==========
文件夹: class152_TreeDiameter
==========

[Markdown 文件]
==========
文件: README.md
==========
树的直径算法专题 (Class121)

概述

本目录包含树的直径相关算法的完整实现，涵盖了从基础概念到高级应用的各个层面。树的直径是指树中任意两点之间最长的简单路径，是一类经典的图论问题，在算法竞赛和实际应用中都有广泛的应用。

核心算法

1. 两次 DFS/BFS 法
- **适用场景**：边权非负的树
- **时间复杂度**：O(n)
- **空间复杂度**：O(n)
- **实现文件**：
  - [Code01_Diameter1.java](file:///d:/Upan/src/algorithm-journey/src/algorithm-journey/src/class121/Code01_Diameter1.java) - 基础实现
  - [SPOJ_PT07Z_LongestPathInTree.java](file:///d:/Upan/src/algorithm-journey/src/algorithm-journey/src/class121/SPOJ_PT07Z_LongestPathInTree.java) - SPOJ 题目实现
  - [CSES1131_TreeDiameter.java](file:///d:/Upan/src/algorithm-journey/src/algorithm-journey/src/class121/CSES1131_TreeDiameter.java) - CSES 题目实现

2. 树形动态规划法
- **适用场景**：可以处理负权边的树
- **时间复杂度**：O(n)
- **空间复杂度**：O(n)
- **实现文件**：
  - [Code01_Diameter2.java](file:///d:/Upan/src/algorithm-journey/src/algorithm-journey/src/class121/Code01_Diameter2.java) - 标准实现
  - [Code01_Diameter3.java](file:///d:/Upan/src/algorithm-journey/src/algorithm-journey/src/class121/Code01_Diameter3.java) - 简化实现

题目分类

基础题目
1. **LeetCode 543. 二叉树的直径**

   - 文件：[LeetCode543_DiameterOfBinaryTree.java](file:///d:/Upan/src/algorithm-journey/src/algorithm-journey/src/class121/LeetCode543_DiameterOfBinaryTree.java) / [LeetCode543_DiameterOfBinaryTree.py](file:///d:/Upan/src/algorithm-journey/src/algorithm-journey/src/class121/LeetCode543_DiameterOfBinaryTree.py) / [LeetCode543_DiameterOfBinaryTree.cpp](file:///d:/Upan/src/algorithm-journey/src/algorithm-journey/src/class121/LeetCode543_DiameterOfBinaryTree.cpp)
   - 难度：简单

2. **SPOJ PT07Z - Longest path in a tree**
   - 文件：[SPOJ_PT07Z_LongestPathInTree.java](file:///d:/Upan/src/algorithm-journey/src/algorithm-journey/src/class121/SPOJ_PT07Z_LongestPathInTree.java) / [SPOJ_PT07Z_LongestPathInTree.py](file:///d:/Upan/src/algorithm-journey/src/algorithm-journey/src/class121/SPOJ_PT07Z_LongestPathInTree.py) / [SPOJ_PT07Z_LongestPathInTree.cpp](file:///d:/Upan/src/algorithm-journey/src/algorithm-journey/src/class121/SPOJ_PT07Z_LongestPathInTree.cpp)
   - 难度：简单

3. **CSES 1131 - Tree Diameter**
   - 文件：[CSES1131_TreeDiameter.java](file:///d:/Upan/src/algorithm-journey/src/algorithm-journey/src/class121/CSES1131_TreeDiameter.java) / [CSES1131_TreeDiameter.py](file:///d:/Upan/src/algorithm-journey/src/algorithm-journey/src/class121/CSES1131_TreeDiameter.py) / [CSES1131_TreeDiameter.cpp](file:///d:/Upan/src/algorithm-journey/src/algorithm-journey/src/class121/CSES1131_TreeDiameter.cpp)
   - 难度：简单

4. **51Nod-2602 - 树的直径**
   - 文件：[Nod2602_TreeDiameter.java](file:///d:/Upan/src/algorithm-journey/src/algorithm-journey/src/class121/Nod2602_TreeDiameter.java) / [Nod2602_TreeDiameter.py](file:///d:/Upan/src/algorithm-journey/src/algorithm-journey/src/class121/Nod2602_TreeDiameter.py) / [Nod2602_TreeDiameter.cpp](file:///d:/Upan/src/algorithm-journey/src/algorithm-journey/src/class121/Nod2602_TreeDiameter.cpp)
   - 难度：简单

5. **洛谷 U81904 - 树的直径**
   - 文件：[LuoguU81904_TreeDiameter.java](file:///d:/Upan/src/algorithm-journey/src/algorithm-journey/src/class121/LuoguU81904_TreeDiameter.java) / [LuoguU81904_TreeDiameter.py](file:///d:/Upan/src/algorithm-journey/src/algorithm-journey/src/class121/LuoguU81904_TreeDiameter.py) / [LuoguU81904_TreeDiameter.cpp](file:///d:/Upan/src/algorithm-journey/src/algorithm-journey/src/class121/LuoguU81904_TreeDiameter.cpp)
   - 难度：简单

进阶题目

1. **AtCoder ABC221F - Diameter Set**
   - 文件：[AtCoderABC221F_DiameterSet.java](file:///d:/Upan/src/algorithm-journey/src/algorithm-journey/src/class121/AtCoderABC221F_DiameterSet.java) / [AtCoderABC221F_DiameterSet.py](file:///d:/Upan/src/algorithm-journey/src/algorithm-journey/src/class121/AtCoderABC221F_DiameterSet.py) / [AtCoderABC221F_DiameterSet.cpp](file:///d:/Upan/src/algorithm-journey/src/algorithm-journey/src/class121/AtCoderABC221F_DiameterSet.cpp)
   - 难度：困难
   - 涉及算法：组合数学、快速幂

2. **Codeforces 1499F - Diameter Cuts**
   - 文件：[Codeforces1499F_DiameterCuts.java](file:///d:/Upan/src/algorithm-journey/src/algorithm-journey/src/class121/Codeforces1499F_DiameterCuts.java) / [Codeforces1499F_DiameterCuts.py](file:///d:/Upan/src/algorithm-journey/src/algorithm-journey/src/class121/Codeforces1499F_DiameterCuts.py) / [Codeforces1499F_DiameterCuts.cpp](file:///d:/Upan/src/algorithm-journey/src/algorithm-journey/src/class121/Codeforces1499F_DiameterCuts.cpp)
   - 难度：困难
   - 涉及算法：树形 DP

3. **LeetCode 1245. Tree Diameter**
   - 文件：[LeetCode1245_TreeDiameter.java](file:///d:/Upan/src/algorithm-journey/src/algorithm-journey/src/class121/LeetCode1245_TreeDiameter.java) / [LeetCode1245_TreeDiameter.py](file:///d:/Upan/src/algorithm-journey/src/algorithm-journey/src/class121/LeetCode1245_TreeDiameter.py) / [LeetCode1245_TreeDiameter.cpp](file:///d:/Upan/src/algorithm-journey/src/algorithm-journey/src/class121/LeetCode1245_TreeDiameter.cpp)
   - 难度：中等

4. **LeetCode 1522. Diameter of N-Ary Tree**
   - 文件：[LeetCode1522_DiameterOfNAryTree.java](file:///d:/Upan/src/algorithm-journey/src/algorithm-journey/src/class121/LeetCode1522_DiameterOfNAryTree.java) / [LeetCode1522_DiameterOfNAryTree.py](file:///d:/Upan/src/algorithm-journey/src/algorithm-journey/src/class121/LeetCode1522_DiameterOfNAryTree.py) / [LeetCode1522_DiameterOfNAryTree.cpp](file:///d:/Upan/src/algorithm-journey/src/algorithm-journey/src/class121/LeetCode1522_DiameterOfNAryTree.cpp)
   - 难度：简单

5. **LeetCode 1617. Count Subtrees With Max Distance**
   - 文件：[LeetCode1617_CountSubtreesWithMaxDistance.java](file:///d:/Upan/src/algorithm-journey/src/algorithm-journey/src/class121/LeetCode1617_CountSubtreesWithMaxDistance.java)

   - 难度：困难
   - 涉及算法：位运算、BFS

6. **SPOJ MDST - Minimum Diameter Spanning Tree**
   - 文件：[SPOJ_MDST_MinimumDiameterSpanningTree.java](file:///d:/Upan/src/algorithm-journey/src/algorithm-journey/src/class121/SPOJ_MDST_MinimumDiameterSpanningTree.java) / [SPOJ_MDST_MinimumDiameterSpanningTree.py](file:///d:/Upan/src/algorithm-journey/src/algorithm-journey/src/class121/SPOJ_MDST_MinimumDiameterSpanningTree.py) / [SPOJ_MDST_MinimumDiameterSpanningTree.cpp](file:///d:/Upan/src/algorithm-journey/src/algorithm-journey/src/class121/SPOJ_MDST_MinimumDiameterSpanningTree.cpp)
   - 难度：困难
   - 涉及算法：Floyd-Warshall、绝对中心

7. **51Nod-1803 - 森林直径**
   - 文件：[Nod1803_ForestDiameter.java](file:///d:/Upan/src/algorithm-journey/src/algorithm-journey/src/class121/Nod1803_ForestDiameter.java) / [Nod1803_ForestDiameter.py](file:///d:/Upan/src/algorithm-journey/src/algorithm-journey/src/class121/Nod1803_ForestDiameter.py) / [Nod1803_ForestDiameter.cpp](file:///d:/Upan/src/algorithm-journey/src/algorithm-journey/src/class121/Nod1803_ForestDiameter.cpp)
   - 难度：困难

文档资料

1. [TreeDiameterProblems.md](file:///d:/Upan/src/algorithm-journey/src/algorithm-journey/src/class121/TreeDiameterProblems.md) - 树的直径算法详解与题目汇总
2. [TreeDiameterProblemsComplete.md](file:///d:/Upan/src/algorithm-journey/src/algorithm-journey/src/class121/TreeDiameterProblemsComplete.md) - 完整版树的直径算法详解与题目汇总

编译和运行

Java 文件
```bash
javac FileName.java
java FileName
```

Python 文件
```bash
python FileName.py
```

C++文件

```bash
g++ -std=c++11 -o FileName.exe FileName.cpp
./FileName.exe
```

算法复杂度总结

算法	时间复杂度	空间复杂度	适用场景
两次 DFS/BFS 法	O(n)	O(n)	边权非负的树
树形 DP 法	O(n)	O(n)	可以处理负权边的树
树形 DP（Codeforces 1499F）	O(n^2)	O(n^2)	计数问题
最小直径生成树	O(n^3)	O(n^2)	无向图

学习建议

1. **初学者**：从基础题目开始，掌握两次 DFS/BFS 法和树形 DP 法的基本思想
2. **进阶学习者**：尝试解决进阶题目，理解如何将树的直径算法与其他算法结合
3. **高手**：挑战困难题目，掌握优化技巧和高级应用

相关算法

- 树的中心
- 树的重心
- 最近公共祖先(LCA)
- 树上差分
- 虚树

==========

文件: TreeDiameterProblems.md
==========
树的直径算法详解与题目汇总

1. 树的直径定义与性质

树的直径是指树中任意两点之间最长的简单路径。一棵树可以有多条直径，但它们的长度相等。

树的直径重要性质：
1. 所有直径都交于一点（对于有权树，交点可能在边上）
2. 所有直径的交点为直径的中点

2. 求解树的直径的常用方法

方法一：两次 DFS/BFS 法
1. 从任意一点开始，找到距离它最远的点 s
2. 从 s 开始，找到距离它最远的点 t
3. s 到 t 的距离即为树的直径

这种方法的时间复杂度为 O(n)，适用于边权非负的情况。

方法二：树形 DP 法
通过一次 DFS，在每个节点计算经过该节点的最长路径。这种方法可以处理负权边的情况。

3. 树的直径相关题目详解

3.0 树的直径基础实现
在 class121 中，我们已经实现了多种树的直径算法：

1. **两次 DFS 法**：适用于边权非负的情况，通过两次深度优先搜索找到树的直径
2. **树形 DP 法**：可以处理负权边的情况，通过一次 DFS 计算每个节点子树的最大深度
3. **迭代实现**：避免递归深度过大导致的栈溢出问题

这些基础实现为我们解决更复杂的树的直径相关问题提供了坚实的基础。

3.1 LeetCode 543. 二叉树的直径
题目描述：给定一棵二叉树，你需要计算它的直径长度。一棵二叉树的直径长度是任意两个结点路径长度中的最大值。这条路径可能穿过也可能不穿过根结点。

解题思路：
1. 对于每个节点，计算其左子树和右子树的最大深度
2. 经过该节点的最长路径就是左子树最大深度+右子树最大深度
3. 遍历所有节点，取最大值

实现要点：
- 使用递归方法同时计算子树深度和更新最大直径
- 注意直径是边的数量，不是节点数量
- 时间复杂度 O(n)，空间复杂度 O(h)，其中 h 是树的高度

3.2 SPOJ PT07Z - Longest path in a tree
题目描述：给定一个无权无向树，求树中最长路径的长度。

解题思路：
使用两次 BFS/DFS 法求解树的直径。

实现要点：

- 使用 BFS 实现，避免递归深度问题
- 第一次 BFS 找到直径的一个端点，第二次 BFS 找到另一个端点
- 时间复杂度 O(n)，空间复杂度 O(n)

3.3 洛谷 P3304 - 直径
题目描述：给定一棵树，边权都为正，求所有直径的公共部分有几条边。

解题思路：
1. 先求出树的直径
2. 找到所有直径的公共部分
3. 计算公共部分的边数

3.4 洛谷 P2195 - 造公园
题目描述：有 n 个节点，可能形成多个连通区，每个连通区保证是树结构。支持查询某个节点到最远节点的距离，以及连接两个连通区的操作。

解题思路：
1. 使用并查集维护连通性
2. 对每个连通块维护其直径
3. 合并两个连通块时，重新计算合并后的直径

3.5 洛谷 P3629 - 巡逻
题目描述：给定一棵树，每天需要遍历所有边并回到起点。可以新建 k 条边来减少遍历次数。

解题思路：
1. k=1 时，添加一条边可以减少直径长度的遍历次数
2. k=2 时，需要考虑两次优化的叠加效果

3.6 洛谷 P2491 - 消防
题目描述：在树上选择一条长度不超过 s 的路径，在路径上建立消防站，使最远居民到消防站的距离最小。

解题思路：
1. 先求出树的直径
2. 在直径上使用滑动窗口技术找到最优路径
3. 计算最远距离的最小值

3.7 Codeforces 1499F - Diameter Cuts
题目描述：给定一棵树和一个整数 k，计算有多少个连通子图的直径恰好为 k。

解题思路：
使用树形 DP，对每个节点计算子树中满足条件的连通子图数量。

实现要点：
- 使用树形DP统计每个节点子树中不同直径的连通子图数量
- 合并子树时考虑连接后的直径变化
- 注意取模运算避免整数溢出
- 时间复杂度O(n*k)，空间复杂度O(n*k)

3.8 Codeforces 804D - Expected diameter of a tree
题目描述：给定多个树，随机选择两个树的节点连接，求连接后新树的直径期望。

解题思路：
1. 对每棵树预处理其所有可能的直径
2. 计算连接后的期望直径

实现要点：
- 预处理每棵树的直径信息以提高查询效率
- 使用数学期望的线性性质简化计算
- 注意处理不同树之间连接后直径的变化情况

3.9 CSES 1131 - Tree Diameter
题目描述：给定一棵树，求树的直径（树中任意两点间最长的简单路径）。

解题思路：
使用两次BFS/DFS法求解树的直径。

实现要点：
- 使用BFS实现，避免递归深度问题
- 第一次BFS找到直径的一个端点，第二次BFS找到另一个端点
- 时间复杂度O(n)，空间复杂度O(n)

3.10 AtCoder ABC221F - Diameter Set
题目描述：给定一棵N个顶点的树，顶点编号为1到N。选择两个或更多顶点并将其涂成红色的方法数是多少，使得红色顶点之间的最大距离等于树的直径？答案对998244353取模。

解题思路：
1. 计算树的直径
2. 根据直径的奇偶性分情况讨论
3. 对于偶数直径，有一个中心点；对于奇数直径，有一个中心边
4. 使用组合数学计算满足条件的方案数

实现要点：
- 使用BFS计算树的直径
- 根据直径奇偶性分别处理
- 使用快速幂和组合数学计算方案数

- 注意取模运算避免整数溢出
- 时间复杂度O(n)，空间复杂度O(n)

3.11 Codeforces 1499F - Diameter Cuts
题目描述：给定一棵n个节点的树和一个整数k，计算有多少个连通子图的直径恰好为k。

解题思路：
使用树形DP，对每个节点计算子树中满足条件的连通子图数量。

实现要点：
- 使用树形DP统计每个节点子树中不同直径的连通子图数量
- 合并子树时考虑连接后的直径变化
- 注意取模运算避免整数溢出
- 时间复杂度O(n*k)，空间复杂度O(n*k)

3.12 SPOJ MDST - Minimum Diameter Spanning Tree
题目描述：给定一个简单无向图G的邻接顶点列表，找到最小直径生成树T，并输出该树的直径diam(T)。

解题思路：
1. 使用Floyd-Warshall算法计算所有点对之间的最短距离
2. 通过绝对中心找到最小直径生成树
3. 绝对中心可以是节点或边上的点
4. 枚举所有可能的中心，计算对应的生成树直径

实现要点：
- 使用Floyd-Warshall算法预处理所有点对最短距离
- 分别考虑节点作为中心和边作为中心的情况
- 时间复杂度O(n^3)，空间复杂度O(n^2)

3.13 51Nod-2602 - 树的直径
题目描述：一棵树的直径就是这棵树上存在的最长路径。现在有一棵n个节点的树，现在想知道这棵树的直径包含的边的个数是多少？

输入：
- 第1行：一个整数n，表示树上的节点个数。(1<=n<=100000)
- 第2-n行：每行有两个整数u,v,表示u与v之间有一条路径。(1<=u,v<=n)

输出：输出一个整数，表示这棵树直径所包含的边的个数。

解题思路：
使用两次BFS/DFS法求解树的直径。第一次BFS/DFS找到距离任意节点最远的节点u，第二次BFS/DFS找到距离u最远的节点v，u到v的路径就是树的直径，路径上的边数即为所求。

实现要点：
- 由于 n 可以达到 1e5，需要使用高效的邻接表存储
- 使用 BFS 避免递归深度过大导致的栈溢出
- 时间复杂度 O(n)，空间复杂度 O(n)

3.14 51Nod-1803 - 森林直径
题目描述：有一棵 n 个结点的树，按顺序给出树边(fa(i),i)，Q 次询问查询如果只选取第(l,r)条树边，问森林的直径。

输入：
- 树边生成方式：fa(i)=rand()mod(i-1)+1
- n,Q<=5*10^5

解题思路：
1. 分析树的结构和查询的特点
2. 对于每个查询，确定森林中的各个连通块
3. 计算每个连通块的直径，取最大值作为森林的直径

实现要点：
- 由于数据规模较大，需要使用高效的算法
- 利用离线处理和单调队列等优化方法
- 时间复杂度和空间复杂度需要优化到 O(n log n)或更好

3.15 洛谷 U81904 - 树的直径
题目描述：给定一棵树，树中每条边都有一个权值，树中两点之间的距离定义为连接两点的路径边权之和。树中最远的两个节点之间的距离被称为树的直径，连接这两点的路径被称为树的最长链。现在让你求出树的最长链的距离。

输入格式：
- 第一行为一个正整数 n，表示这颗树有 n 个节点
- 接下来的 n−1 行，每行三个正整数 u,v,w，表示 u,v（u,v<=n）有一条权值为 w 的边相连

输出格式：输入仅一行，表示树的最长链的距离

解题思路：
使用树形 DP 法求解，因为边权可能为负。对于每个节点，维护两个值：
1. 该节点到其子树中的最长路径长度
2. 该节点到其子树中的次长路径长度
那么，经过该节点的最长路径就是这两个值的和。遍历所有节点，取最大值即为树的直径。

实现要点：
- 使用邻接表存储树结构
- 处理负权边的情况

- 时间复杂度O(n)，空间复杂度O(n)

4. 算法复杂度分析

时间复杂度：
- 两次DFS/BFS法：O(n)
- 树形DP法：O(n)

空间复杂度：
- 两次DFS/BFS法：O(n)
- 树形DP法：O(n)

其中n为树中节点的数量。

5. 工程化考量

异常处理：
1. 空树或单节点树的特殊处理
2. 负权边的处理（使用树形DP）

性能优化：
1. 使用邻接表存储图结构
2. 避免重复计算
3. 使用迭代代替递归防止栈溢出

代码可读性：
1. 添加详细注释
2. 使用有意义的变量名
3. 模块化设计

6. 与机器学习等领域的联系

树的直径算法在以下领域有应用：
1. 网络分析：计算网络中最远节点间的距离
2. 社交网络：分析人际关系的最长路径
3. 生物信息学：分析蛋白质结构或基因树的特性
4. 电路设计：优化芯片布局中的最长信号路径

7. 跨语言实现差异

Java:
- 使用对象和类组织代码
- 自动垃圾回收

- 强类型检查

C++:
- 更低的内存开销
- 手动内存管理
- 指针操作更灵活

Python:
- 代码简洁易读
- 动态类型
- 丰富的内置数据结构

8. 调试与测试

调试技巧：
1. 打印中间结果验证算法正确性
2. 使用断言检查关键变量
3. 构造边界测试用例

测试用例：
1. 空树或单节点树
2. 链状树
3. 星状树
4. 复杂结构树

9. 总结

树的直径是一类经典的图论问题，在算法竞赛和实际应用中都有广泛的应用。掌握树的直径相关算法，不仅有助于解决具体问题，还能加深对树形结构和动态规划等算法思想的理解。通过本文的介绍和代码实现，希望能帮助读者更好地掌握这一重要算法。

==========

文件: TreeDiameterProblemsComplete.md
==========
树的直径算法详解与题目汇总（完整版）

1. 树的直径定义与性质

树的直径是指树中任意两点之间最长的简单路径。一棵树可以有多条直径，但它们的长度相等。

树的直径重要性质：
1. 所有直径都交于一点（对于有权树，交点可能在边上）

2. 所有直径的交点为直径的中点

2. 求解树的直径的常用方法

方法一：两次 DFS/BFS 法
1. 从任意一点开始，找到距离它最远的点 s
2. 从 s 开始，找到距离它最远的点 t
3. s 到 t 的距离即为树的直径

这种方法的时间复杂度为 O(n)，适用于边权非负的情况。

方法二：树形 DP 法
通过一次 DFS，在每个节点计算经过该节点的最长路径。这种方法可以处理负权边的情况。

3. 树的直径相关题目详解

3.0 树的直径基础实现
在 class121 中，我们已经实现了多种树的直径算法：

1. **两次 DFS 法**：适用于边权非负的情况，通过两次深度优先搜索找到树的直径
2. **树形 DP 法**：可以处理负权边的情况，通过一次 DFS 计算每个节点子树的最大深度
3. **迭代实现**：避免递归深度过大导致的栈溢出问题

这些基础实现为我们解决更复杂的树的直径相关问题提供了坚实的基础。

3.1 LeetCode 543. 二叉树的直径
- **题目链接**：https://leetcode.com/problems/diameter-of-binary-tree/
- **题目描述**：给定一棵二叉树，你需要计算它的直径长度。一棵二叉树的直径长度是任意两个结点路径长度中的最大值。这条路径可能穿过也可能不穿过根结点。
- **解题思路**：
  1. 对于每个节点，计算其左子树和右子树的最大深度
  2. 经过该节点的最长路径就是左子树最大深度+右子树最大深度
  3. 遍历所有节点，取最大值
- **实现要点**：
  - 使用递归方法同时计算子树深度和更新最大直径
  - 注意直径是边的数量，不是节点数量
  - 时间复杂度 O(n)，空间复杂度 O(h)，其中 h 是树的高度
- **相关文件**：
  - [LeetCode543_DiameterOfBinaryTree.java](file:///d:/Upan/src/algorithm-journey/src/algorithm-journey/src/class121/LeetCode543_DiameterOfBinaryTree.java)
  - [LeetCode543_DiameterOfBinaryTree.py](file:///d:/Upan/src/algorithm-journey/src/algorithm-journey/src/class121/LeetCode543_DiameterOfBinaryTree.py)
  - [LeetCode543_DiameterOfBinaryTree.cpp](file:///d:/Upan/src/algorithm-journey/src/algorithm-

journey/src/class121/LeetCode543_DiameterOfBinaryTree.cpp)

3.2 SPOJ PT07Z - Longest path in a tree
- **题目链接**：https://www.spoj.com/problems/PT07Z/
- **题目描述**：给定一个无权无向树，求树中最长路径的长度。
- **解题思路**：使用两次 BFS/DFS 法求解树的直径。
- **实现要点**：
  - 使用 BFS 实现，避免递归深度问题
  - 第一次 BFS 找到直径的一个端点，第二次 BFS 找到另一个端点
  - 时间复杂度 O(n)，空间复杂度 O(n)
- **相关文件**：
  - [SPOJ_PT07Z_LongestPathInTree.java](file:///d:/Upan/src/algorithm-journey/src/algorithm-journey/src/class121/SPOJ_PT07Z_LongestPathInTree.java)
  - [SPOJ_PT07Z_LongestPathInTree.py](file:///d:/Upan/src/algorithm-journey/src/algorithm-journey/src/class121/SPOJ_PT07Z_LongestPathInTree.py)
  - [SPOJ_PT07Z_LongestPathInTree.cpp](file:///d:/Upan/src/algorithm-journey/src/algorithm-journey/src/class121/SPOJ_PT07Z_LongestPathInTree.cpp)

3.3 CSES 1131 - Tree Diameter
- **题目链接**：https://cses.fi/problemset/task/1131/
- **题目描述**：给定一棵树，树中每条边都有一个权值，树中两点之间的距离定义为连接两点的路径边权之和。树中最远的两个节点之间的距离被称为树的直径，连接这两点的路径被称为树的最长链。现在让你求出树的最长链的距离。
- **解题思路**：使用两次 BFS/DFS 法求解树的直径。
- **实现要点**：
  - 使用 BFS 实现，避免递归深度问题
  - 第一次 BFS 找到直径的一个端点，第二次 BFS 找到另一个端点
  - 时间复杂度 O(n)，空间复杂度 O(n)
- **相关文件**：
  - [CSES1131_TreeDiameter.java](file:///d:/Upan/src/algorithm-journey/src/algorithm-journey/src/class121/CSES1131_TreeDiameter.java)
  - [CSES1131_TreeDiameter.py](file:///d:/Upan/src/algorithm-journey/src/algorithm-journey/src/class121/CSES1131_TreeDiameter.py)
  - [CSES1131_TreeDiameter.cpp](file:///d:/Upan/src/algorithm-journey/src/algorithm-journey/src/class121/CSES1131_TreeDiameter.cpp)

3.4 51Nod-2602 - 树的直径
- **题目链接**：https://www.51nod.com/Challenge/Problem.html#!#problemId=2602
- **题目描述**：一棵树的直径就是这棵树上存在的最长路径。现在有一棵 n 个节点的树，现在想知道这棵树的直径包含的边的个数是多少？
- **输入**：
  - 第 1 行：一个整数 n，表示树上的节点个数。(1<=n<=100000)
  - 第 2-n 行：每行有两个整数 u,v,表示 u 与 v 之间有一条路径。(1<=u,v<=n)

- **输出**：输出一个整数，表示这棵树直径所包含的边的个数。
- **解题思路**：使用两次 BFS/DFS 法求解树的直径。第一次 BFS/DFS 找到距离任意节点最远的节点 u，第二次 BFS/DFS 找到距离 u 最远的节点 v，u 到 v 的路径就是树的直径，路径上的边数即为所求。
- **实现要点**：
  - 由于 n 可以达到 1e5，需要使用高效的邻接表存储
  - 使用 BFS 避免递归深度过大导致的栈溢出
  - 时间复杂度 O(n)，空间复杂度 O(n)
- **相关文件**：
  - [Nod2602_TreeDiameter.java](file:///d:/Upan/src/algorithm-journey/src/algorithm-journey/src/class121/Nod2602_TreeDiameter.java)
  - [Nod2602_TreeDiameter.py](file:///d:/Upan/src/algorithm-journey/src/algorithm-journey/src/class121/Nod2602_TreeDiameter.py)
  - [Nod2602_TreeDiameter.cpp](file:///d:/Upan/src/algorithm-journey/src/algorithm-journey/src/class121/Nod2602_TreeDiameter.cpp)

3.5 洛谷 U81904 - 树的直径
- **题目链接**：https://www.luogu.com.cn/problem/U81904
- **题目描述**：给定一棵树，树中每条边都有一个权值，树中两点之间的距离定义为连接两点的路径边权之和。树中最远的两个节点之间的距离被称为树的直径，连接这两点的路径被称为树的最长链。现在让你求出树的最长链的距离。
- **输入格式**：
  - 第一行为一个正整数 n，表示这颗树有 n 个节点
  - 接下来的 n-1 行，每行三个正整数 u,v,w，表示 u,v（u,v<=n）有一条权值为 w 的边相连
- **输出格式**：输入仅一行，表示树的最长链的距离
- **解题思路**：使用树形 DP 法求解，因为边权可能为负。对于每个节点，维护两个值：
  1. 该节点到其子树中的最长路径长度
  2. 该节点到其子树中的次长路径长度
  那么，经过该节点的最长路径就是这两个值的和。遍历所有节点，取最大值即为树的直径。
- **实现要点**：
  - 使用邻接表存储树结构
  - 处理负权边的情况
  - 时间复杂度 O(n)，空间复杂度 O(n)
- **相关文件**：
  - [LuoguU81904_TreeDiameter.java](file:///d:/Upan/src/algorithm-journey/src/algorithm-journey/src/class121/LuoguU81904_TreeDiameter.java)
  - [LuoguU81904_TreeDiameter.py](file:///d:/Upan/src/algorithm-journey/src/algorithm-journey/src/class121/LuoguU81904_TreeDiameter.py)
  - [LuoguU81904_TreeDiameter.cpp](file:///d:/Upan/src/algorithm-journey/src/algorithm-journey/src/class121/LuoguU81904_TreeDiameter.cpp)

3.6 AtCoder ABC221F - Diameter Set
- **题目链接**：https://atcoder.jp/contests/abc221/tasks/abc221_f
- **题目描述**：给定一棵 N 个顶点的树，顶点编号为 1 到 N。选择两个或更多顶点并将其涂成红色的方法数是

多少，使得红色顶点之间的最大距离等于树的直径？答案对 998244353 取模。
- **解题思路**：
  1. 计算树的直径
  2. 根据直径的奇偶性分情况讨论
  3. 对于偶数直径，有一个中心点；对于奇数直径，有一个中心边
  4. 使用组合数学计算满足条件的方案数
- **实现要点**：
  - 使用 BFS 计算树的直径
  - 根据直径奇偶性分别处理
  - 使用快速幂和组合数学计算方案数
  - 注意取模运算避免整数溢出
  - 时间复杂度 O(n)，空间复杂度 O(n)
- **相关文件**：
  - [AtCoderABC221F_DiameterSet.java](file:///d:/Upan/src/algorithm-journey/src/algorithm-journey/src/class121/AtCoderABC221F_DiameterSet.java)
  - [AtCoderABC221F_DiameterSet.py](file:///d:/Upan/src/algorithm-journey/src/algorithm-journey/src/class121/AtCoderABC221F_DiameterSet.py)
  - [AtCoderABC221F_DiameterSet.cpp](file:///d:/Upan/src/algorithm-journey/src/algorithm-journey/src/class121/AtCoderABC221F_DiameterSet.cpp)

3.7 Codeforces 1499F - Diameter Cuts
- **题目链接**：https://codeforces.com/problemset/problem/1499/F
- **题目描述**：给定一棵树和一个整数 k，计算有多少个连通子图的直径恰好为 k。
- **解题思路**：使用树形 DP，对每个节点计算子树中满足条件的连通子图数量。
- **实现要点**：
  - 使用树形 DP 统计每个节点子树中不同直径的连通子图数量
  - 合并子树时考虑连接后的直径变化
  - 注意取模运算避免整数溢出
  - 时间复杂度 O(n*k)，空间复杂度 O(n*k)
- **相关文件**：
  - [Codeforces1499F_DiameterCuts.java](file:///d:/Upan/src/algorithm-journey/src/algorithm-journey/src/class121/Codeforces1499F_DiameterCuts.java)
  - [Codeforces1499F_DiameterCuts.py](file:///d:/Upan/src/algorithm-journey/src/algorithm-journey/src/class121/Codeforces1499F_DiameterCuts.py)
  - [Codeforces1499F_DiameterCuts.cpp](file:///d:/Upan/src/algorithm-journey/src/algorithm-journey/src/class121/Codeforces1499F_DiameterCuts.cpp)

3.8 LeetCode 1245. Tree Diameter
- **题目链接**：https://leetcode.com/problems/tree-diameter/
- **题目描述**：给你这棵「无向树」，请你测算并返回它的「直径」：这棵树上最长简单路径的边数。我们用一个由所有「边」组成的数组 edges 来表示一棵无向树，其中 edges[i] = [u, v]表示节点 u 和 v 之间有一条无向边。
- **解题思路**：使用两次 BFS/DFS 法求解树的直径。

- **实现要点**：
  - 使用 BFS 实现，避免递归深度问题
  - 第一次 BFS 找到直径的一个端点，第二次 BFS 找到另一个端点
  - 时间复杂度 O(n)，空间复杂度 O(n)
- **相关文件**：
  - [LeetCode1245_TreeDiameter.java](file:///d:/Upan/src/algorithm-journey/src/algorithm-journey/src/class121/LeetCode1245_TreeDiameter.java)
  - [LeetCode1245_TreeDiameter.py](file:///d:/Upan/src/algorithm-journey/src/algorithm-journey/src/class121/LeetCode1245_TreeDiameter.py)
  - [LeetCode1245_TreeDiameter.cpp](file:///d:/Upan/src/algorithm-journey/src/algorithm-journey/src/class121/LeetCode1245_TreeDiameter.cpp)

3.9 LeetCode 1522. Diameter of N-Ary Tree
- **题目链接**：https://leetcode.com/problems/diameter-of-n-ary-tree/
- **题目描述**：给定一棵 N 叉树的根节点 root，计算这棵树的直径长度。N 叉树的直径指的是树中任意两个节点间路径中最长路径的长度。这条路径可能经过根节点。
- **解题思路**：类似于 LeetCode 543 Diameter of Binary Tree，但此题为 N 叉树。
- **实现要点**：
  - DFS 遍历每个节点
  - 对于每个节点，计算其所有子树中的最大深度和次大深度
  - 经过该节点的最长路径等于最大深度和次大深度之和
  - 时间复杂度 O(n)，空间复杂度 O(h)
- **相关文件**：
  - [LeetCode1522_DiameterOfNAryTree.java](file:///d:/Upan/src/algorithm-journey/src/algorithm-journey/src/class121/LeetCode1522_DiameterOfNAryTree.java)
  - [LeetCode1522_DiameterOfNAryTree.py](file:///d:/Upan/src/algorithm-journey/src/algorithm-journey/src/class121/LeetCode1522_DiameterOfNAryTree.py)
  - [LeetCode1522_DiameterOfNAryTree.cpp](file:///d:/Upan/src/algorithm-journey/src/algorithm-journey/src/class121/LeetCode1522_DiameterOfNAryTree.cpp)

