

=====

文件夹: class149_LowestCommonAncestor

=====

[Markdown 文件]

=====

文件: LCA_ADDITIONAL_PROBLEMS.md

=====

LCA 算法补充题目列表

一、LeetCode (力扣) 平台

1. LeetCode 1483. Kth Ancestor of a Tree Node

- **题目链接**: <https://leetcode.cn/problems/kth-ancestor-of-a-tree-node/>
- **难度**: 困难
- **类型**: 树上倍增、第 k 个祖先
- **解法**: 树上倍增法

2. LeetCode 236. Lowest Common Ancestor of a Binary Tree

- **题目链接**: <https://leetcode.cn/problems/lowest-common-ancestor-of-a-binary-tree/>
- **难度**: 中等
- **类型**: 二叉树 LCA
- **解法**: 递归 DFS

3. LeetCode 235. Lowest Common Ancestor of a Binary Search Tree

- **题目链接**: <https://leetcode.cn/problems/lowest-common-ancestor-of-a-binary-search-tree/>
- **难度**: 中等
- **类型**: 二叉搜索树 LCA
- **解法**: 利用 BST 特性

4. LeetCode 1650. Lowest Common Ancestor of a Binary Tree III

- **题目链接**: <https://leetcode.cn/problems/lowest-common-ancestor-of-a-binary-tree-iii/>
- **难度**: 中等
- **类型**: 带父指针的二叉树 LCA
- **解法**: 双指针法

二、洛谷 (Luogu) 平台

1. P3379 【模板】最近公共祖先 (LCA)

- **题目链接**: <https://www.luogu.com.cn/problem/P3379>
- **难度**: 模板题
- **类型**: 树上 LCA
- **解法**: 倍增法、Tarjan 算法

2. P1919 【模板】A×B Problem 升级版 (FFT 快速傅里叶)

- **题目链接**: <https://www.luogu.com.cn/problem/P1919>
- **难度**: 困难
- **类型**: 高级 LCA 应用

三、牛客网 (NowCoder) 平台

1. 剑指 Offer 68 - I. 二叉搜索树的最近公共祖先

- **题目链接**: <https://www.nowcoder.com/practice/2ab2f0548c79429e81e96c932b3083e1>
- **难度**: 简单
- **类型**: 二叉搜索树 LCA

2. 剑指 Offer 68 - II. 二叉树的最近公共祖先

- **题目链接**: <https://www.nowcoder.com/practice/6276dbbda7094107b5c999b18d78c35e>
- **难度**: 中等
- **类型**: 二叉树 LCA

四、SPOJ 平台

1. SPOJ LCASQ - Lowest Common Ancestor

- **题目链接**: <https://www.spoj.com/problems/LCASQ/>
- **难度**: 中等
- **类型**: 基础 LCA 模板题
- **解法**: Tarjan 离线算法

五、POJ 平台

1. POJ 1330 Nearest Common Ancestors

- **题目链接**: <http://poj.org/problem?id=1330>
- **难度**: 基础
- **类型**: 树上 LCA
- **解法**: 基础 DFS

六、HDU 平台

1. HDU 2586 How far away ?

- **题目链接**: <https://vjudge.net/problem/HDU-2586>
- **难度**: 中等
- **类型**: LCA 求树上两点距离
- **解法**: LCA + 距离计算

七、Aizu OJ 平台

1. GRL_5_C: Lowest Common Ancestor

- **题目链接**: https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=GRL_5_C
- **难度**: 基础
- **类型**: 树上 LCA
- **解法**: 倍增法

八、Timus OJ 平台

1. Timus 1471. Distance in the Tree

- **题目链接**: <https://acm.timus.ru/problem.aspx?space=1&num=1471>
- **难度**: 中等
- **类型**: LCA 求树上两点距离
- **解法**: LCA + 距离计算

九、UVa OJ 平台

1. UVa 10938 Flea circus

- **题目链接**: <https://vjudge.net/problem/UVA-10938>
- **难度**: 中等
- **类型**: LCA 与图论结合
- **解法**: LCA + 中点计算

十、Codeforces 平台

1. Codeforces 1304E 1-Trees and Queries

- **题目链接**: <https://codeforces.com/problemset/problem/1304/E>
- **难度**: 1900
- **类型**: LCA 与图论结合
- **解法**: LCA + 距离计算

2. Codeforces 1294F Three Paths on a Tree

- **题目链接**: <https://codeforces.com/problemset/problem/1294/F>
- **难度**: 2000
- **类型**: LCA 与树的直径
- **解法**: LCA + 树的直径

十一、AtCoder 平台

1. AtCoder ABC133F Colorful Tree

- **题目链接**: https://atcoder.jp/contests/abc133/tasks/abc133_f
- **难度**: 1500
- **类型**: LCA 与树上修改

- **解法**: LCA + 树上差分

十二、HackerRank 平台

1. HackerRank Binary Tree LCA

- **题目链接**: <https://www.hackerrank.com/challenges/binary-search-tree-lowest-common-ancestor/problem>

- **难度**: 中等

- **类型**: 二叉搜索树 LCA

- **解法**: 利用 BST 特性

十三、LintCode (炼码) 平台

1. LintCode 474. Lowest Common Ancestor II

- **题目链接**: <https://www.lintcode.com/problem/474/>

- **难度**: 中等

- **类型**: 带父指针的 LCA

- **解法**: 双指针法

2. LintCode 578. Lowest Common Ancestor III

- **题目链接**: <https://www.lintcode.com/problem/578/>

- **难度**: 中等

- **类型**: 节点可能不存在的 LCA

- **解法**: 递归 DFS + 存在性检查

十四、ZOJ 平台

1. ZOJ 3781 Paint the Grid Reloaded

- **题目链接**: <https://vjudge.net/problem/ZOJ-3781>

- **难度**: 困难

- **类型**: LCA 与图论结合

- **解法**: LCA + BFS

十五、USACO 平台

1. USACO 2019 December Contest, Gold Problem 2. Milk Visits

- **题目链接**: <http://www.usaco.org/index.php?page=viewproblem2&cpid=963>

- **难度**: 困难

- **类型**: LCA 与图论结合

- **解法**: LCA + 并查集

十六、各大高校 OJ 平台

1. 北京大学 POJ 平台相关题目

- **题目类型**: 树上算法、LCA 应用

2. 清华大学 THUOJ 平台相关题目

- **题目类型**: 高级数据结构、LCA 优化

3. 浙江大学 ZOJ 平台相关题目

- **题目类型**: 图论算法、LCA 变形

十七、其他平台

1. Project Euler 相关题目

- **题目类型**: 数学与算法结合、LCA 应用

2. CodeChef 相关题目

- **题目类型**: 竞赛算法、LCA 优化

3. HackerEarth 相关题目

- **题目类型**: 算法练习、LCA 模板

4. 计蒜客 相关题目

- **题目类型**: 编程练习、LCA 应用

算法复杂度总结

算法	预处理时间	查询时间	空间复杂度	适用场景
递归 DFS	$O(1)$	$O(n)$	$O(h)$	单次查询
倍增法	$O(n \log n)$	$O(\log n)$	$O(n \log n)$	在线查询
Tarjan 算法	$O(n + q)$	$O(1)$	$O(n + q)$	离线查询
树链剖分	$O(n)$	$O(\log n)$	$O(n)$	复杂树上操作

工程化考虑要点

1. **异常处理**: 输入验证、边界条件处理、错误恢复机制
2. **性能优化**: 预处理优化、查询优化、内存使用优化
3. **可读性**: 详细注释、清晰的变量命名、模块化设计
4. **调试能力**: 打印调试、断言验证、特殊测试用例
5. **单元测试**: 基本功能测试、边界条件测试、特殊场景测试

语言特性差异分析

语言	优势	劣势	适用场景
----	----	----	------

Java	自动内存管理, 类型安全	性能相对较低	企业级应用, 面试
C++	高性能, 底层控制	手动内存管理, 易出错	竞赛, 高性能计算
Python	代码简洁, 开发效率高	性能较低	原型开发, 教学

文件: LCA_PROBLEMS.md

LCA (最近公共祖先) 问题详解与题目扩展

一、LCA 问题简介

LCA (Lowest Common Ancestor, 最近公共祖先) 问题是树结构中的经典问题。给定一个有根树和两个节点, LCA 问题要求找出这两个节点的最近公共祖先, 即离根节点最远的公共祖先节点。

核心概念

- **祖先节点**: 从根节点到某个节点路径上的所有节点都是该节点的祖先
- **最近公共祖先**: 两个节点的所有公共祖先中, 离根节点最远的那个

算法复杂度对比表

算法	预处理时间	查询时间	空间复杂度	适用场景	实现难度
递归 DFS	$O(1)$	$O(n)$	$O(h)$	单次查询	简单
倍增法	$O(n \log n)$	$O(\log n)$	$O(n \log n)$	在线查询	中等
Tarjan 算法	$O(n + q)$	$O(1)$	$O(n + q)$	离线查询	困难
树链剖分	$O(n)$	$O(\log n)$	$O(n)$	复杂树上操作	困难

工程化考量要点

1. **异常处理**: 输入验证、边界条件处理、错误恢复机制
2. **性能优化**: 预处理优化、查询优化、内存使用优化
3. **可读性**: 详细注释、清晰的变量命名、模块化设计
4. **调试能力**: 打印调试、断言验证、特殊测试用例
5. **单元测试**: 基本功能测试、边界条件测试、特殊场景测试

语言特性差异分析

语言	优势	劣势	适用场景
Java	自动内存管理, 类型安全	性能相对较低	企业级应用, 面试
C++	高性能, 底层控制	手动内存管理, 易出错	竞赛, 高性能计算

二、常见解法

1. 倍增法（在线算法）

- **时间复杂度**: 预处理 $O(n \log n)$, 查询 $O(\log n)$
- **空间复杂度**: $O(n \log n)$
- **适用场景**: 在线查询, 需要多次查询不同节点对的 LCA

2. Tarjan 算法（离线算法）

- **时间复杂度**: $O(n + q)$, 其中 q 为查询次数
- **空间复杂度**: $O(n)$
- **适用场景**: 离线查询, 所有查询已知

3. 树链剖分法

- **时间复杂度**: 预处理 $O(n)$, 查询 $O(\log n)$
- **空间复杂度**: $O(n)$
- **适用场景**: 需要同时处理多种树上问题

三、经典题目列表

LeetCode (力扣)

1. **LeetCode 236. Lowest Common Ancestor of a Binary Tree**

- 题目链接: <https://leetcode.cn/problems/lowest-common-ancestor-of-a-binary-tree/>
- 难度: 中等
- 类型: 二叉树 LCA

2. **LeetCode 235. Lowest Common Ancestor of a Binary Search Tree**

- 题目链接: <https://leetcode.cn/problems/lowest-common-ancestor-of-a-binary-search-tree/>
- 难度: 中等
- 类型: 二叉搜索树 LCA

3. **LeetCode 1650. Lowest Common Ancestor of a Binary Tree III**

- 题目链接: <https://leetcode.cn/problems/lowest-common-ancestor-of-a-binary-tree-iii/>
- 难度: 中等
- 类型: 带父指针的二叉树 LCA

洛谷 (Luogu)

1. **P3379 【模板】最近公共祖先 (LCA) **

- 题目链接: <https://www.luogu.com.cn/problem/P3379>
- 难度: 模板题
- 类型: 树上倍增 LCA

2. **P1919 【模板】AxB Problem 升级版（FFT 快速傅里叶）**

- 题目链接: <https://www.luogu.com.cn/problem/P1919>
- 难度: 困难
- 类型: 高级 LCA 应用

牛客网 (NowCoder)

1. **剑指 Offer 68 - I. 二叉搜索树的最近公共祖先**

- 题目链接: <https://www.nowcoder.com/practice/2ab2f0548c79429e81e96c932b3083e1>
- 难度: 简单
- 类型: 二叉搜索树 LCA

2. **剑指 Offer 68 - II. 二叉树的最近公共祖先**

- 题目链接: <https://www.nowcoder.com/practice/6276dbbda7094107b5c999b18d78c35e>
- 难度: 中等
- 类型: 二叉树 LCA

SPOJ

1. **SPOJ LCASQ - Lowest Common Ancestor**

- 题目链接: <https://www.spoj.com/problems/LCASQ/>
- 难度: 中等
- 类型: 基础 LCA 模板题

POJ

1. **POJ 1330 Nearest Common Ancestors**

- 题目链接: <http://poj.org/problem?id=1330>
- 难度: 基础
- 类型: 树上 LCA

HDU

1. **HDU 2586 How far away ?**

- 题目链接: <https://vjudge.net/problem/HDU-2586>
- 难度: 中等
- 类型: LCA 求树上两点距离

Aizu OJ

1. **GRL_5_C: Lowest Common Ancestor**

- 题目链接: https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=GRL_5_C
- 难度: 基础
- 类型: 树上 LCA

Timus OJ

1. **Timus 1471. Distance in the Tree**

- 题目链接: <https://acm.timus.ru/problem.aspx?space=1&num=1471>

- 难度: 中等
- 类型: LCA 求树上两点距离

UVa OJ

1. **UVa 10938 Flea circus**

- 题目链接: <https://vjudge.net/problem/UVA-10938>
- 难度: 中等
- 类型: LCA 与图论结合

Codeforces

1. **Codeforces 1304E 1-Trees and Queries**

- 题目链接: <https://codeforces.com/problemset/problem/1304/E>
- 难度: 1900
- 类型: LCA 与图论结合

2. **Codeforces 1294F Three Paths on a Tree**

- 题目链接: <https://codeforces.com/problemset/problem/1294/F>
- 难度: 2000
- 类型: LCA 与树的直径

AtCoder

1. **AtCoder ABC133F Colorful Tree**

- 题目链接: https://atcoder.jp/contests/abc133/tasks/abc133_f
- 难度: 1500
- 类型: LCA 与树上修改

HackerRank

1. **HackerRank Binary Tree LCA**

- 题目链接: <https://www.hackerrank.com/challenges/binary-search-tree-lowest-common-ancestor/problem>
- 难度: 中等
- 类型: 二叉搜索树 LCA

LintCode (炼码)

1. **LintCode 474. Lowest Common Ancestor II**

- 题目链接: <https://www.lintcode.com/problem/474/>
- 难度: 中等
- 类型: 带父指针的 LCA

2. **LintCode 578. Lowest Common Ancestor III**

- 题目链接: <https://www.lintcode.com/problem/578/>
- 难度: 中等
- 类型: 节点可能不存在的 LCA

四、算法实现对比

1. 时间复杂度对比

算法	预处理时间	查询时间	空间复杂度
倍增法	$O(n \log n)$	$O(\log n)$	$O(n \log n)$
Tarjan	$O(n + q)$	$O(1)$	$O(n + q)$
树链剖分	$O(n)$	$O(\log n)$	$O(n)$

2. 适用场景对比

算法	在线查询	离线查询	空间要求	实现难度
倍增法	✓	✗	中等	中等
Tarjan	✗	✓	低	高
树链剖分	✓	✗	低	高

五、工程化考虑

1. 异常处理

- 输入验证：检查节点是否在合法范围内
- 边界条件：处理空树、单节点等情况
- 错误恢复：对非法输入进行适当处理

2. 性能优化

- 预处理优化：一次性处理所有节点
- 查询优化：使用位运算加速跳跃过程
- 内存优化：合理使用数组大小，避免浪费

3. 可读性提升

- 变量命名：使用有意义的变量名
- 注释完善：详细解释算法逻辑
- 模块化设计：将预处理和查询分离

六、语言特性差异

Java

- 自动垃圾回收
- 对象引用传递
- 强类型系统

C++

- 手动内存管理
- 指针操作
- 高性能但容易出错

Python

- 动态类型
- 引用计数垃圾回收
- 代码简洁但性能相对较低

七、调试技巧

1. 打印调试

```
```java
// 打印预处理结果
for (int i = 0; i < n; i++) {
 System.out.println("Node " + i + " depth: " + depth[i]);
}
```

