if (i != unionfind[i]) {
```

```
        // 路径压缩：将查找路径上的所有节点直接连接到根节点
        unionfind[i] = find(unionfind[i]);
    }
    return unionfind[i];
}

/**
 * Tarjan 离线 LCA 算法
 *
 * @param u 当前处理的节点
 * @param f 当前节点的父节点
 */
public static void tarjan(int u, int f) {
    // 标记当前节点已被访问
    visited[u] = true;

    // 递归处理所有子节点
    for (int e = headEdge[u], v; e != 0; e = edgeNext[e]) {
        v = edgeTo[e];
        if (v != f) {
            tarjan(v, u);
        }
    }

    // 处理所有与当前节点相关的查询
    for (int e = headQuery[u], v; e != 0; e = queryNext[e]) {
        v = queryTo[e];
        // 如果查询的目标节点已被访问，则可以计算 LCA
        if (visited[v]) {
            // 两个节点的 LCA 就是它们所在集合的代表元素
            ans[queryIndex[e]] = find(v);
        }
    }

    // 将当前节点与父节点合并到同一集合
    unionfind[u] = f;
    // 记录当前节点的父节点
    father[u] = f;
}

/**
 * DFS 回溯计算每个节点的最终糖果数量
 *
```

```
     * @param u 当前处理的节点
     * @param f 当前节点的父节点
     */
    public static void dfs(int u, int f) {
        // 递归处理所有子节点
        for (int e = headEdge[u], v; e != 0; e = edgeNext[e]) {
            v = edgeTo[e];
            if (v != f) {
                dfs(v, u);
            }
        }

        // 将子节点的糖果数量累加到父节点
        for (int e = headEdge[u], v; e != 0; e = edgeNext[e]) {
            v = edgeTo[e];
            if (v != f) {
                num[u] += num[v];
            }
        }
    }

    /**
     * 主函数，处理输入输出并调用相应的算法函数
     *
     * @param args 命令行参数
     * @throws IOException 输入输出异常
     */
    public static void main(String[] args) throws IOException {
        // 使用高效的输入方式
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        StreamTokenizer in = new StreamTokenizer(br);
        // 使用高效的输出方式
        PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));

        // 读取节点数
        in.nextToken();
        int n = (int) in.nval;

        // 初始化数据结构
        build(n);

        // 读取访问顺序
        for (int i = 1; i <= n; i++) {
```

```
        in.nextToken();
        travel[i] = (int) in.nval;
    }

    // 构建树结构
    for (int i = 1, u, v; i < n; i++) {
        in.nextToken();
        u = (int) in.nval;
        in.nextToken();
        v = (int) in.nval;
        addEdge(u, v);
        addEdge(v, u);  // 无向图，添加反向边
    }

    // 添加所有查询（相邻节点对的LCA查询）
    for (int i = 1; i < n; i++) {
        // 添加正向查询
        addQuery(travel[i], travel[i + 1], i);
        // 添加反向查询
        addQuery(travel[i + 1], travel[i], i);
    }

    // 执行计算
    compute(n);

    // 输出每个节点需要的糖果数量
    for (int i = 1; i <= n; i++) {
        out.println(num[i]);
    }

    // 确保输出被刷新
    out.flush();
    // 关闭资源
    out.close();
    br.close();
}

/**
 * 执行核心计算逻辑
 *
 * @param n 节点数量
 */
public static void compute(int n) {
```

```
        // 使用 Tarjan 算法计算所有查询的 LCA
        tarjan(1, 0);

        // 对每一条路径执行树上点差分操作
        for (int i = 1, u, v, lca, lcafather; i < n; i++) {
            // 获取路径的两个端点
            u = travel[i];
            v = travel[i + 1];
            // 获取两个端点的 LCA
            lca = ans[i];
            // 获取 LCA 的父节点
            lcafather = father[lca];

            /**
             * 树上点差分核心操作
             * 对于路径(u,v)，它在原树上会形成一条路径
             * 通过点差分，我们标记路径上的所有节点
             * 1. 在 u 处+1
             * 2. 在 v 处+1
             * 3. 在 LCA 处-1
             * 4. 在 LCA 的父节点处-1
             */
            num[u]++;
            num[v]++;
            num[lca]--;
            num[lcafather]--;
        }

        // 通过 DFS 回溯计算每个节点的最终糖果数量
        dfs(1, 0);

        // 调整除最后一个节点外的所有访问节点的糖果数量
        // 因为这些节点在访问时需要消耗一个糖果
        for (int i = 2; i <= n; i++) {
            num[travel[i]]--;
        }
    }

}
```

==========

文件: Code02_SquirrelHome2.java

```
==========================================
package class122;

/**
 * 松鼠的新家(迭代版)
 *
 * 题目来源：洛谷 P3258 [JLOI2014] 松鼠的新家
 * 题目链接：https://www.luogu.com.cn/problem/P3258
 *
 * 题目描述：
 * 松鼠的新家是一棵树，前几天刚刚装修了新家，新家有 n 个房间，并且有 n-1 根树枝连接，
 * 每个房间都可以相互到达，且俩个房间之间的路线都是唯一的。
 * 天哪，他居然真的住在"树"上。
 * 松鼠想邀请他的好友来玩，恰巧他的好友也是一只松鼠，住在森林的深处。
 * 毫无疑问，这只懒惰的松鼠是不愿意亲自到新家的，他要求松鼠从他的家中把糖果送过去。
 * 松鼠的家在节点 a，好友的家在节点 b，松鼠必须按照顺序访问节点 c1,c2,...,cm。
 * 为了保证送糖果的效率，他想一次性把所有糖果都送到，即按照顺序访问所有节点。
 * 每次移动到相邻节点时，松鼠必须消耗一个糖果。
 * 为了保证松鼠不会饿着肚子上路，节点 ci 必须提供一个糖果（除了最后一个节点 cm）。
 * 给定树的结构和访问顺序，求每个节点最少需要准备多少糖果。
 *
 * 算法原理：树上点差分 + Tarjan 离线 LCA（迭代版）
 * 这是一个树上点差分的典型应用，但与普通的树上点差分不同的是，
 * 这里的路径是按顺序访问的，而不是简单的两点间路径。
 *
 * 解题思路：
 * 1. 对于顺序访问的节点序列 c1,c2,...,cm，我们需要在路径 ci->ci+1 上增加 1 的点权
 * 2. 但要注意，除了最后一个节点 cm，其他节点都需要消耗一个糖果
 * 3. 使用 Tarjan 算法离线计算所有相邻节点对的 LCA
 * 4. 使用树上点差分标记路径，然后通过 DFS 回溯计算每个节点的最终值
 *
 * 时间复杂度分析：
 * - 建图：O(N)
 * - Tarjan 离线 LCA：O(N + M)，其中 M 是查询数量
 * - 树上点差分标记：O(M)
 * - DFS 回溯统计：O(N)
 * 总时间复杂度：O(N + M)
 *
 * 空间复杂度分析：
 * - 树的存储：O(N)
 * - 查询存储：O(M)
 * - 并查集：O(N)
 * - 差分数组：O(N)
```

```
 * - 迭代栈：O(N)
 * 总空间复杂度：O(N + M)
 *
 * 工程化考量：
 * 1. 使用链式前向星存储树结构和查询，提高空间效率和遍历速度
 * 2. 使用 Tarjan 离线算法计算 LCA，避免多次在线查询的开销
 * 3. 使用迭代 DFS 替代递归 DFS，避免在大数据量下栈溢出问题
 * 4. 使用 StreamTokenizer 进行高效输入，处理大量数据时性能优于 Scanner
 * 5. 使用 PrintWriter 进行高效输出，支持缓冲
 * 6. 采用静态成员变量减少对象创建，在算法竞赛中常用此技巧
 *
 * 最优解分析：
 * 本题结合了 Tarjan 离线 LCA 和树上点差分两种技术，是解决此类问题的最优解。
 * 相比于在线 LCA 查询的方法，离线算法可以将时间复杂度从 O(M log N)优化到 O(N + M)。
 * 迭代版相比递归版具有更好的栈安全性，适合处理大规模数据。
 */
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;
import java.io.PrintWriter;
import java.io.StreamTokenizer;
import java.util.Arrays;

public class Code02_SquirrelHome2 {

    /**
     * 最大节点数量
     * 题目中节点数量范围：1 <= N <= 300000
     */
    public static int MAXN = 300001;

    /**
     * 依次去往节点的顺序
     * travel[i]表示第 i 个访问的节点编号
     */
    public static int[] travel = new int[MAXN];

    /**
     * 每个节点需要分配多少糖果
     * num[i]表示节点 i 需要准备的糖果数量
     */
    public static int[] num = new int[MAXN];
```

```
/**
 * 链式前向星建图 - 树边存储
 * headEdge[u]: 节点 u 的第一条树边索引
 * edgeNext[e]: 边 e 的下一条边索引
 * edgeTo[e]: 边 e 指向的节点
 * tcnt: 树边计数器
 */
public static int[] headEdge = new int[MAXN];
public static int[] edgeNext = new int[MAXN << 1];
public static int[] edgeTo = new int[MAXN << 1];
public static int tcnt;

/**
 * 链式前向星建图 - 查询存储
 * headQuery[u]: 节点 u 的第一条查询索引
 * queryNext[e]: 查询 e 的下一条查询索引
 * queryTo[e]: 查询 e 的目标节点
 * queryIndex[e]: 查询 e 在原查询序列中的索引
 * qcnt: 查询计数器
 */
public static int[] headQuery = new int[MAXN];
public static int[] queryNext = new int[MAXN << 1];
public static int[] queryTo = new int[MAXN << 1];
public static int[] queryIndex = new int[MAXN << 1];
public static int qcnt;

/**
 * Tarjan 算法相关数组
 * visited[u]: 节点 u 是否已被访问
 * unionfind[u]: 节点 u 在并查集中的父节点
 * father[u]: 节点 u 在 DFS 过程中的父节点
 * ans[i]: 第 i 次查询的 LCA 结果
 */
public static boolean[] visited = new boolean[MAXN];
public static int[] unionfind = new int[MAXN];
public static int[] father = new int[MAXN];
public static int[] ans = new int[MAXN];

/**
 * 初始化算法所需的数据结构
 * 设置数组初始值，准备处理新的测试用例
 *
```

```
     * @param n 节点数量
     */
    public static void build(int n) {
        // 初始化糖果数组
        Arrays.fill(num, 1, n + 1, 0);
        // 初始化边和查询计数器
        tcnt = qcnt = 1;
        // 初始化链式前向星的头数组
        Arrays.fill(headEdge, 1, n + 1, 0);
        Arrays.fill(headQuery, 1, n + 1, 0);
        // 初始化访问标记数组
        Arrays.fill(visited, 1, n + 1, false);
        // 初始化并查集，每个节点初始时是自己的代表元素
        for (int i = 1; i <= n; i++) {
            unionfind[i] = i;
        }
    }

    /**
     * 向链式前向星中添加一条无向树边
     *
     * @param u 边的一个端点
     * @param v 边的另一个端点
     */
    public static void addEdge(int u, int v) {
        // 添加 u 到 v 的边
        edgeNext[tcnt] = headEdge[u];
        edgeTo[tcnt] = v;
        headEdge[u] = tcnt++;
    }

    /**
     * 向链式前向星中添加一条查询
     *
     * @param u 查询的起始节点
     * @param v 查询的目标节点
     * @param i 查询在原查询序列中的索引
     */
    public static void addQuery(int u, int v, int i) {
        // 添加 u 到 v 的查询
        queryNext[qcnt] = headQuery[u];
        queryTo[qcnt] = v;
        queryIndex[qcnt] = i;
```

```
    headQuery[u] = qcnt++;
}

/**
 * 用于并查集路径压缩的栈
 */
public static int[] stack = new int[MAXN];

/**
 * 并查集的查找操作（迭代版，带路径压缩）
 *
 * @param i 要查找的节点
 * @return 节点 i 所在集合的代表元素
 */
public static int find(int i) {
    // 栈大小
    int size = 0;
    // 找到根节点
    while (i != unionfind[i]) {
        // 将路径上的节点压入栈
        stack[size++] = i;
        i = unionfind[i];
    }
    // 路径压缩：将路径上的所有节点直接连接到根节点
    while (size > 0) {
        unionfind[stack[--size]] = i;
    }
    return i;
}

/**
 * 用于迭代 DFS 的栈元素结构
 * ufe[i][0]: 当前节点
 * ufe[i][1]: 当前节点的父节点
 * ufe[i][2]: 当前处理的边索引
 */
public static int[][] ufe = new int[MAXN][3];

/**
 * 迭代 DFS 的栈大小和当前处理的节点、父节点、边索引
 */
public static int stackSize, u, f, e;
```

```
/**
 * 将节点信息压入栈中
 *
 * @param u 当前节点
 * @param f 当前节点的父节点
 * @param e 当前处理的边索引
 */
public static void push(int u, int f, int e) {
    ufe[stackSize][0] = u;
    ufe[stackSize][1] = f;
    ufe[stackSize][2] = e;
    stackSize++;
}

/**
 * 从栈中弹出节点信息
 */
public static void pop() {
    --stackSize;
    u = ufe[stackSize][0];
    f = ufe[stackSize][1];
    e = ufe[stackSize][2];
}

/**
 * Tarjan 离线 LCA 算法（迭代版）
 *
 * @param root 根节点
 */
public static void tarjan(int root) {
    // 初始化栈
    stackSize = 0;
    // 将根节点压入栈，e=-1 表示第一次访问该节点
    push(root, 0, -1);

    // 当栈不为空时继续处理
    while (stackSize > 0) {
        // 弹出栈顶元素
        pop();

        // 如果是第一次访问该节点
        if (e == -1) {
            // 标记当前节点已被访问
```

```
            visited[u] = true;
            // 设置当前节点的第一个邻接边
            e = headEdge[u];
        } else {
            // 如果不是第一次访问该节点，处理下一条邻接边
            e = edgeNext[e];
        }

        // 如果还有未处理的邻接边
        if (e != 0) {
            // 将当前节点状态重新压入栈
            push(u, f, e);
            // 如果邻接边指向的节点不是父节点
            if (edgeTo[e] != f) {
                // 将邻接边指向的节点压入栈，e=-1 表示第一次访问
                push(edgeTo[e], u, -1);
            }
        } else {
            // 如果所有邻接边都已处理完，处理查询
            for (int q = headQuery[u], v; q != 0; q = queryNext[q]) {
                v = queryTo[q];
                // 如果查询的目标节点已被访问，则可以计算 LCA
                if (visited[v]) {
                    // 两个节点的 LCA 就是它们所在集合的代表元素
                    ans[queryIndex[q]] = find(v);
                }
            }
            // 将当前节点与父节点合并到同一集合
            unionfind[u] = f;
            // 记录当前节点的父节点
            father[u] = f;
        }
    }
}

/**
 * DFS 回溯计算每个节点的最终糖果数量（迭代版）
 *
 * @param root 根节点
 */
public static void dfs(int root) {
    // 初始化栈
    stackSize = 0;
```

```
        // 将根节点压入栈，e=-1 表示第一次访问该节点
        push(root, 0, -1);

        // 当栈不为空时继续处理
        while (stackSize > 0) {
            // 弹出栈顶元素
            pop();

            // 如果是第一次访问该节点
            if (e == -1) {
                // 设置当前节点的第一个邻接边
                e = headEdge[u];
            } else {
                // 如果不是第一次访问该节点，处理下一条邻接边
                e = edgeNext[e];
            }

            // 如果还有未处理的邻接边
            if (e != 0) {
                // 将当前节点状态重新压入栈
                push(u, f, e);
                // 如果邻接边指向的节点不是父节点
                if (edgeTo[e] != f) {
                    // 将邻接边指向的节点压入栈，e=-1 表示第一次访问
                    push(edgeTo[e], u, -1);
                }
            } else {
                // 如果所有邻接边都已处理完，累加子节点的糖果数量
                for (int e = headEdge[u], v; e != 0; e = edgeNext[e]) {
                    v = edgeTo[e];
                    if (v != f) {
                        num[u] += num[v];
                    }
                }
            }
        }
    }

    /**
     * 主函数，处理输入输出并调用相应的算法函数
     *
     * @param args 命令行参数
     * @throws IOException 输入输出异常
```

```
 */
public static void main(String[] args) throws IOException {
    // 使用高效的输入方式
    BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
    StreamTokenizer in = new StreamTokenizer(br);
    // 使用高效的输出方式
    PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));

    // 读取节点数
    in.nextToken();
    int n = (int) in.nval;

    // 初始化数据结构
    build(n);

    // 读取访问顺序
    for (int i = 1; i <= n; i++) {
        in.nextToken();
        travel[i] = (int) in.nval;
    }

    // 构建树结构
    for (int i = 1, u, v; i < n; i++) {
        in.nextToken();
        u = (int) in.nval;
        in.nextToken();
        v = (int) in.nval;
        addEdge(u, v);
        addEdge(v, u);  // 无向图，添加反向边
    }

    // 添加所有查询（相邻节点对的LCA查询）
    for (int i = 1; i < n; i++) {
        // 添加正向查询
        addQuery(travel[i], travel[i + 1], i);
        // 添加反向查询
        addQuery(travel[i + 1], travel[i], i);
    }

    // 执行计算
    compute(n);

    // 输出每个节点需要的糖果数量
```

```
    for (int i = 1; i <= n; i++) {
        out.println(num[i]);
    }

    // 确保输出被刷新
    out.flush();
    // 关闭资源
    out.close();
    br.close();
}

/**
 * 执行核心计算逻辑
 *
 * @param n 节点数量
 */
public static void compute(int n) {
    // 使用 Tarjan 算法计算所有查询的 LCA
    tarjan(1);

    // 对每一条路径执行树上点差分操作
    for (int i = 1, u, v, lca, lcafather; i < n; i++) {
        // 获取路径的两个端点
        u = travel[i];
        v = travel[i + 1];
        // 获取两个端点的 LCA
        lca = ans[i];
        // 获取 LCA 的父节点
        lcafather = father[lca];

        /**
         * 树上点差分核心操作
         * 对于路径(u,v)，它在原树上会形成一条路径
         * 通过点差分，我们标记路径上的所有节点
         * 1. 在 u 处+1
         * 2. 在 v 处+1
         * 3. 在 LCA 处-1
         * 4. 在 LCA 的父节点处-1
         */
        num[u]++;
        num[v]++;
        num[lca]--;
        num[lcafather]--;
```

```
        }

        // 通过 DFS 回溯计算每个节点的最终糖果数量
        dfs(1);

        // 调整除最后一个节点外的所有访问节点的糖果数量
        // 因为这些节点在访问时需要消耗一个糖果
        for (int i = 2; i <= n; i++) {
            num[travel[i]]--;
        }
    }

}

==========================================

文件: Code03_MinimizePriceOfTrips1.java
==========================================
package class122;

import java.util.Arrays;

/**
 * 最小化旅行的价格总和(倍增方法求 lca)
 *
 * 题目来源：LeetCode 2646. 最小化旅行的价格总和
 * 题目链接：https://leetcode.cn/problems/minimize-the-total-price-of-the-trips/
 *
 * 题目描述：
 * 有 n 个节点形成一棵树，每个节点上有点权，再给定很多路径。
 * 每条路径有开始点和结束点，路径代价就是从开始点到结束点的点权和。
 * 所有路径的代价总和就是旅行的价格总和。
 * 你可以选择把某些点的点权减少一半，来降低旅行的价格总和。
 * 但是要求选择的点不能相邻。
 * 返回旅行的价格总和最少能是多少。
 *
 * 算法原理：树上点差分 + 树形 DP
 * 这是一个结合了树上点差分和树形 DP 的综合问题。
 *
 * 解题思路：
 * 1. 首先使用树上点差分统计每个节点被多少条路径经过
 * 2. 然后使用树形 DP，在满足相邻节点不能同时选中的约束下，
 *    决策哪些节点减半以最小化总价格
```

```
 *
 * 时间复杂度分析：
 * - 建图：O(N)
 * - 预处理 LCA：O(N log N)
 * - 树上点差分标记：O(M log N)，其中 M 是路径数
 * - DFS 回溯统计：O(N)
 * - 树形 DP：O(N)
 * 总时间复杂度：O(N log N + M log N)
 *
 * 空间复杂度分析：
 * - 树的存储：O(N)
 * - LCA 倍增数组：O(N log N)
 * - 差分数组：O(N)
 * - DP 状态：O(1)
 * 总空间复杂度：O(N log N)
 *
 * 工程化考量：
 * 1. 使用链式前向星存储树结构，提高空间效率和遍历速度
 * 2. 使用倍增法计算 LCA，避免多次在线查询的开销
 * 3. 采用静态成员变量减少对象创建，在算法竞赛中常用此技巧
 * 4. 题目给定点的编号从 0 号点开始，代码中调整成从 1 号点开始便于处理
 *
 * 最优解分析：
 * 本题结合了树上点差分和树形 DP 两种技术，是解决此类问题的最优解。
 * 相比于暴力遍历每条路径的 O(N*M)复杂度，树上点差分可以将统计时间优化到 O(M log N)。
 * 树形 DP 在 O(N)时间内完成最优决策，整体效率很高。
 */
public class Code03_MinimizePriceOfTrips1 {

    /**
     * 主函数，计算最小化旅行价格的总和
     *
     * @param n 节点数
     * @param es 树的边（LeetCode 格式，节点编号从 0 开始）
     * @param ps 每个节点的价格（LeetCode 格式，节点编号从 0 开始）
     * @param ts 旅行路径（LeetCode 格式，节点编号从 0 开始）
     * @return 最小化旅行价格的总和
     */
    // 题目给定点的编号从 0 号点开始，代码中调整成从 1 号点开始
    public static int minimumTotalPrice(int n, int[][] es, int[] ps, int[][] ts) {
        // 初始化数据结构
        build(n);
```

```
// 转换价格数组的索引（从 0 开始转为从 1 开始）
for (int i = 0, j = 1; i < n; i++, j++) {
    price[j] = ps[i];
}

// 构建无向树（转换节点编号从 0 开始到从 1 开始）
for (int[] edge : es) {
    addEdge(edge[0] + 1, edge[1] + 1);
    addEdge(edge[1] + 1, edge[0] + 1);
}

// 预处理 LCA 所需的数据
dfs1(1, 0);

// 处理所有旅行路径，执行树上点差分
int u, v, lca, lcafather;
for (int[] trip : ts) {
    // 转换节点编号
    u = trip[0] + 1;
    v = trip[1] + 1;
    // 计算 LCA
    lca = lca(u, v);
    // 计算 LCA 的父节点
    lcafather = stjump[lca][0];

    /**
     * 树上点差分核心操作
     * 对于路径(u,v)，它在原树上会形成一条路径
     * 通过点差分，我们标记路径上的所有节点
     * 1. 在 u 处+1
     * 2. 在 v 处+1
     * 3. 在 LCA 处-1
     * 4. 在 LCA 的父节点处-1
     */
    num[u]++;
    num[v]++;
    num[lca]--;
    num[lcafather]--;
}

// 通过 DFS 回溯计算每个节点被经过的次数
dfs2(1, 0);
```

```
    // 执行树形DP，计算最小价格总和
    dp(1, 0);

    // 返回根节点减半或不减半的最小值
    return Math.min(no, yes);
}

/**
 * 最大节点数
 * 题目中N最大为50，设置为51以避免越界
 */
public static int MAXN = 51;

/**
 * 倍增数组的层数限制
 * 2^6 = 64 > 50，足够处理题目中的最大节点数
 */
public static int LIMIT = 6;

/**
 * 最大幂次，根据节点数量动态计算
 */
public static int power;

/**
 * 计算log2(n)的整数部分
 *
 * @param n 输入整数
 * @return log2(n)的整数部分
 */
public static int log2(int n) {
    int ans = 0;
    while ((1 << ans) <= (n >> 1)) {
        ans++;
    }
    return ans;
}

/**
 * 每个节点的价格
 * price[i]表示节点i的价格
 */
public static int[] price = new int[MAXN];
```

```
/**
 * 差分数组
 * num[i]表示节点 i 被路径覆盖的次数
 */
public static int[] num = new int[MAXN];

/**
 * 链式前向星存储树结构
 * head[u]: 节点 u 的第一个边的索引
 * next[e]: 边 e 的下一个边的索引
 * to[e]: 边 e 指向的节点
 * cnt: 当前可用的边索引
 */
public static int[] head = new int[MAXN];
public static int[] next = new int[MAXN << 1];
public static int[] to = new int[MAXN << 1];
public static int cnt;

/**
 * LCA 相关数组
 * deep[u]: 节点 u 的深度
 * stjump[u][p]: 节点 u 的 2^p 级祖先
 */
public static int[] deep = new int[MAXN];
public static int[][] stjump = new int[MAXN][LIMIT];

/**
 * 初始化算法所需的数据结构
 * 设置数组初始值，准备处理新的测试用例
 *
 * @param n 节点数量
 */
public static void build(int n) {
    power = log2(n);
    // 初始化差分数组，从 1 开始因为节点编号从 1 开始
    Arrays.fill(num, 1, n + 1, 0);
    // 边索引从 1 开始，0 表示没有边
    cnt = 1;
    // 初始化链式前向星的 head 数组
    Arrays.fill(head, 1, n + 1, 0);
}
```

```
/**
 * 向链式前向星中添加一条无向边
 *
 * @param u 边的一个端点
 * @param v 边的另一个端点
 */
public static void addEdge(int u, int v) {
    // 添加 u 到 v 的边
    next[cnt] = head[u];
    to[cnt] = v;
    head[u] = cnt++;
}

/**
 * 第一次 DFS，预处理每个节点的深度和倍增跳跃数组
 *
 * @param u 当前处理的节点
 * @param f 当前节点的父节点
 *
 * 时间复杂度：O(N log N)
 * 空间复杂度：O(log N) - 递归调用栈深度
 */
public static void dfs1(int u, int f) {
    // 设置当前节点的深度，父节点深度+1
    deep[u] = deep[f] + 1;
    // 设置当前节点的直接父节点（2^0 级祖先）
    stjump[u][0] = f;

    // 预处理倍增数组
    // 利用动态规划思想：u 的 2^p 级祖先 = u 的 2^(p-1)级祖先的 2^(p-1)级祖先
    for (int p = 1; p <= power; p++) {
        stjump[u][p] = stjump[stjump[u][p - 1]][p - 1];
    }

    // 递归处理所有子节点
    for (int e = head[u]; e != 0; e = next[e]) {
        if (to[e] != f) {
            dfs1(to[e], u);
        }
    }
}

/**
```

```
 * 使用倍增法计算两个节点的最近公共祖先
 *
 * @param a 第一个节点
 * @param b 第二个节点
 * @return a 和 b 的最近公共祖先
 *
 * 时间复杂度：O(log N)
 * 空间复杂度：O(1)
 */
public static int lca(int a, int b) {
    // 确保 a 的深度不小于 b
    if (deep[a] < deep[b]) {
        int tmp = a;
        a = b;
        b = tmp;
    }

    // 将 a 向上跳到与 b 同一深度
    for (int p = power; p >= 0; p--) {
        if (deep[stjump[a][p]] >= deep[b]) {
            a = stjump[a][p];
        }
    }

    // 如果此时 a==b，则找到了 LCA
    if (a == b) {
        return a;
    }

    // 继续向上跳，直到找到 LCA
    for (int p = power; p >= 0; p--) {
        if (stjump[a][p] != stjump[b][p]) {
            a = stjump[a][p];
            b = stjump[b][p];
        }
    }

    // 返回它们的父节点
    return stjump[a][0];
}

/**
 * 第二次 DFS，计算每个节点被覆盖的次数
```

```
 *
 * @param u 当前处理的节点
 * @param f 当前节点的父节点
 *
 * 时间复杂度：O(N)
 * 空间复杂度：O(log N) - 递归调用栈深度
 */
public static void dfs2(int u, int f) {
    // 递归处理所有子节点
    for (int e = head[u], v; e != 0; e = next[e]) {
        v = to[e];
        if (v != f) {
            dfs2(v, u);
        }
    }

    // 将子节点的覆盖次数累加到父节点
    for (int e = head[u], v; e != 0; e = next[e]) {
        v = to[e];
        if (v != f) {
            num[u] += num[v];
        }
    }
}

/**
 * 树形 DP 的状态变量
 * no: 当前节点不减半的最小价格
 * yes: 当前节点减半的最小价格
 */
public static int no, yes;

/**
 * 树形 DP，决策哪些节点减半以最小化总价格
 * 状态转移：
 * no = price[u] * num[u] + Σmin(no[v], yes[v])  // 当前节点不减半
 * yes = (price[u]/2) * num[u] + Σno[v]         // 当前节点减半
 *
 * 约束条件：相邻节点不能同时减半
 *
 * @param u 当前处理的节点
 * @param f 当前节点的父节点
 */
```

```
    public static void dp(int u, int f) {
        // 当前节点不减半的代价
        int n = price[u] * num[u];
        // 当前节点减半的代价
        int y = (price[u] / 2) * num[u];

        // 遍历当前节点的所有子节点
        for (int e = head[u], v; e != 0; e = next[e]) {
            v = to[e];
            if (v != f) {
                // 递归处理子节点
                dp(v, u);

                /**
                 * 状态转移方程：
                 * 1. 当前节点不减半，子节点可以减半或不减半，选择最小值
                 * 2. 当前节点减半，子节点不能减半（约束条件）
                 */
                // 当前节点不减半，子节点可以减半或不减半
                n += Math.min(no, yes);
                // 当前节点减半，子节点不能减半
                y += no;
            }
        }

        // 更新当前节点的状态
        no = n;
        yes = y;
    }

}
```

==========

文件: Code03_MinimizePriceOfTrips2.java

==========

```
package class122;

import java.util.Arrays;

/**
 * 最小化旅行的价格总和(tarjan 方法求 lca)
 *
```

* 题目来源：LeetCode 2646. 最小化旅行的价格总和
* 题目链接：https://leetcode.cn/problems/minimize-the-total-price-of-the-trips/
*
* 算法原理：
* 这是一道综合性的树上算法题，结合了以下几种关键技术：
* 1. 树上差分：用于统计每条路径经过的节点次数
* 2. Tarjan 离线 LCA 算法：用于快速计算树上任意两点间的最近公共祖先
* 3. 树形动态规划：用于在满足约束条件下（选择的点不能相邻）最小化总价格
*
* 解题思路：
* 1. 首先通过 Tarjan 离线算法计算所有查询路径的 LCA
* 2. 利用树上点差分技术统计每个节点在所有路径中被经过的次数
* 3. 通过一次 DFS 遍历累加子树信息，得到每个节点的确切访问次数
* 4. 最后使用树形 DP，在满足"选择的点不能相邻"的约束下，决定哪些节点的价格减半
*
* 树上点差分原理：
* 对于从 u 到 v 的路径，我们执行以下操作：
* num[u]++; num[v]++; num[lca]--; num[father[lca]]--;
* 然后通过 DFS 累加子树和，得到每个节点的真实访问次数
*
* 树形 DP 状态设计：
* no: 当前节点不选择减半时，以当前节点为根的子树的最小价格总和
* yes: 当前节点选择减半时，以当前节点为根的子树的最小价格总和
*
* 状态转移方程：
* no = price[u] * num[u] + Σmin(no[child], yes[child])
* yes = (price[u]/2) * num[u] + Σno[child]
*
* 时间复杂度分析：
* 1. Tarjan 离线 LCA：O(N + M)
* 2. 树上差分标记：O(M)
* 3. DFS 累加：O(N)
* 4. 树形 DP：O(N)
* 总体时间复杂度：O(N + M)
*
* 空间复杂度分析：
* 1. 图存储：O(N + M)
* 2. Tarjan 相关数组：O(N + M)
* 3. DP 状态：O(1)
* 总体空间复杂度：O(N + M)
*
* 工程化考量：
* 1. 由于题目中节点编号从 0 开始，而代码中调整为从 1 开始，需要注意索引转换

```
 * 2. 使用链式前向星存储树结构，提高遍历效率
 * 3. Tarjan 算法中使用并查集优化，保证查询效率
 * 4. 树形 DP 采用后序遍历方式，确保子节点状态已计算完成
 */
public class Code03_MinimizePriceOfTrips2 {

    /**
     * 主函数：计算最小化的旅行价格总和
     *
     * @param n 节点数量
     * @param es 树的边列表，每条边表示为 [from, to]
     * @param ps 节点价格数组，ps[i] 表示节点 i 的价格
     * @param ts 旅行计划列表，每个计划表示为 [start, end]
     * @return 最小化的旅行价格总和
     */
    // 题目给定点的编号从 0 号点开始，代码中调整成从 1 号点开始
    public static int minimumTotalPrice(int n, int[][] es, int[] ps, int[][] ts) {
        build(n);
        for (int i = 0, j = 1; i < n; i++, j++) {
            price[j] = ps[i];
        }
        for (int[] edge : es) {
            addEdge(edge[0] + 1, edge[1] + 1);
            addEdge(edge[1] + 1, edge[0] + 1);
        }
        int m = ts.length;
        for (int i = 0, j = 1; i < m; i++, j++) {
            addQuery(ts[i][0] + 1, ts[i][1] + 1, j);
            addQuery(ts[i][1] + 1, ts[i][0] + 1, j);
        }
        tarjan(1, 0);
        for (int i = 0, j = 1, u, v, lca, lcafather; i < m; i++, j++) {
            u = ts[i][0] + 1;
            v = ts[i][1] + 1;
            lca = ans[j];
            lcafather = father[lca];
            num[u]++;
            num[v]++;
            num[lca]--;
            num[lcafather]--;
        }
        dfs(1, 0);
        dp(1, 0);
```

```
    return Math.min(no, yes);
}

// 常量定义
public static int MAXN = 51;  // 最大节点数
public static int MAXM = 101; // 最大查询数

// 节点价格数组，price[i]表示节点 i 的价格
public static int[] price = new int[MAXN];

// 节点访问次数数组，num[i]表示节点 i 在所有路径中被经过的次数
public static int[] num = new int[MAXN];

// 链式前向星存储树的边
public static int[] headEdge = new int[MAXN];     // 边的头指针数组
public static int[] edgeNext = new int[MAXN << 1]; // 边的下一个指针数组
public static int[] edgeTo = new int[MAXN << 1];   // 边指向的节点数组

// 边计数器
public static int tcnt;

// 链式前向星存储查询
public static int[] headQuery = new int[MAXN];      // 查询的头指针数组
public static int[] queryNext = new int[MAXM << 1]; // 查询的下一个指针数组
public static int[] queryTo = new int[MAXM << 1];   // 查询指向的节点数组
public static int[] queryIndex = new int[MAXM << 1]; // 查询索引数组

// 查询计数器
public static int qcnt;

// 访问标记数组，用于 Tarjan 算法中标记节点是否已访问
public static boolean[] visited = new boolean[MAXN];

// 并查集数组，用于 Tarjan 算法中
public static int[] unionfind = new int[MAXN];

// 父节点数组，father[i]表示节点 i 的父节点
public static int[] father = new int[MAXN];

// 查询结果数组，ans[i]表示第 i 个查询的 LCA
public static int[] ans = new int[MAXM];

/**
```

```
     * 初始化函数：初始化所有数据结构
     *
     * @param n 节点数量
     */
    public static void build(int n) {
        Arrays.fill(num, 1, n + 1, 0);
        tcnt = qcnt = 1;
        Arrays.fill(headEdge, 1, n + 1, 0);
        Arrays.fill(headQuery, 1, n + 1, 0);
        Arrays.fill(visited, 1, n + 1, false);
        for (int i = 1; i <= n; i++) {
            unionfind[i] = i;
        }
    }

    /**
     * 添加边：向链式前向星中添加一条边
     *
     * @param u 起始节点
     * @param v 终止节点
     */
    public static void addEdge(int u, int v) {
        edgeNext[tcnt] = headEdge[u];
        edgeTo[tcnt] = v;
        headEdge[u] = tcnt++;
    }

    /**
     * 添加查询：向链式前向星中添加一个查询
     *
     * @param u 查询起始节点
     * @param v 查询终止节点
     * @param i 查询索引
     */
    public static void addQuery(int u, int v, int i) {
        queryNext[qcnt] = headQuery[u];
        queryTo[qcnt] = v;
        queryIndex[qcnt] = i;
        headQuery[u] = qcnt++;
    }

    /**
     * 并查集查找函数：带路径压缩的查找
```

```
     *
     * @param i 要查找的节点
     * @return 节点 i 所在集合的代表元素
     */
    public static int find(int i) {
        if (i != unionfind[i]) {
            unionfind[i] = find(unionfind[i]);
        }
        return unionfind[i];
    }