3.10 LeetCode 1617. Count Subtrees With Max Distance
- **题目链接**：https://leetcode.com/problems/count-subtrees-with-max-distance-between-cities/
- **题目描述**：给你一棵 n 个节点的树（节点编号从 1 到 n），求出对于 d 从 1 到 n-1，有多少个子树满足子树中任意两个城市之间的最大距离恰好等于 d。
- **解题思路**：
  1. 枚举所有可能的子树（使用位掩码）
  2. 对于每个子树，判断是否连通
  3. 如果连通，计算该子树的直径
  4. 统计每个直径值对应的子树数量
- **实现要点**：
  - 使用位掩码枚举所有可能的节点组合
  - 使用 BFS/DFS 判断子树连通性
  - 使用两次 BFS/DFS 计算子树直径

  - 时间复杂度O(2^n * n^2)，空间复杂度O(n^2)
- **相关文件**：
  - [LeetCode1617_CountSubtreesWithMaxDistance.java](file:///d:/Upan/src/algorithm-journey/src/algorithm-journey/src/class121/LeetCode1617_CountSubtreesWithMaxDistance.java)

3.11 SPOJ MDST - Minimum Diameter Spanning Tree
- **题目链接**：https://www.spoj.com/problems/MDST/
- **题目描述**：对于给定的邻接顶点列表的图G，找到最小直径生成树T，并写出该树的直径diam(T)。
- **解题思路**：
  1. 使用Floyd-Warshall算法计算所有点对之间的最短距离
  2. 通过绝对中心找到最小直径生成树
  3. 绝对中心可以是节点或边上的点
  4. 枚举所有可能的中心，计算对应的生成树直径
- **实现要点**：
  - 使用Floyd-Warshall算法预处理所有点对最短距离
  - 分别考虑节点作为中心和边作为中心的情况
  - 时间复杂度O(n^3)，空间复杂度O(n^2)
- **相关文件**：
  - [SPOJ_MDST_MinimumDiameterSpanningTree.java](file:///d:/Upan/src/algorithm-journey/src/algorithm-journey/src/class121/SPOJ_MDST_MinimumDiameterSpanningTree.java)
  - [SPOJ_MDST_MinimumDiameterSpanningTree.py](file:///d:/Upan/src/algorithm-journey/src/algorithm-journey/src/class121/SPOJ_MDST_MinimumDiameterSpanningTree.py)
  - [SPOJ_MDST_MinimumDiameterSpanningTree.cpp](file:///d:/Upan/src/algorithm-journey/src/algorithm-journey/src/class121/SPOJ_MDST_MinimumDiameterSpanningTree.cpp)

3.12 51Nod-1803 - 森林直径
- **题目链接**：https://www.51nod.com/Challenge/Problem.html#!#problemId=1803
- **题目描述**：有一棵n个结点的树，按顺序给出树边(fa(i),i)，Q次询问查询如果只选取第(l,r)条树边，问森林的直径。
- **输入**：
  - 树边生成方式：fa(i)=rand()mod(i-1)+1
  - n,Q<=5*10^5
- **解题思路**：
  1. 分析树的结构和查询的特点
  2. 对于每个查询，确定森林中的各个连通块
  3. 计算每个连通块的直径，取最大值作为森林的直径
- **实现要点**：
  - 由于数据规模较大，需要使用高效的算法
  - 利用离线处理和单调队列等优化方法
  - 时间复杂度和空间复杂度需要优化到O(n log n)或更好
- **相关文件**：
  - [Nod1803_ForestDiameter.java](file:///d:/Upan/src/algorithm-journey/src/algorithm-journey/src/class121/Nod1803_ForestDiameter.java)

  - [Nod1803_ForestDiameter.py](file:///d:/Upan/src/algorithm-journey/src/algorithm-journey/src/class121/Nod1803_ForestDiameter.py)
  - [Nod1803_ForestDiameter.cpp](file:///d:/Upan/src/algorithm-journey/src/algorithm-journey/src/class121/Nod1803_ForestDiameter.cpp)

3.13 洛谷 P3304 [SDOI2013]直径
- **题目链接**：https://www.luogu.com.cn/problem/P3304
- **题目描述**：现在小 Q 想知道，对于给定的一棵树，其直径的长度是多少，以及有多少条边满足所有的直径都经过该边。
- **输入格式**：
  - 第一行包含一个整数 N，表示节点数
  - 接下来 N-1 行，每行三个正整数 u,v,w，表示 u,v 之间有一条权值为 w 的边相连
- **输出格式**：
  - 第一行包含一个整数，表示直径的长度
  - 第二行包含一个整数，表示所有直径都经过的边数
- **解题思路**：
  1. 先求出树上任意一条直径
  2. 处理出直径上的点到它的子树（不包含在直径上的点）中最深的距离
  3. 如果这个距离和它到直径一端的距离相同的话，就说明存在不经过该边的直径
  4. 统计所有直径都经过的边数
- **实现要点**：
  - 使用两次 DFS/BFS 求出一条直径
  - 标记直径上的所有边
  - 对于直径上的每个点，计算其子树中的最大深度
  - 判断是否存在不经过某条直径边的其他直径
  - 时间复杂度 O(n)，空间复杂度 O(n)
- **相关文件**：
  - [LuoguP3304_Diameter.java](file:///d:/Upan/src/algorithm-journey/src/algorithm-journey/src/class121/LuoguP3304_Diameter.java)（如果存在）

3.14 Codeforces 804D - Expected diameter of a tree
- **题目链接**：https://codeforces.com/problemset/problem/804/D
- **题目描述**：给定多个树，随机选择两个树的节点连接，求连接后新树的直径期望。
- **解题思路**：
  1. 对每棵树预处理其所有可能的直径
  2. 计算连接后的期望直径
- **实现要点**：
  - 预处理每棵树的直径信息以提高查询效率
  - 使用数学期望的线性性质简化计算
  - 注意处理不同树之间连接后直径的变化情况
- **相关文件**：
  - [Codeforces804D_ExpectedDiameter.java](file:///d:/Upan/src/algorithm-journey/src/algorithm-journey/src/class121/Codeforces804D_ExpectedDiameter.java)（如果存在）

4. 算法复杂度分析

时间复杂度：
- 两次 DFS/BFS 法：O(n)
- 树形 DP 法：O(n)

空间复杂度：
- 两次 DFS/BFS 法：O(n)
- 树形 DP 法：O(n)

其中 n 为树中节点的数量。

5. 工程化考量

异常处理：
1. 空树或单节点树的特殊处理
2. 负权边的处理（使用树形 DP）

性能优化：
1. 使用邻接表存储图结构
2. 避免重复计算
3. 使用迭代代替递归防止栈溢出

代码可读性：
1. 添加详细注释
2. 使用有意义的变量名
3. 模块化设计

6. 与机器学习等领域的联系

树的直径算法在以下领域有应用：
1. 网络分析：计算网络中最远节点间的距离
2. 社交网络：分析人际关系的最长路径
3. 生物信息学：分析蛋白质结构或基因树的特性
4. 电路设计：优化芯片布局中的最长信号路径

7. 跨语言实现差异

Java:
- 使用对象和类组织代码
- 自动垃圾回收
- 强类型检查

C++:
- 更低的内存开销
- 手动内存管理
- 指针操作更灵活

Python:
- 代码简洁易读
- 动态类型
- 丰富的内置数据结构

8. 调试与测试

调试技巧：
1. 打印中间结果验证算法正确性
2. 使用断言检查关键变量
3. 构造边界测试用例

测试用例：
1. 空树或单节点树
2. 链状树
3. 星状树
4. 复杂结构树

9. 总结

树的直径是一类经典的图论问题，在算法竞赛和实际应用中都有广泛的应用。掌握树的直径相关算法，不仅有助于解决具体问题，还能加深对树形结构和动态规划等算法思想的理解。通过本文的介绍和代码实现，希望能帮助读者更好地掌握这一重要算法。

==========

[代码文件]
==========
文件: AtCoderABC221F_DiameterSet.cpp
==========
```
// AtCoder ABC221F Diameter Set
// 题目：给定一棵 N 个顶点的树，顶点编号为 1 到 N。
// 选择两个或更多顶点并将其涂成红色的方法数是多少，
// 使得红色顶点之间的最大距离等于树的直径？
// 答案对 998244353 取模。
// 来源：AtCoder Beginner Contest 221 Problem F
// 链接：https://atcoder.jp/contests/abc221/tasks/abc221_f
```

```
#define MAXN 200001
#define MOD 998244353

// 邻接表存储树
int head[MAXN], next[MAXN << 1], to[MAXN << 1], cnt;
int n;  // 节点数

// 树的直径相关变量
int diameter;  // 树的直径

// DFS 计算子树大小和深度
int subtreeSize[MAXN];
int depth[MAXN];

// 队列实现 BFS
int queue[MAXN], front, rear;
int visited[MAXN];

// 添加边
void addEdge(int u, int v) {
    next[cnt] = head[u];
    to[cnt] = v;
    head[u] = cnt++;
}

// 初始化队列
void initQueue() {
    front = rear = 0;
}

// 入队
void enqueue(int x) {
    queue[rear++] = x;
}

// 出队
int dequeue() {
    return queue[front++];
}

// 判断队列是否为空
int isEmpty() {
```

```
    return front == rear;
}

// 获取队列大小
int queueSize() {
    return rear - front;
}

/**
 * 快速幂运算
 * @param base 底数
 * @param exp 指数
 * @return (base^exp) % MOD
 *
 * 时间复杂度：O(log exp)
 * 空间复杂度：O(1)
 */
long long power(long long base, long long exp) {
    long long result = 1;
    while (exp > 0) {
        if (exp % 2 == 1) {
            result = (result * base) % MOD;
        }
        base = (base * base) % MOD;
        exp /= 2;
    }
    return result;
}

/**
 * BFS 求从起点开始的最远节点和距离
 * @param start 起点
 * @param result 返回结果：[最远节点, 距离]
 *
 * 时间复杂度：O(n)
 * 空间复杂度：O(n)
 */
void bfs(int start, int result[2]) {
    // 初始化访问数组
    for (int i = 1; i <= n; i++) {
        visited[i] = 0;
    }
```

```
    initQueue();

    visited[start] = 1;
    enqueue(start);

    int lastNode = start;
    int maxDistance = 0;

    while (!isEmpty()) {
        int size = queueSize();
        for (int i = 0; i < size; i++) {
            int current = dequeue();
            lastNode = current;

            // 遍历当前节点的所有邻居
            for (int e = head[current]; e != 0; e = next[e]) {
                int neighbor = to[e];
                if (!visited[neighbor]) {
                    visited[neighbor] = 1;
                    enqueue(neighbor);
                }
            }
        }
        if (!isEmpty()) {
            maxDistance++;
        }
    }

    result[0] = lastNode;
    result[1] = maxDistance;
}

/**
 * 使用两次 BFS 法求树的直径
 * @return 树的直径
 *
 * 时间复杂度：O(n)
 * 空间复杂度：O(n)
 */
int findDiameter() {
    int firstBFS[2], secondBFS[2];

    // 第一次 BFS，从节点 1 开始找到最远节点
```

```
    bfs(1, firstBFS);

    // 第二次 BFS，从第一次找到的最远节点开始找到另一个最远节点
    bfs(firstBFS[0], secondBFS);

    // 第二次 BFS 的距离就是树的直径
    return secondBFS[1];
}

/**
 * DFS 计算子树大小
 * @param u 当前节点
 * @param parent 父节点
 *
 * 时间复杂度：O(n)
 * 空间复杂度：O(n)
 */
void dfsSubtreeSize(int u, int parent) {
    subtreeSize[u] = 1;
    for (int e = head[u]; e != 0; e = next[e]) {
        int v = to[e];
        if (v != parent) {
            dfsSubtreeSize(v, u);
            subtreeSize[u] += subtreeSize[v];
        }
    }
}

/**
 * DFS 计算深度
 * @param u 当前节点
 * @param parent 父节点
 * @param d 当前深度
 *
 * 时间复杂度：O(n)
 * 空间复杂度：O(n)
 */
void dfsDepth(int u, int parent, int d) {
    depth[u] = d;
    for (int e = head[u]; e != 0; e = next[e]) {
        int v = to[e];
        if (v != parent) {
            dfsDepth(v, u, d + 1);
```

```
        }
    }
}

/**
 * 计算满足条件的方案数
 * @return 满足条件的方案数
 *
 * 时间复杂度：O(n)
 * 空间复杂度：O(n)
 */
long long solve() {
    // 计算树的直径
    diameter = findDiameter();

    // 特殊情况：直径为0（只有一个节点）
    if (diameter == 0) {
        return 1;  // 只有一种方案：选择这个节点
    }

    // 计算子树大小
    dfsSubtreeSize(1, 0);

    // 计算深度
    dfsDepth(1, 0, 0);

    // 找到深度最大的节点
    int deepestNode = 1;
    for (int i = 2; i <= n; i++) {
        if (depth[i] > depth[deepestNode]) {
            deepestNode = i;
        }
    }

    // 从最深节点再次DFS，找到直径的端点
    dfsDepth(deepestNode, 0, 0);
    int endpoint1 = deepestNode;
    for (int i = 1; i <= n; i++) {
        if (depth[i] > depth[endpoint1]) {
            endpoint1 = i;
        }
    }
```

```
// 从 endpoint1 再次 DFS，找到另一个端点
dfsDepth(endpoint1, 0, 0);
int endpoint2 = 1;
for (int i = 2; i <= n; i++) {
    if (depth[i] > depth[endpoint2]) {
        endpoint2 = i;
    }
}

// 计算满足条件的方案数
// 如果直径是偶数，有一个中心点
// 如果直径是奇数，有一个中心边
if (diameter % 2 == 0) {
    // 直径为偶数，有一个中心点
    // 找到中心点
    int center = 0;
    dfsDepth(endpoint1, 0, 0);
    for (int i = 1; i <= n; i++) {
        if (depth[i] == diameter / 2 &&
            bfs(i, (int[2]){0,0}), ((int[2]){0,0})[1] == diameter / 2) {
            center = i;
            break;
        }
    }

    // 计算以中心点为根的子树中满足条件的方案数
    long long result = 1;
    for (int e = head[center]; e != 0; e = next[e]) {
        int v = to[e];
        // 计算每个子树中满足条件的方案数
        long long subtreeWays = power(2, subtreeSize[v]) - 1;
        result = (result * subtreeWays) % MOD;
    }

    // 至少选择两个节点
    result = (result - 1 + MOD) % MOD;
    return result;
} else {
    // 直径为奇数，有一个中心边
    // 找到中心边的两个端点
    dfsDepth(endpoint1, 0, 0);
    int center1 = 0, center2 = 0;
    for (int i = 1; i <= n; i++) {
```

```
            if (depth[i] == diameter / 2) {
                if (center1 == 0) {
                    center1 = i;
                } else {
                    center2 = i;
                    break;
                }
            }
        }

        // 计算每个部分中满足条件的方案数
        long long ways1 = power(2, subtreeSize[center1]) - 1;
        long long ways2 = power(2, subtreeSize[center2]) - 1;

        long long result = (ways1 * ways2) % MOD;
        return result;
    }
}

/**
 * 主方法
 * 时间复杂度：O(n)
 * 空间复杂度：O(n)
 */
int main() {
    // 由于编译环境限制，这里只展示算法实现
    // 实际使用时需要根据具体环境添加输入输出代码

    // 示例：n = 4, 边为 1-2, 2-3, 3-4
    n = 4;
    cnt = 1;
    for (int i = 1; i <= n; i++) {
        head[i] = 0;
    }

    addEdge(1, 2);
    addEdge(2, 1);
    addEdge(2, 3);
    addEdge(3, 2);
    addEdge(3, 4);
    addEdge(4, 3);

    // 计算并输出结果
```

```
    // printf("%lld\n", solve()); // 应该输出某种结果

    return 0;
}
```

==========

文件: AtCoderABC221F_DiameterSet.java

==========

```
package class121;

// AtCoder ABC221F Diameter Set
// 题目：给定一棵 N 个顶点的树，顶点编号为 1 到 N。
// 选择两个或更多顶点并将其涂成红色的方法数是多少，
// 使得红色顶点之间的最大距离等于树的直径？
// 答案对 998244353 取模。
// 来源：AtCoder Beginner Contest 221 Problem F
// 链接：https://atcoder.jp/contests/abc221/tasks/abc221_f

// 算法标签：树、广度优先搜索、两次 BFS、组合数学、快速幂
// 难度：困难
// 时间复杂度：O(n)，其中 n 是树中节点的数量
// 空间复杂度：O(n)，用于存储邻接表和辅助数组

// 相关题目：
// - LeetCode 543. 二叉树的直径
// - LeetCode 1245. Tree Diameter (无向树的直径)
// - LeetCode 1522. Diameter of N-Ary Tree (N 叉树的直径)
// - SPOJ PT07Z - Longest path in a tree (树中最长路径)
// - CSES 1131 - Tree Diameter (树的直径)
// - 51Nod 2602 - 树的直径
// - 洛谷 U81904 树的直径

// 解题思路：
// 1. 首先计算树的直径
// 2. 根据直径的奇偶性分情况讨论
// 3. 对于偶数直径，有一个中心点；对于奇数直径，有一个中心边
// 4. 使用组合数学计算满足条件的方案数

import java.io.*;
import java.util.*;

public class AtCoderABC221F_DiameterSet {
```

```
static final int MAXN = 200001;
static final int MOD = 998244353;

// 邻接表存储树
static ArrayList<Integer>[] graph;
static int n;  // 节点数

// 树的直径相关变量
static int diameter;  // 树的直径
static int center1, center2;  // 直径的中心点

// DFS 计算子树大小和深度
static int[] subtreeSize;
static int[] depth;

/**
 * 快速幂运算
 *
 * 算法思路：
 * 使用二进制指数法计算(base^exp) % MOD
 *
 * @param base 底数
 * @param exp 指数
 * @return (base^exp) % MOD
 *
 * 时间复杂度：O(log exp)
 * 空间复杂度：O(1)
 */
static long power(long base, long exp) {
    long result = 1;
    while (exp > 0) {
        if (exp % 2 == 1) {
            result = (result * base) % MOD;
        }
        base = (base * base) % MOD;
        exp /= 2;
    }
    return result;
}

/**
 * BFS 求从起点开始的最远节点和距离
```

```
     *
     * 算法思路：
     * 1. 从指定起点开始进行广度优先搜索
     * 2. 记录访问过的节点，避免重复访问
     * 3. 记录每一层的节点，直到遍历完所有节点
     * 4. 返回最后一层的节点（最远节点）和距离
     *
     * @param start 起点
     * @return Pair 对象，包含最远节点和距离
     *
     * 时间复杂度：O(n)
     * 空间复杂度：O(n)
     */
    static int[] bfs(int start) {
        boolean[] visited = new boolean[n + 1];
        Queue<Integer> queue = new LinkedList<>();

        visited[start] = true;
        queue.offer(start);

        int lastNode = start;
        int maxDistance = 0;

        while (!queue.isEmpty()) {
            int size = queue.size();
            for (int i = 0; i < size; i++) {
                int current = queue.poll();
                lastNode = current;

                // 遍历当前节点的所有邻居
                for (int neighbor : graph[current]) {
                    if (!visited[neighbor]) {
                        visited[neighbor] = true;
                        queue.offer(neighbor);
                    }
                }
            }
            if (!queue.isEmpty()) {
                maxDistance++;
            }
        }

        return new int[]{lastNode, maxDistance};
```

```
}

/**
 * 使用两次 BFS 法求树的直径
 *
 * 算法思路：
 * 1. 第一次 BFS，从任意节点（如节点 1）开始找到最远节点
 * 2. 第二次 BFS，从第一次找到的最远节点开始找到另一个最远节点
 * 3. 第二次 BFS 的距离就是树的直径
 *
 * @return 树的直径
 *
 * 时间复杂度：O(n)
 * 空间复杂度：O(n)
 */
static int findDiameter() {
    // 第一次 BFS，从节点 1 开始找到最远节点
    int[] firstBFS = bfs(1);

    // 第二次 BFS，从第一次找到的最远节点开始找到另一个最远节点
    int[] secondBFS = bfs(firstBFS[0]);

    // 第二次 BFS 的距离就是树的直径
    return secondBFS[1];
}

/**
 * DFS 计算子树大小
 *
 * 算法思路：
 * 1. 从指定节点开始进行深度优先搜索
 * 2. 递归计算每个子树的大小
 * 3. 子树大小等于所有子节点子树大小之和加 1（当前节点）
 *
 * @param u 当前节点
 * @param parent 父节点
 *
 * 时间复杂度：O(n)
 * 空间复杂度：O(n)
 */
static void dfsSubtreeSize(int u, int parent) {
    subtreeSize[u] = 1;
    for (int v : graph[u]) {
```

```
        if (v != parent) {
            dfsSubtreeSize(v, u);
            subtreeSize[u] += subtreeSize[v];
        }
    }
}

/**
 * DFS 计算深度
 *
 * 算法思路：
 * 1. 从指定节点开始进行深度优先搜索
 * 2. 递归计算每个节点的深度
 * 3. 节点深度等于父节点深度加 1
 *
 * @param u 当前节点
 * @param parent 父节点
 * @param d 当前深度
 *
 * 时间复杂度：O(n)
 * 空间复杂度：O(n)
 */
static void dfsDepth(int u, int parent, int d) {
    depth[u] = d;
    for (int v : graph[u]) {
        if (v != parent) {
            dfsDepth(v, u, d + 1);
        }
    }
}

/**
 * 计算满足条件的方案数
 *
 * 算法思路：
 * 1. 首先计算树的直径
 * 2. 根据直径的奇偶性分情况讨论
 * 3. 对于偶数直径，有一个中心点；对于奇数直径，有一个中心边
 * 4. 使用组合数学计算满足条件的方案数
 *
 * @return 满足条件的方案数
 *
 * 时间复杂度：O(n)
```

```
 * 空间复杂度：O(n)
 */
static long solve() {
    // 计算树的直径
    diameter = findDiameter();

    // 特殊情况：直径为0（只有一个节点）
    if (diameter == 0) {
        return 1;  // 只有一种方案：选择这个节点
    }

    // 计算子树大小
    subtreeSize = new int[n + 1];
    dfsSubtreeSize(1, 0);

    // 计算深度
    depth = new int[n + 1];
    dfsDepth(1, 0, 0);

    // 找到深度最大的节点
    int deepestNode = 1;
    for (int i = 2; i <= n; i++) {
        if (depth[i] > depth[deepestNode]) {
            deepestNode = i;
        }
    }

    // 从最深节点再次DFS，找到直径的端点
    dfsDepth(deepestNode, 0, 0);
    int endpoint1 = deepestNode;
    for (int i = 1; i <= n; i++) {
        if (depth[i] > depth[endpoint1]) {
            endpoint1 = i;
        }
    }

    // 从endpoint1再次DFS，找到另一个端点
    dfsDepth(endpoint1, 0, 0);
    int endpoint2 = 1;
    for (int i = 2; i <= n; i++) {
        if (depth[i] > depth[endpoint2]) {
            endpoint2 = i;
        }
```

```
}

// 计算满足条件的方案数
// 如果直径是偶数，有一个中心点
// 如果直径是奇数，有一个中心边
if (diameter % 2 == 0) {
    // 直径为偶数，有一个中心点
    // 找到中心点
    int center = 0;
    dfsDepth(endpoint1, 0, 0);
    for (int i = 1; i <= n; i++) {
        if (depth[i] == diameter / 2 &&
            bfs(i)[1] == diameter / 2) {
            center = i;
            break;
        }
    }

    // 计算以中心点为根的子树中满足条件的方案数
    long result = 1;
    for (int v : graph[center]) {
        // 计算每个子树中满足条件的方案数
        long subtreeWays = power(2, subtreeSize[v]) - 1;
        result = (result * subtreeWays) % MOD;
    }

    // 至少选择两个节点
    result = (result - 1 + MOD) % MOD;
    return result;
} else {
    // 直径为奇数，有一个中心边
    // 找到中心边的两个端点
    dfsDepth(endpoint1, 0, 0);
    int center1 = 0, center2 = 0;
    for (int i = 1; i <= n; i++) {
        if (depth[i] == diameter / 2) {
            if (center1 == 0) {
                center1 = i;
            } else {
                center2 = i;
                break;
            }
        }
```

```
        }

        // 计算每个部分中满足条件的方案数
        long ways1 = power(2, subtreeSize[center1]) - 1;
        long ways2 = power(2, subtreeSize[center2]) - 1;

        long result = (ways1 * ways2) % MOD;
        return result;
    }
}

/**
 * 主方法
 *
 * 输入格式：
 * - 第一行包含一个整数 N，表示树中节点的数量
 * - 接下来 N-1 行，每行包含两个整数 u 和 v，表示节点 u 和 v 之间有一条边
 *
 * 输出格式：
 * - 输出一个整数，表示满足条件的方案数，对 998244353 取模
 *
 * 时间复杂度：O(n)
 * 空间复杂度：O(n)
 */
public static void main(String[] args) throws IOException {
    BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
    PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));

    // 读取节点数
    n = Integer.parseInt(br.readLine());

    // 初始化邻接表
    graph = new ArrayList[n + 1];
    for (int i = 1; i <= n; i++) {
        graph[i] = new ArrayList<>();
    }

    // 读取边信息
    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());
        graph[u].add(v);
```

```
            graph[v].add(u);
        }

        // 计算并输出结果
        out.println(solve());
        out.flush();
        out.close();
        br.close();
    }
}
```

==========

文件: AtCoderABC221F_DiameterSet.py

==========

```
# AtCoder ABC221F Diameter Set
# 题目：给定一棵 N 个顶点的树，顶点编号为 1 到 N。
# 选择两个或更多顶点并将其涂成红色的方法数是多少，
# 使得红色顶点之间的最大距离等于树的直径？
# 答案对 998244353 取模。
# 来源：AtCoder Beginner Contest 221 Problem F
# 链接：https://atcoder.jp/contests/abc221/tasks/abc221_f

from collections import deque, defaultdict

MOD = 998244353

class AtCoderABC221FDiameterSet:
    def __init__(self):
        self.n = 0  # 节点数
        self.graph = defaultdict(list)  # 邻接表存储树
        self.diameter = 0  # 树的直径

    def add_edge(self, u, v):
        """添加无向边"""
        self.graph[u].append(v)
        self.graph[v].append(u)

    def power(self, base, exp):
        """
        快速幂运算
        :param base: 底数
        :param exp: 指数
```

```
    :return: (base^exp) % MOD

    时间复杂度：O(log exp)
    空间复杂度：O(1)
    """
    result = 1
    while exp > 0:
        if exp % 2 == 1:
            result = (result * base) % MOD
        base = (base * base) % MOD
        exp //= 2
    return result

def bfs(self, start):
    """
    BFS 求从起点开始的最远节点
    :param start: 起点
    :return: (最远节点, 距离)

    时间复杂度：O(n)
    空间复杂度：O(n)
    """
    visited = [False] * (self.n + 1)
    queue = deque([(start, 0)])  # (节点, 距离)
    visited[start] = True

    farthest_node = start
    max_distance = 0

    while queue:
        current, distance = queue.popleft()
        farthest_node = current
        max_distance = distance

        # 遍历当前节点的所有邻居
        for neighbor in self.graph[current]:
            if not visited[neighbor]:
                visited[neighbor] = True
                queue.append((neighbor, distance + 1))

    return farthest_node, max_distance

def find_diameter(self):
```

```
    """
    使用两次 BFS 法求树的直径
    :return: 树的直径

    时间复杂度：O(n)
    空间复杂度：O(n)
    """
    # 第一次 BFS，从节点 1 开始找到最远节点
    first_node, _ = self.bfs(1)

    # 第二次 BFS，从第一次找到的最远节点开始找到另一个最远节点
    _, diameter = self.bfs(first_node)

    return diameter

def dfs_subtree_size(self, u, parent, subtree_size):
    """
    DFS 计算子树大小
    :param u: 当前节点
    :param parent: 父节点
    :param subtree_size: 子树大小数组
    :return: 子树大小

    时间复杂度：O(n)
    空间复杂度：O(n)
    """
    subtree_size[u] = 1
    for v in self.graph[u]:
        if v != parent:
            subtree_size[u] += self.dfs_subtree_size(v, u, subtree_size)
    return subtree_size[u]

def dfs_depth(self, u, parent, d, depth):
    """
    DFS 计算深度
    :param u: 当前节点
    :param parent: 父节点
    :param d: 当前深度
    :param depth: 深度数组
    :return: None

    时间复杂度：O(n)
    空间复杂度：O(n)
```

```
    """
    depth[u] = d
    for v in self.graph[u]:
        if v != parent:
            self.dfs_depth(v, u, d + 1, depth)

def solve(self):
    """
    计算满足条件的方案数
    :return: 满足条件的方案数

    时间复杂度：O(n)
    空间复杂度：O(n)
    """
    # 计算树的直径
    self.diameter = self.find_diameter()

    # 特殊情况：直径为0（只有一个节点）
    if self.diameter == 0:
        return 1  # 只有一种方案：选择这个节点

    # 计算子树大小
    subtree_size = [0] * (self.n + 1)
    self.dfs_subtree_size(1, 0, subtree_size)

    # 计算深度
    depth = [0] * (self.n + 1)
    self.dfs_depth(1, 0, 0, depth)

    # 找到深度最大的节点
    deepest_node = 1
    for i in range(2, self.n + 1):
        if depth[i] > depth[deepest_node]:
            deepest_node = i

    # 从最深节点再次DFS，找到直径的端点
    self.dfs_depth(deepest_node, 0, 0, depth)
    endpoint1 = deepest_node
    for i in range(1, self.n + 1):
        if depth[i] > depth[endpoint1]:
            endpoint1 = i

    # 从endpoint1再次DFS，找到另一个端点
```

```
self.dfs_depth(endpoint1, 0, 0, depth)
endpoint2 = 1
for i in range(2, self.n + 1):
    if depth[i] > depth[endpoint2]:
        endpoint2 = i

# 计算满足条件的方案数
# 如果直径是偶数，有一个中心点
# 如果直径是奇数，有一个中心边
if self.diameter % 2 == 0:
    # 直径为偶数，有一个中心点
    # 找到中心点
    center = 0
    self.dfs_depth(endpoint1, 0, 0, depth)
    for i in range(1, self.n + 1):
        if depth[i] == self.diameter // 2 and self.bfs(i)[1] == self.diameter // 2:
            center = i
            break

    # 计算以中心点为根的子树中满足条件的方案数
    result = 1
    for v in self.graph[center]:
        # 计算每个子树中满足条件的方案数
        subtree_ways = self.power(2, subtree_size[v]) - 1
        result = (result * subtree_ways) % MOD

    # 至少选择两个节点
    result = (result - 1 + MOD) % MOD
    return result
else:
    # 直径为奇数，有一个中心边
    # 找到中心边的两个端点
    self.dfs_depth(endpoint1, 0, 0, depth)
    center1 = 0
    center2 = 0
    for i in range(1, self.n + 1):
        if depth[i] == self.diameter // 2:
            if center1 == 0:
                center1 = i
            else:
                center2 = i
                break
```

```
            # 计算每个部分中满足条件的方案数
            ways1 = self.power(2, subtree_size[center1]) - 1
            ways2 = self.power(2, subtree_size[center2]) - 1

            result = (ways1 * ways2) % MOD
            return result

    def read_input_and_solve(self):
        """
        读取输入并求解
        时间复杂度：O(n)
        空间复杂度：O(n)
        """
        # 读取节点数
        self.n = int(input())

        # 读取边信息
        for _ in range(self.n - 1):
            u, v = map(int, input().split())
            self.add_edge(u, v)

        # 计算并输出结果
        print(self.solve())

# 主函数
if __name__ == "__main__":
    # 由于这是在线评测题目，实际提交时需要取消下面的注释
    # solution = AtCoderABC221FDiameterSet()
    # solution.read_input_and_solve()

    # 示例测试
    solution = AtCoderABC221FDiameterSet()
    solution.n = 4
    solution.add_edge(1, 2)
    solution.add_edge(2, 3)
    solution.add_edge(3, 4)
    print("示例输出:", solution.solve())
```