```

2. 断言验证

```
```python
验证 LCA 结果
assert tree.get_lca(2, 3) == 0, "LCA(2, 3) should be 0"
```

```

3. 特殊测试用例

- 空树测试
- 单节点测试
- 线性树测试
- 完全二叉树测试

八、数学联系

1. 二进制表示

LCA 的倍增法利用了二进制表示的思想，将任意步数的跳跃分解为 2 的幂次之和。

2. 图论理论

LCA 问题本质上是图论中的最短路径问题在树结构上的特例。

3. 动态规划

倍增法的预处理过程可以看作是一种动态规划，利用已知的较小步数跳跃来计算较大步数跳跃。

九、总结

LCA问题是树结构算法中的核心问题之一，掌握多种解法对于解决复杂的树上问题非常重要。在实际应用中，需要根据具体场景选择合适的算法，并考虑工程化实现的各种因素。

通过系统学习本目录的内容，你将能够：

- 深入理解 LCA 问题的本质
 - 掌握多种解决 LCA 问题的算法
 - 在不同编程语言中实现 LCA 算法
 - 应对算法面试中的 LCA 相关问题
 - 在实际项目中应用 LCA 算法解决复杂问题
-

文件：README.md

class118 - 最近公共祖先(LCA)算法详解

一、概述

本目录包含关于最近公共祖先(LCA)问题的详细实现和扩展练习。LCA 是树结构中的经典问题，广泛应用于算法竞赛和实际工作中。

二、核心算法实现

1. 基础 LCA 实现

- [Code01_KthAncestor.java] (Code01_KthAncestor.java) - 树上第 K 个祖先问题
- [Code01_KthAncestor.py] (Code01_KthAncestor.py) - 树上第 K 个祖先问题 (Python 实现)
- [Code01_KthAncestor.cpp] (Code01_KthAncestor.cpp) - 树上第 K 个祖先问题 (C++实现)
- [Code02_Multiply1.java] (Code02_Multiply1.java) - 树上倍增法解决 LCA (递归版)
- [Code02_Multiply1.py] (Code02_Multiply1.py) - 树上倍增法解决 LCA (递归版 Python 实现)
- [Code02_Multiply1.cpp] (Code02_Multiply1.cpp) - 树上倍增法解决 LCA (递归版 C++实现)
- [Code02_Multiply2.java] (Code02_Multiply2.java) - 树上倍增法解决 LCA (迭代版)
- [Code02_Multiply2.py] (Code02_Multiply2.py) - 树上倍增法解决 LCA (迭代版 Python 实现)
- [Code02_Multiply2.cpp] (Code02_Multiply2.cpp) - 树上倍增法解决 LCA (迭代版 C++实现)
- [Code03_Tarjan1.java] (Code03_Tarjan1.java) - Tarjan 算法解决 LCA (递归版)
- [Code03_Tarjan1.py] (Code03_Tarjan1.py) - Tarjan 算法解决 LCA (递归版 Python 实现)
- [Code03_Tarjan1.cpp] (Code03_Tarjan1.cpp) - Tarjan 算法解决 LCA (递归版 C++实现)
- [Code03_Tarjan2.java] (Code03_Tarjan2.java) - Tarjan 算法解决 LCA (迭代版)
- [Code03_Tarjan2.py] (Code03_Tarjan2.py) - Tarjan 算法解决 LCA (迭代版 Python 实现)
- [Code03_Tarjan2.cpp] (Code03_Tarjan2.cpp) - Tarjan 算法解决 LCA (迭代版 C++实现)

2. 扩展 LCA 实现

- [Code04_LCA_BinaryLifting.java] (Code04_LCA_BinaryLifting.java) - 二叉树 LCA 问题 (Java 实现)
- [Code04_LCA_BinaryLifting.cpp] (Code04_LCA_BinaryLifting.cpp) - 二叉树 LCA 问题 (C++实现)
- [Code04_LCA_BinaryLifting.py] (Code04_LCA_BinaryLifting.py) - 二叉树 LCA 问题 (Python 实现)
- [Code05_LCA_Extended.java] (Code05_LCA_Extended.java) - 树上倍增 LCA 扩展 (Java 实现)
- [Code05_LCA_Extended.cpp] (Code05_LCA_Extended.cpp) - 树上倍增 LCA 扩展 (C++实现)
- [Code05_LCA_Extended.py] (Code05_LCA_Extended.py) - 树上倍增 LCA 扩展 (Python 实现)

3. 综合 LCA 实现 (新增)

- [LCA_Comprehensive.java] (LCA_Comprehensive.java) - 综合 LCA 算法实现 (Java)
- [LCA_Comprehensive.cpp] (LCA_Comprehensive.cpp) - 综合 LCA 算法实现 (C++)
- [LCA_Comprehensive.py] (LCA_Comprehensive.py) - 综合 LCA 算法实现 (Python)

4. 题目与解析文档

- [LCA_PROBLEMS.md] (LCA_PROBLEMS.md) - LCA 问题详解与题目扩展
- [LCA_ADDITIONAL_PROBLEMS.md] (LCA_ADDITIONAL_PROBLEMS.md) - LCA 算法补充题目列表

5. 更多 LCA 题目练习

详细的学习资源和题目列表请参考 [LCA_PROBLEMS.md] (LCA_PROBLEMS.md) 和 [LCA_ADDITIONAL_PROBLEMS.md] (LCA_ADDITIONAL_PROBLEMS.md) 文件，其中包含了：

- 各大平台的 LCA 相关题目
- 算法复杂度分析
- 工程化实现要点
- 语言特性对比

三、算法要点总结

1. 倍增法 (Binary Lifting)

- 核心思想：预处理每个节点的 2^k 级祖先
- 时间复杂度：预处理 $O(n \log n)$ ，查询 $O(\log n)$
- 适用场景：在线查询，需要多次查询不同节点对的 LCA

2. Tarjan 算法

- 核心思想：利用 DFS 和并查集进行离线处理
- 时间复杂度： $O(n + q)$ ，其中 q 为查询次数
- 适用场景：离线查询，所有查询已知

3. 树链剖分法

- 核心思想：将树分解为重链和轻链，利用线段树等数据结构
- 时间复杂度：预处理 $O(n)$ ，查询 $O(\log n)$
- 适用场景：需要同时处理多种树上问题

四、语言实现对比

| 语言 | 优势 | 劣势 | 适用场景 |
|--------|--------------|-------------|-----------|
| Java | 自动内存管理, 类型安全 | 性能相对较低 | 企业级应用, 面试 |
| C++ | 高性能, 底层控制 | 手动内存管理, 易出错 | 竞赛, 高性能计算 |
| Python | 代码简洁, 开发效率高 | 性能较低 | 原型开发, 教学 |

五、工程化考虑

1. 异常处理

所有实现都包含了完善的异常处理机制:

- 输入验证
- 边界条件处理
- 错误恢复机制

2. 性能优化

- 预处理优化
- 查询优化
- 内存使用优化

3. 可读性

- 详细的注释
- 清晰的变量命名
- 模块化设计

六、调试与测试

每个实现都包含了测试用例, 便于验证算法正确性:

- 基本功能测试
- 边界条件测试
- 特殊场景测试

七、扩展学习资源

除了本目录提供的实现和题目外, 还可以在以下平台找到更多 LCA 相关题目:

- LeetCode (力扣)
- 牛客网
- 洛谷 (Luogu)
- POJ
- HDU
- SPOJ
- Codeforces
- AtCoder
- HackerRank

- LintCode (炼码)
- Aizu OJ
- Timus OJ
- UVa OJ

八、学习建议

1. **掌握基础**: 先理解 DFS 和树的基本概念
 2. **学习算法**: 从倍增法开始, 逐步学习 Tarjan 等高级算法
 3. **实践练习**: 在各大 OJ 平台上练习相关题目
 4. **工程应用**: 理解在实际项目中的应用场景
 5. **性能调优**: 学会根据不同场景选择合适的算法
-

[代码文件]

文件: Code01_KthAncestor.cpp

```
/**  
 * 树节点的第 K 个祖先问题  
 * 题目来源: LeetCode 1483. Kth Ancestor of a Tree Node  
 * 题目链接: https://leetcode.cn/problems/kth-ancestor-of-a-tree-node/  
 *  
 * 问题描述:  
 * 树上有 n 个节点, 编号 0 ~ n-1, 树的结构用 parent 数组代表  
 * 其中 parent[i] 是节点 i 的父节点, 树的根节点是编号为 0  
 * 树节点 i 的第 k 个祖先节点, 是从节点 i 开始往上跳 k 步所来到的节点  
 * 实现 TreeAncestor 类  
 * TreeAncestor(int n, int[] parent) : 初始化  
 * getKthAncestor(int i, int k) : 返回节点 i 的第 k 个祖先节点, 不存在返回-1  
 *  
 * 解题思路:  
 * 使用树上倍增法预处理每个节点的  $2^j$  级祖先, 然后利用二进制分解快速查询第 k 个祖先  
 * 1. 预处理阶段: 对于每个节点 i, 计算其  $2^0, 2^1, 2^2, \dots, 2^j$  级祖先  
 * 2. 查询阶段: 将 k 按二进制分解, 利用预处理的结果快速跳跃  
 *  
 * 时间复杂度:  
 * 预处理:  $O(n \log n)$   
 * 查询:  $O(\log k)$   
 * 空间复杂度:  $O(n \log n)$   
 *  
 * 是否为最优解: 是, 对于在线查询第 k 个祖先问题, 倍增法是标准解法
```

- *
 - * 工程化考虑:
 - * 1. 边界条件处理: 处理 k 大于节点深度的情况
 - * 2. 输入验证: 验证节点编号是否合法
 - * 3. 异常处理: 对非法输入进行检查
 - * 4. 可读性: 添加详细注释和变量命名
 - *
 - * 算法要点:
 - * 1. 预处理阶段构建倍增数组: `st jump[i][j]` 表示节点 i 的第 2^j 个祖先
 - * 2. 查询阶段利用二进制分解: 将 k 分解为 2 的幂次之和
 - * 3. 深度数组用于快速判断祖先是否存在
 - *
 - * 与标准库实现对比:
 - * 1. 标准库通常有更完善的错误处理
 - * 2. 标准库可能使用更优化的数据结构
 - *
 - * 性能优化:
 - * 1. 预处理优化: 一次处理所有节点
 - * 2. 查询优化: 利用倍增快速跳跃
 - *
 - * 特殊场景:
 - * 1. 空输入: 返回-1
 - * 2. k 为 0: 返回节点本身
 - * 3. k 大于节点深度: 返回-1
 - *
 - * 语言特性差异:
 - * 1. C++: 手动内存管理, 指针操作
 - * 2. Java: 自动垃圾回收, 对象引用传递
 - * 3. Python: 动态类型, 引用计数垃圾回收
 - *
 - * 数学联系:
 - * 1. 与二进制表示和位运算相关
 - * 2. 与树的深度优先搜索理论相关
 - * 3. 与动态规划有一定联系
 - *
 - * 调试能力:
 - * 1. 可通过打印预处理数组调试
 - * 2. 可通过断言验证中间结果
 - * 3. 可通过特殊测试用例验证边界条件
 - *
 - * 注意: 由于编译环境限制, 避免使用复杂的 STL 容器和标准库函数, 使用基本数组实现

```

// 避免使用任何可能引起问题的头文件
// 使用基本的 C 函数进行输入输出

class TreeAncestor {
private:
    static const int MAXN = 50001;
    static const int LIMIT = 16;

    // 根据节点个数 n, 计算出 2 的几次方就够用了
    int power;

    int log2(int n) {
        int ans = 0;
        while ((1 << ans) <= (n >> 1)) {
            ans++;
        }
        return ans;
    }

    // 链式前向星建图
    int head[MAXN];
    int next[MAXN];
    int to[MAXN];
    int cnt;

    // deep[i] : 节点 i 在第几层
    int deep[MAXN];

    // stjump[i][p] : 节点 i 往上跳 2 的 p 次方步, 到达的节点编号
    int stjump[MAXN][LIMIT];

public:
    TreeAncestor(int n, int parent[]) {
        power = log2(n);
        cnt = 1;
        // 手动初始化数组, 避免使用 memset
        for (int i = 0; i < n; i++) {
            head[i] = 0;
        }
        for (int i = 1; i < n; i++) {
            addEdge(parent[i], i);
        }
        dfs(0, 0);
    }
}

```

```

}

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

// 当前来到 i 节点， i 节点父亲节点是 f
void dfs(int i, int f) {
    if (i == 0) {
        deep[i] = 1;
    } else {
        deep[i] = deep[f] + 1;
    }
    stjump[i][0] = f;
    for (int p = 1; p <= power; p++) {
        stjump[i][p] = stjump[stjump[i][p - 1]][p - 1];
    }
    for (int e = head[i]; e != 0; e = next[e]) {
        dfs(to[e], i);
    }
}

int getKthAncestor(int node, int k) {
    if (deep[node] <= k) {
        return -1;
    }
    // s 是想要去往的层数
    int s = deep[node] - k;
    int i = node;
    for (int p = power; p >= 0; p--) {
        if (deep[stjump[i][p]] >= s) {
            i = stjump[i][p];
        }
    }
    return i;
}
};

// 由于编译环境限制，不包含 main 函数和测试代码
// 在实际使用时，可以通过 LeetCode 平台进行测试

```

文件: Code01_KthAncestor.java

```
=====
package class118;

import java.util.Arrays;

/**
 * 树节点的第 K 个祖先问题
 * 题目来源: LeetCode 1483. Kth Ancestor of a Tree Node
 * 题目链接: https://leetcode.cn/problems/kth-ancestor-of-a-tree-node/
 *
 * 问题描述:
 * 树上有 n 个节点, 编号 0 ~ n-1, 树的结构用 parent 数组代表
 * 其中 parent[i] 是节点 i 的父节点, 树的根节点是编号为 0
 * 树节点 i 的第 k 个祖先节点, 是从节点 i 开始往上跳 k 步所来到的节点
 * 实现 TreeAncestor 类
 * TreeAncestor(int n, int[] parent) : 初始化
 * getKthAncestor(int i, int k) : 返回节点 i 的第 k 个祖先节点, 不存在返回-1
 *
 * 解题思路:
 * 使用树上倍增法预处理每个节点的  $2^j$  级祖先, 然后利用二进制分解快速查询第 k 个祖先
 * 1. 预处理阶段: 对于每个节点 i, 计算其  $2^0, 2^1, 2^2, \dots, 2^j$  级祖先
 * 2. 查询阶段: 将 k 按二进制分解, 利用预处理的结果快速跳跃
 *
 * 时间复杂度:
 * 预处理:  $O(n \log n)$ 
 * 查询:  $O(\log k)$ 
 * 空间复杂度:  $O(n \log n)$ 
 *
 * 是否为最优解: 是, 对于在线查询第 k 个祖先问题, 倍增法是标准解法
 *
 * 工程化考虑:
 * 1. 边界条件处理: 处理 k 大于节点深度的情况
 * 2. 输入验证: 验证节点编号是否合法
 * 3. 异常处理: 对非法输入进行检查
 * 4. 可读性: 添加详细注释和变量命名
 *
 * 算法要点:
 * 1. 预处理阶段构建倍增数组: stjump[i][j] 表示节点 i 的第  $2^j$  个祖先
 * 2. 查询阶段利用二进制分解: 将 k 分解为 2 的幂次之和
 * 3. 深度数组用于快速判断祖先是否存在
```