    /**
     * Tarjan 离线 LCA 算法：计算所有查询的最近公共祖先
     *
     * 算法原理：
     * 1. 使用 DFS 遍历树，当第一次访问某个节点时，标记为已访问
     * 2. 递归处理所有子节点
     * 3. 处理完所有子节点后，处理与当前节点相关的查询
     * 4. 如果查询的另一个节点已经被访问，则它们的 LCA 就是另一个节点的并查集代表元素
     * 5. 将当前节点合并到其父节点所在的集合中
     *
     * @param u 当前节点
     * @param f 父节点
     */
    public static void tarjan(int u, int f) {
        visited[u] = true;
        for (int e = headEdge[u], v; e != 0; e = edgeNext[e]) {
            v = edgeTo[e];
            if (v != f) {
                tarjan(v, u);
            }
        }
        for (int e = headQuery[u], v; e != 0; e = queryNext[e]) {
            v = queryTo[e];
            if (visited[v]) {
                ans[queryIndex[e]] = find(v);
            }
        }
        unionfind[u] = f;
        father[u] = f;
    }

    /**
```

```
     * DFS 累加函数：通过 DFS 遍历累加子树信息，得到每个节点的确切访问次数
     *
     * 算法原理：
     * 1. 后序遍历树结构，先处理所有子节点
     * 2. 将子节点的访问次数累加到当前节点
     *
     * @param u 当前节点
     * @param f 父节点
     */
    public static void dfs(int u, int f) {
        for (int e = headEdge[u], v; e != 0; e = edgeNext[e]) {
            v = edgeTo[e];
            if (v != f) {
                dfs(v, u);
            }
        }
        for (int e = headEdge[u], v; e != 0; e = edgeNext[e]) {
            v = edgeTo[e];
            if (v != f) {
                num[u] += num[v];
            }
        }
    }

    // 树形 DP 的状态变量
    public static int no, yes; // no 表示当前节点不选择减半的最小值，yes 表示当前节点选择减半的最小值

    /**
     * 树形 DP 函数：在满足"选择的点不能相邻"的约束下，计算最小价格总和
     *
     * 状态转移方程：
     * no = price[u] * num[u] + Σmin(no[child], yes[child])
     * yes = (price[u]/2) * num[u] + Σno[child]
     *
     * @param u 当前节点
     * @param f 父节点
     */
    public static void dp(int u, int f) {
        int n = price[u] * num[u];           // 当前节点不减半时的基础价格
        int y = (price[u] / 2) * num[u];     // 当前节点减半时的基础价格
        for (int e = headEdge[u], v; e != 0; e = edgeNext[e]) {
            v = edgeTo[e];
```

```
            if (v != f) {
                dp(v, u);
                n += Math.min(no, yes);      // 不减半时，子节点可以选择减半或不减半
                y += no;                     // 减半时，子节点必须不减半
            }
        }
        no = n;
        yes = y;
    }

}
```

==========

文件: Code04_Network.java

==========

```
package class122;

/**
 * 边差分实战 - 使图不连通的方法数
 *
 * 题目来源：POJ 3417 Network
 * 题目链接：http://poj.org/problem?id=3417
 *
 * 题目描述：
 * 有 n 个节点，给定 n-1 条老边使其连接成一棵树，再给定 m 条新边额外加在树上。
 * 你可以切断两条边让这个图不连通，切断的两条边必须是一条老边和一条新边。
 * 返回方法数。
 *
 * 算法原理：树上边差分
 * 树上边差分是处理树上路径操作的一种高效技术。
 * 对于每条新边(u,v)，它在原树上会形成一个环，环上的所有老边都被这条新边覆盖。
 * 通过边差分，我们可以：
 * 1. num[u]++
 * 2. num[v]++
 * 3. num[lca(u,v)] -= 2
 * 最后通过一次 DFS 回溯累加子节点的差分标记，得到每条老边被覆盖的次数。
 *
 * 解题思路：
 * 1. 对于每条老边，统计它被多少条新边覆盖
 * 2. 根据覆盖次数计算切断方案：
 *    - 覆盖 0 次：可以与任意一条新边配对，有 m 种方案
 *    - 覆盖 1 次：只能与对应的那条新边配对，有 1 种方案
```

```
 *     - 覆盖≥2 次：无法通过切断一条老边和一条新边使图不连通，有 0 种方案
 *
 * 时间复杂度分析：
 * - 预处理 LCA：O(N log N)
 * - 差分标记：O(M log N)，其中 M 是新边数量，每次需要计算 LCA
 * - DFS 回溯统计：O(N)
 * 总时间复杂度：O(N log N + M log N)
 *
 * 空间复杂度分析：
 * - 树的存储：O(N)
 * - LCA 倍增数组：O(N log N)
 * - 差分数组：O(N)
 * 总空间复杂度：O(N log N)
 *
 * 工程化考量：
 * 1. 使用链式前向星存储树结构，提高空间效率和遍历速度
 * 2. 使用 StreamTokenizer 进行高效输入，处理大量数据时性能优于 Scanner
 * 3. 使用 PrintWriter 进行高效输出，支持缓冲
 * 4. 采用静态成员变量减少对象创建，在算法竞赛中常用此技巧
 *
 * 最优解分析：
 * 树上边差分是解决此类问题的最优解，通过 O(1)的操作标记每条新边的影响范围，
 * 避免了暴力遍历每条环上的老边，时间复杂度比暴力方法的 O(M*N)有极大提升。
 */

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;
import java.io.PrintWriter;
import java.io.StreamTokenizer;
import java.util.Arrays;

public class Code04_Network {

    /**
     * 最大节点数量
     * 题目中节点数量范围：1 <= N <= 100000
     */
    public static int MAXN = 100001;

    /**
     * 倍增数组的层数限制
```

```
 * 2^17 = 131072 > 100000，足够处理题目中的最大节点数
 */
public static int LIMIT = 17;

/**
 * 最大幂次，根据节点数量动态计算
 */
public static int power;

/**
 * 计算 log2(n)的整数部分
 *
 * @param n 输入整数
 * @return log2(n)的整数部分
 */
public static int log2(int n) {
    int ans = 0;
    while ((1 << ans) <= (n >> 1)) {
        ans++;
    }
    return ans;
}

/**
 * 节点数量和新边数量
 */
public static int n, m;

/**
 * 差分数组
 * num[i]表示节点 i 到其父节点的边，被多少条新边覆盖
 * 注意：边被表示为子节点的属性，这样可以避免处理根节点的特殊情况
 */
public static int[] num = new int[MAXN];

/**
 * 链式前向星存储树结构
 * head[u]: 节点 u 的第一个边的索引
 * next[e]: 边 e 的下一个边的索引
 * to[e]: 边 e 指向的节点
 * cnt: 当前可用的边索引
 */
public static int[] head = new int[MAXN];
```

```
public static int[] next = new int[MAXN << 1];
public static int[] to = new int[MAXN << 1];
public static int cnt;

/**
 * LCA 相关数组
 * deep[u]: 节点 u 的深度
 * stjump[u][p]: 节点 u 的 2^p 级祖先
 */
public static int[] deep = new int[MAXN];
public static int[][] stjump = new int[MAXN][LIMIT];

/**
 * 最终答案
 */
public static int ans;

/**
 * 初始化算法所需的数据结构
 * 设置数组初始值，准备处理新的测试用例
 */
public static void build() {
    power = log2(n);
    // 初始化差分数组，从 1 开始因为节点编号从 1 开始
    Arrays.fill(num, 1, n + 1, 0);
    // 边索引从 1 开始，0 表示没有边
    cnt = 1;
    // 初始化链式前向星的 head 数组
    Arrays.fill(head, 1, n + 1, 0);
    // 初始化答案
    ans = 0;
}

/**
 * 向链式前向星中添加一条无向边
 *
 * @param u 边的一个端点
 * @param v 边的另一个端点
 */
public static void addEdge(int u, int v) {
    // 添加 u 到 v 的边
    next[cnt] = head[u];
    to[cnt] = v;
```

```
    head[u] = cnt++;
}

/**
 * 第一次DFS，预处理每个节点的深度和倍增跳跃数组
 *
 * @param u 当前处理的节点
 * @param f 当前节点的父节点
 *
 * 时间复杂度：O(N log N)
 * 空间复杂度：O(log N) - 递归调用栈深度
 */
public static void dfs1(int u, int f) {
    // 设置当前节点的深度，父节点深度+1
    deep[u] = deep[f] + 1;
    // 设置当前节点的直接父节点（2^0级祖先）
    stjump[u][0] = f;

    // 预处理倍增数组
    // 利用动态规划思想：u的2^p级祖先 = u的2^(p-1)级祖先的2^(p-1)级祖先
    for (int p = 1; p <= power; p++) {
        stjump[u][p] = stjump[stjump[u][p - 1]][p - 1];
    }

    // 递归处理所有子节点
    for (int e = head[u]; e != 0; e = next[e]) {
        if (to[e] != f) {
            dfs1(to[e], u);
        }
    }
}

/**
 * 使用倍增法计算两个节点的最近公共祖先
 *
 * @param a 第一个节点
 * @param b 第二个节点
 * @return a和b的最近公共祖先
 *
 * 时间复杂度：O(log N)
 * 空间复杂度：O(1)
 */
public static int lca(int a, int b) {
```

```
    // 确保 a 的深度不小于 b
    if (deep[a] < deep[b]) {
        int tmp = a;
        a = b;
        b = tmp;
    }

    // 将 a 向上跳到与 b 同一深度
    for (int p = power; p >= 0; p--) {
        if (deep[stjump[a][p]] >= deep[b]) {
            a = stjump[a][p];
        }
    }

    // 如果此时 a==b，则找到了 LCA
    if (a == b) {
        return a;
    }

    // 继续向上跳，直到找到 LCA
    for (int p = power; p >= 0; p--) {
        if (stjump[a][p] != stjump[b][p]) {
            a = stjump[a][p];
            b = stjump[b][p];
        }
    }

    // 返回它们的父节点
    return stjump[a][0];
}

/**
 * 第二次 DFS，计算每条边被覆盖的次数，并统计满足条件的方案数
 *
 * @param u 当前处理的节点
 * @param f 当前节点的父节点
 *
 * 时间复杂度：O(N)
 * 空间复杂度：O(log N) - 递归调用栈深度
 *
 * 算法逻辑：
 * 1. 先递归处理所有子节点
 * 2. 统计每个子节点到父节点这条边的覆盖次数
```

```
 * 3. 根据覆盖次数计算切断这条老边的可行方案数
 * 4. 累加子节点的覆盖次数到当前节点
 */
public static void dfs2(int u, int f) {
    // 递归处理所有子节点
    for (int e = head[u], v; e != 0; e = next[e]) {
        v = to[e];
        if (v != f) {
            dfs2(v, u);
        }
    }

    // 统计每条边的覆盖次数和方案数
    for (int e = head[u], v, w; e != 0; e = next[e]) {
        v = to[e];
        if (v != f) {
            // 获取边(u,v)的覆盖次数，存储在 v 的 num 数组中
            w = 0 + num[v];

            /**
             * 方案数统计逻辑：
             * - 覆盖次数为 0：这条老边不在任何新边形成的环中
             *   切断它后，无论切断哪条新边，图都会不连通
             *   共有 m 种方案（m 是新边的数量）
             *
             * - 覆盖次数为 1：这条老边恰好只在一个新边形成的环中
             *   只有切断对应的那条新边，图才会不连通
             *   共有 1 种方案
             *
             * - 覆盖次数>=2：这条老边在多个新边形成的环中
             *   切断它后，需要切断所有对应的新边才能使图不连通
             *   但题目要求只切断一条新边，因此没有可行方案
             *   共有 0 种方案
             */
            if (w == 0) {
                ans += m;  // 可以与任意一条新边配对
            } else if (w == 1) {
                ans += 1;  // 只能与对应的那条新边配对
            } else {
                ans += 0;  // 不能与任何新边配对
            }

            // 将子节点的覆盖次数累加到父节点，完成差分标记的传播
```

```
            num[u] += num[v];
        }
    }
}

/**
 * 主函数，处理输入输出并调用相应的算法函数
 *
 * @param args 命令行参数
 * @throws IOException 输入输出异常
 */
public static void main(String[] args) throws IOException {
    // 使用高效的输入方式
    BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
    StreamTokenizer in = new StreamTokenizer(br);
    // 使用高效的输出方式
    PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));

    // 读取节点数
    in.nextToken();
    n = (int) in.nval;
    // 初始化数据结构
    build();
    // 读取新边数
    in.nextToken();
    m = (int) in.nval;

    // 构建树结构，添加 n-1 条老边
    for (int i = 1, u, v; i < n; i++) {
        in.nextToken();
        u = (int) in.nval;
        in.nextToken();
        v = (int) in.nval;
        addEdge(u, v);
        addEdge(v, u);  // 无向图，添加反向边
    }

    // 预处理 LCA 所需的数据
    // 以节点 1 为根节点进行 DFS
    dfs1(1, 0);

    // 处理所有新边，执行树上边差分操作
    for (int i = 1, u, v, lca_node; i <= m; i++) {
```

```
            in.nextToken();
            u = (int) in.nval;
            in.nextToken();
            v = (int) in.nval;
            // 计算 u 和 v 的 LCA
            lca_node = lca(u, v);

            /**
             * 树上边差分核心操作
             * 对于新边(u,v)，它在原树上会形成一个环
             * 通过边差分，我们标记环上的所有老边
             * 1. 在 u 处+1
             * 2. 在 v 处+1
             * 3. 在 LCA 处-2，抵消多余的标记
             */
            num[u]++;
            num[v]++;
            num[lca_node] -= 2;
        }

        // 计算最终答案
        dfs2(1, 0);

        // 输出结果
        out.println(ans);
        // 确保输出被刷新
        out.flush();
        // 关闭资源
        out.close();
        br.close();
    }

}
```

==========

文件: Code05_TransportPlan1.java

==========

```
package class122;

/**
 * 运输计划，java 递归版
 *
```

* 题目来源：洛谷 P2680 [NOIP2015 提高组] 运输计划
* 题目链接：https://www.luogu.com.cn/problem/P2680
*
* 题目描述：
* 有 n 个节点，给定 n-1 条边使其连接成一棵树，每条边有正数边权。
* 给定很多运输计划，每个运输计划(a,b)表示从 a 去往 b。
* 每个运输计划的代价就是沿途边权和，运输计划之间完全互不干扰。
* 你只能选择一条边，将其边权变成 0。
* 你的目的是让所有运输计划代价的最大值尽量小。
* 返回所有运输计划代价的最大值最小能是多少。
*
* 算法原理：二分答案 + 树上边差分 + Tarjan 离线 LCA
* 这是一个典型的二分答案问题，结合了多种高级算法技术。
*
* 解题思路：
* 1. 二分答案：二分最大代价 limit，判断是否能让所有运输计划代价都<=limit
* 2. 对于给定的 limit，找出所有代价>limit 的运输计划
* 3. 这些运输计划必须都经过同一条边，才能通过将该边权置 0 来满足要求
* 4. 使用 Tarjan 算法离线计算所有运输计划的 LCA 和代价
* 5. 使用树上边差分统计每条边被超过 limit 的运输计划覆盖的次数
* 6. 找到被所有超过 limit 的运输计划都覆盖的边中权值最大的那条
*
* 时间复杂度分析：
* - 二分答案：O(log maxCost)
* - Tarjan 离线 LCA：O(N + M)
* - 树上边差分标记：O(M)
* - DFS 回溯统计：O(N)
* 总时间复杂度：O((N + M) * log maxCost)
*
* 空间复杂度分析：
* - 树的存储：O(N)
* - 查询存储：O(M)
* - LCA 倍增数组：O(N log N)
* - 差分数组：O(N)
* 总空间复杂度：O(N log N + M)
*
* 工程化考量：
* 1. 使用链式前向星存储树结构和查询，提高空间效率和遍历速度
* 2. 使用 Tarjan 离线算法计算 LCA，避免多次在线查询的开销
* 3. 使用 StreamTokenizer 进行高效输入，处理大量数据时性能优于 Scanner
* 4. 使用 PrintWriter 进行高效输出，支持缓冲
* 5. 采用静态成员变量减少对象创建，在算法竞赛中常用此技巧
*

```
 * 最优解分析：
 * 本题结合了二分答案、Tarjan 离线 LCA 和树上边差分三种技术，是解决此类问题的最优解。
 * 相比于暴力枚举每条边的方法，时间复杂度从 O(N*(N+M))优化到 O((N+M)*log maxCost)。
 */

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;
import java.io.PrintWriter;
import java.io.StreamTokenizer;
import java.util.Arrays;

public class Code05_TransportPlan1 {

    /**
     * 最大节点数和运输计划数
     * 题目中 N 和 M 最大为 3e5，设置为 300001 以避免越界
     */
    public static int MAXN = 300001;
    public static int MAXM = 300001;

    /**
     * 节点数和运输计划数
     */
    public static int n;
    public static int m;

    /**
     * 差分数组
     * num[i]表示节点 i 和其父节点的边，有多少代价>=limit 的运输计划用到
     */
    public static int[] num = new int[MAXN];

    /**
     * 链式前向星建图 - 树边存储
     * headEdge[u]: 节点 u 的第一条树边索引
     * edgeNext[e]: 边 e 的下一条边索引
     * edgeTo[e]: 边 e 指向的节点
     * edgeWeight[e]: 边 e 的权值
     * tcnt: 树边计数器
     */
    public static int[] headEdge = new int[MAXN];
```

```
public static int[] edgeNext = new int[MAXN << 1];
public static int[] edgeTo = new int[MAXN << 1];
public static int[] edgeWeight = new int[MAXN << 1];
public static int tcnt;

/**
 * 链式前向星建图 - 查询存储
 * headQuery[u]: 节点 u 的第一条查询索引
 * queryNext[e]: 查询 e 的下一条查询索引
 * queryTo[e]: 查询 e 的目标节点
 * queryIndex[e]: 查询 e 在原查询序列中的索引
 * qcnt: 查询计数器
 */
public static int[] headQuery = new int[MAXN];
public static int[] queryNext = new int[MAXM << 1];
public static int[] queryTo = new int[MAXM << 1];
public static int[] queryIndex = new int[MAXM << 1];
public static int qcnt;

/**
 * Tarjan 算法相关数组
 * visited[u]: 节点 u 是否已被访问
 * unionfind[u]: 节点 u 在并查集中的父节点
 */
public static boolean[] visited = new boolean[MAXN];
public static int[] unionfind = new int[MAXN];

/**
 * 运输计划的起止节点
 * quesu[i]: 第 i 号运输计划的起点
 * quesv[i]: 第 i 号运输计划的终点
 */
public static int[] quesu = new int[MAXM];
public static int[] quesv = new int[MAXM];

/**
 * 距离数组和 LCA 相关数组
 * distance[u]: 头节点到 u 号点的距离，tarjan 算法过程中更新
 * lca[i]: 第 i 号运输计划的两端点 lca，tarjan 算法过程中更新
 * cost[i]: 第 i 号运输计划代价是多少，tarjan 算法过程中更新
 */
public static int[] distance = new int[MAXN];
public static int[] lca = new int[MAXM];
```

```
public static int[] cost = new int[MAXM];

/**
 * 所有运输计划的最大代价，tarjan 算法过程中更新
 */
public static int maxCost;

/**
 * 初始化算法所需的数据结构
 * 设置数组初始值，准备处理新的测试用例
 */
public static void build() {
    // 初始化边和查询计数器
    tcnt = qcnt = 1;
    // 初始化链式前向星的头数组
    Arrays.fill(headEdge, 1, n + 1, 0);
    Arrays.fill(headQuery, 1, n + 1, 0);
    // 初始化访问标记数组
    Arrays.fill(visited, 1, n + 1, false);
    // 初始化并查集，每个节点初始时是自己的代表元素
    for (int i = 1; i <= n; i++) {
        unionfind[i] = i;
    }
    // 初始化最大代价
    maxCost = 0;
}

/**
 * 向链式前向星中添加一条无向树边
 *
 * @param u 边的一个端点
 * @param v 边的另一个端点
 * @param w 边的权值
 */
public static void addEdge(int u, int v, int w) {
    // 添加 u 到 v 的边
    edgeNext[tcnt] = headEdge[u];
    edgeTo[tcnt] = v;
    edgeWeight[tcnt] = w;
    headEdge[u] = tcnt++;
}

/**
```

```
     * 向链式前向星中添加一条查询
     *
     * @param u 查询的起始节点
     * @param v 查询的目标节点
     * @param i 查询在原查询序列中的索引
     */
    public static void addQuery(int u, int v, int i) {
        // 添加 u 到 v 的查询
        queryNext[qcnt] = headQuery[u];
        queryTo[qcnt] = v;
        queryIndex[qcnt] = i;
        headQuery[u] = qcnt++;
    }

    /**
     * 并查集的查找操作（带路径压缩）
     *
     * @param i 要查找的节点
     * @return 节点 i 所在集合的代表元素
     */
    public static int find(int i) {
        if (i != unionfind[i]) {
            // 路径压缩：将查找路径上的所有节点直接连接到根节点
            unionfind[i] = find(unionfind[i]);
        }
        return unionfind[i];
    }

    /**
     * Tarjan 离线 LCA 算法
     * 同时计算每个节点到根节点的距离、每个运输计划的 LCA 和代价
     *
     * @param u 当前处理的节点
     * @param f 当前节点的父节点
     * @param w 当前节点到父节点的边权
     */
    public static void tarjan(int u, int f, int w) {
        // 标记当前节点已被访问
        visited[u] = true;
        // 计算当前节点到根节点的距离
        distance[u] = distance[f] + w;

        // 递归处理所有子节点
```

```
    for (int e = headEdge[u], v; e != 0; e = edgeNext[e]) {
        v = edgeTo[e];
        if (v != f) {
            tarjan(v, u, edgeWeight[e]);
        }
    }

    // 处理所有与当前节点相关的查询
    for (int e = headQuery[u], v, i; e != 0; e = queryNext[e]) {
        v = queryTo[e];
        // 如果查询的目标节点已被访问，则可以计算 LCA
        if (visited[v]) {
            i = queryIndex[e];
            // 两个节点的 LCA 就是它们所在集合的代表元素
            lca[i] = find(v);
            // 计算运输计划的代价：distance[u] + distance[v] - 2 * distance[lca[i]]
            cost[i] = distance[u] + distance[v] - 2 * distance[lca[i]];
            // 更新最大代价
            maxCost = Math.max(maxCost, cost[i]);
        }
    }

    // 将当前节点与父节点合并到同一集合
    unionfind[u] = f;
}

/**
 * 判断是否能让所有运输计划代价都<=limit
 *
 * @param limit 限制的最大代价
 * @return 能否做到
 */
public static boolean f(int limit) {
    // 至少要减少的边权
    atLeast = maxCost - limit;
    // 初始化差分数组
    Arrays.fill(num, 1, n + 1, 0);
    // 超过要求的运输计划有几个
    beyond = 0;

    // 对所有代价>limit 的运输计划执行树上边差分
    for (int i = 1; i <= m; i++) {
        if (cost[i] > limit) {
```

```
            // 执行树上边差分操作
            num[quesu[i]]++;
            num[quesv[i]]++;
            num[lca[i]] -= 2;
            // 增加超过要求的运输计划数
            beyond++;
        }
    }

    // 如果没有超过要求的运输计划，直接返回 true
    // 否则通过 DFS 查找满足条件的边
    return beyond == 0 || dfs(1, 0, 0);
}

/**
 * 至少要减少多少边权
 */
public static int atLeast;

/**
 * 超过要求的运输计划有几个
 */
public static int beyond;

/**
 * DFS 查找满足条件的边
 *
 * @param u 当前处理的节点
 * @param f 当前节点的父节点
 * @param w 当前节点到父节点的边权
 * @return 是否找到满足条件的边
 */
public static boolean dfs(int u, int f, int w) {
    // 递归处理所有子节点
    for (int e = headEdge[u], v; e != 0; e = edgeNext[e]) {
        v = edgeTo[e];
        if (v != f) {
            if (dfs(v, u, edgeWeight[e])) {
                return true;
            }
        }
    }
```

```
    // 将子节点的差分标记累加到当前节点
    for (int e = headEdge[u], v; e != 0; e = edgeNext[e]) {
        v = edgeTo[e];
        if (v != f) {
            num[u] += num[v];
        }
    }

    // 判断当前边是否满足条件：
    // 1. 被所有超过 limit 的运输计划都覆盖（num[u] == beyond）
    // 2. 边权足够大（w >= atLeast）
    return num[u] == beyond && w >= atLeast;
}

/**
 * 计算运输计划代价的最大值最小能是多少
 *
 * @return 最小的最大代价
 */
public static int compute() {
    // 使用 Tarjan 算法计算所有运输计划的 LCA 和代价
    tarjan(1, 0, 0);

    // 二分答案
    int l = 0, r = maxCost, mid;
    int ans = 0;
    while (l <= r) {
        mid = (l + r) / 2;
        // 判断是否能让所有运输计划代价都<=mid
        if (f(mid)) {
            // 可以做到，尝试更小的答案
            ans = mid;
            r = mid - 1;
        } else {
            // 做不到，需要更大的答案
            l = mid + 1;
        }
    }
    return ans;
}

/**
 * 主函数，处理输入输出并调用相应的算法函数
```

```
 *
 * @param args 命令行参数
 * @throws IOException 输入输出异常
 */
public static void main(String[] args) throws IOException {
    // 使用高效的输入方式
    BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
    StreamTokenizer in = new StreamTokenizer(br);
    // 使用高效的输出方式
    PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));

    // 读取节点数
    in.nextToken();
    n = (int) in.nval;
    // 初始化数据结构
    build();
    // 读取运输计划数
    in.nextToken();
    m = (int) in.nval;

    // 构建树结构
    for (int i = 1, u, v, w; i < n; i++) {
        in.nextToken();
        u = (int) in.nval;
        in.nextToken();
        v = (int) in.nval;
        in.nextToken();
        w = (int) in.nval;
        // 添加无向树边
        addEdge(u, v, w);
        addEdge(v, u, w);
    }

    // 添加所有运输计划查询
    for (int i = 1, u, v; i <= m; i++) {
        in.nextToken();
        u = (int) in.nval;
        in.nextToken();
        v = (int) in.nval;
        // 记录运输计划的起止节点
        quesu[i] = u;
        quesv[i] = v;
        // 添加正向和反向查询
```

```
            addQuery(u, v, i);
            addQuery(v, u, i);
        }

        // 计算并输出结果
        out.println(compute());

        // 确保输出被刷新
        out.flush();
        // 关闭资源
        out.close();
        br.close();
    }

}
```

==========

文件: Code05_TransportPlan2.java

==========

```
package class122;

/**
 * 运输计划，java 迭代版
 *
 * 题目来源：洛谷 P2680 运输计划
 * 题目链接：https://www.luogu.com.cn/problem/P2680
 *
 * 算法原理：
 * 这是一道经典的树上问题，结合了以下几种关键技术：
 * 1. 二分答案：答案具有单调性，可以通过二分法寻找最优解
 * 2. 树上边差分：用于统计每条边被超过限制的路径覆盖的次数
 * 3. Tarjan 离线 LCA 算法：用于快速计算树上任意两点间的最近公共祖先
 * 4. DFS 遍历：用于检查是否存在一条边满足删除条件
 *
 * 解题思路：
 * 1. 二分答案：二分最大运输代价，检查能否通过删除一条边达到该代价
 * 2. 对于每个二分的值 limit，找出所有超过 limit 的运输计划
 * 3. 使用树上边差分技术统计每条边被这些超限路径覆盖的次数
 * 4. 通过 DFS 遍历检查是否存在一条边，满足：
 *    a. 被所有超限路径覆盖（即覆盖次数等于超限路径数）
 *    b. 边权不小于需要减少的值（maxCost - limit）
 * 5. 如果存在这样的边，则当前 limit 可行，尝试更小的值；否则尝试更大的值
```

```
 *
 * 树上边差分原理：
 * 对于从 u 到 v 的路径，我们执行以下操作：
 * num[u]++; num[v]++; num[lca] -= 2;
 * 然后通过 DFS 累加子树和，得到每条边的真实覆盖次数
 * 注意：这里统计的是点的标记，实际应用中需要转换为边的覆盖次数
 *
 * 时间复杂度分析：
 * 1. 二分答案：O(log(maxCost))
 * 2. Tarjan 离线 LCA：O(N + M)
 * 3. 树上边差分标记：O(M)
 * 4. DFS 检查：O(N)
 * 总体时间复杂度：O((N + M) * log(maxCost))
 *
 * 空间复杂度分析：
 * 1. 图存储：O(N + M)
 * 2. Tarjan 相关数组：O(N + M)
 * 3. 差分数组：O(N)
 * 总体空间复杂度：O(N + M)
 *
 * 工程化考量：
 * 1. 由于 Java 的运行效率相对较低，有时会有一个测试用例超时
 * 2. 使用迭代版实现避免递归深度过大导致的栈溢出
 * 3. 使用快速 IO 提高输入效率
 * 4. 使用链式前向星存储树结构，提高遍历效率
 * 5. Tarjan 算法中使用并查集优化，保证查询效率
 */

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

public class Code05_TransportPlan2 {

    // 常量定义
    public static int MAXN = 300001; // 最大节点数
    public static int MAXM = 300001; // 最大查询数

    // 全局变量
    public static int n; // 节点数
    public static int m; // 运输计划数
```

```
// 节点访问次数数组，用于树上边差分
public static int[] num = new int[MAXN];

// 链式前向星存储树的边
public static int[] headEdge = new int[MAXN];     // 边的头指针数组
public static int[] edgeNext = new int[MAXN << 1]; // 边的下一个指针数组
public static int[] edgeTo = new int[MAXN << 1];   // 边指向的节点数组
public static int[] edgeWeight = new int[MAXN << 1]; // 边权数组

// 边计数器
public static int tcnt;

// 链式前向星存储查询
public static int[] headQuery = new int[MAXN];      // 查询的头指针数组
public static int[] queryNext = new int[MAXM << 1]; // 查询的下一个指针数组
public static int[] queryTo = new int[MAXM << 1];   // 查询指向的节点数组
public static int[] queryIndex = new int[MAXM << 1]; // 查询索引数组

// 查询计数器
public static int qcnt;

// 访问标记数组，用于 Tarjan 算法中标记节点是否已访问
public static boolean[] visited = new boolean[MAXN];

// 并查集数组，用于 Tarjan 算法中
public static int[] unionfind = new int[MAXN];

// 运输计划起点和终点数组
public static int[] quesu = new int[MAXM]; // 运输计划起点
public static int[] quesv = new int[MAXM]; // 运输计划终点

// 距离数组，distance[i]表示节点 i 到根节点的距离
public static int[] distance = new int[MAXN];

// LCA 数组，lca[i]表示第 i 个运输计划的最近公共祖先
public static int[] lca = new int[MAXM];

// 运输计划代价数组，cost[i]表示第 i 个运输计划的代价
public static int[] cost = new int[MAXM];

// 最大运输代价
public static int maxCost;
```

```
/**
 * 初始化函数：初始化所有数据结构
 */
public static void build() {
    tcnt = qcnt = 1;
    Arrays.fill(headEdge, 1, n + 1, 0);
    Arrays.fill(headQuery, 1, n + 1, 0);
    Arrays.fill(visited, 1, n + 1, false);
    for (int i = 1; i <= n; i++) {
        unionfind[i] = i;
    }
    maxCost = 0;
}

/**
 * 添加边：向链式前向星中添加一条边
 *
 * @param u 起始节点
 * @param v 终止节点
 * @param w 边权
 */
public static void addEdge(int u, int v, int w) {
    edgeNext[tcnt] = headEdge[u];
    edgeTo[tcnt] = v;
    edgeWeight[tcnt] = w;
    headEdge[u] = tcnt++;
}

/**
 * 添加查询：向链式前向星中添加一个查询
 *
 * @param u 查询起始节点
 * @param v 查询终止节点
 * @param i 查询索引
 */
public static void addQuery(int u, int v, int i) {
    queryNext[qcnt] = headQuery[u];
    queryTo[qcnt] = v;
    queryIndex[qcnt] = i;
    headQuery[u] = qcnt++;
}
```

```
// find 方法的递归版改迭代版
public static int[] stack = new int[MAXN]; // 用于迭代版 find 的栈

/**
 * 并查集查找函数：迭代版实现，带路径压缩
 *
 * @param i 要查找的节点
 * @return 节点 i 所在集合的代表元素
 */
public static int find(int i) {
    int size = 0;
    while (i != unionfind[i]) {
        stack[size++] = i;
        i = unionfind[i];
    }
    while (size > 0) {
        unionfind[stack[--size]] = i;
    }
    return i;
}

// tarjan 方法的递归版改迭代版
// 不会改，看讲解 118，讲了怎么从递归版改成迭代版
public static int[][] ufwe = new int[MAXN][4]; // 用于迭代版 tarjan 的栈元素

public static int stackSize, u, f, w, e; // 迭代版 tarjan 的栈相关变量

/**
 * 向栈中压入元素
 *
 * @param u 当前节点
 * @param f 父节点
 * @param w 边权
 * @param e 边索引
 */
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++;
}
```

```
/**
 * 从栈中弹出元素
 */
public static void pop() {
    --stackSize;
    u = ufwe[stackSize][0];
    f = ufwe[stackSize][1];
    w = ufwe[stackSize][2];
    e = ufwe[stackSize][3];
}

/**
 * Tarjan 离线 LCA 算法：迭代版实现
 *
 * 算法原理：
 * 1. 使用显式栈模拟递归过程
 * 2. 当 e == -1 时，表示第一次访问节点 u，进行初始化操作
 * 3. 当 e != 0 时，表示正在遍历 u 的邻接边，处理下一条边
 * 4. 当 e == 0 时，表示已处理完 u 的所有邻接边，进行收尾操作
 *
 * @param root 根节点
 */
public static void tarjan(int root) {
    stackSize = 0;
    push(root, 0, 0, -1);
    while (stackSize > 0) {
        pop();
        if (e == -1) {
            visited[u] = true;
            distance[u] = distance[f] + w;
            e = headEdge[u];
        } else {
            e = edgeNext[e];
        }
        if (e != 0) {
            push(u, f, w, e);
            if (edgeTo[e] != f) {
                push(edgeTo[e], u, edgeWeight[e], -1);
            }
        } else {
            for (int q = headQuery[u], v, i; q != 0; q = queryNext[q]) {
                v = queryTo[q];
                if (visited[v]) {
```

```
                    i = queryIndex[q];
                    lca[i] = find(v);
                    cost[i] = distance[u] + distance[v] - 2 * distance[lca[i]];
                    maxCost = Math.max(maxCost, cost[i]);
                }
            }
            unionfind[u] = f;
        }
    }
}

/**
 * 检查函数：检查给定的 limit 是否可行
 *
 * @param limit 当前二分的运输代价上限
 * @return 如果可行返回 true，否则返回 false
 */
public static boolean f(int limit) {
    atLeast = maxCost - limit;
    Arrays.fill(num, 1, n + 1, 0);
    beyond = 0;
    for (int i = 1; i <= m; i++) {
        if (cost[i] > limit) {
            num[quesu[i]]++;
            num[quesv[i]]++;
            num[lca[i]] -= 2;
            beyond++;
        }
    }
    return beyond == 0 || dfs(1);
}

// 至少要减少多少边权
public static int atLeast;

// 超过要求的运输计划有几个
public static int beyond;

/**
 * DFS 遍历函数：迭代版实现，检查是否存在满足条件的边
 *
 * 算法原理：
 * 1. 使用显式栈模拟递归过程
```

```
 * 2. 遍历过程中累加子树信息
 * 3. 检查当前边是否满足条件：
 *    a. 被所有超限路径覆盖（num[u] == beyond）
 *    b. 边权不小于需要减少的值（w >= atLeast）
 *
 * @param root 根节点
 * @return 如果存在满足条件的边返回 true，否则返回 false
 */
// dfs 方法的递归版改迭代版
// 不会改，看讲解 118，讲了怎么从递归版改成迭代版
public static boolean dfs(int root) {
    stackSize = 0;
    push(root, 0, 0, -1);
    while (stackSize > 0) {
        pop();
        if (e == -1) {
            e = headEdge[u];
        } else {
            e = edgeNext[e];
        }
        if (e != 0) {
            push(u, f, w, e);
            if (edgeTo[e] != f) {
                push(edgeTo[e], u, edgeWeight[e], -1);
            }
        } else {
            for (int e = headEdge[u], v; e != 0; e = edgeNext[e]) {
                v = edgeTo[e];
                if (v != f) {
                    num[u] += num[v];
                }
            }
            if (num[u] == beyond && w >= atLeast) {
                return true;
            }
        }
    }
    return false;
}

/**
 * 计算函数：通过二分答案计算最小运输代价
 *
```

```
     * @return 最小运输代价
     */
    public static int compute() {
        tarjan(1);
        int l = 0, r = maxCost, mid;
        int ans = 0;
        while (l <= r) {
            mid = (l + r) / 2;
            if (f(mid)) {
                ans = mid;
                r = mid - 1;
            } else {
                l = mid + 1;
            }
        }
        return ans;
    }