==========

文件: AtCoder_ABC221F_DiameterSet.java

==========

```
package class121;
```

```
// AtCoder ABC221 F - Diameter set
// 题目：给定一棵树，找出有多少种点的集合，满足集合内的点两两间的距离均为树的直径。

import java.io.*;
import java.util.*;

public class AtCoder_ABC221F_DiameterSet {

    static final int MAXN = 200001;
    static final long MOD = 998244353;

    // 树的邻接表表示
    static ArrayList<Integer>[] tree;
    static int n;

    // 存储树的直径相关信息
    static int diameter;  // 树的直径
    static int diameterStart, diameterEnd;  // 直径的两个端点

    // dist[i] 表示从某个节点到节点 i 的距离
    static int[] dist;

    // 标记节点是否在直径路径上
    static boolean[] onDiameterPath;

    /**
     * BFS 计算从起点开始到所有节点的距离，并找到最远节点
     * @param start 起点
     * @return 最远节点
     *
     * 时间复杂度：O(n)
     * 空间复杂度：O(n)
     */
    static int bfs(int start) {
        Arrays.fill(dist, -1);
        Queue<Integer> queue = new LinkedList<>();

        dist[start] = 0;
        queue.offer(start);

        int farthestNode = start;
        int maxDist = 0;
```

```
    while (!queue.isEmpty()) {
        int u = queue.poll();

        if (dist[u] > maxDist) {
            maxDist = dist[u];
            farthestNode = u;
        }

        for (int v : tree[u]) {
            if (dist[v] == -1) {
                dist[v] = dist[u] + 1;
                queue.offer(v);
            }
        }
    }

    return farthestNode;
}

/**
 * 计算树的直径并标记直径路径上的节点
 *
 * 时间复杂度：O(n)
 * 空间复杂度：O(n)
 */
static void findDiameter() {
    // 第一次 BFS 找到直径的一个端点
    int firstEnd = bfs(1);

    // 第二次 BFS 找到直径的另一个端点，并计算直径长度
    int secondEnd = bfs(firstEnd);

    diameterStart = firstEnd;
    diameterEnd = secondEnd;
    diameter = dist[secondEnd];

    // 标记直径路径上的节点
    Arrays.fill(onDiameterPath, false);

    // 从直径的一端到另一端标记路径
    int current = secondEnd;
    onDiameterPath[current] = true;
```

```
    // 重构从起点到终点的路径
    while (current != firstEnd) {
        int next = -1;
        for (int neighbor : tree[current]) {
            if (dist[neighbor] == dist[current] - 1) {
                next = neighbor;
                break;
            }
        }
        current = next;
        onDiameterPath[current] = true;
    }
}

/**
 * DFS 计算以当前节点为根的子树中，到直径中点距离为 d 的节点数量
 * @param u 当前节点
 * @param parent 父节点
 * @param depth 当前深度
 * @param count 数组，count[d]表示到直径中点距离为 d 的节点数量
 *
 * 时间复杂度：O(n)
 * 空间复杂度：O(n)
 */
static void dfs(int u, int parent, int depth, long[] count) {
    // 如果当前节点在直径路径上，则不继续深入
    if (onDiameterPath[u]) {
        return;
    }

    // 统计到直径中点距离为 depth 的节点数量
    count[depth]++;

    // 递归处理子节点
    for (int v : tree[u]) {
        if (v != parent && !onDiameterPath[v]) {
            dfs(v, u, depth + 1, count);
        }
    }
}

/**
```

```
 * 计算满足条件的点集数量
 * @return 答案
 *
 * 时间复杂度：O(n)
 * 空间复杂度：O(n)
 */
static long solve() {
    // 初始化
    dist = new int[n + 1];
    onDiameterPath = new boolean[n + 1];

    // 计算树的直径
    findDiameter();

    // 计算直径中点
    int diameterMid = diameter / 2;

    // 以直径中点为根，计算每个子树中到根距离为 d 的节点数量
    long result = 1;

    // 遍历直径路径上的每个节点，计算以其为根的子树贡献
    int current = diameterStart;
    while (current != diameterEnd) {
        // 计算当前节点的子树贡献
        long[] count = new long[diameter + 1];
        for (int v : tree[current]) {
            if (!onDiameterPath[v]) {
                dfs(v, current, 1, count);
            }
        }

        // 计算当前节点子树的贡献
        long contribution = 1;  // 空集的贡献
        for (int i = 1; i <= diameter; i++) {
            // 每个节点可以选择加入或不加入点集
            contribution = (contribution + count[i]) % MOD;
        }

        result = (result * contribution) % MOD;

        // 移动到下一个节点
        int next = -1;
        for (int neighbor : tree[current]) {
```

```
                if (onDiameterPath[neighbor] && dist[neighbor] == dist[current] + 1) {
                    next = neighbor;
                    break;
                }
            }
            current = next;
        }

        return result;
    }

    // 主方法（用于测试）
    public static void main(String[] args) {
        // 由于这是 AtCoder 题目，实际提交时需要按照题目要求的输入格式处理
        // 这里我们只展示算法实现

        // 示例输入：
        // n = 4
        // 边: 1-2, 2-3, 3-4
        // 预期输出：8

        n = 4;
        tree = new ArrayList[n + 1];
        for (int i = 1; i <= n; i++) {
            tree[i] = new ArrayList<>();
        }

        // 添加边
        tree[1].add(2);
        tree[2].add(1);
        tree[2].add(3);
        tree[3].add(2);
        tree[3].add(4);
        tree[4].add(3);

        System.out.println("满足条件的点集数量: " + solve()); // 应该输出 8
    }
}
```

==========

文件: AtCoder_ABC221F_DiameterSet.py

==========

```
# AtCoder ABC221 F - Diameter set
# 题目：给定一棵树，找出有多少种点的集合，满足集合内的点两两间的距离均为树的直径。

from collections import deque, defaultdict

# 全局变量
n = 0
MOD = 998244353

# 树的邻接表表示
tree = defaultdict(list)

# 存储树的直径相关信息
diameter = 0  # 树的直径
diameter_start = 0
diameter_end = 0  # 直径的两个端点

# dist[i] 表示从某个节点到节点 i 的距离
dist = []

# 标记节点是否在直径路径上
on_diameter_path = []

def bfs(start):
    """
    BFS 计算从起点开始到所有节点的距离，并找到最远节点
    :param start: 起点
    :return: 最远节点

    时间复杂度：O(n)
    空间复杂度：O(n)
    """
    global dist
    dist = [-1] * (n + 1)
    queue = deque()

    dist[start] = 0
    queue.append(start)

    farthest_node = start
    max_dist = 0

    while queue:
```

```
        u = queue.popleft()

        if dist[u] > max_dist:
            max_dist = dist[u]
            farthest_node = u

        for v in tree[u]:
            if dist[v] == -1:
                dist[v] = dist[u] + 1
                queue.append(v)

    return farthest_node

def find_diameter():
    """
    计算树的直径并标记直径路径上的节点

    时间复杂度：O(n)
    空间复杂度：O(n)
    """
    global diameter, diameter_start, diameter_end, on_diameter_path

    # 第一次 BFS 找到直径的一个端点
    first_end = bfs(1)

    # 第二次 BFS 找到直径的另一个端点，并计算直径长度
    second_end = bfs(first_end)

    diameter_start = first_end
    diameter_end = second_end
    diameter = dist[second_end]

    # 标记直径路径上的节点
    on_diameter_path = [False] * (n + 1)

    # 从直径的一端到另一端标记路径
    current = second_end
    on_diameter_path[current] = True

    # 重构从起点到终点的路径
    while current != first_end:
        next_node = -1
        for neighbor in tree[current]:
```

```
            if dist[neighbor] == dist[current] - 1:
                next_node = neighbor
                break
        current = next_node
        on_diameter_path[current] = True

def dfs(u, parent, depth, count):
    """
    DFS 计算以当前节点为根的子树中，到直径中点距离为 d 的节点数量
    :param u: 当前节点
    :param parent: 父节点
    :param depth: 当前深度
    :param count: 数组，count[d]表示到直径中点距离为 d 的节点数量

    时间复杂度：O(n)
    空间复杂度：O(n)
    """
    # 如果当前节点在直径路径上，则不继续深入
    if on_diameter_path[u]:
        return

    # 统计到直径中点距离为 depth 的节点数量
    count[depth] += 1

    # 递归处理子节点
    for v in tree[u]:
        if v != parent and not on_diameter_path[v]:
            dfs(v, u, depth + 1, count)

def solve():
    """
    计算满足条件的点集数量
    :return: 答案

    时间复杂度：O(n)
    空间复杂度：O(n)
    """
    # 计算树的直径
    find_diameter()

    # 计算直径中点
    diameter_mid = diameter // 2
```

```
    # 以直径中点为根，计算每个子树中到根距离为 d 的节点数量
    result = 1

    # 遍历直径路径上的每个节点，计算以其为根的子树贡献
    current = diameter_start
    while current != diameter_end:
        # 计算当前节点的子树贡献
        count = [0] * (diameter + 1)
        for v in tree[current]:
            if not on_diameter_path[v]:
                dfs(v, current, 1, count)

        # 计算当前节点子树的贡献
        contribution = 1  # 空集的贡献
        for i in range(1, diameter + 1):
            # 每个节点可以选择加入或不加入点集
            contribution = (contribution + count[i]) % MOD

        result = (result * contribution) % MOD

        # 移动到下一个节点
        next_node = -1
        for neighbor in tree[current]:
            if on_diameter_path[neighbor] and dist[neighbor] == dist[current] + 1:
                next_node = neighbor
                break
        current = next_node

    return result

# 主方法（用于测试）
if __name__ == "__main__":
    # 由于这是 AtCoder 题目，实际提交时需要按照题目要求的输入格式处理
    # 这里我们只展示算法实现

    # 示例输入：
    # n = 4
    # 边: 1-2, 2-3, 3-4
    # 预期输出：8

    n = 4
    tree.clear()
```

```
    # 添加边
    tree[1].append(2)
    tree[2].append(1)
    tree[2].append(3)
    tree[3].append(2)
    tree[3].append(4)
    tree[4].append(3)

    print("满足条件的点集数量:", solve())  # 应该输出 8
```

==========

文件: Code01_Diameter1.java
==========

```
package class121;

// 树的直径模版(两遍 dfs)
// 给定一棵树，边权可能为负，求直径长度
// 测试链接 : https://www.luogu.com.cn/problem/U81904
// 提交以下的 code，提交时请把类名改成"Main"
// 会有无法通过的用例，因为树上有负边

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;

/**
 * 树的直径求解器 - 两次 DFS 方法
 *
 * 算法说明：
 * 1. 从任意一点开始，找到距离它最远的点 s
 * 2. 从 s 开始，找到距离它最远的点 t
 * 3. s 到 t 的距离即为树的直径
 *
 * 适用场景：
 * - 适用于边权非负的情况
 * - 对于有负权边的树，此方法可能不适用
 *
 * 时间复杂度：O(n)，其中 n 是树中节点的数量
```

```
 * 空间复杂度：O(n)，用于存储邻接表和辅助数组
 *
 * 相关题目：
 * - 洛谷 U81904 树的直径
 * - LeetCode 543. 二叉树的直径
 * - SPOJ PT07Z - Longest path in a tree
 * - CSES 1131 - Tree Diameter
 * - 51Nod 2602 - 树的直径
 * - AtCoder ABC221F - Diameter Set
 * - Codeforces 1499F - Diameter Cuts
 */
public class Code01_Diameter1 {

    public static int MAXN = 500001;

    public static int n;

    // 邻接表头数组，head[i]表示节点 i 的第一条边在 next 数组中的索引
    public static int[] head = new int[MAXN];

    // 边的邻接表，next[i]表示第 i 条边的下一条边的索引
    public static int[] next = new int[MAXN << 1];

    // 边的邻接表，to[i]表示第 i 条边指向的节点
    public static int[] to = new int[MAXN << 1];

    // 边的邻接表，weight[i]表示第 i 条边的权重
    public static int[] weight = new int[MAXN << 1];

    // 边的计数器
    public static int cnt;

    // 直径的开始点
    public static int start;

    // 直径的结束点
    public static int end;

    // 直径长度
    public static int diameter;

    // dist[i] : 从规定的头节点出发，走到 i 的距离
    public static int[] dist = new int[MAXN];
```

```
// last[i] : 从规定的头节点出发，i 节点的上一个节点
public static int[] last = new int[MAXN];

/**
 * 初始化数据结构
 * 重置边的计数器和邻接表头数组
 */
public static void build() {
    cnt = 1;
    Arrays.fill(head, 1, n + 1, 0);
}

/**
 * 添加一条无向边
 * @param u 起点
 * @param v 终点
 * @param w 边的权重
 */
public static void addEdge(int u, int v, int w) {
    next[cnt] = head[u];
    to[cnt] = v;
    weight[cnt] = w;
    head[u] = cnt++;
}

/**
 * 计算树的直径
 * 使用两次 DFS 方法：
 * 1. 从节点 1 开始 DFS，找到距离最远的节点 start
 * 2. 从 start 开始 DFS，找到距离最远的节点 end
 * 3. start 到 end 的距离就是树的直径
 */
public static void road() {
    dfs(1, 0, 0);
    start = 1;
    for (int i = 2; i <= n; i++) {
        if (dist[i] > dist[start]) {
            start = i;
        }
    }
    dfs(start, 0, 0);
    end = 1;
```

```
    for (int i = 2; i <= n; i++) {
        if (dist[i] > dist[end]) {
            end = i;
        }
    }
    diameter = dist[end];
}

/**
 * 深度优先搜索
 * @param u 当前节点
 * @param f 父节点
 * @param w 到父节点的边的权重
 */
public static void dfs(int u, int f, int w) {
    last[u] = f;
    dist[u] = dist[f] + w;
    for (int e = head[u]; e != 0; e = next[e]) {
        if (to[e] != f) {
            dfs(to[e], u, weight[e]);
        }
    }
}

/**
 * 主方法
 * 读取输入数据，构建树，计算直径并输出结果
 * @param args 命令行参数
 * @throws IOException 输入输出异常
 */
public static void main(String[] args) throws IOException {
    BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
    StreamTokenizer in = new StreamTokenizer(br);
    PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));
    in.nextToken();
    n = (int) in.nval;
    build();
    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;
            addEdge(u, v, w);
            addEdge(v, u, w);
        }
        road();
        out.println(diameter);
        out.flush();
        out.close();
        br.close();
    }

}
```

==========

文件: Code01_Diameter2.java

==========

```
package class121;

// 树的直径模版(树型 dp)
// 给定一棵树，边权可能为负，求直径长度
// 测试链接 : https://www.luogu.com.cn/problem/U81904
// 提交以下的 code，提交时请把类名改成"Main"，可以通过所有的用例

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;

/**
 * 树的直径求解器 - 树形动态规划方法
 *
 * 算法说明：
 * 通过一次 DFS，在每个节点计算经过该节点的最长路径。
 * 这种方法可以处理负权边的情况。
 *
 * 核心思想：
 * 对于每个节点 u，维护两个值：
 * 1. dist[u]：从 u 开始必须往下走，能走出的最大距离（可以不选任何边）
 * 2. ans[u]：路径必须包含点 u 的情况下，最大路径和
```

```
 *
 * 适用场景：
 * - 可以处理负权边的情况
 * - 适用于任何树结构
 *
 * 时间复杂度：O(n)，其中 n 是树中节点的数量
 * 空间复杂度：O(n)，用于存储邻接表和辅助数组
 *
 * 相关题目：
 * - 洛谷 U81904 树的直径
 * - LeetCode 543. 二叉树的直径
 * - SPOJ PT07Z - Longest path in a tree
 * - CSES 1131 - Tree Diameter
 * - 51Nod 2602 - 树的直径
 * - AtCoder ABC221F - Diameter Set
 * - Codeforces 1499F - Diameter Cuts
 */
public class Code01_Diameter2 {

    public static int MAXN = 500001;

    public static int n;

    // 邻接表头数组，head[i]表示节点 i 的第一条边在 next 数组中的索引
    public static int[] head = new int[MAXN];

    // 边的邻接表，next[i]表示第 i 条边的下一条边的索引
    public static int[] next = new int[MAXN << 1];

    // 边的邻接表，to[i]表示第 i 条边指向的节点
    public static int[] to = new int[MAXN << 1];

    // 边的邻接表，weight[i]表示第 i 条边的权重
    public static int[] weight = new int[MAXN << 1];

    // 边的计数器
    public static int cnt;

    // dist[u] : 从 u 开始必须往下走，能走出的最大距离，可以不选任何边
    public static int[] dist = new int[MAXN];

    // ans[u] : 路径必须包含点 u 的情况下，最大路径和
    public static int[] ans = new int[MAXN];
```

```
/**
 * 初始化数据结构
 * 重置边的计数器和相关数组
 */
public static void build() {
    cnt = 1;
    Arrays.fill(head, 1, n + 1, 0);
    Arrays.fill(dist, 1, n + 1, 0);
    Arrays.fill(ans, 1, n + 1, 0);
}

/**
 * 添加一条无向边
 * @param u 起点
 * @param v 终点
 * @param w 边的权重
 */
public static void addEdge(int u, int v, int w) {
    next[cnt] = head[u];
    to[cnt] = v;
    weight[cnt] = w;
    head[u] = cnt++;
}

/**
 * 树形动态规划计算树的直径
 * @param u 当前节点
 * @param f 父节点
 */
public static void dp(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) {
            dp(v, u);
        }
    }

    // 计算经过当前节点的最大路径
    for (int e = head[u], v; e != 0; e = next[e]) {
        v = to[e];
        if (v != f) {
```

```
                // 更新经过当前节点的最大路径
                ans[u] = Math.max(ans[u], dist[u] + dist[v] + weight[e]);
                // 更新从当前节点向下走的最大距离
                dist[u] = Math.max(dist[u], dist[v] + weight[e]);
            }
        }
    }

    /**
     * 主方法
     * 读取输入数据，构建树，使用树形 DP 计算直径并输出结果
     * @param args 命令行参数
     * @throws IOException 输入输出异常
     */
    public static void main(String[] args) throws IOException {
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        StreamTokenizer in = new StreamTokenizer(br);
        PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));
        in.nextToken();
        n = (int) in.nval;
        build();
        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;
            addEdge(u, v, w);
            addEdge(v, u, w);
        }
        dp(1, 0);
        int diameter = Integer.MIN_VALUE;
        for (int i = 1; i <= n; i++) {
            diameter = Math.max(diameter, ans[i]);
        }
        out.println(diameter);
        out.flush();
        out.close();
        br.close();
    }

}
```

```
==========================================

文件: Code01_Diameter3.java
==========================================
package class121;

// 树的直径模版(树型 dp 版逻辑小化简)
// 给定一棵树，边权可能为负，求直径长度
// 测试链接 : https://www.luogu.com.cn/problem/U81904
// 提交以下的 code，提交时请把类名改成"Main"，可以通过所有的用例

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;

/**
 * 树的直径求解器 - 树形动态规划方法（简化版）
 *
 * 算法说明：
 * 通过一次 DFS，在每个节点计算经过该节点的最长路径。
 * 这是 Code01_Diameter2 的简化版本，直接在 DP 过程中更新直径。
 *
 * 核心思想：
 * 对于每个节点 u，维护 dist[u]表示从 u 开始必须往下走能走出的最大距离。
 * 在遍历子节点时，直接更新全局直径变量。
 *
 * 适用场景：
 * - 可以处理负权边的情况
 * - 适用于任何树结构
 * - 代码更简洁，逻辑更清晰
 *
 * 时间复杂度：O(n)，其中 n 是树中节点的数量
 * 空间复杂度：O(n)，用于存储邻接表和辅助数组
 *
 * 相关题目：
 * - 洛谷 U81904 树的直径
 * - LeetCode 543. 二叉树的直径
 * - SPOJ PT07Z - Longest path in a tree
```

```
 * - CSES 1131 - Tree Diameter
 * - 51Nod 2602 - 树的直径
 * - AtCoder ABC221F - Diameter Set
 * - Codeforces 1499F - Diameter Cuts
 */
public class Code01_Diameter3 {

    public static int MAXN = 500001;

    public static int n;

    // 邻接表头数组，head[i]表示节点 i 的第一条边在 next 数组中的索引
    public static int[] head = new int[MAXN];

    // 边的邻接表，next[i]表示第 i 条边的下一条边的索引
    public static int[] next = new int[MAXN << 1];

    // 边的邻接表，to[i]表示第 i 条边指向的节点
    public static int[] to = new int[MAXN << 1];

    // 边的邻接表，weight[i]表示第 i 条边的权重
    public static int[] weight = new int[MAXN << 1];

    // 边的计数器
    public static int cnt;

    // dist[u] : 从 u 开始必须往下走，能走出的最大距离，可以不选任何边
    public static int[] dist = new int[MAXN];

    // 直径的长度
    public static int diameter;

    /**
     * 初始化数据结构
     * 重置边的计数器和相关数组
     */
    public static void build() {
        cnt = 1;
        Arrays.fill(head, 1, n + 1, 0);
        Arrays.fill(dist, 1, n + 1, 0);
        diameter = Integer.MIN_VALUE;
    }
```

```
/**
 * 添加一条无向边
 * @param u 起点
 * @param v 终点
 * @param w 边的权重
 */
public static void addEdge(int u, int v, int w) {
    next[cnt] = head[u];
    to[cnt] = v;
    weight[cnt] = w;
    head[u] = cnt++;
}

/**
 * 树形动态规划计算树的直径（简化版）
 * @param u 当前节点
 * @param f 父节点
 */
public static void dp(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) {
            dp(v, u);
        }
    }

    // 计算经过当前节点的最大路径并更新直径
    for (int e = head[u], v; e != 0; e = next[e]) {
        v = to[e];
        if (v != f) {
            // 直接更新直径
            diameter = Math.max(diameter, dist[u] + dist[v] + weight[e]);
            // 更新从当前节点向下走的最大距离
            dist[u] = Math.max(dist[u], dist[v] + weight[e]);
        }
    }
}

/**
 * 主方法
 * 读取输入数据，构建树，使用树形 DP 计算直径并输出结果
 * @param args 命令行参数
```

```
     * @throws IOException 输入输出异常
     */
    public static void main(String[] args) throws IOException {
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        StreamTokenizer in = new StreamTokenizer(br);
        PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));
        in.nextToken();
        n = (int) in.nval;
        build();
        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;
            addEdge(u, v, w);
            addEdge(v, u, w);
        }
        dp(1, 0);
        out.println(diameter);
        out.flush();
        out.close();
        br.close();
    }

}
```

==========

文件: Code02_DiameterAndCommonEdges1.java

==========

```
package class121;

// 所有直径的公共部分(递归版)
// 给定一棵树，边权都为正
// 打印直径长度、所有直径的公共部分有几条边
// 测试链接 : https://www.luogu.com.cn/problem/P3304
// 提交以下的 code，提交时请把类名改成"Main"
// C++这么写能通过，java 会因为递归层数太多而爆栈
// java 能通过的写法参考本节课 Code02_DiameterAndCommonEdges2 文件

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_DiameterAndCommonEdges1 {

    public static int MAXN = 200001;

    public static int n;

    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[] weight = new int[MAXN << 1];

    public static int cnt;

    public static int start;

    public static int end;

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

    public static int[] last = new int[MAXN];

    public static long diameter;

    public static boolean[] diameterPath = new boolean[MAXN];

    public static int commonEdges;

    public static void build() {
        cnt = 1;
        Arrays.fill(head, 1, n + 1, 0);
        Arrays.fill(diameterPath, 1, n + 1, false);
    }
```

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

public static void road() {
    dfs(1, 0, 0);
    start = 1;
    for (int i = 2; i <= n; i++) {
        if (dist[i] > dist[start]) {
            start = i;
        }
    }
    dfs(start, 0, 0);
    end = 1;
    for (int i = 2; i <= n; i++) {
        if (dist[i] > dist[end]) {
            end = i;
        }
    }
    diameter = dist[end];
}

public static void dfs(int u, int f, long c) {
    last[u] = f;
    dist[u] = c;
    for (int e = head[u]; e != 0; e = next[e]) {
        if (to[e] != f) {
            dfs(to[e], u, c + weight[e]);
        }
    }
}

// 不能走向直径路径上的节点
// 能走出的最大距离
public static long maxDistanceExceptDiameter(int u, int f, long c) {
    long ans = c;
    for (int e = head[u], v; e != 0; e = next[e]) {
        v = to[e];
        if (!diameterPath[v] && v != f) {
            ans = Math.max(ans, maxDistanceExceptDiameter(v, u, c + weight[e]));
```

```
        }
    }
    return ans;
}

public static void compute() {
    road();
    for (int i = end; i != 0; i = last[i]) {
        diameterPath[i] = true;
    }
    int l = start;
    int r = end;
    long maxDist;
    for (int i = last[end]; i != start; i = last[i]) {
        maxDist = maxDistanceExceptDiameter(i, 0, 0);
        if (maxDist == diameter - dist[i]) {
            r = i;
        }
        if (maxDist == dist[i] && l == start) {
            l = i;
        }
    }
    if (l == r) {
        commonEdges = 0;
    } else {
        commonEdges = 1;
        for (int i = last[r]; i != l; i = last[i]) {
            commonEdges++;
        }
    }
}

public static void main(String[] args) throws IOException {
    BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
    StreamTokenizer in = new StreamTokenizer(br);
    PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));
    in.nextToken();
    n = (int) in.nval;
    build();
    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;
            addEdge(u, v, w);
            addEdge(v, u, w);
        }
        compute();
        out.println(diameter);
        out.println(commonEdges);
        out.flush();
        out.close();
        br.close();
    }

}
```

==========

文件: Code02_DiameterAndCommonEdges2.java

==========

```
package class121;

// 所有直径的公共部分(迭代版)
// 给定一棵树，边权都为正
// 打印直径长度、所有直径的公共部分有几条边
// 测试链接 : https://www.luogu.com.cn/problem/P3304
// 提交以下的 code，提交时请把类名改成"Main"，可以通过所有用例

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_DiameterAndCommonEdges2 {

    public static int MAXN = 200001;

    public static int n;

    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[] weight = new int[MAXN << 1];

public static int cnt;

public static int start;

public static int end;

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

public static int[] last = new int[MAXN];

public static long diameter;

public static boolean[] diameterPath = new boolean[MAXN];

public static int commonEdges;

public static void build() {
    cnt = 1;
    Arrays.fill(head, 1, n + 1, 0);
    Arrays.fill(diameterPath, 1, n + 1, false);
}

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

public static void road() {
    dfs(1);
    start = 1;
    for (int i = 2; i <= n; i++) {
        if (dist[i] > dist[start]) {
            start = i;
        }
```

```
    }
    dfs(start);
    end = 1;
    for (int i = 2; i <= n; i++) {
        if (dist[i] > dist[end]) {
            end = i;
        }
    }
    diameter = dist[end];
}

// dfs 方法改迭代版
// 不会改，看讲解 118，讲了怎么从递归版改成迭代版
public static int[][] ufeStack = new int[MAXN][3];

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

public static int stackSize;

public static int u, f, e;

public static long c;

public static void push(int u, int f, int e, long c) {
    ufeStack[stackSize][0] = u;
    ufeStack[stackSize][1] = f;
    ufeStack[stackSize][2] = e;
    distStack[stackSize] = c;
    stackSize++;
}

public static void pop() {
    --stackSize;
    u = ufeStack[stackSize][0];
    f = ufeStack[stackSize][1];
    e = ufeStack[stackSize][2];
    c = distStack[stackSize];
}

public static void dfs(int root) {
    stackSize = 0;
    push(root, 0, -1, 0);
    while (stackSize > 0) {
```

```
        pop();
        if (e == -1) {
            last[u] = f;
            dist[u] = c;
            e = head[u];
        } else {
            e = next[e];
        }
        if (e != 0) {
            push(u, f, e, c);
            if (to[e] != f) {
                push(to[e], u, -1, c + weight[e]);
            }
        }
    }
}

// maxDistanceExceptDiameter 方法改迭代版
// 不会改，看讲解 118，讲了怎么从递归版改成迭代版
public static long maxDistanceExceptDiameter(int root) {
    stackSize = 0;
    push(root, 0, -1, 0);
    long ans = 0;
    while (stackSize > 0) {
        pop();
        if (e == -1) {
            e = head[u];
        } else {
            e = next[e];
        }
        int v;
        if (e != 0) {
            push(u, f, e, c);
            v = to[e];
            if (!diameterPath[v] && v != f) {
                push(v, u, -1, c + weight[e]);
            }
        } else {
            ans = Math.max(ans, c);
        }
    }
    return ans;
}
```

```
public static void compute() {
    road();
    for (int i = end; i != 0; i = last[i]) {
        diameterPath[i] = true;
    }
    int l = start;
    int r = end;
    long maxDist;
    for (int i = last[end]; i != start; i = last[i]) {
        maxDist = maxDistanceExceptDiameter(i);
        if (maxDist == diameter - dist[i]) {
            r = i;
        }
        if (maxDist == dist[i] && l == start) {
            l = i;
        }
    }
    if (l == r) {
        commonEdges = 0;
    } else {
        commonEdges = 1;
        for (int i = last[r]; i != l; i = last[i]) {
            commonEdges++;
        }
    }
}

public static void main(String[] args) throws IOException {
    BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
    StreamTokenizer in = new StreamTokenizer(br);
    PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));
    in.nextToken();
    n = (int) in.nval;
    build();
    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;
        addEdge(u, v, w);
```

```
            addEdge(v, u, w);
        }
        compute();
        out.println(diameter);
        out.println(commonEdges);
        out.flush();
        out.close();
        br.close();
    }

}
```

==========

文件: Code03_BuildPark.java

==========

```
package class121;

// 造公园
// 一共 n 个节点，编号 1~n，有 m 条边连接，边权都是 1
// 所有节点可能形成多个连通区，每个连通区保证是树结构
// 有两种类型的操作
// 操作 1 x   : 返回 x 到离它最远的点的距离
// 操作 2 x y : 如果 x 和 y 已经连通，那么忽略
//              如果不连通，那么执行连通操作，把 x 和 y 各自的区域连通起来
//              并且要保证连通成的大区域的直径长度最小
// 测试链接 : https://www.luogu.com.cn/problem/P2195
// 提交以下的 code，提交时请把类名改成"Main"，可以通过所有用例

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 Code03_BuildPark {

    public static int MAXN = 300001;

    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;

// 并查集需要
public static int[] father = new int[MAXN];

// 树型 dp 需要
// dist[u] : 从 u 开始必须往下走，能走出的最大距离，可以不选任何边
public static int[] dist = new int[MAXN];

// diameter[i] : 如果 i 是集合的头节点，diameter[i]表示整个集合的直径长度
//               如果 i 不再是集合的头节点，diameter[i]的值没有用
// 并查集 + 集合打标签技巧，不会的看讲解 056、讲解 057
public static int[] diameter = new int[MAXN];

public static void build(int n) {
    cnt = 1;
    Arrays.fill(head, 1, n, 0);
    for (int i = 1; i <= n; i++) {
        father[i] = i;
    }
    Arrays.fill(dist, 1, n, 0);
    Arrays.fill(diameter, 1, n, 0);
}

public static void addEdge(int u, int v) {
    next[cnt] = head[u];
    to[cnt] = v;
    head[u] = cnt++;
}

public static int find(int i) {
    if (i != father[i]) {
        father[i] = find(father[i]);
    }
    return father[i];
}

// 树型 dp 的方式求直径长度
public static void dp(int u, int f) {
```

```
    for (int e = head[u], v; e != 0; e = next[e]) {
        v = to[e];
        if (v != f) {
            dp(v, u);
        }
    }
    for (int e = head[u], v; e != 0; e = next[e]) {
        v = to[e];
        if (v != f) {
            diameter[u] = Math.max(diameter[u], Math.max(diameter[v], dist[u] + dist[v] + 1));
            dist[u] = Math.max(dist[u], dist[v] + 1);
        }
    }
}

public static void main(String[] args) throws IOException {
    BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
    StreamTokenizer in = new StreamTokenizer(br);
    PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));
    in.nextToken();
    int n = (int) in.nval;
    in.nextToken();
    int m = (int) in.nval;
    in.nextToken();
    int q = (int) in.nval;
    build(n);
    for (int i = 1, u, v; i <= m; i++) {
        in.nextToken();
        u = (int) in.nval;
        in.nextToken();
        v = (int) in.nval;
        addEdge(u, v);
        addEdge(v, u);
        u = find(u);
        v = find(v);
        father[u] = v;
    }
    for (int i = 1; i <= n; i++) {
        if (i == father[i]) {
            dp(i, 0);
        }
    }
    for (int i = 1, op, x, y; i <= q; i++) {
```

```
            in.nextToken();
            op = (int) in.nval;
            if (op == 1) {
                in.nextToken();
                x = (int) in.nval;
                x = find(x);
                out.println(diameter[x]);
            } else {
                in.nextToken();
                x = (int) in.nval;
                in.nextToken();
                y = (int) in.nval;
                x = find(x);
                y = find(y);
                if (x != y) {
                    father[x] = y;
                    diameter[y] = Math.max((diameter[x] + 1) / 2 + (diameter[y] + 1) / 2 + 1,
                            Math.max(diameter[x], diameter[y]));
                }
            }
        }
        out.flush();
        out.close();
        br.close();
    }

}
```

==========

文件: Code04_Patrol.java

==========

```
package class121;

// 巡逻
// 一共 n 个节点，编号 1~n，结构是一棵树，每条边都是双向的
// 警察局在 1 号点，警车每天从 1 号点出发，一定要走过树上所有的边，最后回到 1 号点
// 现在为了减少经过边的数量，你可以新建 k 条边，把树上任意两点直接相连
// 并且每天警车必须经过新建的道路正好一次
// 计算出最佳的新建道路的方案，返回巡逻走边数量的最小值
// 测试链接 : https://www.luogu.com.cn/problem/P3629
// 1 <= n <= 10^5
// 1 <= k <= 2
```

```
// 提交以下的 code，提交时请把类名改成"Main"，可以通过所有用例

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_Patrol {

    public static int MAXN = 100001;

    public static int n;

    public static int k;

    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;

    public static int start;

    public static int end;

    public static int[] dist = new int[MAXN];

    public static int[] last = new int[MAXN];

    public static int diameter1;

    public static int diameter2;

    public static boolean[] diameterPath = new boolean[MAXN];

    public static void build() {
        cnt = 1;
        Arrays.fill(head, 1, n + 1, 0);
```

```
    diameter1 = 0;
    diameter2 = 0;
    Arrays.fill(diameterPath, 1, n, false);
}

public static void addEdge(int u, int v) {
    next[cnt] = head[u];
    to[cnt] = v;
    head[u] = cnt++;
}

public static void road() {
    dfs(1, 0, 0);
    start = 1;
    for (int i = 2; i <= n; i++) {
        if (dist[i] > dist[start]) {
            start = i;
        }
    }
    dfs(start, 0, 0);
    end = 1;
    for (int i = 2; i <= n; i++) {
        if (dist[i] > dist[end]) {
            end = i;
        }
    }
    diameter1 = dist[end];
}

public static void dfs(int u, int f, int w) {
    last[u] = f;
    dist[u] = dist[f] + w;
    for (int e = head[u]; e != 0; e = next[e]) {
        if (to[e] != f) {
            dfs(to[e], u, 1);
        }
    }
}

// 树型 dp 第二次求直径长度
public static void dp(int u, int f) {
    for (int e = head[u], v; e != 0; e = next[e]) {
        v = to[e];
```

```
        if (v != f) {
            dp(v, u);
        }
    }
    for (int e = head[u], v, w; e != 0; e = next[e]) {
        v = to[e];
        if (v != f) {
            w = diameterPath[u] && diameterPath[v] ? -1 : 1;
            diameter2 = Math.max(diameter2, dist[u] + dist[v] + w);
            dist[u] = Math.max(dist[u], dist[v] + w);
        }
    }
}

public static void main(String[] args) throws IOException {
    BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
    StreamTokenizer in = new StreamTokenizer(br);
    PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));
    in.nextToken();
    n = (int) in.nval;
    in.nextToken();
    k = (int) in.nval;
    build();
    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);
    }
    out.println(compute());
    out.flush();
    out.close();
    br.close();
}

public static int compute() {
    road();
    if (k == 1) {
        return 2 * (n - 1) - diameter1 + 1;
    } else {
        for (int i = end; i != 0; i = last[i]) {
```

```
                diameterPath[i] = true;
            }
            Arrays.fill(dist, 1, n + 1, 0);
            dp(1, 0);
            return 2 * (n - 1) - diameter1 - diameter2 + 2;
        }
    }

}
```

==========

文件: Code05_FireFighting1.java

==========

```
package class121;