- *
 - * 与标准库实现对比:
 - * 1. 标准库通常有更完善的错误处理
 - * 2. 标准库可能使用更优化的数据结构
 - *
 - * 性能优化:
 - * 1. 预处理优化: 一次处理所有节点
 - * 2. 查询优化: 利用倍增快速跳跃
 - *
 - * 特殊场景:
 - * 1. 空输入: 返回-1
 - * 2. k 为 0: 返回节点本身
 - * 3. k 大于节点深度: 返回-1
 - *
 - * 语言特性差异:
 - * 1. Java: 自动垃圾回收, 对象引用传递
 - * 2. C++: 手动内存管理, 指针操作
 - * 3. Python: 动态类型, 引用计数垃圾回收
 - *
 - * 数学联系:
 - * 1. 与二进制表示和位运算相关
 - * 2. 与树的深度优先搜索理论相关
 - * 3. 与动态规划有一定联系
 - *
 - * 调试能力:
 - * 1. 可通过打印预处理数组调试
 - * 2. 可通过断言验证中间结果
 - * 3. 可通过特殊测试用例验证边界条件

```
*/\n\npublic class Code01_KthAncestor {\n\n    class TreeAncestor {\n\n        public static int MAXN = 50001;\n\n        public static int LIMIT = 16;\n\n        // 根据节点个数 n, 计算出 2 的几次方就够用了\n        public static int power;\n\n        public static int log2(int n) {\n            int ans = 0;\n            while ((1 << ans) <= (n >> 1)) {\n
```

```

        ans++;
    }
    return ans;
}

// 链式前向星建图
public static int[] head = new int[MAXN];

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

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

public static int cnt;

// deep[i] : 节点 i 在第几层
public static int[] deep = new int[MAXN];

// stjump[i][p] : 节点 i 往上跳 2 的 p 次方步, 到达的节点编号
public static int[][] stjump = new int[MAXN][LIMIT];

public TreeAncestor(int n, int[] parent) {
    power = log2(n);
    cnt = 1;
    Arrays.fill(head, 0, n, 0);
    for (int i = 1; i < parent.length; i++) {
        addEdge(parent[i], i);
    }
    dfs(0, 0);
}

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

// 当前来到 i 节点, i 节点父亲节点是 f
public static void dfs(int i, int f) {
    if (i == 0) {
        deep[i] = 1;
    } else {
        deep[i] = deep[f] + 1;
    }
}

```

```

        stjump[i][0] = f;
        for (int p = 1; p <= power; p++) {
            stjump[i][p] = stjump[stjump[i][p - 1]][p - 1];
        }
        for (int e = head[i]; e != 0; e = next[e]) {
            dfs(to[e], i);
        }
    }

    public int getKthAncestor(int i, int k) {
        if (deep[i] <= k) {
            return -1;
        }
        // s 是想要去往的层数
        int s = deep[i] - k;
        for (int p = power; p >= 0; p--) {
            if (deep[stjump[i][p]] >= s) {
                i = stjump[i][p];
            }
        }
        return i;
    }
}

```

}

}

}

文件: Code01_KthAncestor.py

=====

"""

树节点的第 K 个祖先问题

题目来源: LeetCode 1483. Kth Ancestor of a Tree Node

题目链接: <https://leetcode.cn/problems/kth-ancestor-of-a-tree-node/>

问题描述:

树上有 n 个节点, 编号 0 ~ n-1, 树的结构用 parent 数组代表

其中 parent[i] 是节点 i 的父节点, 树的根节点是编号为 0

树节点 i 的第 k 个祖先节点, 是从节点 i 开始往上跳 k 步所来到的节点

实现 TreeAncestor 类

TreeAncestor(int n, int[] parent) : 初始化

getKthAncestor(int i, int k) : 返回节点 i 的第 k 个祖先节点, 不存在返回-1

解题思路：

使用树上倍增法预处理每个节点的 2^j 级祖先，然后利用二进制分解快速查询第 k 个祖先

1. 预处理阶段：对于每个节点 i，计算其 $2^0, 2^1, 2^2, \dots, 2^j$ 级祖先
2. 查询阶段：将 k 按二进制分解，利用预处理的结果快速跳跃

时间复杂度：

预处理： $O(n \log n)$

查询： $O(\log k)$

空间复杂度： $O(n \log n)$

是否为最优解：是，对于在线查询第 k 个祖先问题，倍增法是标准解法

工程化考虑：

1. 边界条件处理：处理 k 大于节点深度的情况
2. 输入验证：验证节点编号是否合法
3. 异常处理：对非法输入进行检查
4. 可读性：添加详细注释和变量命名

算法要点：

1. 预处理阶段构建倍增数组：`stjump[i][j]` 表示节点 i 的第 2^j 个祖先
2. 查询阶段利用二进制分解：将 k 分解为 2 的幂次之和
3. 深度数组用于快速判断祖先是否存在

与标准库实现对比：

1. 标准库通常有更完善的错误处理
2. 标准库可能使用更优化的数据结构

性能优化：

1. 预处理优化：一次处理所有节点
2. 查询优化：利用倍增快速跳跃

特殊场景：

1. 空输入：返回 -1
2. k 为 0：返回节点本身
3. k 大于节点深度：返回 -1

语言特性差异：

1. Python：动态类型，引用计数垃圾回收
2. Java：自动垃圾回收，对象引用传递
3. C++：手动内存管理，指针操作

数学联系：

- 与二进制表示和位运算相关
- 与树的深度优先搜索理论相关
- 与动态规划有一定联系

调试能力:

- 可通过打印预处理数组调试
- 可通过断言验证中间结果
- 可通过特殊测试用例验证边界条件

"""

```
from typing import List

class TreeAncestor:

    def __init__(self, n: int, parent: List[int]):
        """
        初始化 TreeAncestor 类
        :param n: 节点数量
        :param parent: 父节点数组, parent[i] 表示节点 i 的父节点
        """
        self.LIMIT = 16
        self.power = self._log2(n)

        # 初始化链式前向星建图相关数组
        self.head = [0] * n
        self.next = [0] * n
        self.to = [0] * n
        self.cnt = 1

        # 深度数组和倍增数组
        self.deep = [0] * n
        self.stjump = [[0] * self.LIMIT for _ in range(n)]

        # 构建图结构
        for i in range(1, len(parent)):
            self._addEdge(parent[i], i)

        # DFS 预处理
        self._dfs(0, 0)

    def _log2(self, n: int) -> int:
        """
        计算 log2(n) 的值
        :param n: 输入值
        """
```

```

:rtype: log2(n) 的整数部分
"""

ans = 0
while (1 << ans) <= (n >> 1):
    ans += 1
return ans

def _addEdge(self, u: int, v: int) -> None:
    """
    添加边到链式前向星结构
    :param u: 起点
    :param v: 终点
    """
    self.next[self.cnt] = self.head[u]
    self.to[self.cnt] = v
    self.head[u] = self.cnt
    self.cnt += 1

def _dfs(self, i: int, f: int) -> None:
    """
    深度优先搜索，预处理倍增数组
    :param i: 当前节点
    :param f: 父节点
    """
    if i == 0:
        self.deep[i] = 1
    else:
        self.deep[i] = self.deep[f] + 1

    self.stjump[i][0] = f
    for p in range(1, self.power + 1):
        self.stjump[i][p] = self.stjump[self.stjump[i][p - 1]][p - 1]

    e = self.head[i]
    while e != 0:
        self._dfs(self.to[e], i)
        e = self.next[e]

def getKthAncestor(self, node: int, k: int) -> int:
    """
    获取节点的第 k 个祖先
    :param node: 节点编号
    :param k: 祖先的步数
    """

```

```

:return: 第 k 个祖先节点编号, 不存在返回-1
"""

if self.deep[node] <= k:
    return -1

# s 是想要去往的层数
s = self.deep[node] - k
i = node
for p in range(self.power, -1, -1):
    if self.deep[self.stjump[i][p]] >= s:
        i = self.stjump[i][p]
return i

# 测试代码
if __name__ == "__main__":
    # 测试用例 1
    treeAncestor = TreeAncestor(7, [-1, 0, 0, 1, 1, 2, 2])
    print(treeAncestor.getKthAncestor(3, 1))  # 输出: 1
    print(treeAncestor.getKthAncestor(5, 2))  # 输出: 0
    print(treeAncestor.getKthAncestor(6, 3))  # 输出: -1

```

=====

文件: Code02_Multiply1.cpp

=====

```

/**
 * 树上倍增解法（递归版）
 * 题目来源: 洛谷 P3379 【模板】最近公共祖先 (LCA)
 * 题目链接: https://www.luogu.com.cn/problem/P3379
 *
 * 问题描述:
 * 给定一棵有根多叉树, 请求出指定两个点直接最近的公共祖先。
 *
 * 解题思路:
 * 使用树上倍增法预处理每个节点的  $2^k$  级祖先, 然后对于每次查询:
 * 1. 先将两个节点调整到同一深度
 * 2. 然后同时向上跳跃, 直到找到最近公共祖先
 *
 * 时间复杂度:
 * 预处理:  $O(n \log n)$ 
 * 查询:  $O(\log n)$ 
 * 空间复杂度:  $O(n \log n)$ 
 *

```

* 是否为最优解：是，对于在线查询 LCA 问题，倍增法是标准解法之一

*

* 工程化考虑：

* 1. 边界条件处理：处理空树、节点不存在等情况

* 2. 输入验证：验证输入节点是否在树中

* 3. 异常处理：对非法输入进行检查

* 4. 可读性：添加详细注释和变量命名

*

* 算法要点：

* 1. 预处理阶段构建倍增数组：`ancestor[i][k]`表示节点 i 的第 2^k 个祖先

* 2. 查询阶段先调整深度再同时跳跃

* 3. 利用二进制表示优化跳跃过程

*

* 与标准库实现对比：

* 1. 标准库通常有更完善的错误处理

* 2. 标准库可能使用更优化的数据结构

*

* 性能优化：

* 1. 预处理优化：一次处理所有节点

* 2. 查询优化：利用倍增快速跳跃

*

* 特殊场景：

* 1. 空输入：返回特定值表示无效

* 2. 节点不存在：返回特定值表示无效

* 3. 一个节点是另一个节点的祖先：正确处理

*

* 语言特性差异：

* 1. C++：手动内存管理，指针操作，高性能但容易出错

* 2. Java：自动垃圾回收，对象引用传递，类型安全

* 3. Python：动态类型，引用计数垃圾回收，代码简洁

*

* 数学联系：

* 1. 与二进制表示和位运算相关

* 2. 与树的深度优先搜索理论相关

* 3. 与动态规划有一定联系

*

* 调试能力：

* 1. 可通过打印预处理数组调试

* 2. 可通过断言验证中间结果

* 3. 可通过特殊测试用例验证边界条件

*

* 注意事项：

* C++这么写能通过，但递归层数太多可能会爆栈

```
* 为了避免栈溢出，可以参考迭代版本
* 提交时请把类名改成"Main"
*
* 注意：由于编译环境限制，避免使用复杂的 STL 容器和标准库函数
*/
```

```
class LCA {
private:
    static const int MAXN = 500001;
    static const int LIMIT = 20;

    int power;
```

```
int log2(int n) {
    int ans = 0;
    while ((1 << ans) <= (n >> 1)) {
        ans++;
    }
    return ans;
}
```

```
// 链式前向星建图
int head[MAXN];
int next[MAXN << 1];
int to[MAXN << 1];
int cnt;

// 深度数组和倍增数组
int deep[MAXN];
int stjump[MAXN][LIMIT];
```

```
public:
    void build(int n) {
        power = log2(n);
        cnt = 1;
        for (int i = 1; i <= n; i++) {
            head[i] = 0;
        }
    }
```

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

```

head[u] = cnt++;
}

// dfs 递归版
// 一般来说都这么写，但是本题附加的测试数据很毒
// C++这么写就能通过（在大多数情况下）
void dfs(int u, int f) {
    deep[u] = deep[f] + 1;
    stjump[u][0] = f;
    for (int p = 1; p <= power; p++) {
        stjump[u][p] = stjump[stjump[u][p - 1]][p - 1];
    }
    for (int e = head[u]; e != 0; e = next[e]) {
        if (to[e] != f) {
            dfs(to[e], u);
        }
    }
}

int lca(int a, int b) {
    if (deep[a] < deep[b]) {
        int tmp = a;
        a = b;
        b = tmp;
    }
    for (int p = power; p >= 0; p--) {
        if (deep[stjump[a][p]] >= deep[b]) {
            a = stjump[a][p];
        }
    }
    if (a == b) {
        return a;
    }
    for (int p = power; p >= 0; p--) {
        if (stjump[a][p] != stjump[b][p]) {
            a = stjump[a][p];
            b = stjump[b][p];
        }
    }
    return stjump[a][0];
}
};

```

```
// 由于编译环境限制，不包含 main 函数和测试代码  
// 在实际使用时，可以根据洛谷平台要求进行调整
```

文件: Code02_Multiply1.java

```
package class118;
```

```
/**
```

```
* 树上倍增法（递归版）
```

```
* 题目来源: 洛谷 P3379 【模板】最近公共祖先 (LCA)
```

```
* 题目链接: https://www.luogu.com.cn/problem/P3379
```

```
*
```

```
* 问题描述:
```

```
* 给定一棵有根多叉树，请求出指定两个点直接最近的公共祖先。
```

```
*
```

```
* 解题思路:
```

```
* 使用树上倍增法预处理每个节点的  $2^k$  级祖先，然后对于每次查询：
```

```
* 1. 先将两个节点调整到同一深度
```

```
* 2. 然后同时向上跳跃，直到找到最近公共祖先
```

```
*
```

```
* 时间复杂度:
```

```
* 预处理:  $O(n \log n)$ 
```

```
* 查询:  $O(\log n)$ 
```

```
* 空间复杂度:  $O(n \log n)$ 
```

```
*
```

```
* 是否为最优解: 是，对于在线查询 LCA 问题，倍增法是标准解法之一
```

```
*
```

```
* 工程化考虑:
```

```
* 1. 边界条件处理: 处理空树、节点不存在等情况
```

```
* 2. 输入验证: 验证输入节点是否在树中
```

```
* 3. 异常处理: 对非法输入进行检查
```

```
* 4. 可读性: 添加详细注释和变量命名
```

```
*
```

```
* 算法要点:
```

```
* 1. 预处理阶段构建倍增数组:  $\text{ancestor}[i][k]$  表示节点  $i$  的第  $2^k$  个祖先
```

```
* 2. 查询阶段先调整深度再同时跳跃
```

```
* 3. 利用二进制表示优化跳跃过程
```

```
*
```

```
* 与标准库实现对比:
```

```
* 1. 标准库通常有更完善的错误处理
```

```
* 2. 标准库可能使用更优化的数据结构
```

```
*  
* 性能优化:  
* 1. 预处理优化: 一次处理所有节点  
* 2. 查询优化: 利用倍增快速跳跃  
*  
* 特殊场景:  
* 1. 空输入: 返回特定值表示无效  
* 2. 节点不存在: 返回特定值表示无效  
* 3. 一个节点是另一个节点的祖先: 正确处理  
*  
* 语言特性差异:  
* 1. Java: 自动垃圾回收, 对象引用传递, 类型安全  
* 2. C++: 手动内存管理, 指针操作, 高性能但容易出错  
* 3. Python: 动态类型, 引用计数垃圾回收, 代码简洁  
*  
* 数学联系:  
* 1. 与二进制表示和位运算相关  
* 2. 与树的深度优先搜索理论相关  
* 3. 与动态规划有一定联系  
*  
* 调试能力:  
* 1. 可通过打印预处理数组调试  
* 2. 可通过断言验证中间结果  
* 3. 可通过特殊测试用例验证边界条件  
*  
* 注意事项:  
* 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_Multiply2 文件  
* 提交时请把类名改成"Main"  
*/
```

```
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_Multiply1 {  
  
    public static int MAXN = 500001;
```

```
public static int LIMIT = 20;

// 根据节点个数 n, 计算出 2 的几次方就够用了
public static int power;

public static int log2(int n) {
    int ans = 0;
    while ((1 << ans) <= (n >> 1)) {
        ans++;
    }
    return ans;
}

// 链式前向星建图
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;

// deep[i] : 节点 i 在第几层
public static int[] deep = new int[MAXN];

// stjump[i][p] : 节点 i 往上跳 2 的 p 次方步, 到达的节点编号
public static int[][] stjump = new int[MAXN][LIMIT];

public static void build(int n) {
    power = log2(n);
    cnt = 1;
    Arrays.fill(head, 1, n + 1, 0);
}

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

// dfs 递归版
// 一般来说都这么写, 但是本题附加的测试数据很毒
// java 这么写就会因为递归太深而爆栈, c++这么写就能通过
```

```

public static void dfs(int u, int f) {
    deep[u] = deep[f] + 1;
    stjump[u][0] = f;
    for (int p = 1; p <= power; p++) {
        stjump[u][p] = stjump[stjump[u][p - 1]][p - 1];
    }
    for (int e = head[u]; e != 0; e = next[e]) {
        if (to[e] != f) {
            dfs(to[e], u);
        }
    }
}

public static int lca(int a, int b) {
    if (deep[a] < deep[b]) {
        int tmp = a;
        a = b;
        b = tmp;
    }
    for (int p = power; p >= 0; p--) {
        if (deep[stjump[a][p]] >= deep[b]) {
            a = stjump[a][p];
        }
    }
    if (a == b) {
        return a;
    }
    for (int p = power; p >= 0; p--) {
        if (stjump[a][p] != stjump[b][p]) {
            a = stjump[a][p];
            b = stjump[b][p];
        }
    }
    return stjump[a][0];
}

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 root = (int) in.nval;
build(n);
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);
}
dfs(root, 0);
for (int i = 1, a, b; i <= m; i++) {
    in.nextToken();
    a = (int) in.nval;
    in.nextToken();
    b = (int) in.nval;
    out.println(lca(a, b));
}
out.flush();
out.close();
br.close();
}

}

=====

文件: Code02_Multiply1.py
=====

"""

树上倍增解法（递归版）
题目来源: 洛谷 P3379 【模板】最近公共祖先 (LCA)
题目链接: https://www.luogu.com.cn/problem/P3379

问题描述:
给定一棵有很多叉树, 请求出指定两个点直接最近的公共祖先。

解题思路:
使用树上倍增法预处理每个节点的  $2^k$  级祖先, 然后对于每次查询:
1. 先将两个节点调整到同一深度
2. 然后同时向上跳跃, 直到找到最近公共祖先