    /**
     * 主函数：程序入口
     */
    public static void main(String[] args) throws IOException {
        FastIO in = new FastIO();
        PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out), false);
        n = in.nextInt();
        build();
        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);
            addEdge(v, u, w);
        }
        for (int i = 1, u, v; i <= m; i++) {
            u = in.nextInt();
            v = in.nextInt();
            quesu[i] = u;
            quesv[i] = v;
            addQuery(u, v, i);
            addQuery(v, u, i);
        }
        out.println(compute());
```

```
    out.flush();
    out.close();
}

/**
 * IO 工具类：用于快速输入输出
 */
static class FastIO {
    private final int SIZE = 1 << 20; // 缓冲区大小
    private byte[] buf;               // 输入缓冲区
    private int pos;                  // 当前读取位置
    private int count;                // 缓冲区中有效字节数
    private InputStream is;           // 输入流

    /**
     * 构造函数：初始化 IO 工具类
     */
    public FastIO() {
        buf = new byte[SIZE];
        pos = 0;
        count = 0;
        is = System.in;
    }

    /**
     * 读取一个字节
     *
     * @return 读取的字节，如果到达文件末尾返回-1
     * @throws IOException IO 异常
     */
    private int readByte() throws IOException {
        if (count == -1) {
            return -1;
        }
        if (pos >= count) {
            pos = 0;
            count = is.read(buf);
            if (count == -1) {
                return -1;
            }
        }
        return buf[pos++] & 0xff;
    }
```

```
        /**
         * 读取一个整数
         *
         * @return 读取的整数，如果到达文件末尾返回-1
         * @throws IOException IO异常
         */
        public int nextInt() throws IOException {
            int c, value = 0;
            boolean neg = false;
            do {
                c = readByte();
                if (c == -1) {
                    return -1;
                }
            } while (c <= ' ');
            if (c == '-') {
                neg = true;
                c = readByte();
            }
            for (; c >= '0' && c <= '9'; c = readByte()) {
                value = value * 10 + (c - '0');
            }
            return neg ? -value : value;
        }
    }

}
```

==========

文件: Code05_TransportPlan3.java

==========

```
package class122;

/**
 * 运输计划，C++版，递归不用改迭代
 *
 * 题目来源：洛谷 P2680 运输计划
 * 题目链接：https://www.luogu.com.cn/problem/P2680
 *
 * 算法原理：
 * 这是一道经典的树上问题，结合了以下几种关键技术：
```

* 1. 二分答案：答案具有单调性，可以通过二分法寻找最优解
* 2. 树上边差分：用于统计每条边被超过限制的路径覆盖的次数
* 3. Tarjan 离线 LCA 算法：用于快速计算树上任意两点间的最近公共祖先
* 4. DFS 遍历：用于检查是否存在一条边满足删除条件
*
* 解题思路：
* 1. 二分答案：二分最大运输代价，检查能否通过删除一条边达到该代价
* 2. 对于每个二分的值 limit，找出所有超过 limit 的运输计划
* 3. 使用树上边差分技术统计每条边被这些超限路径覆盖的次数
* 4. 通过 DFS 遍历检查是否存在一条边，满足：
*   a. 被所有超限路径覆盖（即覆盖次数等于超限路径数）
*   b. 边权不小于需要减少的值（maxCost - limit）
* 5. 如果存在这样的边，则当前 limit 可行，尝试更小的值；否则尝试更大的值
*
* 树上边差分原理：
* 对于从 u 到 v 的路径，我们执行以下操作：
* num[u]++; num[v]++; num[lca] -= 2;
* 然后通过 DFS 累加子树和，得到每条边的真实覆盖次数
* 注意：这里统计的是点的标记，实际应用中需要转换为边的覆盖次数
*
* 时间复杂度分析：
* 1. 二分答案：O(log(maxCost))
* 2. Tarjan 离线 LCA：O(N + M)
* 3. 树上边差分标记：O(M)
* 4. DFS 检查：O(N)
* 总体时间复杂度：O((N + M) * log(maxCost))
*
* 空间复杂度分析：
* 1. 图存储：O(N + M)
* 2. Tarjan 相关数组：O(N + M)
* 3. 差分数组：O(N)
* 总体空间复杂度：O(N + M)
*
* 工程化考量：
* 1. 这是 C++版本的实现，与 Java 版本逻辑完全一样
* 2. C++版本可以顺利通过所有测试用例，因为运行效率更高
* 3. 使用递归实现，代码更简洁易懂
* 4. 使用链式前向星存储树结构，提高遍历效率
* 5. Tarjan 算法中使用并查集优化，保证查询效率
* 6. 使用快速 IO 提高输入效率
*/

// 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 MAXM = 300001;
//int n, m;
//int num[MAXN];
//int headEdge[MAXN];
//int edgeNext[MAXN << 1];
//int edgeTo[MAXN << 1];
//int edgeWeight[MAXN << 1];
//int tcnt;
//int headQuery[MAXN];
//int queryNext[MAXM << 1];
//int queryTo[MAXM << 1];
//int queryIndex[MAXM << 1];
//int qcnt;
//bool visited[MAXN];
//int unionfind[MAXN];
//int quesu[MAXM];
//int quesv[MAXM];
//int dist[MAXN];
//int lca[MAXM];
//int cost[MAXM];
//int maxCost;
//int atLeast;
//int beyond;
//
//void build() {
//    tcnt = 1;
//    qcnt = 1;
//    for(int i = 1; i <= n; i++) {
//        headEdge[i] = 0;
//        headQuery[i] = 0;
//        visited[i] = false;
//        unionfind[i] = i;
//    }
//    maxCost = 0;
//}
//
```

```
//void addEdge(int u, int v, int w) {
//    edgeNext[tcnt] = headEdge[u];
//    edgeTo[tcnt] = v;
//    edgeWeight[tcnt] = w;
//    headEdge[u] = tcnt++;
//}
//
//void addQuery(int u, int v, int i) {
//    queryNext[qcnt] = headQuery[u];
//    queryTo[qcnt] = v;
//    queryIndex[qcnt] = i;
//    headQuery[u] = qcnt++;
//}
//
//int find(int i) {
//    if(i != unionfind[i]) {
//        unionfind[i] = find(unionfind[i]);
//    }
//    return unionfind[i];
//}
//
//void tarjan(int u, int f, int w) {
//    visited[u] = true;
//    dist[u] = dist[f] + w;
//    for(int e = headEdge[u]; e != 0; e = edgeNext[e]) {
//        int v = edgeTo[e];
//        if(v != f) {
//            tarjan(v, u, edgeWeight[e]);
//        }
//    }
//    for(int e = headQuery[u]; e != 0; e = queryNext[e]) {
//        int v = queryTo[e];
//        if(visited[v]) {
//            int i = queryIndex[e];
//            lca[i] = find(v);
//            cost[i] = dist[u] + dist[v] - 2 * dist[lca[i]];
//            maxCost = max(maxCost, cost[i]);
//        }
//    }
//    unionfind[u] = f;
//}
//
//bool dfs(int u, int f, int w) {
```

```
//    for(int e = headEdge[u]; e != 0; e = edgeNext[e]) {
//        int v = edgeTo[e];
//        if(v != f) {
//            if(dfs(v, u, edgeWeight[e])) {
//                return true;
//            }
//        }
//    }
//    for(int e = headEdge[u]; e != 0; e = edgeNext[e]) {
//        int v = edgeTo[e];
//        if(v != f) {
//            num[u] += num[v];
//        }
//    }
//    return (num[u] == beyond && w >= atLeast);
//}
//
//bool f(int limit) {
//    atLeast = maxCost - limit;
//    for(int i = 1; i <= n; i++) {
//        num[i] = 0;
//    }
//    beyond = 0;
//    for(int i = 1; i <= m; i++) {
//        if(cost[i] > limit) {
//            num[quesu[i]]++;
//            num[quesv[i]]++;
//            num[lca[i]] -= 2;
//            beyond++;
//        }
//    }
//    if(beyond == 0) return true;
//    return dfs(1, 0, 0);
//}
//
//int compute() {
//    tarjan(1, 0, 0);
//    int l = 0;
//    int r = maxCost;
//    int ans = 0;
//    while(l <= r) {
//        int mid = (l + r) / 2;
//        if(f(mid)) {
```

```
//            ans = mid;
//            r = mid - 1;
//        } else {
//            l = mid + 1;
//        }
//    }
//    return ans;
//}
//
//int main(){
//    ios::sync_with_stdio(false);
//    cin.tie(nullptr);
//    cin >> n;
//    build();
//    cin >> m;
//    for(int i = 1; i < n; i++){
//        int u,v,w;
//        cin >> u >> v >> w;
//        addEdge(u,v,w);
//        addEdge(v,u,w);
//    }
//    for(int i = 1; i <= m; i++){
//        int u,v;
//        cin >> u >> v;
//        quesu[i] = u;
//        quesv[i] = v;
//        addQuery(u,v,i);
//        addQuery(v,u,i);
//    }
//    cout << compute() << "\n";
//    return 0;
//}
```

==========

文件: Code06_DarkLock1.cpp

==========

```
/*
 * 暗的连锁 (LOJ 10131)
 * 算法：树上边差分 + LCA (最近公共祖先) + DFS 回溯
 *
 * 【题目来源】
 * LibreOJ
```

```
 * 题目链接：https://loj.ac/problem/10131
 *
 * 【题目描述】
 * 给定一棵包含 N 个节点的树（N-1 条边），以及 M 条额外的边（非树边）
 * 每条非树边连接两个节点，与树边一起构成一个连通图
 * 求有多少种方案，通过切断一条树边和一条非树边，使得图变得不连通
 *
 * 【算法原理】
 * 1. 每条非树边(u,v)会在树上形成一个环
 * 2. 使用边差分标记所有被非树边覆盖的树边
 * 3. 根据覆盖次数计算切断方案：
 *    - 覆盖 0 次：可以与任意一条非树边配对，有 m 种方案
 *    - 覆盖 1 次：只能与对应的非树边配对，有 1 种方案
 *    - 覆盖≥2 次：无法通过切断一条树边和一条非树边使图不连通，有 0 种方案
 *
 * 【复杂度分析】
 * - 时间复杂度：O(N log N + M log N)
 *   预处理 LCA: O(N log N)
 *   处理每条非树边: O(M log N)
 *   计算边覆盖次数: O(N)
 * - 空间复杂度：O(N log N)
 *   链式前向星存储树: O(N)
 *   倍增数组 stjump: O(N log N)
 *
 * 【工程化考量】
 * 1. 链式前向星作为高效的图存储结构，适合树结构的实现
 * 2. 变量命名：使用 next_而非 next，避免与 C++标准库中的 next 冲突
 * 3. 输入输出效率：使用 scanf/printf 保证大数据量下的性能
 * 4. 边界处理：根节点(1)的父节点设置为 0，避免数组越界
 *
 * 【最优解分析】
 * 此解法是本题的最优解，时间复杂度为 O(N log N + M log N)，无法进一步优化
 * 其他可能的算法如暴力枚举所有树边和非树边需要 O(NM)时间，远不如本解法高效
 */

#include <cstdio>
#include <algorithm>
#include <cstring>
using namespace std;

const int MAXN = 100001;  // 最大节点数
const int LIMIT = 17;     // 倍增数组的大小限制，2^17 足够处理 1e5 规模的树
```

```
// 全局变量定义
int power;                // 最大幂次，2^power > n
int n, m;                 // 节点数和非树边数
int num[MAXN];            // num[i]表示节点 i 到父节点的边被非树边覆盖的次数

// 链式前向星存储树结构
int head[MAXN];           // 邻接表头节点数组
int next_[MAXN << 1];     // 下一条边的索引（使用 next_避免与 C++标准库冲突）
int to[MAXN << 1];        // 边的目标节点
int cnt;                  // 边计数器

// LCA 相关数组
int deep[MAXN];           // 节点深度数组
int stjump[MAXN][LIMIT];  // 倍增数组，stjump[u][p]表示 u 的 2^p 级祖先
long long ans;            // 最终答案，使用 long long 避免溢出

/**
 * 计算 log2(n)的整数值，表示最大需要的幂次
 * @param n 节点数量
 * @return 最大幂次，满足 2^ans <= n/2
 */
int log2(int n) {
    int ans = 0;
    while ((1 << ans) <= (n >> 1)) {
        ans++;
    }
    return ans;
}

/**
 * 初始化数据结构
 */
void build() {
    power = log2(n);
    memset(num, 0, sizeof(num));  // 初始化差分数组
    cnt = 1;                      // 边计数器从 1 开始，方便链式前向星操作
    memset(head, 0, sizeof(head)); // 初始化邻接表头数组
    ans = 0;                      // 初始化答案
}

/**
 * 添加边到树中（无向边，需要添加双向）
 * @param u 边的起点
```

```
 * @param v 边的终点
 */
void addEdge(int u, int v) {
    next_[cnt] = head[u];  // 当前边的 next 指向当前 head[u]
    to[cnt] = v;           // 当前边的目标节点是 v
    head[u] = cnt++;       // 更新 head[u]为当前边的索引，并递增计数器
}

/**
 * 第一次 DFS：预处理深度数组和倍增数组
 * @param u 当前节点
 * @param f 父节点
 */
void dfs1(int u, int f) {
    deep[u] = deep[f] + 1;              // 设置当前节点深度
    stjump[u][0] = f;                   // 设置直接父节点

    // 预处理倍增数组
    for (int p = 1; p <= power; p++) {
        stjump[u][p] = stjump[stjump[u][p - 1]][p - 1];
    }

    // 递归处理所有子节点
    for (int e = head[u]; e != 0; e = next_[e]) {
        if (to[e] != f) {  // 避免回到父节点
            dfs1(to[e], u);
        }
    }
}

/**
 * 使用倍增法求两个节点的最近公共祖先(LCA)
 * @param a 节点 a
 * @param b 节点 b
 * @return a 和 b 的最近公共祖先
 */
int lca(int a, int b) {
    // 确保 a 的深度不小于 b
    if (deep[a] < deep[b]) {
        int tmp = a;
        a = b;
        b = tmp;
    }
```

```
    // 将 a 上移到与 b 同一深度
    for (int p = power; p >= 0; p--) {
        if (deep[stjump[a][p]] >= deep[b]) {
            a = stjump[a][p];
        }
    }

    // 如果此时 a == b，则找到 LCA
    if (a == b) {
        return a;
    }

    // 同时上移 a 和 b，直到找到 LCA
    for (int p = power; p >= 0; p--) {
        if (stjump[a][p] != stjump[b][p]) {
            a = stjump[a][p];
            b = stjump[b][p];
        }
    }

    // LCA 是 a 和 b 的父节点
    return stjump[a][0];
}

/**
 * 第二次 DFS：回溯计算每条边的覆盖次数，并计算答案
 * @param u 当前节点
 * @param f 父节点
 */
void dfs2(int u, int f) {
    // 递归处理所有子节点，计算子节点的覆盖次数
    for (int e = head[u], v; e != 0; e = next_[e]) {
        v = to[e];
        if (v != f) {
            dfs2(v, u);  // 后序遍历，先处理子节点
        }
    }

    // 计算每条边的贡献
    for (int e = head[u], v, w; e != 0; e = next_[e]) {
        v = to[e];
        if (v != f) {
```

```
            w = num[v];  // 获取 v 到 u 这条边的覆盖次数

            // 【方案数统计逻辑】
            // 1. 覆盖次数为 0：这条树边不在任何环中，切断它后图会被分成两部分
            //    此时需要再切断一条非树边，但即使切断任意非树边，图仍然不连通
            //    所以共有 m 种方案
            if (w == 0) {
                ans += m;
            }
            // 2. 覆盖次数为 1：这条树边只在一个环中，切断它后需要切断形成该环的非树边
            //    此时只有 1 种方案
            else if (w == 1) {
                ans += 1;
            }
            // 3. 覆盖次数≥2：这条树边在多个环中，切断它后图仍然连通
            //    此时无论切断哪条非树边，图仍然连通，所以有 0 种方案
            // (此处 else 分支可以省略，因为 ans += 0 无影响)

            // 将子节点的覆盖次数累加到父节点，继续向上传递
            num[u] += num[v];
        }
    }
}

int main() {
    // 读取节点数和非树边数
    scanf("%d", &n);
    build();
    scanf("%d", &m);

    // 读取树边并构建邻接表（无向图，每条边添加两次）
    for (int i = 1, u, v; i < n; i++) {
        scanf("%d%d", &u, &v);
        addEdge(u, v);
        addEdge(v, u);
    }

    // 预处理 LCA 所需的深度数组和倍增数组
    dfs1(1, 0);  // 根节点为 1，父节点为 0（无效节点）

    // 处理每条非树边，执行边差分操作
    for (int i = 1, u, v, l; i <= m; i++) {
        scanf("%d%d", &u, &v);
```

```
        l = lca(u, v);  // 计算 u 和 v 的 LCA

        // 【树上边差分操作】
        // 1. 在 u 和 v 处各+1
        // 2. 在 LCA 处-2
        // 3. 通过 DFS 回溯时累加，就能得到每条边被覆盖的次数
        num[u]++;
        num[v]++;
        num[l] -= 2;
    }

    // 计算最终答案
    dfs2(1, 0);

    // 输出结果
    printf("%lld\n", ans);
    return 0;
}
```

==========

文件: Code06_DarkLock1.java

==========

```
/**
 * 暗的连锁 (LOJ 10131)
 * 题目来源：LibreOJ
 * 题目链接：https://loj.ac/problem/10131
 *
 * 题目描述：
 * 给定一棵包含 N 个节点的树（N-1 条边），以及 M 条额外的边（非树边）
 * 每条非树边连接两个节点，与树边一起构成一个连通图
 * 求有多少种方案，通过切断一条树边和一条非树边，使得图变得不连通
 *
 * 算法原理：树上边差分
 * 树上边差分与点差分类似，但针对边进行操作。
 * 对于每条非树边(u,v)，它在原树上会形成一个环，环上的所有树边都被这条非树边覆盖。
 * 通过边差分，我们可以：
 * 1. num[u]++
 * 2. num[v]++
 * 3. num[lca(u,v)] -= 2
 * 最后通过一次 DFS 回溯累加子节点的差分标记，得到每条边被覆盖的次数。
 *
 * 时间复杂度分析：
```

```
 * - 预处理 LCA：O(N log N)
 * - 差分标记：O(M log N)，其中 M 是非树边数量，每次需要计算 LCA
 * - DFS 回溯统计：O(N)
 * 总时间复杂度：O((N + M) log N)
 *
 * 空间复杂度分析：
 * - 树的存储：O(N)
 * - LCA 倍增数组：O(N log N)
 * - 差分数组：O(N)
 * 总空间复杂度：O(N log N)
 *
 * 工程化考量：
 * 1. 使用链式前向星存储树结构，提高空间效率和遍历速度
 * 2. 使用 StreamTokenizer 进行高效输入，处理大量数据时性能优于 Scanner
 * 3. 使用 PrintWriter 进行高效输出，支持缓冲
 * 4. 采用静态成员变量减少对象创建，在算法竞赛中常用此技巧
 *
 * 最优解分析：
 * 树上边差分是解决此类问题的最优解，通过 O(1)的操作标记每条非树边的影响范围，
 * 避免了暴力遍历每条环上的树边，时间复杂度比暴力方法的 O(M*N)有极大提升。
 */
package class122;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;
import java.io.PrintWriter;
import java.io.StreamTokenizer;
import java.util.Arrays;

public class Code06_DarkLock1 {

    /**
     * 最大节点数量
     * 题目中节点数量范围：1 <= N <= 100000
     */
    public static final int MAXN = 100001;

    /**
     * 倍增数组的层数限制
     * 2^17 = 131072 > 100000，足够处理题目中的最大节点数
     */
```

```
public static final int LIMIT = 17;

/**
 * 最大幂次，根据节点数量动态计算
 */
public static int power;

/**
 * 计算 log2(n)的整数部分
 *
 * @param n 输入整数
 * @return log2(n)的整数部分
 */
public static int log2(int n) {
    int ans = 0;
    while ((1 << ans) <= (n >> 1)) {
        ans++;
    }
    return ans;
}

/**
 * 节点数量和非树边数量
 */
public static int n, m;

/**
 * 差分数组
 * num[i]表示节点 i 到其父节点的边，被多少条非树边覆盖
 * 注意：边被表示为子节点的属性，这样可以避免处理根节点的特殊情况
 */
public static int[] num = new int[MAXN];

/**
 * 链式前向星存储树结构
 * head[u]: 节点 u 的第一个边的索引
 * next[e]: 边 e 的下一个边的索引
 * to[e]: 边 e 指向的节点
 * cnt: 当前可用的边索引
 */
public static int[] head = new int[MAXN];
public static int[] next_edge = new int[MAXN << 1]; // 避免与关键字 next 冲突
public static int[] to = new int[MAXN << 1];
```

```
public static int cnt;

/**
 * LCA 相关数组
 * deep[u]: 节点 u 的深度
 * stjump[u][p]: 节点 u 的 2^p 级祖先
 */
public static int[] deep = new int[MAXN];
public static int[][] stjump = new int[MAXN][LIMIT];

/**
 * 最终答案
 */
public static int ans;

/**
 * 初始化算法所需的数据结构
 * 设置数组初始值，准备处理新的测试用例
 */
public static void build() {
    power = log2(n);
    // 初始化差分数组，从 1 开始因为节点编号从 1 开始
    Arrays.fill(num, 1, n + 1, 0);
    // 边索引从 1 开始，0 表示没有边
    cnt = 1;
    // 初始化链式前向星的 head 数组
    Arrays.fill(head, 1, n + 1, 0);
    // 初始化答案
    ans = 0;
}

/**
 * 向链式前向星中添加一条无向边
 *
 * @param u 边的一个端点
 * @param v 边的另一个端点
 */
public static void addEdge(int u, int v) {
    // 添加 u 到 v 的边
    next_edge[cnt] = head[u];
    to[cnt] = v;
    head[u] = cnt++;
}
```

```
/**
 * 第一次DFS，预处理每个节点的深度和倍增跳跃数组
 *
 * @param u 当前处理的节点
 * @param f 当前节点的父节点
 *
 * 时间复杂度：O(N log N)
 * 空间复杂度：O(log N) - 递归调用栈深度
 */
public static void dfs1(int u, int f) {
    // 设置当前节点的深度，父节点深度+1
    deep[u] = deep[f] + 1;
    // 设置当前节点的直接父节点（2^0级祖先）
    stjump[u][0] = f;

    // 预处理倍增数组
    // 利用动态规划思想：u的2^p级祖先 = u的2^(p-1)级祖先的2^(p-1)级祖先
    for (int p = 1; p <= power; p++) {
        stjump[u][p] = stjump[stjump[u][p - 1]][p - 1];
    }

    // 递归处理所有子节点
    for (int e = head[u]; e != 0; e = next_edge[e]) {
        if (to[e] != f) {
            dfs1(to[e], u);
        }
    }
}

/**
 * 使用倍增法计算两个节点的最近公共祖先
 *
 * @param a 第一个节点
 * @param b 第二个节点
 * @return a和b的最近公共祖先
 *
 * 时间复杂度：O(log N)
 * 空间复杂度：O(1)
 */
public static int lca(int a, int b) {
    // 确保a的深度不小于b
    if (deep[a] < deep[b]) {
```

```
        int tmp = a;
        a = b;
        b = tmp;
    }

    // 将 a 向上跳到与 b 同一深度
    for (int p = power; p >= 0; p--) {
        if (deep[stjump[a][p]] >= deep[b]) {
            a = stjump[a][p];
        }
    }

    // 如果此时 a==b，则找到了 LCA
    if (a == b) {
        return a;
    }

    // 继续向上跳，直到找到 LCA
    for (int p = power; p >= 0; p--) {
        if (stjump[a][p] != stjump[b][p]) {
            a = stjump[a][p];
            b = stjump[b][p];
        }
    }

    // 返回它们的父节点
    return stjump[a][0];
}

/**
 * 第二次 DFS，计算每条边被覆盖的次数，并统计满足条件的方案数
 *
 * @param u 当前处理的节点
 * @param f 当前节点的父节点
 *
 * 时间复杂度：O(N)
 * 空间复杂度：O(log N) - 递归调用栈深度
 *
 * 算法逻辑：
 * 1. 先递归处理所有子节点
 * 2. 统计每个子节点到父节点这条边的覆盖次数
 * 3. 根据覆盖次数计算切断这条树边的可行方案数
 * 4. 累加子节点的覆盖次数到当前节点
```

```
 */
public static void dfs2(int u, int f) {
    // 递归处理所有子节点
    for (int e = head[u], v; e != 0; e = next_edge[e]) {
        v = to[e];
        if (v != f) {
            dfs2(v, u);
        }
    }

    // 统计每条边的覆盖次数和方案数
    for (int e = head[u], v, coverage; e != 0; e = next_edge[e]) {
        v = to[e];
        if (v != f) {
            // 获取边(u,v)的覆盖次数，存储在 v 的 num 数组中
            coverage = num[v];

            /**
             * 方案数统计逻辑：
             * - 覆盖次数为 0：这条树边不在任何非树边形成的环中
             *   切断它后，无论切断哪条非树边，图都会不连通
             *   共有 m 种方案（m 是非树边的数量）
             *
             * - 覆盖次数为 1：这条树边恰好只在一个非树边形成的环中
             *   只有切断对应的那条非树边，图才会不连通
             *   共有 1 种方案
             *
             * - 覆盖次数>=2：这条树边在多个非树边形成的环中
             *   切断它后，需要切断所有对应的非树边才能使图不连通
             *   但题目要求只切断一条非树边，因此没有可行方案
             *   共有 0 种方案
             */
            if (coverage == 0) {
                ans += m;  // 可以与任意一条非树边配对
            } else if (coverage == 1) {
                ans += 1;  // 只能与对应的那条非树边配对
            }
            // coverage >= 2 时不需要增加 ans

            // 将子节点的覆盖次数累加到父节点，完成差分标记的传播
            num[u] += num[v];
        }
    }
```

```
}

/**
 * 主函数，处理输入输出并调用相应的算法函数
 *
 * @param args 命令行参数
 * @throws IOException 输入输出异常
 */
public static void main(String[] args) throws IOException {
    // 使用高效的输入方式
    BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
    StreamTokenizer in = new StreamTokenizer(br);
    // 使用高效的输出方式
    PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));

    // 读取节点数
    in.nextToken();
    n = (int) in.nval;
    // 读取非树边数
    in.nextToken();
    m = (int) in.nval;

    // 初始化数据结构
    build();

    // 构建树结构，添加 n-1 条树边
    for (int i = 1, u, v; i < n; i++) {
        in.nextToken();
        u = (int) in.nval;
        in.nextToken();
        v = (int) in.nval;
        addEdge(u, v);
        addEdge(v, u);  // 无向图，添加反向边
    }

    // 预处理 LCA 所需的数据
    // 以节点 1 为根节点进行 DFS
    dfs1(1, 0);

    // 处理所有非树边，执行树上边差分操作
    for (int i = 1, u, v, lca_node; i <= m; i++) {
        in.nextToken();
        u = (int) in.nval;
```

```
            in.nextToken();
            v = (int) in.nval;
            // 计算u和v的LCA
            lca_node = lca(u, v);

            /**
             * 树上边差分核心操作
             * 对于非树边(u,v)，它在原树上会形成一个环
             * 通过边差分，我们标记环上的所有树边
             * 1. 在u处+1
             * 2. 在v处+1
             * 3. 在LCA处-2，抵消多余的标记
             */
            num[u]++;
            num[v]++;
            num[lca_node] -= 2;
        }

        // 计算最终答案
        dfs2(1, 0);

        // 输出结果
        out.println(ans);
        // 确保输出被刷新
        out.flush();
        // 关闭资源
        out.close();
        br.close();
    }

}
```

==========

文件: Code06_DarkLock1.py

==========

```
"""
暗的连锁 (LOJ 10131)

题目来源：LibreOJ
题目链接：https://loj.ac/problem/10131

题目描述：
```

给定一棵包含 N 个节点的树（N-1 条边），以及 M 条额外的边（非树边）
每条非树边连接两个节点，与树边一起构成一个连通图
求有多少种方案，通过切断一条树边和一条非树边，使得图变得不连通

算法原理：树上边差分
树上边差分与点差分类似，但针对边进行操作。
对于每条非树边(u,v)，它在原树上会形成一个环，环上的所有树边都被这条非树边覆盖。
通过边差分，我们可以：
1. num[u]++
2. num[v]++
3. num[lca(u,v)] -= 2
最后通过一次 DFS 回溯累加子节点的差分标记，得到每条边被覆盖的次数。

时间复杂度分析：
- 预处理 LCA：O(N log N)
- 差分标记：O(M log N)，其中 M 是非树边数量，每次需要计算 LCA
- DFS 回溯统计：O(N)
总时间复杂度：O((N + M) log N)

空间复杂度分析：
- 树的存储：O(N)
- LCA 倍增数组：O(N log N)
- 差分数组：O(N)
总空间复杂度：O(N log N)

工程化考量：
1. 使用链式前向星存储树结构，提高空间效率和遍历速度
2. 预处理 log2 值，优化倍增数组的大小
3. 在 Python 中，递归 DFS 对于大规模数据可能存在栈溢出问题
4. 对于节点数量较大的情况，使用全局变量可以减少函数参数传递的开销

最优解分析：
树上边差分是解决此类问题的最优解，通过 O(1)的操作标记每条非树边的影响范围，
避免了暴力遍历每条环上的树边，时间复杂度比暴力方法的 O(M*N)有极大提升。
"""

```
import sys
import math

class DarkChainSolver:
    """
    暗的连锁问题求解类
```

```
该类使用树上边差分算法来解决暗的连锁问题，计算切断一条树边和一条非树边使图不连通的方案数。
"""

def __init__(self, max_nodes=100001):
    """
    初始化求解器

    Args:
        max_nodes: 最大节点数量，默认为 100001
    """
    self.MAXN = max_nodes
    self.LIMIT = 17  # 足够处理 3e5 节点的倍增层数

    # 链式前向星相关变量
    self.head = [0] * self.MAXN
    self.next_edge = [0] * (self.MAXN << 1)  # 乘以 2 处理无向图
    self.to = [0] * (self.MAXN << 1)
    self.cnt = 1  # 边的计数器，从 1 开始避免 0 作为有效索引

    # 差分数组，num[i]表示节点 i 到其父节点的边被多少条非树边覆盖
    self.num = [0] * self.MAXN

    # LCA 相关变量
    self.deep = [0] * self.MAXN  # 深度数组
    self.stjump = [[0] * self.LIMIT for _ in range(self.MAXN)]  # 倍增跳跃数组
    self.power = 0  # 最大幂次

    # 结果变量
    self.ans = 0
    self.n = 0  # 节点数
    self.m = 0  # 非树边数

def add_edge(self, u, v):
    """
    向链式前向星结构中添加一条无向边

    Args:
        u: 边的一个端点
        v: 边的另一个端点
    """
    # 添加 u 到 v 的边
    self.next_edge[self.cnt] = self.head[u]
    self.to[self.cnt] = v
```

```
    self.head[u] = self.cnt
    self.cnt += 1

    # 添加 v 到 u 的边（无向图）
    self.next_edge[self.cnt] = self.head[v]
    self.to[self.cnt] = u
    self.head[v] = self.cnt
    self.cnt += 1

def dfs1(self, u, f):
    """
    第一次 DFS，预处理每个节点的深度和倍增跳跃数组

    Args:
        u: 当前处理的节点
        f: 当前节点的父节点
    """
    # 设置当前节点的深度
    self.deep[u] = self.deep[f] + 1
    # 设置当前节点的直接父节点（2^0 级祖先）
    self.stjump[u][0] = f

    # 预处理倍增数组
    # 利用动态规划思想：u 的 2^p 级祖先 = u 的 2^(p-1)级祖先的 2^(p-1)级祖先
    p = 1
    while p <= self.power:
        self.stjump[u][p] = self.stjump[self.stjump[u][p-1]][p-1]
        p += 1

    # 递归处理所有子节点
    e = self.head[u]
    while e != 0:
        if self.to[e] != f:
            self.dfs1(self.to[e], u)
        e = self.next_edge[e]

def lca(self, a, b):
    """
    使用倍增法计算两个节点的最近公共祖先

    Args:
        a: 第一个节点
        b: 第二个节点
```

```
    Returns:
        a 和 b 的最近公共祖先
    """
    # 确保 a 的深度不小于 b
    if self.deep[a] < self.deep[b]:
        a, b = b, a

    # 将 a 向上跳到与 b 同一深度
    p = self.power
    while p >= 0:
        if self.deep[self.stjump[a][p]] >= self.deep[b]:
            a = self.stjump[a][p]
        p -= 1

    # 如果此时 a==b，则找到了 LCA
    if a == b:
        return a

    # 继续向上跳，直到找到 LCA
    p = self.power
    while p >= 0:
        if self.stjump[a][p] != self.stjump[b][p]:
            a = self.stjump[a][p]
            b = self.stjump[b][p]
        p -= 1

    # 返回它们的父节点
    return self.stjump[a][0]

def dfs2(self, u, f):
    """
    第二次 DFS，计算每条边被覆盖的次数，并统计满足条件的方案数

    Args:
        u: 当前处理的节点
        f: 当前节点的父节点

    算法逻辑：
    1. 先递归处理所有子节点
    2. 统计每个子节点到父节点这条边的覆盖次数
    3. 根据覆盖次数计算切断这条树边的可行方案数
    4. 累加子节点的覆盖次数到当前节点
```

```
    """
    # 递归处理所有子节点
    e = self.head[u]
    while e != 0:
        v = self.to[e]
        if v != f:
            self.dfs2(v, u)
        e = self.next_edge[e]

    # 统计每条边的覆盖次数和方案数
    e = self.head[u]
    while e != 0:
        v = self.to[e]
        if v != f:
            # 获取边(u,v)的覆盖次数，存储在 v 的 num 数组中
            coverage = self.num[v]

            """
            方案数统计逻辑：
            - 覆盖次数为 0：这条树边不在任何非树边形成的环中
              切断它后，无论切断哪条非树边，图都会不连通
              共有 m 种方案（m 是非树边的数量）

            - 覆盖次数为 1：这条树边恰好只在一个非树边形成的环中
              只有切断对应的那条非树边，图才会不连通
              共有 1 种方案

            - 覆盖次数>=2：这条树边在多个非树边形成的环中
              切断它后，需要切断所有对应的非树边才能使图不连通
              但题目要求只切断一条非树边，因此没有可行方案
              共有 0 种方案
            """
            if coverage == 0:
                self.ans += self.m  # 可以与任意一条非树边配对
            elif coverage == 1:
                self.ans += 1       # 只能与对应的那条非树边配对
            # coverage >= 2 时不需要增加 ans

            # 将子节点的覆盖次数累加到父节点，完成差分标记的传播
            self.num[u] += self.num[v]
        e = self.next_edge[e]

def process_non_tree_edges(self):
```

```
    """
    处理所有非树边，执行树上边差分操作

    对于每条非树边(u,v)，它在原树上会形成一个环，环上的所有树边都被这条非树边覆盖。
    使用边差分技巧，我们只需要修改三个点：
    1. num[u]++ - 在u点增加覆盖标记
    2. num[v]++ - 在v点增加覆盖标记
    3. num[lca(u,v)] -= 2 - 在LCA处抵消多余的标记
    """
    for _ in range(self.m):
        u, v = map(int, sys.stdin.readline().split())
        lca_node = self.lca(u, v)

        # 树上边差分核心操作
        self.num[u] += 1
        self.num[v] += 1
        self.num[lca_node] -= 2

def solve(self):
    """
    解决暗的连锁问题

    Returns:
        满足条件的方案数
    """
    # 读取输入
    input_line = sys.stdin.readline().strip()
    while not input_line:
        input_line = sys.stdin.readline().strip()
    self.n, self.m = map(int, input_line.split())

    # 计算最大幂次
    self.power = math.floor(math.log2(self.n)) + 1

    # 构建树结构
    for _ in range(1, self.n):
        input_line = sys.stdin.readline().strip()
        while not input_line:
            input_line = sys.stdin.readline().strip()
        u, v = map(int, input_line.split())
        self.add_edge(u, v)

    # 预处理LCA所需的数据
```

```
        self.dfs1(1, 0)

        # 处理所有非树边
        self.process_non_tree_edges()

        # 计算最终答案
        self.dfs2(1, 0)

        return self.ans

def main():
    """
    主函数

    输入格式：
    第一行：两个整数 n 和 m，分别表示节点数和非树边数
    接下来 n-1 行：每行两个整数，表示树边
    接下来 m 行：每行两个整数，表示非树边

    输出格式：
    一个整数，表示满足条件的方案数
    """
    # 创建求解器实例
    solver = DarkChainSolver()

    # 求解并输出结果
    result = solver.solve()
    print(result)

if __name__ == "__main__":
    main()
```