// 消防(递归版)
// 一共 n 个节点，编号 1~n，有 n-1 条边连接成一棵树，每条边上有非负权值
// 给定一个非负整数 s，表示可以在树上选择一条长度不超过 s 的路径
// 然后在这条路径的点上建立消防站，每个居民可以去往这条路径上的任何消防站
// 目标：哪怕最远的居民走到消防站的距离也要尽量少
// 返回最远居民走到消防站的最短距离
// 测试链接 : https://www.luogu.com.cn/problem/P2491
// 提交以下的 code，提交时请把类名改成"Main"
// C++这么写能通过，java 会因为递归层数太多而爆栈
// java 能通过的写法参考本节课 Code05_FireFighting2 文件

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_FireFighting1 {

    public static int MAXN = 300001;

    public static int n;

    public static int s;
```

```
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[] weight = new int[MAXN << 1];

public static int cnt;

public static int start;

public static int end;

public static int diameter;

public static int[] dist = new int[MAXN];

public static int[] last = new int[MAXN];

// pred[i] : i 节点在直径上，和前一个点之间的距离，以 start 做根
public static int[] pred = new int[MAXN];

public static boolean[] diameterPath = new boolean[MAXN];

public static int[] maxDist = new int[MAXN];

public static void build() {
    cnt = 1;
    Arrays.fill(head, 1, n + 1, 0);
    Arrays.fill(diameterPath, 1, n, false);
}

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

public static void road() {
    dfs(1, 0, 0);
    start = 1;
```

```
    for (int i = 2; i <= n; i++) {
        if (dist[i] > dist[start]) {
            start = i;
        }
    }
    dfs(start, 0, 0);
    end = 1;
    for (int i = 2; i <= n; i++) {
        if (dist[i] > dist[end]) {
            end = i;
        }
    }
    diameter = dist[end];
}

public static void dfs(int u, int f, int w) {
    last[u] = f;
    dist[u] = dist[f] + w;
    pred[u] = w;
    for (int e = head[u]; e != 0; e = next[e]) {
        if (to[e] != f) {
            dfs(to[e], u, weight[e]);
        }
    }
}

public static void distance() {
    for (int i = end; i != 0; i = last[i]) {
        diameterPath[i] = true;
    }
    for (int i = end; i != 0; i = last[i]) {
        maxDist[i] = maxDistanceExceptDiameter(i, 0, 0);
    }
}

// 不能走向直径路径上的节点
// 能走出的最大距离
public static int maxDistanceExceptDiameter(int u, int f, int c) {
    int ans = c;
    for (int e = head[u], v; e != 0; e = next[e]) {
        v = to[e];
        if (!diameterPath[v] && v != f) {
            ans = Math.max(ans, maxDistanceExceptDiameter(v, u, c + weight[e]));
```

```
        }
    }
    return ans;
}

// 单调队列维护窗口内最大值
// 不会的看讲解 054
public static int[] queue = new int[MAXN];

public static int compute() {
    int suml = 0, sumr = 0;
    // 用 h 和 t 表示单调队列的头和尾
    int h = 0, t = 0;
    int ans = Integer.MAX_VALUE;
    // 窗口范围[l,r)，左闭右开，直径上的窗口[l...r-1]
    // l 是窗口左端点，r 是窗口右端点的再下一个点
    // 课上图解是从 start 到 end，实际是从 end 到 start，思路没有区别
    for (int l = end, r = end; l != 0; l = last[l]) {
        while (r != 0 && sumr - suml + pred[r] <= s) {
            while (h < t && maxDist[queue[t - 1]] <= maxDist[r]) {
                t--;
            }
            sumr += pred[r];
            queue[t++] = r;
            r = last[r];
        }
        ans = Math.min(ans, Math.max(Math.max(suml, maxDist[queue[h]]), diameter - sumr));
        if (queue[h] == l) {
            h++;
        }
        suml += pred[l];
    }
    return ans;
}

public static void main(String[] args) throws IOException {
    BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
    StreamTokenizer in = new StreamTokenizer(br);
    PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));
    in.nextToken();
    n = (int) in.nval;
    in.nextToken();
    s = (int) in.nval;
```

```
        build();
        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;
            addEdge(u, v, w);
            addEdge(v, u, w);
        }
        road();
        distance();
        out.println(compute());
        out.flush();
        out.close();
        br.close();
    }

}
```

==========

文件: Code05_FireFighting2.java

==========

```
package class121;

// 消防(迭代版)
// 一共 n 个节点，编号 1~n，有 n-1 条边连接成一棵树，每条边上有非负权值
// 给定一个非负整数 s，表示可以在树上选择一条长度不超过 s 的路径
// 然后在这条路径的点上建立消防站，每个居民可以去往这条路径上的任何消防站
// 目标：哪怕最远的居民走到消防站的距离也要尽量少
// 返回最远居民走到消防站的最短距离
// 测试链接 : https://www.luogu.com.cn/problem/P2491
// 提交以下的 code，提交时请把类名改成"Main"，可以通过所有用例

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_FireFighting2 {

    public static int MAXN = 300001;

    public static int n;

    public static int s;

    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[] weight = new int[MAXN << 1];

    public static int cnt;

    public static int start;

    public static int end;

    public static int diameter;

    public static int[] dist = new int[MAXN];

    public static int[] last = new int[MAXN];

    public static int[] pred = new int[MAXN];

    public static boolean[] diameterPath = new boolean[MAXN];

    public static int[] maxDist = new int[MAXN];

    public static void build() {
        cnt = 1;
        Arrays.fill(head, 1, n + 1, 0);
        Arrays.fill(diameterPath, 1, n, false);
    }

    public static void addEdge(int u, int v, int w) {
        next[cnt] = head[u];
```

```
    to[cnt] = v;
    weight[cnt] = w;
    head[u] = cnt++;
}

public static void road() {
    dfs(1);
    start = 1;
    for (int i = 2; i <= n; i++) {
        if (dist[i] > dist[start]) {
            start = i;
        }
    }
    dfs(start);
    end = 1;
    for (int i = 2; i <= n; i++) {
        if (dist[i] > dist[end]) {
            end = i;
        }
    }
    diameter = dist[end];
}

// dfs 方法改迭代版
// 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];

public static int stackSize;

public static int u, f, w, e;

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

public static void pop() {
    --stackSize;
    u = ufwe[stackSize][0];
    f = ufwe[stackSize][1];
```

```
    w = ufwe[stackSize][2];
    e = ufwe[stackSize][3];
}

public static void dfs(int root) {
    stackSize = 0;
    push(root, 0, 0, -1);
    while (stackSize > 0) {
        pop();
        if (e == -1) {
            last[u] = f;
            dist[u] = dist[f] + w;
            pred[u] = w;
            e = head[u];
        } else {
            e = next[e];
        }
        if (e != 0) {
            push(u, f, w, e);
            if (to[e] != f) {
                push(to[e], u, weight[e], -1);
            }
        }
    }
}

public static void distance() {
    for (int i = end; i != 0; i = last[i]) {
        diameterPath[i] = true;
    }
    for (int i = end; i != 0; i = last[i]) {
        maxDist[i] = maxDistanceExceptDiameter(i, 0, 0);
    }
}

// maxDistanceExceptDiameter 方法不用改迭代居然能通过
// 那就不改了
public static int maxDistanceExceptDiameter(int u, int f, int c) {
    int ans = c;
    for (int e = head[u], v; e != 0; e = next[e]) {
        v = to[e];
        if (!diameterPath[v] && v != f) {
            ans = Math.max(ans, maxDistanceExceptDiameter(v, u, c + weight[e]));
```

```
        }
    }
    return ans;
}

public static int[] queue = new int[MAXN];

public static int compute() {
    int suml = 0, sumr = 0;
    int h = 0, t = 0;
    int ans = Integer.MAX_VALUE;
    for (int l = end, r = end; l != 0; l = last[l]) {
        while (r != 0 && sumr + pred[r] - suml <= s) {
            while (h < t && maxDist[queue[t - 1]] <= maxDist[r]) {
                t--;
            }
            sumr += pred[r];
            queue[t++] = r;
            r = last[r];
        }
        ans = Math.min(ans, Math.max(Math.max(suml, maxDist[queue[h]]), diameter - sumr));
        if (queue[h] == l) {
            h++;
        }
        suml += pred[l];
    }
    return ans;
}

public static void main(String[] args) throws IOException {
    BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
    StreamTokenizer in = new StreamTokenizer(br);
    PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));
    in.nextToken();
    n = (int) in.nval;
    in.nextToken();
    s = (int) in.nval;
    build();
    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;
            addEdge(u, v, w);
            addEdge(v, u, w);
        }
        road();
        distance();
        out.println(compute());
        out.flush();
        out.close();
        br.close();
    }

}
```

==========

文件: Codeforces1499F_DiameterCuts.cpp

==========

```
// Codeforces 1499F Diameter Cuts
// 题目：给定一棵 n 个节点的树和一个整数 k，计算有多少个连通子图的直径恰好为 k。
// 树的直径是指树中任意两点之间最长的简单路径。
// 来源：Codeforces Educational Round 106 Problem F
// 链接：https://codeforces.com/contest/1499/problem/F

#define MAXN 5001
#define MOD 998244353

// 邻接表存储树
int head[MAXN], next[MAXN << 1], to[MAXN << 1], cnt;
int n, k;  // 节点数和目标直径

// DP 状态
// f[u][i] 表示以 u 为根的子树中，所有连通子图都合法且从 u 向下延伸的最长路径长度为 i 的方案数
long long f[MAXN][MAXN];
int size[MAXN];  // 子树大小
long long g[MAXN];    // 临时数组用于 DP 转移

// 添加边
void addEdge(int u, int v) {
    next[cnt] = head[u];
    to[cnt] = v;
    head[u] = cnt++;
```

```
}

// 初始化
void init() {
    cnt = 0;
    for (int i = 1; i <= n; i++) {
        head[i] = -1;
    }
}

/**
 * 树形DP求解
 * @param u 当前节点
 * @param parent 父节点
 *
 * 时间复杂度：O(n^2)
 * 空间复杂度：O(n^2)
 */
void dfs(int u, int parent) {
    // 初始化当前节点的DP状态
    size[u] = 1;
    f[u][0] = 1;  // 只选择节点u本身

    // 遍历所有子节点
    for (int e = head[u]; e != -1; e = next[e]) {
        int v = to[e];
        if (v != parent) {
            dfs(v, u);

            // DP转移
            // 清空临时数组
            for (int i = 0; i <= k; i++) {
                g[i] = 0;
            }

            // 合并u和v的子树信息
            for (int i = 0; i <= (k < size[u] ? k : size[u]); i++) {
                for (int j = 0; j <= (size[v] < k - i ? size[v] : k - i); j++) {
                    // 合并后的最长路径长度为max(i, j+1)
                    // j+1是因为连接u和v需要增加1条边
                    int newLength = (i > j + 1) ? i : j + 1;
                    if (newLength <= k) {
                        g[newLength] = (g[newLength] + f[u][i] * f[v][j]) % MOD;
```

```
                    }
                }
            }

            // 更新 u 的子树大小和 DP 状态
            size[u] += size[v];
            for (int i = 0; i <= (k < size[u] ? k : size[u]); i++) {
                f[u][i] = g[i];
            }
        }
    }

    // 计算以 u 为根的子树中所有合法连通子图的总数
    long long sum = 0;
    for (int i = 0; i <= (k < size[u] ? k : size[u]); i++) {
        sum = (sum + f[u][i]) % MOD;
    }

    // 如果不是根节点，需要调整 DP 状态
    if (u != 1) {
        // 将所有路径长度加 1（因为要连接到父节点）
        for (int i = (k < size[u] ? k : size[u]); i > 0; i--) {
            f[u][i] = f[u][i - 1];
        }
        // 不连接到父节点的情况
        f[u][0] = sum;
    }
}

/**
 * 主方法
 * 时间复杂度：O(n^2)
 * 空间复杂度：O(n^2)
 */
int main() {
    // 由于编译环境限制，这里只展示算法实现
    // 实际使用时需要根据具体环境添加输入输出代码

    // 示例：n = 4, k = 0, 边为 1-2, 2-3, 3-4
    n = 4;
    k = 0;
    init();
```

```
    addEdge(1, 2);
    addEdge(2, 1);
    addEdge(2, 3);
    addEdge(3, 2);
    addEdge(3, 4);
    addEdge(4, 3);

    // 执行树形DP
    dfs(1, 0);

    // 计算结果
    long long result = 0;
    for (int i = 0; i <= k; i++) {
        result = (result + f[1][i]) % MOD;
    }

    // printf("%lld\n", result); // 应该输出某种结果

    return 0;
}
```

==========

文件: Codeforces1499F_DiameterCuts.java

==========

```
package class121;

// Codeforces 1499F Diameter Cuts
// 题目：给定一棵n个节点的树和一个整数k，计算有多少个连通子图的直径恰好为k。
// 树的直径是指树中任意两点之间最长的简单路径。
// 来源：Codeforces Educational Round 106 Problem F
// 链接：https://codeforces.com/contest/1499/problem/F

// 算法标签：树、动态规划、树形DP
// 难度：困难
// 时间复杂度：O(n^2)，其中n是树中节点的数量
// 空间复杂度：O(n^2)，用于存储DP状态

// 相关题目：
// - LeetCode 543. 二叉树的直径
// - LeetCode 1245. Tree Diameter (无向树的直径)
// - LeetCode 1522. Diameter of N-Ary Tree (N叉树的直径)
// - SPOJ PT07Z - Longest path in a tree (树中最长路径)
```

// - CSES 1131 - Tree Diameter (树的直径)
// - 51Nod 2602 - 树的直径
// - 洛谷 U81904 树的直径
// - AtCoder ABC221F - Diameter Set

// 解题思路：
// 使用树形DP，对每个节点计算子树中满足条件的连通子图数量。
// 状态定义：f[u][i] 表示以u为根的子树中，所有连通子图都合法且从u向下延伸的最长路径长度为i的方案数

```
import java.io.*;
import java.util.*;

public class Codeforces1499F_DiameterCuts {

    static final int MAXN = 5001;
    static final int MOD = 998244353;

    // 邻接表存储树
    static ArrayList<Integer>[] graph;
    static int n, k;  // 节点数和目标直径

    // DP状态
    // f[u][i] 表示以u为根的子树中，所有连通子图都合法且从u向下延伸的最长路径长度为i的方案数
    static long[][] f;
    static int[] size;  // 子树大小
    static long[] g;    // 临时数组用于DP转移

    /**
     * 树形DP求解
     *
     * 算法思路：
     * 1. 对每个节点u，维护f[u][i]表示以u为根的子树中，所有连通子图都合法且从u向下延伸的最长路径长度为i的方案数
     * 2. 对于每个节点u的子节点v，合并u和v的子树信息
     * 3. 合并时考虑连接u和v需要增加1条边，所以路径长度为max(i, j+1)
     * 4. 如果合并后的路径长度超过k，则不合法
     *
     * @param u 当前节点
     * @param parent 父节点
     *
     * 时间复杂度：O(n^2)
     * 空间复杂度：O(n^2)
```

```
 */
static void dfs(int u, int parent) {
    // 初始化当前节点的DP状态
    size[u] = 1;
    f[u][0] = 1;  // 只选择节点u本身

    // 遍历所有子节点
    for (int v : graph[u]) {
        if (v != parent) {
            dfs(v, u);

            // DP转移
            // 清空临时数组
            Arrays.fill(g, 0);

            // 合并u和v的子树信息
            for (int i = 0; i <= Math.min(k, size[u]); i++) {
                for (int j = 0; j <= Math.min(size[v], k - i); j++) {
                    // 合并后的最长路径长度为max(i, j+1)
                    // j+1是因为连接u和v需要增加1条边
                    int newLength = Math.max(i, j + 1);
                    if (newLength <= k) {
                        g[newLength] = (g[newLength] + f[u][i] * f[v][j]) % MOD;
                    }
                }
            }

            // 更新u的子树大小和DP状态
            size[u] += size[v];
            for (int i = 0; i <= Math.min(k, size[u]); i++) {
                f[u][i] = g[i];
            }
        }
    }

    // 计算以u为根的子树中所有合法连通子图的总数
    long sum = 0;
    for (int i = 0; i <= Math.min(k, size[u]); i++) {
        sum = (sum + f[u][i]) % MOD;
    }

    // 如果不是根节点，需要调整DP状态
    if (u != 1) {
```

```
            // 将所有路径长度加 1（因为要连接到父节点）
            for (int i = Math.min(k, size[u]); i > 0; i--) {
                f[u][i] = f[u][i - 1];
            }
            // 不连接到父节点的情况
            f[u][0] = sum;
        }
    }

    /**
     * 主方法
     *
     * 输入格式：
     * - 第一行包含两个整数 n 和 k，表示树中节点的数量和目标直径
     * - 接下来 n-1 行，每行包含两个整数 u 和 v，表示节点 u 和 v 之间有一条边
     *
     * 输出格式：
     * - 输出一个整数，表示满足条件的连通子图数量，对 998244353 取模
     *
     * 时间复杂度：O(n^2)
     * 空间复杂度：O(n^2)
     */
    public static void main(String[] args) throws IOException {
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));

        // 读取节点数和目标直径
        String[] line = br.readLine().split(" ");
        n = Integer.parseInt(line[0]);
        k = Integer.parseInt(line[1]);

        // 初始化数据结构
        graph = new ArrayList[n + 1];
        for (int i = 1; i <= n; i++) {
            graph[i] = new ArrayList<>();
        }

        f = new long[n + 1][n + 1];
        size = new int[n + 1];
        g = new long[n + 1];

        // 读取边信息
        for (int i = 1; i < n; i++) {
```

```
            line = br.readLine().split(" ");
            int u = Integer.parseInt(line[0]);
            int v = Integer.parseInt(line[1]);
            graph[u].add(v);
            graph[v].add(u);
        }

        // 执行树形DP
        dfs(1, 0);

        // 计算并输出结果
        long result = 0;
        for (int i = 0; i <= k; i++) {
            result = (result + f[1][i]) % MOD;
        }
        out.println(result);

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

==========

文件: Codeforces1499F_DiameterCuts.py

==========

```
# Codeforces 1499F Diameter Cuts
# 题目：给定一棵n个节点的树和一个整数k，计算有多少个连通子图的直径恰好为k。
# 树的直径是指树中任意两点之间最长的简单路径。
# 来源：Codeforces Educational Round 106 Problem F
# 链接：https://codeforces.com/contest/1499/problem/F

from collections import defaultdict

MOD = 998244353

class Codeforces1499FDiameterCuts:
    def __init__(self):
        self.n = 0  # 节点数
        self.k = 0  # 目标直径
        self.graph = defaultdict(list)  # 邻接表存储树
        self.f = []  # DP状态数组
```

```
        self.size = []  # 子树大小数组
        self.g = []  # 临时数组用于DP转移

    def add_edge(self, u, v):
        """添加无向边"""
        self.graph[u].append(v)
        self.graph[v].append(u)

    def dfs(self, u, parent):
        """
        树形DP求解
        :param u: 当前节点
        :param parent: 父节点
        :return: None

        时间复杂度：O(n^2)
        空间复杂度：O(n^2)
        """
        # 初始化当前节点的DP状态
        self.size[u] = 1
        self.f[u][0] = 1  # 只选择节点u本身

        # 遍历所有子节点
        for v in self.graph[u]:
            if v != parent:
                self.dfs(v, u)

                # DP转移
                # 清空临时数组
                for i in range(self.k + 1):
                    self.g[i] = 0

                # 合并u和v的子树信息
                for i in range(min(self.k, self.size[u]) + 1):
                    for j in range(min(self.size[v], self.k - i) + 1):
                        # 合并后的最长路径长度为max(i, j+1)
                        # j+1是因为连接u和v需要增加1条边
                        new_length = max(i, j + 1)
                        if new_length <= self.k:
                            self.g[new_length] = (self.g[new_length] + self.f[u][i] *
self.f[v][j]) % MOD

                # 更新u的子树大小和DP状态
```

```
            self.size[u] += self.size[v]
            for i in range(min(self.k, self.size[u]) + 1):
                self.f[u][i] = self.g[i]

    # 计算以u为根的子树中所有合法连通子图的总数
    sum_val = 0
    for i in range(min(self.k, self.size[u]) + 1):
        sum_val = (sum_val + self.f[u][i]) % MOD

    # 如果不是根节点，需要调整DP状态
    if u != 1:
        # 将所有路径长度加1（因为要连接到父节点）
        for i in range(min(self.k, self.size[u]), 0, -1):
            self.f[u][i] = self.f[u][i - 1]
        # 不连接到父节点的情况
        self.f[u][0] = sum_val

def solve(self):
    """
    主方法
    时间复杂度：O(n^2)
    空间复杂度：O(n^2)
    """
    # 初始化DP数组
    self.f = [[0] * (self.n + 1) for _ in range(self.n + 1)]
    self.size = [0] * (self.n + 1)
    self.g = [0] * (self.n + 1)

    # 执行树形DP
    self.dfs(1, 0)

    # 计算并返回结果
    result = 0
    for i in range(self.k + 1):
        result = (result + self.f[1][i]) % MOD
    return result

def read_input_and_solve(self):
    """
    读取输入并求解
    """
    # 读取节点数和目标直径
    line = input().split()
```

```
        self.n = int(line[0])
        self.k = int(line[1])

        # 读取边信息
        for _ in range(self.n - 1):
            line = input().split()
            u = int(line[0])
            v = int(line[1])
            self.add_edge(u, v)

        # 计算并输出结果
        print(self.solve())

# 主函数
if __name__ == "__main__":
    # 由于这是在线评测题目，实际提交时需要取消下面的注释
    # solution = Codeforces1499FDiameterCuts()
    # solution.read_input_and_solve()

    # 示例测试
    solution = Codeforces1499FDiameterCuts()
    solution.n = 4
    solution.k = 0
    solution.add_edge(1, 2)
    solution.add_edge(2, 3)
    solution.add_edge(3, 4)
    print("示例输出:", solution.solve())

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

文件: CSES1131_TreeDiameter.cpp
==============================================
// CSES 1131 Tree Diameter
// 题目：给定一棵树，求树的直径（树中任意两点间最长的简单路径）
// 来源：CSES Problem Set - Tree Algorithms
// 链接：https://cses.fi/problemset/task/1131

#define MAXN 200001
#define MAXM 400001

// 邻接表存储树
int head[MAXN], next[MAXM], to[MAXM], cnt;
int n;  // 节点数
```

```
// 队列实现 BFS
int queue[MAXN], front, rear;
int visited[MAXN];

// Pair 结构体，用于存储节点和距离
struct Pair {
    int node;
    int distance;
};

// 添加边
void addEdge(int u, int v) {
    next[cnt] = head[u];
    to[cnt] = v;
    head[u] = cnt++;
}

// 初始化队列
void initQueue() {
    front = rear = 0;
}

// 入队
void enqueue(int x) {
    queue[rear++] = x;
}

// 出队
int dequeue() {
    return queue[front++];
}

// 判断队列是否为空
int isEmpty() {
    return front == rear;
}

// 获取队列大小
int queueSize() {
    return rear - front;
}
```

```
/**
 * BFS 求从起点开始的最远节点
 * @param start 起点
 * @return Pair 对象，包含最远节点和距离
 *
 * 时间复杂度：O(n)
 * 空间复杂度：O(n)
 */
struct Pair bfs(int start) {
    struct Pair result;

    // 初始化访问数组
    for (int i = 1; i <= n; i++) {
        visited[i] = 0;
    }

    initQueue();

    visited[start] = 1;
    enqueue(start);

    int lastNode = start;
    int maxDistance = 0;

    while (!isEmpty()) {
        int size = queueSize();
        for (int i = 0; i < size; i++) {
            int current = dequeue();
            lastNode = current;

            // 遍历当前节点的所有邻居
            for (int e = head[current]; e != 0; e = next[e]) {
                int neighbor = to[e];
                if (!visited[neighbor]) {
                    visited[neighbor] = 1;
                    enqueue(neighbor);
                }
            }
        }
        if (!isEmpty()) {
            maxDistance++;
        }
    }
```

```
    result.node = lastNode;
    result.distance = maxDistance;
    return result;
}

/**
 * 使用两次 BFS 法求树的直径
 * @return 树的直径
 *
 * 时间复杂度：O(n)
 * 空间复杂度：O(n)
 */
int findDiameter() {
    // 第一次 BFS，从节点 1 开始找到最远节点
    struct Pair firstBFS = bfs(1);

    // 第二次 BFS，从第一次找到的最远节点开始找到另一个最远节点
    struct Pair secondBFS = bfs(firstBFS.node);

    // 第二次 BFS 的距离就是树的直径
    return secondBFS.distance;
}

/**
 * 主方法
 * 时间复杂度：O(n)
 * 空间复杂度：O(n)
 */
int main() {
    // 由于编译环境限制，这里只展示算法实现
    // 实际使用时需要根据具体环境添加输入输出代码

    // 示例：n = 4, 边为 1-2, 2-3, 3-4
    n = 4;
    cnt = 1;
    for (int i = 1; i <= n; i++) {
        head[i] = 0;
    }

    addEdge(1, 2);
    addEdge(2, 1);
    addEdge(2, 3);
```

```
    addEdge(3, 2);
    addEdge(3, 4);
    addEdge(4, 3);

    // 计算并输出树的直径
    // printf("%d\n", findDiameter()); // 应该输出 3

    return 0;
}
```

==========

文件: CSES1131_TreeDiameter.java

==========

```
package class121;

// CSES 1131 Tree Diameter
// 题目：给定一棵树，求树的直径（树中任意两点间最长的简单路径）
// 来源：CSES Problem Set - Tree Algorithms
// 链接：https://cses.fi/problemset/task/1131
// 提交时请把类名改成"Main"

// 算法标签：树、广度优先搜索、两次 BFS
// 难度：简单
// 时间复杂度：O(n)，其中 n 是树中节点的数量
// 空间复杂度：O(n)，用于存储邻接表和辅助数组

// 相关题目：
// - LeetCode 543. 二叉树的直径
// - LeetCode 1245. Tree Diameter (无向树的直径)
// - LeetCode 1522. Diameter of N-Ary Tree (N 叉树的直径)
// - SPOJ PT07Z - Longest path in a tree (树中最长路径)
// - 51Nod 2602 - 树的直径
// - 洛谷 U81904 树的直径
// - AtCoder ABC221F - Diameter Set

// 解题思路：
// 使用两次 BFS 法求解树的直径：
// 1. 从任意一点开始，找到距离它最远的点 s
// 2. 从 s 开始，找到距离它最远的点 t
// 3. s 到 t 的距离即为树的直径

import java.io.*;
```

```
import java.util.*;

public class CSES1131_TreeDiameter {

    static final int MAXN = 200001;

    // 邻接表存储树
    static ArrayList<Integer>[] graph;
    static int n;  // 节点数

    // BFS 方法求从起点开始的最远节点和距离
    static class Pair {
        int node;
        int distance;

        Pair(int node, int distance) {
            this.node = node;
            this.distance = distance;
        }
    }

    /**
     * BFS 求从起点开始的最远节点
     *
     * 算法思路：
     * 1. 从指定起点开始进行广度优先搜索
     * 2. 记录访问过的节点，避免重复访问
     * 3. 记录每一层的节点，直到遍历完所有节点
     * 4. 返回最后一层的节点（最远节点）和距离
     *
     * @param start 起点
     * @return Pair 对象，包含最远节点和距离
     *
     * 时间复杂度：O(n)
     * 空间复杂度：O(n)
     */
    static Pair bfs(int start) {
        boolean[] visited = new boolean[n + 1];
        Queue<Integer> queue = new LinkedList<>();

        visited[start] = true;
        queue.offer(start);
```

```
    int lastNode = start;
    int maxDistance = 0;

    while (!queue.isEmpty()) {
        int size = queue.size();
        for (int i = 0; i < size; i++) {
            int current = queue.poll();
            lastNode = current;

            // 遍历当前节点的所有邻居
            for (int neighbor : graph[current]) {
                if (!visited[neighbor]) {
                    visited[neighbor] = true;
                    queue.offer(neighbor);
                }
            }
        }
        if (!queue.isEmpty()) {
            maxDistance++;
        }
    }

    return new Pair(lastNode, maxDistance);
}

/**
 * 使用两次 BFS 法求树的直径
 *
 * 算法思路：
 * 1. 第一次 BFS，从任意节点（如节点 1）开始找到最远节点
 * 2. 第二次 BFS，从第一次找到的最远节点开始找到另一个最远节点
 * 3. 第二次 BFS 的距离就是树的直径
 *
 * @return 树的直径
 *
 * 时间复杂度：O(n)
 * 空间复杂度：O(n)
 */
static int findDiameter() {
    // 第一次 BFS，从节点 1 开始找到最远节点
    Pair firstBFS = bfs(1);

    // 第二次 BFS，从第一次找到的最远节点开始找到另一个最远节点
```

```
    Pair secondBFS = bfs(firstBFS.node);

    // 第二次 BFS 的距离就是树的直径
    return secondBFS.distance;
}

/**
 * 主方法
 *
 * 输入格式：
 * - 第一行包含一个整数 n，表示树中节点的数量
 * - 接下来 n-1 行，每行包含两个整数 u 和 v，表示节点 u 和 v 之间有一条边
 *
 * 输出格式：
 * - 输出一个整数，表示树的直径
 *
 * 时间复杂度：O(n)
 * 空间复杂度：O(n)
 */
public static void main(String[] args) throws IOException {
    BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
    PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));

    // 读取节点数
    n = Integer.parseInt(br.readLine());

    // 初始化邻接表
    graph = new ArrayList[n + 1];
    for (int i = 1; i <= n; i++) {
        graph[i] = new ArrayList<>();
    }

    // 读取边信息
    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());
        graph[u].add(v);
        graph[v].add(u);
    }

    // 计算并输出树的直径
    out.println(findDiameter());
```

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

==========

文件: CSES1131_TreeDiameter.py

==========

```
# CSES 1131 Tree Diameter
# 题目：给定一棵树，求树的直径（树中任意两点间最长的简单路径）
# 来源：CSES Problem Set - Tree Algorithms
# 链接：https://cses.fi/problemset/task/1131

from collections import deque, defaultdict

class CS ESTreeDiameter:
    def __init__(self):
        self.n = 0  # 节点数
        self.graph = defaultdict(list)  # 邻接表存储树

    def add_edge(self, u, v):
        """添加无向边"""
        self.graph[u].append(v)
        self.graph[v].append(u)

    def bfs(self, start):
        """
        BFS 求从起点开始的最远节点
        :param start: 起点
        :return: (最远节点, 距离)

        时间复杂度：O(n)
        空间复杂度：O(n)
        """
        visited = [False] * (self.n + 1)
        queue = deque([(start, 0)])  # (节点, 距离)
        visited[start] = True

        farthest_node = start
        max_distance = 0
```

```
    while queue:
        current, distance = queue.popleft()
        farthest_node = current
        max_distance = distance

        # 遍历当前节点的所有邻居
        for neighbor in self.graph[current]:
            if not visited[neighbor]:
                visited[neighbor] = True
                queue.append((neighbor, distance + 1))

    return farthest_node, max_distance

def find_diameter(self):
    """
    使用两次 BFS 法求树的直径
    :return: 树的直径

    时间复杂度：O(n)
    空间复杂度：O(n)
    """
    # 第一次 BFS，从节点 1 开始找到最远节点
    first_node, _ = self.bfs(1)

    # 第二次 BFS，从第一次找到的最远节点开始找到另一个最远节点
    _, diameter = self.bfs(first_node)

    return diameter

def solve(self):
    """
    主方法
    时间复杂度：O(n)
    空间复杂度：O(n)
    """
    # 读取节点数
    self.n = int(input())

    # 读取边信息
    for _ in range(self.n - 1):
        u, v = map(int, input().split())
        self.add_edge(u, v)
```

```
        # 计算并输出树的直径
        print(self.find_diameter())

# 主函数
if __name__ == "__main__":
    # 由于这是在线评测题目，实际提交时需要取消下面的注释
    # solution = CSES1131TreeDiameter()
    # solution.solve()

    # 示例测试
    solution = CSES1131TreeDiameter()
    solution.n = 4
    solution.add_edge(1, 2)
    solution.add_edge(2, 3)
    solution.add_edge(3, 4)
    print("示例输出:", solution.find_diameter())  # 应该输出 3
```