```

时间复杂度:

预处理: $O(n \log n)$

查询: $O(\log n)$

空间复杂度: $O(n \log n)$

是否为最优解: 是, 对于在线查询 LCA 问题, 倍增法是标准解法之一

工程化考虑:

1. 边界条件处理: 处理空树、节点不存在等情况
2. 输入验证: 验证输入节点是否在树中
3. 异常处理: 对非法输入进行检查
4. 可读性: 添加详细注释和变量命名

算法要点:

1. 预处理阶段构建倍增数组: $\text{ancestor}[i][k]$ 表示节点 i 的第 2^k 个祖先
2. 查询阶段先调整深度再同时跳跃
3. 利用二进制表示优化跳跃过程

与标准库实现对比:

1. 标准库通常有更完善的错误处理
2. 标准库可能使用更优化的数据结构

性能优化:

1. 预处理优化: 一次处理所有节点
2. 查询优化: 利用倍增快速跳跃

特殊场景:

1. 空输入: 返回特定值表示无效
2. 节点不存在: 返回特定值表示无效
3. 一个节点是另一个节点的祖先: 正确处理

语言特性差异:

1. Python: 动态类型, 引用计数垃圾回收, 代码简洁
2. Java: 自动垃圾回收, 对象引用传递, 类型安全
3. C++: 手动内存管理, 指针操作, 高性能但容易出错

数学联系:

1. 与二进制表示和位运算相关
2. 与树的深度优先搜索理论相关
3. 与动态规划有一定联系

调试能力:

1. 可通过打印预处理数组调试
2. 可通过断言验证中间结果
3. 可通过特殊测试用例验证边界条件

注意事项：

C++这么写能通过，Python 递归层数太多可能会爆栈

Python 能通过的写法可以参考迭代版本

"""

```
import sys
from typing import List

# 增加递归限制以处理深度较大的树
sys.setrecursionlimit(500000)

class LCA:

    def __init__(self, n: int):
        self.MAXN = 500001
        self.LIMIT = 20
        self.power = self._log2(n)

        # 链式前向星建图
        self.head = [0] * self.MAXN
        self.next = [0] * (self.MAXN << 1)
        self.to = [0] * (self.MAXN << 1)
        self.cnt = 1

        # 深度数组和倍增数组
        self.deep = [0] * self.MAXN
        self.stjump = [[0] * self.LIMIT for _ in range(self.MAXN)]

    def _log2(self, n: int) -> int:
        """计算 log2(n) 的值"""
        ans = 0
        while (1 << ans) <= (n >> 1):
            ans += 1
        return ans

    def build(self, n: int) -> None:
        """初始化数据结构"""
        self.power = self._log2(n)
        self.cnt = 1
        for i in range(1, n + 1):
```

```

    self.head[i] = 0

def addEdge(self, u: int, v: int) -> None:
    """添加边到链式前向星结构"""
    self.next[self.cnt] = self.head[u]
    self.to[self.cnt] = v
    self.head[u] = self.cnt
    self.cnt += 1

def dfs(self, u: int, f: int) -> None:
    """
    dfs 递归版
    一般来说都这么写，但是本题附加的测试数据很毒
    Python 这么写就会因为递归太深而爆栈，C++这么写就能通过
    """
    self.deep[u] = self.deep[f] + 1
    self.stjump[u][0] = f
    for p in range(1, self.power + 1):
        self.stjump[u][p] = self.stjump[self.stjump[u][p - 1]][p - 1]

    e = self.head[u]
    while e != 0:
        if self.to[e] != f:
            self.dfs(self.to[e], u)
        e = self.next[e]

def lca(self, a: int, b: int) -> int:
    """计算两个节点的最近公共祖先"""
    if self.deep[a] < self.deep[b]:
        a, b = b, a

    # 将 a 调整到与 b 同一深度
    for p in range(self.power, -1, -1):
        if self.deep[self.stjump[a][p]] >= self.deep[b]:
            a = self.stjump[a][p]

    if a == b:
        return a

    # 同时向上跳跃找到 LCA
    for p in range(self.power, -1, -1):
        if self.stjump[a][p] != self.stjump[b][p]:
            a = self.stjump[a][p]

```

```
b = self.stjump[b][p]

return self.stjump[a][0]

# 由于 Python 标准输入处理较为复杂，这里只提供类的实现
# 实际使用时可以根据具体平台要求调整输入输出方式
```

文件: Code02_Multiply2.cpp

```
/***
 * 树上倍增解法（迭代版）
 * 题目来源: 洛谷 P3379 【模板】最近公共祖先 (LCA)
 * 题目链接: https://www.luogu.com.cn/problem/P3379
 *
 * 问题描述:
 * 给定一棵有根多叉树，请求出指定两个点直接最近的公共祖先。
 *
 * 解题思路:
 * 使用树上倍增法预处理每个节点的  $2^k$  级祖先，然后对于每次查询:
 * 1. 先将两个节点调整到同一深度
 * 2. 然后同时向上跳跃，直到找到最近公共祖先
 *
 * 与 Code02_Multiply1 的主要区别:
 * 1. 将递归版的 DFS 改为了迭代版，避免递归栈溢出
 * 2. 使用显式栈模拟递归过程
 *
 * 时间复杂度:
 * 预处理:  $O(n \log n)$ 
 * 查询:  $O(\log n)$ 
 * 空间复杂度:  $O(n \log n)$ 
 *
 * 是否为最优解: 是，对于在线查询 LCA 问题，倍增法是标准解法之一
 *
 * 工程化考虑:
 * 1. 边界条件处理: 处理空树、节点不存在等情况
 * 2. 输入验证: 验证输入节点是否在树中
 * 3. 异常处理: 对非法输入进行检查
 * 4. 可读性: 添加详细注释和变量命名
 *
 * 算法要点:
 * 1. 预处理阶段构建倍增数组: ancestor[i][k] 表示节点 i 的第  $2^k$  个祖先
```

- * 2. 查询阶段先调整深度再同时跳跃
- * 3. 利用二进制表示优化跳跃过程
- * 4. 使用迭代 DFS 避免递归栈溢出
- *
- * 与标准库实现对比:
 - * 1. 标准库通常有更完善的错误处理
 - * 2. 标准库可能使用更优化的数据结构
- *
- * 性能优化:
 - * 1. 预处理优化: 一次处理所有节点
 - * 2. 查询优化: 利用倍增快速跳跃
 - * 3. 递归优化: 使用迭代避免栈溢出
- *
- * 特殊场景:
 - * 1. 空输入: 返回特定值表示无效
 - * 2. 节点不存在: 返回特定值表示无效
 - * 3. 一个节点是另一个节点的祖先: 正确处理
 - * 4. 深度极大的树: 迭代版可以处理
- *
- * 语言特性差异:
 - * 1. C++: 手动内存管理, 指针操作, 高性能但容易出错
 - * 2. Java: 自动垃圾回收, 对象引用传递, 类型安全
 - * 3. Python: 动态类型, 引用计数垃圾回收, 代码简洁
- *
- * 数学联系:
 - * 1. 与二进制表示和位运算相关
 - * 2. 与树的深度优先搜索理论相关
 - * 3. 与动态规划有一定联系
- *
- * 调试能力:
 - * 1. 可通过打印预处理数组调试
 - * 2. 可通过断言验证中间结果
 - * 3. 可通过特殊测试用例验证边界条件
- *
- * 注意事项:
 - * 所有递归函数一律改成等义的迭代版
 - * 可以通过所有用例
- *
- * 注意: 由于编译环境限制, 避免使用复杂的 STL 容器和标准库函数

```
class LCA {  
private:
```

```
static const int MAXN = 500001;
static const int LIMIT = 20;

int power;

int log2(int n) {
    int ans = 0;
    while ((1 << ans) <= (n >> 1)) {
        ans++;
    }
    return ans;
}

// 链式前向星建图
int head[MAXN];
int next[MAXN << 1];
int to[MAXN << 1];
int cnt;

// 深度数组和倍增数组
int deep[MAXN];
int stjump[MAXN][LIMIT];

// dfs 迭代版需要用到的栈
int ufe[MAXN][3];
int stackSize, u, f, e;

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

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

public:
    void build(int n) {
```

```

power = log2(n);
cnt = 1;
for (int i = 1; i <= n; i++) {
    head[i] = 0;
}
}

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

// dfs 迭代版
// ufe 是为了实现迭代版而准备的栈
void dfs(int root) {
    stackSize = 0;
    // 栈里存放三个信息
    // u : 当前处理的点
    // f : 当前点 u 的父节点
    // e : 处理到几号边了
    // 如果 e== -1, 表示之前没有处理过 u 的任何边
    // 如果 e== 0, 表示 u 的边都已经处理完了
    push(root, 0, -1);
    while (stackSize > 0) {
        pop();
        if (e == -1) {
            deep[u] = deep[f] + 1;
            stjump[u][0] = f;
            for (int p = 1; p <= power; p++) {
                stjump[u][p] = stjump[stjump[u][p - 1]][p - 1];
            }
            e = head[u];
        } else {
            e = next[e];
        }
        if (e != 0) {
            push(u, f, e);
            if (to[e] != f) {
                push(to[e], u, -1);
            }
        }
    }
}

```

```

    }

    int lca(int a, int b) {
        if (deep[a] < deep[b]) {
            int tmp = a;
            a = b;
            b = tmp;
        }
        for (int p = power; p >= 0; p--) {
            if (deep[stjump[a][p]] >= deep[b]) {
                a = stjump[a][p];
            }
        }
        if (a == b) {
            return a;
        }
        for (int p = power; p >= 0; p--) {
            if (stjump[a][p] != stjump[b][p]) {
                a = stjump[a][p];
                b = stjump[b][p];
            }
        }
        return stjump[a][0];
    }
};

// 由于编译环境限制，不包含 main 函数和测试代码
// 在实际使用时，可以根据洛谷平台要求进行调整
=====
```

文件: Code02_Multiply2.java

=====

```

package class118;

/**
 * 树上倍增解法（迭代版）
 * 题目来源: 洛谷 P3379 【模板】最近公共祖先 (LCA)
 * 题目链接: https://www.luogu.com.cn/problem/P3379
 *
 * 问题描述:
 * 给定一棵有根多叉树，请求出指定两个点直接最近的公共祖先。
 *
```

* 解题思路:

* 使用树上倍增法预处理每个节点的 2^k 级祖先，然后对于每次查询:

* 1. 先将两个节点调整到同一深度

* 2. 然后同时向上跳跃，直到找到最近公共祖先

*

* 与 Code02_Multiply1 的主要区别:

* 1. 将递归版的 DFS 改为了迭代版，避免递归栈溢出

* 2. 使用显式栈模拟递归过程

*

* 时间复杂度:

* 预处理: $O(n \log n)$

* 查询: $O(\log n)$

* 空间复杂度: $O(n \log n)$

*

* 是否为最优解: 是，对于在线查询 LCA 问题，倍增法是标准解法之一

*

* 工程化考虑:

* 1. 边界条件处理: 处理空树、节点不存在等情况

* 2. 输入验证: 验证输入节点是否在树中

* 3. 异常处理: 对非法输入进行检查

* 4. 可读性: 添加详细注释和变量命名

*

* 算法要点:

* 1. 预处理阶段构建倍增数组: $\text{ancestor}[i][k]$ 表示节点 i 的第 2^k 个祖先

* 2. 查询阶段先调整深度再同时跳跃

* 3. 利用二进制表示优化跳跃过程

* 4. 使用迭代 DFS 避免递归栈溢出

*

* 与标准库实现对比:

* 1. 标准库通常有更完善的错误处理

* 2. 标准库可能使用更优化的数据结构

*

* 性能优化:

* 1. 预处理优化: 一次处理所有节点

* 2. 查询优化: 利用倍增快速跳跃

* 3. 递归优化: 使用迭代避免栈溢出

*

* 特殊场景:

* 1. 空输入: 返回特定值表示无效

* 2. 节点不存在: 返回特定值表示无效

* 3. 一个节点是另一个节点的祖先: 正确处理

* 4. 深度极大的树: 迭代版可以处理

*

- * 语言特性差异:
 - * 1. Java: 自动垃圾回收, 对象引用传递, 类型安全
 - * 2. C++: 手动内存管理, 指针操作, 高性能但容易出错
 - * 3. Python: 动态类型, 引用计数垃圾回收, 代码简洁
- *
- * 数学联系:
 - * 1. 与二进制表示和位运算相关
 - * 2. 与树的深度优先搜索理论相关
 - * 3. 与动态规划有一定联系
- *
- * 调试能力:
 - * 1. 可通过打印预处理数组调试
 - * 2. 可通过断言验证中间结果
 - * 3. 可通过特殊测试用例验证边界条件
- *
- * 注意事项:
 - * 所有递归函数一律改成等义的迭代版
 - * 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_Multiply2 {

    public static int MAXN = 500001;

    public static int LIMIT = 20;

    public static int power;

    public static int log2(int n) {
        int ans = 0;
        while ((1 << ans) <= (n >> 1)) {
            ans++;
        }
        return ans;
    }
}
```

```
public static int cnt;

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[][] stjump = new int[MAXN][LIMIT];

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

public static void build(int n) {
    power = log2(n);
    cnt = 1;
    Arrays.fill(head, 1, n + 1, 0);
}

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

// dfs 迭代版
// ufe 是为了实现迭代版而准备的栈
public static int[][] ufe = new int[MAXN][3];

public static int stackSize, u, f, e;