==========

文件: Code07_TallestCow.cpp

==========

```
#include <stdio.h>
#include <set>
#include <algorithm>

// Tallest Cow (POJ 3263)
// 有 N 头牛站成一行，两头牛能够相互看见，当且仅当它们中间的牛身高都比它们矮
// 已知最高的牛是第 P 头，身高为 H，还知道 R 对关系，每对关系表示两头牛可以相互看见
```

```
// 求每头牛的身高最大可能是多少
// 测试链接 : http://poj.org/problem?id=3263

using namespace std;

const int MAXN = 10001;

int n, p, h, r;
int diff[MAXN];
set<long long> seen;

int main() {
    // 读取输入
    scanf("%d%d%d%d", &n, &p, &h, &r);

    // 处理每对关系
    for (int i = 0; i < r; i++) {
        int a, b;
        scanf("%d%d", &a, &b);

        // 确保 a < b，便于处理
        if (a > b) {
            int temp = a;
            a = b;
            b = temp;
        }

        // 用一个长整型表示一对关系，用于去重
        long long pair = (long long)a * (n + 1) + b;
        if (seen.find(pair) != seen.end()) {
            continue;
        }
        seen.insert(pair);

        // 差分操作：在区间(a, b)内的牛身高要减 1
        // 即在 a+1 位置-1，在 b 位置+1
        diff[a + 1] -= 1;
        diff[b] += 1;
    }

    // 通过前缀和计算每头牛的相对身高，然后加上最高身高 h 得到实际身高
    int height = 0;
    for (int i = 1; i <= n; i++) {
```

```
        height += diff[i];
        printf("%d\n", h + height);
    }

    return 0;
}
```

==========

文件: Code07_TallestCow.java

==========

```
package class122;

/**
 * Tallest Cow (POJ 3263)
 *
 * 题目来源：POJ 3263
 * 题目链接：http://poj.org/problem?id=3263
 *
 * 题目描述：
 * 有N头牛站成一行，两头牛能够相互看见，当且仅当它们中间的牛身高都比它们矮。
 * 已知最高的牛是第P头，身高为H，还知道R对关系，每对关系表示两头牛可以相互看见。
 * 求每头牛的身高最大可能是多少。
 *
 * 算法原理：线性差分数组
 * 这是一个线性差分数组的经典应用。对于每对可以相互看见的牛(a,b)，它们之间的所有牛身高都要比它们矮，
 * 也就是区间[a+1, b-1]内的牛身高都要减1。
 *
 * 差分数组的核心思想：
 * 对于区间[l,r]的修改操作，我们只需要：
 * 1. diff[l] += d
 * 2. diff[r+1] -= d
 * 最后通过前缀和还原原数组。
 *
 * 在本题中，对于每对关系(a,b)：
 * 1. diff[a+1] -= 1
 * 2. diff[b] += 1
 * 最后通过前缀和计算每头牛的相对身高，然后加上最高身高H得到实际身高。
 *
 * 时间复杂度分析：
 * - 处理关系：O(R)
 * - 计算前缀和：O(N)
```

```
 * 总时间复杂度：O(N + R)
 *
 * 空间复杂度分析：
 * - 差分数组：O(N)
 * - 去重集合：O(R)
 * 总空间复杂度：O(N + R)
 *
 * 工程化考量：
 * 1. 使用 HashSet 进行关系去重，避免重复处理相同的关系
 * 2. 使用 StreamTokenizer 进行高效输入，处理大量数据时性能优于 Scanner
 * 3. 使用 PrintWriter 进行高效输出，支持缓冲
 * 4. 关系表示采用长整型编码，避免使用 Pair 类的开销
 *
 * 最优解分析：
 * 线性差分数组是解决此类区间修改问题的最优解，通过 O(1) 的操作标记区间修改，
 * 避免了暴力遍历每个区间的所有元素，时间复杂度比暴力方法的 O(R*N) 有极大提升。
 */
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;
import java.io.PrintWriter;
import java.io.StreamTokenizer;
import java.util.HashSet;
import java.util.Set;

public class Code07_TallestCow {

    /**
     * 主函数，处理输入输出并调用相应的算法函数
     *
     * @param args 命令行参数
     * @throws IOException 输入输出异常
     */
    public static void main(String[] args) throws IOException {
        // 使用高效的输入方式
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        StreamTokenizer in = new StreamTokenizer(br);
        // 使用高效的输出方式
        PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));

        // 读取输入
        in.nextToken();
```

```
int n = (int) in.nval; // 牛的数量
in.nextToken();
int p = (int) in.nval; // 最高牛的编号
in.nextToken();
int h = (int) in.nval; // 最高牛的身高
in.nextToken();
int r = (int) in.nval; // 关系数量

// 差分数组，初始值为0
// diff[i]表示第i头牛相对身高与前一头牛的差值
int[] diff = new int[n + 1];

// 用于去重，避免重复处理相同的关系
// 使用HashSet存储已处理的关系，提高查找效率
Set<Long> seen = new HashSet<>();

// 处理每对关系
for (int i = 0; i < r; i++) {
    in.nextToken();
    int a = (int) in.nval;
    in.nextToken();
    int b = (int) in.nval;

    // 确保a < b，便于处理
    // 这样可以统一处理逻辑，无论输入顺序如何
    if (a > b) {
        int temp = a;
        a = b;
        b = temp;
    }

    // 用一个长整型表示一对关系，用于去重
    // 编码方式：a * (n + 1) + b，确保唯一性
    long pair = (long) a * (n + 1) + b;
    if (seen.contains(pair)) {
        // 如果关系已处理过，则跳过
        continue;
    }
    seen.add(pair);

    /**
     * 差分操作：在区间(a, b)内的牛身高要减1
     * 即在a+1位置-1，在b位置+1
```

```
             * 这样通过前缀和计算时，区间(a,b)内的元素都会减 1
             */
            diff[a + 1] -= 1;
            diff[b] += 1;
        }

        // 通过前缀和计算每头牛的相对身高，然后加上最高身高 h 得到实际身高
        // height 表示当前牛相对最高牛的身高差
        int height = 0;
        for (int i = 1; i <= n; i++) {
            // 累加差分值，得到相对身高
            height += diff[i];
            // 输出实际身高：最高身高 + 相对身高差
            out.println(h + height);
        }

        // 确保输出被刷新
        out.flush();
        // 关闭资源
        out.close();
        br.close();
    }
}
```

==========

文件: Code07_TallestCow.py

==========

```
import sys

"""
Tallest Cow (POJ 3263)

题目来源：POJ 3263
题目链接：http://poj.org/problem?id=3263

题目描述：
有 N 头牛站成一行，两头牛能够相互看见，当且仅当它们中间的牛身高都比它们矮。
已知最高的牛是第 P 头，身高为 H，还知道 R 对关系，每对关系表示两头牛可以相互看见。
求每头牛的身高最大可能是多少。

算法原理：线性差分数组
这是一个线性差分数组的经典应用。对于每对可以相互看见的牛(a,b)，它们之间的所有牛身高都要比它们矮，
```

也就是区间[a+1, b-1]内的牛身高都要减 1。

差分数组的核心思想：
对于区间[l,r]的修改操作，我们只需要：
1. diff[l] += d
2. diff[r+1] -= d
最后通过前缀和还原原数组。

在本题中，对于每对关系(a,b)：
1. diff[a+1] -= 1
2. diff[b] += 1
最后通过前缀和计算每头牛的相对身高，然后加上最高身高 H 得到实际身高。

时间复杂度分析：
- 处理关系：O(R)
- 计算前缀和：O(N)
总时间复杂度：O(N + R)

空间复杂度分析：
- 差分数组：O(N)
- 去重集合：O(R)
总空间复杂度：O(N + R)

工程化考量：
1. 使用 set 进行关系去重，避免重复处理相同的关系
2. 使用 sys.stdin.readline()进行高效输入，处理大量数据时性能优于 input()
3. 关系表示采用元组编码，简洁且高效

最优解分析：
线性差分数组是解决此类区间修改问题的最优解，通过 O(1)的操作标记区间修改，
避免了暴力遍历每个区间的所有元素，时间复杂度比暴力方法的 O(R*N)有极大提升。
"""

```
def main():
    """
    主函数，处理输入输出并调用相应的算法函数
    """
    # 读取输入
    # n: 牛的数量
    # p: 最高牛的编号
    # h: 最高牛的身高
    # r: 关系数量
    n, p, h, r = map(int, sys.stdin.readline().split())
```

```
    # 差分数组，初始值为0
    # diff[i]表示第i头牛相对身高与前一头牛的差值
    diff = [0] * (n + 1)

    # 用于去重，避免重复处理相同的关系
    # 使用set存储已处理的关系，提高查找效率
    seen = set()

    # 处理每对关系
    for i in range(r):
        a, b = map(int, sys.stdin.readline().split())

        # 确保a < b，便于处理
        # 这样可以统一处理逻辑，无论输入顺序如何
        if a > b:
            a, b = b, a

        # 用一个元组表示一对关系，用于去重
        pair = (a, b)
        if pair in seen:
            # 如果关系已处理过，则跳过
            continue
        seen.add(pair)

        """
        差分操作：在区间(a, b)内的牛身高要减1
        即在a+1位置-1，在b位置+1
        这样通过前缀和计算时，区间(a,b)内的元素都会减1
        """
        diff[a + 1] -= 1
        diff[b] += 1

    # 通过前缀和计算每头牛的相对身高，然后加上最高身高h得到实际身高
    # height表示当前牛相对最高牛的身高差
    height = 0
    for i in range(1, n + 1):
        # 累加差分值，得到相对身高
        height += diff[i]
        # 输出实际身高：最高身高 + 相对身高差
        print(h + height)

if __name__ == "__main__":
```

main()

==========

文件: Code08_SquirrelNewHome.cpp
==========
```
/**
 * 松鼠的新家 (洛谷 P3258)
 * 题目来源：洛谷
 * 题目链接：https://www.luogu.com.cn/problem/P3258
 *
 * 题目描述：
 * 松鼠家族的成员需要在树上移动，从一个节点到另一个节点。
 * 给定一棵包含 N 个节点的树，以及 N-1 次移动操作。
 * 每次移动操作表示从节点 a 移动到节点 b，经过的路径上的所有节点（包括起点和终点）都会被访问一次。
 * 求每个节点被访问的次数。
 *
 * 算法原理：树上点差分
 * 树上差分是一种将路径操作转化为点标记操作的高效算法。
 * 对于树上的路径 u->v，我们需要让路径上的所有节点计数加 1。
 * 通过点差分，我们可以：
 * 1. diff[u]++
 * 2. diff[v]++
 * 3. diff[lca(u,v)]--
 * 4. diff[parent(lca(u,v))]--
 * 最后通过一次 DFS 回溯累加子节点的差分标记，得到每个节点的最终计数。
 *
 * 时间复杂度分析：
 * - 预处理 LCA：O(N log N)
 * - 差分标记：O(M)，其中 M 是操作次数
 * - DFS 回溯统计：O(N)
 * 总时间复杂度：O(N log N + M)，对于本题 M=N-1，所以总时间复杂度为 O(N log N)
 *
 * 空间复杂度分析：
 * - 树的存储：O(N)
 * - LCA 倍增数组：O(N log N)
 * - 差分数组：O(N)
 * 总空间复杂度：O(N log N)
 *
 * 工程化考量：
 * 1. 使用链式前向星存储树结构，节省空间
 * 2. 使用 scanf/printf 进行高效输入输出，避免 cin/cout 的性能开销
 * 3. 预处理 log2 值，优化倍增数组的大小
```

```
 * 4. 递归DFS实现简洁明了，但需注意在C++中递归深度过大可能导致栈溢出
 * 5. 注意全局变量的初始化和数组大小的设置
 *
 * 最优解分析：
 * 树上差分是解决此类路径覆盖问题的最优解，相比暴力遍历每条路径的O(N*M)复杂度，
 * 树上差分可以将时间复杂度优化到O(N log N)，在大规模数据下效率提升显著。
 */

#include <iostream>
#include <cstdio>
#include <algorithm>

using namespace std;

/**
 * 松鼠的新家 (洛谷 P3258) - C++实现
 * 树上点差分算法的经典应用
 */

/**
 * 最大节点数，根据题目数据范围设定
 * 题目中N最大为3e5，设置为300001以避免越界
 */
const int MAXN = 300001;

/**
 * 倍增数组的最大级别
 * 2^19 = 524,288 > 3e5，足够处理最大节点数
 */
const int MAX_LEVEL = 19;

/**
 * 全局变量声明
 */
int n;                          // 节点数
int max_level;                  // 倍增数组的最大级别
int visit_order[MAXN];          // 存储节点访问顺序

/**
 * 差分数组，用于记录每个节点的差分标记
 */
int diff[MAXN];
```

```
/**
 * 链式前向星存储树结构
 * head[u]：节点 u 的第一条边
 * next_edge[e]：边 e 的下一条边
 * to[e]：边 e 的目标节点
 * edge_count：边计数器
 */
int head[MAXN], next_edge[MAXN << 1], to_[MAXN << 1], edge_count;

/**
 * LCA 相关数组
 * depth[u]：节点 u 的深度
 * stjump[u][p]：节点 u 的 2^p 级祖先
 */
int depth[MAXN];
int stjump[MAXN][MAX_LEVEL];

// 函数声明
int find_lca(int a, int b);
void calculate_access_counts(int u, int parent);

/**
 * 计算 log2(n)的整数部分，用于确定倍增数组需要的最大级别
 *
 * @param n 输入的节点数
 * @return 最大的 k 使得 2^k <= n/2
 *
 * 该函数通过位运算高效计算 log2 值，避免使用浮点数运算
 */
int calculate_max_level(int n) {
    int result = 0;
    while ((1 << result) <= (n >> 1)) {
        result++;
    }
    return result;
}

/**
 * 初始化树结构和相关数据
 *
 * 功能：
 * 1. 计算倍增数组所需的最大级别
 * 2. 初始化差分数组和邻接表
```

```
 * 3. 重置边计数器
 *
 * 注意：
 * 在C++中，全局变量会被自动初始化为0，这里显式初始化确保正确性
 */
void initialize_tree() {
    max_level = calculate_max_level(n);
    for (int i = 0; i < MAXN; i++) {
        diff[i] = 0;
        head[i] = 0;
    }
    edge_count = 1;  // 边编号从1开始，方便链式前向星操作
}

/**
 * 向树中添加一条无向边
 *
 * @param u 边的起始节点
 * @param v 边的结束节点
 *
 * 注意：
 * 由于树是无向的，通常需要调用两次add_edge(u, v)和add_edge(v, u)
 * 这里只实现了单向添加，在main函数中需要双向添加
 */
void add_edge(int u, int v) {
    next_edge[edge_count] = head[u];
    to_[edge_count] = v;
    head[u] = edge_count++;
}

/**
 * 预处理LCA所需的数据结构
 * 通过深度优先搜索，记录每个节点的深度和倍增跳跃数组
 *
 * @param u 当前处理的节点
 * @param parent 当前节点的父节点
 *
 * 时间复杂度：O(N log N)，每个节点需要处理log N次倍增跳跃
 */
void preprocess_lca(int u, int parent) {
    // 设置当前节点的深度（父节点深度+1）
    depth[u] = depth[parent] + 1;
    // 设置当前节点的直接父节点
```

```
    stjump[u][0] = parent;

    // 预处理倍增数组，stjump[u][p]表示 u 的 2^p 级祖先
    // 利用动态规划的思想：u 的 2^p 级祖先 = u 的 2^(p-1)级祖先的 2^(p-1)级祖先
    for (int p = 1; p <= max_level; p++) {
        // 确保不会越界到根节点的父节点（0）
        if (stjump[u][p-1] != 0) {
            stjump[u][p] = stjump[stjump[u][p - 1]][p - 1];
        }
    }

    // 深度优先遍历所有子节点
    for (int e = head[u]; e != 0; e = next_edge[e]) {
        int v = to_[e];
        // 避免回到父节点，造成无限递归
        if (v != parent) {
            preprocess_lca(v, u);
        }
    }
}

/**
 * 主函数，处理输入、算法执行和输出
 *
 * 输入格式：
 * - 第一行：一个整数 n，表示节点数
 * - 第二行：n 个整数，表示访问顺序
 * - 接下来 n-1 行：每行两个整数 u 和 v，表示树中的一条无向边
 *
 * 输出格式：
 * - 输出 n 行，每行一个整数，表示对应节点被访问的次数
 */
int main() {
    // 读取节点数量
    scanf("%d", &n);

    // 初始化数据结构
    initialize_tree();

    // 读取每个节点的访问顺序
    for (int i = 1; i <= n; i++) {
        scanf("%d", &visit_order[i]);
    }
```

```
// 读取树的边，构建无向树
for (int i = 1, u, v; i < n; i++) {
    scanf("%d%d", &u, &v);
    add_edge(u, v);
    add_edge(v, u);  // 无向树需要添加双向边
}

// 预处理LCA所需的数据（根节点设为1）
preprocess_lca(1, 0);

// 处理每次移动操作 - 执行树上点差分
for (int i = 1; i < n; i++) {
    int u = visit_order[i];          // 当前移动的起点
    int v = visit_order[i + 1];      // 当前移动的终点
    int lca_node = find_lca(u, v);  // 计算u和v的最近公共祖先
    int lca_father = stjump[lca_node][0]; // LCA的父节点

    /**
     * 树上点差分核心操作：
     * 对于路径u->v，我们希望路径上的所有节点计数加1
     * 通过差分技巧，我们只需要修改四个点：
     * 1. diff[u]++ - 在起点增加标记
     * 2. diff[v]++ - 在终点增加标记
     * 3. diff[lca_node]-- - 在LCA处抵消一次（因为u和v都会到达LCA）
     * 4. diff[lca_father]-- - 在LCA的父节点处抵消一次
     */
    diff[u]++;
    diff[v]++;
    diff[lca_node]--;
    // 注意：根节点的父节点是0，不需要对0进行操作
    if (lca_father != 0) {
        diff[lca_father]--;
    }
}

// 执行DFS回溯，计算每个节点的最终访问次数
calculate_access_counts(1, 0);

// 输出结果，需要注意题目中的特殊处理
/**
 * 为什么需要特殊处理？
 * 因为题目中松鼠的移动路径是连续的，除了最后一个终点外，
```

```
     * 每个节点如果是某次移动的终点，它也会是下一次移动的起点。
     * 但实际上，松鼠在移动时，起点只算一次访问，而不是两次。
     * 因此，除了最后一个节点外，其他节点的访问次数需要减 1。
     */
    for (int i = 1; i <= n; i++) {
        // 最后一个节点（即 visit_order[n]）不需要减 1
        if (i == visit_order[n]) {
            printf("%d\n", diff[i]);
        } else {
            // 其他节点需要减 1
            printf("%d\n", diff[i] - 1);
        }
    }

    return 0;
}

/**
 * 使用倍增法计算两个节点的最近公共祖先(LCA)
 *
 * @param a 第一个节点
 * @param b 第二个节点
 * @return a 和 b 的最近公共祖先
 *
 * 算法步骤：
 * 1. 确保 a 的深度不小于 b
 * 2. 将 a 向上跳跃到与 b 同一深度
 * 3. 如果此时 a==b，则直接返回 a 作为 LCA
 * 4. 否则，a 和 b 同时向上跳跃，直到它们的父节点相同
 * 5. 返回最终的父节点作为 LCA
 *
 * 时间复杂度：O(log N)，每次查询最多跳跃 log N 次
 */
int find_lca(int a, int b) {
    // 步骤 1：确保 a 的深度不小于 b
    if (depth[a] < depth[b]) {
        // 交换 a 和 b
        swap(a, b);
    }

    // 步骤 2：将 a 向上跳到与 b 同一深度
    // 从最高级别开始尝试跳跃，确保最大步长
    for (int p = max_level; p >= 0; p--) {
```

```
        // 只有当跳跃后的深度不小于 b 的深度时才跳跃
        if (stjump[a][p] != 0 && depth[stjump[a][p]] >= depth[b]) {
            a = stjump[a][p];
        }
    }

    // 步骤 3：如果 a 和 b 相遇，说明找到了 LCA
    if (a == b) {
        return a;
    }

    // 步骤 4：同时向上跳跃，直到找到 LCA 的直接子节点
    for (int p = max_level; p >= 0; p--) {
        if (stjump[a][p] != stjump[b][p]) {
            a = stjump[a][p];
            b = stjump[b][p];
        }
    }

    // 步骤 5：返回它们的父节点作为 LCA
    return stjump[a][0];
}

/**
 * 计算每个节点的最终访问次数
 * 通过深度优先搜索回溯，累加子节点的差分标记
 * 这是树上差分的关键步骤，将局部标记转化为全局计数
 *
 * @param u 当前处理的节点
 * @param parent 当前节点的父节点
 *
 * 时间复杂度：O(N)，每个节点和边只被访问一次
 *
 * 注意：该 DFS 必须在所有差分标记完成后执行
 */
void calculate_access_counts(int u, int parent) {
    // 步骤 1：先递归处理所有子节点
    // 采用后序遍历的方式，确保子节点的计数先计算完成
    for (int e = head[u]; e != 0; e = next_edge[e]) {
        int v = to_[e];
        if (v != parent) {
            calculate_access_counts(v, u);
        }
```

```
    }

    // 步骤2：将子节点的访问次数累加到当前节点
    // 这一步实现了差分标记的传播和累加
    for (int e = head[u]; e != 0; e = next_edge[e]) {
        int v = to_[e];
        if (v != parent) {
            diff[u] += diff[v];
        }
    }
}
```

==========

文件: Code08_SquirrelNewHome.java

==========

```
package class122;

/**
 * 松鼠的新家 (洛谷 P3258)
 * 题目来源：洛谷
 * 题目链接：https://www.luogu.com.cn/problem/P3258
 *
 * 题目描述：
 * 松鼠家族的成员需要在树上移动，从一个节点到另一个节点。
 * 给定一棵包含N个节点的树，以及N-1次移动操作。
 * 每次移动操作表示从节点a移动到节点b，经过的路径上的所有节点（包括起点和终点）都会被访问一次。
 * 求每个节点被访问的次数。
 *
 * 算法原理：树上点差分
 * 树上差分是一种将路径操作转化为点标记操作的高效算法。
 * 对于树上的路径u->v，我们需要让路径上的所有节点计数加1。
 * 通过点差分，我们可以：
 * 1. diff[u]++
 * 2. diff[v]++
 * 3. diff[lca(u,v)]--
 * 4. diff[parent(lca(u,v))]--
 * 最后通过一次DFS回溯累加子节点的差分标记，得到每个节点的最终计数。
 *
 * 时间复杂度分析：
 * - 预处理LCA：O(N log N)
 * - 差分标记：O(M)，其中M是操作次数
 * - DFS回溯统计：O(N)
```

```
 * 总时间复杂度：O(N log N + M)，对于本题 M=N-1，所以总时间复杂度为 O(N log N)
 *
 * 空间复杂度分析：
 * - 树的存储：O(N)
 * - LCA 倍增数组：O(N log N)
 * - 差分数组：O(N)
 * 总空间复杂度：O(N log N)
 *
 * 工程化考量：
 * 1. 使用链式前向星存储树结构，节省空间
 * 2. 采用 BufferedReader + StreamTokenizer 进行快速输入，提高处理大数据的能力
 * 3. 使用 PrintWriter 进行高效输出
 * 4. 预处理 log2 值，优化倍增数组的大小
 * 5. 递归 DFS 实现简洁明了，但需注意递归深度问题
 *
 * 最优解分析：
 * 树上差分是解决此类路径覆盖问题的最优解，相比暴力遍历每条路径的 O(N*M) 复杂度，
 * 树上差分可以将时间复杂度优化到 O(N log N)，在大规模数据下效率提升显著。
 */

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;
import java.io.PrintWriter;
import java.io.StreamTokenizer;
import java.util.Arrays;

public class Code08_SquirrelNewHome {

    /**
     * 最大节点数，根据题目数据范围设定
     * 题目中 N 最大为 3e5，设置为 300001 以避免越界
     */
    public static final int MAXN = 300001;

    /**
     * 倍增数组的大小限制
     * 2^19 = 524,288 > 3e5，足够处理最大节点数
     */
    public static final int LIMIT = 19;

    // 节点数量
```

```
public static int n;

// 倍增数组的幂次
public static int power;

/**
 * 计算 log2(n)的整数部分，用于确定倍增数组需要的最大幂次
 *
 * @param n 输入的节点数
 * @return 最大的 k 使得 2^k <= n/2
 *
 * 该方法通过位运算高效计算 log2 值，避免使用 Math.log 函数带来的精度问题
 */
public static int log2(int n) {
    int ans = 0;
    while ((1 << ans) <= (n >> 1)) {
        ans++;
    }
    return ans;
}

// 差分数组，用于记录每个节点被访问的次数
public static int[] diff = new int[MAXN];

// 链式前向星存储树结构
public static int[] head = new int[MAXN];
// 使用 next_edge 避免与关键字冲突
public static int[] next_edge = new int[MAXN << 1];
public static int[] to = new int[MAXN << 1];
public static int edgeCount;

// 深度数组和倍增跳跃数组，用于 LCA 计算
public static int[] depth = new int[MAXN];
public static int[][] stjump = new int[MAXN][LIMIT];

/**
 * 初始化函数，重置数据结构
 *
 * 功能：
 * 1. 计算倍增数组所需的最大幂次
 * 2. 初始化差分数组和邻接表
 * 3. 重置边计数器
 */
```

```
public static void build() {
    power = log2(n);
    // 只初始化有效范围的数组元素，提高效率
    Arrays.fill(diff, 1, n + 1, 0);
    edgeCount = 1;  // 边编号从 1 开始
    Arrays.fill(head, 1, n + 1, 0);
}

/**
 * 向树中添加一条无向边
 *
 * @param u 边的起始节点
 * @param v 边的结束节点
 *
 * 注意：由于树是无向的，需要调用两次 addEdge(u, v)和 addEdge(v, u)
 */
public static void addEdge(int u, int v) {
    next_edge[edgeCount] = head[u];
    to[edgeCount] = v;
    head[u] = edgeCount++;
}

/**
 * 第一次 DFS，预处理每个节点的深度和倍增跳跃数组
 * 该 DFS 构建 LCA 所需的数据结构
 *
 * @param u 当前处理的节点
 * @param f 当前节点的父节点
 *
 * 时间复杂度：O(N log N)，每个节点需要处理 log N 次倍增跳跃
 */
public static void dfs1(int u, int f) {
    // 设置当前节点的深度（父节点深度+1）
    depth[u] = depth[f] + 1;
    // 设置当前节点的直接父节点
    stjump[u][0] = f;

    // 预处理倍增数组，stjump[u][p]表示 u 的 2^p 级祖先
    // 利用动态规划的思想：u 的 2^p 级祖先 = u 的 2^(p-1)级祖先的 2^(p-1)级祖先
    for (int p = 1; p <= power; p++) {
        stjump[u][p] = stjump[stjump[u][p - 1]][p - 1];
    }
```

```
    // 深度优先遍历所有子节点
    for (int e = head[u]; e != 0; e = next_edge[e]) {
        int v = to[e];
        // 避免回到父节点，造成无限递归
        if (v != f) {
            dfs1(v, u);
        }
    }
}

/**
 * 使用倍增法计算两个节点的最近公共祖先(LCA)
 *
 * @param a 第一个节点
 * @param b 第二个节点
 * @return a 和 b 的最近公共祖先
 *
 * 算法步骤：
 * 1. 确保 a 的深度不小于 b
 * 2. 将 a 向上跳跃到与 b 同一深度
 * 3. 如果此时 a==b，则直接返回 a 作为 LCA
 * 4. 否则，a 和 b 同时向上跳跃，直到它们的父节点相同
 * 5. 返回最终的父节点作为 LCA
 *
 * 时间复杂度：O(log N)，每次查询最多跳跃 log N 次
 */
public static int lca(int a, int b) {
    // 步骤 1：确保 a 的深度不小于 b
    if (depth[a] < depth[b]) {
        // 交换 a 和 b
        int tmp = a;
        a = b;
        b = tmp;
    }

    // 步骤 2：将 a 向上跳到与 b 同一深度
    // 从最高幂次开始尝试跳跃，确保最大步长
    for (int p = power; p >= 0; p--) {
        // 只有当跳跃后的深度不小于 b 的深度时才跳跃
        if (depth[stjump[a][p]] >= depth[b]) {
            a = stjump[a][p];
        }
    }
```

```
    // 步骤 3：如果 a 和 b 相遇，说明找到了 LCA
    if (a == b) {
        return a;
    }

    // 步骤 4：同时向上跳跃，直到找到 LCA 的直接子节点
    for (int p = power; p >= 0; p--) {
        if (stjump[a][p] != stjump[b][p]) {
            a = stjump[a][p];
            b = stjump[b][p];
        }
    }

    // 步骤 5：返回它们的父节点作为 LCA
    return stjump[a][0];
}

/**
 * 第二次 DFS，通过回溯累加子节点的差分标记，计算每个节点被访问的最终次数
 * 这是树上差分的关键步骤，将局部标记转化为全局计数
 *
 * @param u 当前处理的节点
 * @param f 当前节点的父节点
 *
 * 时间复杂度：O(N)，每个节点和边只被访问一次
 *
 * 注意：该 DFS 必须在所有差分标记完成后执行
 */
public static void dfs2(int u, int f) {
    // 步骤 1：先递归处理所有子节点
    // 采用后序遍历的方式，确保子节点的计数先计算完成
    for (int e = head[u], v; e != 0; e = next_edge[e]) {
        v = to[e];
        if (v != f) {
            dfs2(v, u);
        }
    }

    // 步骤 2：将子节点的访问次数累加到当前节点
    // 这一步实现了差分标记的传播和累加
    for (int e = head[u], v; e != 0; e = next_edge[e]) {
        v = to[e];
```

```
            if (v != f) {
                diff[u] += diff[v];
            }
        }

        // 注意：此时 diff[u]已经包含了该节点的所有子树中的差分标记累加结果
        // 对于根节点，其 diff 值即为整个树的总覆盖次数
    }

    /**
     * 主函数，处理输入、算法执行和输出
     *
     * 输入格式：
     * 第一行：节点数 n
     * 第二行：n 个整数，表示节点访问顺序
     * 接下来 n-1 行：每行两个整数，表示树的边
     *
     * 输出格式：
     * 输出 n 行，每行一个整数，表示每个节点被访问的次数
     */
    public static void main(String[] args) throws IOException {
// 使用高效的输入输出方式
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        StreamTokenizer in = new StreamTokenizer(br);
        PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));

        // 读取节点数量
        in.nextToken();
        n = (int) in.nval;

        // 初始化
        build();

        // 读取每个节点的访问顺序
        int[] order = new int[n + 1];
        for (int i = 1; i <= n; i++) {
            in.nextToken();
            order[i] = (int) in.nval;
        }

        // 读取树的边，构建无向树
        for (int i = 1, u, v; i < n; i++) {
            in.nextToken();
```

```
        u = (int) in.nval;
        in.nextToken();
        v = (int) in.nval;
        // 无向树需要添加双向边
        addEdge(u, v);
        addEdge(v, u);
    }

    // 预处理深度和倍增数组（根节点设为1）
    dfs1(1, 0);

    // 处理每次移动操作 - 执行树上点差分
    for (int i = 1, u, v, lcaNode, lcaFather; i < n; i++) {
        u = order[i];      // 当前移动的起点
        v = order[i + 1];  // 当前移动的终点
        lcaNode = lca(u, v);   // 计算u和v的最近公共祖先
        lcaFather = stjump[lcaNode][0]; // LCA的父节点

        /**
         * 树上点差分核心操作：
         * 对于路径u->v，我们希望路径上的所有节点计数加1
         * 通过差分技巧，我们只需要修改四个点：
         * 1. diff[u]++ - 在起点增加标记
         * 2. diff[v]++ - 在终点增加标记
         * 3. diff[lca_node]-- - 在LCA处抵消一次（因为u和v都会到达LCA）
         * 4. diff[lcafather]-- - 在LCA的父节点处抵消一次
         *
         * 这样，当执行dfs2回溯累分时，整个路径上的节点都会被正确计数
         */
        diff[u]++;
        diff[v]++;
        diff[lcaNode]--;
        diff[lcaFather]--;
    }

    // 执行第二次DFS，通过回溯累加子节点的差分标记，计算每个节点的最终访问次数
    dfs2(1, 0);

    // 输出结果，需要注意题目中的特殊处理
    /**
     * 为什么需要特殊处理？
     * 因为题目中松鼠的移动路径是连续的，除了最后一个终点外，
     * 每个节点如果是某次移动的终点，它也会是下一次移动的起点。
```

```
         * 但实际上，松鼠在移动时，起点只算一次访问，而不是两次。
         * 因此，除了最后一个节点外，其他节点的访问次数需要减 1。
         */
        for (int i = 1; i <= n; i++) {
            // 最后一个节点（即 order[n]）不需要减 1
            if (i == order[n]) {
                out.println(diff[i]);
            } else {
                // 其他节点需要减 1
                out.println(diff[i] - 1);
            }
        }

        // 关闭资源
        out.flush();
        out.close();
        br.close();
    }
}
```

==========

文件: Code08_SquirrelNewHome.py

==========

```
# 松鼠的新家 (洛谷 P3258)
# 题目来源：洛谷
# 题目链接：https://www.luogu.com.cn/problem/P3258

"""
松鼠的新家问题（树上点差分算法实现）

题目描述：
松鼠家族的成员需要在树上移动，从一个节点到另一个节点。
给定一棵包含 N 个节点的树，以及 N-1 次移动操作。
每次移动操作表示从节点 a 移动到节点 b，经过的路径上的所有节点（包括起点和终点）都会被访问一次。
求每个节点被访问的次数。

算法原理：树上点差分
树上差分是一种将路径操作转化为点标记操作的高效算法。
对于树上的路径 u->v，我们需要让路径上的所有节点计数加 1。
通过点差分，我们可以：
1. diff[u]++ - 在起点增加标记
2. diff[v]++ - 在终点增加标记
```

3. diff[lca(u,v)]-- - 在LCA处抵消一次（因为u和v都会到达LCA）
4. diff[parent(lca(u,v))]-- - 在LCA的父节点处抵消一次
最后通过一次DFS回溯累加子节点的差分标记，得到每个节点的最终计数。

时间复杂度分析：
- 预处理LCA：O(N log N)
- 差分标记：O(M)，其中M是操作次数
- DFS回溯统计：O(N)
总时间复杂度：O(N log N + M)，对于本题M=N-1，所以总时间复杂度为O(N log N)

空间复杂度分析：
- 树的存储：O(N)
- LCA倍增数组：O(N log N)
- 差分数组：O(N)
总空间复杂度：O(N log N)

工程化考量：
1. 使用邻接表存储树结构，Python中使用defaultdict(list)实现
2. 使用sys.stdin.read进行一次性读取，避免多次IO操作
3. 预处理log2值，优化倍增数组的大小
4. 递归DFS在Python中存在递归深度限制，对于大规模数据（>1000层）可能需要使用非递归DFS
5. 注意Python中的索引从0开始还是从1开始的问题