==========

文件: LeetCode1245_TreeDiameter.cpp

==========

```
// LeetCode 1245. 树的直径（非二叉树版本）
// 题目：给你一棵树，树中包含 n 个节点，节点编号从 0 到 n-1。
// 树用一个边列表来表示，其中 edges[i] = [u, v] 表示节点 u 和 v 之间有一条无向边。
// 返回这棵树的直径长度。
// 树的直径是树中任意两个节点之间最长路径的长度。
// 这条路径可能不经过根节点。
// 来源：LeetCode
// 链接：https://leetcode.cn/problems/tree-diameter/

#include <iostream>
#include <vector>
#include <queue>
#include <algorithm>
#include <cstring>

using namespace std;

class LeetCode1245_TreeDiameter {
public:
    /**
     * 计算树的直径（两次 BFS 法）
     * @param edges 边列表，表示树结构
```

```
 * @return 树的直径长度
 *
 * 时间复杂度：O(n)，其中 n 是节点数
 * 空间复杂度：O(n)，用于存储邻接表和 BFS 队列
 */
int treeDiameter(vector<vector<int>>& edges) {
    int n = edges.size() + 1; // 节点数 = 边数 + 1

    // 特殊情况处理
    if (n == 0) return 0;
    if (n == 1) return 0;

    // 构建邻接表
    vector<vector<int>> graph(n);

    for (auto& edge : edges) {
        int u = edge[0];
        int v = edge[1];
        graph[u].push_back(v);
        graph[v].push_back(u);
    }

    // 第一次 BFS：从任意节点（如 0）出发，找到最远的节点 A
    auto firstBFS = bfs(0, graph, n);
    int nodeA = firstBFS.first;

    // 第二次 BFS：从节点 A 出发，找到最远的节点 B，距离就是直径
    auto secondBFS = bfs(nodeA, graph, n);

    return secondBFS.second; // 返回直径长度
}

/**
 * BFS 方法，返回最远节点和距离
 * @param start 起始节点
 * @param graph 邻接表
 * @param n 节点总数
 * @return pair，第一个元素是最远节点，第二个元素是距离
 */
pair<int, int> bfs(int start, vector<vector<int>>& graph, int n) {
    vector<int> distance(n, -1);
    distance[start] = 0;
```

```
    queue<int> q;
    q.push(start);

    int farthestNode = start;
    int maxDistance = 0;

    while (!q.empty()) {
        int current = q.front();
        q.pop();

        // 遍历当前节点的所有邻居
        for (int neighbor : graph[current]) {
            if (distance[neighbor] == -1) {
                distance[neighbor] = distance[current] + 1;
                q.push(neighbor);

                // 更新最远节点和最大距离
                if (distance[neighbor] > maxDistance) {
                    maxDistance = distance[neighbor];
                    farthestNode = neighbor;
                }
            }
        }
    }

    return {farthestNode, maxDistance};
}

/**
 * 树形DP方法计算树的直径（可以处理负权边）
 * @param edges 边列表
 * @return 树的直径长度
 *
 * 时间复杂度：O(n)
 * 空间复杂度：O(n)
 */
int treeDiameterDP(vector<vector<int>>& edges) {
    int n = edges.size() + 1;

    // 特殊情况处理
    if (n == 0) return 0;
    if (n == 1) return 0;
```

```
    // 构建邻接表
    vector<vector<int>> graph(n);

    for (auto& edge : edges) {
        int u = edge[0];
        int v = edge[1];
        graph[u].push_back(v);
        graph[v].push_back(u);
    }

    // 全局变量记录最大直径
    int maxDiameter = 0;

    // DFS 计算每个节点的最大深度
    dfs(0, -1, graph, maxDiameter);

    return maxDiameter;
}

/**
 * DFS 计算节点深度并更新最大直径
 * @param node 当前节点
 * @param parent 父节点
 * @param graph 邻接表
 * @param maxDiameter 全局最大直径
 * @return 当前节点的最大深度
 */
int dfs(int node, int parent, vector<vector<int>>& graph, int& maxDiameter) {
    int maxDepth1 = 0; // 最大深度
    int maxDepth2 = 0; // 次大深度

    for (int neighbor : graph[node]) {
        if (neighbor != parent) {
            int depth = dfs(neighbor, node, graph, maxDiameter);

            if (depth > maxDepth1) {
                maxDepth2 = maxDepth1;
                maxDepth1 = depth;
            } else if (depth > maxDepth2) {
                maxDepth2 = depth;
            }
        }
    }
```

```
        // 更新最大直径：经过当前节点的最长路径 = maxDepth1 + maxDepth2
        maxDiameter = max(maxDiameter, maxDepth1 + maxDepth2);

        // 返回当前节点的最大深度
        return maxDepth1 + 1;
    }

    // 测试方法
    void test() {
        LeetCode1245_TreeDiameter solution;

        // 测试用例 1: [[0,1],[0,2]]
        // 树结构：
        //   0
        //  / \
        // 1   2
        // 预期输出：2（路径 1-0-2）
        vector<vector<int>> edges1 = {{0,1},{0,2}};
        cout << "测试用例 1 结果: " << solution.treeDiameter(edges1) << endl; // 应该输出 2
        cout << "测试用例 1(DP)结果: " << solution.treeDiameterDP(edges1) << endl; // 应该输出 2

        // 测试用例 2: [[0,1],[1,2],[2,3],[1,4],[4,5]]
        // 树结构：
        //     0
        //     |
        //     1
        //    / \
        //   2   4
        //  /     \
        // 3       5
        // 预期输出：4（路径 3-2-1-4-5）
        vector<vector<int>> edges2 = {{0,1},{1,2},{2,3},{1,4},{4,5}};
        cout << "测试用例 2 结果: " << solution.treeDiameter(edges2) << endl; // 应该输出 4
        cout << "测试用例 2(DP)结果: " << solution.treeDiameterDP(edges2) << endl; // 应该输出 4

        // 测试用例 3: 单节点树
        vector<vector<int>> edges3 = {};
        cout << "测试用例 3 结果: " << solution.treeDiameter(edges3) << endl; // 应该输出 0
        cout << "测试用例 3(DP)结果: " << solution.treeDiameterDP(edges3) << endl; // 应该输出 0
    }
};
```

```
int main() {
    LeetCode1245_TreeDiameter solution;
    solution.test();
    return 0;
}
```

==========

文件: LeetCode1245_TreeDiameter.java

==========

```
package class121;

// LeetCode 1245. 树的直径（非二叉树版本）
// 题目：给你一棵树，树中包含 n 个节点，节点编号从 0 到 n-1。
// 树用一个边列表来表示，其中 edges[i] = [u, v] 表示节点 u 和 v 之间有一条无向边。
// 返回这棵树的直径长度。
// 树的直径是树中任意两个节点之间最长路径的长度。
// 这条路径可能不经过根节点。
// 来源：LeetCode
// 链接：https://leetcode.cn/problems/tree-diameter/

import java.util.*;

public class LeetCode1245_TreeDiameter {

    /**
     * 计算树的直径（两次 BFS 法）
     * @param edges 边列表，表示树结构
     * @return 树的直径长度
     *
     * 时间复杂度：O(n)，其中 n 是节点数
     * 空间复杂度：O(n)，用于存储邻接表和 BFS 队列
     */
    public int treeDiameter(int[][] edges) {
        int n = edges.length + 1; // 节点数 = 边数 + 1

        // 特殊情况处理
        if (n == 0) return 0;
        if (n == 1) return 0;

        // 构建邻接表
        List<Integer>[] graph = new ArrayList[n];
        for (int i = 0; i < n; i++) {
```

```
        graph[i] = new ArrayList<>();
    }

    for (int[] edge : edges) {
        int u = edge[0];
        int v = edge[1];
        graph[u].add(v);
        graph[v].add(u);
    }

    // 第一次 BFS：从任意节点（如 0）出发，找到最远的节点 A
    int[] firstBFS = bfs(0, graph, n);
    int nodeA = firstBFS[0];

    // 第二次 BFS：从节点 A 出发，找到最远的节点 B，距离就是直径
    int[] secondBFS = bfs(nodeA, graph, n);

    return secondBFS[1]; // 返回直径长度
}

/**
 * BFS 方法，返回最远节点和距离
 * @param start 起始节点
 * @param graph 邻接表
 * @param n 节点总数
 * @return 数组，第一个元素是最远节点，第二个元素是距离
 */
private int[] bfs(int start, List<Integer>[] graph, int n) {
    int[] distance = new int[n];
    Arrays.fill(distance, -1);
    distance[start] = 0;

    Queue<Integer> queue = new LinkedList<>();
    queue.offer(start);

    int farthestNode = start;
    int maxDistance = 0;

    while (!queue.isEmpty()) {
        int current = queue.poll();

        // 遍历当前节点的所有邻居
        for (int neighbor : graph[current]) {
```

```
            if (distance[neighbor] == -1) {
                distance[neighbor] = distance[current] + 1;
                queue.offer(neighbor);

                // 更新最远节点和最大距离
                if (distance[neighbor] > maxDistance) {
                    maxDistance = distance[neighbor];
                    farthestNode = neighbor;
                }
            }
        }
    }

    return new int[]{farthestNode, maxDistance};
}

/**
 * 树形DP方法计算树的直径（可以处理负权边）
 * @param edges 边列表
 * @return 树的直径长度
 *
 * 时间复杂度：O(n)
 * 空间复杂度：O(n)
 */
public int treeDiameterDP(int[][] edges) {
    int n = edges.length + 1;

    // 特殊情况处理
    if (n == 0) return 0;
    if (n == 1) return 0;

    // 构建邻接表
    List<Integer>[] graph = new ArrayList[n];
    for (int i = 0; i < n; i++) {
        graph[i] = new ArrayList<>();
    }

    for (int[] edge : edges) {
        int u = edge[0];
        int v = edge[1];
        graph[u].add(v);
        graph[v].add(u);
    }
```

```
    // 全局变量记录最大直径
    int[] maxDiameter = {0};

    // DFS 计算每个节点的最大深度
    dfs(0, -1, graph, maxDiameter);

    return maxDiameter[0];
}

/**
 * DFS 计算节点深度并更新最大直径
 * @param node 当前节点
 * @param parent 父节点
 * @param graph 邻接表
 * @param maxDiameter 全局最大直径
 * @return 当前节点的最大深度
 */
private int dfs(int node, int parent, List<Integer>[] graph, int[] maxDiameter) {
    int maxDepth1 = 0; // 最大深度
    int maxDepth2 = 0; // 次大深度

    for (int neighbor : graph[node]) {
        if (neighbor != parent) {
            int depth = dfs(neighbor, node, graph, maxDiameter);

            if (depth > maxDepth1) {
                maxDepth2 = maxDepth1;
                maxDepth1 = depth;
            } else if (depth > maxDepth2) {
                maxDepth2 = depth;
            }
        }
    }

    // 更新最大直径：经过当前节点的最长路径 = maxDepth1 + maxDepth2
    maxDiameter[0] = Math.max(maxDiameter[0], maxDepth1 + maxDepth2);

    // 返回当前节点的最大深度
    return maxDepth1 + 1;
}

// 测试方法
```

```
    public static void main(String[] args) {
        LeetCode1245_TreeDiameter solution = new LeetCode1245_TreeDiameter();

        // 测试用例 1: [[0,1],[0,2]]
        // 树结构：
        //   0
        //  / \
        // 1   2
        // 预期输出：2（路径 1-0-2）
        int[][] edges1 = {{0,1},{0,2}};
        System.out.println("测试用例 1 结果: " + solution.treeDiameter(edges1)); // 应该输出 2
        System.out.println("测试用例 1(DP)结果: " + solution.treeDiameterDP(edges1)); // 应该输出
2

        // 测试用例 2: [[0,1],[1,2],[2,3],[1,4],[4,5]]
        // 树结构：
        //     0
        //     |
        //     1
        //    / \
        //   2   4
        //  /     \
        // 3       5
        // 预期输出：4（路径 3-2-1-4-5）
        int[][] edges2 = {{0,1},{1,2},{2,3},{1,4},{4,5}};
        System.out.println("测试用例 2 结果: " + solution.treeDiameter(edges2)); // 应该输出 4
        System.out.println("测试用例 2(DP)结果: " + solution.treeDiameterDP(edges2)); // 应该输出
4

        // 测试用例 3: 单节点树
        int[][] edges3 = {};
        System.out.println("测试用例 3 结果: " + solution.treeDiameter(edges3)); // 应该输出 0
        System.out.println("测试用例 3(DP)结果: " + solution.treeDiameterDP(edges3)); // 应该输出
0
    }
}

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

文件: LeetCode1245_TreeDiameter.py
==========================================
# LeetCode 1245. 树的直径（非二叉树版本）
# 题目：给你一棵树，树中包含 n 个节点，节点编号从 0 到 n-1。
```

```
# 树用一个边列表来表示，其中 edges[i] = [u, v] 表示节点 u 和 v 之间有一条无向边。
# 返回这棵树的直径长度。
# 树的直径是树中任意两个节点之间最长路径的长度。
# 这条路径可能不经过根节点。
# 来源：LeetCode
# 链接：https://leetcode.cn/problems/tree-diameter/

from collections import deque
from typing import List, Tuple

class LeetCode1245_TreeDiameter:
    """
    树的直径计算类
    提供两种方法：两次 BFS 法和树形 DP 法
    """

    def treeDiameter(self, edges: List[List[int]]) -> int:
        """
        计算树的直径（两次 BFS 法）

        Args:
            edges: 边列表，表示树结构

        Returns:
            int: 树的直径长度

        时间复杂度：O(n)，其中 n 是节点数
        空间复杂度：O(n)，用于存储邻接表和 BFS 队列
        """
        n = len(edges) + 1  # 节点数 = 边数 + 1

        # 特殊情况处理
        if n == 0:
            return 0
        if n == 1:
            return 0

        # 构建邻接表
        graph = [[] for _ in range(n)]

        for u, v in edges:
            graph[u].append(v)
            graph[v].append(u)
```

```
    # 第一次 BFS：从任意节点（如 0）出发，找到最远的节点 A
    nodeA, _ = self._bfs(0, graph, n)

    # 第二次 BFS：从节点 A 出发，找到最远的节点 B，距离就是直径
    _, diameter = self._bfs(nodeA, graph, n)

    return diameter

def _bfs(self, start: int, graph: List[List[int]], n: int) -> Tuple[int, int]:
    """
    BFS 方法，返回最远节点和距离

    Args:
        start: 起始节点
        graph: 邻接表
        n: 节点总数

    Returns:
        Tuple[int, int]: (最远节点, 距离)
    """
    distance = [-1] * n
    distance[start] = 0

    queue = deque([start])

    farthest_node = start
    max_distance = 0

    while queue:
        current = queue.popleft()

        # 遍历当前节点的所有邻居
        for neighbor in graph[current]:
            if distance[neighbor] == -1:
                distance[neighbor] = distance[current] + 1
                queue.append(neighbor)

                # 更新最远节点和最大距离
                if distance[neighbor] > max_distance:
                    max_distance = distance[neighbor]
                    farthest_node = neighbor
```

```
        return farthest_node, max_distance

    def treeDiameterDP(self, edges: List[List[int]]) -> int:
        """
        树形DP方法计算树的直径（可以处理负权边）

        Args:
            edges: 边列表

        Returns:
            int: 树的直径长度

        时间复杂度：O(n)
        空间复杂度：O(n)
        """
        n = len(edges) + 1

        # 特殊情况处理
        if n == 0:
            return 0
        if n == 1:
            return 0

        # 构建邻接表
        graph = [[] for _ in range(n)]

        for u, v in edges:
            graph[u].append(v)
            graph[v].append(u)

        # 全局变量记录最大直径
        max_diameter = [0]

        # DFS计算每个节点的最大深度
        self._dfs(0, -1, graph, max_diameter)

        return max_diameter[0]

    def _dfs(self, node: int, parent: int, graph: List[List[int]], max_diameter: List[int]) ->
int:
        """
        DFS计算节点深度并更新最大直径
```

```
    Args:
        node: 当前节点
        parent: 父节点
        graph: 邻接表
        max_diameter: 全局最大直径

    Returns:
        int: 当前节点的最大深度
    """
    max_depth1 = 0  # 最大深度
    max_depth2 = 0  # 次大深度

    for neighbor in graph[node]:
        if neighbor != parent:
            depth = self._dfs(neighbor, node, graph, max_diameter)

            if depth > max_depth1:
                max_depth2 = max_depth1
                max_depth1 = depth
            elif depth > max_depth2:
                max_depth2 = depth

    # 更新最大直径：经过当前节点的最长路径 = max_depth1 + max_depth2
    max_diameter[0] = max(max_diameter[0], max_depth1 + max_depth2)

    # 返回当前节点的最大深度
    return max_depth1 + 1

def test(self):
    """测试方法"""
    solution = LeetCode1245_TreeDiameter()

    # 测试用例 1: [[0,1],[0,2]]
    # 树结构：
    #   0
    #  / \
    # 1   2
    # 预期输出：2（路径 1-0-2）
    edges1 = [[0,1],[0,2]]
    print(f"测试用例 1 结果: {solution.treeDiameter(edges1)}")  # 应该输出 2
    print(f"测试用例 1(DP)结果: {solution.treeDiameterDP(edges1)}")  # 应该输出 2

    # 测试用例 2: [[0,1],[1,2],[2,3],[1,4],[4,5]]
```

```
        # 树结构：
        #     0
        #     |
        #     1
        #    / \
        #   2   4
        #  /     \
        # 3       5
        # 预期输出：4（路径 3-2-1-4-5）
        edges2 = [[0,1],[1,2],[2,3],[1,4],[4,5]]
        print(f"测试用例 2 结果: {solution.treeDiameter(edges2)}")  # 应该输出 4
        print(f"测试用例 2(DP)结果: {solution.treeDiameterDP(edges2)}")  # 应该输出 4

        # 测试用例 3: 单节点树
        edges3 = []
        print(f"测试用例 3 结果: {solution.treeDiameter(edges3)}")  # 应该输出 0
        print(f"测试用例 3(DP)结果: {solution.treeDiameterDP(edges3)}")  # 应该输出 0

# 主函数
if __name__ == "__main__":
    solution = LeetCode1245_TreeDiameter()
    solution.test()
```

```
================================================
```

文件: LeetCode1443_MinimumTimeToCollectApples.java

```
================================================
package class121;

// LeetCode 1443. 收集树上所有苹果的最少时间
// 题目：给你一棵有 n 个节点的无向树，节点编号为 0 到 n-1，它们中有一些节点有苹果。
// 给你一个二维整数数组 edges，其中 edges[i] = [ai, bi] 表示节点 ai 和 bi 之间有一条边。
// 另给你一个布尔数组 hasApple，其中 hasApple[i] = true 表示节点 i 有一个苹果，否则为 false。
// 你需要从节点 0 出发，收集树上所有苹果，并返回到节点 0 所需的最少时间（秒）。
// 每经过一条边需要 1 秒。
// 来源：LeetCode
// 链接：https://leetcode.cn/problems/minimum-time-to-collect-all-apples-in-a-tree/

import java.util.*;

public class LeetCode1443_MinimumTimeToCollectApples {

    /**
```

```
     * 计算收集所有苹果的最少时间
     * @param n 节点数量
     * @param edges 边列表
     * @param hasApple 苹果分布数组
     * @return 最少时间（秒）
     *
     * 时间复杂度：O(n)，其中 n 是节点数量
     * 空间复杂度：O(n)，用于存储邻接表和 DFS 递归栈
     */
    public int minTime(int n, int[][] edges, List<Boolean> hasApple) {
        // 构建邻接表
        List<Integer>[] graph = new ArrayList[n];
        for (int i = 0; i < n; i++) {
            graph[i] = new ArrayList<>();
        }

        for (int[] edge : edges) {
            int u = edge[0];
            int v = edge[1];
            graph[u].add(v);
            graph[v].add(u);
        }

        // 使用 DFS 计算需要访问的路径总长度
        boolean[] visited = new boolean[n];
        return dfs(0, -1, graph, hasApple, visited);
    }

    /**
     * DFS 遍历计算子树中收集苹果所需的时间
     * @param node 当前节点
     * @param parent 父节点
     * @param graph 邻接表
     * @param hasApple 苹果分布
     * @param visited 访问标记
     * @return 收集当前子树中苹果所需的时间
     */
    private int dfs(int node, int parent, List<Integer>[] graph, List<Boolean> hasApple,
boolean[] visited) {
        visited[node] = true;

        int totalTime = 0;
```

```
    // 遍历所有邻居（除了父节点）
    for (int neighbor : graph[node]) {
        if (neighbor != parent && !visited[neighbor]) {
            // 递归处理子树
            int childTime = dfs(neighbor, node, graph, hasApple, visited);

            // 如果子树中有苹果或者子树需要访问，则需要加上往返时间
            if (childTime > 0 || hasApple.get(neighbor)) {
                totalTime += childTime + 2; // 往返需要 2 秒
            }
        }
    }

    return totalTime;
}

/**
 * 使用 BFS 的迭代实现（避免递归深度过大）
 * @param n 节点数量
 * @param edges 边列表
 * @param hasApple 苹果分布
 * @return 最少时间
 *
 * 时间复杂度：O(n)
 * 空间复杂度：O(n)
 */
public int minTimeIterative(int n, int[][] edges, List<Boolean> hasApple) {
    // 构建邻接表
    List<Integer>[] graph = new ArrayList[n];
    for (int i = 0; i < n; i++) {
        graph[i] = new ArrayList<>();
    }

    for (int[] edge : edges) {
        int u = edge[0];
        int v = edge[1];
        graph[u].add(v);
        graph[v].add(u);
    }

    // 计算每个节点的深度和父节点
    int[] depth = new int[n];
    int[] parent = new int[n];
```

```
Arrays.fill(parent, -1);

// BFS 计算深度和父节点
Queue<Integer> queue = new LinkedList<>();
queue.offer(0);
depth[0] = 0;

while (!queue.isEmpty()) {
    int current = queue.poll();

    for (int neighbor : graph[current]) {
        if (neighbor != parent[current]) {
            parent[neighbor] = current;
            depth[neighbor] = depth[current] + 1;
            queue.offer(neighbor);
        }
    }
}

// 标记需要访问的节点
boolean[] needVisit = new boolean[n];

// 从有苹果的叶子节点开始，标记所有需要访问的路径
for (int i = 0; i < n; i++) {
    if (hasApple.get(i)) {
        // 标记当前节点和所有祖先节点
        int currentNode = i;
        while (currentNode != -1 && !needVisit[currentNode]) {
            needVisit[currentNode] = true;
            currentNode = parent[currentNode];
        }
    }
}

// 计算需要访问的边数（每条边需要走 2 次：去和回）
int totalEdges = 0;
for (int i = 1; i < n; i++) { // 从 1 开始，因为节点 0 没有父节点
    if (needVisit[i]) {
        totalEdges++;
    }
}

// 总时间 = 需要访问的边数 * 2
```

```
    return totalEdges * 2;
}

/**
 * 使用树形DP的优化实现
 * @param n 节点数量
 * @param edges 边列表
 * @param hasApple 苹果分布
 * @return 最少时间
 *
 * 时间复杂度：O(n)
 * 空间复杂度：O(n)
 */
public int minTimeDP(int n, int[][] edges, List<Boolean> hasApple) {
    // 构建邻接表
    List<Integer>[] graph = new ArrayList[n];
    for (int i = 0; i < n; i++) {
        graph[i] = new ArrayList<>();
    }

    for (int[] edge : edges) {
        int u = edge[0];
        int v = edge[1];
        graph[u].add(v);
        graph[v].add(u);
    }

    // DP数组：dp[i]表示从节点i出发收集子树中所有苹果所需的时间
    int[] dp = new int[n];
    boolean[] visited = new boolean[n];

    // 后序遍历计算DP值
    Stack<Integer> stack = new Stack<>();
    stack.push(0);

    // 记录每个节点的处理状态：0-未处理，1-正在处理子节点，2-处理完成
    int[] state = new int[n];

    while (!stack.isEmpty()) {
        int node = stack.peek();

        if (state[node] == 0) {
            // 第一次访问，标记为正在处理
```

```
                state[node] = 1;

                // 将所有未访问的子节点入栈
                for (int neighbor : graph[node]) {
                    if (state[neighbor] == 0) {
                        stack.push(neighbor);
                    }
                }
            } else if (state[node] == 1) {
                // 所有子节点处理完成，计算当前节点的 DP 值
                stack.pop();
                state[node] = 2;

                int totalTime = 0;
                boolean hasAppleInSubtree = hasApple.get(node);

                for (int neighbor : graph[node]) {
                    if (state[neighbor] == 2) {
                        // 如果子树需要访问或者子树中有苹果
                        if (dp[neighbor] > 0 || hasApple.get(neighbor)) {
                            totalTime += dp[neighbor] + 2;
                            hasAppleInSubtree = true;
                        }
                    }
                }

                // 如果当前子树有苹果或者需要访问，则更新 DP 值
                if (hasAppleInSubtree) {
                    dp[node] = totalTime;
                } else {
                    dp[node] = 0;
                }
            }
        }

        return dp[0];
    }

    // 测试方法
    public static void main(String[] args) {
        LeetCode1443_MinimumTimeToCollectApples solution = new
LeetCode1443_MinimumTimeToCollectApples();
```

```
        // 测试用例 1: n=7, edges=[[0,1],[0,2],[1,4],[1,5],[2,3],[2,6]],
hasApple=[false,false,true,false,true,true,false]
        // 树结构：
        //       0
        //      / \
        //     1   2
        //    / \  / \
        //   4  5 3  6
        // 苹果在节点 2,4,5
        // 预期输出：8（路径：0->1->4->1->5->1->0->2->0）
        int n1 = 7;
        int[][] edges1 = {{0,1},{0,2},{1,4},{1,5},{2,3},{2,6}};
        List<Boolean> hasApple1 = Arrays.asList(false, false, true, false, true, true, false);

        System.out.println("测试用例 1 结果: " + solution.minTime(n1, edges1, hasApple1)); // 应该
输出 8
        System.out.println("测试用例 1(迭代)结果: " + solution.minTimeIterative(n1, edges1,
hasApple1)); // 应该输出 8
        System.out.println("测试用例 1(DP)结果: " + solution.minTimeDP(n1, edges1, hasApple1)); //
应该输出 8

        // 测试用例 2: n=4, edges=[[0,2],[0,3],[1,2]], hasApple=[false,true,false,false]
        // 树结构：
        //   0
        //  / \
        // 2   3
        // |
        // 1
        // 苹果在节点 1
        // 预期输出：4（路径：0->2->1->2->0）
        int n2 = 4;
        int[][] edges2 = {{0,2},{0,3},{1,2}};
        List<Boolean> hasApple2 = Arrays.asList(false, true, false, false);

        System.out.println("测试用例 2 结果: " + solution.minTime(n2, edges2, hasApple2)); // 应该
输出 4
        System.out.println("测试用例 2(迭代)结果: " + solution.minTimeIterative(n2, edges2,
hasApple2)); // 应该输出 4
        System.out.println("测试用例 2(DP)结果: " + solution.minTimeDP(n2, edges2, hasApple2)); //
应该输出 4

        // 测试用例 3: 没有苹果
        int n3 = 4;
```

```
        int[][] edges3 = {{0,1},{1,2},{2,3}};
        List<Boolean> hasApple3 = Arrays.asList(false, false, false, false);

        System.out.println("测试用例 3 结果: " + solution.minTime(n3, edges3, hasApple3)); // 应该输出 0
        System.out.println("测试用例 3(迭代)结果: " + solution.minTimeIterative(n3, edges3, hasApple3)); // 应该输出 0
        System.out.println("测试用例 3(DP)结果: " + solution.minTimeDP(n3, edges3, hasApple3)); // 应该输出 0

        // 测试用例 4: 所有节点都有苹果
        int n4 = 3;
        int[][] edges4 = {{0,1},{1,2}};
        List<Boolean> hasApple4 = Arrays.asList(true, true, true);

        System.out.println("测试用例 4 结果: " + solution.minTime(n4, edges4, hasApple4)); // 应该输出 4
        System.out.println("测试用例 4(迭代)结果: " + solution.minTimeIterative(n4, edges4, hasApple4)); // 应该输出 4
        System.out.println("测试用例 4(DP)结果: " + solution.minTimeDP(n4, edges4, hasApple4)); // 应该输出 4
    }
}
```

```
==========================================
```

文件: LeetCode1522_DiameterOfNAryTree.cpp

```
==========================================
// LeetCode 1522. N 叉树的直径
// 题目：给定一棵 N 叉树，你需要计算它的直径长度。
// 一棵 N 叉树的直径长度是任意两个结点路径长度中的最大值。
// 这条路径可能穿过也可能不穿过根结点。
// 两结点之间的路径长度是以它们之间边的数目表示。
// 来源：LeetCode
// 链接：https://leetcode.cn/problems/diameter-of-n-ary-tree/

#include <iostream>
#include <vector>
#include <algorithm>
#include <queue>
#include <stack>
#include <unordered_map>
```

```
using namespace std;

// N 叉树节点定义
class Node {
public:
    int val;
    vector<Node*> children;

    Node() {}

    Node(int _val) {
        val = _val;
    }

    Node(int _val, vector<Node*> _children) {
        val = _val;
        children = _children;
    }
};

class LeetCode1522_DiameterOfNAryTree {
public:
    /**
     * 计算 N 叉树的直径
     * @param root N 叉树根节点
     * @return 树的直径（边数）
     *
     * 时间复杂度：O(n)，其中 n 是 N 叉树的节点数，每个节点只访问一次
     * 空间复杂度：O(h)，其中 h 是 N 叉树的高度，递归调用栈的深度
     */
    int diameter(Node* root) {
        maxDiameter = 0;  // 重置全局变量
        depth(root);      // 计算每个节点的深度并更新最大直径
        return maxDiameter;
    }

    /**
     * 使用优先队列优化的深度计算方法
     * @param root N 叉树根节点
     * @return 树的直径
     */
    int diameterOptimized(Node* root) {
        maxDiameter = 0;
```

```
    depthOptimized(root);
    return maxDiameter;
}

/**
 * 迭代实现（避免递归深度过大）
 * @param root N 叉树根节点
 * @return 树的直径
 *
 * 时间复杂度：O(n)
 * 空间复杂度：O(n)
 */
int diameterIterative(Node* root) {
    if (root == nullptr) {
        return 0;
    }

    // 使用后序遍历计算每个节点的深度
    unordered_map<Node*, int> depthMap;
    stack<Node*> stk;
    stk.push(root);

    int maxDiameter = 0;

    while (!stk.empty()) {
        Node* node = stk.top();

        // 检查是否所有子节点都已经处理过
        bool allChildrenProcessed = true;

        for (Node* child : node->children) {
            if (depthMap.find(child) == depthMap.end()) {
                stk.push(child);
                allChildrenProcessed = false;
                break;
            }
        }

        if (allChildrenProcessed) {
            stk.pop();

            // 计算当前节点的深度
            vector<int> childDepths;
```

```
                for (Node* child : node->children) {
                    childDepths.push_back(depthMap[child]);
                }

                // 排序找到最大的两个深度
                sort(childDepths.rbegin(), childDepths.rend());

                int max1 = childDepths.size() >= 1 ? childDepths[0] : 0;
                int max2 = childDepths.size() >= 2 ? childDepths[1] : 0;

                // 更新最大直径
                maxDiameter = max(maxDiameter, max1 + max2);

                // 当前节点的深度 = 最大子节点深度 + 1
                depthMap[node] = max1 + 1;
            }
        }

        return maxDiameter;
    }

private:
    int maxDiameter; // 全局变量，用于记录最大直径

    /**
     * 计算以当前节点为根的子树深度，并更新最大直径
     * @param node 当前节点
     * @return 当前节点为根的子树的最大深度
     */
    int depth(Node* node) {
        // 基本情况：空节点深度为0
        if (node == nullptr) {
            return 0;
        }

        // 特殊情况：没有子节点，深度为0
        if (node->children.empty()) {
            return 0;
        }

        // 存储所有子节点的深度
        vector<int> depths;
```

```
    // 递归计算所有子节点的最大深度
    for (Node* child : node->children) {
        int childDepth = depth(child);
        depths.push_back(childDepth);
    }

    // 对深度进行排序，找到最大的两个深度
    sort(depths.rbegin(), depths.rend());

    // 计算经过当前节点的最长路径
    int pathThroughNode = 0;
    if (depths.size() >= 1) {
        pathThroughNode += depths[0];
    }
    if (depths.size() >= 2) {
        pathThroughNode += depths[1];
    }

    // 更新全局最大直径
    maxDiameter = max(maxDiameter, pathThroughNode);

    // 返回以当前节点为根的子树的最大深度
    return depths.empty() ? 0 : depths[0] + 1;
}

/**
 * 使用优先队列优化的深度计算方法
 * @param node 当前节点
 * @return 当前节点为根的子树的最大深度
 */
int depthOptimized(Node* node) {
    if (node == nullptr) {
        return 0;
    }

    if (node->children.empty()) {
        return 0;
    }

    // 使用优先队列（最大堆）来维护最大的两个深度
    priority_queue<int> maxHeap;

    for (Node* child : node->children) {
```

```
            int childDepth = depthOptimized(child);
            maxHeap.push(childDepth);
        }

        // 取出最大的两个深度
        int max1 = maxHeap.empty() ? 0 : maxHeap.top();
        if (!maxHeap.empty()) maxHeap.pop();
        int max2 = maxHeap.empty() ? 0 : maxHeap.top();

        // 更新最大直径
        maxDiameter = max(maxDiameter, max1 + max2);

        // 返回最大深度
        return max1 + 1;
    }

    // 测试方法
    void test() {
        LeetCode1522_DiameterOfNAryTree solution;

        // 测试用例 1: 简单的三叉树
        //     1
        //   / | \
        //  2  3  4
        // 预期输出：2（路径 2-1-3 或 2-1-4 或 3-1-4）
        Node* root1 = new Node(1);
        root1->children.push_back(new Node(2));
        root1->children.push_back(new Node(3));
        root1->children.push_back(new Node(4));

        cout << "测试用例 1 结果: " << solution.diameter(root1) << endl; // 应该输出 2
        cout << "测试用例 1(优化)结果: " << solution.diameterOptimized(root1) << endl; // 应该输出
2
        cout << "测试用例 1(迭代)结果: " << solution.diameterIterative(root1) << endl; // 应该输出
2

        // 测试用例 2: 更复杂的 N 叉树
        //       1
        //     / | \
        //    2  3  4
        //   /|     |
        //  5 6     7
        // 预期输出：4（路径 5-2-1-4-7）
```

```
        Node* root2 = new Node(1);
        Node* node2 = new Node(2);
        Node* node3 = new Node(3);
        Node* node4 = new Node(4);

        root2->children.push_back(node2);
        root2->children.push_back(node3);
        root2->children.push_back(node4);

        node2->children.push_back(new Node(5));
        node2->children.push_back(new Node(6));
        node4->children.push_back(new Node(7));

        cout << "测试用例 2 结果: " << solution.diameter(root2) << endl; // 应该输出 4
        cout << "测试用例 2(优化)结果: " << solution.diameterOptimized(root2) << endl; // 应该输出
4
        cout << "测试用例 2(迭代)结果: " << solution.diameterIterative(root2) << endl; // 应该输出
4

        // 测试用例 3: 单节点树
        Node* root3 = new Node(1);
        cout << "测试用例 3 结果: " << solution.diameter(root3) << endl; // 应该输出 0
        cout << "测试用例 3(优化)结果: " << solution.diameterOptimized(root3) << endl; // 应该输出
0
        cout << "测试用例 3(迭代)结果: " << solution.diameterIterative(root3) << endl; // 应该输出
0

        // 测试用例 4: 空树
        cout << "测试用例 4 结果: " << solution.diameter(nullptr) << endl; // 应该输出 0
        cout << "测试用例 4(优化)结果: " << solution.diameterOptimized(nullptr) << endl; // 应该输
出 0
        cout << "测试用例 4(迭代)结果: " << solution.diameterIterative(nullptr) << endl; // 应该输
出 0

        // 清理内存
        delete root1;
        delete root2;
        delete root3;
    }
};

int main() {
    LeetCode1522_DiameterOfNAryTree solution;
```

```
    solution.test();
    return 0;
}
```