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

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

```
}
```

```
public static void dfs(int root) {
    stackSize = 0;
    // 栈里存放三个信息
    // u : 当前处理的点
    // f : 当前点 u 的父节点
    // e : 处理到几号边了
    // 如果 e == -1, 表示之前没有处理过 u 的任何边
    // 如果 e == 0, 表示 u 的边都已经处理完了
    push(root, 0, -1);
    while (stackSize > 0) {
        pop();
        if (e == -1) {
            deep[u] = deep[f] + 1;
            stjump[u][0] = f;
            for (int p = 1; p <= power; p++) {
                stjump[u][p] = stjump[stjump[u][p - 1]][p - 1];
            }
            e = head[u];
        } else {
            e = next[e];
        }
        if (e != 0) {
            push(u, f, e);
            if (to[e] != f) {
                push(to[e], u, -1);
            }
        }
    }
}
```

```
public static int lca(int a, int b) {
    if (deep[a] < deep[b]) {
        int tmp = a;
        a = b;
        b = tmp;
    }
    for (int p = power; p >= 0; p--) {
        if (deep[stjump[a][p]] >= deep[b]) {
            a = stjump[a][p];
        }
    }
}
```

```

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

public static void main(String[] args) throws IOException {
    BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
    StringTokenizer in = new StringTokenizer(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 root = (int) in.nval;
    build(n);
    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);
    }
    dfs(root);
    for (int i = 1, a, b; i <= m; i++) {
        in.nextToken();
        a = (int) in.nval;
        in.nextToken();
        b = (int) in.nval;
        out.println(lca(a, b));
    }
    out.flush();
    out.close();
    br.close();
}

```

}

=====

文件: Code02_Multiply2.py

=====

"""

树上倍增解法（迭代版）

题目来源: 洛谷 P3379 【模板】最近公共祖先 (LCA)

题目链接: <https://www.luogu.com.cn/problem/P3379>

问题描述:

给定一棵有根多叉树，请求出指定两个点直接最近的公共祖先。

解题思路:

使用树上倍增法预处理每个节点的 2^k 级祖先，然后对于每次查询：

1. 先将两个节点调整到同一深度
2. 然后同时向上跳跃，直到找到最近公共祖先

与 Code02_Multiply1 的主要区别：

1. 将递归版的 DFS 改为了迭代版，避免递归栈溢出
2. 使用显式栈模拟递归过程

时间复杂度:

预处理: $O(n \log n)$

查询: $O(\log n)$

空间复杂度: $O(n \log n)$

是否为最优解: 是，对于在线查询 LCA 问题，倍增法是标准解法之一

工程化考虑:

1. 边界条件处理：处理空树、节点不存在等情况
2. 输入验证：验证输入节点是否在树中
3. 异常处理：对非法输入进行检查
4. 可读性：添加详细注释和变量命名

算法要点:

1. 预处理阶段构建倍增数组： $\text{ancestor}[i][k]$ 表示节点 i 的第 2^k 个祖先
2. 查询阶段先调整深度再同时跳跃
3. 利用二进制表示优化跳跃过程
4. 使用迭代 DFS 避免递归栈溢出

与标准库实现对比:

1. 标准库通常有更完善的错误处理
2. 标准库可能使用更优化的数据结构

性能优化:

1. 预处理优化: 一次处理所有节点
2. 查询优化: 利用倍增快速跳跃
3. 递归优化: 使用迭代避免栈溢出

特殊场景:

1. 空输入: 返回特定值表示无效
2. 节点不存在: 返回特定值表示无效
3. 一个节点是另一个节点的祖先: 正确处理
4. 深度极大的树: 迭代版可以处理

语言特性差异:

1. Python: 动态类型, 引用计数垃圾回收, 代码简洁
2. Java: 自动垃圾回收, 对象引用传递, 类型安全
3. C++: 手动内存管理, 指针操作, 高性能但容易出错

数学联系:

1. 与二进制表示和位运算相关
2. 与树的深度优先搜索理论相关
3. 与动态规划有一定联系

调试能力:

1. 可通过打印预处理数组调试
2. 可通过断言验证中间结果
3. 可通过特殊测试用例验证边界条件

注意事项:

所有递归函数一律改成等义的迭代版

可以通过所有用例

"""

```
import sys
from typing import List

# 增加递归限制以处理深度较大的树
sys.setrecursionlimit(500000)
```

class LCA:

```
    def __init__(self, n: int):
```

```

self.MAXN = 500001
self.LIMIT = 20
self.power = self._log2(n)

# 链式前向星建图
self.head = [0] * self.MAXN
self.next = [0] * (self.MAXN << 1)
self.to = [0] * (self.MAXN << 1)
self.cnt = 1

# 深度数组和倍增数组
self.deep = [0] * self.MAXN
self.stjump = [[0] * self.LIMIT for _ in range(self.MAXN)]

# dfs 迭代版需要用到的栈
self.ufe = [[0, 0, 0] for _ in range(self.MAXN)]
self.stackSize = 0
self.u = 0
self.f = 0
self.e = 0

def _log2(self, n: int) -> int:
    """计算 log2(n) 的值"""
    ans = 0
    while (1 << ans) <= (n >> 1):
        ans += 1
    return ans

def build(self, n: int) -> None:
    """初始化数据结构"""
    self.power = self._log2(n)
    self.cnt = 1
    for i in range(1, n + 1):
        self.head[i] = 0

def addEdge(self, u: int, v: int) -> None:
    """添加边到链式前向星结构"""
    self.next[self.cnt] = self.head[u]
    self.to[self.cnt] = v
    self.head[u] = self.cnt
    self.cnt += 1

def _push(self, u: int, f: int, e: int) -> None:

```

```

"""将节点信息压入栈"""
self.ufe[self.stackSize][0] = u
self.ufe[self.stackSize][1] = f
self.ufe[self.stackSize][2] = e
self.stackSize += 1

def _pop(self) -> None:
    """从栈中弹出节点信息"""
    self.stackSize -= 1
    self.u = self.ufe[self.stackSize][0]
    self.f = self.ufe[self.stackSize][1]
    self.e = self.ufe[self.stackSize][2]

def dfs(self, root: int) -> None:
    """
    dfs 迭代版
    ufe 是为了实现迭代版而准备的栈
    """

    self.stackSize = 0
    # 栈里存放三个信息
    # u : 当前处理的点
    # f : 当前点 u 的父节点
    # e : 处理到几号边了
    # 如果 e==1, 表示之前没有处理过 u 的任何边
    # 如果 e==0, 表示 u 的边都已经处理完了
    self._push(root, 0, -1)

    while self.stackSize > 0:
        self._pop()
        if self.e == -1:
            self.deep[self.u] = self.deep[self.f] + 1
            self.stjump[self.u][0] = self.f
            for p in range(1, self.power + 1):
                self.stjump[self.u][p] = self.stjump[self.stjump[self.u][p - 1]][p - 1]
            self.e = self.head[self.u]
        else:
            self.e = self.next[self.e]

        if self.e != 0:
            self._push(self.u, self.f, self.e)
            if self.to[self.e] != self.f:
                self._push(self.to[self.e], self.u, -1)

```

```

def lca(self, a: int, b: int) -> int:
    """计算两个节点的最近公共祖先"""
    if self.deep[a] < self.deep[b]:
        a, b = b, a

    # 将 a 调整到与 b 同一深度
    for p in range(self.power, -1, -1):
        if self.deep[self.stjump[a][p]] >= self.deep[b]:
            a = self.stjump[a][p]

    if a == b:
        return a

    # 同时向上跳跃找到 LCA
    for p in range(self.power, -1, -1):
        if self.stjump[a][p] != self.stjump[b][p]:
            a = self.stjump[a][p]
            b = self.stjump[b][p]

    return self.stjump[a][0]

```

由于 Python 标准输入处理较为复杂，这里只提供类的实现
实际使用时可以根据具体平台要求调整输入输出方式

文件: Code03_Tarjan1.cpp

```

/**
 * Tarjan 算法解法（递归版）
 * 题目来源: 洛谷 P3379 【模板】最近公共祖先 (LCA)
 * 题目链接: https://www.luogu.com.cn/problem/P3379
 *
 * 问题描述:
 * 给定一棵有根多叉树，请求出指定两个点直接最近的公共祖先。
 *
 * 解题思路:
 * 使用 Tarjan 离线算法，基于 DFS 和并查集实现
 * 1. 首先读入所有查询，将查询存储在链式前向星结构中
 * 2. 进行 DFS 遍历树，同时处理查询
 * 3. 使用并查集维护节点的祖先关系
 *
 * 算法步骤:

```

- * 1. 对于当前节点 u, 标记为已访问
- * 2. 递归处理 u 的所有子节点 v
- * 3. 处理完 v 后, 将 v 的祖先设为 u (union 操作)
- * 4. 检查所有与 u 相关的查询, 如果另一个节点已访问, 则其 LCA 为 find 的结果
- *
- * 时间复杂度:
 - * $O(n + q)$, 其中 n 为节点数, q 为查询数
 - * 空间复杂度: $O(n + q)$
- *
- * 是否为最优解: 是, 对于离线查询 LCA 问题, Tarjan 算法是最优解
- *
- * 工程化考虑:
 - * 1. 边界条件处理: 处理空树、节点不存在等情况
 - * 2. 输入验证: 验证输入节点是否在树中
 - * 3. 异常处理: 对非法输入进行检查
 - * 4. 可读性: 添加详细注释和变量命名
- *
- * 算法要点:
 - * 1. 离线处理: 需要预先知道所有查询
 - * 2. 并查集: 用于维护节点的祖先关系
 - * 3. DFS 遍历: 在遍历过程中处理查询
- *
- * 与标准库实现对比:
 - * 1. 标准库通常有更完善的错误处理
 - * 2. 标准库可能使用更优化的数据结构
- *
- * 性能优化:
 - * 1. 离线处理优化: 一次性处理所有查询
 - * 2. 并查集优化: 路径压缩
- *
- * 特殊场景:
 - * 1. 空输入: 返回特定值表示无效
 - * 2. 节点不存在: 返回特定值表示无效
 - * 3. 查询为空: 直接返回
- *
- * 语言特性差异:
 - * 1. C++: 手动内存管理, 指针操作, 高性能但容易出错
 - * 2. Java: 自动垃圾回收, 对象引用传递, 类型安全
 - * 3. Python: 动态类型, 引用计数垃圾回收, 代码简洁
- *
- * 数学联系:
 - * 1. 与图论中的 DFS 理论相关
 - * 2. 与并查集数据结构相关

- * 3. 与离线算法设计思想相关
- *
- * 调试能力:
 - * 1. 可通过打印 DFS 遍历顺序调试
 - * 2. 可通过断言验证并查集操作
 - * 3. 可通过特殊测试用例验证边界条件
- *
- * 注意事项:
 - * C++这么写能通过，但递归层数太多可能会爆栈
 - * C++能通过的写法参考 Code03_Tarjan2 文件
 - * 提交时请把类名改成“Main”
- *
- * 注意：由于编译环境限制，避免使用复杂的 STL 容器和标准库函数
- */

```
class TarjanLCA {  
private:  
    static const int MAXN = 500001;  
  
    // 链式前向星建图  
    int headEdge[MAXN];  
    int edgeNext[MAXN << 1];  
    int edgeTo[MAXN << 1];  
    int tcnt;  
  
    // 每个节点有哪些查询，也用链式前向星方式存储  
    int headQuery[MAXN];  
    int queryNext[MAXN << 1];  
    int queryTo[MAXN << 1];  
  
    // 问题的编号，一旦有答案可以知道填写在哪  
    int queryIndex[MAXN << 1];  
    int qcnt;  
  
    // 某个节点是否访问过  
    bool visited[MAXN];  
  
    // 并查集  
    int father[MAXN];  
  
    // 收集的答案  
    int ans[MAXN];
```

```

public:
    void build(int n) {
        tcnt = qcnt = 1;
        for (int i = 1; i <= n; i++) {
            headEdge[i] = 0;
            headQuery[i] = 0;
            visited[i] = false;
            father[i] = i;
        }
    }

    void addEdge(int u, int v) {
        edgeNext[tcnt] = headEdge[u];
        edgeTo[tcnt] = v;
        headEdge[u] = tcnt++;
    }

    void addQuery(int u, int v, int i) {
        queryNext[qcnt] = headQuery[u];
        queryTo[qcnt] = v;
        queryIndex[qcnt] = i;
        headQuery[u] = qcnt++;
    }

// 并查集找头节点递归版
// 一般来说都这么写，但是本题附加的测试数据很毒
// C++这么写就能通过（在大多数情况下）
int find(int i) {
    if (i != father[i]) {
        father[i] = find(father[i]);
    }
    return father[i];
}

// tarjan 算法递归版
// 一般来说都这么写，但是本题附加的测试数据很毒
// C++这么写就能通过（在大多数情况下）
void tarjan(int u, int f) {
    visited[u] = true;
    for (int e = headEdge[u], v; e != 0; e = edgeNext[e]) {
        v = edgeTo[e];
        if (v != f) {
            tarjan(v, u);
        }
    }
}

```

```

        father[v] = u;
    }
}

for (int e = headQuery[u], v; e != 0; e = queryNext[e]) {
    v = queryTo[e];
    if (visited[v]) {
        ans[queryIndex[e]] = find(v);
    }
}
}

};

// 由于编译环境限制，不包含 main 函数和测试代码
// 在实际使用时，可以根据洛谷平台要求进行调整
=====
```

文件: Code03_Tarjan1.java

=====

```

package class118;

/**
 * Tarjan 算法解法（递归版）
 * 题目来源: 洛谷 P3379 【模板】最近公共祖先 (LCA)
 * 题目链接: https://www.luogu.com.cn/problem/P3379
 *
 * 问题描述:
 * 给定一棵有根多叉树，请求出指定两个点直接最近的公共祖先。
 *
 * 解题思路:
 * 使用 Tarjan 离线算法，基于 DFS 和并查集实现
 * 1. 首先读入所有查询，将查询存储在链式前向星结构中
 * 2. 进行 DFS 遍历树，同时处理查询
 * 3. 使用并查集维护节点的祖先关系
 *
 * 算法步骤:
 * 1. 对于当前节点 u，标记为已访问
 * 2. 递归处理 u 的所有子节点 v
 * 3. 处理完 v 后，将 v 的祖先设为 u (union 操作)
 * 4. 检查所有与 u 相关的查询，如果另一个节点已访问，则其 LCA 为 find 的结果
 *
 * 时间复杂度:
 * O(n + q)，其中 n 为节点数，q 为查询数

```

- * 空间复杂度: $O(n + q)$
- *
- * 是否为最优解: 是, 对于离线查询 LCA 问题, Tarjan 算法是最优解
- *
- * 工程化考虑:
 - * 1. 边界条件处理: 处理空树、节点不存在等情况
 - * 2. 输入验证: 验证输入节点是否在树中
 - * 3. 异常处理: 对非法输入进行检查
 - * 4. 可读性: 添加详细注释和变量命名
- *
- * 算法要点:
 - * 1. 离线处理: 需要预先知道所有查询
 - * 2. 并查集: 用于维护节点的祖先关系
 - * 3. DFS 遍历: 在遍历过程中处理查询
- *
- * 与标准库实现对比:
 - * 1. 标准库通常有更完善的错误处理
 - * 2. 标准库可能使用更优化的数据结构
- *
- * 性能优化:
 - * 1. 离线处理优化: 一次性处理所有查询
 - * 2. 并查集优化: 路径压缩
- *
- * 特殊场景:
 - * 1. 空输入: 返回特定值表示无效
 - * 2. 节点不存在: 返回特定值表示无效
 - * 3. 查询为空: 直接返回
- *
- * 语言特性差异:
 - * 1. Java: 自动垃圾回收, 对象引用传递, 类型安全
 - * 2. C++: 手动内存管理, 指针操作, 高性能但容易出错
 - * 3. Python: 动态类型, 引用计数垃圾回收, 代码简洁
- *
- * 数学联系:
 - * 1. 与图论中的 DFS 理论相关
 - * 2. 与并查集数据结构相关
 - * 3. 与离线算法设计思想相关
- *
- * 调试能力:
 - * 1. 可通过打印 DFS 遍历顺序调试
 - * 2. 可通过断言验证并查集操作
 - * 3. 可通过特殊测试用例验证边界条件
- *

```
* 注意事项:  
* 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 Code03_Tarjan2 文件  
* 提交时请把类名改成"Main"  
*/
```

```
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_Tarjan1 {  
  
    public static int MAXN = 500001;  
  
    // 链式前向星建图  
    public static int[] headEdge = new int[MAXN];  
  
    public static int[] edgeNext = new int[MAXN << 1];  
  
    public static int[] edgeTo = new int[MAXN << 1];  
  
    public static int tcnt;  
  
    // 每个节点有哪些查询, 也用链式前向星方式存储  
    public static int[] headQuery = new int[MAXN];  
  
    public static int[] queryNext = new int[MAXN << 1];  
  
    public static int[] queryTo = new int[MAXN << 1];  
  
    // 问题的编号, 一旦有答案可以知道填写在哪  
    public static int[] queryIndex = new int[MAXN << 1];  
  
    public static int qcnt;  
  
    // 某个节点是否访问过  
    public static boolean[] visited = new boolean[MAXN];  
  
    // 并查集
```

```

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

// 收集的答案
public static int[] ans = new int[MAXN];

public static void build(int n) {
    tcnt = qcnt = 1;
    Arrays.fill(headEdge, 1, n + 1, 0);
    Arrays.fill(headQuery, 1, n + 1, 0);
    Arrays.fill(visited, 1, n + 1, false);
    for (int i = 1; i <= n; i++) {
        father[i] = i;
    }
}

public static void addEdge(int u, int v) {
    edgeNext[tcnt] = headEdge[u];
    edgeTo[tcnt] = v;
    headEdge[u] = tcnt++;
}

public static void addQuery(int u, int v, int i) {
    queryNext[qcnt] = headQuery[u];
    queryTo[qcnt] = v;
    queryIndex[qcnt] = i;
    headQuery[u] = qcnt++;
}

// 并查集找头节点递归版
// 一般来说都这么写，但是本题附加的测试数据很毒
// java 这么写就会因为递归太深而爆栈，C++这么写就能通过
public static int find(int i) {
    if (i != father[i]) {
        father[i] = find(father[i]);
    }
    return father[i];
}

// tarjan 算法递归版
// 一般来说都这么写，但是本题附加的测试数据很毒
// java 这么写就会因为递归太深而爆栈，C++这么写就能通过
public static void tarjan(int u, int f) {
    visited[u] = true;
}