最优解分析：
树上差分是解决此类路径覆盖问题的最优解，相比暴力遍历每条路径的O(N*M)复杂度，
树上差分可以将时间复杂度优化到O(N log N)，在大规模数据下效率提升显著。
"""

```
import sys
import math
from collections import defaultdict

class SquirrelTreeSolver:
    """
    松鼠的新家问题求解器

    该类实现了树上点差分算法，通过两次DFS和LCA（最近公共祖先）计算，
    可以在O(N log N)的时间复杂度内高效解决树上路径覆盖问题。
    """

    # 最大倍增层数，2^19 = 524,288 > 3e5，足够处理题目中最大数据范围
    MAX_LEVEL = 19
```

```
def __init__(self, n):
    """
    初始化树差分数据结构

    参数:
        n: 节点数量
    """
    self.n = n
    # 计算最大的幂次，确保 2^power > n
    self.max_level = math.floor(math.log2(n)) + 1
    # 确保不超过预定义的最大层数
    self.max_level = min(self.max_level, self.MAX_LEVEL)
    # 差分数组，索引从 1 开始（与题目节点编号保持一致）
    self.diff = [0] * (n + 1)
    # 邻接表存储树结构
    self.tree = defaultdict(list)
    # 深度数组，记录每个节点的深度
    self.depth = [0] * (n + 1)
    # 父节点数组，记录每个节点的直接父节点
    self.parent = [0] * (n + 1)
    # 倍增数组，stjump[u][p]表示 u 的 2^p 级祖先
    self.stjump = [[0] * (self.MAX_LEVEL + 1) for _ in range(n + 1)]

def add_edge(self, u, v):
    """
    向树中添加一条无向边

    参数:
        u: 边的起始节点
        v: 边的结束节点

    说明：由于树是无向的，需要在两个方向都添加边，
    这样在遍历树时可以双向访问相邻节点。
    """
    self.tree[u].append(v)
    self.tree[v].append(u)

def _preprocess_lca(self, u, parent_node):
    """
    第一次 DFS，预处理每个节点的深度、父节点和倍增跳跃数组

    参数:
        u: 当前处理的节点
```

```
        parent_node: 当前节点的父节点

    说明：该方法通过深度优先搜索构建 LCA 算法所需的数据结构，
    包括每个节点的深度信息和倍增跳跃表。
    """
    # 设置当前节点的深度（父节点深度+1）
    self.depth[u] = self.depth[parent_node] + 1
    # 设置当前节点的父节点
    self.parent[u] = parent_node
    # 设置当前节点的直接父节点（2^0 级祖先）
    self.stjump[u][0] = parent_node

    # 预处理倍增数组
    # 利用动态规划思想：u 的 2^p 级祖先 = u 的 2^(p-1)级祖先的 2^(p-1)级祖先
    # 从 p=1 开始，直到最大层数
    for p in range(1, self.max_level + 1):
        # 确保不会越界，根节点的祖先仍为 0
        if self.stjump[u][p-1] != 0:
            self.stjump[u][p] = self.stjump[self.stjump[u][p-1]][p-1]
        else:
            # 当祖先不存在时，保持为 0
            self.stjump[u][p] = 0

    # 递归处理所有子节点
    for v in self.tree[u]:
        # 避免回到父节点，造成无限递归
        if v != parent_node:
            self._preprocess_lca(v, u)

def _find_lca(self, a, b):
    """
    使用倍增法计算两个节点的最近公共祖先

    参数:
        a: 第一个节点
        b: 第二个节点

    返回:
        a 和 b 的最近公共祖先节点编号

    算法步骤：
    1. 确保 a 的深度不小于 b
    2. 将 a 向上跳到与 b 同一深度
```

```
    3. 如果此时 a==b，则直接返回 a 作为 LCA
    4. 否则，a 和 b 同时向上跳跃，直到它们的父节点相同
    5. 返回最终的父节点作为 LCA
    """
    # 步骤 1：确保 a 的深度不小于 b
    if self.depth[a] < self.depth[b]:
        a, b = b, a

    # 步骤 2：将 a 向上跳到与 b 同一深度
    # 从最高幂次开始尝试跳跃，确保最大步长
    for p in range(self.max_level, -1, -1):
        # 只有当跳跃后的深度不小于 b 的深度时才跳跃
        if self.depth[self.stjump[a][p]] >= self.depth[b]:
            a = self.stjump[a][p]

    # 步骤 3：如果 a 和 b 相遇，说明找到了 LCA
    if a == b:
        return a

    # 步骤 4：同时向上跳跃，直到找到 LCA 的直接子节点
    for p in range(self.max_level, -1, -1):
        if self.stjump[a][p] != self.stjump[b][p]:
            a = self.stjump[a][p]
            b = self.stjump[b][p]

    # 步骤 5：返回它们的父节点作为 LCA
    return self.stjump[a][0]

def _compute_diff_accumulation(self, u, parent_node):
    """
    第二次 DFS，通过回溯累加子节点的差分标记，计算每个节点被访问的最终次数

    参数:
        u: 当前处理的节点
        parent_node: 当前节点的父节点

    说明：该方法通过后序遍历，将子节点的差分标记累加到父节点，
    是树上差分算法的关键步骤，将局部标记转化为全局计数。
    注意：该 DFS 必须在所有差分标记完成后执行。
    """
    # 步骤 1：先递归处理所有子节点
    # 采用后序遍历的方式，确保子节点的计数先计算完成
    for v in self.tree[u]:
```

```
        if v != parent_node:
            self._compute_diff_accumulation(v, u)

    # 步骤2：将子节点的访问次数累加到当前节点
    # 这一步实现了差分标记的传播和累加，使得路径上的所有节点都能被正确计数
    for v in self.tree[u]:
        if v != parent_node:
            self.diff[u] += self.diff[v]

def process_movements(self, movement_order):
    """
    处理所有移动操作，执行树上点差分

    参数:
        movement_order: 节点访问顺序列表（索引从1开始）

    说明：对于每一条移动路径u->v，执行树上点差分操作，
    通过修改四个关键点的差分值，实现路径覆盖标记。
    """
    for i in range(1, self.n):
        u = movement_order[i]      # 当前移动的起点
        v = movement_order[i + 1]  # 当前移动的终点
        lca_node = self._find_lca(u, v)     # 计算u和v的最近公共祖先
        lca_father = self.stjump[lca_node][0]  # LCA的父节点

        """
        树上点差分核心操作：
        对于路径u->v，我们希望路径上的所有节点计数加1
        通过差分技巧，我们只需要修改四个点：
        1. diff[u]++ - 在起点增加标记
        2. diff[v]++ - 在终点增加标记
        3. diff[lca_node]-- - 在LCA处抵消一次（因为u和v都会到达LCA）
        4. diff[lca_father]-- - 在LCA的父节点处抵消一次

        这种标记方式的原理是：
        - 在u和v增加标记，相当于在u到根和v到根的路径上都增加了标记
        - 在LCA和其父亲减少标记，是为了抵消重复计算的部分
        - 最终通过DFS累加，每个节点的差分值就是其在所有路径中的覆盖次数
        """
        self.diff[u] += 1
        self.diff[v] += 1
        self.diff[lca_node] -= 1
        # 根节点的父节点是0，不需要处理
```

```
            if lca_father != 0:
                self.diff[lca_father] -= 1

    def solve(self, movement_order):
        """
        解决松鼠的新家问题，获取每个节点的访问次数

        参数:
            movement_order: 节点访问顺序列表（索引从 1 开始）

        返回:
            每个节点的最终访问次数列表

        处理流程：
        1. 执行第二次 DFS，累加差分标记
        2. 处理特殊情况：除最后一个节点外，其他节点需要减 1
        """
        # 执行第二次 DFS，通过回溯累加子节点的差分标记
        self._compute_diff_accumulation(1, 0)

        # 处理特殊情况，除了最后一个节点外，其他节点的访问次数需要减 1
        # 原因：松鼠的移动是连续的，除了最后一个终点外，每个节点如果是某次移动的终点，
        # 它也会是下一次移动的起点。但在题目中，这种情况下节点只应被计数一次，而不是两次。
        result = []
        last_node = movement_order[self.n]  # 最后一个访问的节点
        for i in range(1, self.n + 1):
            if i == last_node:
                # 最后一个节点不需要减 1
                result.append(self.diff[i])
            else:
                # 其他节点需要减 1
                result.append(self.diff[i] - 1)

        return result

def main():
    """
    主函数，处理输入、算法执行和输出

    输入格式：
    第一行：节点数 n
    第二行：n 个整数，表示松鼠访问节点的顺序
    接下来 n-1 行：每行两个整数，表示树的边
```

```
输出格式：
输出 n 行，每行一个整数，表示每个节点被访问的次数
"""
# 设置递归深度限制，避免大规模数据下栈溢出
sys.setrecursionlimit(1 << 25)

# 使用 sys.stdin.read 一次性读取所有输入，提高性能
input = sys.stdin.read
tokens = input().split()
ptr = 0  # 指针，用于遍历 tokens

# 读取节点数量
n = int(tokens[ptr])
ptr += 1

# 读取访问顺序（索引从 1 开始，与题目节点编号保持一致）
movement_order = [0] * (n + 1)
for i in range(1, n + 1):
    movement_order[i] = int(tokens[ptr])
    ptr += 1

# 创建求解器实例
solver = SquirrelTreeSolver(n)

# 读取树的边，构建邻接表
for _ in range(n - 1):
    u = int(tokens[ptr])
    v = int(tokens[ptr + 1])
    solver.add_edge(u, v)
    ptr += 2

# 预处理 LCA 所需的数据（以节点 1 为根）
solver._preprocess_lca(1, 0)

# 处理所有移动操作，执行树上点差分
solver.process_movements(movement_order)

# 获取每个节点的最终访问次数
result = solver.solve(movement_order)

# 输出结果，每个节点占一行
print('\n'.join(map(str, result)))
```

```
if __name__ == '__main__':
    main()
```

==========

文件: Code09_AlienStone.cpp

==========

```
// 异象石 (LOJ 10132)
// 题目描述：
// 在一个圆上有 n 个点，按顺时针编号为 0 到 n-1。
// 有 m 次操作，每次操作会在两个点之间连一条弦。
// 每次操作后，求所有弦将圆分割成多少个区域。
// 测试链接 : https://loj.ac/problem/10132

const int MAXN = 100001;
const int LIMIT = 17;

int n, power;

// 链式前向星存储树结构
int head[MAXN], next[MAXN << 1], to[MAXN << 1], cnt;

// 深度数组和倍增跳跃数组，用于 LCA 计算
int deep[MAXN];
int stjump[MAXN][LIMIT];

// 子树大小数组
int size[MAXN];

// dfs 序数组
int dfn[MAXN];
int dfn2[MAXN];
int dfc;

// 虚树相关
int stack[MAXN];
int top;

// 被选中的点数组
int chosen[MAXN];
int chosen_cnt;
```

```
// 计算 log2(n)的函数
int log2(int n) {
    int ans = 0;
    while ((1 << ans) <= (n >> 1)) {
        ans++;
    }
    return ans;
}

// 初始化函数
void build() {
    power = log2(n);
    cnt = 1;
    for (int i = 0; i <= n; i++) {
        head[i] = 0;
        chosen[i] = 0;
    }
    dfc = 0;
    chosen_cnt = 0;
}

// 添加边的函数
void addEdge(int u, int v) {
    next[cnt] = head[u];
    to[cnt] = v;
    head[u] = cnt++;
}

// 第一次 DFS，预处理深度、dfs 序和子树大小
void dfs1(int u, int f) {
    deep[u] = deep[f] + 1;
    stjump[u][0] = f;
    dfn[u] = ++dfc;
    size[u] = 1;
    // 预处理倍增数组
    for (int p = 1; p <= power; p++) {
        stjump[u][p] = stjump[stjump[u][p - 1]][p - 1];
    }
    // 遍历所有子节点
    for (int e = head[u]; e != 0; e = next[e]) {
        if (to[e] != f) {
            dfs1(to[e], u);
            size[u] += size[to[e]];
```

```
        }
    }
    dfn2[u] = dfc;
}

// 计算最近公共祖先(LCA)
int lca(int a, int b) {
    // 确保 a 的深度不小于 b
    if (deep[a] < deep[b]) {
        int tmp = a;
        a = b;
        b = tmp;
    }
    // 将 a 向上跳到与 b 同一深度
    for (int p = power; p >= 0; p--) {
        if (deep[stjump[a][p]] >= deep[b]) {
            a = stjump[a][p];
        }
    }
    // 如果 a 和 b 在同一位置，说明 b 是 a 的祖先
    if (a == b) {
        return a;
    }
    // 同时向上跳，直到找到最近公共祖先
    for (int p = power; p >= 0; p--) {
        if (stjump[a][p] != stjump[b][p]) {
            a = stjump[a][p];
            b = stjump[b][p];
        }
    }
    return stjump[a][0];
}

// 计算两点间的距离
int dis(int a, int b) {
    int l = lca(a, b);
    return deep[a] + deep[b] - 2 * deep[l];
}
```

==========

文件: Code09_AlienStone.java

==========

```
package class122;

/**
 * 异象石 (LOJ 10132)
 *
 * 题目来源：LibreOJ
 * 题目链接：https://loj.ac/problem/10132
 *
 * 题目描述：
 * 在一个圆上有 n 个点，按顺时针编号为 0 到 n-1。
 * 有 m 次操作，每次操作会在两个点之间连一条弦。
 * 每次操作后，求所有弦将圆分割成多少个区域。
 *
 * 算法原理：虚树 + LCA
 * 这是一个结合了虚树和 LCA 的高级树上算法问题。
 *
 * 解题思路：
 * 1. 将圆上的点按照顺时针顺序构建一棵树（实际上是环的生成树）
 * 2. 使用虚树技术维护被选中的点集合
 * 3. 通过计算虚树上所有边的长度之和来确定区域数量
 * 4. 区域数量 = 边长之和/2 + 1
 *
 * 时间复杂度分析：
 * - 预处理 LCA：O(N log N)
 * - 每次操作构建虚树：O(K log N)，其中 K 是被选中点的数量
 * 总时间复杂度：O(N log N + M * K log N)
 *
 * 空间复杂度分析：
 * - 树的存储：O(N)
 * - LCA 倍增数组：O(N log N)
 * - 虚树相关数组：O(N)
 * 总空间复杂度：O(N log N)
 *
 * 工程化考量：
 * 1. 使用链式前向星存储树结构，提高空间效率
 * 2. 使用虚树技术减少不必要的计算
 * 3. 使用 StreamTokenizer 进行高效输入
 * 4. 使用 HashSet 维护被选中的点集合
 *
 * 最优解分析：
 * 虚树是解决此类动态点集问题的最优解，相比每次重新构建整个结构，
 * 虚树只关注关键点，大大提高了效率。
 */
```

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;
import java.io.PrintWriter;
import java.io.StreamTokenizer;
import java.util.Arrays;
import java.util.HashSet;
import java.util.Set;

public class Code09_AlienStone {

    /**
     * 最大节点数
     * 题目中 N 最大为 1e5，设置为 100001 以避免越界
     */
    public static int MAXN = 100001;

    /**
     * 倍增数组的大小限制
     * 2^17 = 131072 > 1e5，足够处理题目中的最大节点数
     */
    public static int LIMIT = 17;

    /**
     * 节点数量和最大幂次
     */
    public static int n, power;

    /**
     * 链式前向星存储树结构
     * head[u]: 节点 u 的第一条边的索引
     * next[e]: 边 e 的下一条边的索引
     * to[e]: 边 e 指向的节点
     * cnt: 当前可用的边索引
     */
    public static int[] head = new int[MAXN];
    public static int[] next = new int[MAXN << 1];
    public static int[] to = new int[MAXN << 1];
    public static int cnt;

    /**
```

```
 * LCA 相关数组
 * deep[u]: 节点 u 的深度
 * stjump[u][p]: 节点 u 的 2^p 级祖先
 */
public static int[] deep = new int[MAXN];
public static int[][] stjump = new int[MAXN][LIMIT];

/**
 * 子树大小数组
 * size[u]: 以节点 u 为根的子树大小
 */
public static int[] size = new int[MAXN];

/**
 * DFS 序数组
 * dfn[u]: 节点 u 的进入时间戳
 * dfn2[u]: 节点 u 的离开时间戳
 * dfc: 时间戳计数器
 */
public static int[] dfn = new int[MAXN];
public static int[] dfn2 = new int[MAXN];
public static int dfc;

/**
 * 虚树相关数组
 * stack: 虚树构建过程中的栈
 * top: 栈顶指针
 */
public static int[] stack = new int[MAXN];
public static int top;

/**
 * 被选中的点集合
 */
public static Set<Integer> chosen = new HashSet<>();

/**
 * 计算 log2(n)的整数部分
 *
 * @param n 输入整数
 * @return log2(n)的整数部分
 */
public static int log2(int n) {
```

```
    int ans = 0;
    while ((1 << ans) <= (n >> 1)) {
        ans++;
    }
    return ans;
}

/**
 * 初始化算法所需的数据结构
 * 设置数组初始值，准备处理新的测试用例
 */
public static void build() {
    power = log2(n);
    // 边索引从 1 开始，0 表示没有边
    cnt = 1;
    // 初始化链式前向星的 head 数组
    Arrays.fill(head, 1, n + 1, 0);
    // 重置时间戳计数器
    dfc = 0;
    // 清空被选中的点集合
    chosen.clear();
}

/**
 * 向链式前向星中添加一条无向边
 *
 * @param u 边的一个端点
 * @param v 边的另一个端点
 */
public static void addEdge(int u, int v) {
    // 添加 u 到 v 的边
    next[cnt] = head[u];
    to[cnt] = v;
    head[u] = cnt++;
}

/**
 * 第一次 DFS，预处理深度、DFS 序和子树大小
 *
 * @param u 当前处理的节点
 * @param f 当前节点的父节点
 */
public static void dfs1(int u, int f) {
```

```
    // 设置当前节点的深度，父节点深度+1
    deep[u] = deep[f] + 1;
    // 设置当前节点的直接父节点（2^0 级祖先）
    stjump[u][0] = f;
    // 设置进入时间戳
    dfn[u] = ++dfc;
    // 初始化子树大小
    size[u] = 1;

    // 预处理倍增数组
    // 利用动态规划思想：u 的 2^p 级祖先 = u 的 2^(p-1)级祖先的 2^(p-1)级祖先
    for (int p = 1; p <= power; p++) {
        stjump[u][p] = stjump[stjump[u][p - 1]][p - 1];
    }

    // 遍历所有子节点
    for (int e = head[u]; e != 0; e = next[e]) {
        if (to[e] != f) {
            dfs1(to[e], u);
            // 累加子节点的子树大小
            size[u] += size[to[e]];
        }
    }

    // 设置离开时间戳
    dfn2[u] = dfc;
}

/**
 * 使用倍增法计算两个节点的最近公共祖先
 *
 * @param a 第一个节点
 * @param b 第二个节点
 * @return a 和 b 的最近公共祖先
 *
 * 时间复杂度：O(log N)
 * 空间复杂度：O(1)
 */
public static int lca(int a, int b) {
    // 确保 a 的深度不小于 b
    if (deep[a] < deep[b]) {
        int tmp = a;
        a = b;
```

```
            b = tmp;
        }

        // 将 a 向上跳到与 b 同一深度
        for (int p = power; p >= 0; p--) {
            if (deep[stjump[a][p]] >= deep[b]) {
                a = stjump[a][p];
            }
        }

        // 如果 a 和 b 在同一位置，说明 b 是 a 的祖先
        if (a == b) {
            return a;
        }

        // 同时向上跳，直到找到最近公共祖先
        for (int p = power; p >= 0; p--) {
            if (stjump[a][p] != stjump[b][p]) {
                a = stjump[a][p];
                b = stjump[b][p];
            }
        }

        // 返回它们的父节点作为 LCA
        return stjump[a][0];
    }

    /**
     * 比较两个节点的 DFS 序
     *
     * @param a 第一个节点
     * @param b 第二个节点
     * @return a 的 DFS 序是否小于 b 的 DFS 序
     */
    public static boolean cmp(int a, int b) {
        return dfn[a] < dfn[b];
    }

    /**
     * 计算两点间的距离
     *
     * @param a 第一个节点
     * @param b 第二个节点
```

```
     * @return a 和 b 之间的距离
     */
    public static int dis(int a, int b) {
        // 计算 a 和 b 的 LCA
        int l = lca(a, b);
        // 距离 = depth[a] + depth[b] - 2 * depth[lca]
        return deep[a] + deep[b] - 2 * deep[l];
    }

    /**
     * 主函数，处理输入输出并调用相应的算法函数
     *
     * @param args 命令行参数
     * @throws IOException 输入输出异常
     */
    public static void main(String[] args) throws IOException {
        // 使用高效的输入方式
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        StreamTokenizer in = new StreamTokenizer(br);
        // 使用高效的输出方式
        PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));

        // 读取节点数量
        in.nextToken();
        n = (int) in.nval;

        // 初始化数据结构
        build();

        // 读取树的边
        for (int i = 1, u, v; i < n; i++) {
            in.nextToken();
            u = (int) in.nval;
            in.nextToken();
            v = (int) in.nval;
            // 添加无向边
            addEdge(u, v);
            addEdge(v, u);
        }

        // 预处理深度、DFS 序和子树大小
        dfs1(1, 0);
```

```
// 读取操作数量
in.nextToken();
int m = (int) in.nval;

// 处理每次操作
for (int i = 1; i <= m; i++) {
    in.nextToken();
    String op = in.sval;

    if (op.equals("+")) {
        // 添加点操作
        in.nextToken();
        int x = (int) in.nval;
        chosen.add(x);
    } else if (op.equals("-")) {
        // 删除点操作
        in.nextToken();
        int x = (int) in.nval;
        chosen.remove(x);
    } else { // op.equals("?")
        // 查询操作
        if (chosen.size() <= 1) {
            // 特殊情况：选中点数小于等于 1
            out.println(chosen.size());
        } else {
            /**
             * 构建虚树并计算答案
             * 虚树是原树的一个简化版本，只包含关键点和它们的 LCA
             * 区域数量 = 虚树上所有边的长度之和/2 + 1
             */
            // 获取所有被选中的点
            int[] nodes = chosen.stream().mapToInt(Integer::intValue).toArray();
            // 按节点编号排序
            Arrays.sort(nodes);

            // 按 DFS 序排序
            Integer[] sorted = new Integer[nodes.length];
            for (int j = 0; j < nodes.length; j++) {
                sorted[j] = nodes[j];
            }
            Arrays.sort(sorted, (a, b) -> dfn[a] - dfn[b]);

            // 构建虚树
```

```
stack[0] = 1;  // 根节点入栈
top = 1;       // 栈顶指针
long ans = 0;  // 边长之和

// 遍历所有排序后的节点
for (int j = 0; j < sorted.length; j++) {
    int u = sorted[j];
    if (top == 1) {
        // 栈中只有一个元素，直接入栈
        stack[top++] = u;
    } else {
        // 计算栈顶元素和当前节点的LCA
        int l = lca(stack[top - 1], u);
        // 维护虚树结构
        while (top > 1 && deep[stack[top - 1]] > deep[l]) {
            if (deep[stack[top - 2]] <= deep[l]) {
                // 栈顶元素到LCA的距离加入答案
                ans += dis(stack[top - 1], l);
                // 更新栈顶元素为LCA
                stack[top - 1] = l;
                break;
            } else {
                // 栈顶元素到其父节点的距离加入答案
                ans += dis(stack[top - 1], stack[top - 2]);
                // 出栈
                top--;
            }
        }
        // 如果当前节点不是栈顶元素，则入栈
        if (stack[top - 1] != u) {
            stack[top++] = u;
        }
    }
}

// 处理栈中剩余元素
while (top > 1) {
    // 栈顶元素到其父节点的距离加入答案
    ans += dis(stack[top - 1], stack[top - 2]);
    // 出栈
    top--;
}
```

```
                    // 输出结果：区域数量 = 边长之和/2 + 1
                    out.println(ans / 2 + 1);
                }
            }
        }

        // 确保输出被刷新
        out.flush();
        // 关闭资源
        out.close();
        br.close();
    }
}
```

==========

文件: Code09_AlienStone.py

==========

```
# 异象石 (LOJ 10132)
# 题目描述：
# 在一个圆上有 n 个点，按顺时针编号为 0 到 n-1。
# 有 m 次操作，每次操作会在两个点之间连一条弦。
# 每次操作后，求所有弦将圆分割成多少个区域。
# 测试链接 : https://loj.ac/problem/10132

import sys
from collections import defaultdict

# 读取输入
input = sys.stdin.read
tokens = input().split()

# 读取节点数量
n = int(tokens[0])

# 读取树的边
edges = []
idx = 1
for i in range(n - 1):
    u = int(tokens[idx])
    v = int(tokens[idx + 1])
    edges.append((u, v))
    idx += 2
```

```
# 构建邻接表表示的树
tree = defaultdict(list)
for u, v in edges:
    tree[u].append(v)
    tree[v].append(u)

# 深度数组和父节点数组
deep = [0] * (n + 1)
parent = [0] * (n + 1)

# 倍增数组的大小限制
LIMIT = 17
# 倍增跳跃数组，用于 LCA 计算
stjump = [[0] * LIMIT for _ in range(n + 1)]

# dfs 序数组
dfn = [0] * (n + 1)
dfn2 = [0] * (n + 1)
dfc = 0

# 子树大小数组
size = [0] * (n + 1)

# 被选中的点集合
chosen = set()

# 计算 log2(n)的函数
def log2(n):
    ans = 0
    while (1 << ans) <= (n >> 1):
        ans += 1
    return ans

power = log2(n)

# 第一次 DFS，预处理深度、dfs 序和子树大小
def dfs1(u, f):
    global dfc
    deep[u] = deep[f] + 1
    parent[u] = f
    stjump[u][0] = f
    dfc += 1
```

```
    dfn[u] = dfc
    size[u] = 1
    # 预处理倍增数组
    for p in range(1, power + 1):
        stjump[u][p] = stjump[stjump[u][p - 1]][p - 1]
    # 遍历所有子节点
    for v in tree[u]:
        if v != f:
            dfs1(v, u)
            size[u] += size[v]
    dfn2[u] = dfc

# 计算最近公共祖先(LCA)
def lca(a, b):
    # 确保 a 的深度不小于 b
    if deep[a] < deep[b]:
        a, b = b, a
    # 将 a 向上跳到与 b 同一深度
    for p in range(power, -1, -1):
        if deep[stjump[a][p]] >= deep[b]:
            a = stjump[a][p]
    # 如果 a 和 b 在同一位置，说明 b 是 a 的祖先
    if a == b:
        return a
    # 同时向上跳，直到找到最近公共祖先
    for p in range(power, -1, -1):
        if stjump[a][p] != stjump[b][p]:
            a = stjump[a][p]
            b = stjump[b][p]
    return stjump[a][0]

# 计算两点间的距离
def dis(a, b):
    l = lca(a, b)
    return deep[a] + deep[b] - 2 * deep[l]

# 预处理深度、dfs 序和子树大小
dfs1(1, 0)

# 读取操作数量
m = int(tokens[idx])
idx += 1
```

```
# 处理每次操作
for i in range(m):
    op = tokens[idx]
    idx += 1

    if op == "+":
        x = int(tokens[idx])
        idx += 1
        chosen.add(x)
    elif op == "-":
        x = int(tokens[idx])
        idx += 1
        chosen.discard(x)
    else:  # op == "?"
        if len(chosen) <= 1:
            print(len(chosen))
        else:
            # 构建虚树并计算答案
            nodes = list(chosen)
            nodes.sort(key=lambda x: dfn[x])

            # 构建虚树
            stack = [1]
            ans = 0

            for u in nodes:
                if len(stack) == 1:
                    stack.append(u)
                else:
                    l = lca(stack[-1], u)
                    while len(stack) > 1 and deep[stack[-1]] > deep[l]:
                        if deep[stack[-2]] <= deep[l]:
                            ans += dis(stack[-1], l)
                            stack[-1] = l
                            break
                        else:
                            ans += dis(stack[-1], stack[-2])
                            stack.pop()
                    if stack[-1] != u:
                        stack.append(u)

            while len(stack) > 1:
                ans += dis(stack[-1], stack[-2])
```

```
                stack.pop()

            print(ans // 2 + 1)
```

==========

文件: Code10_AlyonaAndTree.cpp

==========

```
/*
 * Alyona and a tree (Codeforces 739B)
 * 算法：倍增法 + 二分查找 + 树上差分
 *
 * 【算法原理】
 * 1. 对于每个节点 v，我们需要找到最远的祖先 u，使得 dist(u,v) <= a[v]
 * 2. 这意味着所有从 u 到 v 路径上的节点都是满足条件的祖先
 * 3. 使用树上差分标记这个区间：diff[v]++, diff[parent(u)]--
 * 4. 最后通过 DFS 回溯累加差分数组，得到每个节点的答案
 *
 * 【复杂度分析】
 * - 时间复杂度：O(N log N)
 *   预处理倍增数组: O(N log N)
 *   二分查找每个节点: O(N log N)
 *   DFS 回溯统计: O(N)
 * - 空间复杂度：O(N log N)
 *   链式前向星存储树: O(N)
 *   倍增数组 stjump 和 stsum: O(N log N)
 *
 * 【工程化考量】
 * 1. 链式前向星作为高效的图存储结构，适合树结构的实现
 * 2. 变量命名：使用 next_edge 而非 next，避免与 C++关键字冲突
 * 3. 使用 long long 类型存储距离，避免溢出
 * 4. 边界处理：根节点(1)的特殊处理
 *
 * 【最优解分析】
 * 此解法是本题的最优解，时间复杂度为 O(N log N)
 * 暴力枚举每个节点的所有祖先需要 O(N^2)时间，远不如本解法高效
 */

#include <iostream>
#include <cstdio>
using namespace std;

const int MAXN = 200001; // 最大节点数
```

```
int n;                  // 节点数量
int a[MAXN];            // 节点权值数组

// 链式前向星存储树结构
int head[MAXN];         // 邻接表头节点数组
int next_edge[MAXN << 1]; // 下一条边的索引（避免与C++关键字冲突）
int to[MAXN << 1];      // 边的目标节点
int weight[MAXN << 1];  // 边的权值
int cnt;                // 边计数器

// 深度数组和到根节点的距离数组
int deep[MAXN];         // 节点深度数组
long long dist[MAXN];   // 节点到根节点的距离数组（使用long long避免溢出）

// 倍增数组相关
const int LIMIT = 18;   // 倍增数组的大小限制，2^18足够处理2e5规模的树
int stjump[MAXN][LIMIT]; // 倍增跳跃数组，stjump[u][p]表示u的2^p级祖先
long long stsum[MAXN][LIMIT]; // 倍增距离数组，记录跳跃的路径距离和

// 差分数组和DFS序相关
int diff[MAXN];         // 差分数组，用于标记区间修改
int dfn[MAXN];          // 节点的进入时间戳
int dfn2[MAXN];         // 节点的离开时间戳
int dfc;                // 时间戳计数器

/**
 * 初始化数据结构
 */
void build() {
    cnt = 1;            // 边计数器从1开始，方便链式前向星操作
    for (int i = 0; i <= n; i++) {
        head[i] = 0;    // 初始化邻接表头数组
    }
    dfc = 0;            // 重置时间戳计数器
    for (int i = 1; i <= n; i++) {
        diff[i] = 0;    // 初始化差分数组
    }
}

/**
 * 添加边到树中（无向边，需要添加双向）
 * @param u 边的起点
 * @param v 边的终点
```

```
 * @param w 边的权值
 */
void addEdge(int u, int v, int w) {
    next_edge[cnt] = head[u]; // 当前边的 next 指向当前 head[u]
    to[cnt] = v;              // 当前边的目标节点是 v
    weight[cnt] = w;          // 当前边的权值是 w
    head[u] = cnt++;          // 更新 head[u]为当前边的索引，并递增计数器
}

/**
 * 第一次 DFS：预处理深度、距离和倍增数组
 * @param u 当前节点
 * @param f 父节点
 * @param d 当前节点到根节点的距离
 */
void dfs1(int u, int f, long long d) {
    deep[u] = deep[f] + 1;   // 设置当前节点深度
    dist[u] = d;             // 设置当前节点到根节点的距离
    dfn[u] = ++dfc;          // 设置进入时间戳
    stjump[u][0] = f;        // 设置直接父节点
    stsum[u][0] = d - dist[f]; // 设置到父节点的距离

    // 预处理倍增数组
    for (int p = 1; p < LIMIT; p++) {
        stjump[u][p] = stjump[stjump[u][p - 1]][p - 1];
        stsum[u][p] = stsum[u][p - 1] + stsum[stjump[u][p - 1]][p - 1];
    }

    // 递归处理所有子节点
    for (int e = head[u]; e != 0; e = next_edge[e]) {
        if (to[e] != f) {   // 避免回到父节点
            dfs1(to[e], u, d + weight[e]);
        }
    }

    dfn2[u] = dfc;           // 设置离开时间戳
}

/**
 * 二分查找满足条件的最远祖先
 * 找到最远的祖先 u，使得 dist(u,v) <= a[v]
 *
 * @param v 当前节点
```

```
 * @return 最远的满足条件的祖先节点编号
 */
int findFarthestAncestor(int v) {
    // 计算 v 节点的最远允许距离边界
    long long need = dist[v] - a[v];

    // 如果允许距离超过根节点，直接返回根节点
    if (need < 0) {
        return 1; // 根节点
    }

    int u = v; // 从 v 开始向上查找

    // 【倍增法向上查找】
    // 从最高幂次开始，尽可能向上跳，但保持祖先的距离 > need
    for (int p = LIMIT - 1; p >= 0; p--) {
        // 确保祖先存在且其距离仍大于 need
        if (stjump[u][p] > 0 && dist[stjump[u][p]] > need) {
            u = stjump[u][p];
        }
    }

    // 最终确定最远的满足条件的祖先
    int ans;
    if (dist[u] > need) {
        ans = stjump[u][0]; // 如果当前节点距离仍大于 need，再向上跳一级
    } else {
        ans = u; // 当前节点距离已满足条件
    }

    return ans;
}

/**
 * 第二次 DFS：回溯计算差分数组的累加结果
 * 将子节点的差分值累加到父节点，得到每个节点的最终答案
 *
 * @param u 当前节点
 * @param f 父节点
 */
void dfs2(int u, int f) {
    // 递归处理所有子节点
    for (int e = head[u]; e != 0; e = next_edge[e]) {
```

```
        int v = to[e];
        if (v != f) {   // 避免回到父节点
            dfs2(v, u); // 后序遍历，先处理子节点
            diff[u] += diff[v]; // 累加子节点的差分值
        }
    }
}

int main() {
    // 读取节点数量
    scanf("%d", &n);

    // 初始化数据结构
    build();

    // 读取每个节点的权值
    for (int i = 1; i <= n; i++) {
        scanf("%d", &a[i]);
    }

    // 读取树的边（题目中树是有根的，父节点编号小于子节点）
    for (int i = 2, p, w; i <= n; i++) {
        scanf("%d %d", &p, &w); // 读取父节点和边权
        addEdge(p, i, w);         // 添加正向边
        addEdge(i, p, w);         // 添加反向边（无向图）
    }

    // 预处理深度、距离和倍增数组
    dfs1(1, 0, 0); // 根节点为1，父节点为0（无效节点），距离为0

    // 【树上差分标记】
    // 对每个节点v，找到满足条件的最远祖先u，并标记区间[u, v]
    for (int v = 1; v <= n; v++) {
        int u = findFarthestAncestor(v); // 找到最远的满足条件的祖先

        // 执行区间标记：所有从u到v的祖先都满足条件
        // 这等价于：在v处+1，在u的父节点处-1（如果u不是根节点）
        if (u != 1) { // 如果u不是根节点
            diff[stjump[u][0]]--; // u的父节点减1，结束区间
            diff[v]++;            // v处加1，开始区间
        } else { // 如果u是根节点
            diff[v]++;            // 直接在v处加1，区间从根节点开始
        }
```

```
    }

    // 计算差分数组的累加结果，得到每个节点的最终答案
    dfs2(1, 0);

    // 输出结果，每个节点的满足条件的祖先数量
    for (int i = 1; i <= n; i++) {
        printf("%d ", diff[i]);
    }
    printf("\n");

    return 0;
}
```

==========

文件: Code10_AlyonaAndTree.java

==========

```
package class122;

/*
 * Alyona and a tree (Codeforces 739B)
 * 算法：倍增法 + 二分查找 + 树上差分
 *
 * 【算法原理】
 * 1. 对于每个节点 v，我们需要找到最远的祖先 u，使得 dist(u,v) <= a[v]
 * 2. 这意味着所有从 u 到 v 路径上的节点都是满足条件的祖先
 * 3. 使用树上差分标记这个区间：diff[v]++, diff[parent(u)]--
 * 4. 最后通过 DFS 回溯累加差分数组，得到每个节点的答案
 *
 * 【复杂度分析】
 * - 时间复杂度：O(N log N)
 *   预处理倍增数组: O(N log N)
 *   二分查找每个节点: O(N log N)
 *   DFS 回溯统计: O(N)
 * - 空间复杂度：O(N log N)
 *   链式前向星存储树: O(N)
 *   倍增数组 stjump 和 stsum: O(N log N)
 *
 * 【工程化考量】
 * 1. 链式前向星作为高效的图存储结构，适合树结构的实现
 * 2. 变量命名：使用 next_edge 而非 next，避免与 Java 关键字冲突
 * 3. 输入输出效率：使用 BufferedReader 和 PrintWriter 保证大数据量下的性能
```

```
 * 4. 边界处理：根节点(1)的特殊处理，以及距离计算的溢出问题
 *
 * 【最优解分析】
 * 此解法是本题的最优解，时间复杂度为O(N log N)
 * 暴力枚举每个节点的所有祖先需要O(N^2)时间，远不如本解法高效
 */

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;
import java.io.PrintWriter;
import java.io.StreamTokenizer;
import java.util.Arrays;

public class Code10_AlyonaAndTree {

    // 最大节点数（根据题目约束设置）
    public static int MAXN = 200001;

    // 全局变量定义
    public static int n;                    // 节点数量
    public static int[] a = new int[MAXN]; // 节点权值数组，a[v]表示节点 v 的最大距离限制

    // 链式前向星存储树结构
    public static int[] head = new int[MAXN];           // 邻接表头节点数组
    public static int[] next_edge = new int[MAXN << 1]; // 下一条边的索引（避免与 Java 关键字冲突）
    public static int[] to = new int[MAXN << 1];        // 边的目标节点
    public static int[] weight = new int[MAXN << 1];    // 边的权值
    public static int cnt;                              // 边计数器

    // 深度数组和到根节点的距离数组
    public static int[] deep = new int[MAXN];           // 节点深度数组
    public static long[] dist = new long[MAXN];         // 节点到根节点的距离数组（使用 long 避免溢出）