==========

文件: LeetCode1522_DiameterOfNAryTree.java

==========

```
package class121;

// LeetCode 1522. N 叉树的直径
// 题目：给定一棵 N 叉树，你需要计算它的直径长度。
// 一棵 N 叉树的直径长度是任意两个结点路径长度中的最大值。
// 这条路径可能穿过也可能不穿过根结点。
// 两结点之间的路径长度是以它们之间边的数目表示。
// 来源：LeetCode
// 链接：https://leetcode.cn/problems/diameter-of-n-ary-tree/

import java.util.*;

// N 叉树节点定义
class Node {
    public int val;
    public List<Node> children;

    public Node() {
        children = new ArrayList<Node>();
    }

    public Node(int _val) {
        val = _val;
        children = new ArrayList<Node>();
    }

    public Node(int _val, ArrayList<Node> _children) {
        val = _val;
        children = _children;
    }
}

public class LeetCode1522_DiameterOfNAryTree {

    // 全局变量，用于记录最大直径
```

```
private int maxDiameter = 0;

/**
 * 计算 N 叉树的直径
 * @param root N 叉树根节点
 * @return 树的直径（边数）
 *
 * 时间复杂度：O(n)，其中 n 是 N 叉树的节点数，每个节点只访问一次
 * 空间复杂度：O(h)，其中 h 是 N 叉树的高度，递归调用栈的深度
 */
public int diameter(Node root) {
    maxDiameter = 0;  // 重置全局变量
    depth(root);      // 计算每个节点的深度并更新最大直径
    return maxDiameter;
}

/**
 * 计算以当前节点为根的子树深度，并更新最大直径
 * @param node 当前节点
 * @return 当前节点为根的子树的最大深度
 */
private int depth(Node node) {
    // 基本情况：空节点深度为 0
    if (node == null) {
        return 0;
    }

    // 特殊情况：没有子节点，深度为 0
    if (node.children == null || node.children.isEmpty()) {
        return 0;
    }

    // 存储所有子节点的深度
    List<Integer> depths = new ArrayList<>();

    // 递归计算所有子节点的最大深度
    for (Node child : node.children) {
        int childDepth = depth(child);
        depths.add(childDepth);
    }

    // 对深度进行排序，找到最大的两个深度
    Collections.sort(depths, Collections.reverseOrder());
```

```
        // 计算经过当前节点的最长路径
        int pathThroughNode = 0;
        if (depths.size() >= 1) {
            pathThroughNode += depths.get(0);
        }
        if (depths.size() >= 2) {
            pathThroughNode += depths.get(1);
        }

        // 更新全局最大直径
        maxDiameter = Math.max(maxDiameter, pathThroughNode);

        // 返回以当前节点为根的子树的最大深度
        // 如果当前节点有子节点，最大深度是最大子节点深度 + 1
        // 如果当前节点没有子节点，深度为0
        return depths.isEmpty() ? 0 : depths.get(0) + 1;
    }

    /**
     * 使用优先队列优化的深度计算方法
     * @param node 当前节点
     * @return 当前节点为根的子树的最大深度
     */
    private int depthOptimized(Node node) {
        if (node == null) {
            return 0;
        }

        if (node.children == null || node.children.isEmpty()) {
            return 0;
        }

        // 使用优先队列（最大堆）来维护最大的两个深度
        PriorityQueue<Integer> maxHeap = new PriorityQueue<>(Collections.reverseOrder());

        for (Node child : node.children) {
            int childDepth = depthOptimized(child);
            maxHeap.offer(childDepth);
        }

        // 取出最大的两个深度
        int max1 = maxHeap.isEmpty() ? 0 : maxHeap.poll();
```

```
        int max2 = maxHeap.isEmpty() ? 0 : maxHeap.poll();

        // 更新最大直径
        maxDiameter = Math.max(maxDiameter, max1 + max2);

        // 返回最大深度
        return max1 + 1;
    }

    /**
     * 迭代实现（避免递归深度过大）
     * @param root N 叉树根节点
     * @return 树的直径
     *
     * 时间复杂度：O(n)
     * 空间复杂度：O(n)
     */
    public int diameterIterative(Node root) {
        if (root == null) {
            return 0;
        }

        // 使用后序遍历计算每个节点的深度
        Map<Node, Integer> depthMap = new HashMap<>();
        Stack<Node> stack = new Stack<>();
        stack.push(root);

        int maxDiameter = 0;

        while (!stack.isEmpty()) {
            Node node = stack.peek();

            // 如果所有子节点都已经处理过
            boolean allChildrenProcessed = true;

            for (Node child : node.children) {
                if (!depthMap.containsKey(child)) {
                    stack.push(child);
                    allChildrenProcessed = false;
                    break;
                }
            }
```

```
            if (allChildrenProcessed) {
                stack.pop();

                // 计算当前节点的深度
                List<Integer> childDepths = new ArrayList<>();
                for (Node child : node.children) {
                    childDepths.add(depthMap.get(child));
                }

                // 排序找到最大的两个深度
                Collections.sort(childDepths, Collections.reverseOrder());

                int max1 = childDepths.size() >= 1 ? childDepths.get(0) : 0;
                int max2 = childDepths.size() >= 2 ? childDepths.get(1) : 0;

                // 更新最大直径
                maxDiameter = Math.max(maxDiameter, max1 + max2);

                // 当前节点的深度 = 最大子节点深度 + 1
                depthMap.put(node, max1 + 1);
            }
        }

        return maxDiameter;
    }

    // 测试方法
    public static void main(String[] args) {
        LeetCode1522_DiameterOfNAryTree solution = new LeetCode1522_DiameterOfNAryTree();

        // 测试用例 1: 简单的三叉树
        //     1
        //   / | \
        //  2  3  4
        // 预期输出：2（路径 2-1-3 或 2-1-4 或 3-1-4）
        Node root1 = new Node(1);
        root1.children.add(new Node(2));
        root1.children.add(new Node(3));
        root1.children.add(new Node(4));

        System.out.println("测试用例 1 结果: " + solution.diameter(root1)); // 应该输出 2
        System.out.println("测试用例 1(优化)结果: " + solution.diameterOptimized(root1)); // 应该
输出 2
```

```
        System.out.println("测试用例 1(迭代)结果: " + solution.diameterIterative(root1)); // 应该
输出 2

        // 测试用例 2: 更复杂的 N 叉树
        //       1
        //     / | \
        //    2  3  4
        //   /|     |
        //  5 6     7
        // 预期输出：4（路径 5-2-1-4-7）
        Node root2 = new Node(1);
        Node node2 = new Node(2);
        Node node3 = new Node(3);
        Node node4 = new Node(4);

        root2.children.add(node2);
        root2.children.add(node3);
        root2.children.add(node4);

        node2.children.add(new Node(5));
        node2.children.add(new Node(6));
        node4.children.add(new Node(7));

        System.out.println("测试用例 2 结果: " + solution.diameter(root2)); // 应该输出 4
        System.out.println("测试用例 2(优化)结果: " + solution.diameterOptimized(root2)); // 应该
输出 4
        System.out.println("测试用例 2(迭代)结果: " + solution.diameterIterative(root2)); // 应该
输出 4

        // 测试用例 3: 单节点树
        Node root3 = new Node(1);
        System.out.println("测试用例 3 结果: " + solution.diameter(root3)); // 应该输出 0
        System.out.println("测试用例 3(优化)结果: " + solution.diameterOptimized(root3)); // 应该
输出 0
        System.out.println("测试用例 3(迭代)结果: " + solution.diameterIterative(root3)); // 应该
输出 0

        // 测试用例 4: 空树
        System.out.println("测试用例 4 结果: " + solution.diameter(null)); // 应该输出 0
        System.out.println("测试用例 4(优化)结果: " + solution.diameterOptimized(null)); // 应该输
出 0
        System.out.println("测试用例 4(迭代)结果: " + solution.diameterIterative(null)); // 应该输
出 0
```

```
    }
}
```

==========

文件: LeetCode1522_DiameterOfNAryTree.py

==========

```
# LeetCode 1522. N 叉树的直径
# 题目：给定一棵 N 叉树，你需要计算它的直径长度。
# 一棵 N 叉树的直径长度是任意两个结点路径长度中的最大值。
# 这条路径可能穿过也可能不穿过根结点。
# 两结点之间的路径长度是以它们之间边的数目表示。
# 来源：LeetCode
# 链接：https://leetcode.cn/problems/diameter-of-n-ary-tree/

from typing import List, Optional
from collections import deque
import heapq

# N 叉树节点定义
class Node:
    def __init__(self, val=None, children=None):
        self.val = val
        self.children = children if children is not None else []

class LeetCode1522_DiameterOfNAryTree:
    """
    N 叉树直径计算类
    提供三种方法：递归法、优化递归法、迭代法
    """

    def __init__(self):
        self.max_diameter = 0  # 全局变量，用于记录最大直径

    def diameter(self, root: Optional[Node]) -> int:
        """
        计算 N 叉树的直径

        Args:
            root: N 叉树根节点

        Returns:
            int: 树的直径（边数）
```

```
    时间复杂度：O(n)，其中 n 是 N 叉树的节点数，每个节点只访问一次
    空间复杂度：O(h)，其中 h 是 N 叉树的高度，递归调用栈的深度
    """
    self.max_diameter = 0  # 重置全局变量
    self._depth(root)      # 计算每个节点的深度并更新最大直径
    return self.max_diameter

def _depth(self, node: Optional[Node]) -> int:
    """
    计算以当前节点为根的子树深度，并更新最大直径

    Args:
        node: 当前节点

    Returns:
        int: 当前节点为根的子树的最大深度
    """
    # 基本情况：空节点深度为 0
    if node is None:
        return 0

    # 特殊情况：没有子节点，深度为 0
    if not node.children:
        return 0

    # 存储所有子节点的深度
    depths = []

    # 递归计算所有子节点的最大深度
    for child in node.children:
        child_depth = self._depth(child)
        depths.append(child_depth)

    # 对深度进行排序，找到最大的两个深度
    depths.sort(reverse=True)

    # 计算经过当前节点的最长路径
    path_through_node = 0
    if len(depths) >= 1:
        path_through_node += depths[0]
    if len(depths) >= 2:
        path_through_node += depths[1]
```

```
        # 更新全局最大直径
        self.max_diameter = max(self.max_diameter, path_through_node)

        # 返回以当前节点为根的子树的最大深度
        return depths[0] + 1 if depths else 0

    def diameter_optimized(self, root: Optional[Node]) -> int:
        """
        使用最大堆优化的深度计算方法

        Args:
            root: N 叉树根节点

        Returns:
            int: 树的直径
        """
        self.max_diameter = 0
        self._depth_optimized(root)
        return self.max_diameter

    def _depth_optimized(self, node: Optional[Node]) -> int:
        """
        使用最大堆优化的深度计算

        Args:
            node: 当前节点

        Returns:
            int: 当前节点为根的子树的最大深度
        """
        if node is None:
            return 0

        if not node.children:
            return 0

        # 使用最大堆来维护最大的两个深度
        max_heap = []

        for child in node.children:
            child_depth = self._depth_optimized(child)
            heapq.heappush(max_heap, -child_depth)  # 使用负数模拟最大堆
```

```
    # 取出最大的两个深度
    max1 = -heapq.heappop(max_heap) if max_heap else 0
    max2 = -heapq.heappop(max_heap) if max_heap else 0

    # 更新最大直径
    self.max_diameter = max(self.max_diameter, max1 + max2)

    # 返回最大深度
    return max1 + 1

def diameter_iterative(self, root: Optional[Node]) -> int:
    """
    迭代实现（避免递归深度过大）

    Args:
        root: N 叉树根节点

    Returns:
        int: 树的直径

    时间复杂度：O(n)
    空间复杂度：O(n)
    """
    if root is None:
        return 0

    # 使用后序遍历计算每个节点的深度
    depth_map = {}
    stack = [root]
    visited = set()

    max_diameter = 0

    while stack:
        node = stack[-1]

        # 如果所有子节点都已经处理过
        all_children_processed = True

        for child in node.children:
            if child not in depth_map:
                stack.append(child)
```

```
                all_children_processed = False
                break

        if all_children_processed:
            stack.pop()

            # 计算当前节点的深度
            child_depths = []
            for child in node.children:
                child_depths.append(depth_map[child])

            # 排序找到最大的两个深度
            child_depths.sort(reverse=True)

            max1 = child_depths[0] if child_depths else 0
            max2 = child_depths[1] if len(child_depths) >= 2 else 0

            # 更新最大直径
            max_diameter = max(max_diameter, max1 + max2)

            # 当前节点的深度 = 最大子节点深度 + 1
            depth_map[node] = max1 + 1

    return max_diameter

def test(self):
    """测试方法"""
    solution = LeetCode1522_DiameterOfNAryTree()

    # 测试用例 1: 简单的三叉树
    #     1
    #   / | \
    #  2  3  4
    # 预期输出：2（路径 2-1-3 或 2-1-4 或 3-1-4）
    root1 = Node(1, [Node(2), Node(3), Node(4)])

    print(f"测试用例 1 结果: {solution.diameter(root1)}")  # 应该输出 2
    print(f"测试用例 1(优化)结果: {solution.diameter_optimized(root1)}")  # 应该输出 2
    print(f"测试用例 1(迭代)结果: {solution.diameter_iterative(root1)}")  # 应该输出 2

    # 测试用例 2: 更复杂的 N 叉树
    #       1
    #     / | \
```

```
        #    2  3  4
        #   /|     |
        #  5 6     7
        # 预期输出：4（路径 5-2-1-4-7）
        node2 = Node(2, [Node(5), Node(6)])
        node4 = Node(4, [Node(7)])
        root2 = Node(1, [node2, Node(3), node4])

        print(f"测试用例 2 结果: {solution.diameter(root2)}")  # 应该输出 4
        print(f"测试用例 2(优化)结果: {solution.diameter_optimized(root2)}")  # 应该输出 4
        print(f"测试用例 2(迭代)结果: {solution.diameter_iterative(root2)}")  # 应该输出 4

        # 测试用例 3: 单节点树
        root3 = Node(1)
        print(f"测试用例 3 结果: {solution.diameter(root3)}")  # 应该输出 0
        print(f"测试用例 3(优化)结果: {solution.diameter_optimized(root3)}")  # 应该输出 0
        print(f"测试用例 3(迭代)结果: {solution.diameter_iterative(root3)}")  # 应该输出 0

        # 测试用例 4: 空树
        print(f"测试用例 4 结果: {solution.diameter(None)}")  # 应该输出 0
        print(f"测试用例 4(优化)结果: {solution.diameter_optimized(None)}")  # 应该输出 0
        print(f"测试用例 4(迭代)结果: {solution.diameter_iterative(None)}")  # 应该输出 0

# 主函数
if __name__ == "__main__":
    solution = LeetCode1522_DiameterOfNAryTree()
    solution.test()

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

文件: LeetCode1617_CountSubtreesWithMaxDistance.java
================================================
package class121;

// LeetCode 1617. 统计子树中城市之间最大距离
// 题目：有 n 个城市，编号从 1 到 n。给你一个整数 n 和一个数组 edges，
// 其中 edges[i] = [ui, vi] 表示城市 ui 和 vi 之间有一条双向道路。
// 题目保证所有城市形成一棵树。
// 对于每个 d（1 ≤ d ≤ n-1），请你统计有多少个连通子图（子树）的直径恰好为 d。
// 返回一个长度为 n-1 的数组 answer，其中 answer[d-1] 表示直径为 d 的子树数目。
// 来源：LeetCode
// 链接：https://leetcode.cn/problems/count-subtrees-with-max-distance-between-cities/
```

```java
import java.util.*;

public class LeetCode1617_CountSubtreesWithMaxDistance {

    /**
     * 主方法：统计所有直径的子树数量
     * @param n 城市数量
     * @param edges 道路列表
     * @return 长度为 n-1 的数组，answer[d-1]表示直径为 d 的子树数目
     *
     * 时间复杂度：O(n^3 * 2^n)，其中 n 是城市数量
     * 空间复杂度：O(n^2)
     */
    public int[] countSubgraphsForEachDiameter(int n, int[][] edges) {
        // 构建邻接矩阵
        int[][] dist = new int[n][n];

        // 初始化距离矩阵
        for (int i = 0; i < n; i++) {
            Arrays.fill(dist[i], Integer.MAX_VALUE);
            dist[i][i] = 0;
        }

        // 填充直接相连的边
        for (int[] edge : edges) {
            int u = edge[0] - 1; // 转换为 0-based 索引
            int v = edge[1] - 1;
            dist[u][v] = 1;
            dist[v][u] = 1;
        }

        // Floyd-Warshall 算法计算所有点对最短距离
        for (int k = 0; k < n; k++) {
            for (int i = 0; i < n; i++) {
                for (int j = 0; j < n; j++) {
                    if (dist[i][k] != Integer.MAX_VALUE && dist[k][j] != Integer.MAX_VALUE) {
                        dist[i][j] = Math.min(dist[i][j], dist[i][k] + dist[k][j]);
                    }
                }
            }
        }

        // 结果数组
```

```
    int[] result = new int[n - 1];

    // 枚举所有非空子集（使用位掩码）
    for (int mask = 1; mask < (1 << n); mask++) {
        // 检查子集是否连通
        if (isConnected(mask, dist, n)) {
            // 计算子集的直径
            int diameter = getDiameter(mask, dist, n);
            if (diameter > 0 && diameter <= n - 1) {
                result[diameter - 1]++;
            }
        }
    }

    return result;
}

/**
 * 检查子集是否连通
 * @param mask 位掩码表示的子集
 * @param dist 距离矩阵
 * @param n 总节点数
 * @return 子集是否连通
 */
private boolean isConnected(int mask, int[][] dist, int n) {
    // 找到子集中的第一个节点
    int start = -1;
    for (int i = 0; i < n; i++) {
        if ((mask & (1 << i)) != 0) {
            start = i;
            break;
        }
    }

    if (start == -1) return false; // 空子集

    // BFS 检查连通性
    Queue<Integer> queue = new LinkedList<>();
    boolean[] visited = new boolean[n];
    queue.offer(start);
    visited[start] = true;

    int count = 1; // 已访问节点数
```

```
    while (!queue.isEmpty()) {
        int current = queue.poll();

        for (int neighbor = 0; neighbor < n; neighbor++) {
            // 检查邻居是否在子集中且直接相连
            if ((mask & (1 << neighbor)) != 0 &&
                dist[current][neighbor] == 1 &&
                !visited[neighbor]) {
                visited[neighbor] = true;
                queue.offer(neighbor);
                count++;
            }
        }
    }

    // 检查是否所有子集节点都被访问
    int subsetSize = Integer.bitCount(mask);
    return count == subsetSize;
}

/**
 * 计算子集的直径
 * @param mask 位掩码表示的子集
 * @param dist 距离矩阵
 * @param n 总节点数
 * @return 子集的直径
 */
private int getDiameter(int mask, int[][] dist, int n) {
    int diameter = 0;

    // 遍历子集中的所有点对，找到最大距离
    for (int i = 0; i < n; i++) {
        if ((mask & (1 << i)) != 0) {
            for (int j = i + 1; j < n; j++) {
                if ((mask & (1 << j)) != 0 && dist[i][j] != Integer.MAX_VALUE) {
                    diameter = Math.max(diameter, dist[i][j]);
                }
            }
        }
    }

    return diameter;
```

```
}

/**
 * 优化的方法：使用树形DP思想
 * @param n 城市数量
 * @param edges 道路列表
 * @return 结果数组
 *
 * 时间复杂度：O(n^2 * 2^n)
 * 空间复杂度：O(n * 2^n)
 */
public int[] countSubgraphsForEachDiameterOptimized(int n, int[][] edges) {
    // 构建邻接表
    List<Integer>[] graph = new ArrayList[n];
    for (int i = 0; i < n; i++) {
        graph[i] = new ArrayList<>();
    }

    for (int[] edge : edges) {
        int u = edge[0] - 1;
        int v = edge[1] - 1;
        graph[u].add(v);
        graph[v].add(u);
    }

    // 结果数组
    int[] result = new int[n - 1];

    // 枚举所有非空子集
    for (int mask = 1; mask < (1 << n); mask++) {
        // 找到子集中的节点
        List<Integer> nodes = new ArrayList<>();
        for (int i = 0; i < n; i++) {
            if ((mask & (1 << i)) != 0) {
                nodes.add(i);
            }
        }

        // 如果子集大小小于2，直径至少为1
        if (nodes.size() < 2) continue;

        // 构建子图
        Map<Integer, List<Integer>> subgraph = buildSubgraph(mask, graph, n);
```

```
            // 检查连通性并计算直径
            if (isConnectedSubgraph(subgraph, nodes.get(0), nodes.size())) {
                int diameter = calculateDiameter(subgraph, nodes);
                if (diameter > 0 && diameter <= n - 1) {
                    result[diameter - 1]++;
                }
            }
        }

        return result;
    }

    /**
     * 构建子图的邻接表
     * @param mask 位掩码
     * @param graph 原图邻接表
     * @param n 节点总数
     * @return 子图邻接表
     */
    private Map<Integer, List<Integer>> buildSubgraph(int mask, List<Integer>[] graph, int n) {
        Map<Integer, List<Integer>> subgraph = new HashMap<>();

        for (int i = 0; i < n; i++) {
            if ((mask & (1 << i)) != 0) {
                subgraph.put(i, new ArrayList<>());
                for (int neighbor : graph[i]) {
                    if ((mask & (1 << neighbor)) != 0) {
                        subgraph.get(i).add(neighbor);
                    }
                }
            }
        }

        return subgraph;
    }

    /**
     * 检查子图是否连通
     * @param subgraph 子图邻接表
     * @param start 起始节点
     * @param expectedSize 期望的连通分量大小
     * @return 是否连通
```

```
     */
    private boolean isConnectedSubgraph(Map<Integer, List<Integer>> subgraph, int start, int
expectedSize) {
        if (!subgraph.containsKey(start)) return false;

        Set<Integer> visited = new HashSet<>();
        Queue<Integer> queue = new LinkedList<>();
        queue.offer(start);
        visited.add(start);

        while (!queue.isEmpty()) {
            int current = queue.poll();

            for (int neighbor : subgraph.get(current)) {
                if (!visited.contains(neighbor)) {
                    visited.add(neighbor);
                    queue.offer(neighbor);
                }
            }
        }

        return visited.size() == expectedSize;
    }

    /**
     * 计算子图的直径
     * @param subgraph 子图邻接表
     * @param nodes 节点列表
     * @return 直径
     */
    private int calculateDiameter(Map<Integer, List<Integer>> subgraph, List<Integer> nodes) {
        if (nodes.size() == 1) return 0;

        // 使用两次BFS法计算直径
        int start = nodes.get(0);

        // 第一次BFS找到最远节点
        int[] firstBFS = bfs(subgraph, start);
        int farthest = firstBFS[0];

        // 第二次BFS找到直径
        int[] secondBFS = bfs(subgraph, farthest);
```

```
        return secondBFS[1];
    }

    /**
     * BFS 计算从起点开始的最远节点和距离
     * @param subgraph 子图
     * @param start 起点
     * @return [最远节点, 距离]
     */
    private int[] bfs(Map<Integer, List<Integer>> subgraph, int start) {
        Map<Integer, Integer> distance = new HashMap<>();
        Queue<Integer> queue = new LinkedList<>();

        distance.put(start, 0);
        queue.offer(start);

        int farthestNode = start;
        int maxDistance = 0;

        while (!queue.isEmpty()) {
            int current = queue.poll();
            int currentDist = distance.get(current);

            if (currentDist > maxDistance) {
                maxDistance = currentDist;
                farthestNode = current;
            }

            for (int neighbor : subgraph.get(current)) {
                if (!distance.containsKey(neighbor)) {
                    distance.put(neighbor, currentDist + 1);
                    queue.offer(neighbor);
                }
            }
        }

        return new int[]{farthestNode, maxDistance};
    }

    // 测试方法
    public static void main(String[] args) {
        LeetCode1617_CountSubtreesWithMaxDistance solution = new
LeetCode1617_CountSubtreesWithMaxDistance();
```

```
        // 测试用例 1: n=4, edges=[[1,2],[2,3],[2,4]]
        // 树结构：
        //   1
        //   |
        //   2
        //  / \
        // 3   4
        // 预期输出：[3,4,0]（直径 1:3 个，直径 2:4 个，直径 3:0 个）
        int n1 = 4;
        int[][] edges1 = {{1,2},{2,3},{2,4}};
        int[] result1 = solution.countSubgraphsForEachDiameter(n1, edges1);
        System.out.println("测试用例 1 结果: " + Arrays.toString(result1));

        // 测试用例 2: n=2, edges=[[1,2]]
        // 树结构：1-2
        // 预期输出：[1]（直径 1:1 个）
        int n2 = 2;
        int[][] edges2 = {{1,2}};
        int[] result2 = solution.countSubgraphsForEachDiameter(n2, edges2);
        System.out.println("测试用例 2 结果: " + Arrays.toString(result2));

        // 测试用例 3: n=3, edges=[[1,2],[2,3]]
        // 树结构：1-2-3
        // 预期输出：[2,1]（直径 1:2 个，直径 2:1 个）
        int n3 = 3;
        int[][] edges3 = {{1,2},{2,3}};
        int[] result3 = solution.countSubgraphsForEachDiameter(n3, edges3);
        System.out.println("测试用例 3 结果: " + Arrays.toString(result3));

        // 使用优化方法测试
        System.out.println("=== 使用优化方法 ===");
        int[] result1Opt = solution.countSubgraphsForEachDiameterOptimized(n1, edges1);
        System.out.println("测试用例 1(优化)结果: " + Arrays.toString(result1Opt));

        int[] result2Opt = solution.countSubgraphsForEachDiameterOptimized(n2, edges2);
        System.out.println("测试用例 2(优化)结果: " + Arrays.toString(result2Opt));

        int[] result3Opt = solution.countSubgraphsForEachDiameterOptimized(n3, edges3);
        System.out.println("测试用例 3(优化)结果: " + Arrays.toString(result3Opt));
    }
}
```

```
==========================================

文件: LeetCode543_DiameterOfBinaryTree.cpp
==========================================
// LeetCode 543. 二叉树的直径
// 题目：给定一棵二叉树，你需要计算它的直径长度。
// 一棵二叉树的直径长度是任意两个结点路径长度中的最大值。
// 这条路径可能穿过也可能不穿过根结点。
// 两结点之间的路径长度是以它们之间边的数目表示。

// 算法标签：树、深度优先搜索、递归
// 难度：简单
// 时间复杂度：O(n)，其中 n 是二叉树的节点数，每个节点只访问一次
// 空间复杂度：O(h)，其中 h 是二叉树的高度，递归调用栈的深度

// 相关题目：
// - LeetCode 1522. Diameter of N-Ary Tree (N 叉树的直径)
// - LeetCode 1245. Tree Diameter (无向树的直径)
// - LeetCode 1617. Count Subtrees With Max Distance Between Cities (统计子树中城市之间最大距离)
// - SPOJ PT07Z - Longest path in a tree (树中最长路径)
// - CSES 1131 - Tree Diameter (树的直径)
// - 51Nod 2602 - 树的直径
// - 洛谷 U81904 树的直径
// - AtCoder ABC221F - Diameter Set

// 解题思路：
// 1. 二叉树的直径不一定经过根节点
// 2. 对于每个节点，经过该节点的最长路径等于左子树的最大深度加上右子树的最大深度
// 3. 使用递归方法，在计算每个节点子树深度的同时更新全局最大直径
// 4. 递归函数返回以当前节点为根的子树的最大深度

// 如果编译器不支持标准库头文件，可以使用以下替代方案
#ifndef max
#define max(a,b) ((a) > (b) ? (a) : (b))
#endif

// 二叉树节点定义
struct TreeNode {
    int val;
    TreeNode *left;
    TreeNode *right;
    TreeNode() : val(0), left(nullptr), right(nullptr) {}
    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
```

```
    TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}
};

class Solution {
private:
    int maxDiameter;  // 全局变量，用于记录最大直径

    /**
     * 计算以当前节点为根的子树深度，并更新最大直径
     *
     * 算法思路：
     * 1. 递归计算左子树和右子树的最大深度
     * 2. 经过当前节点的最长路径 = 左子树最大深度 + 右子树最大深度
     * 3. 更新全局最大直径
     * 4. 返回以当前节点为根的子树的最大深度
     *
     * @param node 当前节点
     * @return 当前节点为根的子树的最大深度
     */
    int depth(TreeNode* node) {
        // 基本情况：空节点深度为0
        if (node == nullptr) {
            return 0;
        }

        // 递归计算左子树和右子树的最大深度
        int leftDepth = depth(node->left);
        int rightDepth = depth(node->right);

        // 经过当前节点的最长路径 = 左子树最大深度 + 右子树最大深度
        // 更新全局最大直径
        maxDiameter = max(maxDiameter, leftDepth + rightDepth);

        // 返回以当前节点为根的子树的最大深度
        return max(leftDepth, rightDepth) + 1;
    }

public:
    /**
     * 计算二叉树的直径
     *
     * 算法思路：
     * 1. 对于每个节点，计算其左子树和右子树的最大深度
```

```
     * 2. 经过该节点的最长路径就是左子树最大深度+右子树最大深度
     * 3. 遍历所有节点，取最大值
     *
     * @param root 二叉树根节点
     * @return 树的直径（边数）
     *
     * 时间复杂度：O(n)，其中 n 是二叉树的节点数，每个节点只访问一次
     * 空间复杂度：O(h)，其中 h 是二叉树的高度，递归调用栈的深度
     */
    int diameterOfBinaryTree(TreeNode* root) {
        maxDiameter = 0;  // 重置全局变量
        depth(root);      // 计算每个节点的深度并更新最大直径
        return maxDiameter;
    }
};

// 测试方法
int main() {
    Solution solution;

    // 测试用例 1: [1,2,3,4,5]
    //     1
    //    / \
    //   2   3
    //  / \
    // 4   5
    // 预期输出: 3 (路径 [4,2,1,3] 或 [5,2,1,3])
    TreeNode* root1 = new TreeNode(1);
    root1->left = new TreeNode(2);
    root1->right = new TreeNode(3);
    root1->left->left = new TreeNode(4);
    root1->left->right = new TreeNode(5);

    // 测试用例 1 结果: solution.diameterOfBinaryTree(root1) 应该输出 3

    // 测试用例 2: [1,2]
    //   1
    //  /
    // 2
    // 预期输出: 1 (路径 [1,2])
    TreeNode* root2 = new TreeNode(1);
    root2->left = new TreeNode(2);
```

```
    // 测试用例 2 结果: solution.diameterOfBinaryTree(root2) 应该输出 1

    return 0;
}

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

文件: LeetCode543_DiameterOfBinaryTree.java
==========================================
package class121;

// LeetCode 543. 二叉树的直径
// 题目：给定一棵二叉树，你需要计算它的直径长度。
// 一棵二叉树的直径长度是任意两个结点路径长度中的最大值。
// 这条路径可能穿过也可能不穿过根结点。
// 两结点之间的路径长度是以它们之间边的数目表示。

import java.util.*;

// 二叉树节点定义
class TreeNode {
    int val;
    TreeNode left;
    TreeNode right;
    TreeNode() {}
    TreeNode(int val) { this.val = val; }
    TreeNode(int val, TreeNode left, TreeNode right) {
        this.val = val;
        this.left = left;
        this.right = right;
    }
}

public class LeetCode543_DiameterOfBinaryTree {

    // 全局变量，用于记录最大直径
    private int maxDiameter = 0;

    /**
     * 计算二叉树的直径
     * @param root 二叉树根节点
     * @return 树的直径（边数）
     *
```

```
     * 时间复杂度：O(n)，其中 n 是二叉树的节点数，每个节点只访问一次
     * 空间复杂度：O(h)，其中 h 是二叉树的高度，递归调用栈的深度
     */
    public int diameterOfBinaryTree(TreeNode root) {
        maxDiameter = 0;  // 重置全局变量
        depth(root);      // 计算每个节点的深度并更新最大直径
        return maxDiameter;
    }

    /**
     * 计算以当前节点为根的子树深度，并更新最大直径
     * @param node 当前节点
     * @return 当前节点为根的子树的最大深度
     */
    private int depth(TreeNode node) {
        // 基本情况：空节点深度为 0
        if (node == null) {
            return 0;
        }

        // 递归计算左子树和右子树的最大深度
        int leftDepth = depth(node.left);
        int rightDepth = depth(node.right);

        // 经过当前节点的最长路径 = 左子树最大深度 + 右子树最大深度
        // 更新全局最大直径
        maxDiameter = Math.max(maxDiameter, leftDepth + rightDepth);

        // 返回以当前节点为根的子树的最大深度
        return Math.max(leftDepth, rightDepth) + 1;
    }

    // 测试方法
    public static void main(String[] args) {
        LeetCode543_DiameterOfBinaryTree solution = new LeetCode543_DiameterOfBinaryTree();

        // 测试用例 1: [1,2,3,4,5]
        //     1
        //    / \
        //   2   3
        //  / \
        // 4   5
        // 预期输出: 3 (路径 [4,2,1,3] 或 [5,2,1,3])
```

```
        TreeNode root1 = new TreeNode(1);
        root1.left = new TreeNode(2);
        root1.right = new TreeNode(3);
        root1.left.left = new TreeNode(4);
        root1.left.right = new TreeNode(5);

        System.out.println("测试用例 1 结果: " + solution.diameterOfBinaryTree(root1)); // 应该输出 3

        // 测试用例 2: [1,2]
        //   1
        //  /
        // 2
        // 预期输出: 1 (路径 [1,2])
        TreeNode root2 = new TreeNode(1);
        root2.left = new TreeNode(2);

        System.out.println("测试用例 2 结果: " + solution.diameterOfBinaryTree(root2)); // 应该输出 1
    }
}
```