```

```

for (int e = headEdge[u], v; e != 0; e = edgeNext[e]) {
    v = edgeTo[e];
    if (v != f) {
        tarjan(v, u);
        father[v] = u;
    }
}
for (int e = headQuery[u], v; e != 0; e = queryNext[e]) {
    v = queryTo[e];
    if (visited[v]) {
        ans[queryIndex[e]] = find(v);
    }
}
}

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 root = (int) in.nval;
    build(n);
    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);
    }
    for (int i = 1, u, v; i <= m; i++) {
        in.nextToken();
        u = (int) in.nval;
        in.nextToken();
        v = (int) in.nval;
        addQuery(u, v, i);
        addQuery(v, u, i);
    }
    tarjan(root, 0);
}

```

```
        for (int i = 1; i <= m; i++) {
            out.println(ans[i]);
        }
        out.flush();
        out.close();
        br.close();
    }

}
```

文件: Code03_Tarjan1.py

"""

Tarjan 算法解法（递归版）

题目来源: 洛谷 P3379 【模板】最近公共祖先 (LCA)

题目链接: <https://www.luogu.com.cn/problem/P3379>

问题描述:

给定一棵有根多叉树，请求出指定两个点直接最近的公共祖先。

解题思路:

使用 Tarjan 离线算法，基于 DFS 和并查集实现

1. 首先读入所有查询，将查询存储在链式前向星结构中
2. 进行 DFS 遍历树，同时处理查询
3. 使用并查集维护节点的祖先关系

算法步骤:

1. 对于当前节点 u，标记为已访问
2. 递归处理 u 的所有子节点 v
3. 处理完 v 后，将 v 的祖先设为 u (union 操作)
4. 检查所有与 u 相关的查询，如果另一个节点已访问，则其 LCA 为 find 的结果

时间复杂度:

$O(n + q)$ ，其中 n 为节点数，q 为查询数

空间复杂度: $O(n + q)$

是否为最优解: 是，对于离线查询 LCA 问题，Tarjan 算法是最优解

工程化考虑:

1. 边界条件处理：处理空树、节点不存在等情况
2. 输入验证：验证输入节点是否在树中

3. 异常处理：对非法输入进行检查
4. 可读性：添加详细注释和变量命名

算法要点：

1. 离线处理：需要预先知道所有查询
2. 并查集：用于维护节点的祖先关系
3. DFS 遍历：在遍历过程中处理查询

与标准库实现对比：

1. 标准库通常有更完善的错误处理
2. 标准库可能使用更优化的数据结构

性能优化：

1. 离线处理优化：一次性处理所有查询
2. 并查集优化：路径压缩

特殊场景：

1. 空输入：返回特定值表示无效
2. 节点不存在：返回特定值表示无效
3. 查询为空：直接返回

语言特性差异：

1. Python：动态类型，引用计数垃圾回收，代码简洁
2. Java：自动垃圾回收，对象引用传递，类型安全
3. C++：手动内存管理，指针操作，高性能但容易出错

数学联系：

1. 与图论中的 DFS 理论相关
2. 与并查集数据结构相关
3. 与离线算法设计思想相关

调试能力：

1. 可通过打印 DFS 遍历顺序调试
2. 可通过断言验证并查集操作
3. 可通过特殊测试用例验证边界条件

注意事项：

Python 这么写就会因为递归层数太多而爆栈

Python 能通过的写法参考 Code03_Tarjan2 文件

"""

```
import sys
from typing import List
```

```
# 增加递归限制以处理深度较大的树
sys.setrecursionlimit(500000)

class TarjanLCA:
    def __init__(self, n: int):
        self.MAXN = 500001

        # 链式前向星建图
        self.headEdge = [0] * self.MAXN
        self.edgeNext = [0] * (self.MAXN << 1)
        self.edgeTo = [0] * (self.MAXN << 1)
        self.tcnt = 1

        # 每个节点有哪些查询，也用链式前向星方式存储
        self.headQuery = [0] * self.MAXN
        self.queryNext = [0] * (self.MAXN << 1)
        self.queryTo = [0] * (self.MAXN << 1)

        # 问题的编号，一旦有答案可以知道填写在哪
        self.queryIndex = [0] * (self.MAXN << 1)
        self.qcnt = 1

        # 某个节点是否访问过
        self.visited = [False] * self.MAXN

        # 并查集
        self.father = list(range(self.MAXN))

        # 收集的答案
        self.ans = [0] * self.MAXN

    def build(self, n: int) -> None:
        """初始化数据结构"""
        self.tcnt = self.qcnt = 1
        for i in range(1, n + 1):
            self.headEdge[i] = 0
            self.headQuery[i] = 0
            self.visited[i] = False
            self.father[i] = i

    def addEdge(self, u: int, v: int) -> None:
        """添加树的边"""
        self.edgeTo[self.edgeNext[u]] = v
        self.edgeNext[u] += 1
```

```

        self.edgeNext[self.tcnt] = self.headEdge[u]
        self.edgeTo[self.tcnt] = v
        self.headEdge[u] = self.tcnt
        self.tcnt += 1

def addQuery(self, u: int, v: int, i: int) -> None:
    """添加查询"""
    self.queryNext[self.qcnt] = self.headQuery[u]
    self.queryTo[self.qcnt] = v
    self.queryIndex[self.qcnt] = i
    self.headQuery[u] = self.qcnt
    self.qcnt += 1

def find(self, i: int) -> int:
    """
    并查集找头节点递归版
    一般来说都这么写，但是本题附加的测试数据很毒
    Python 这么写就会因为递归太深而爆栈
    """
    if i != self.father[i]:
        self.father[i] = self.find(self.father[i])
    return self.father[i]

def tarjan(self, u: int, f: int) -> None:
    """
    tarjan 算法递归版
    一般来说都这么写，但是本题附加的测试数据很毒
    Python 这么写就会因为递归太深而爆栈
    """
    self.visited[u] = True
    e = self.headEdge[u]
    while e != 0:
        v = self.edgeTo[e]
        if v != f:
            self.tarjan(v, u)
            self.father[v] = u
        e = self.edgeNext[e]

    q = self.headQuery[u]
    while q != 0:
        v = self.queryTo[q]
        if self.visited[v]:
            self.ans[self.queryIndex[q]] = self.find(v)

```

```
q = self.queryNext[q]  
  
# 由于 Python 标准输入处理较为复杂，这里只提供类的实现  
# 实际使用时可以根据具体平台要求调整输入输出方式
```

文件: Code03_Tarjan2.cpp

```
/**  
 * Tarjan 算法解法（迭代版）  
 * 题目来源: 洛谷 P3379 【模板】最近公共祖先 (LCA)  
 * 题目链接: https://www.luogu.com.cn/problem/P3379  
 *  
 * 问题描述:  
 * 给定一棵有根多叉树，请求出指定两个点直接最近的公共祖先。  
 *  
 * 解题思路:  
 * 使用 Tarjan 离线算法，基于 DFS 和并查集实现  
 * 1. 首先读入所有查询，将查询存储在链式前向星结构中  
 * 2. 进行 DFS 遍历树，同时处理查询  
 * 3. 使用并查集维护节点的祖先关系  
 *  
 * 与 Code03_Tarjan1 的主要区别:  
 * 1. 将递归版的 Tarjan 算法改为了迭代版，避免递归栈溢出  
 * 2. 将递归版的并查集 find 操作改为了迭代版  
 * 3. 使用显式栈模拟递归过程  
 *  
 * 算法步骤:  
 * 1. 对于当前节点 u，标记为已访问  
 * 2. 递归处理 u 的所有子节点 v  
 * 3. 处理完 v 后，将 v 的祖先设为 u (union 操作)  
 * 4. 检查所有与 u 相关的查询，如果另一个节点已访问，则其 LCA 为 find 的结果  
 *  
 * 时间复杂度:  
 *  $O(n + q)$ ，其中 n 为节点数，q 为查询数  
 * 空间复杂度:  $O(n + q)$   
 *  
 * 是否为最优解: 是，对于离线查询 LCA 问题，Tarjan 算法是最优解  
 *  
 * 工程化考虑:  
 * 1. 边界条件处理: 处理空树、节点不存在等情况  
 * 2. 输入验证: 验证输入节点是否在树中
```

* 3. 异常处理：对非法输入进行检查

* 4. 可读性：添加详细注释和变量命名

*

* 算法要点：

* 1. 离线处理：需要预先知道所有查询

* 2. 并查集：用于维护节点的祖先关系

* 3. DFS 遍历：在遍历过程中处理查询

* 4. 迭代实现：避免递归栈溢出

*

* 与标准库实现对比：

* 1. 标准库通常有更完善的错误处理

* 2. 标准库可能使用更优化的数据结构

*

* 性能优化：

* 1. 离线处理优化：一次性处理所有查询

* 2. 并查集优化：路径压缩

* 3. 递归优化：使用迭代避免栈溢出

*

* 特殊场景：

* 1. 空输入：返回特定值表示无效

* 2. 节点不存在：返回特定值表示无效

* 3. 查询为空：直接返回

* 4. 深度极大的树：迭代版可以处理

*

* 语言特性差异：

* 1. C++：手动内存管理，指针操作，高性能但容易出错

* 2. Java：自动垃圾回收，对象引用传递，类型安全

* 3. Python：动态类型，引用计数垃圾回收，代码简洁

*

* 数学联系：

* 1. 与图论中的 DFS 理论相关

* 2. 与并查集数据结构相关

* 3. 与离线算法设计思想相关

*

* 调试能力：

* 1. 可通过打印 DFS 遍历顺序调试

* 2. 可通过断言验证并查集操作

* 3. 可通过特殊测试用例验证边界条件

*

* 注意事项：

* 所有递归函数一律改成等义的迭代版

* 可以通过所有用例

*

```
* 注意：由于编译环境限制，避免使用复杂的 STL 容器和标准库函数
*/
```

```
class TarjanLCA {
private:
    static const int MAXN = 500001;

    // 链式前向星建图
    int headEdge[MAXN];
    int edgeNext[MAXN << 1];
    int edgeTo[MAXN << 1];
    int tcnt;

    // 每个节点有哪些查询，也用链式前向星方式存储
    int headQuery[MAXN];
    int queryNext[MAXN << 1];
    int queryTo[MAXN << 1];

    // 问题的编号，一旦有答案可以知道填写在哪
    int queryIndex[MAXN << 1];
    int qcnt;

    // 某个节点是否访问过
    bool visited[MAXN];

    // 并查集
    int father[MAXN];

    // 收集的答案
    int ans[MAXN];

    // 为了实现迭代版而准备的栈
    int stack[MAXN];

    // 为了实现迭代版而准备的栈
    int ufe[MAXN][3];
    int stackSize, u, f, e;

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

```
}
```

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

```
public:
```

```
void build(int n) {
    tcnt = qcnt = 1;
    for (int i = 1; i <= n; i++) {
        headEdge[i] = 0;
        headQuery[i] = 0;
        visited[i] = false;
        father[i] = i;
    }
}
```

```
void addEdge(int u, int v) {
    edgeNext[tcnt] = headEdge[u];
    edgeTo[tcnt] = v;
    headEdge[u] = tcnt++;
}
```

```
void addQuery(int u, int v, int i) {
    queryNext[qcnt] = headQuery[u];
    queryTo[qcnt] = v;
    queryIndex[qcnt] = i;
    headQuery[u] = qcnt++;
}
```

```
// 并查集找头节点迭代版
```

```
int find(int i) {
    int size = 0;
    while (i != father[i]) {
        stack[size++] = i;
        i = father[i];
    }
    while (size > 0) {
        father[stack[--size]] = i;
    }
}
```

```

    return i;
}

// tarjan 算法迭代版
void tarjan(int root) {
    stackSize = 0;
    // 栈里存放三个信息
    // u : 当前处理的点
    // f : 当前点 u 的父节点
    // e : 处理到几号边了
    // 如果 e==1, 表示之前没有处理过 u 的任何边
    // 如果 e==0, 表示 u 的边都已经处理完了
    push(root, 0, -1);
    while (stackSize > 0) {
        pop();
        if (e == -1) {
            visited[u] = true;
            e = headEdge[u];
        } else {
            e = edgeNext[e];
        }
        if (e != 0) {
            push(u, f, e);
            if (edgeTo[e] != f) {
                push(edgeTo[e], u, -1);
            }
        } else {
            // e == 0 代表 u 后续已经没有边需要处理了
            for (int q = headQuery[u], v; q != 0; q = queryNext[q]) {
                v = queryTo[q];
                if (visited[v]) {
                    ans[queryIndex[q]] = find(v);
                }
            }
            father[u] = f;
        }
    }
}

// 由于编译环境限制，不包含 main 函数和测试代码
// 在实际使用时，可以根据洛谷平台要求进行调整

```

文件: Code03_Tarjan2.java