    // 倍增数组相关
    public static int LIMIT = 18;                       // 倍增数组的大小限制，2^18 足够处理 2e5 规模的树
    public static int[][] stjump = new int[MAXN][LIMIT]; // 倍增跳跃数组，stjump[u][p]表示 u 的 2^p 级祖先
    public static long[][] stsum = new long[MAXN][LIMIT]; // 倍增距离数组，记录跳跃的路径距离和
```

```
// 差分数组和DFS序相关
public static int[] diff = new int[MAXN];           // 差分数组，用于标记区间修改
public static int[] dfn = new int[MAXN];            // 节点的进入时间戳
public static int[] dfn2 = new int[MAXN];           // 节点的离开时间戳
public static int dfc;                              // 时间戳计数器

/**
 * 初始化数据结构
 */
public static void build() {
    cnt = 1;                            // 边计数器从1开始，方便链式前向星操作
    Arrays.fill(head, 1, n + 1, 0);     // 初始化邻接表头数组
    dfc = 0;                            // 重置时间戳计数器
    Arrays.fill(diff, 1, n + 1, 0);     // 初始化差分数组
}

/**
 * 添加边到树中（无向边，需要添加双向）
 * @param u 边的起点
 * @param v 边的终点
 * @param w 边的权值
 */
public static void addEdge(int u, int v, int w) {
    next_edge[cnt] = head[u];  // 当前边的next指向当前head[u]
    to[cnt] = v;               // 当前边的目标节点是v
    weight[cnt] = w;           // 当前边的权值是w
    head[u] = cnt++;           // 更新head[u]为当前边的索引，并递增计数器
}

/**
 * 第一次DFS：预处理深度、距离和倍增数组
 * @param u 当前节点
 * @param f 父节点
 * @param d 当前节点到根节点的距离
 */
public static void dfs1(int u, int f, long d) {
    deep[u] = deep[f] + 1;              // 设置当前节点深度
    dist[u] = d;                        // 设置当前节点到根节点的距离
    dfn[u] = ++dfc;                     // 设置进入时间戳
    stjump[u][0] = f;                   // 设置直接父节点
    stsum[u][0] = d - dist[f];          // 设置到父节点的距离
```

```
    // 预处理倍增数组
    for (int p = 1; p < LIMIT; p++) {
        stjump[u][p] = stjump[stjump[u][p - 1]][p - 1];
        stsum[u][p] = stsum[u][p - 1] + stsum[stjump[u][p - 1]][p - 1];
    }

    // 递归处理所有子节点
    for (int e = head[u]; e != 0; e = next_edge[e]) {
        if (to[e] != f) {  // 避免回到父节点
            dfs1(to[e], u, d + weight[e]);
        }
    }

    dfn2[u] = dfc;  // 设置离开时间戳
}

/**
 * 二分查找满足条件的最远祖先
 * 找到最远的祖先u，使得 dist(u,v) <= a[v]
 *
 * @param v 当前节点
 * @return 最远的满足条件的祖先节点编号
 */
public static int find(int v) {
    // 计算 v 节点的最远允许距离边界
    long need = dist[v] - a[v];

    // 如果允许距离超过根节点，直接返回根节点
    if (need < 0) {
        return 1; // 根节点
    }

    int u = v; // 从 v 开始向上查找

    // 【倍增法向上查找】
    // 从最高幂次开始，尽可能向上跳，但保持祖先的距离 > need
    for (int p = LIMIT - 1; p >= 0; p--) {
        // 确保祖先存在且其距离仍大于 need
        if (stjump[u][p] > 0 && dist[stjump[u][p]] > need) {
            u = stjump[u][p];
        }
    }
```

```
    // 最终确定最远的满足条件的祖先
    int ans;
    if (dist[u] > need) {
        ans = stjump[u][0]; // 如果当前节点距离仍大于 need，再向上跳一级
    } else {
        ans = u; // 当前节点距离已满足条件
    }

    return ans;
}

/**
 * 第二次 DFS：回溯计算差分数组的累加结果
 * 将子节点的差分值累加到父节点，得到每个节点的最终答案
 *
 * @param u 当前节点
 * @param f 父节点
 */
public static void dfs2(int u, int f) {
    // 递归处理所有子节点
    for (int e = head[u]; e != 0; e = next_edge[e]) {
        int v = to[e];
        if (v != f) {  // 避免回到父节点
            dfs2(v, u);  // 后序遍历，先处理子节点
            diff[u] += diff[v];  // 累加子节点的差分值
        }
    }
}

public static void main(String[] args) throws IOException {
    // 输入输出优化，使用快速 IO
    BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
    StreamTokenizer in = new StreamTokenizer(br);
    PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));

    // 读取节点数量
    in.nextToken();
    n = (int) in.nval;

    // 初始化数据结构
    build();

    // 读取每个节点的权值
```

```
for (int i = 1; i <= n; i++) {
    in.nextToken();
    a[i] = (int) in.nval;
}

// 读取树的边（题目中树是有根的，父节点编号小于子节点）
for (int i = 2, p, w; i <= n; i++) {
    in.nextToken();
    p = (int) in.nval;  // 父节点
    in.nextToken();
    w = (int) in.nval;  // 边权
    addEdge(p, i, w);   // 添加正向边
    addEdge(i, p, w);   // 添加反向边（无向图）
}

// 预处理深度、距离和倍增数组
dfs1(1, 0, 0);  // 根节点为1，父节点为0（无效节点），距离为0

// 【树上差分标记】
// 对每个节点v，找到满足条件的最远祖先u，并标记区间[u, v]
for (int v = 1; v <= n; v++) {
    int u = find(v);  // 找到最远的满足条件的祖先

    // 执行区间标记：所有从u到v的祖先都满足条件
    // 这等价于：在v处+1，在u的父节点处-1（如果u不是根节点）
    if (u != 1) {  // 如果u不是根节点
        diff[stjump[u][0]]--;  // u的父节点减1，结束区间
        diff[v]++;             // v处加1，开始区间
    } else {  // 如果u是根节点
        diff[v]++;             // 直接在v处加1，区间从根节点开始
    }
}

// 计算差分数组的累加结果，得到每个节点的最终答案
dfs2(1, 0);

// 输出结果，每个节点的满足条件的祖先数量
for (int i = 1; i <= n; i++) {
    out.print(diff[i] + " ");
}
out.println();

// 刷新输出流并关闭资源
```

```
        out.flush();
        out.close();
        br.close();
    }
}
```

==========

文件: Code10_AlyonaAndTree.py

==========

"""

Alyona and a tree (Codeforces 739B)

算法：倍增法 + 二分查找 + 树上差分

【算法原理】

1. 对于每个节点 v，我们需要找到最远的祖先 u，使得 dist(u,v) <= a[v]
2. 这意味着所有从 u 到 v 路径上的节点都是满足条件的祖先
3. 使用树上差分标记这个区间：diff[v]++, diff[parent(u)]--
4. 最后通过 DFS 回溯累加差分数组，得到每个节点的答案

【复杂度分析】

- 时间复杂度：O(N log N)
  预处理倍增数组: O(N log N)
  二分查找每个节点: O(N log N)
  DFS 回溯统计: O(N)
- 空间复杂度：O(N log N)
  邻接表存储树: O(N)
  倍增数组 stjump 和 stsum: O(N log N)

【工程化考量】

1. 使用邻接表作为高效的树存储结构
2. Python 中使用 defaultdict 简化邻接表的构建
3. 输入优化：使用 sys.stdin.read 一次性读取所有输入数据
4. 边界处理：根节点(1)的特殊处理

【最优解分析】

此解法是本题的最优解，时间复杂度为 O(N log N)

暴力枚举每个节点的所有祖先需要 O(N^2)时间，远不如本解法高效

"""

```
import sys
from collections import defaultdict
```

```
class AlyonaTreeSolver:
    """
    Alyona and a tree 问题求解器
    使用倍增法、二分查找和树上差分算法解决问题
    """

    def __init__(self, n, a_values, edges):
        """
        初始化求解器

        Args:
            n: 节点数量
            a_values: 每个节点的权值数组
            edges: 树的边列表，每个元素为(p, v, w)表示父节点、子节点和边权
        """
        self.n = n
        self.a = a_values  # 节点权值数组
        self.LIMIT = 18  # 倍增数组的大小限制，2^18 足够处理 2e5 规模的树

        # 构建邻接表表示的树
        self.tree = defaultdict(list)
        for p, v, w in edges:
            self.tree[p].append((v, w))
            self.tree[v].append((p, w))

        # 初始化数据结构
        self.deep = [0] * (n + 1)  # 节点深度数组
        self.dist = [0] * (n + 1)  # 节点到根节点的距离数组
        self.stjump = [[0] * self.LIMIT for _ in range(n + 1)]  # 倍增跳跃数组
        self.stsum = [[0] * self.LIMIT for _ in range(n + 1)]  # 倍增距离和数组
        self.diff = [0] * (n + 1)  # 差分数组
        self.dfn = [0] * (n + 1)  # 进入时间戳
        self.dfn2 = [0] * (n + 1)  # 离开时间戳
        self.dfc = 0  # 时间戳计数器

    def dfs1(self, u, f, d):
        """
        第一次 DFS：预处理深度、距离和倍增数组

        Args:
            u: 当前节点
            f: 父节点
            d: 当前节点到根节点的距离
```

```
    """
    self.deep[u] = self.deep[f] + 1  # 设置当前节点深度
    self.dist[u] = d  # 设置当前节点到根节点的距离
    self.dfc += 1  # 增加时间戳
    self.dfn[u] = self.dfc  # 设置进入时间戳
    self.stjump[u][0] = f  # 设置直接父节点
    self.stsum[u][0] = d - self.dist[f]  # 设置到父节点的距离

    # 预处理倍增数组
    for p in range(1, self.LIMIT):
        self.stjump[u][p] = self.stjump[self.stjump[u][p-1]][p-1]
        self.stsum[u][p] = self.stsum[u][p-1] + self.stsum[self.stjump[u][p-1]][p-1]

    # 递归处理所有子节点
    for v, w in self.tree[u]:
        if v != f:  # 避免回到父节点
            self.dfs1(v, u, d + w)

    self.dfn2[u] = self.dfc  # 设置离开时间戳

def find_farthest_ancestor(self, v):
    """
    二分查找满足条件的最远祖先
    找到最远的祖先u，使得 dist(u,v) <= a[v]

    Args:
        v: 当前节点

    Returns:
        最远的满足条件的祖先节点编号
    """
    # 计算 v 节点的最远允许距离边界
    need = self.dist[v] - self.a[v]

    # 如果允许距离超过根节点，直接返回根节点
    if need < 0:
        return 1  # 根节点

    u = v  # 从 v 开始向上查找

    # 【倍增法向上查找】
    # 从最高幂次开始，尽可能向上跳，但保持祖先的距离 > need
    for p in range(self.LIMIT - 1, -1, -1):
```

```
            # 确保祖先存在且其距离仍大于 need
            if self.stjump[u][p] > 0 and self.dist[self.stjump[u][p]] > need:
                u = self.stjump[u][p]

        # 最终确定最远的满足条件的祖先
        if self.dist[u] > need:
            return self.stjump[u][0]  # 如果当前节点距离仍大于 need，再向上跳一级
        else:
            return u  # 当前节点距离已满足条件

    def dfs2(self, u, f):
        """
        第二次 DFS：回溯计算差分数组的累加结果
        将子节点的差分值累加到父节点，得到每个节点的最终答案

        Args:
            u: 当前节点
            f: 父节点
        """
        # 递归处理所有子节点
        for v, w in self.tree[u]:
            if v != f:  # 避免回到父节点
                self.dfs2(v, u)  # 后序遍历，先处理子节点
                self.diff[u] += self.diff[v]  # 累加子节点的差分值

    def solve(self):
        """
        解决问题的主函数

        Returns:
            每个节点满足条件的祖先数量数组
        """
        # 预处理深度、距离和倍增数组
        self.dfs1(1, 0, 0)  # 根节点为 1，父节点为 0（无效节点），距离为 0

        # 【树上差分标记】
        # 对每个节点 v，找到满足条件的最远祖先 u，并标记区间[u, v]
        for v in range(1, self.n + 1):
            u = self.find_farthest_ancestor(v)  # 找到最远的满足条件的祖先

            # 执行区间标记：所有从 u 到 v 的祖先都满足条件
            # 这等价于：在 v 处+1，在 u 的父节点处-1（如果 u 不是根节点）
            if u != 1:  # 如果 u 不是根节点
```

```
                self.diff[self.stjump[u][0]] -= 1  # u 的父节点减 1，结束区间
                self.diff[v] += 1  # v 处加 1，开始区间
            else:  # 如果 u 是根节点
                self.diff[v] += 1  # 直接在 v 处加 1，区间从根节点开始

        # 计算差分数组的累加结果，得到每个节点的最终答案
        self.dfs2(1, 0)

        return self.diff[1:self.n + 1]  # 返回 1~n 的结果

def main():
    """
    主函数：读取输入数据，调用求解器，输出结果
    """
    # 输入优化：一次性读取所有输入数据
    input_data = sys.stdin.read()
    tokens = input_data.split()
    ptr = 0

    # 读取节点数量
    n = int(tokens[ptr])
    ptr += 1

    # 读取每个节点的权值
    a = [0] * (n + 1)  # a[0]不使用
    for i in range(1, n + 1):
        a[i] = int(tokens[ptr])
        ptr += 1

    # 读取树的边（题目中树是有根的，父节点编号小于子节点）
    edges = []
    for i in range(2, n + 1):  # 节点 2~n，每个节点有一个父节点
        p = int(tokens[ptr])  # 父节点
        w = int(tokens[ptr + 1])  # 边权
        edges.append((p, i, w))  # (父节点, 子节点, 边权)
        ptr += 2

    # 创建求解器并求解
    solver = AlyonaTreeSolver(n, a, edges)
    result = solver.solve()

    # 输出结果，每个节点的满足条件的祖先数量
    print(' '.join(map(str, result)))
```

```
if __name__ == "__main__":
    main()
```

==========

文件: Code11_AtCoderABC187E.cpp

==========

```
/**
 * AtCoder ABC187 E - Through Path（树上差分）
 * 题目链接：https://atcoder.jp/contests/abc187/tasks/abc187_e
 * 题目描述：给定一棵树，有Q次操作，每次操作有两种类型：
 *   1. 选择一条边(a,b)，给所有从a出发不经过b能到达的节点加上x
 *   2. 选择一条边(a,b)，给所有从b出发不经过a能到达的节点加上x
 * 所有操作完成后，输出每个节点的值
 * 解法：树上差分 + DFS
 *
 * 算法思路：
 * 1. 对于每条边(a,b)，将树分为两个连通分量
 * 2. 操作1：给a所在的连通分量（不包含b）的所有节点加上x
 * 3. 操作2：给b所在的连通分量（不包含a）的所有节点加上x
 * 4. 使用树上差分技术，在根节点处打标记，通过DFS计算子树和
 *
 * 时间复杂度：O(N + Q)
 * 空间复杂度：O(N)
 */

#include <iostream>
#include <vector>
#include <cstring>
using namespace std;

const int MAXN = 200001;

// 邻接表存储树
vector<int> graph[MAXN];

// 存储边的信息
int edges[MAXN][2];

// 差分数组
long long diff[MAXN];
```

```
// DFS 相关
int parent[MAXN];
int depth[MAXN];
int size[MAXN];

// DFS 预处理树结构
void dfs(int u, int fa) {
    parent[u] = fa;
    depth[u] = depth[fa] + 1;
    size[u] = 1;
    for (int v : graph[u]) {
        if (v != fa) {
            dfs(v, u);
            size[u] += size[v];
        }
    }
}

// DFS 计算子树和
void dfsCalc(int u, int fa) {
    for (int v : graph[u]) {
        if (v != fa) {
            diff[v] += diff[u];
            dfsCalc(v, u);
        }
    }
}

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);

    int n;
    cin >> n;

    // 读入边
    for (int i = 1; i < n; i++) {
        int a, b;
        cin >> a >> b;
        edges[i][0] = a;
        edges[i][1] = b;
        graph[a].push_back(b);
        graph[b].push_back(a);
```

```
}

// 以 1 为根节点进行 DFS
depth[0] = -1;
dfs(1, 0);

int q;
cin >> q;

for (int i = 0; i < q; i++) {
    int t, e;
    long long x;
    cin >> t >> e >> x;

    int a = edges[e][0];
    int b = edges[e][1];

    // 确保 a 是 b 的父节点
    if (depth[a] > depth[b]) {
        swap(a, b);
    }

    if (t == 1) {
        // 操作 1：给 a 所在的连通分量（不包含 b）加上 x
        // 相当于给整棵树加上 x，然后给 b 的子树减去 x
        diff[1] += x;
        diff[b] -= x;
    } else {
        // 操作 2：给 b 所在的连通分量（不包含 a）加上 x
        // 相当于给 b 的子树加上 x
        diff[b] += x;
    }
}

// DFS 计算最终结果
dfsCalc(1, 0);

// 输出结果
for (int i = 1; i <= n; i++) {
    cout << diff[i] << "\n";
}

return 0;
```

```
}

==========================================

文件: Code11_AtCoderABC187E.java
==========================================
package class122;

/**
 * AtCoder ABC187 E - Through Path（树上差分）
 *
 * 题目来源：AtCoder
 * 题目链接：https://atcoder.jp/contests/abc187/tasks/abc187_e
 *
 * 题目描述：
 * 给定一棵树，有 Q 次操作，每次操作有两种类型：
 * 1. 选择一条边(a,b)，给所有从 a 出发不经过 b 能到达的节点加上 x
 * 2. 选择一条边(a,b)，给所有从 b 出发不经过 a 能到达的节点加上 x
 * 所有操作完成后，输出每个节点的值
 *
 * 算法原理：树上差分 + DFS
 * 树上差分是处理树上区间操作的一种高效技术。
 * 对于每条边(a,b)，它将树分为两个连通分量。
 * 通过差分技术，我们可以在根节点和特定节点打标记，然后通过 DFS 计算子树和得到最终结果。
 *
 * 解题思路：
 * 1. 对于每条边(a,b)，将树分为两个连通分量
 * 2. 操作 1：给 a 所在的连通分量（不包含 b）的所有节点加上 x
 * 3. 操作 2：给 b 所在的连通分量（不包含 a）的所有节点加上 x
 * 4. 使用树上差分技术，在根节点处打标记，通过 DFS 计算子树和
 *
 * 时间复杂度分析：
 * - 建图和预处理：O(N)
 * - 处理操作：O(Q)
 * - DFS 计算结果：O(N)
 * 总时间复杂度：O(N + Q)
 *
 * 空间复杂度分析：
 * - 树的存储：O(N)
 * - 边信息存储：O(N)
 * - 差分数组：O(N)
 * - DFS 相关信息：O(N)
 * 总空间复杂度：O(N)
```

```
 *
 * 工程化考量：
 * 1. 使用链式前向星存储树结构，提高空间效率和遍历速度
 * 2. 使用 BufferedReader 和 PrintWriter 进行高效输入输出
 * 3. 通过预处理确定父子关系，简化操作逻辑
 * 4. 使用 long 类型避免整数溢出
 *
 * 最优解分析：
 * 树上差分是解决此类问题的最优解，通过 O(1)的操作标记区间修改，
 * 避免了暴力遍历每个连通分量的所有节点，时间复杂度比暴力方法的 O(N*Q)有极大提升。
 */

import java.io.*;
import java.util.*;

public class Code11_AtCoderABC187E {

    /**
     * 最大节点数，根据题目数据范围设定
     * 题目中 N 最大为 2e5，设置为 200001 以避免越界
     */
    public static int MAXN = 200001;

    /**
     * 链式前向星建图
     * head[u]: 节点 u 的第一条边的索引
     * next[e]: 边 e 的下一条边的索引
     * to[e]: 边 e 指向的节点
     * cnt: 边的计数器
     */
    public static int[] head = new int[MAXN];
    public static int[] next = new int[MAXN << 1];
    public static int[] to = new int[MAXN << 1];
    public static int cnt = 1;

    /**
     * 存储边的信息
     * edges[i][0]和 edges[i][1]分别表示第 i 条边的两个端点
     */
    public static int[][] edges = new int[MAXN][2];

    /**
     * 差分数组
```

```
 * diff[u]表示节点 u 的差分标记值
 */
public static long[] diff = new long[MAXN];

/**
 * DFS 相关数组
 * parent[u]: 节点 u 的父节点
 * depth[u]: 节点 u 的深度
 * size[u]: 以节点 u 为根的子树大小
 */
public static int[] parent = new int[MAXN];
public static int[] depth = new int[MAXN];
public static int[] size = new int[MAXN];

/**
 * 向链式前向星中添加一条边
 *
 * @param u 边的起始节点
 * @param v 边的结束节点
 */
public static void addEdge(int u, int v) {
    // 添加 u 到 v 的边
    next[cnt] = head[u];
    to[cnt] = v;
    head[u] = cnt++;
}

/**
 * 第一次 DFS，预处理树结构信息
 *
 * @param u 当前处理的节点
 * @param fa 当前节点的父节点
 */
public static void dfs(int u, int fa) {
    // 设置当前节点的父节点
    parent[u] = fa;
    // 设置当前节点的深度
    depth[u] = depth[fa] + 1;
    // 初始化当前节点的子树大小
    size[u] = 1;

    // 遍历当前节点的所有邻接节点
    for (int e = head[u]; e != 0; e = next[e]) {
```

```
            int v = to[e];
            // 只处理子节点（避免回到父节点）
            if (v != fa) {
                // 递归处理子节点
                dfs(v, u);
                // 累加子节点的子树大小
                size[u] += size[v];
            }
        }
    }

    /**
     * 主函数，处理输入、算法执行和输出
     *
     * 输入格式：
     * 第一行：节点数 n
     * 接下来 n-1 行：每行两个整数，表示树的边
     * 接下来一行：操作数 q
     * 接下来 q 行：每行三个整数，表示一次操作
     *
     * 输出格式：
     * n 行，每行一个整数，表示每个节点的最终值
     */
    public static void main(String[] args) throws IOException {
        // 使用高效的输入输出方式
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));

        // 读取节点数量
        int n = Integer.parseInt(br.readLine().trim());

        // 读入边并构建无向树
        for (int i = 1; i < n; i++) {
            StringTokenizer st = new StringTokenizer(br.readLine());
            int a = Integer.parseInt(st.nextToken());
            int b = Integer.parseInt(st.nextToken());
            // 存储边的信息
            edges[i][0] = a;
            edges[i][1] = b;
            // 无向树需要添加双向边
            addEdge(a, b);
            addEdge(b, a);
        }
```

```
// 以 1 为根节点进行 DFS，预处理树结构信息
dfs(1, 0);

// 读取操作数
int q = Integer.parseInt(br.readLine().trim());

// 处理每次操作
for (int i = 0; i < q; i++) {
    StringTokenizer st = new StringTokenizer(br.readLine());
    int t = Integer.parseInt(st.nextToken());  // 操作类型
    int e = Integer.parseInt(st.nextToken());  // 边的编号
    long x = Long.parseLong(st.nextToken());   // 增加的值

    // 获取边的两个端点
    int a = edges[e][0];
    int b = edges[e][1];

    // 确保 a 是 b 的父节点（通过深度判断）
    if (depth[a] > depth[b]) {
        int temp = a;
        a = b;
        b = temp;
    }

    // 根据操作类型执行相应的差分标记
    if (t == 1) {
        /**
         * 操作 1：给 a 所在的连通分量（不包含 b）的所有节点加上 x
         * 实现方法：
         * 1. 给整棵树加上 x（在根节点标记）
         * 2. 给 b 的子树减去 x（在 b 节点标记），抵消不需要增加的部分
         */
        diff[1] += x;    // 给整棵树加上 x
        diff[b] -= x;    // 给 b 的子树减去 x
    } else {
        /**
         * 操作 2：给 b 所在的连通分量（不包含 a）的所有节点加上 x
         * 实现方法：
         * 直接给 b 的子树加上 x（在 b 节点标记）
         */
        diff[b] += x;    // 给 b 的子树加上 x
    }
```

```
        }

        // 通过 DFS 计算差分数组的累加结果，得到每个节点的最终值
        dfsCalc(1, 0);

        // 输出每个节点的最终值
        for (int i = 1; i <= n; i++) {
            out.println(diff[i]);
        }

        // 关闭资源
        out.flush();
        out.close();
    }

    /**
     * 第二次 DFS，通过回溯累加差分标记，计算每个节点的最终值
     *
     * @param u 当前处理的节点
     * @param fa 当前节点的父节点
     */
    public static void dfsCalc(int u, int fa) {
        // 遍历当前节点的所有邻接节点
        for (int e = head[u]; e != 0; e = next[e]) {
            int v = to[e];
            // 只处理子节点（避免回到父节点）
            if (v != fa) {
                // 将父节点的差分值累加到子节点
                diff[v] += diff[u];
                // 递归处理子节点
                dfsCalc(v, u);
            }
        }
    }
}

==========

文件: Code11_AtCoderABC187E.py
==========
"""
AtCoder ABC187 E - Through Path（树上差分）
题目链接：https://atcoder.jp/contests/abc187/tasks/abc187_e
```

题目描述：给定一棵树，有 Q 次操作，每次操作有两种类型：
1. 选择一条边(a,b)，给所有从 a 出发不经过 b 能到达的节点加上 x
2. 选择一条边(a,b)，给所有从 b 出发不经过 a 能到达的节点加上 x
所有操作完成后，输出每个节点的值
解法：树上差分 + DFS

算法思路：
1. 对于每条边(a,b)，将树分为两个连通分量
2. 操作 1：给 a 所在的连通分量（不包含 b）的所有节点加上 x
3. 操作 2：给 b 所在的连通分量（不包含 a）的所有节点加上 x
4. 使用树上差分技术，在根节点处打标记，通过 DFS 计算子树和

时间复杂度：O(N + Q)
空间复杂度：O(N)
"""

```
import sys
sys.setrecursionlimit(300000)

def main():
    import sys
    input = sys.stdin.readline

    n = int(input().strip())

    # 邻接表存储树
    graph = [[] for _ in range(n+1)]

    # 存储边的信息
    edges = [[0, 0] for _ in range(n)]

    # 读入边
    for i in range(1, n):
        a, b = map(int, input().split())
        edges[i][0] = a
        edges[i][1] = b
        graph[a].append(b)
        graph[b].append(a)

    # DFS 相关数组
    parent = [0] * (n+1)
    depth = [0] * (n+1)
    size = [0] * (n+1)
```

```
# 差分数组
diff = [0] * (n+1)

# DFS 预处理树结构
def dfs(u, fa):
    parent[u] = fa
    depth[u] = depth[fa] + 1
    size[u] = 1
    for v in graph[u]:
        if v != fa:
            dfs(v, u)
            size[u] += size[v]

# 以 1 为根节点进行 DFS
depth[0] = -1
dfs(1, 0)

q = int(input().strip())

for _ in range(q):
    t, e, x = map(int, input().split())

    a = edges[e][0]
    b = edges[e][1]

    # 确保 a 是 b 的父节点
    if depth[a] > depth[b]:
        a, b = b, a

    if t == 1:
        # 操作 1：给 a 所在的连通分量（不包含 b）加上 x
        # 相当于给整棵树加上 x，然后给 b 的子树减去 x
        diff[1] += x
        diff[b] -= x
    else:
        # 操作 2：给 b 所在的连通分量（不包含 a）加上 x
        # 相当于给 b 的子树加上 x
        diff[b] += x

# DFS 计算子树和
def dfs_calc(u, fa):
    for v in graph[u]:
```

```
            if v != fa:
                diff[v] += diff[u]
                dfs_calc(v, u)

    dfs_calc(1, 0)

    # 输出结果
    for i in range(1, n+1):
        print(diff[i])

if __name__ == "__main__":
    main()
```

==========

文件: Code12_Codeforces191C.cpp

==========

```
/**
 * Codeforces 191C - Fools and Roads（树上边差分）
 * 题目链接：https://codeforces.com/contest/191/problem/C
 * 题目描述：给定一棵树，有 k 对节点(u,v)，对于每对节点，它们之间的路径上的每条边都会被经过一次
 * 求每条边被经过的次数
 * 解法：树上边差分 + LCA
 *
 * 算法思路：
 * 1. 对于每对节点(u,v)，它们之间的路径上的每条边都会被经过一次
 * 2. 使用树上边差分技术：
 *    - 在 u 和 v 处+1
 *    - 在 LCA(u,v)处-2
 * 3. 通过 DFS 计算子树和，得到每条边的经过次数
 *
 * 时间复杂度：O(N log N + K log N)
 * 空间复杂度：O(N log N)
 */

#include <iostream>
#include <vector>
#include <cstring>
#include <algorithm>
using namespace std;

const int MAXN = 100001;
const int LIMIT = 17;
```

```
// 邻接表存储树
vector<pair<int, int>> graph[MAXN]; // pair<节点, 边 ID>

// LCA 相关
int depth[MAXN];
int stjump[MAXN][LIMIT];

// 差分数组
int diff[MAXN];

// 边对应的答案
int ans[MAXN];

// DFS 预处理 LCA
void dfs(int u, int fa) {
    depth[u] = depth[fa] + 1;
    stjump[u][0] = fa;
    for (int p = 1; p < LIMIT; p++) {
        stjump[u][p] = stjump[stjump[u][p-1]][p-1];
    }
    for (auto &edge : graph[u]) {
        int v = edge.first;
        if (v != fa) {
            dfs(v, u);
        }
    }
}

// 求 LCA
int lca(int a, int b) {
    if (depth[a] < depth[b]) {
        swap(a, b);
    }
    // 将 a 调整到与 b 同一深度
    for (int p = LIMIT - 1; p >= 0; p--) {
        if (depth[stjump[a][p]] >= depth[b]) {
            a = stjump[a][p];
        }
    }
    if (a == b) return a;
    // 同时向上跳
    for (int p = LIMIT - 1; p >= 0; p--) {
```

```
        if (stjump[a][p] != stjump[b][p]) {
            a = stjump[a][p];
            b = stjump[b][p];
        }
    }
    return stjump[a][0];
}

// DFS 计算子树和
void dfsCalc(int u, int fa) {
    for (auto &edge : graph[u]) {
        int v = edge.first;
        int edgeId = edge.second;
        if (v != fa) {
            dfsCalc(v, u);
            diff[u] += diff[v];
            ans[edgeId] = diff[v];
        }
    }
}

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);

    int n;
    cin >> n;

    // 读入边
    for (int i = 1; i < n; i++) {
        int u, v;
        cin >> u >> v;
        graph[u].push_back({v, i});
        graph[v].push_back({u, i});
    }

    // 预处理 LCA
    depth[0] = -1;
    dfs(1, 0);

    int k;
    cin >> k;
```

```
    // 处理每对节点
    for (int i = 0; i < k; i++) {
        int u, v;
        cin >> u >> v;

        int l = lca(u, v);

        // 树上边差分
        diff[u] += 1;
        diff[v] += 1;
        diff[l] -= 2;
    }

    // 计算最终结果
    dfsCalc(1, 0);

    // 输出每条边的经过次数
    for (int i = 1; i < n; i++) {
        cout << ans[i] << " ";
    }
    cout << endl;

    return 0;
}
```

==========

文件: Code12_Codeforces191C.java

==========

```
package class122;

/**
 * Codeforces 191C - Fools and Roads（树上边差分）
 *
 * 题目来源：Codeforces
 * 题目链接：https://codeforces.com/contest/191/problem/C
 *
 * 题目描述：
 * 给定一棵包含 N 个节点的树，以及 K 对节点(u,v)。
 * 对于每对节点，它们之间的路径上的每条边都会被经过一次。
 * 求每条边被经过的次数。
 *
 * 算法原理：树上边差分 + LCA (最近公共祖先)
```

```
 * 树上边差分是处理树上路径操作的一种高效技术。
 * 对于每对节点(u,v)之间的路径，我们需要让路径上的所有边计数加 1。
 * 通过边差分，我们可以：
 * 1. diff[u]++
 * 2. diff[v]++
 * 3. diff[lca(u,v)] -= 2
 * 最后通过一次 DFS 回溯累加子节点的差分标记，得到每条边的最终计数。
 *
 * 时间复杂度分析：
 * - 预处理 LCA：O(N log N)
 * - 差分标记：O(K log N)，其中 K 是操作次数
 * - DFS 回溯统计：O(N)
 * 总时间复杂度：O(N log N + K log N)
 *
 * 空间复杂度分析：
 * - 树的存储：O(N)
 * - LCA 倍增数组：O(N log N)
 * - 差分数组：O(N)
 * - 答案数组：O(N)
 * 总空间复杂度：O(N log N)
 *
 * 工程化考量：
 * 1. 使用链式前向星存储树结构，提高空间效率和遍历速度
 * 2. 使用 BufferedReader 和 PrintWriter 进行高效输入输出
 * 3. 预处理 log2 值，优化倍增数组的大小
 * 4. 通过 edgeId 数组将边与答案数组关联，便于输出
 *
 * 最优解分析：
 * 树上边差分是解决此类路径覆盖问题的最优解，相比暴力遍历每条路径的 O(N*K)复杂度，
 * 树上边差分可以将时间复杂度优化到 O(N log N + K log N)，在大规模数据下效率提升显著。
 */

import java.io.*;
import java.util.*;

public class Code12_Codeforces191C {

    /**
     * 最大节点数，根据题目数据范围设定
     * 题目中 N 最大为 1e5，设置为 100001 以避免越界
     */
    public static int MAXN = 100001;
```

```
/**
 * 倍增数组的大小限制
 * 2^17 = 131,072 > 1e5，足够处理最大节点数
 */
public static int LIMIT = 17;

/**
 * 链式前向星建图
 * head[u]: 节点 u 的第一条边的索引
 * next[e]: 边 e 的下一条边的索引
 * to[e]: 边 e 指向的节点
 * edgeId[e]: 边 e 的编号（用于关联答案数组）
 * cnt: 边的计数器
 */
public static int[] head = new int[MAXN];
public static int[] next = new int[MAXN << 1];
public static int[] to = new int[MAXN << 1];
public static int[] edgeId = new int[MAXN << 1];
public static int cnt = 1;

/**
 * LCA 相关数组
 * depth[u]: 节点 u 的深度
 * stjump[u][p]: 节点 u 的 2^p 级祖先
 */
public static int[] depth = new int[MAXN];
public static int[][] stjump = new int[MAXN][LIMIT];

/**
 * 差分数组
 * diff[u]: 节点 u 到其父节点的边被路径覆盖的次数的差分值
 */
public static int[] diff = new int[MAXN];

/**
 * 答案数组
 * ans[i]: 第 i 条边被经过的次数
 */
public static int[] ans = new int[MAXN];

/**
 * 向链式前向星中添加一条边
 *
```

```
     * @param u 边的起始节点
     * @param v 边的结束节点
     * @param id 边的编号
     */
    public static void addEdge(int u, int v, int id) {
        // 添加u到v的边
        next[cnt] = head[u];
        to[cnt] = v;
        edgeId[cnt] = id;  // 记录边的编号
        head[u] = cnt++;
    }

    /**
     * 第一次DFS，预处理每个节点的深度和倍增跳跃数组
     * 该DFS构建LCA所需的数据结构
     *
     * @param u 当前处理的节点
     * @param fa 当前节点的父节点
     */
    public static void dfs(int u, int fa) {
        // 设置当前节点的深度（父节点深度+1）
        depth[u] = depth[fa] + 1;
        // 设置当前节点的直接父节点
        stjump[u][0] = fa;

        // 预处理倍增数组，stjump[u][p]表示u的2^p级祖先
        // 利用动态规划的思想：u的2^p级祖先 = u的2^(p-1)级祖先的2^(p-1)级祖先
        for (int p = 1; p < LIMIT; p++) {
            stjump[u][p] = stjump[stjump[u][p-1]][p-1];
        }

        // 深度优先遍历所有子节点
        for (int e = head[u]; e != 0; e = next[e]) {
            int v = to[e];
            // 避免回到父节点，造成无限递归
            if (v != fa) {
                dfs(v, u);
            }
        }
    }

    /**
     * 使用倍增法计算两个节点的最近公共祖先(LCA)
```

```
     *
     * @param a 第一个节点
     * @param b 第二个节点
     * @return a 和 b 的最近公共祖先
     *
     * 算法步骤：
     * 1. 确保 a 的深度不小于 b
     * 2. 将 a 向上跳跃到与 b 同一深度
     * 3. 如果此时 a==b，则直接返回 a 作为 LCA
     * 4. 否则，a 和 b 同时向上跳跃，直到它们的父节点相同
     * 5. 返回最终的父节点作为 LCA
     */
    public static int lca(int a, int b) {
        // 步骤 1：确保 a 的深度不小于 b
        if (depth[a] < depth[b]) {
            int temp = a;
            a = b;
            b = temp;
        }

        // 步骤 2：将 a 向上跳到与 b 同一深度
        // 从最高幂次开始尝试跳跃，确保最大步长
        for (int p = LIMIT - 1; p >= 0; p--) {
            if (depth[stjump[a][p]] >= depth[b]) {
                a = stjump[a][p];
            }
        }