==========

文件: LeetCode543_DiameterOfBinaryTree.py

==========

```
# LeetCode 543. 二叉树的直径
# 题目：给定一棵二叉树，你需要计算它的直径长度。
# 一棵二叉树的直径长度是任意两个结点路径长度中的最大值。
# 这条路径可能穿过也可能不穿过根结点。
# 两结点之间的路径长度是以它们之间边的数目表示。

# 算法标签：树、深度优先搜索、递归
# 难度：简单
# 时间复杂度：O(n)，其中 n 是二叉树的节点数，每个节点只访问一次
# 空间复杂度：O(h)，其中 h 是二叉树的高度，递归调用栈的深度

# 相关题目：
# - LeetCode 1522. Diameter of N-Ary Tree (N 叉树的直径)
# - LeetCode 1245. Tree Diameter (无向树的直径)
# - LeetCode 1617. Count Subtrees With Max Distance Between Cities (统计子树中城市之间最大距离)
# - SPOJ PT07Z - Longest path in a tree (树中最长路径)
# - CSES 1131 - Tree Diameter (树的直径)
```

```
# - 51Nod 2602 - 树的直径
# - 洛谷 U81904 树的直径
# - AtCoder ABC221F - Diameter Set

# 解题思路：
# 1. 二叉树的直径不一定经过根节点
# 2. 对于每个节点，经过该节点的最长路径等于左子树的最大深度加上右子树的最大深度
# 3. 使用递归方法，在计算每个节点子树深度的同时更新全局最大直径
# 4. 递归函数返回以当前节点为根的子树的最大深度

# 二叉树节点定义
class TreeNode:
    """
    二叉树节点类

    属性:
        val (int): 节点值
        left (TreeNode): 左子节点
        right (TreeNode): 右子节点
    """
    def __init__(self, val=0, left=None, right=None):
        """
        初始化二叉树节点

        参数:
            val (int): 节点值，默认为 0
            left (TreeNode): 左子节点，默认为 None
            right (TreeNode): 右子节点，默认为 None
        """
        self.val = val
        self.left = left
        self.right = right

class Solution:
    """
    二叉树直径求解器

    方法:
        diameterOfBinaryTree: 计算二叉树的直径
        depth: 计算子树深度并更新最大直径
    """

    def __init__(self):
```

```
    """
    初始化 Solution 类
    """
    # 全局变量，用于记录最大直径
    self.max_diameter = 0

def diameterOfBinaryTree(self, root: TreeNode) -> int:
    """
    计算二叉树的直径

    算法思路：
    1. 对于每个节点，计算其左子树和右子树的最大深度
    2. 经过该节点的最长路径就是左子树最大深度+右子树最大深度
    3. 遍历所有节点，取最大值

    参数:
        root (TreeNode): 二叉树根节点

    返回:
        int: 树的直径（边数）

    时间复杂度：O(n)，其中 n 是二叉树的节点数，每个节点只访问一次
    空间复杂度：O(h)，其中 h 是二叉树的高度，递归调用栈的深度
    """
    self.max_diameter = 0  # 重置全局变量
    self.depth(root)       # 计算每个节点的深度并更新最大直径
    return self.max_diameter

def depth(self, node: TreeNode | None) -> int:
    """
    计算以当前节点为根的子树深度，并更新最大直径

    算法思路：
    1. 递归计算左子树和右子树的最大深度
    2. 经过当前节点的最长路径 = 左子树最大深度 + 右子树最大深度
    3. 更新全局最大直径
    4. 返回以当前节点为根的子树的最大深度

    参数:
        node (TreeNode): 当前节点

    返回:
        int: 当前节点为根的子树的最大深度
```

```
        """
        # 基本情况：空节点深度为0
        if not node:
            return 0

        # 递归计算左子树和右子树的最大深度
        left_depth = self.depth(node.left)
        right_depth = self.depth(node.right)

        # 经过当前节点的最长路径 = 左子树最大深度 + 右子树最大深度
        # 更新全局最大直径
        self.max_diameter = max(self.max_diameter, left_depth + right_depth)

        # 返回以当前节点为根的子树的最大深度
        return max(left_depth, right_depth) + 1

# 测试方法
if __name__ == "__main__":
    solution = Solution()

    # 测试用例1: [1,2,3,4,5]
    #     1
    #    / \
    #   2   3
    #  / \
    # 4   5
    # 预期输出: 3 (路径 [4,2,1,3] 或 [5,2,1,3])
    root1 = TreeNode(1)
    root1.left = TreeNode(2)
    root1.right = TreeNode(3)
    root1.left.left = TreeNode(4)
    root1.left.right = TreeNode(5)

    print("测试用例1结果:", solution.diameterOfBinaryTree(root1))  # 应该输出3

    # 测试用例2: [1,2]
    #   1
    #  /
    # 2
    # 预期输出: 1 (路径 [1,2])
    root2 = TreeNode(1)
    root2.left = TreeNode(2)
```

```
    print("测试用例 2 结果:", solution.diameterOfBinaryTree(root2))  # 应该输出 1

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

文件: LuoguU81904_TreeDiameter.cpp
==========================================
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

/**
 * 洛谷 U81904 - 树的直径
 * 题目描述：给定一棵树，树中每条边都有一个权值，树中两点之间的距离定义为连接两点的路径边权之和。
 * 树中最远的两个节点之间的距离被称为树的直径，连接这两点的路径被称为树的最长链。
 * 现在让你求出树的最长链的距离。
 *
 * 输入格式：
 * - 第一行为一个正整数 n，表示这颗树有 n 个节点
 * - 接下来的 n-1 行，每行三个正整数 u,v,w，表示 u,v（u,v<=n）有一条权值为 w 的边相连
 *
 * 输出格式：输入仅一行，表示树的最长链的距离
 *
 * 解题思路：使用树形 DP 法求解，因为边权可能为负
 * 对于每个节点，维护两个值：
 * 1. 该节点到其子树中的最长路径长度
 * 2. 该节点到其子树中的次长路径长度
 * 那么，经过该节点的最长路径就是这两个值的和。遍历所有节点，取最大值即为树的直径
 *
 * 时间复杂度：O(n)，空间复杂度：O(n)
 */

struct Edge {
    int to;     // 目标节点
    int weight; // 边权
    Edge(int t, int w) : to(t), weight(w) {}
};

int n;                  // 节点数量
vector<vector<Edge>> graph; // 邻接表存储树结构
int maxDiameter;        // 记录树的直径

/**
```

```
 * 树形 DP 函数，计算每个节点的最长路径和次长路径，并更新全局最大直径
 * @param current 当前节点
 * @param parent 当前节点的父节点，避免回到父节点
 * @return 当前节点到其子树中的最长路径长度
 */
int treeDP(int current, int parent) {
    int max1 = 0; // 最长路径
    int max2 = 0; // 次长路径

    for (const Edge& edge : graph[current]) {
        int next = edge.to;
        int weight = edge.weight;

        // 避免回到父节点
        if (next == parent) {
            continue;
        }

        // 递归计算子节点的最长路径
        int depth = treeDP(next, current) + weight;

        // 更新最长和次长路径
        if (depth > max1) {
            max2 = max1;
            max1 = depth;
        } else if (depth > max2) {
            max2 = depth;
        }
    }

    // 更新全局最大直径
    maxDiameter = max(maxDiameter, max1 + max2);

    // 返回当前节点的最长路径
    return max1;
}

int main() {
    cin >> n;

    // 初始化邻接表
    graph.resize(n + 1);
```

```
    // 读取边
    for (int i = 0; i < n - 1; i++) {
        int u, v, w;
        cin >> u >> v >> w;
        // 无向树，添加双向边
        graph[u].emplace_back(v, w);
        graph[v].emplace_back(u, w);
    }

    maxDiameter = 0;
    // 从任意节点开始树形 DP，这里选择 1 号节点
    treeDP(1, -1);

    // 输出树的直径
    cout << maxDiameter << endl;

    return 0;
}
```

==========

文件: LuoguU81904_TreeDiameter.java

==========

```
import java.util.*;

/**
 * 洛谷 U81904 - 树的直径
 * 题目描述：给定一棵树，树中每条边都有一个权值，树中两点之间的距离定义为连接两点的路径边权之和。
 * 树中最远的两个节点之间的距离被称为树的直径，连接这两点的路径被称为树的最长链。
 * 现在让你求出树的最长链的距离。
 *
 * 输入格式：
 * - 第一行为一个正整数 n，表示这颗树有 n 个节点
 * - 接下来的 n-1 行，每行三个正整数 u,v,w，表示 u,v（u,v<=n）有一条权值为 w 的边相连
 *
 * 输出格式：输入仅一行，表示树的最长链的距离
 *
 * 解题思路：使用树形 DP 法求解，因为边权可能为负
 * 对于每个节点，维护两个值：
 * 1. 该节点到其子树中的最长路径长度
 * 2. 该节点到其子树中的次长路径长度
 * 那么，经过该节点的最长路径就是这两个值的和。遍历所有节点，取最大值即为树的直径
 *
```

```
 * 时间复杂度：O(n)，空间复杂度：O(n)
 */
public class LuoguU81904_TreeDiameter {
    private static int n; // 节点数量
    private static List<List<Edge>> graph; // 邻接表存储树结构
    private static int maxDiameter; // 记录树的直径

    // 边类，存储连接的节点和边权
    static class Edge {
        int to; // 目标节点
        int weight; // 边权

        Edge(int to, int weight) {
            this.to = to;
            this.weight = weight;
        }
    }

    /**
     * 树形DP函数，计算每个节点的最长路径和次长路径，并更新全局最大直径
     * @param current 当前节点
     * @param parent 当前节点的父节点，避免回到父节点
     * @return 当前节点到其子树中的最长路径长度
     */
    private static int treeDP(int current, int parent) {
        int max1 = 0; // 最长路径
        int max2 = 0; // 次长路径

        for (Edge edge : graph.get(current)) {
            int next = edge.to;
            int weight = edge.weight;

            // 避免回到父节点
            if (next == parent) {
                continue;
            }

            // 递归计算子节点的最长路径
            int depth = treeDP(next, current) + weight;

            // 更新最长和次长路径
            if (depth > max1) {
                max2 = max1;
```

```
                max1 = depth;
            } else if (depth > max2) {
                max2 = depth;
            }
        }

        // 更新全局最大直径
        maxDiameter = Math.max(maxDiameter, max1 + max2);

        // 返回当前节点的最长路径
        return max1;
    }

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        n = scanner.nextInt();

        // 初始化邻接表
        graph = new ArrayList<>();
        for (int i = 0; i <= n; i++) {
            graph.add(new ArrayList<>());
        }

        // 读取边
        for (int i = 0; i < n - 1; i++) {
            int u = scanner.nextInt();
            int v = scanner.nextInt();
            int w = scanner.nextInt();
            // 无向树，添加双向边
            graph.get(u).add(new Edge(v, w));
            graph.get(v).add(new Edge(u, w));
        }
        scanner.close();

        maxDiameter = 0;
        // 从任意节点开始树形 DP，这里选择 1 号节点
        treeDP(1, -1);

        // 输出树的直径
        System.out.println(maxDiameter);
    }
}
```

==========

文件: LuoguU81904_TreeDiameter.py
==========

```
import sys
from sys import stdin

"""
洛谷 U81904 - 树的直径
题目描述：给定一棵树，树中每条边都有一个权值，树中两点之间的距离定义为连接两点的路径边权之和。
树中最远的两个节点之间的距离被称为树的直径，连接这两点的路径被称为树的最长链。
现在让你求出树的最长链的距离。

输入格式：
- 第一行为一个正整数 n，表示这颗树有 n 个节点
- 接下来的 n-1 行，每行三个正整数 u,v,w，表示 u,v（u,v<=n）有一条权值为 w 的边相连

输出格式：输入仅一行，表示树的最长链的距离

解题思路：使用树形 DP 法求解，因为边权可能为负
对于每个节点，维护两个值：
1. 该节点到其子树中的最长路径长度
2. 该节点到其子树中的次长路径长度
那么，经过该节点的最长路径就是这两个值的和。遍历所有节点，取最大值即为树的直径

时间复杂度：O(n)，空间复杂度：O(n)
"""

def main():
    sys.setrecursionlimit(1 << 25)  # 增加递归深度限制

    n = int(stdin.readline())

    # 初始化邻接表
    graph = [[] for _ in range(n + 1)]  # 节点编号从 1 开始

    # 读取边
    for _ in range(n - 1):
        u, v, w = map(int, stdin.readline().split())
        # 无向树，添加双向边
        graph[u].append((v, w))
        graph[v].append((u, w))
```

```
max_diameter = [0]  # 使用列表包装，以便在递归中修改

def tree_dp(current, parent):
    """
    树形 DP 函数，计算每个节点的最长路径和次长路径，并更新全局最大直径

    Args:
        current: 当前节点
        parent: 当前节点的父节点，避免回到父节点

    Returns:
        int: 当前节点到其子树中的最长路径长度
    """
    max1 = 0  # 最长路径
    max2 = 0  # 次长路径

    for next_node, weight in graph[current]:
        # 避免回到父节点
        if next_node == parent:
            continue

        # 递归计算子节点的最长路径
        depth = tree_dp(next_node, current) + weight

        # 更新最长和次长路径
        if depth > max1:
            max2 = max1
            max1 = depth
        elif depth > max2:
            max2 = depth

    # 更新全局最大直径
    max_diameter[0] = max(max_diameter[0], max1 + max2)

    # 返回当前节点的最长路径
    return max1

# 从任意节点开始树形 DP，这里选择 1 号节点
tree_dp(1, -1)

# 输出树的直径
print(max_diameter[0])
```

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

==========

文件: Nod1803_ForestDiameter.cpp

==========

```
#include <iostream>
#include <vector>
#include <queue>
#include <map>
#include <algorithm>
using namespace std;

/**
 * 51Nod-1803 - 森林的直径
 * 题目描述：有一个由 n 个节点组成的森林，每个节点属于一棵树。
 * 支持两种操作：
 * 1. 连接两棵树：输入格式为"1 u v"，表示将节点 u 所在的树与节点 v 所在的树合并
 * 2. 查询操作：输入格式为"2 u"，表示询问节点 u 所在的树的直径
 *
 * 解题思路：
 * 1. 使用并查集来管理各个树的合并
 * 2. 对于每个树，维护其直径的两个端点
 * 3. 当合并两棵树时，新树的直径只可能是原两棵树的直径之一，或者通过连接边形成的新路径（即 u 树的两个端点和 v 树的两个端点之间的最长路径）
 *
 * 时间复杂度分析：
 * - 并查集操作的均摊时间复杂度接近 O(1)
 * - 合并操作需要计算 4 种可能的路径长度，需要额外的 BFS/DFS 操作，单次时间复杂度为 O(n)，但实际应用中树的大小通常不大
 *
 * 空间复杂度：O(n)
 */

vector<int> parent;         // 并查集父节点数组
vector<pair<int, int>> treeDiameter; // 存储每个树的直径的两个端点
vector<vector<int>> graph; // 邻接表存储森林结构
int n, m;                   // n 是节点数，m 是操作数

/**
 * 并查集的查找操作，带路径压缩
 * @param x 要查找的节点
```

```
 * @return 节点 x 所在树的根节点
 */
int find(int x) {
    if (parent[x] != x) {
        parent[x] = find(parent[x]);
    }
    return parent[x];
}

/**
 * BFS 函数，从指定节点开始，找到距离最远的节点和最远距离
 * @param start 起始节点
 * @param visited 标记数组，用于在合并过程中避免越界
 * @return 包含最远节点和最远距离的 pair
 */
pair<int, int> bfs(int start, vector<bool>& visited) {
    queue<int> q;
    map<int, int> distance;

    q.push(start);
    distance[start] = 0;
    visited[start] = true;

    int maxDistance = 0;
    int farthestNode = start;

    while (!q.empty()) {
        int current = q.front();
        q.pop();

        for (int neighbor : graph[current]) {
            if (!visited[neighbor]) {
                visited[neighbor] = true;
                int newDistance = distance[current] + 1;
                distance[neighbor] = newDistance;
                q.push(neighbor);

                if (newDistance > maxDistance) {
                    maxDistance = newDistance;
                    farthestNode = neighbor;
                }
            }
        }
```

```
    }

    return {farthestNode, maxDistance};
}

/**
 * 计算两个节点之间的距离
 * @param u 起始节点
 * @param v 目标节点
 * @return u 和 v 之间的距离
 */
int getDistance(int u, int v) {
    vector<bool> visited(n + 1, false);
    queue<int> q;
    map<int, int> distance;

    q.push(u);
    distance[u] = 0;
    visited[u] = true;

    while (!q.empty()) {
        int current = q.front();
        q.pop();

        if (current == v) {
            return distance[current];
        }

        for (int neighbor : graph[current]) {
            if (!visited[neighbor]) {
                visited[neighbor] = true;
                distance[neighbor] = distance[current] + 1;
                q.push(neighbor);
            }
        }
    }

    return -1; // 应该不会到达这里，因为 u 和 v 在同一棵树中
}

/**
 * 计算一棵树的直径
 * @param root 树的根节点
```

```
 * @return 树的直径的两个端点
 */
pair<int, int> computeDiameter(int root) {
    vector<bool> visited(n + 1, false);
    // 第一次 BFS 找到距离 root 最远的节点 u
    auto result1 = bfs(root, visited);
    int u = result1.first;

    // 重置 visited 数组
    fill(visited.begin(), visited.end(), false);
    // 第二次 BFS 找到距离 u 最远的节点 v
    auto result2 = bfs(u, visited);
    int v = result2.first;

    return {u, v};
}

int main() {
    cin >> n >> m;

    // 初始化并查集
    parent.resize(n + 1);
    for (int i = 1; i <= n; i++) {
        parent[i] = i;
    }

    // 初始化邻接表
    graph.resize(n + 1);

    // 初始化每个树的直径（初始时每个节点自身就是一棵树）
    treeDiameter.resize(n + 1);
    for (int i = 1; i <= n; i++) {
        treeDiameter[i] = {i, i};
    }

    // 处理每个操作
    for (int i = 0; i < m; i++) {
        int op;
        cin >> op;
        if (op == 1) {
            // 连接操作
            int u, v;
            cin >> u >> v;
```

```
// 添加边
graph[u].push_back(v);
graph[v].push_back(u);

// 合并两个集合
int rootU = find(u);
int rootV = find(v);
if (rootU != rootV) {
    parent[rootV] = rootU;

    // 计算新树的直径
    int a1 = treeDiameter[rootU].first;
    int a2 = treeDiameter[rootU].second;
    int b1 = treeDiameter[rootV].first;
    int b2 = treeDiameter[rootV].second;

    // 可能的四种路径
    int d1 = getDistance(a1, a2); // 原 u 树的直径
    int d2 = getDistance(b1, b2); // 原 v 树的直径
    int d3 = getDistance(a1, b1); // a1 到 b1
    int d4 = getDistance(a1, b2); // a1 到 b2
    int d5 = getDistance(a2, b1); // a2 到 b1
    int d6 = getDistance(a2, b2); // a2 到 b2

    // 找出最长的路径
    int maxDistance = d1;
    pair<int, int> newDiameter = {a1, a2};

    if (d2 > maxDistance) {
        maxDistance = d2;
        newDiameter = {b1, b2};
    }
    if (d3 > maxDistance) {
        maxDistance = d3;
        newDiameter = {a1, b1};
    }
    if (d4 > maxDistance) {
        maxDistance = d4;
        newDiameter = {a1, b2};
    }
    if (d5 > maxDistance) {
        maxDistance = d5;
```

```
                    newDiameter = {a2, b1};
                }
                if (d6 > maxDistance) {
                    maxDistance = d6;
                    newDiameter = {a2, b2};
                }

                // 更新新树的直径
                treeDiameter[rootU] = newDiameter;
            }
        } else if (op == 2) {
            // 查询操作
            int u;
            cin >> u;
            int root = find(u);
            int a = treeDiameter[root].first;
            int b = treeDiameter[root].second;
            // 计算直径长度并输出
            cout << getDistance(a, b) << endl;
        }
    }

    return 0;
}
```

==========

文件: Nod1803_ForestDiameter.java

==========

```
import java.util.*;

/**
 * 51Nod-1803 - 森林的直径
 * 题目描述：有一个由 n 个节点组成的森林，每个节点属于一棵树。
 * 支持两种操作：
 * 1. 连接两棵树：输入格式为"1 u v"，表示将节点 u 所在的树与节点 v 所在的树合并
 * 2. 查询操作：输入格式为"2 u"，表示询问节点 u 所在的树的直径
 *
 * 解题思路：
 * 1. 使用并查集来管理各个树的合并
 * 2. 对于每个树，维护其直径的两个端点
 * 3. 当合并两棵树时，新树的直径只可能是原两棵树的直径之一，或者通过连接边形成的新路径（即 u 树的两个端点和 v 树的两个端点之间的最长路径）
```

```
 *
 * 时间复杂度分析：
 * - 并查集操作的均摊时间复杂度接近 O(1)
 * - 合并操作需要计算 4 种可能的路径长度，需要额外的 BFS/DFS 操作，单次时间复杂度为 O(n)，但实际应用中树的大小通常不大
 *
 * 空间复杂度：O(n)
 */
public class Nod1803_ForestDiameter {
    private static int n, m; // n 是节点数，m 是操作数
    private static int[] parent; // 并查集父节点数组
    private static int[][] treeDiameter; // 存储每个树的直径的两个端点，treeDiameter[root][0]和 treeDiameter[root][1]
    private static List<List<Integer>> graph; // 邻接表存储森林结构

    /**
     * 并查集的查找操作，带路径压缩
     * @param x 要查找的节点
     * @return 节点 x 所在树的根节点
     */
    private static int find(int x) {
        if (parent[x] != x) {
            parent[x] = find(parent[x]);
        }
        return parent[x];
    }

    /**
     * BFS 函数，从指定节点开始，找到距离最远的节点和最远距离
     * @param start 起始节点
     * @param visited 标记数组，用于在合并过程中避免越界
     * @return 包含最远节点和最远距离的数组，格式为[最远节点, 最远距离]
     */
    private static int[] bfs(int start, boolean[] visited) {
        Queue<Integer> queue = new LinkedList<>();
        Map<Integer, Integer> distance = new HashMap<>();

        queue.offer(start);
        distance.put(start, 0);
        visited[start] = true;

        int maxDistance = 0;
        int farthestNode = start;
```

```
    while (!queue.isEmpty()) {
        int current = queue.poll();

        for (int neighbor : graph.get(current)) {
            if (!visited[neighbor]) {
                visited[neighbor] = true;
                int newDistance = distance.get(current) + 1;
                distance.put(neighbor, newDistance);
                queue.offer(neighbor);

                if (newDistance > maxDistance) {
                    maxDistance = newDistance;
                    farthestNode = neighbor;
                }
            }
        }
    }

    return new int[]{farthestNode, maxDistance};
}

/**
 * 计算两个节点之间的距离
 * @param u 起始节点
 * @param v 目标节点
 * @return u 和 v 之间的距离
 */
private static int getDistance(int u, int v) {
    boolean[] visited = new boolean[n + 1];
    Queue<Integer> queue = new LinkedList<>();
    Map<Integer, Integer> distance = new HashMap<>();

    queue.offer(u);
    distance.put(u, 0);
    visited[u] = true;

    while (!queue.isEmpty()) {
        int current = queue.poll();

        if (current == v) {
            return distance.get(current);
        }
```

```
        for (int neighbor : graph.get(current)) {
            if (!visited[neighbor]) {
                visited[neighbor] = true;
                distance.put(neighbor, distance.get(current) + 1);
                queue.offer(neighbor);
            }
        }
    }

    return -1; // 应该不会到达这里，因为u和v在同一棵树中
}

/**
 * 计算一棵树的直径
 * @param root 树的根节点
 * @return 树的直径的两个端点，格式为[端点1, 端点2]
 */
private static int[] computeDiameter(int root) {
    boolean[] visited = new boolean[n + 1];
    // 第一次BFS找到距离root最远的节点u
    int[] result1 = bfs(root, visited);
    int u = result1[0];

    // 重置visited数组
    Arrays.fill(visited, false);
    // 第二次BFS找到距离u最远的节点v
    int[] result2 = bfs(u, visited);
    int v = result2[0];

    return new int[]{u, v};
}

public static void main(String[] args) {
    Scanner scanner = new Scanner(System.in);
    n = scanner.nextInt();
    m = scanner.nextInt();

    // 初始化并查集
    parent = new int[n + 1];
    for (int i = 1; i <= n; i++) {
        parent[i] = i;
    }
```

```
// 初始化邻接表
graph = new ArrayList<>();
for (int i = 0; i <= n; i++) {
    graph.add(new ArrayList<>());
}

// 初始化每个树的直径（初始时每个节点自身就是一棵树）
treeDiameter = new int[n + 1][2];
for (int i = 1; i <= n; i++) {
    treeDiameter[i][0] = i;
    treeDiameter[i][1] = i;
}

// 处理每个操作
for (int i = 0; i < m; i++) {
    int op = scanner.nextInt();
    if (op == 1) {
        // 连接操作
        int u = scanner.nextInt();
        int v = scanner.nextInt();

        // 添加边
        graph.get(u).add(v);
        graph.get(v).add(u);

        // 合并两个集合
        int rootU = find(u);
        int rootV = find(v);
        if (rootU != rootV) {
            parent[rootV] = rootU;

            // 计算新树的直径
            int a1 = treeDiameter[rootU][0];
            int a2 = treeDiameter[rootU][1];
            int b1 = treeDiameter[rootV][0];
            int b2 = treeDiameter[rootV][1];

            // 可能的四种路径
            int d1 = getDistance(a1, a2); // 原 u 树的直径
            int d2 = getDistance(b1, b2); // 原 v 树的直径
            int d3 = getDistance(a1, b1); // a1 到 b1
            int d4 = getDistance(a1, b2); // a1 到 b2
```

```
                int d5 = getDistance(a2, b1); // a2到b1
                int d6 = getDistance(a2, b2); // a2到b2

                // 找出最长的路径
                int maxDistance = d1;
                int[] newDiameter = {a1, a2};

                if (d2 > maxDistance) {
                    maxDistance = d2;
                    newDiameter = new int[]{b1, b2};
                }
                if (d3 > maxDistance) {
                    maxDistance = d3;
                    newDiameter = new int[]{a1, b1};
                }
                if (d4 > maxDistance) {
                    maxDistance = d4;
                    newDiameter = new int[]{a1, b2};
                }
                if (d5 > maxDistance) {
                    maxDistance = d5;
                    newDiameter = new int[]{a2, b1};
                }
                if (d6 > maxDistance) {
                    maxDistance = d6;
                    newDiameter = new int[]{a2, b2};
                }

                // 更新新树的直径
                treeDiameter[rootU] = newDiameter;
            }
        } else if (op == 2) {
            // 查询操作
            int u = scanner.nextInt();
            int root = find(u);
            int a = treeDiameter[root][0];
            int b = treeDiameter[root][1];
            // 计算直径长度并输出
            System.out.println(getDistance(a, b));
        }
    }
    scanner.close();
}
```

}

==========

文件: Nod1803_ForestDiameter.py
==========

```
import sys
from collections import deque

"""
51Nod-1803 - 森林的直径
题目描述：有一个由 n 个节点组成的森林，每个节点属于一棵树。
支持两种操作：
1. 连接两棵树：输入格式为"1 u v"，表示将节点 u 所在的树与节点 v 所在的树合并
2. 查询操作：输入格式为"2 u"，表示询问节点 u 所在的树的直径

解题思路：
1. 使用并查集来管理各个树的合并
2. 对于每个树，维护其直径的两个端点
3. 当合并两棵树时，新树的直径只可能是原两棵树的直径之一，或者通过连接边形成的新路径（即 u 树的两个端点和 v 树的两个端点之间的最长路径）

时间复杂度分析：
- 并查集操作的均摊时间复杂度接近 O(1)
- 合并操作需要计算 4 种可能的路径长度，需要额外的 BFS/DFS 操作，单次时间复杂度为 O(n)，但实际应用中树的大小通常不大

空间复杂度：O(n)
"""

def main():
    sys.setrecursionlimit(1 << 25)
    input = sys.stdin.read().split()
    ptr = 0
    n = int(input[ptr])
    ptr += 1
    m = int(input[ptr])
    ptr += 1

    # 初始化并查集
    parent = list(range(n + 1))  # 节点编号从 1 开始

    # 初始化邻接表
```

```
graph = [[] for _ in range(n + 1)]

# 初始化每个树的直径（初始时每个节点自身就是一棵树）
tree_diameter = [[i, i] for i in range(n + 1)]

def find(x):
    """并查集的查找操作，带路径压缩"""
    if parent[x] != x:
        parent[x] = find(parent[x])
    return parent[x]

def bfs(start, visited):
    """BFS 函数，从指定节点开始，找到距离最远的节点和最远距离"""
    q = deque()
    distance = {}

    q.append(start)
    distance[start] = 0
    visited[start] = True

    max_distance = 0
    farthest_node = start

    while q:
        current = q.popleft()

        for neighbor in graph[current]:
            if not visited[neighbor]:
                visited[neighbor] = True
                new_distance = distance[current] + 1
                distance[neighbor] = new_distance
                q.append(neighbor)

                if new_distance > max_distance:
                    max_distance = new_distance
                    farthest_node = neighbor

    return farthest_node, max_distance

def get_distance(u, v):
    """计算两个节点之间的距离"""
    visited = [False] * (n + 1)
    q = deque()
```

```
    distance = {}

    q.append(u)
    distance[u] = 0
    visited[u] = True

    while q:
        current = q.popleft()

        if current == v:
            return distance[current]

        for neighbor in graph[current]:
            if not visited[neighbor]:
                visited[neighbor] = True
                distance[neighbor] = distance[current] + 1
                q.append(neighbor)

    return -1  # 应该不会到达这里，因为u和v在同一棵树中

def compute_diameter(root):
    """计算一棵树的直径"""
    visited = [False] * (n + 1)
    # 第一次BFS找到距离root最远的节点u
    u, _ = bfs(root, visited)

    # 重置visited数组
    visited = [False] * (n + 1)
    # 第二次BFS找到距离u最远的节点v
    v, _ = bfs(u, visited)

    return u, v

# 处理每个操作
for _ in range(m):
    op = int(input[ptr])
    ptr += 1
    if op == 1:
        # 连接操作
        u = int(input[ptr])
        ptr += 1
        v = int(input[ptr])
        ptr += 1
```

```
# 添加边
graph[u].append(v)
graph[v].append(u)

# 合并两个集合
root_u = find(u)
root_v = find(v)
if root_u != root_v:
    parent[root_v] = root_u

    # 计算新树的直径
    a1, a2 = tree_diameter[root_u]
    b1, b2 = tree_diameter[root_v]

    # 可能的六种路径
    d1 = get_distance(a1, a2)  # 原 u 树的直径
    d2 = get_distance(b1, b2)  # 原 v 树的直径
    d3 = get_distance(a1, b1)  # a1 到 b1
    d4 = get_distance(a1, b2)  # a1 到 b2
    d5 = get_distance(a2, b1)  # a2 到 b1
    d6 = get_distance(a2, b2)  # a2 到 b2

    # 找出最长的路径
    max_distance = d1
    new_diameter = [a1, a2]

    if d2 > max_distance:
        max_distance = d2
        new_diameter = [b1, b2]
    if d3 > max_distance:
        max_distance = d3
        new_diameter = [a1, b1]
    if d4 > max_distance:
        max_distance = d4
        new_diameter = [a1, b2]
    if d5 > max_distance:
        max_distance = d5
        new_diameter = [a2, b1]
    if d6 > max_distance:
        max_distance = d6
        new_diameter = [a2, b2]
```

```
                # 更新新树的直径
                tree_diameter[root_u] = new_diameter
        elif op == 2:
            # 查询操作
            u = int(input[ptr])
            ptr += 1
            root = find(u)
            a, b = tree_diameter[root]
            # 计算直径长度并输出
            print(get_distance(a, b))

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

==========

文件: Nod2602_TreeDiameter.cpp

==========

```
#include <iostream>
#include <vector>
#include <queue>
using namespace std;

/**
 * 51Nod-2602 - 树的直径
 * 题目描述：一棵树的直径就是这棵树上存在的最长路径。现在有一棵 n 个节点的树，现在想知道这棵树的直径包含的边的个数是多少？
 * 输入：第 1 行一个整数 n 表示节点个数，接下来 n-1 行每行两个整数 u,v 表示边
 * 输出：树的直径包含的边的个数
 *
 * 解题思路：使用两次 BFS 法求树的直径
 * 第一次 BFS 从任意节点出发找到最远节点 u，第二次 BFS 从 u 出发找到最远节点 v，u 到 v 的距离即为树的直径
 *
 * 时间复杂度：O(n)，空间复杂度：O(n)
 */

int n; // 节点数量
vector<vector<int>> graph; // 邻接表存储树结构

/**
 * BFS 函数，从指定节点开始，找到距离最远的节点和最远距离
 * @param start 起始节点
```

```
 * @return 包含最远节点和最远距离的 pair
 */
pair<int, int> bfs(int start) {
    vector<bool> visited(n + 1, false); // 标记节点是否被访问过
    vector<int> distance(n + 1, 0); // 存储每个节点到起始节点的距离
    queue<int> q;

    q.push(start);
    visited[start] = true;
    distance[start] = 0;

    int maxDistance = 0;
    int farthestNode = start;

    while (!q.empty()) {
        int current = q.front();
        q.pop();

        for (int neighbor : graph[current]) {
            if (!visited[neighbor]) {
                visited[neighbor] = true;
                distance[neighbor] = distance[current] + 1;
                q.push(neighbor);

                // 更新最远距离和最远节点
                if (distance[neighbor] > maxDistance) {
                    maxDistance = distance[neighbor];
                    farthestNode = neighbor;
                }
            }
        }
    }

    return {farthestNode, maxDistance};
}

int main() {
    cin >> n;

    // 初始化邻接表
    graph.resize(n + 1);

    // 读取边
```

```
    for (int i = 0; i < n - 1; i++) {
        int u, v;
        cin >> u >> v;
        // 无向树，添加双向边
        graph[u].push_back(v);
        graph[v].push_back(u);
    }

    // 第一次BFS，找到距离任意节点(这里选择1号节点)最远的节点u
    auto result1 = bfs(1);
    int u = result1.first;

    // 第二次BFS，找到距离u最远的节点v，此时的距离即为树的直径
    auto result2 = bfs(u);
    int diameter = result2.second;

    // 输出树的直径包含的边的个数
    cout << diameter << endl;

    return 0;
}
```

==========

文件: Nod2602_TreeDiameter.java

==========

```
package class121;

import java.util.*;

/**
 * 51Nod-2602 - 树的直径
 *
 * 题目链接：https://www.51nod.com/Challenge/Problem.html#!#problemId=2602
 * 题目描述：一棵树的直径就是这棵树上存在的最长路径。现在有一棵n个节点的树，现在想知道这棵树的直径包含的边的个数是多少？
 *
 * 输入格式：
 * - 第1行：一个整数n，表示树上的节点个数。(1<=n<=100000)
 * - 第2-n行：每行有两个整数u,v,表示u与v之间有一条路径。(1<=u,v<=n)
 *
 * 输出格式：
 * - 输出一个整数，表示这棵树直径所包含的边的个数。
```

```
 *
 * 解题思路：
 * 使用两次 BFS 法求树的直径：
 * 1. 第一次 BFS，从任意节点（如节点 1）开始找到最远节点 u
 * 2. 第二次 BFS，从第一次找到的最远节点 u 开始找到另一个最远节点 v
 * 3. u 到 v 的距离即为树的直径，也就是直径包含的边的个数
 *
 * 算法标签：树、广度优先搜索、两次 BFS
 * 难度：简单
 * 时间复杂度：O(n)，其中 n 是树中节点的数量
 * 空间复杂度：O(n)，用于存储邻接表和辅助数组
 *
 * 相关题目：
 * - LeetCode 543. 二叉树的直径
 * - LeetCode 1245. Tree Diameter (无向树的直径)
 * - LeetCode 1522. Diameter of N-Ary Tree (N 叉树的直径)
 * - SPOJ PT07Z - Longest path in a tree (树中最长路径)
 * - CSES 1131 - Tree Diameter (树的直径)
 * - 洛谷 U81904 树的直径
 * - AtCoder ABC221F - Diameter Set
 */
public class Nod2602_TreeDiameter {
    private static int n; // 节点数量
    private static List<List<Integer>> graph; // 邻接表存储树结构

    /**
     * BFS 函数，从指定节点开始，找到距离最远的节点和最远距离
     *
     * 算法思路：
     * 1. 从指定起点开始进行广度优先搜索
     * 2. 记录访问过的节点，避免重复访问
     * 3. 记录每个节点到起点的距离
     * 4. 在遍历过程中更新最远节点和最远距离
     *
     * @param start 起始节点
     * @return 包含最远节点和最远距离的数组 [最远节点, 最远距离]
     */
    private static int[] bfs(int start) {
        boolean[] visited = new boolean[n + 1]; // 标记节点是否被访问过
        int[] distance = new int[n + 1]; // 存储每个节点到起始节点的距离
        Queue<Integer> queue = new LinkedList<>();

        queue.offer(start);
```

```
        visited[start] = true;
        distance[start] = 0;

        int maxDistance = 0;
        int farthestNode = start;

        while (!queue.isEmpty()) {
            int current = queue.poll();

            for (int neighbor : graph.get(current)) {
                if (!visited[neighbor]) {
                    visited[neighbor] = true;
                    distance[neighbor] = distance[current] + 1;
                    queue.offer(neighbor);