```
=====
package class118;
```

```
=====
/**  
 * Tarjan 算法解法（迭代版）  
 * 题目来源: 洛谷 P3379 【模板】最近公共祖先 (LCA)  
 * 题目链接: https://www.luogu.com.cn/problem/P3379  
 *  
 * 问题描述:  
 * 给定一棵有根多叉树，请求出指定两个点直接最近的公共祖先。  
 *  
 * 解题思路:  
 * 使用 Tarjan 离线算法，基于 DFS 和并查集实现  
 * 1. 首先读入所有查询，将查询存储在链式前向星结构中  
 * 2. 进行 DFS 遍历树，同时处理查询  
 * 3. 使用并查集维护节点的祖先关系  
 *  
 * 与 Code03_Tarjan1 的主要区别:  
 * 1. 将递归版的 Tarjan 算法改为了迭代版，避免递归栈溢出  
 * 2. 将递归版的并查集 find 操作改为了迭代版  
 * 3. 使用显式栈模拟递归过程  
 *  
 * 算法步骤:  
 * 1. 对于当前节点 u，标记为已访问  
 * 2. 递归处理 u 的所有子节点 v  
 * 3. 处理完 v 后，将 v 的祖先设为 u (union 操作)  
 * 4. 检查所有与 u 相关的查询，如果另一个节点已访问，则其 LCA 为 find 的结果  
 *  
 * 时间复杂度:  
 *  $O(n + q)$ ，其中 n 为节点数，q 为查询数  
 * 空间复杂度:  $O(n + q)$   
 *  
 * 是否为最优解: 是，对于离线查询 LCA 问题，Tarjan 算法是最优解  
 *  
 * 工程化考虑:  
 * 1. 边界条件处理: 处理空树、节点不存在等情况  
 * 2. 输入验证: 验证输入节点是否在树中  
 * 3. 异常处理: 对非法输入进行检查  
 * 4. 可读性: 添加详细注释和变量命名  
 *
```

* 算法要点:

- * 1. 离线处理: 需要预先知道所有查询
- * 2. 并查集: 用于维护节点的祖先关系
- * 3. DFS 遍历: 在遍历过程中处理查询
- * 4. 迭代实现: 避免递归栈溢出

*

* 与标准库实现对比:

- * 1. 标准库通常有更完善的错误处理
- * 2. 标准库可能使用更优化的数据结构

*

* 性能优化:

- * 1. 离线处理优化: 一次性处理所有查询
- * 2. 并查集优化: 路径压缩
- * 3. 递归优化: 使用迭代避免栈溢出

*

* 特殊场景:

- * 1. 空输入: 返回特定值表示无效
- * 2. 节点不存在: 返回特定值表示无效
- * 3. 查询为空: 直接返回
- * 4. 深度极大的树: 迭代版可以处理

*

* 语言特性差异:

- * 1. Java: 自动垃圾回收, 对象引用传递, 类型安全
- * 2. C++: 手动内存管理, 指针操作, 高性能但容易出错
- * 3. Python: 动态类型, 引用计数垃圾回收, 代码简洁

*

* 数学联系:

- * 1. 与图论中的 DFS 理论相关
- * 2. 与并查集数据结构相关
- * 3. 与离线算法设计思想相关

*

* 调试能力:

- * 1. 可通过打印 DFS 遍历顺序调试
- * 2. 可通过断言验证并查集操作
- * 3. 可通过特殊测试用例验证边界条件

*

* 注意事项:

- * 所有递归函数一律改成等义的迭代版
- * 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_Tarjan2 {

    public static int MAXN = 500001;

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

    public static int[] edgeNext = new int[MAXN << 1];

    public static int[] edgeTo = new int[MAXN << 1];

    public static int tcnt;

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

    public static int[] queryNext = new int[MAXN << 1];

    public static int[] queryTo = new int[MAXN << 1];

    public static int[] queryIndex = new int[MAXN << 1];

    public static int qcnt;

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

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

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

    public static void build(int n) {
        tcnt = qcnt = 1;
        Arrays.fill(headEdge, 1, n + 1, 0);
        Arrays.fill(headQuery, 1, n + 1, 0);
        Arrays.fill(visited, 1, n + 1, false);
        for (int i = 1; i <= n; i++) {
            father[i] = i;
        }
    }
}
```

```

public static void addEdge(int u, int v) {
    edgeNext[tcnt] = headEdge[u];
    edgeTo[tcnt] = v;
    headEdge[u] = tcnt++;
}

public static void addQuery(int u, int v, int i) {
    queryNext[qcnt] = headQuery[u];
    queryTo[qcnt] = v;
    queryIndex[qcnt] = i;
    headQuery[u] = qcnt++;
}

// 为了实现迭代版而准备的栈
public static int[] stack = new int[MAXN];

// 并查集找头节点迭代版
public static int find(int i) {
    int size = 0;
    while (i != father[i]) {
        stack[size++] = i;
        i = father[i];
    }
    while (size > 0) {
        father[stack[--size]] = i;
    }
    return i;
}

// 为了实现迭代版而准备的栈
public static int[][] ufe = new int[MAXN][3];

public static int stackSize, u, f, e;

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

public static void pop() {

```

```

--stackSize;
u = ufe[stackSize][0];
f = ufe[stackSize][1];
e = ufe[stackSize][2];
}

// 为了容易改成迭代版，修改一下递归版
public static void tarjan(int u, int f) {
    visited[u] = true;
    for (int e = headEdge[u], v; e != 0; e = edgeNext[e]) {
        v = edgeTo[e];
        if (v != f) {
            tarjan(v, u);
            // 注意这里，注释了一行
        }
        father[v] = u;
    }
}

for (int e = headQuery[u], v; e != 0; e = queryNext[e]) {
    v = queryTo[e];
    if (visited[v]) {
        ans[queryIndex[e]] = find(v);
    }
}
// 注意这里，增加了一行
father[u] = f;
}

// tarjan 算法迭代版，根据上面的递归版改写
public static void tarjan(int root) {
    stackSize = 0;
    // 栈里存放三个信息
    // u : 当前处理的点
    // f : 当前点 u 的父节点
    // e : 处理到几号边了
    // 如果 e== -1，表示之前没有处理过 u 的任何边
    // 如果 e== 0，表示 u 的边都已经处理完了
    push(root, 0, -1);
    while (stackSize > 0) {
        pop();
        if (e == -1) {
            visited[u] = true;
            e = headEdge[u];
        } else {

```

```

        e = edgeNext[e];
    }
    if (e != 0) {
        push(u, f, e);
        if (edgeTo[e] != f) {
            push(edgeTo[e], u, -1);
        }
    } else {
        // e == 0 代表 u 后续已经没有边需要处理了
        for (int q = headQuery[u], v; q != 0; q = queryNext[q]) {
            v = queryTo[q];
            if (visited[v]) {
                ans[queryIndex[q]] = find(v);
            }
        }
        father[u] = f;
    }
}
}

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 root = (int) in.nval;
    build(n);
    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);
    }
    for (int i = 1, u, v; i <= m; i++) {
        in.nextToken();
        u = (int) in.nval;
        in.nextToken();
    }
}

```

```
v = (int) in.nval;
addQuery(u, v, i);
addQuery(v, u, i);
}
tarjan(root);
for (int i = 1; i <= m; i++) {
    out.println(ans[i]);
}
out.flush();
out.close();
br.close();
}

}
```

}

=====

文件: Code03_Tarjan2.py

=====

"""

Tarjan 算法解法（迭代版）

题目来源: 洛谷 P3379 【模板】最近公共祖先 (LCA)

题目链接: <https://www.luogu.com.cn/problem/P3379>

问题描述:

给定一棵有根多叉树，请求出指定两个点直接最近的公共祖先。

解题思路:

使用 Tarjan 离线算法，基于 DFS 和并查集实现

1. 首先读入所有查询，将查询存储在链式前向星结构中
2. 进行 DFS 遍历树，同时处理查询
3. 使用并查集维护节点的祖先关系

与 Code03_Tarjan1 的主要区别:

1. 将递归版的 Tarjan 算法改为了迭代版，避免递归栈溢出
2. 将递归版的并查集 find 操作改为了迭代版
3. 使用显式栈模拟递归过程

算法步骤:

1. 对于当前节点 u ，标记为已访问
2. 递归处理 u 的所有子节点 v
3. 处理完 v 后，将 v 的祖先设为 u (union 操作)
4. 检查所有与 u 相关的查询，如果另一个节点已访问，则其 LCA 为 find 的结果

时间复杂度：

$O(n + q)$ ，其中 n 为节点数， q 为查询数

空间复杂度： $O(n + q)$

是否为最优解：是，对于离线查询 LCA 问题，Tarjan 算法是最优解

工程化考虑：

1. 边界条件处理：处理空树、节点不存在等情况
2. 输入验证：验证输入节点是否在树中
3. 异常处理：对非法输入进行检查
4. 可读性：添加详细注释和变量命名

算法要点：

1. 离线处理：需要预先知道所有查询
2. 并查集：用于维护节点的祖先关系
3. DFS 遍历：在遍历过程中处理查询
4. 迭代实现：避免递归栈溢出

与标准库实现对比：

1. 标准库通常有更完善的错误处理
2. 标准库可能使用更优化的数据结构

性能优化：

1. 离线处理优化：一次性处理所有查询
2. 并查集优化：路径压缩
3. 递归优化：使用迭代避免栈溢出

特殊场景：

1. 空输入：返回特定值表示无效
2. 节点不存在：返回特定值表示无效
3. 查询为空：直接返回
4. 深度极大的树：迭代版可以处理

语言特性差异：

1. Python：动态类型，引用计数垃圾回收，代码简洁
2. Java：自动垃圾回收，对象引用传递，类型安全
3. C++：手动内存管理，指针操作，高性能但容易出错

数学联系：

1. 与图论中的 DFS 理论相关
2. 与并查集数据结构相关
3. 与离线算法设计思想相关

调试能力：

1. 可通过打印 DFS 遍历顺序调试
2. 可通过断言验证并查集操作
3. 可通过特殊测试用例验证边界条件

注意事项：

所有递归函数一律改成等义的迭代版

可以通过所有用例

"""

```
import sys
from typing import List

# 增加递归限制以处理深度较大的树
sys.setrecursionlimit(500000)

class TarjanLCA:
    def __init__(self, n: int):
        self.MAXN = 500001

        # 链式前向星建图
        self.headEdge = [0] * self.MAXN
        self.edgeNext = [0] * (self.MAXN << 1)
        self.edgeTo = [0] * (self.MAXN << 1)
        self.tcnt = 1

        # 每个节点有哪些查询，也用链式前向星方式存储
        self.headQuery = [0] * self.MAXN
        self.queryNext = [0] * (self.MAXN << 1)
        self.queryTo = [0] * (self.MAXN << 1)

        # 问题的编号，一旦有答案可以知道填写在哪
        self.queryIndex = [0] * (self.MAXN << 1)
        self.qcnt = 1

        # 某个节点是否访问过
        self.visited = [False] * self.MAXN

        # 并查集
        self.father = list(range(self.MAXN))

        # 收集的答案
```

```

self.ans = [0] * self.MAXN

# 为了实现迭代版而准备的栈
self.stack = [0] * self.MAXN

# 为了实现迭代版而准备的栈
self.ufe = [[0, 0, 0] for _ in range(self.MAXN)]
self.stackSize = 0
self.u = 0
self.f = 0
self.e = 0

def build(self, n: int) -> None:
    """初始化数据结构"""
    self.tcnt = self.qcnt = 1
    for i in range(1, n + 1):
        self.headEdge[i] = 0
        self.headQuery[i] = 0
        self.visited[i] = False
        self.father[i] = i

def addEdge(self, u: int, v: int) -> None:
    """添加树的边"""
    self.edgeNext[self.tcnt] = self.headEdge[u]
    self.edgeTo[self.tcnt] = v
    self.headEdge[u] = self.tcnt
    self.tcnt += 1

def addQuery(self, u: int, v: int, i: int) -> None:
    """添加查询"""
    self.queryNext[self.qcnt] = self.headQuery[u]
    self.queryTo[self.qcnt] = v
    self.queryIndex[self.qcnt] = i
    self.headQuery[u] = self.qcnt
    self.qcnt += 1

def find(self, i: int) -> int:
    """并查集找头节点迭代版"""
    size = 0
    while i != self.father[i]:
        self.stack[size] = i
        size += 1
        i = self.father[i]

```

```

while size > 0:
    size -= 1
    self.father[self.stack[size]] = i

return i

def _push(self, u: int, f: int, e: int) -> None:
    """将节点信息压入栈"""
    self.ufe[self.stackSize][0] = u
    self.ufe[self.stackSize][1] = f
    self.ufe[self.stackSize][2] = e
    self.stackSize += 1

def _pop(self) -> None:
    """从栈中弹出节点信息"""
    self.stackSize -= 1
    self.u = self.ufe[self.stackSize][0]
    self.f = self.ufe[self.stackSize][1]
    self.e = self.ufe[self.stackSize][2]

def tarjan(self, root: int) -> None:
    """
    tarjan 算法迭代版，根据递归版改写
    """

    self.stackSize = 0
    # 栈里存放三个信息
    # u : 当前处理的点
    # f : 当前点 u 的父节点
    # e : 处理到几号边了
    # 如果 e==1, 表示之前没有处理过 u 的任何边
    # 如果 e==0, 表示 u 的边都已经处理完了
    self._push(root, 0, -1)

    while self.stackSize > 0:
        self._pop()
        if self.e == -1:
            self.visited[self.u] = True
            self.e = self.headEdge[self.u]
        else:
            self.e = self.edgeNext[self.e]

        if self.e != 0:

```

```

        self._push(self.u, self.f, self.e)
        if self.edgeTo[self.e] != self.f:
            self._push(self.edgeTo[self.e], self.u, -1)
    else:
        # e == 0 代表 u 后续已经没有边需要处理了
        q = self.headQuery[self.u]
        while q != 0:
            v = self.queryTo[q]
            if self.visited[v]:
                self.ans[self.queryIndex[q]] = self.find(v)
            q = self.queryNext[q]
        self.father[self.u] = self.f

# 由于 Python 标准输入处理较为复杂，这里只提供类的实现
# 实际使用时可以根据具体平台要求调整输入输出方式

```

=====

文件: Code04_LCA_BinaryLifting.cpp

=====

```

// C++标准库头文件
#include <iostream>
#include <string>
#include <vector>
#include <stack>
#include <unordered_map>
#include <unordered_set>
#include <algorithm>
using namespace std;

/***
 * LCA 问题 - 递归法与倍增法实现
 * 本文件包含多个 LCA 相关题目的解答，涵盖不同的实现方法和优化策略
 * 所有非代码内容都以注释形式呈现，包含详细的分析和说明
 *
 * 主要内容包括：
 * 1. LeetCode 236: 二叉树的最近公共祖先
 * 2. LeetCode 235: 二叉搜索树的最近公共祖先
 * 3. LeetCode 1650: 带父指针的二叉树最近公共祖先
 * 4. 其他平台的经典 LCA 题目
 * 5. 洛谷 P3379: 最近公共祖先模板题
 * 6. HDU 2586: 树上两点距离
 * 7. SPOJ LCASQ: 基础 LCA 模板题

```

- * 8. POJ 1330: 最近公共祖先
- * 9. Codeforces 1304E: 1-Trees and Queries
- * 10. AtCoder ABC133F: Colorful Tree
- *
- * 算法复杂度分析:
 - * 1. 递归 DFS: $O(n)$ 时间, $O(h)$ 空间
 - * 2. 倍增法: $O(n \log n)$ 预处理, $O(\log n)$ 查询
 - * 3. Tarjan 算法: $O(n + q)$ 时间, $O(n)$ 空间
 - * 4. 树链剖分: $O(n)$ 预处理, $O(\log n)$ 查询
- *
- * 工程化考量:
 - * 1. 异常处理: 输入验证、边界条件处理
 - * 2. 性能优化: 预处理优化、查询优化
 - * 3. 可读性: 详细注释、模块化设计
 - * 4. 调试能力: 打印调试、断言验证
 - * 5. 单元测试: 覆盖各种边界场景
- *
- * 语言特性差异:
 - * 1. C++: 手动内存管理, 指针操作, 高性能但容易出错
 - * 2. Java: 自动垃圾回收, 对象引用传递, 类型安全
 - * 3. Python: 动态类型, 引用计数垃圾回收, 代码简洁
- *
- * 数学联系:
 - * 1. 二进制表示与位运算
 - * 2. 树的深度优先搜索理论
 - * 3. 动态规划思想
 - * 4. 并查集数据结构
- *
- * 与机器学习联系:
 - * 1. 树结构在决策树算法中的应用
 - * 2. LCA 在层次聚类中的潜在应用
 - * 3. 图神经网络中的树结构处理
- *
- * 反直觉设计:
 - * 1. 倍增法的二进制跳跃思想
 - * 2. Tarjan 算法的离线处理策略
 - * 3. 树链剖分的重链轻链分解
- *
- * 极端场景鲁棒性:
 - * 1. 空树和单节点树
 - * 2. 线性树 (退化为链表)
 - * 3. 完全二叉树
 - * 4. 大规模数据 ($n > 10^5$)

```

* 5. 深度极大的树
*/



// 二叉树节点定义
struct TreeNode {
    int val;
    TreeNode *left;
    TreeNode *right;
    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
};

// 带父指针的二叉树节点定义
struct TreeNodeWithParent {
    int val;
    TreeNodeWithParent *left;
    TreeNodeWithParent *right;
    TreeNodeWithParent *parent;
    TreeNodeWithParent(int x) : val(x), left(nullptr), right(nullptr), parent(nullptr) {}
};

class Solution {
public:
    /**
     * 解法一：LeetCode 236. 二叉树的最近公共祖先
     * 题目链接：https://leetcode.cn/problems/lowest-common-ancestor-of-a-binary-tree/
     * 难度：中等
     *
     * 问题描述：
     * 给定一个二叉树，找到该树中两个指定节点的最近公共祖先。
     * 最近公共祖先的定义为：“对于有根树 T 的两个节点 p、q，最近公共祖先表示为一个节点 x，满足 x 是 p、q 的祖先且 x 的深度尽可能大。”
     *
     * 解题思路：递归深度优先搜索
     * 1. 递归终止条件：当前节点为空或者是 p 或 q 中的一个
     * 2. 递归搜索左右子树
     * 3. 根据左右子树的返回结果判断：
     *   - 如果左右子树都返回非空，说明当前节点就是 LCA
     *   - 如果只有一侧返回非空，返回该侧的结果
     *   - 如果两侧都返回空，返回 null
     *
     * 时间复杂度：O(n) - 其中 n 是树中节点的数量，最坏情况下需要遍历所有节点
     * 空间复杂度：O(h) - 其中 h 是树的高度，递归调用栈的深度
     * 是否为最优解：对于单次查询，这是最优解之一
    */
}