        // 步骤 3：如果 a 和 b 相遇，说明找到了 LCA
        if (a == b) return a;

        // 步骤 4：同时向上跳跃，直到找到 LCA 的直接子节点
        for (int p = LIMIT - 1; p >= 0; p--) {
            if (stjump[a][p] != stjump[b][p]) {
                a = stjump[a][p];
                b = stjump[b][p];
            }
        }

        // 步骤 5：返回它们的父节点作为 LCA
        return stjump[a][0];
    }
```

```
/**
 * 第二次 DFS，通过回溯累加子节点的差分标记，计算每条边被经过的最终次数
 * 这是树上边差分的关键步骤，将局部标记转化为全局计数
 *
 * @param u 当前处理的节点
 * @param fa 当前节点的父节点
 */
public static void dfsCalc(int u, int fa) {
    // 遍历当前节点的所有邻接边
    for (int e = head[u]; e != 0; e = next[e]) {
        int v = to[e];
        // 只处理子节点（避免回到父节点）
        if (v != fa) {
            // 递归处理子节点
            dfsCalc(v, u);

            // 将子节点的差分标记累加到当前节点
            diff[u] += diff[v];

            // 将子节点的最终计数存储到对应边的答案数组中
            ans[edgeId[e]] = diff[v];
        }
    }
}

/**
 * 主函数，处理输入、算法执行和输出
 *
 * 输入格式：
 * 第一行：节点数 n
 * 接下来 n-1 行：每行两个整数，表示树的边
 * 接下来一行：操作数 k
 * 接下来 k 行：每行两个整数，表示一次操作的两个节点
 *
 * 输出格式：
 * 一行 n-1 个整数，表示每条边被经过的次数
 */
public static void main(String[] args) throws IOException {
    // 使用高效的输入输出方式
    BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
    PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));

    // 读取节点数量
```

```
int n = Integer.parseInt(br.readLine().trim());

// 读入边并构建无向树
for (int i = 1; i < n; i++) {
    StringTokenizer st = new StringTokenizer(br.readLine());
    int u = Integer.parseInt(st.nextToken());
    int v = Integer.parseInt(st.nextToken());
    // 无向树需要添加双向边，并记录边的编号
    addEdge(u, v, i);
    addEdge(v, u, i);
}

// 预处理 LCA 所需的深度数组和倍增数组
// 设置根节点的父节点深度为-1，避免越界
depth[0] = -1;
dfs(1, 0);

// 读取操作数
int k = Integer.parseInt(br.readLine().trim());

// 处理每对节点 - 执行树上边差分
for (int i = 0; i < k; i++) {
    StringTokenizer st = new StringTokenizer(br.readLine());
    int u = Integer.parseInt(st.nextToken());
    int v = Integer.parseInt(st.nextToken());

    // 计算 u 和 v 的最近公共祖先
    int l = lca(u, v);

    /**
     * 树上边差分核心操作：
     * 对于路径 u->v，我们希望路径上的所有边计数加 1
     * 通过边差分技巧，我们只需要修改三个点：
     * 1. diff[u]++ - 在 u 点增加标记
     * 2. diff[v]++ - 在 v 点增加标记
     * 3. diff[lca] -= 2 - 在 LCA 处抵消多余的标记
     *
     * 这样，当执行 dfsCalc 回溯累分时，整个路径上的边都会被正确计数
     */
    diff[u] += 1;
    diff[v] += 1;
    diff[l] -= 2;
}
```

```
        // 执行第二次DFS，通过回溯累加子节点的差分标记，计算每条边的最终经过次数
        dfsCalc(1, 0);

        // 输出每条边的经过次数
        for (int i = 1; i < n; i++) {
            out.print(ans[i] + " ");
        }
        out.println();

        // 关闭资源
        out.flush();
        out.close();
    }
}
```

==========

文件: Code12_Codeforces191C.py

==========

```
"""
Codeforces 191C - Fools and Roads（树上边差分）
题目链接：https://codeforces.com/contest/191/problem/C
题目描述：给定一棵树，有k对节点(u,v)，对于每对节点，它们之间的路径上的每条边都会被经过一次
求每条边被经过的次数
解法：树上边差分 + LCA

算法思路：
1. 对于每对节点(u,v)，它们之间的路径上的每条边都会被经过一次
2. 使用树上边差分技术：
   - 在u和v处+1
   - 在LCA(u,v)处-2
3. 通过DFS计算子树和，得到每条边的经过次数

时间复杂度：O(N log N + K log N)
空间复杂度：O(N log N)
"""

import sys
sys.setrecursionlimit(300000)

def main():
    import sys
```

```
input = sys.stdin.readline

n = int(input().strip())

# 邻接表存储树，存储(节点, 边 ID)
graph = [[] for _ in range(n+1)]

# 读入边
for i in range(1, n):
    u, v = map(int, input().split())
    graph[u].append((v, i))
    graph[v].append((u, i))

# LCA 相关
LIMIT = 17
depth = [0] * (n+1)
stjump = [[0] * LIMIT for _ in range(n+1)]

# 差分数组
diff = [0] * (n+1)

# 边对应的答案
ans = [0] * (n+1)

# DFS 预处理 LCA
def dfs(u, fa):
    depth[u] = depth[fa] + 1
    stjump[u][0] = fa
    for p in range(1, LIMIT):
        stjump[u][p] = stjump[stjump[u][p-1]][p-1]
    for v, edge_id in graph[u]:
        if v != fa:
            dfs(v, u)

# 以 1 为根节点进行 DFS
depth[0] = -1
dfs(1, 0)

# 求 LCA
def lca(a, b):
    if depth[a] < depth[b]:
        a, b = b, a
    # 将 a 调整到与 b 同一深度
```

```
        for p in range(LIMIT-1, -1, -1):
            if depth[stjump[a][p]] >= depth[b]:
                a = stjump[a][p]
        if a == b:
            return a
        # 同时向上跳
        for p in range(LIMIT-1, -1, -1):
            if stjump[a][p] != stjump[b][p]:
                a = stjump[a][p]
                b = stjump[b][p]
        return stjump[a][0]

    k = int(input().strip())

    # 处理每对节点
    for _ in range(k):
        u, v = map(int, input().split())

        l = lca(u, v)

        # 树上边差分
        diff[u] += 1
        diff[v] += 1
        diff[l] -= 2

    # DFS 计算子树和
    def dfs_calc(u, fa):
        for v, edge_id in graph[u]:
            if v != fa:
                dfs_calc(v, u)
                diff[u] += diff[v]
                ans[edge_id] = diff[v]

    dfs_calc(1, 0)

    # 输出每条边的经过次数
    result = []
    for i in range(1, n):
        result.append(str(ans[i]))
    print(' '.join(result))

if __name__ == "__main__":
    main()
```

==========

文件: Code13_LeetCode1483.cpp
==========

```
/**
 * LeetCode 1483. 树节点的第 K 个祖先（树上倍增法）
 * 题目链接：https://leetcode.com/problems/kth-ancestor-of-a-tree-node/
 * 题目描述：给定一棵树，每个节点有唯一的父节点，实现一个类 TreeAncestor：
 *   - TreeAncestor(int n, int[] parent): 初始化，n 个节点，parent[i]是节点 i 的父节点
 *   - int getKthAncestor(int node, int k): 返回节点 node 的第 k 个祖先节点
 * 解法：树上倍增法（二进制拆分）
 *
 * 算法思路：
 * 1. 使用倍增法预处理每个节点的 2^i 级祖先
 * 2. 对于查询 getKthAncestor(node, k)，将 k 二进制拆分
 * 3. 从高位到低位，如果 k 的第 i 位为 1，则 node 跳到其 2^i 级祖先
 * 4. 重复直到 k 为 0 或 node 为-1
 *
 * 时间复杂度：
 *   - 初始化：O(N log N)
 *   - 查询：O(log K)
 * 空间复杂度：O(N log N)
 */

#include <iostream>
#include <vector>
#include <cmath>
using namespace std;

class TreeAncestor {
private:
    int n;
    int maxLevel;
    vector<vector<int>> stjump;

public:
    /**
     * 构造函数：初始化倍增数组
     * @param n 节点数量
     * @param parent 父节点数组，parent[i]是节点 i 的父节点
     */
    TreeAncestor(int n, vector<int>& parent) {
```

```
    this->n = n;
    // 计算最大层数：log2(n)
    this->maxLevel = 0;
    while ((1 << maxLevel) <= n) {
        maxLevel++;
    }

    // 初始化倍增数组
    stjump.resize(n, vector<int>(maxLevel, -1));

    // 初始化第0级祖先（直接父节点）
    for (int i = 0; i < n; i++) {
        stjump[i][0] = parent[i];
    }

    // 预处理倍增数组
    for (int j = 1; j < maxLevel; j++) {
        for (int i = 0; i < n; i++) {
            if (stjump[i][j-1] == -1) {
                stjump[i][j] = -1;
            } else {
                stjump[i][j] = stjump[stjump[i][j-1]][j-1];
            }
        }
    }
}

/**
 * 查询节点node的第k个祖先
 * @param node 当前节点
 * @param k 祖先级别
 * @return 第k个祖先节点，如果不存在返回-1
 */
int getKthAncestor(int node, int k) {
    if (node < 0 || node >= n || k < 0) {
        return -1;
    }

    // 二进制拆分k
    for (int j = 0; j < maxLevel; j++) {
        if ((k >> j) & 1) {
            node = stjump[node][j];
            if (node == -1) {
```

```
                    return -1;
                }
            }
        }

        return node;
    }
};

/**
 * 测试用例
 */
int main() {
    // 测试用例 1：简单的链式结构
    int n1 = 7;
    vector<int> parent1 = {-1, 0, 0, 1, 1, 2, 2};
    TreeAncestor ta1(n1, parent1);

    // 测试查询
    cout << "测试用例 1:" << endl;
    cout << "节点 3 的第 1 个祖先: " << ta1.getKthAncestor(3, 1) << endl; // 期望: 1
    cout << "节点 3 的第 2 个祖先: " << ta1.getKthAncestor(3, 2) << endl; // 期望: 0
    cout << "节点 3 的第 3 个祖先: " << ta1.getKthAncestor(3, 3) << endl; // 期望: -1

    // 测试用例 2：更复杂的树结构
    int n2 = 5;
    vector<int> parent2 = {-1, 0, 0, 1, 2};
    TreeAncestor ta2(n2, parent2);

    cout << "\\n 测试用例 2:" << endl;
    cout << "节点 4 的第 1 个祖先: " << ta2.getKthAncestor(4, 1) << endl; // 期望: 2
    cout << "节点 4 的第 2 个祖先: " << ta2.getKthAncestor(4, 2) << endl; // 期望: 0
    cout << "节点 4 的第 3 个祖先: " << ta2.getKthAncestor(4, 3) << endl; // 期望: -1

    return 0;
}

==========================================

文件: Code13_LeetCode1483.java
==========================================
package class122;
```

```
/**
 * LeetCode 1483. 树节点的第 K 个祖先（树上倍增法）
 *
 * 题目来源：LeetCode
 * 题目链接：https://leetcode.com/problems/kth-ancestor-of-a-tree-node/
 *
 * 题目描述：
 * 给定一棵树，每个节点有唯一的父节点，实现一个类 TreeAncestor：
 * - TreeAncestor(int n, int[] parent): 初始化，n 个节点，parent[i]是节点 i 的父节点
 * - int getKthAncestor(int node, int k): 返回节点 node 的第 k 个祖先节点
 *
 * 算法原理：树上倍增法（二进制拆分）
 * 树上倍增法是一种高效的处理树上祖先查询的技术。
 *
 * 解题思路：
 * 1. 使用倍增法预处理每个节点的 2^i 级祖先
 * 2. 对于查询 getKthAncestor(node, k)，将 k 二进制拆分
 * 3. 从高位到低位，如果 k 的第 i 位为 1，则 node 跳到其 2^i 级祖先
 * 4. 重复直到 k 为 0 或 node 为-1
 *
 * 时间复杂度分析：
 * - 初始化：O(N log N)
 * - 查询：O(log K)
 *
 * 空间复杂度分析：
 * - 倍增数组：O(N log N)
 *
 * 工程化考量：
 * 1. 使用二维数组存储倍增信息，便于快速查询
 * 2. 预处理阶段计算所有可能的跳跃，查询时只需常数次操作
 * 3. 边界处理：根节点的父节点为-1，查询时注意越界检查
 *
 * 最优解分析：
 * 树上倍增法是解决此类问题的最优解，相比朴素方法的 O(K)查询时间，
 * 倍增法将查询时间优化到 O(log K)，在大规模数据下效率提升显著。
 */

import java.util.*;

public class Code13_LeetCode1483 {

    class TreeAncestor {
        private int n;           // 节点数量
```

```
private int maxLevel;    // 最大层数（log2(n)）
private int[][] stjump;  // 倍增数组，stjump[i][j]表示节点 i 的 2^j 级祖先

/**
 * 构造函数：初始化倍增数组
 *
 * @param n 节点数量
 * @param parent 父节点数组，parent[i]是节点 i 的父节点
 */
public TreeAncestor(int n, int[] parent) {
    this.n = n;
    // 计算最大层数：log2(n)
    this.maxLevel = 0;
    while ((1 << maxLevel) <= n) {
        maxLevel++;
    }

    // 初始化倍增数组
    stjump = new int[n][maxLevel];

    // 初始化第 0 级祖先（直接父节点）
    for (int i = 0; i < n; i++) {
        stjump[i][0] = parent[i];
    }

    // 预处理倍增数组
    // 利用动态规划思想：节点 i 的 2^j 级祖先 = 节点 i 的 2^(j-1)级祖先的 2^(j-1)级祖先
    for (int j = 1; j < maxLevel; j++) {
        for (int i = 0; i < n; i++) {
            if (stjump[i][j-1] == -1) {
                // 如果 2^(j-1)级祖先不存在，则 2^j 级祖先也不存在
                stjump[i][j] = -1;
            } else {
                // 节点 i 的 2^j 级祖先 = 节点 i 的 2^(j-1)级祖先的 2^(j-1)级祖先
                stjump[i][j] = stjump[stjump[i][j-1]][j-1];
            }
        }
    }
}

/**
 * 查询节点 node 的第 k 个祖先
 *
```

```
     * @param node 当前节点
     * @param k 祖先级别
     * @return 第k个祖先节点，如果不存在返回-1
     *
     * 算法原理：
     * 将k进行二进制拆分，例如k=13=1101(2)=2^3+2^2+2^0
     * 则节点node的第13个祖先 = node的第8个祖先的第4个祖先的第1个祖先
     */
    public int getKthAncestor(int node, int k) {
        // 边界检查
        if (node < 0 || node >= n || k < 0) {
            return -1;
        }

        // 二进制拆分k，从低位到高位检查每一位
        for (int j = 0; j < maxLevel; j++) {
            // 如果k的第j位为1
            if (((k >> j) & 1) == 1) {
                // 节点跳到其2^j级祖先
                node = stjump[node][j];
                // 如果祖先不存在，直接返回-1
                if (node == -1) {
                    return -1;
                }
            }
        }

        return node;
    }
}

/**
 * 测试用例
 */
public static void main(String[] args) {
    // 测试用例1：简单的链式结构
    int n1 = 7;
    int[] parent1 = {-1, 0, 0, 1, 1, 2, 2};
    TreeAncestor ta1 = new Code13_LeetCode1483().new TreeAncestor(n1, parent1);

    // 测试查询
    System.out.println("测试用例1:");
    System.out.println("节点3的第1个祖先: " + ta1.getKthAncestor(3, 1)); // 期望: 1
```

```
        System.out.println("节点 3 的第 2 个祖先: " + ta1.getKthAncestor(3, 2)); // 期望: 0
        System.out.println("节点 3 的第 3 个祖先: " + ta1.getKthAncestor(3, 3)); // 期望: -1

        // 测试用例 2：更复杂的树结构
        int n2 = 5;
        int[] parent2 = {-1, 0, 0, 1, 2};
        TreeAncestor ta2 = new Code13_LeetCode1483().new TreeAncestor(n2, parent2);

        System.out.println("\n 测试用例 2:");
        System.out.println("节点 4 的第 1 个祖先: " + ta2.getKthAncestor(4, 1)); // 期望: 2
        System.out.println("节点 4 的第 2 个祖先: " + ta2.getKthAncestor(4, 2)); // 期望: 0
        System.out.println("节点 4 的第 3 个祖先: " + ta2.getKthAncestor(4, 3)); // 期望: -1
    }
}

==========================================

文件: Code13_LeetCode1483.py
==========================================
"""
LeetCode 1483. 树节点的第 K 个祖先（树上倍增法）
题目链接：https://leetcode.com/problems/kth-ancestor-of-a-tree-node/
题目描述：给定一棵树，每个节点有唯一的父节点，实现一个类 TreeAncestor：
   - TreeAncestor(n, parent): 初始化，n 个节点，parent[i]是节点 i 的父节点
   - getKthAncestor(node, k): 返回节点 node 的第 k 个祖先节点
解法：树上倍增法（二进制拆分）

算法思路：
1. 使用倍增法预处理每个节点的 2^i 级祖先
2. 对于查询 getKthAncestor(node, k)，将 k 二进制拆分
3. 从高位到低位，如果 k 的第 i 位为 1，则 node 跳到其 2^i 级祖先
4. 重复直到 k 为 0 或 node 为-1

时间复杂度：
   - 初始化：O(N log N)
   - 查询：O(log K)
空间复杂度：O(N log N)
"""

import math

class TreeAncestor:
    """
```

```
树节点的第 K 个祖先类
"""

def __init__(self, n: int, parent: list):
    """
    构造函数：初始化倍增数组
    :param n: 节点数量
    :param parent: 父节点数组，parent[i]是节点 i 的父节点
    """
    self.n = n
    # 计算最大层数：log2(n)
    self.max_level = 0
    while (1 << self.max_level) <= n:
        self.max_level += 1

    # 初始化倍增数组
    self.stjump = [[-1] * self.max_level for _ in range(n)]

    # 初始化第 0 级祖先（直接父节点）
    for i in range(n):
        self.stjump[i][0] = parent[i]

    # 预处理倍增数组
    for j in range(1, self.max_level):
        for i in range(n):
            if self.stjump[i][j-1] == -1:
                self.stjump[i][j] = -1
            else:
                self.stjump[i][j] = self.stjump[self.stjump[i][j-1]][j-1]

def getKthAncestor(self, node: int, k: int) -> int:
    """
    查询节点 node 的第 k 个祖先
    :param node: 当前节点
    :param k: 祖先级别
    :return: 第 k 个祖先节点，如果不存在返回-1
    """
    if node < 0 or node >= self.n or k < 0:
        return -1

    # 二进制拆分 k
    for j in range(self.max_level):
        if (k >> j) & 1:
```

```
                node = self.stjump[node][j]
                if node == -1:
                    return -1

        return node

def main():
    """
    测试用例
    """
    # 测试用例 1：简单的链式结构
    n1 = 7
    parent1 = [-1, 0, 0, 1, 1, 2, 2]
    ta1 = TreeAncestor(n1, parent1)

    # 测试查询
    print("测试用例 1:")
    print(f"节点 3 的第 1 个祖先: {ta1.getKthAncestor(3, 1)}")  # 期望: 1
    print(f"节点 3 的第 2 个祖先: {ta1.getKthAncestor(3, 2)}")  # 期望: 0
    print(f"节点 3 的第 3 个祖先: {ta1.getKthAncestor(3, 3)}")  # 期望: -1

    # 测试用例 2：更复杂的树结构
    n2 = 5
    parent2 = [-1, 0, 0, 1, 2]
    ta2 = TreeAncestor(n2, parent2)

    print("\n测试用例 2:")
    print(f"节点 4 的第 1 个祖先: {ta2.getKthAncestor(4, 1)}")  # 期望: 2
    print(f"节点 4 的第 2 个祖先: {ta2.getKthAncestor(4, 2)}")  # 期望: 0
    print(f"节点 4 的第 3 个祖先: {ta2.getKthAncestor(4, 3)}")  # 期望: -1

if __name__ == "__main__":
    main()

==============================================

文件: Code14_LeetCode235.cpp
==============================================
/**
 * LeetCode 235. 二叉搜索树的最近公共祖先（LCA）
 * 题目链接：https://leetcode.com/problems/lowest-common-ancestor-of-a-binary-search-tree/
 * 题目描述：给定一个二叉搜索树，找到两个指定节点的最近公共祖先
 * 解法：利用二叉搜索树的性质
```

```
 *
 * 算法思路：
 * 1. 利用二叉搜索树的性质：左子树所有节点值 < 根节点值 < 右子树所有节点值
 * 2. 如果 p 和 q 的值都小于当前节点值，则 LCA 在左子树
 * 3. 如果 p 和 q 的值都大于当前节点值，则 LCA 在右子树
 * 4. 否则当前节点就是 LCA
 *
 * 时间复杂度：O(h)，h 为树的高度
 * 空间复杂度：O(1)
 */

#include <iostream>
using namespace std;

// 二叉树节点定义
struct TreeNode {
    int val;
    TreeNode *left;
    TreeNode *right;
    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
};

class Solution {
public:
    /**
     * 二叉搜索树的最近公共祖先（迭代版本）
     * @param root 根节点
     * @param p 节点 p
     * @param q 节点 q
     * @return 最近公共祖先
     */
    TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q) {
        TreeNode* current = root;

        while (current != nullptr) {
            // 如果 p 和 q 的值都小于当前节点值，LCA 在左子树
            if (p->val < current->val && q->val < current->val) {
                current = current->left;
            }
            // 如果 p 和 q 的值都大于当前节点值，LCA 在右子树
            else if (p->val > current->val && q->val > current->val) {
                current = current->right;
            }
```

```
            // 否则当前节点就是 LCA
            else {
                return current;
            }
        }

        return nullptr; // 理论上不会执行到这里
    }

    /**
     * 二叉搜索树的最近公共祖先（递归版本）
     * @param root 根节点
     * @param p 节点 p
     * @param q 节点 q
     * @return 最近公共祖先
     */
    TreeNode* lowestCommonAncestorRecursive(TreeNode* root, TreeNode* p, TreeNode* q) {
        // 如果 p 和 q 的值都小于当前节点值，LCA 在左子树
        if (p->val < root->val && q->val < root->val) {
            return lowestCommonAncestorRecursive(root->left, p, q);
        }

        // 如果 p 和 q 的值都大于当前节点值，LCA 在右子树
        if (p->val > root->val && q->val > root->val) {
            return lowestCommonAncestorRecursive(root->right, p, q);
        }

        // 否则当前节点就是 LCA
        return root;
    }
};

/**
 * 测试用例
 */
int main() {
    // 构建测试二叉搜索树
    //       6
    //      / \
    //     2   8
    //    / \ / \
    //   0  4 7  9
    //     / \
```

```
//    3   5

TreeNode* root = new TreeNode(6);
root->left = new TreeNode(2);
root->right = new TreeNode(8);
root->left->left = new TreeNode(0);
root->left->right = new TreeNode(4);
root->right->left = new TreeNode(7);
root->right->right = new TreeNode(9);
root->left->right->left = new TreeNode(3);
root->left->right->right = new TreeNode(5);

Solution solution;

// 测试用例 1：节点 2 和 8 的 LCA
TreeNode* p1 = root->left;  // 节点 2
TreeNode* q1 = root->right;  // 节点 8
TreeNode* lca1 = solution.lowestCommonAncestor(root, p1, q1);
cout << "节点 2 和 8 的 LCA: " << lca1->val << endl; // 期望: 6

// 测试用例 2：节点 2 和 4 的 LCA
TreeNode* p2 = root->left;      // 节点 2
TreeNode* q2 = root->left->right; // 节点 4
TreeNode* lca2 = solution.lowestCommonAncestor(root, p2, q2);
cout << "节点 2 和 4 的 LCA: " << lca2->val << endl; // 期望: 2

// 测试用例 3：节点 3 和 5 的 LCA
TreeNode* p3 = root->left->right->left;  // 节点 3
TreeNode* q3 = root->left->right->right;  // 节点 5
TreeNode* lca3 = solution.lowestCommonAncestor(root, p3, q3);
cout << "节点 3 和 5 的 LCA: " << lca3->val << endl; // 期望: 4

// 测试用例 4：节点 0 和 5 的 LCA
TreeNode* p4 = root->left->left;        // 节点 0
TreeNode* q4 = root->left->right->right; // 节点 5
TreeNode* lca4 = solution.lowestCommonAncestor(root, p4, q4);
cout << "节点 0 和 5 的 LCA: " << lca4->val << endl; // 期望: 2

// 清理内存
delete root->left->right->right;
delete root->left->right->left;
delete root->right->right;
delete root->right->left;
```

```
    delete root->left->right;
    delete root->left->left;
    delete root->right;
    delete root->left;
    delete root;

    return 0;
}
```

==========

文件: Code14_LeetCode235.java

==========

```
package class122;

/**
 * LeetCode 235. 二叉搜索树的最近公共祖先（LCA）
 *
 * 题目来源：LeetCode
 * 题目链接：https://leetcode.com/problems/lowest-common-ancestor-of-a-binary-search-tree/
 *
 * 题目描述：
 * 给定一个二叉搜索树，找到两个指定节点的最近公共祖先。
 * 最近公共祖先的定义：对于有根树 T 的两个节点 p、q，最近公共祖先表示为一个节点 x，
 * 满足 x 是 p、q 的祖先且 x 的深度尽可能大。
 *
 * 算法原理：利用二叉搜索树的性质
 * 二叉搜索树具有重要性质：左子树所有节点值 < 根节点值 < 右子树所有节点值。
 * 利用这个性质可以高效地找到 LCA。
 *
 * 解题思路：
 * 1. 从根节点开始遍历
 * 2. 如果 p 和 q 的值都小于当前节点值，则 LCA 在左子树
 * 3. 如果 p 和 q 的值都大于当前节点值，则 LCA 在右子树
 * 4. 否则当前节点就是 LCA（包括以下情况：p 和 q 分别在左右子树，或其中一个节点就是当前节点）
 *
 * 时间复杂度分析：
 * - 迭代版本：O(h)，其中 h 为树的高度
 * - 递归版本：O(h)，其中 h 为树的高度
 *
 * 空间复杂度分析：
 * - 迭代版本：O(1)
 * - 递归版本：O(h)，递归调用栈空间
```

```
 *
 * 工程化考量：
 * 1. 提供迭代和递归两种实现，满足不同场景需求
 * 2. 迭代版本空间效率更高，避免栈溢出风险
 * 3. 利用BST性质，相比普通二叉树LCA算法更高效
 *
 * 最优解分析：
 * 本解法充分利用了二叉搜索树的性质，是解决此类问题的最优解。
 * 相比于普通二叉树的LCA算法O(N)时间复杂度，本解法将时间复杂度优化到O(h)。
 */

// 二叉树节点定义
class TreeNode {
    int val;
    TreeNode left;
    TreeNode right;
    TreeNode(int x) { val = x; }
}

public class Code14_LeetCode235 {

    /**
     * 二叉搜索树的最近公共祖先（迭代版本）
     *
     * @param root 根节点
     * @param p 节点p
     * @param q 节点q
     * @return 最近公共祖先
     *
     * 算法原理：
     * 利用二叉搜索树的性质进行迭代查找
     * 1. 如果p和q都小于当前节点，则LCA在左子树
     * 2. 如果p和q都大于当前节点，则LCA在右子树
     * 3. 否则当前节点就是LCA
     */
    public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q) {
        TreeNode current = root;

        // 迭代查找LCA
        while (current != null) {
            // 如果p和q的值都小于当前节点值，说明LCA在左子树
            if (p.val < current.val && q.val < current.val) {
                current = current.left;
```

```
        }
        // 如果 p 和 q 的值都大于当前节点值，说明 LCA 在右子树
        else if (p.val > current.val && q.val > current.val) {
            current = current.right;
        }
        // 否则当前节点就是 LCA，包括以下情况：
        // 1. p 和 q 分别在当前节点的左右子树
        // 2. p 或 q 其中一个就是当前节点
        // 3. p 和 q 都是当前节点（理论上不会出现）
        else {
            return current;
        }
    }

    return null; // 理论上不会执行到这里
}

/**
 * 二叉搜索树的最近公共祖先（递归版本）
 *
 * @param root 根节点
 * @param p 节点 p
 * @param q 节点 q
 * @return 最近公共祖先
 *
 * 算法原理：
 * 利用二叉搜索树的性质进行递归查找
 * 1. 如果 p 和 q 都小于当前节点，则 LCA 在左子树
 * 2. 如果 p 和 q 都大于当前节点，则 LCA 在右子树
 * 3. 否则当前节点就是 LCA
 */
public TreeNode lowestCommonAncestorRecursive(TreeNode root, TreeNode p, TreeNode q) {
    // 基本情况：如果节点为空，返回 null

    // 如果 p 和 q 的值都小于当前节点值，说明 LCA 在左子树
    if (p.val < root.val && q.val < root.val) {
        return lowestCommonAncestorRecursive(root.left, p, q);
    }

    // 如果 p 和 q 的值都大于当前节点值，说明 LCA 在右子树
    if (p.val > root.val && q.val > root.val) {
        return lowestCommonAncestorRecursive(root.right, p, q);
    }
```

```
    // 否则当前节点就是 LCA
    return root;
}

/**
 * 测试用例
 */
public static void main(String[] args) {
    // 构建测试二叉搜索树
    //       6
    //      / \
    //     2   8
    //    / \ / \
    //   0  4 7  9
    //     / \
    //    3   5

    TreeNode root = new TreeNode(6);
    root.left = new TreeNode(2);
    root.right = new TreeNode(8);
    root.left.left = new TreeNode(0);
    root.left.right = new TreeNode(4);
    root.right.left = new TreeNode(7);
    root.right.right = new TreeNode(9);
    root.left.right.left = new TreeNode(3);
    root.left.right.right = new TreeNode(5);

    Code14_LeetCode235 solution = new Code14_LeetCode235();

    // 测试用例 1：节点 2 和 8 的 LCA
    TreeNode p1 = root.left;  // 节点 2
    TreeNode q1 = root.right;  // 节点 8
    TreeNode lca1 = solution.lowestCommonAncestor(root, p1, q1);
    System.out.println("节点 2 和 8 的 LCA: " + lca1.val); // 期望: 6

    // 测试用例 2：节点 2 和 4 的 LCA
    TreeNode p2 = root.left;      // 节点 2
    TreeNode q2 = root.left.right; // 节点 4
    TreeNode lca2 = solution.lowestCommonAncestor(root, p2, q2);
    System.out.println("节点 2 和 4 的 LCA: " + lca2.val); // 期望: 2

    // 测试用例 3：节点 3 和 5 的 LCA
```

```
        TreeNode p3 = root.left.right.left;  // 节点 3
        TreeNode q3 = root.left.right.right;  // 节点 5
        TreeNode lca3 = solution.lowestCommonAncestor(root, p3, q3);
        System.out.println("节点 3 和 5 的 LCA: " + lca3.val); // 期望: 4

        // 测试用例 4：节点 0 和 5 的 LCA
        TreeNode p4 = root.left.left;        // 节点 0
        TreeNode q4 = root.left.right.right; // 节点 5
        TreeNode lca4 = solution.lowestCommonAncestor(root, p4, q4);
        System.out.println("节点 0 和 5 的 LCA: " + lca4.val); // 期望: 2
    }
}
```

==========

文件: Code15_LeetCode2646.cpp

==========

```
#include <iostream>
#include <vector>
#include <queue>
#include <algorithm>
#include <cstring>
using namespace std;

/**
 * LeetCode 2646. 最小化旅行的价格
 * 题目描述：给定一棵树，每个节点有一个价格。可以选择将某些节点的价格减半。
 * 有多个旅行路径，每个路径从 u 到 v。要求最小化所有旅行路径的总价格。
 * 使用树上差分统计每条边被经过的次数，然后使用树形 DP 决策哪些节点减半。
 */

class Solution {
private:
    vector<vector<int>> graph;
    vector<int> price;
    vector<int> count; // 节点被经过的次数
    vector<vector<int>> dp; // dp[u][0]:不减半, dp[u][1]:减半

    void dfsCount(int u, int parent) {
        for (int v : graph[u]) {
            if (v != parent) {
                dfsCount(v, u);
                count[u] += count[v];
```

```
        }
    }
}

void dfsDP(int u, int parent) {
    // 不减半的情况
    dp[u][0] = price[u] * count[u];
    // 减半的情况
    dp[u][1] = (price[u] / 2) * count[u];

    for (int v : graph[u]) {
        if (v != parent) {
            dfsDP(v, u);
            // 当前节点不减半，子节点可以减半或不减半
            dp[u][0] += min(dp[v][0], dp[v][1]);
            // 当前节点减半，子节点不能减半
            dp[u][1] += dp[v][0];
        }
    }
}

int getLCA(int u, int v, vector<vector<int>>& parent, vector<int>& depth, int LOG) {
    if (depth[u] < depth[v]) {
        swap(u, v);
    }

    // 将 u 提升到和 v 同一深度
    for (int j = LOG - 1; j >= 0; j--) {
        if (depth[u] - (1 << j) >= depth[v]) {
            u = parent[u][j];
        }
    }

    if (u == v) return u;

    // 同时向上提升
    for (int j = LOG - 1; j >= 0; j--) {
        if (parent[u][j] != parent[v][j]) {
            u = parent[u][j];
            v = parent[v][j];
        }
    }
```

```
        return parent[u][0];
    }

public:
    int minimumTotalPrice(int n, vector<vector<int>>& edges, vector<int>& price,
vector<vector<int>>& trips) {
        // 构建图
        graph.resize(n);
        for (auto& edge : edges) {
            int u = edge[0], v = edge[1];
            graph[u].push_back(v);
            graph[v].push_back(u);
        }

        this->price = price;
        count.resize(n, 0);

        // 预处理 LCA
        int LOG = 20;
        vector<vector<int>> parent(n, vector<int>(LOG, -1));
        vector<int> depth(n, -1);

        // BFS 预处理深度和父节点
        depth[0] = 0;
        queue<int> q;
        q.push(0);

        while (!q.empty()) {
            int u = q.front();
            q.pop();
            for (int v : graph[u]) {
                if (depth[v] == -1) {
                    depth[v] = depth[u] + 1;
                    parent[v][0] = u;
                    q.push(v);
                }
            }
        }

        // 预处理倍增数组
        for (int j = 1; j < LOG; j++) {
            for (int i = 0; i < n; i++) {
                if (parent[i][j-1] != -1) {
```

```
                    parent[i][j] = parent[parent[i][j-1]][j-1];
                }
            }
        }

        // 树上差分统计每条边被经过的次数
        for (auto& trip : trips) {
            int u = trip[0], v = trip[1];
            int lca = getLCA(u, v, parent, depth, LOG);

            count[u]++;
            count[v]++;
            count[lca] -= 2;
        }

        // DFS 统计每个节点被经过的次数
        dfsCount(0, -1);

        // 树形 DP
        dp.resize(n, vector<int>(2, 0));
        dfsDP(0, -1);

        return min(dp[0][0], dp[0][1]);
    }
};

int main() {
    Solution solution;

    // 测试用例 1
    int n1 = 4;
    vector<vector<int>> edges1 = {{0,1},{1,2},{1,3}};
    vector<int> price1 = {2,2,10,6};
    vector<vector<int>> trips1 = {{0,3},{2,1},{2,3}};
    cout << solution.minimumTotalPrice(n1, edges1, price1, trips1) << endl; // 输出: 23

    // 测试用例 2
    int n2 = 2;
    vector<vector<int>> edges2 = {{0,1}};
    vector<int> price2 = {2,2};
    vector<vector<int>> trips2 = {{0,0}};
    cout << solution.minimumTotalPrice(n2, edges2, price2, trips2) << endl; // 输出: 1
```

```
    return 0;
}
```

==========

文件: Code15_LeetCode2646.java
==========

```
package class122;

import java.util.*;

/**
 * LeetCode 2646. 最小化旅行的价格
 *
 * 题目来源：LeetCode
 * 题目链接：https://leetcode.cn/problems/minimize-the-total-price-of-the-trips/
 *
 * 题目描述：
 * 给定一棵树，每个节点有一个价格。可以选择将某些节点的价格减半（但相邻节点不能同时减半）。
 * 有多个旅行路径，每个路径从 u 到 v。要求最小化所有旅行路径的总价格。
 *
 * 算法原理：树上差分 + 树形 DP
 * 1. 使用树上差分统计每条路径经过的节点次数
 * 2. 使用树形 DP 决策哪些节点减半以最小化总价格
 *
 * 解题思路：
 * 1. 首先统计每条路径经过的节点次数（使用树上差分技术）
 * 2. 然后使用树形 DP，在满足相邻节点不能同时减半的约束下，
 *    决策哪些节点减半以最小化总价格
 *
 * 时间复杂度分析：
 * - 建图：O(N)
 * - 预处理 LCA：O(N log N)
 * - 树上差分统计：O(M log N)，其中 M 是旅行路径数
 * - DFS 统计节点次数：O(N)
 * - 树形 DP：O(N)
 * 总时间复杂度：O(N log N + M log N)
 *
 * 空间复杂度分析：
 * - 图的存储：O(N)
 * - LCA 倍增数组：O(N log N)
 * - 差分数组：O(N)
 * - DP 数组：O(N)
```