                    // 更新最远距离和最远节点
                    if (distance[neighbor] > maxDistance) {
                        maxDistance = distance[neighbor];
                        farthestNode = neighbor;
                    }
                }
            }
        }

        return new int[] { farthestNode, maxDistance };
    }

    /**
     * 主方法
     *
     * 算法流程：
     * 1. 读取输入数据，构建树的邻接表表示
     * 2. 第一次BFS，找到距离任意节点(这里选择1号节点)最远的节点u
     * 3. 第二次BFS，找到距离u最远的节点v，此时的距离即为树的直径
     * 4. 输出树的直径包含的边的个数
     */
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        n = scanner.nextInt();

        // 初始化邻接表
        graph = new ArrayList<>();
        for (int i = 0; i <= n; i++) {
```

```
            graph.add(new ArrayList<>());
        }

        // 读取边
        for (int i = 0; i < n - 1; i++) {
            int u = scanner.nextInt();
            int v = scanner.nextInt();
            // 无向树，添加双向边
            graph.get(u).add(v);
            graph.get(v).add(u);
        }
        scanner.close();

        // 第一次 BFS，找到距离任意节点(这里选择 1 号节点)最远的节点 u
        int[] result1 = bfs(1);
        int u = result1[0];

        // 第二次 BFS，找到距离 u 最远的节点 v，此时的距离即为树的直径
        int[] result2 = bfs(u);
        int diameter = result2[1];

        // 输出树的直径包含的边的个数
        System.out.println(diameter);
    }
}
```

==========

文件: Nod2602_TreeDiameter.py

==========

```
import sys
from collections import deque

"""
51Nod-2602 - 树的直径
题目描述：一棵树的直径就是这棵树上存在的最长路径。现在有一棵 n 个节点的树，现在想知道这棵树的直径包含的边的个数是多少？
输入：第 1 行一个整数 n 表示节点个数，接下来 n-1 行每行两个整数 u,v 表示边
输出：树的直径包含的边的个数

解题思路：使用两次 BFS 法求树的直径
第一次 BFS 从任意节点出发找到最远节点 u，第二次 BFS 从 u 出发找到最远节点 v，u 到 v 的距离即为树的直径
```

时间复杂度：O(n)，空间复杂度：O(n)
"""

```
def bfs(start, graph, n):
    """
    BFS 函数，从指定节点开始，找到距离最远的节点和最远距离

    Args:
        start: 起始节点
        graph: 邻接表表示的树结构
        n: 节点数量

    Returns:
        tuple: (最远节点, 最远距离)
    """
    visited = [False] * (n + 1)  # 标记节点是否被访问过
    distance = [0] * (n + 1)     # 存储每个节点到起始节点的距离
    queue = deque()

    queue.append(start)
    visited[start] = True
    distance[start] = 0

    max_distance = 0
    farthest_node = start

    while queue:
        current = queue.popleft()

        for neighbor in graph[current]:
            if not visited[neighbor]:
                visited[neighbor] = True
                distance[neighbor] = distance[current] + 1
                queue.append(neighbor)

                # 更新最远距离和最远节点
                if distance[neighbor] > max_distance:
                    max_distance = distance[neighbor]
                    farthest_node = neighbor

    return farthest_node, max_distance

def main():
```

```
    # 读取输入
    n = int(sys.stdin.readline())

    # 初始化邻接表
    graph = [[] for _ in range(n + 1)]

    # 读取边
    for _ in range(n - 1):
        u, v = map(int, sys.stdin.readline().split())
        # 无向树，添加双向边
        graph[u].append(v)
        graph[v].append(u)

    # 第一次 BFS，找到距离任意节点(这里选择 1 号节点)最远的节点 u
    u, _ = bfs(1, graph, n)

    # 第二次 BFS，找到距离 u 最远的节点 v，此时的距离即为树的直径
    _, diameter = bfs(u, graph, n)

    # 输出树的直径包含的边的个数
    print(diameter)

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

```
==============================================
```

文件: SPOJ_MDST_MinimumDiameterSpanningTree.cpp

```
==============================================
// SPOJ MDST - Minimum Diameter Spanning Tree
// 题目：给定一个简单无向图 G 的邻接顶点列表，找到最小直径生成树 T，并输出该树的直径 diam(T)。
// 树的直径是指树中任意两点之间最长的简单路径。
// 来源：SPOJ Problem Set
// 链接：https://www.spoj.com/problems/MDST/

#define MAXN 501
#define INF 0x3f3f3f3f

int n, m;  // 节点数和边数
int graph[MAXN][MAXN];  // 邻接矩阵表示图
int dist[MAXN][MAXN];   // 所有点对之间的最短距离
int parent[MAXN][MAXN]; // 用于重构路径
```

```
/**
 * Floyd-Warshall 算法计算所有点对之间的最短距离
 *
 * 时间复杂度：O(n^3)
 * 空间复杂度：O(n^2)
 */
void floydWarshall() {
    // 初始化距离矩阵
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= n; j++) {
            if (i == j) {
                dist[i][j] = 0;
            } else if (graph[i][j] != 0) {
                dist[i][j] = graph[i][j];
            } else {
                dist[i][j] = INF;
            }
            parent[i][j] = i;
        }
    }

    // Floyd-Warshall 算法
    for (int k = 1; k <= n; k++) {
        for (int i = 1; i <= n; i++) {
            for (int j = 1; j <= n; j++) {
                if (dist[i][k] + dist[k][j] < dist[i][j]) {
                    dist[i][j] = dist[i][k] + dist[k][j];
                    parent[i][j] = parent[k][j];
                }
            }
        }
    }
}

/**
 * 通过绝对中心找到最小直径生成树
 * 绝对中心是边上的一个点，使得以该点为中心的生成树直径最小
 *
 * 时间复杂度：O(n^3)
 * 空间复杂度：O(n^2)
 *
 * @return 最小直径生成树的直径
 */
```

```
int findMinimumDiameterSpanningTree() {
    int minDiameter = INF;

    // 检查每个节点作为中心的情况
    for (int center = 1; center <= n; center++) {
        // 计算以 center 为根的生成树的直径
        int diameter = 0;
        for (int i = 1; i <= n; i++) {
            for (int j = 1; j <= n; j++) {
                if (i != j) {
                    diameter = (diameter > dist[i][j]) ? diameter : dist[i][j];
                }
            }
        }

        minDiameter = (minDiameter < diameter) ? minDiameter : diameter;
    }

    // 检查每条边上的点作为中心的情况
    for (int u = 1; u <= n; u++) {
        for (int v = u + 1; v <= n; v++) {
            if (graph[u][v] != 0) {
                // 边(u,v)上的点作为中心
                // 计算以这条边为中心的生成树的直径
                int diameter = 0;
                for (int i = 1; i <= n; i++) {
                    for (int j = 1; j <= n; j++) {
                        if (i != j) {
                            // 计算通过边(u,v)的最短路径
                            int dist1 = dist[i][u] + graph[u][v] + dist[v][j];
                            int dist2 = dist[i][v] + graph[u][v] + dist[u][j];
                            int distViaEdge = (dist1 < dist2) ? dist1 : dist2;
                            diameter = (diameter > distViaEdge) ? diameter : distViaEdge;
                        }
                    }
                }

                minDiameter = (minDiameter < diameter) ? minDiameter : diameter;
            }
        }
    }

    return minDiameter;
```

```
}

/**
 * 更高效的算法：使用绝对中心算法
 *
 * 时间复杂度：O(n^3)
 * 空间复杂度：O(n^2)
 *
 * @return 最小直径生成树的直径
 */
int findMinimumDiameterSpanningTreeOptimized() {
    int minDiameter = INF;

    // 对于每个节点作为中心
    for (int center = 1; center <= n; center++) {
        // 计算直径
        int diameter = 0;
        for (int i = 1; i <= n; i++) {
            for (int j = i + 1; j <= n; j++) {
                diameter = (diameter > dist[i][j]) ? diameter : dist[i][j];
            }
        }

        minDiameter = (minDiameter < diameter) ? minDiameter : diameter;
    }

    // 对于每条边作为中心
    for (int u = 1; u <= n; u++) {
        for (int v = u + 1; v <= n; v++) {
            if (graph[u][v] != 0) {
                // 计算通过边(u,v)的直径
                int diameter = 0;
                for (int i = 1; i <= n; i++) {
                    for (int j = i + 1; j <= n; j++) {
                        int dist1 = dist[i][u] + graph[u][v] + dist[v][j];
                        int dist2 = dist[i][v] + graph[u][v] + dist[u][j];
                        int distViaEdge = (dist1 < dist2) ? dist1 : dist2;
                        diameter = (diameter > distViaEdge) ? diameter : distViaEdge;
                    }
                }

                minDiameter = (minDiameter < diameter) ? minDiameter : diameter;
            }
```

```
        }
    }

    return minDiameter;
}

/**
 * 主方法
 * 时间复杂度：O(n^3)
 * 空间复杂度：O(n^2)
 */
int main() {
    // 由于编译环境限制，这里只展示算法实现
    // 实际使用时需要根据具体环境添加输入输出代码

    // 示例：n = 4, 一个简单的路径图
    n = 4;

    // 初始化图
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= n; j++) {
            graph[i][j] = 0;
        }
    }

    // 添加边：1-2, 2-3, 3-4
    graph[1][2] = graph[2][1] = 1;
    graph[2][3] = graph[3][2] = 1;
    graph[3][4] = graph[4][3] = 1;

    // 计算所有点对之间的最短距离
    floydWarshall();

    // 计算最小直径生成树的直径
    int result = findMinimumDiameterSpanningTreeOptimized();

    // printf("%d\n", result); // 应该输出某种结果

    return 0;
}

==============================================
```

文件: SPOJ_MDST_MinimumDiameterSpanningTree.java

```
==========================================
package class121;

// SPOJ MDST - Minimum Diameter Spanning Tree
// 题目：给定一个简单无向图 G 的邻接顶点列表，找到最小直径生成树 T，并输出该树的直径 diam(T)。
// 树的直径是指树中任意两点之间最长的简单路径。
// 来源：SPOJ Problem Set
// 链接：https://www.spoj.com/problems/MDST/

import java.io.*;
import java.util.*;

public class SPOJ_MDST_MinimumDiameterSpanningTree {

    static final int MAXN = 501;
    static final int INF = 0x3f3f3f3f;

    static int n, m;  // 节点数和边数
    static int[][] graph;  // 邻接矩阵表示图
    static int[][] dist;   // 所有点对之间的最短距离
    static int[][] parent; // 用于重构路径

    /**
     * Floyd-Warshall 算法计算所有点对之间的最短距离
     *
     * 时间复杂度：O(n^3)
     * 空间复杂度：O(n^2)
     */
    static void floydWarshall() {
        // 初始化距离矩阵
        for (int i = 1; i <= n; i++) {
            for (int j = 1; j <= n; j++) {
                if (i == j) {
                    dist[i][j] = 0;
                } else if (graph[i][j] != 0) {
                    dist[i][j] = graph[i][j];
                } else {
                    dist[i][j] = INF;
                }
                parent[i][j] = i;
            }
        }
```

```
        // Floyd-Warshall 算法
        for (int k = 1; k <= n; k++) {
            for (int i = 1; i <= n; i++) {
                for (int j = 1; j <= n; j++) {
                    if (dist[i][k] + dist[k][j] < dist[i][j]) {
                        dist[i][j] = dist[i][k] + dist[k][j];
                        parent[i][j] = parent[k][j];
                    }
                }
            }
        }
    }

    /**
     * 通过绝对中心找到最小直径生成树
     * 绝对中心是边上的一个点，使得以该点为中心的生成树直径最小
     *
     * 时间复杂度：O(n^3)
     * 空间复杂度：O(n^2)
     *
     * @return 最小直径生成树的直径
     */
    static int findMinimumDiameterSpanningTree() {
        int minDiameter = INF;

        // 检查每个节点作为中心的情况
        for (int center = 1; center <= n; center++) {
            // 对节点按到中心的距离排序
            Integer[] nodes = new Integer[n];
            for (int i = 0; i < n; i++) {
                nodes[i] = i + 1;
            }

            // 按照到中心的距离排序
            final int finalCenter = center;
            Arrays.sort(nodes, (a, b) -> Integer.compare(dist[finalCenter][a],
dist[finalCenter][b]));

            // 计算以 center 为根的生成树的直径
            int diameter = 0;
            for (int i = 1; i <= n; i++) {
                for (int j = 1; j <= n; j++) {
```

```
                if (i != j) {
                    diameter = Math.max(diameter, dist[i][j]);
                }
            }
        }

        minDiameter = Math.min(minDiameter, diameter);
    }

    // 检查每条边上的点作为中心的情况
    for (int u = 1; u <= n; u++) {
        for (int v = u + 1; v <= n; v++) {
            if (graph[u][v] != 0) {
                // 边(u,v)上的点作为中心
                // 计算以这条边为中心的生成树的直径
                int diameter = 0;
                for (int i = 1; i <= n; i++) {
                    for (int j = 1; j <= n; j++) {
                        if (i != j) {
                            // 计算通过边(u,v)的最短路径
                            int distViaEdge = Math.min(
                                dist[i][u] + graph[u][v] + dist[v][j],
                                dist[i][v] + graph[u][v] + dist[u][j]
                            );
                            diameter = Math.max(diameter, distViaEdge);
                        }
                    }
                }

                minDiameter = Math.min(minDiameter, diameter);
            }
        }
    }

    return minDiameter;
}

/**
 * 更高效的算法：使用绝对中心算法
 *
 * 时间复杂度：O(n^3)
 * 空间复杂度：O(n^2)
 *
```

```
     * @return 最小直径生成树的直径
     */
    static int findMinimumDiameterSpanningTreeOptimized() {
        int minDiameter = INF;

        // 对于每个节点作为中心
        for (int center = 1; center <= n; center++) {
            // 按距离排序所有节点
            Integer[] nodes = new Integer[n];
            for (int i = 0; i < n; i++) {
                nodes[i] = i + 1;
            }

            final int finalCenter = center;
            Arrays.sort(nodes, (a, b) -> Integer.compare(dist[finalCenter][a],
dist[finalCenter][b]));

            // 计算直径
            int diameter = 0;
            for (int i = 1; i <= n; i++) {
                for (int j = i + 1; j <= n; j++) {
                    diameter = Math.max(diameter, dist[i][j]);
                }
            }

            minDiameter = Math.min(minDiameter, diameter);
        }

        // 对于每条边作为中心
        for (int u = 1; u <= n; u++) {
            for (int v = u + 1; v <= n; v++) {
                if (graph[u][v] != 0) {
                    // 计算通过边(u,v)的直径
                    int diameter = 0;
                    for (int i = 1; i <= n; i++) {
                        for (int j = i + 1; j <= n; j++) {
                            int distViaEdge = Math.min(
                                dist[i][u] + graph[u][v] + dist[v][j],
                                dist[i][v] + graph[u][v] + dist[u][j]
                            );
                            diameter = Math.max(diameter, distViaEdge);
                        }
                    }
```

```
                minDiameter = Math.min(minDiameter, diameter);
            }
        }
    }

    return minDiameter;
}

/**
 * 主方法
 * 时间复杂度：O(n^3)
 * 空间复杂度：O(n^2)
 */
public static void main(String[] args) throws IOException {
    BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
    PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));

    String line;
    while ((line = br.readLine()) != null && !line.isEmpty()) {
        String[] parts = line.split(" ");
        n = Integer.parseInt(parts[0]);

        if (n == 0) break;

        // 初始化数据结构
        graph = new int[n + 1][n + 1];
        dist = new int[n + 1][n + 1];
        parent = new int[n + 1][n + 1];

        // 读取邻接信息
        for (int i = 1; i <= n; i++) {
            parts = br.readLine().split(" ");
            int degree = Integer.parseInt(parts[0]);
            for (int j = 1; j <= degree; j++) {
                int neighbor = Integer.parseInt(parts[j]);
                graph[i][neighbor] = 1;  // 无权图，边权为1
            }
        }

        // 计算所有点对之间的最短距离
        floydWarshall();
```

```
            // 计算最小直径生成树的直径
            int result = findMinimumDiameterSpanningTreeOptimized();
            out.println(result);
        }

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

==========

文件: SPOJ_MDST_MinimumDiameterSpanningTree.py

==========

```
# SPOJ MDST - Minimum Diameter Spanning Tree
# 题目：给定一个简单无向图 G 的邻接顶点列表，找到最小直径生成树 T，并输出该树的直径 diam(T)。
# 树的直径是指树中任意两点之间最长的简单路径。
# 来源：SPOJ Problem Set
# 链接：https://www.spoj.com/problems/MDST/

import sys
from collections import defaultdict

INF = 1000000000  # 使用一个大整数代替无穷大

class SPOJMDSTMinimumDiameterSpanningTree:
    def __init__(self):
        self.n = 0  # 节点数
        self.graph = []  # 邻接矩阵表示图
        self.dist = []   # 所有点对之间的最短距离
        self.parent = [] # 用于重构路径

    def floyd_warshall(self):
        """
        Floyd-Warshall 算法计算所有点对之间的最短距离

        时间复杂度：O(n^3)
        空间复杂度：O(n^2)
        """
        # 初始化距离矩阵
        for i in range(1, self.n + 1):
            for j in range(1, self.n + 1):
```

```
            if i == j:
                self.dist[i][j] = 0
            elif self.graph[i][j] != 0:
                self.dist[i][j] = self.graph[i][j]
            else:
                self.dist[i][j] = INF
            self.parent[i][j] = i

    # Floyd-Warshall 算法
    for k in range(1, self.n + 1):
        for i in range(1, self.n + 1):
            for j in range(1, self.n + 1):
                if self.dist[i][k] + self.dist[k][j] < self.dist[i][j]:
                    self.dist[i][j] = self.dist[i][k] + self.dist[k][j]
                    self.parent[i][j] = self.parent[k][j]

def find_minimum_diameter_spanning_tree(self):
    """
    通过绝对中心找到最小直径生成树
    绝对中心是边上的一个点，使得以该点为中心的生成树直径最小

    时间复杂度：O(n^3)
    空间复杂度：O(n^2)

    :return: 最小直径生成树的直径
    """
    min_diameter = INF

    # 检查每个节点作为中心的情况
    for center in range(1, self.n + 1):
        # 计算以 center 为根的生成树的直径
        diameter = 0
        for i in range(1, self.n + 1):
            for j in range(1, self.n + 1):
                if i != j:
                    diameter = max(diameter, self.dist[i][j])

        min_diameter = min(min_diameter, diameter)

    # 检查每条边上的点作为中心的情况
    for u in range(1, self.n + 1):
        for v in range(u + 1, self.n + 1):
            if self.graph[u][v] != 0:
```

```
                # 边(u,v)上的点作为中心
                # 计算以这条边为中心的生成树的直径
                diameter = 0
                for i in range(1, self.n + 1):
                    for j in range(1, self.n + 1):
                        if i != j:
                            # 计算通过边(u,v)的最短路径
                            dist_via_edge = min(
                                self.dist[i][u] + self.graph[u][v] + self.dist[v][j],
                                self.dist[i][v] + self.graph[u][v] + self.dist[u][j]
                            )
                            diameter = max(diameter, dist_via_edge)

                min_diameter = min(min_diameter, diameter)

    return min_diameter

def find_minimum_diameter_spanning_tree_optimized(self):
    """
    更高效的算法：使用绝对中心算法

    时间复杂度：O(n^3)
    空间复杂度：O(n^2)

    :return: 最小直径生成树的直径
    """
    min_diameter = INF

    # 对于每个节点作为中心
    for center in range(1, self.n + 1):
        # 计算直径
        diameter = 0
        for i in range(1, self.n + 1):
            for j in range(i + 1, self.n + 1):
                diameter = max(diameter, self.dist[i][j])

        min_diameter = min(min_diameter, diameter)

    # 对于每条边作为中心
    for u in range(1, self.n + 1):
        for v in range(u + 1, self.n + 1):
            if self.graph[u][v] != 0:
                # 计算通过边(u,v)的直径
```

```
                diameter = 0
                for i in range(1, self.n + 1):
                    for j in range(i + 1, self.n + 1):
                        dist_via_edge = min(
                            self.dist[i][u] + self.graph[u][v] + self.dist[v][j],
                            self.dist[i][v] + self.graph[u][v] + self.dist[u][j]
                        )
                        diameter = max(diameter, dist_via_edge)

                min_diameter = min(min_diameter, diameter)

    return min_diameter

def read_input_and_solve(self):
    """
    读取输入并求解
    时间复杂度：O(n^3)
    空间复杂度：O(n^2)
    """
    try:
        while True:
            line = input().strip()
            if not line:
                break

            parts = line.split()
            self.n = int(parts[0])

            if self.n == 0:
                break

            # 初始化数据结构
            self.graph = [[0] * (self.n + 1) for _ in range(self.n + 1)]
            self.dist = [[0] * (self.n + 1) for _ in range(self.n + 1)]
            self.parent = [[0] * (self.n + 1) for _ in range(self.n + 1)]

            # 读取邻接信息
            for i in range(1, self.n + 1):
                parts = input().split()
                degree = int(parts[0])
                for j in range(1, degree + 1):
                    neighbor = int(parts[j])
                    self.graph[i][neighbor] = 1  # 无权图，边权为1
```

```
                # 计算所有点对之间的最短距离
                self.floyd_warshall()

                # 计算最小直径生成树的直径
                result = self.find_minimum_diameter_spanning_tree_optimized()
                print(result)
        except EOFError:
            pass

# 主函数
if __name__ == "__main__":
    # 由于这是在线评测题目，实际提交时需要取消下面的注释
    # solution = SPOJMDSTMinimumDiameterSpanningTree()
    # solution.read_input_and_solve()

    # 示例测试
    solution = SPOJMDSTMinimumDiameterSpanningTree()
    solution.n = 4

    # 初始化数据结构
    solution.graph = [[0] * 5 for _ in range(5)]
    solution.dist = [[0] * 5 for _ in range(5)]
    solution.parent = [[0] * 5 for _ in range(5)]

    # 添加边：1-2, 2-3, 3-4
    solution.graph[1][2] = solution.graph[2][1] = 1
    solution.graph[2][3] = solution.graph[3][2] = 1
    solution.graph[3][4] = solution.graph[4][3] = 1

    # 计算所有点对之间的最短距离
    solution.floyd_warshall()

    # 计算最小直径生成树的直径
    result = solution.find_minimum_diameter_spanning_tree_optimized()
    print("示例输出:", result)

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

文件: SPOJ_PT07Z_LongestPathInTree.cpp
==============================================
// SPOJ PT07Z - Longest path in a tree
// 题目：给定一个无权无向树，求树中最长路径的长度。
```

```
// 算法标签：树、广度优先搜索、两次 BFS
// 难度：简单
// 时间复杂度：O(n)，其中 n 是树中节点的数量
// 空间复杂度：O(n)，用于存储邻接表和辅助数组

// 相关题目：
// - LeetCode 543. 二叉树的直径
// - LeetCode 1245. Tree Diameter (无向树的直径)
// - LeetCode 1522. Diameter of N-Ary Tree (N 叉树的直径)
// - CSES 1131 - Tree Diameter (树的直径)
// - 51Nod 2602 - 树的直径
// - 洛谷 U81904 树的直径
// - AtCoder ABC221F - Diameter Set

// 解题思路：
// 使用两次 BFS 法求解树的直径：
// 1. 从任意一点开始，找到距离它最远的点 s
// 2. 从 s 开始，找到距离它最远的点 t
// 3. s 到 t 的距离即为树的直径

const int MAXN = 10001;

// 邻接表存储树
int graph[MAXN][MAXN];  // 简化的邻接表表示
int graph_size[MAXN];   // 每个节点的邻接点数量
int n;  // 节点数

// Pair 结构体存储节点和距离
struct Pair {
    int node;
    int distance;

    Pair(int n, int d) : node(n), distance(d) {}
};

/**
 * BFS 求从起点开始的最远节点
 *
 * 算法思路：
 * 1. 从指定起点开始进行广度优先搜索
 * 2. 记录访问过的节点，避免重复访问
 * 3. 记录每一层的节点，直到遍历完所有节点
```

```
 * 4. 返回最后一层的节点（最远节点）和距离
 *
 * @param start 起点
 * @return Pair 对象，包含最远节点和距离
 *
 * 时间复杂度：O(n)
 * 空间复杂度：O(n)
 */
Pair bfs(int start) {
    bool visited[MAXN];
    // 初始化 visited 数组
    for (int i = 0; i <= n; i++) {
        visited[i] = false;
    }

    int queue[MAXN];
    int front = 0, rear = 0;

    visited[start] = true;
    queue[rear++] = start;

    int lastNode = start;
    int maxDistance = 0;

    while (front < rear) {
        int size = rear - front;
        for (int i = 0; i < size; i++) {
            int current = queue[front++];
            lastNode = current;

            // 遍历当前节点的所有邻居
            for (int j = 0; j < graph_size[current]; j++) {
                int neighbor = graph[current][j];
                if (!visited[neighbor]) {
                    visited[neighbor] = true;
                    queue[rear++] = neighbor;
                }
            }
        }
        if (front < rear) {
            maxDistance++;
        }
    }
```

```
    return Pair(lastNode, maxDistance);
}

/**
 * 使用两次 BFS 法求树的直径
 *
 * 算法思路：
 * 1. 第一次 BFS，从任意节点（如节点 1）开始找到最远节点
 * 2. 第二次 BFS，从第一次找到的最远节点开始找到另一个最远节点
 * 3. 第二次 BFS 的距离就是树的直径
 *
 * @return 树的直径
 *
 * 时间复杂度：O(n)
 * 空间复杂度：O(n)
 */
int findDiameter() {
    // 第一次 BFS，从节点 1 开始找到最远节点
    Pair firstBFS = bfs(1);

    // 第二次 BFS，从第一次找到的最远节点开始找到另一个最远节点
    Pair secondBFS = bfs(firstBFS.node);

    // 第二次 BFS 的距离就是树的直径
    return secondBFS.distance;
}

// 主方法（用于测试）
int main() {
    // 由于这是 SPOJ 题目，实际提交时需要按照题目要求的输入格式处理
    // 这里我们只展示算法实现

    // 示例输入：
    // n = 4
    // 边: 1-2, 2-3, 3-4
    // 预期输出：3

    n = 4;
    // 初始化图
    for (int i = 0; i <= n; i++) {
        graph_size[i] = 0;
    }
```

```
    // 添加边
    graph[1][graph_size[1]++] = 2;
    graph[2][graph_size[2]++] = 1;
    graph[2][graph_size[2]++] = 3;
    graph[3][graph_size[3]++] = 2;
    graph[3][graph_size[3]++] = 4;
    graph[4][graph_size[4]++] = 3;

    // 输出结果
    int diameter = findDiameter();
    // 应该输出 3
    return 0;
}
```

==========

文件: SPOJ_PT07Z_LongestPathInTree.java

==========

```
package class121;

// SPOJ PT07Z - Longest path in a tree
// 题目：给定一个无权无向树，求树中最长路径的长度。

// 算法标签：树、广度优先搜索、两次 BFS
// 难度：简单
// 时间复杂度：O(n)，其中 n 是树中节点的数量
// 空间复杂度：O(n)，用于存储邻接表和辅助数组

// 相关题目：
// - LeetCode 543. 二叉树的直径
// - LeetCode 1245. Tree Diameter (无向树的直径)
// - LeetCode 1522. Diameter of N-Ary Tree (N 叉树的直径)
// - CSES 1131 - Tree Diameter (树的直径)
// - 51Nod 2602 - 树的直径
// - 洛谷 U81904 树的直径
// - AtCoder ABC221F - Diameter Set

// 解题思路：
// 使用两次 BFS 法求解树的直径：
// 1. 从任意一点开始，找到距离它最远的点 s
// 2. 从 s 开始，找到距离它最远的点 t
// 3. s 到 t 的距离即为树的直径
```

```
import java.io.*;
import java.util.*;

public class SPOJ_PT07Z_LongestPathInTree {

    static final int MAXN = 10001;

    // 邻接表存储树
    static ArrayList<Integer>[] graph;
    static int n;  // 节点数

    // BFS 方法求从起点开始的最远节点和距离
    static class Pair {
        int node;
        int distance;

        Pair(int node, int distance) {
            this.node = node;
            this.distance = distance;
        }
    }

    /**
     * BFS 求从起点开始的最远节点
     *
     * 算法思路：
     * 1. 从指定起点开始进行广度优先搜索
     * 2. 记录访问过的节点，避免重复访问
     * 3. 记录每一层的节点，直到遍历完所有节点
     * 4. 返回最后一层的节点（最远节点）和距离
     *
     * @param start 起点
     * @return Pair 对象，包含最远节点和距离
     *
     * 时间复杂度：O(n)
     * 空间复杂度：O(n)
     */
    static Pair bfs(int start) {
        boolean[] visited = new boolean[n + 1];
        Queue<Integer> queue = new LinkedList<>();

        visited[start] = true;
```

```
    queue.offer(start);

    int lastNode = start;
    int maxDistance = 0;

    while (!queue.isEmpty()) {
        int size = queue.size();
        for (int i = 0; i < size; i++) {
            int current = queue.poll();
            lastNode = current;

            // 遍历当前节点的所有邻居
            for (int neighbor : graph[current]) {
                if (!visited[neighbor]) {
                    visited[neighbor] = true;
                    queue.offer(neighbor);
                }
            }
        }
        if (!queue.isEmpty()) {
            maxDistance++;
        }
    }

    return new Pair(lastNode, maxDistance);
}

/**
 * 使用两次 BFS 法求树的直径
 *
 * 算法思路：
 * 1. 第一次 BFS，从任意节点（如节点 1）开始找到最远节点
 * 2. 第二次 BFS，从第一次找到的最远节点开始找到另一个最远节点
 * 3. 第二次 BFS 的距离就是树的直径
 *
 * @return 树的直径
 *
 * 时间复杂度：O(n)
 * 空间复杂度：O(n)
 */
static int findDiameter() {
    // 第一次 BFS，从节点 1 开始找到最远节点
    Pair firstBFS = bfs(1);
```

```
        // 第二次 BFS，从第一次找到的最远节点开始找到另一个最远节点
        Pair secondBFS = bfs(firstBFS.node);

        // 第二次 BFS 的距离就是树的直径
        return secondBFS.distance;
    }

    // 主方法（用于测试）
    public static void main(String[] args) throws IOException {
        // 由于这是 SPOJ 题目，实际提交时需要按照题目要求的输入格式处理
        // 这里我们只展示算法实现

        // 示例输入：
        // n = 4
        // 边: 1-2, 2-3, 3-4
        // 预期输出：3

        n = 4;
        graph = new ArrayList[n + 1];
        for (int i = 1; i <= n; i++) {
            graph[i] = new ArrayList<>();
        }

        // 添加边
        graph[1].add(2);
        graph[2].add(1);
        graph[2].add(3);
        graph[3].add(2);
        graph[3].add(4);
        graph[4].add(3);

        System.out.println("树的直径: " + findDiameter()); // 应该输出 3
    }
}
```

==========

文件: SPOJ_PT07Z_LongestPathInTree.py

==========

```
# SPOJ PT07Z - Longest path in a tree
# 题目：给定一个无权无向树，求树中最长路径的长度。
```

```
# 算法标签：树、广度优先搜索、两次 BFS
# 难度：简单
# 时间复杂度：O(n)，其中 n 是树中节点的数量
# 空间复杂度：O(n)，用于存储邻接表和辅助数组

# 相关题目：
# - LeetCode 543. 二叉树的直径
# - LeetCode 1245. Tree Diameter (无向树的直径)
# - LeetCode 1522. Diameter of N-Ary Tree (N 叉树的直径)
# - CSES 1131 - Tree Diameter (树的直径)
# - 51Nod 2602 - 树的直径
# - 洛谷 U81904 树的直径
# - AtCoder ABC221F - Diameter Set

# 解题思路：
# 使用两次 BFS 法求解树的直径：
# 1. 从任意一点开始，找到距离它最远的点 s
# 2. 从 s 开始，找到距离它最远的点 t
# 3. s 到 t 的距离即为树的直径

from collections import deque, defaultdict

# 全局变量
n = 0  # 节点数
graph = defaultdict(list)  # 邻接表存储树

def bfs(start):
    """
    BFS 求从起点开始的最远节点

    算法思路：
    1. 从指定起点开始进行广度优先搜索
    2. 记录访问过的节点，避免重复访问
    3. 记录每一层的节点，直到遍历完所有节点
    4. 返回最后一层的节点（最远节点）和距离

    参数:
        start (int): 起点节点编号

    返回:
        tuple: (最远节点, 距离)

    时间复杂度：O(n)
```

```
    空间复杂度：O(n)
    """
    visited = [False] * (n + 1)
    queue = deque()

    visited[start] = True
    queue.append(start)

    last_node = start
    max_distance = 0

    while queue:
        size = len(queue)
        for _ in range(size):
            current = queue.popleft()
            last_node = current

            # 遍历当前节点的所有邻居
            for neighbor in graph[current]:
                if not visited[neighbor]:
                    visited[neighbor] = True
                    queue.append(neighbor)

        if queue:
            max_distance += 1

    return (last_node, max_distance)

def find_diameter():
    """
    使用两次 BFS 法求树的直径

    算法思路：
    1. 第一次 BFS，从任意节点（如节点 1）开始找到最远节点
    2. 第二次 BFS，从第一次找到的最远节点开始找到另一个最远节点
    3. 第二次 BFS 的距离就是树的直径

    返回:
        int: 树的直径

    时间复杂度：O(n)
    空间复杂度：O(n)
    """
```

```
    # 第一次 BFS，从节点 1 开始找到最远节点
    first_bfs_node, _ = bfs(1)

    # 第二次 BFS，从第一次找到的最远节点开始找到另一个最远节点
    _, diameter = bfs(first_bfs_node)

    # 第二次 BFS 的距离就是树的直径
    return diameter

# 主方法（用于测试）
if __name__ == "__main__":
    # 由于这是 SPOJ 题目，实际提交时需要按照题目要求的输入格式处理
    # 这里我们只展示算法实现

    # 示例输入：
    # n = 4
    # 边: 1-2, 2-3, 3-4
    # 预期输出：3

    n = 4
    graph.clear()

    # 添加边
    graph[1].append(2)
    graph[2].append(1)
    graph[2].append(3)
    graph[3].append(2)
    graph[3].append(4)
    graph[4].append(3)

    print("树的直径:", find_diameter())  # 应该输出 3

==============================================
```