```
 * 总空间复杂度：O(N log N)
 *
 * 工程化考量：
 * 1. 使用邻接表存储树结构，节省空间
 * 2. 使用 BFS 预处理深度和父节点，避免递归栈溢出
 * 3. 使用倍增法计算 LCA，提高查询效率
 * 4. 树形 DP 状态设计清晰，便于理解和维护
 *
 * 最优解分析：
 * 本解法结合了树上差分和树形 DP，是解决此类问题的最优解。
 * 相比于暴力遍历每条路径的 O(N*M)复杂度，树上差分可以将统计时间优化到 O(M log N)。
 * 树形 DP 在 O(N)时间内完成最优决策，整体效率很高。
 */
public class Code15_LeetCode2646 {

    static class Solution {
        private List<Integer>[] graph;  // 邻接表存储树结构
        private int[] price;            // 每个节点的价格
        private int[] count;            // 每个节点被经过的次数
        private int[][] dp;             // dp[u][0]:节点 u 不减半的最小价格, dp[u][1]:节点 u 减半的
最小价格

        /**
         * 主函数，计算最小化旅行价格的总和
         *
         * @param n 节点数
         * @param edges 树的边
         * @param price 每个节点的价格
         * @param trips 旅行路径
         * @return 最小化旅行价格的总和
         */
        public int minimumTotalPrice(int n, int[][] edges, int[] price, int[][] trips) {
            // 构建图
            graph = new ArrayList[n];
            for (int i = 0; i < n; i++) {
                graph[i] = new ArrayList<>();
            }
            for (int[] edge : edges) {
                int u = edge[0], v = edge[1];
                graph[u].add(v);
                graph[v].add(u);
            }
```

```
this.price = price;
count = new int[n];  // 初始化节点被经过次数数组

// 预处理LCA相关数据
int LOG = 20;  // 倍增数组的大小，2^20足够处理1e5规模的树
int[][] parent = new int[n][LOG];  // parent[i][j]表示节点i的2^j级祖先
int[] depth = new int[n];          // depth[i]表示节点i的深度

// BFS预处理深度和父节点
Arrays.fill(depth, -1);
depth[0] = 0;
Queue<Integer> queue = new LinkedList<>();
queue.offer(0);

while (!queue.isEmpty()) {
    int u = queue.poll();
    for (int v : graph[u]) {
        if (depth[v] == -1) {
            depth[v] = depth[u] + 1;
            parent[v][0] = u;  // 设置直接父节点
            queue.offer(v);
        }
    }
}

// 预处理倍增数组
for (int j = 1; j < LOG; j++) {
    for (int i = 0; i < n; i++) {
        if (parent[i][j-1] != -1) {
            parent[i][j] = parent[parent[i][j-1]][j-1];
        }
    }
}

// 树上差分统计每条路径经过的节点次数
for (int[] trip : trips) {
    int u = trip[0], v = trip[1];
    // 计算u和v的最近公共祖先
    int lca = getLCA(u, v, parent, depth, LOG);

    /**
     * 树上点差分核心操作：
     * 对于路径u->v，我们需要让路径上的所有节点计数加1
```

```
         * 通过点差分技巧，我们只需要修改三个点：
         * 1. count[u]++ - 在起点增加标记
         * 2. count[v]++ - 在终点增加标记
         * 3. count[lca] -= 2 - 在LCA处抵消多余的标记
         *
         * 这样，当执行dfsCount回溯累分时，整个路径上的节点都会被正确计数
         */
        count[u]++;
        count[v]++;
        count[lca] -= 2;
    }

    // DFS统计每个节点被经过的次数
    dfsCount(0, -1);

    // 树形DP决策哪些节点减半
    dp = new int[n][2];
    dfsDP(0, -1);

    // 返回根节点减半或不减半的最小值
    return Math.min(dp[0][0], dp[0][1]);
}

/**
 * DFS统计每个节点被经过的次数
 * 通过回溯累加子节点的差分标记，计算每个节点的最终经过次数
 *
 * @param u 当前处理的节点
 * @param parent 当前节点的父节点
 */
private void dfsCount(int u, int parent) {
    // 遍历当前节点的所有子节点
    for (int v : graph[u]) {
        if (v != parent) {
            // 递归处理子节点
            dfsCount(v, u);
            // 将子节点的经过次数累加到当前节点
            count[u] += count[v];
        }
    }
}

/**
```

```
     * 树形DP，决策哪些节点减半以最小化总价格
     * 状态转移：
     * dp[u][0] = price[u] * count[u] + Σmin(dp[v][0], dp[v][1])  // 当前节点不减半
     * dp[u][1] = (price[u]/2) * count[u] + Σdp[v][0]            // 当前节点减半
     *
     * 约束条件：相邻节点不能同时减半
     *
     * @param u 当前处理的节点
     * @param parent 当前节点的父节点
     */
    private void dfsDP(int u, int parent) {
        // 不减半的情况：当前节点价格不变
        dp[u][0] = price[u] * count[u];
        // 减半的情况：当前节点价格减半
        dp[u][1] = (price[u] / 2) * count[u];

        // 遍历当前节点的所有子节点
        for (int v : graph[u]) {
            if (v != parent) {
                // 递归处理子节点
                dfsDP(v, u);

                /**
                 * 状态转移方程：
                 * 1. 当前节点不减半，子节点可以减半或不减半，选择最小值
                 * 2. 当前节点减半，子节点不能减半（约束条件）
                 */
                // 当前节点不减半，子节点可以减半或不减半
                dp[u][0] += Math.min(dp[v][0], dp[v][1]);
                // 当前节点减半，子节点不能减半
                dp[u][1] += dp[v][0];
            }
        }
    }

    /**
     * 使用倍增法计算两个节点的最近公共祖先(LCA)
     *
     * @param u 第一个节点
     * @param v 第二个节点
     * @param parent 倍增父节点数组
     * @param depth 深度数组
     * @param LOG 倍增数组大小
```

```
     * @return u 和 v 的最近公共祖先
     */
    private int getLCA(int u, int v, int[][] parent, int[] depth, int LOG) {
        // 确保 u 的深度不小于 v
        if (depth[u] < depth[v]) {
            int temp = u;
            u = v;
            v = temp;
        }

        // 将 u 提升到和 v 同一深度
        for (int j = LOG - 1; j >= 0; j--) {
            if (depth[u] - (1 << j) >= depth[v]) {
                u = parent[u][j];
            }
        }

        // 如果此时 u==v，则找到了 LCA
        if (u == v) return u;

        // 同时向上提升，直到找到 LCA 的直接子节点
        for (int j = LOG - 1; j >= 0; j--) {
            if (parent[u][j] != parent[v][j]) {
                u = parent[u][j];
                v = parent[v][j];
            }
        }

        // 返回它们的父节点作为 LCA
        return parent[u][0];
    }
}

/**
 * 主函数，用于测试
 */
public static void main(String[] args) {
    Solution solution = new Solution();

    // 测试用例 1
    int n1 = 4;
    int[][] edges1 = {{0,1},{1,2},{1,3}};
    int[] price1 = {2,2,10,6};
```

```
        int[][] trips1 = {{0,3},{2,1},{2,3}};
        System.out.println(solution.minimumTotalPrice(n1, edges1, price1, trips1)); // 输出: 23

        // 测试用例 2
        int n2 = 2;
        int[][] edges2 = {{0,1}};
        int[] price2 = {2,2};
        int[][] trips2 = {{0,0}};
        System.out.println(solution.minimumTotalPrice(n2, edges2, price2, trips2)); // 输出: 1
    }
}
```

==========

文件: Code15_LeetCode2646.py

==========

```
from typing import List
from collections import deque

"""
LeetCode 2646. 最小化旅行的价格
题目描述：给定一棵树，每个节点有一个价格。可以选择将某些节点的价格减半。
有多个旅行路径，每个路径从 u 到 v。要求最小化所有旅行路径的总价格。
使用树上差分统计每条边被经过的次数，然后使用树形 DP 决策哪些节点减半。
"""

class Solution:
    def minimumTotalPrice(self, n: int, edges: List[List[int]], price: List[int], trips:
List[List[int]]) -> int:
        # 构建图
        graph = [[] for _ in range(n)]
        for u, v in edges:
            graph[u].append(v)
            graph[v].append(u)

        self.graph = graph
        self.price = price
        self.count = [0] * n  # 节点被经过的次数

        # 预处理 LCA
        LOG = 20
        parent = [[-1] * LOG for _ in range(n)]
        depth = [-1] * n
```

```
        # BFS 预处理深度和父节点
        depth[0] = 0
        queue = deque([0])

        while queue:
            u = queue.popleft()
            for v in graph[u]:
                if depth[v] == -1:
                    depth[v] = depth[u] + 1
                    parent[v][0] = u
                    queue.append(v)

        # 预处理倍增数组
        for j in range(1, LOG):
            for i in range(n):
                if parent[i][j-1] != -1:
                    parent[i][j] = parent[parent[i][j-1]][j-1]

        # 树上差分统计每条边被经过的次数
        for u, v in trips:
            lca = self.getLCA(u, v, parent, depth, LOG)

            self.count[u] += 1
            self.count[v] += 1
            self.count[lca] -= 2

        # DFS 统计每个节点被经过的次数
        self.dfsCount(0, -1)

        # 树形 DP
        self.dp = [[0, 0] for _ in range(n)]
        self.dfsDP(0, -1)

        return min(self.dp[0][0], self.dp[0][1])

    def dfsCount(self, u: int, parent: int):
        for v in self.graph[u]:
            if v != parent:
                self.dfsCount(v, u)
                self.count[u] += self.count[v]

    def dfsDP(self, u: int, parent: int):
```

```
        # 不减半的情况
        self.dp[u][0] = self.price[u] * self.count[u]
        # 减半的情况
        self.dp[u][1] = (self.price[u] // 2) * self.count[u]

        for v in self.graph[u]:
            if v != parent:
                self.dfsDP(v, u)
                # 当前节点不减半，子节点可以减半或不减半
                self.dp[u][0] += min(self.dp[v][0], self.dp[v][1])
                # 当前节点减半，子节点不能减半
                self.dp[u][1] += self.dp[v][0]

    def getLCA(self, u: int, v: int, parent: List[List[int]], depth: List[int], LOG: int) -> int:
        if depth[u] < depth[v]:
            u, v = v, u

        # 将 u 提升到和 v 同一深度
        for j in range(LOG - 1, -1, -1):
            if depth[u] - (1 << j) >= depth[v]:
                u = parent[u][j]

        if u == v:
            return u

        # 同时向上提升
        for j in range(LOG - 1, -1, -1):
            if parent[u][j] != parent[v][j]:
                u = parent[u][j]
                v = parent[v][j]

        return parent[u][0]

# 测试代码
if __name__ == "__main__":
    solution = Solution()

    # 测试用例 1
    n1 = 4
    edges1 = [[0,1],[1,2],[1,3]]
    price1 = [2,2,10,6]
    trips1 = [[0,3],[2,1],[2,3]]
    print(solution.minimumTotalPrice(n1, edges1, price1, trips1))  # 输出: 23
```

```
    # 测试用例 2
    n2 = 2
    edges2 = [[0,1]]
    price2 = [2,2]
    trips2 = [[0,0]]
    print(solution.minimumTotalPrice(n2, edges2, price2, trips2))  # 输出: 1
```

==========

文件: Code16_Codeforces519E.cpp

==========

```
#include <iostream>
#include <vector>
#include <cmath>
#include <algorithm>
using namespace std;

/**
 * Codeforces 519E. A and B and Lecture Rooms
 * 题目描述：给定一棵树，有多个查询，每个查询给出两个节点 u 和 v，
 * 要求找到树上到 u 和 v 距离相等的节点数量。
 * 使用 LCA 和树上差分思想解决。
 */

class Solution {
private:
    vector<vector<int>> graph;
    vector<vector<int>> parent;
    vector<int> depth;
    vector<int> size; // 子树大小
    int LOG;
    int n;

    void dfs(int u, int p) {
        parent[u][0] = p;
        depth[u] = depth[p] + 1;
        size[u] = 1;

        for (int v : graph[u]) {
            if (v != p) {
                dfs(v, u);
                size[u] += size[v];
```

```
        }
    }
}

int getLCA(int u, int v) {
    if (depth[u] < depth[v]) {
        swap(u, v);
    }

    // 将 u 提升到和 v 同一深度
    for (int j = LOG - 1; j >= 0; j--) {
        if (depth[u] - (1 << j) >= depth[v]) {
            u = parent[u][j];
        }
    }

    if (u == v) return u;

    // 同时向上提升
    for (int j = LOG - 1; j >= 0; j--) {
        if (parent[u][j] != parent[v][j]) {
            u = parent[u][j];
            v = parent[v][j];
        }
    }

    return parent[u][0];
}

int getKthParent(int u, int k) {
    for (int j = 0; j < LOG; j++) {
        if (k & (1 << j)) {
            u = parent[u][j];
        }
    }
    return u;
}

int query(int u, int v) {
    if (u == v) return n; // 所有节点都满足

    int lca = getLCA(u, v);
    int dist = depth[u] + depth[v] - 2 * depth[lca];
```

```
        if (dist % 2 == 1) return 0; // 距离为奇数，没有满足的节点

        int midDist = dist / 2;

        if (depth[u] == depth[v]) {
            // u 和 v 在同一深度，中点在 lca
            int uMid = getKthParent(u, midDist - 1);
            int vMid = getKthParent(v, midDist - 1);
            return n - size[uMid] - size[vMid];
        } else {
            // u 和 v 在不同深度，找到中点
            if (depth[u] < depth[v]) {
                swap(u, v);
            }

            int mid = getKthParent(u, midDist);
            int prev = getKthParent(u, midDist - 1);
            return size[mid] - size[prev];
        }
    }

public:
    vector<int> solve(int n, vector<vector<int>>& edges, vector<vector<int>>& queries) {
        this->n = n;
        // 构建图（节点编号从 1 开始）
        graph.resize(n + 1);
        for (auto& edge : edges) {
            int u = edge[0], v = edge[1];
            graph[u].push_back(v);
            graph[v].push_back(u);
        }

        // 预处理
        LOG = log2(n) + 1;
        parent.resize(n + 1, vector<int>(LOG, 0));
        depth.resize(n + 1);
        size.resize(n + 1);

        // DFS 预处理
        dfs(1, 0);

        // 预处理倍增数组
```

```
        for (int j = 1; j < LOG; j++) {
            for (int i = 1; i <= n; i++) {
                if (parent[i][j-1] != 0) {
                    parent[i][j] = parent[parent[i][j-1]][j-1];
                }
            }
        }

        vector<int> result;
        for (auto& q : queries) {
            int u = q[0], v = q[1];
            result.push_back(query(u, v));
        }

        return result;
    }
};

int main() {
    Solution solution;

    // 测试用例
    int n = 4;
    vector<vector<int>> edges = {{1,2}, {1,3}, {2,4}};
    vector<vector<int>> queries = {{1,2}, {2,3}, {3,4}, {2,4}};

    vector<int> result = solution.solve(n, edges, queries);
    for (int res : result) {
        cout << res << " ";
    }
    cout << endl; // 输出: 2 1 1 1

    return 0;
}

==========================================

文件: Code16_Codeforces519E.java
==========================================
package class122;

import java.util.*;
```

```
/**
 * Codeforces 519E. A and B and Lecture Rooms
 *
 * 题目来源：Codeforces
 * 题目链接：https://codeforces.com/contest/519/problem/E
 *
 * 题目描述：
 * 给定一棵树，有多个查询，每个查询给出两个节点 u 和 v，
 * 要求找到树上到 u 和 v 距离相等的节点数量。
 *
 * 算法原理：LCA + 树上几何性质
 * 这是一个结合了 LCA 和树上几何性质的综合问题。
 *
 * 解题思路：
 * 1. 对于两个节点 u 和 v，到它们距离相等的节点满足：dist(x,u) = dist(x,v)
 * 2. 根据树的性质，这些节点的分布有以下几种情况：
 *    - 如果 u==v，则所有节点都满足条件
 *    - 如果 u 和 v 的距离为奇数，则没有满足条件的节点
 *    - 如果 u 和 v 在同一深度，则满足条件的节点数量为 n - size[u 到中点路径上的子节点] - size[v 到中点路径上的子节点]
 *    - 如果 u 和 v 在不同深度，则满足条件的节点数量为中点子树大小 - 中点父节点子树大小
 *
 * 时间复杂度分析：
 * - 建图：O(N)
 * - 预处理 LCA：O(N log N)
 * - 每次查询：O(log N)
 * 总时间复杂度：O(N log N + Q log N)，其中 Q 是查询数
 *
 * 空间复杂度分析：
 * - 图的存储：O(N)
 * - LCA 倍增数组：O(N log N)
 * - 深度和子树大小数组：O(N)
 * 总空间复杂度：O(N log N)
 *
 * 工程化考量：
 * 1. 使用邻接表存储树结构，节省空间
 * 2. 使用倍增法计算 LCA，提高查询效率
 * 3. 预处理子树大小，便于快速计算满足条件的节点数量
 * 4. 边界处理：注意距离为奇数和节点重合的特殊情况
 *
 * 最优解分析：
 * 本解法结合了 LCA 和树上几何性质，是解决此类问题的最优解。
 * 相比于暴力枚举每个节点的 O(N*Q)复杂度，本解法将时间复杂度优化到 O(N log N + Q log N)。
```

```
 */
public class Code16_Codeforces519E {

    static class Solution {
        private List<Integer>[] graph;  // 邻接表存储树结构
        private int[][] parent;         // 倍增数组，parent[i][j]表示节点 i 的 2^j 级祖先
        private int[] depth;            // depth[i]表示节点 i 的深度
        private int[] size;             // size[i]表示以节点 i 为根的子树大小
        private int LOG;                // 倍增数组的最大层数
        private int n;                  // 节点数量

        /**
         * 主函数，解决所有查询
         *
         * @param n 节点数
         * @param edges 树的边
         * @param queries 查询数组
         * @return 每个查询的结果数组
         */
        public int[] solve(int n, int[][] edges, int[][] queries) {
            this.n = n;

            // 构建图
            graph = new ArrayList[n + 1];
            for (int i = 1; i <= n; i++) {
                graph[i] = new ArrayList<>();
            }
            for (int[] edge : edges) {
                int u = edge[0], v = edge[1];
                graph[u].add(v);
                graph[v].add(u);
            }

            // 预处理 LCA 相关数据
            LOG = (int) (Math.log(n) / Math.log(2)) + 1;
            parent = new int[n + 1][LOG];
            depth = new int[n + 1];
            size = new int[n + 1];

            // DFS 预处理深度、父节点和子树大小
            dfs(1, 0);

            // 预处理倍增数组
```

```
        for (int j = 1; j < LOG; j++) {
            for (int i = 1; i <= n; i++) {
                if (parent[i][j-1] != 0) {
                    parent[i][j] = parent[parent[i][j-1]][j-1];
                }
            }
        }

        // 处理所有查询
        int[] result = new int[queries.length];
        for (int i = 0; i < queries.length; i++) {
            int u = queries[i][0], v = queries[i][1];
            result[i] = query(u, v);
        }

        return result;
    }

    /**
     * DFS 预处理深度、父节点和子树大小
     *
     * @param u 当前处理的节点
     * @param p 当前节点的父节点
     */
    private void dfs(int u, int p) {
        // 设置父节点
        parent[u][0] = p;
        // 设置深度
        depth[u] = depth[p] + 1;
        // 初始化子树大小
        size[u] = 1;

        // 遍历当前节点的所有子节点
        for (int v : graph[u]) {
            if (v != p) {
                // 递归处理子节点
                dfs(v, u);
                // 累加子节点的子树大小
                size[u] += size[v];
            }
        }
    }
}
```

```
/**
 * 使用倍增法计算两个节点的最近公共祖先(LCA)
 *
 * @param u 第一个节点
 * @param v 第二个节点
 * @return u 和 v 的最近公共祖先
 */
private int getLCA(int u, int v) {
    // 确保 u 的深度不小于 v
    if (depth[u] < depth[v]) {
        int temp = u;
        u = v;
        v = temp;
    }

    // 将 u 提升到和 v 同一深度
    for (int j = LOG - 1; j >= 0; j--) {
        if (depth[u] - (1 << j) >= depth[v]) {
            u = parent[u][j];
        }
    }

    // 如果此时 u==v，则找到了 LCA
    if (u == v) return u;

    // 同时向上提升，直到找到 LCA 的直接子节点
    for (int j = LOG - 1; j >= 0; j--) {
        if (parent[u][j] != parent[v][j]) {
            u = parent[u][j];
            v = parent[v][j];
        }
    }

    // 返回它们的父节点作为 LCA
    return parent[u][0];
}

/**
 * 获取节点 u 的第 k 个祖先
 *
 * @param u 当前节点
 * @param k 祖先级别
 * @return 节点 u 的第 k 个祖先
```

```
     */
    private int getKthParent(int u, int k) {
        // 二进制拆分 k
        for (int j = 0; j < LOG; j++) {
            // 如果 k 的第 j 位为 1
            if ((k & (1 << j)) != 0) {
                // 节点跳到其 2^j 级祖先
                u = parent[u][j];
            }
        }
        return u;
    }

    /**
     * 查询到 u 和 v 距离相等的节点数量
     *
     * @param u 第一个节点
     * @param v 第二个节点
     * @return 到 u 和 v 距离相等的节点数量
     */
    private int query(int u, int v) {
        // 特殊情况：如果 u 和 v 是同一个节点，则所有节点都满足条件
        if (u == v) return n;

        // 计算 u 和 v 的 LCA
        int lca = getLCA(u, v);
        // 计算 u 和 v 之间的距离
        int dist = depth[u] + depth[v] - 2 * depth[lca];

        // 如果距离为奇数，则没有满足条件的节点
        if (dist % 2 == 1) return 0;

        // 计算中点到 u 和 v 的距离
        int midDist = dist / 2;

        // 如果 u 和 v 在同一深度
        if (depth[u] == depth[v]) {
            /**
             * 当 u 和 v 在同一深度时，满足条件的节点分布：
             * 1. 中点在 lca 上
             * 2. 满足条件的节点数量 = 总节点数 - u 到中点路径上的子树大小 - v 到中点路径上的
子树大小
             */
```

```
            // 获取 u 到中点路径上的子节点
            int uMid = getKthParent(u, midDist - 1);
            // 获取 v 到中点路径上的子节点
            int vMid = getKthParent(v, midDist - 1);
            // 计算满足条件的节点数量
            return n - size[uMid] - size[vMid];
        } else {
            /**
             * 当 u 和 v 在不同深度时，满足条件的节点分布：
             * 1. 中点在深度较大的节点到 lca 的路径上
             * 2. 满足条件的节点数量 = 中点子树大小 - 中点父节点子树大小
             */
            // 确保 u 是深度较大的节点
            if (depth[u] < depth[v]) {
                int temp = u;
                u = v;
                v = temp;
            }

            // 获取中点
            int mid = getKthParent(u, midDist);
            // 获取中点的父节点
            int prev = getKthParent(u, midDist - 1);
            // 计算满足条件的节点数量
            return size[mid] - size[prev];
        }
    }
}

/**
 * 主函数，用于测试
 */
public static void main(String[] args) {
    Solution solution = new Solution();

    // 测试用例
    int n = 4;
    int[][] edges = {{1,2}, {1,3}, {2,4}};
    int[][] queries = {{1,2}, {2,3}, {3,4}, {2,4}};

    int[] result = solution.solve(n, edges, queries);
    System.out.println(Arrays.toString(result)); // 输出: [2, 1, 1, 1]
}
```

```
}

==========================================

文件: Code16_Codeforces519E.py
==========================================
import math
from typing import List

"""
Codeforces 519E. A and B and Lecture Rooms
题目描述：给定一棵树，有多个查询，每个查询给出两个节点 u 和 v，
要求找到树上到 u 和 v 距离相等的节点数量。
使用 LCA 和树上差分思想解决。
"""

class Solution:
    def solve(self, n: int, edges: List[List[int]], queries: List[List[int]]) -> List[int]:
        # 构建图（节点编号从 1 开始）
        graph = [[] for _ in range(n + 1)]
        for u, v in edges:
            graph[u].append(v)
            graph[v].append(u)

        self.graph = graph
        self.n = n

        # 预处理
        LOG = int(math.log2(n)) + 1
        self.parent = [[0] * LOG for _ in range(n + 1)]
        self.depth = [0] * (n + 1)
        self.size = [0] * (n + 1)

        # DFS 预处理
        self.dfs(1, 0)

        # 预处理倍增数组
        for j in range(1, LOG):
            for i in range(1, n + 1):
                if self.parent[i][j-1] != 0:
                    self.parent[i][j] = self.parent[self.parent[i][j-1]][j-1]

        result = []
```

```
    for u, v in queries:
        result.append(self.query(u, v))

    return result

def dfs(self, u: int, p: int):
    self.parent[u][0] = p
    self.depth[u] = self.depth[p] + 1
    self.size[u] = 1

    for v in self.graph[u]:
        if v != p:
            self.dfs(v, u)
            self.size[u] += self.size[v]

def getLCA(self, u: int, v: int) -> int:
    if self.depth[u] < self.depth[v]:
        u, v = v, u

    # 将 u 提升到和 v 同一深度
    for j in range(len(self.parent[0]) - 1, -1, -1):
        if self.depth[u] - (1 << j) >= self.depth[v]:
            u = self.parent[u][j]

    if u == v:
        return u

    # 同时向上提升
    for j in range(len(self.parent[0]) - 1, -1, -1):
        if self.parent[u][j] != self.parent[v][j]:
            u = self.parent[u][j]
            v = self.parent[v][j]

    return self.parent[u][0]

def getKthParent(self, u: int, k: int) -> int:
    for j in range(len(self.parent[0])):
        if k & (1 << j):
            u = self.parent[u][j]
    return u

def query(self, u: int, v: int) -> int:
    if u == v:
```

```
            return self.n  # 所有节点都满足

        lca = self.getLCA(u, v)
        dist = self.depth[u] + self.depth[v] - 2 * self.depth[lca]

        if dist % 2 == 1:
            return 0  # 距离为奇数，没有满足的节点

        midDist = dist // 2

        if self.depth[u] == self.depth[v]:
            # u 和 v 在同一深度，中点在 lca
            uMid = self.getKthParent(u, midDist - 1)
            vMid = self.getKthParent(v, midDist - 1)
            return self.n - self.size[uMid] - self.size[vMid]
        else:
            # u 和 v 在不同深度，找到中点
            if self.depth[u] < self.depth[v]:
                u, v = v, u

            mid = self.getKthParent(u, midDist)
            prev = self.getKthParent(u, midDist - 1)
            return self.size[mid] - self.size[prev]

# 测试代码
if __name__ == "__main__":
    solution = Solution()

    # 测试用例
    n = 4
    edges = [[1,2], [1,3], [2,4]]
    queries = [[1,2], [2,3], [3,4], [2,4]]

    result = solution.solve(n, edges, queries)
    print(result)  # 输出: [2, 1, 1, 1]

==============================================

文件: Code17_LeetCode2096.cpp
==============================================
#include <iostream>
#include <string>
#include <algorithm>
```

```
using namespace std;

/**
 * LeetCode 2096. 从二叉树一个节点到另一个节点的方向
 * 题目描述：给定一棵二叉树，找到从起点节点到目标节点的路径方向。
 * 使用 LCA 和路径重建解决。
 */

struct TreeNode {
    int val;
    TreeNode *left;
    TreeNode *right;
    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
};

class Solution {
public:
    string getDirections(TreeNode* root, int startValue, int destValue) {
        // 找到起点和目标节点的 LCA
        TreeNode* lca = findLCA(root, startValue, destValue);

        // 从起点到 LCA 的路径（全部是 U）
        string startToLCA;
        findPath(lca, startValue, startToLCA);

        // 从 LCA 到目标的路径
        string lcaToDest;
        findPath(lca, destValue, lcaToDest);

        // 起点到 LCA 的路径全部替换为 U
        string upPath(startToLCA.size(), 'U');

        return upPath + lcaToDest;
    }

private:
    TreeNode* findLCA(TreeNode* root, int p, int q) {
        if (root == nullptr || root->val == p || root->val == q) {
            return root;
        }

        TreeNode* left = findLCA(root->left, p, q);
        TreeNode* right = findLCA(root->right, p, q);
```

```
        if (left != nullptr && right != nullptr) {
            return root;
        }

        return left != nullptr ? left : right;
    }

    bool findPath(TreeNode* node, int target, string& path) {
        if (node == nullptr) return false;

        if (node->val == target) return true;

        // 尝试左子树
        path.push_back('L');
        if (findPath(node->left, target, path)) {
            return true;
        }
        path.pop_back();

        // 尝试右子树
        path.push_back('R');
        if (findPath(node->right, target, path)) {
            return true;
        }
        path.pop_back();

        return false;
    }
};

int main() {
    Solution solution;

    // 测试用例 1
    TreeNode* root1 = new TreeNode(5);
    root1->left = new TreeNode(1);
    root1->right = new TreeNode(2);
    root1->left->left = new TreeNode(3);
    root1->right->left = new TreeNode(6);
    root1->right->right = new TreeNode(4);

    cout << solution.getDirections(root1, 3, 6) << endl; // 输出: "UURL"
```

```
    // 测试用例 2
    TreeNode* root2 = new TreeNode(2);
    root2->left = new TreeNode(1);

    cout << solution.getDirections(root2, 2, 1) << endl; // 输出: "L"

    return 0;
}
```

==========

文件: Code17_LeetCode2096.java

==========

```
package class122;

import java.util.*;

/**
 * LeetCode 2096. 从二叉树一个节点到另一个节点的方向
 *
 * 题目来源：LeetCode
 * 题目链接：https://leetcode.cn/problems/step-by-step-directions-from-a-binary-tree-node-to-another/
 *
 * 题目描述：
 * 给定一棵二叉树，找到从起点节点到目标节点的路径方向。
 * 路径方向用字符串表示，其中'L'表示向左移动，'R'表示向右移动，'U'表示向上移动到父节点。
 *
 * 算法原理：LCA + 路径重建
 * 这是一个结合了 LCA 和路径重建的二叉树问题。
 *
 * 解题思路：
 * 1. 首先找到起点节点和目标节点的最近公共祖先(LCA)
 * 2. 然后分别找到从 LCA 到起点和从 LCA 到目标节点的路径
 * 3. 起点到 LCA 的路径全部转换为'U'（向上移动）
 * 4. LCA 到目标节点的路径保持不变
 * 5. 将两段路径拼接即为最终结果
 *
 * 时间复杂度分析：
 * - 找 LCA：O(N)
 * - 找路径：O(N)
 * 总时间复杂度：O(N)
```

```
 *
 * 空间复杂度分析：
 * - 递归栈空间：O(H)，其中 H 是树的高度
 * - 路径字符串：O(N)
 * 总空间复杂度：O(N)
 *
 * 工程化考量：
 * 1. 使用 StringBuilder 提高字符串拼接效率
 * 2. 递归实现简洁明了，但需注意栈溢出问题
 * 3. 路径查找采用回溯法，确保找到正确的路径
 *
 * 最优解分析：
 * 本解法是解决此类问题的最优解，时间复杂度为 O(N)。
 * 相比于分别找到两个节点的路径再合并的方法，本解法通过 LCA 避免了重复遍历。
 */
public class Code17_LeetCode2096 {

    /**
     * 二叉树节点定义
     */
    static class TreeNode {
        int val;
        TreeNode left;
        TreeNode right;
        TreeNode(int x) { val = x; }
    }

    static class Solution {
        /**
         * 获取从起点节点到目标节点的方向路径
         *
         * @param root 二叉树根节点
         * @param startValue 起点节点值
         * @param destValue 目标节点值
         * @return 从起点到目标的方向路径字符串
         */
        public String getDirections(TreeNode root, int startValue, int destValue) {
            // 找到起点和目标节点的 LCA
            TreeNode lca = findLCA(root, startValue, destValue);

            // 从 LCA 到起点的路径（全部是向上移动'U'）
            StringBuilder startToLCA = new StringBuilder();
            findPath(lca, startValue, startToLCA);
```

```
    // 从LCA到目标的路径
    StringBuilder lcaToDest = new StringBuilder();
    findPath(lca, destValue, lcaToDest);

    // 起点到LCA的路径全部替换为'U'
    String upPath = "U".repeat(startToLCA.length());

    // 拼接两段路径并返回
    return upPath + lcaToDest.toString();
}

/**
 * 找到两个节点的最近公共祖先(LCA)
 *
 * @param root 当前节点
 * @param p 第一个节点值
 * @param q 第二个节点值
 * @return 两个节点的LCA
 */
private TreeNode findLCA(TreeNode root, int p, int q) {
    // 基本情况：节点为空或找到目标节点
    if (root == null || root.val == p || root.val == q) {
        return root;
    }

    // 在左子树中查找
    TreeNode left = findLCA(root.left, p, q);
    // 在右子树中查找
    TreeNode right = findLCA(root.right, p, q);

    // 如果左右子树都找到了目标节点，则当前节点就是LCA
    if (left != null && right != null) {
        return root;
    }

    // 返回非空的子树结果
    return left != null ? left : right;
}

/**
 * 查找从node节点到target节点的路径
 *
```

```
     * @param node 当前节点
     * @param target 目标节点值
     * @param path 路径字符串
     * @return 是否找到路径
     */
    private boolean findPath(TreeNode node, int target, StringBuilder path) {
        // 基本情况：节点为空
        if (node == null) return false;

        // 找到目标节点
        if (node.val == target) return true;

        // 尝试左子树
        path.append('L');
        if (findPath(node.left, target, path)) {
            return true;
        }
        // 回溯：删除最后添加的字符
        path.deleteCharAt(path.length() - 1);

        // 尝试右子树
        path.append('R');
        if (findPath(node.right, target, path)) {
            return true;
        }
        // 回溯：删除最后添加的字符
        path.deleteCharAt(path.length() - 1);

        return false;
    }
}

/**
 * 主函数，用于测试
 */
public static void main(String[] args) {
    Solution solution = new Solution();

    // 测试用例 1
    TreeNode root1 = new TreeNode(5);
    root1.left = new TreeNode(1);
    root1.right = new TreeNode(2);
    root1.left.left = new TreeNode(3);
```

```
        root1.right.left = new TreeNode(6);
        root1.right.right = new TreeNode(4);

        System.out.println(solution.getDirections(root1, 3, 6)); // 输出: "UURL"

        // 测试用例 2
        TreeNode root2 = new TreeNode(2);
        root2.left = new TreeNode(1);

        System.out.println(solution.getDirections(root2, 2, 1)); // 输出: "L"
    }
}
```

==========

文件: Code17_LeetCode2096.py

==========

```
# Definition for a binary tree node.
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

class Solution:
    def getDirections(self, root: TreeNode, startValue: int, destValue: int) -> str:
        # 找到起点和目标节点的 LCA
        lca = self.findLCA(root, startValue, destValue)

        # 从起点到 LCA 的路径（全部是 U）
        startToLCA = []
        self.findPath(lca, startValue, startToLCA)

        # 从 LCA 到目标的路径
        lcaToDest = []
        self.findPath(lca, destValue, lcaToDest)

        # 起点到 LCA 的路径全部替换为 U
        upPath = 'U' * len(startToLCA)

        return upPath + ''.join(lcaToDest)

    def findLCA(self, root: TreeNode, p: int, q: int) -> TreeNode:
```

```
        if root is None or root.val == p or root.val == q:
            return root

        left = self.findLCA(root.left, p, q)
        right = self.findLCA(root.right, p, q)

        if left is not None and right is not None:
            return root

        return left if left is not None else right

    def findPath(self, node: TreeNode, target: int, path: list) -> bool:
        if node is None:
            return False

        if node.val == target:
            return True

        # 尝试左子树
        path.append('L')
        if self.findPath(node.left, target, path):
            return True
        path.pop()

        # 尝试右子树
        path.append('R')
        if self.findPath(node.right, target, path):
            return True
        path.pop()

        return False

# 测试代码
if __name__ == "__main__":
    solution = Solution()

    # 测试用例 1
    root1 = TreeNode(5)
    root1.left = TreeNode(1)
    root1.right = TreeNode(2)
    root1.left.left = TreeNode(3)
    root1.right.left = TreeNode(6)
    root1.right.right = TreeNode(4)
```

```
    print(solution.getDirections(root1, 3, 6))  # 输出: "UURL"

    # 测试用例 2
    root2 = TreeNode(2)
    root2.left = TreeNode(1)

    print(solution.getDirections(root2, 2, 1))  # 输出: "L"
```

==========