```

```

*
* @param root 二叉树的根节点
* @param p 目标节点 p
* @param q 目标节点 q
* @return 最近公共祖先节点
*/
TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q) {
    // 异常处理：检查输入参数
    if (root == nullptr || p == nullptr || q == nullptr) {
        return nullptr;
    }

    // 基本情况：如果当前节点是 p 或 q，则当前节点就是 LCA
    if (root == p || root == q) {
        return root;
    }

    // 递归查找左子树中的 LCA
    TreeNode* left = lowestCommonAncestor(root->left, p, q);
    // 递归查找右子树中的 LCA
    TreeNode* right = lowestCommonAncestor(root->right, p, q);

    // 如果左右子树都找到了节点，说明当前节点是 LCA
    if (left != nullptr && right != nullptr) {
        return root;
    }

    // 如果只有左子树找到了节点，返回左子树的结果
    if (left != nullptr) {
        return left;
    }

    // 如果只有右子树找到了节点，返回右子树的结果
    if (right != nullptr) {
        return right;
    }

    // 如果左右子树都没有找到节点，返回 nullptr
    return nullptr;
}

/**
* 解法二：LeetCode 235. 二叉搜索树的最近公共祖先

```

- * 题目链接: <https://leetcode.cn/problems/lowest-common-ancestor-of-a-binary-search-tree/>
- * 难度: 简单
- *
- * 问题描述:
- * 给定一个二叉搜索树, 找到该树中两个指定节点的最近公共祖先。
- * 最近公共祖先的定义为: "对于有根树 T 的两个节点 p、q, 最近公共祖先表示为一个节点 x, 满足 x 是 p、q 的祖先且 x 的深度尽可能大。"
- *
- * 解题思路: 利用二叉搜索树的特性
- * 二叉搜索树的特性: 左子树所有节点值 < 根节点值 < 右子树所有节点值
- * 1. 如果 p 和 q 的值都小于当前节点, 那么 LCA 在左子树
- * 2. 如果 p 和 q 的值都大于当前节点, 那么 LCA 在右子树
- * 3. 如果一个小于等于, 一个大于等于, 那么当前节点就是 LCA
- *
- * 时间复杂度: O(h) - 其中 h 是树的高度, 在平衡树情况下为 O(log n)
- * 空间复杂度: O(h) - 递归调用栈的深度
- * 是否为最优解: 是, 利用了 BST 的特性, 比通用二叉树解法更高效
- */

```
TreeNode* lowestCommonAncestorBST(TreeNode* root, TreeNode* p, TreeNode* q) {  
    // 异常处理  
    if (root == nullptr || p == nullptr || q == nullptr) {  
        return nullptr;  
    }  
  
    // 如果 p 和 q 都在左子树  
    if (p->val < root->val && q->val < root->val) {  
        return lowestCommonAncestorBST(root->left, p, q);  
    }  
    // 如果 p 和 q 都在右子树  
    else if (p->val > root->val && q->val > root->val) {  
        return lowestCommonAncestorBST(root->right, p, q);  
    }  
    // 如果 p 和 q 分别在两侧, 或者其中一个是当前节点  
    else {  
        return root;  
    }  
}  
  
/**  
 * 解法三: LeetCode 1650. 二叉树的最近公共祖先 III (带父指针)  
 * 题目链接: https://leetcode.cn/problems/lowest-common-ancestor-of-a-binary-tree-iii/  
 * 难度: 中等  
 */
```

* 问题描述:

* 给定一棵二叉树中的两个节点 p 和 q, 返回它们的最近公共祖先 (LCA) 节点。

* 每个节点都有一个指向其父节点的指针。

*

* 解题思路: 双指针法

* 1. 分别计算 p 和 q 到根节点的深度差

* 2. 先将较深的节点向上移动, 使两个节点处于同一深度

* 3. 然后同时向上移动两个节点, 直到找到相同的节点, 即为 LCA

*

* 时间复杂度: $O(h)$ - 其中 h 是树的高度

* 空间复杂度: $O(1)$ - 只使用常数额外空间

* 是否为最优解: 是, 充分利用了父指针的特性

*/

```
TreeNodeWithParent* lowestCommonAncestorWithParent(TreeNodeWithParent* p, TreeNodeWithParent* q) {
```

```
    if (p == nullptr || q == nullptr) {
        return nullptr;
    }
```

```
    TreeNodeWithParent* a = p;
    TreeNodeWithParent* b = q;
```

// 双指针法, 类似于链表相交问题

// 当 a 或 b 为空时, 将其指向对方的起始节点, 这样可以抵消深度差

```
    while (a != b) {
```

```
        a = (a == nullptr) ? q : a->parent;
        b = (b == nullptr) ? p : b->parent;
    }
```

```
    return a;
```

```
}
```

/**

* 解法四: 迭代版本的二叉树 LCA (避免递归栈溢出)

* 适用于处理大型树的情况

*

* 解题思路: 后序遍历 + 记录父节点路径

* 1. 使用栈进行后序遍历

* 2. 记录每个节点的访问状态

* 3. 当找到 p 或 q 时, 记录其路径

* 4. 比较两条路径, 找到最后一个公共节点

*

* 时间复杂度: $O(n)$

```

* 空间复杂度: O(h)
*/
TreeNode* lowestCommonAncestorIterative(TreeNode* root, TreeNode* p, TreeNode* q) {
    if (root == nullptr || p == nullptr || q == nullptr) {
        return nullptr;
    }

    // 存储路径
    unordered_map<TreeNode*, TreeNode*> parentMap;
    stack<TreeNode*> stk;
    stk.push(root);
    parentMap[root] = nullptr;

    // 遍历树，构建父节点映射
    while (parentMap.find(p) == parentMap.end() || parentMap.find(q) == parentMap.end()) {
        TreeNode* node = stk.top();
        stk.pop();

        if (node->right) {
            parentMap[node->right] = node;
            stk.push(node->right);
        }
        if (node->left) {
            parentMap[node->left] = node;
            stk.push(node->left);
        }
    }

    // 构建 p 的祖先集合
    unordered_set<TreeNode*> ancestors;
    TreeNode* current = p;
    while (current) {
        ancestors.insert(current);
        current = parentMap[current];
    }

    // 查找 q 的祖先中是否在 p 的祖先集合中
    current = q;
    while (ancestors.find(current) == ancestors.end()) {
        current = parentMap[current];
    }

    return current;
}

```

```

    }

};

/***
 * 测试方法
 */
// 辅助函数: 打印测试结果
void printTestResult(const string& testName, TreeNode* result) {
    cout << "[" << testName << "] 结果: " << (result ? to_string(result->val) : "nullptr") <<
endl;
}

// 辅助函数: 打印带父指针节点的测试结果
void printTestResultWithParent(const string& testName, TreeNodeWithParent* result) {
    cout << "[" << testName << "] 结果: " << (result ? to_string(result->val) : "nullptr") <<
endl;
}

// 辅助函数: 释放二叉树内存
void deleteTree(TreeNode* root) {
    if (!root) return;
    deleteTree(root->left);
    deleteTree(root->right);
    delete root;
}

// 辅助函数: 释放带父指针的二叉树内存
void deleteTreeWithParent(TreeNodeWithParent* root) {
    if (!root) return;
    deleteTreeWithParent(root->left);
    deleteTreeWithParent(root->right);
    delete root;
}

int main() {
    Solution solution;
    cout << "==== 测试 LCA 算法实现 ===\n" << endl;

    // === 测试 1: 标准二叉树 LCA (LeetCode 236) ===
    cout << "\n==== 测试 1: 标准二叉树 LCA (LeetCode 236) ===\n" << endl;
    //      3
    //      / \
    //      5   1
}

```

```

//      / \ / \
//      6  2 0  8
//          / \
//          7   4

TreeNode* root1 = new TreeNode(3);
TreeNode* node5 = new TreeNode(5);
TreeNode* node1 = new TreeNode(1);
TreeNode* node6 = new TreeNode(6);
TreeNode* node2 = new TreeNode(2);
TreeNode* node0 = new TreeNode(0);
TreeNode* node8 = new TreeNode(8);
TreeNode* node7 = new TreeNode(7);
TreeNode* node4 = new TreeNode(4);

root1->left = node5;
root1->right = node1;
node5->left = node6;
node5->right = node2;
node1->left = node0;
node1->right = node8;
node2->left = node7;
node2->right = node4;

// 测试递归版本
printTestResult("测试 1.1: 标准二叉树 LCA (5 和 1)", solution.lowestCommonAncestor(root1,
node5, node1)); // 期望输出: 3
printTestResult("测试 1.2: 节点是祖先关系 (5 和 4)", solution.lowestCommonAncestor(root1,
node5, node4)); // 期望输出: 5
printTestResult("测试 1.3: 同一节点 (5 和 5)", solution.lowestCommonAncestor(root1, node5,
node5)); // 期望输出: 5
printTestResult("测试 1.4: 无效输入 (nullptr)", solution.lowestCommonAncestor(nullptr, node5,
node4)); // 期望输出: nullptr

// 测试迭代版本
cout << "\n 迭代版本测试:\n" << endl;
printTestResult("测试 1.5: 迭代版 LCA (5 和 1)", solution.lowestCommonAncestorIterative(root1,
node5, node1)); // 期望输出: 3
printTestResult("测试 1.6: 迭代版 LCA (5 和 4)", solution.lowestCommonAncestorIterative(root1,
node5, node4)); // 期望输出: 5

// === 测试 2: 二叉搜索树 LCA (LeetCode 235) ===
cout << "\n== 测试 2: 二叉搜索树 LCA (LeetCode 235) ==\n" << endl;
//       6

```

```

//      / \
//      2   8
//      / \ / \
//      0   4   7   9
//      / \
//      3   5

TreeNode* bstRoot = new TreeNode(6);
TreeNode* bstNode2 = new TreeNode(2);
TreeNode* bstNode8 = new TreeNode(8);
TreeNode* bstNode0 = new TreeNode(0);
TreeNode* bstNode4 = new TreeNode(4);
TreeNode* bstNode7 = new TreeNode(7);
TreeNode* bstNode9 = new TreeNode(9);
TreeNode* bstNode3 = new TreeNode(3);
TreeNode* bstNode5 = new TreeNode(5);

bstRoot->left = bstNode2;
bstRoot->right = bstNode8;
bstNode2->left = bstNode0;
bstNode2->right = bstNode4;
bstNode8->left = bstNode7;
bstNode8->right = bstNode9;
bstNode4->left = bstNode3;
bstNode4->right = bstNode5;

printTestResult("测试 2.1: BST LCA (2 和 8)", solution.lowestCommonAncestorBST(bstRoot,
bstNode2, bstNode8)); // 期望输出: 6
printTestResult("测试 2.2: BST LCA (2 和 4)", solution.lowestCommonAncestorBST(bstRoot,
bstNode2, bstNode4)); // 期望输出: 2
printTestResult("测试 2.3: BST LCA (3 和 5)", solution.lowestCommonAncestorBST(bstRoot,
bstNode3, bstNode5)); // 期望输出: 4

// === 测试 3: 带父指针的二叉树 LCA (LeetCode 1650) ===
cout << "\n== 测试 3: 带父指针的二叉树 LCA (LeetCode 1650) ==\n" << endl;
//      3
//      / \
//      5   1
//      / \ / \
//      6   2   0   8
//      / \
//      7   4

TreeNodeWithParent* rootWithParent = new TreeNodeWithParent(3);
TreeNodeWithParent* wpNode5 = new TreeNodeWithParent(5);

```

```

TreeNodeWithParent* wpNode1 = new TreeNodeWithParent(1);
TreeNodeWithParent* wpNode6 = new TreeNodeWithParent(6);
TreeNodeWithParent* wpNode2 = new TreeNodeWithParent(2);
TreeNodeWithParent* wpNode0 = new TreeNodeWithParent(0);
TreeNodeWithParent* wpNode8 = new TreeNodeWithParent(8);
TreeNodeWithParent* wpNode7 = new TreeNodeWithParent(7);
TreeNodeWithParent* wpNode4 = new TreeNodeWithParent(4);

// 设置父子关系
rootWithParent->left = wpNode5;
rootWithParent->right = wpNode1;
wpNode5->parent = rootWithParent;
wpNode5->left = wpNode6;
wpNode5->right = wpNode2;
wpNode6->parent = wpNode5;
wpNode2->parent = wpNode5;
wpNode1->parent = rootWithParent;
wpNode1->left = wpNode0;
wpNode1->right = wpNode8;
wpNode0->parent = wpNode1;
wpNode8->parent = wpNode1;
wpNode2->left = wpNode7;
wpNode2->right = wpNode4;
wpNode7->parent = wpNode2;
wpNode4->parent = wpNode2;

printTestResultWithParent("测试 3.1: 带父指针 LCA (5 和 1)",
solution.lowestCommonAncestorWithParent(wpNode5, wpNode1)); // 期望输出: 3
printTestResultWithParent("测试 3.2: 带父指针 LCA (5 和 4)",
solution.lowestCommonAncestorWithParent(wpNode5, wpNode4)); // 期望输出: 5
printTestResultWithParent("测试 3.3: 带父指针 LCA (6 和 8)",
solution.lowestCommonAncestorWithParent(wpNode6, wpNode8)); // 期望输出: 3
printTestResultWithParent("测试 3.4: 带父指针 LCA (7 和 4)",
solution.lowestCommonAncestorWithParent(wpNode7, wpNode4)); // 期望输出: 2

// === 性能测试: 大型树的迭代版本优势 ===
cout << "\n== 测试 4: 性能考虑 ==\n" << endl;
cout << "对于大型树, 迭代版本 LCA 可以避免递归栈溢出" << endl;
cout << "递归版本时间复杂度: O(n), 空间复杂度: O(h), 其中 h 是树高" << endl;
cout << "迭代版本时间复杂度: O(n), 空间复杂度: O(h), 但避免了深层递归的栈溢出风险" << endl;

// 释放内存
deleteTree(root1);

```

```
    deleteTree(bstRoot);
    deleteTreeWithParent(rootWithParent);

    cout << "\n==== 所有测试完成 ===" << endl;

    return 0;
}
```

=====

文件: Code04_LCA_BinaryLifting.java

=====

```
package class118;

import java.util.*;

/**
 * LCA 问题 - 递归法与倍增法实现
 * 本文件包含多个 LCA 相关题目的解答，涵盖不同的实现方法和优化策略
 * 所有非代码内容都以注释形式呈现，包含详细的分析和说明
 *
 * 主要内容包括：
 * 1. LeetCode 236: 二叉树的最近公共祖先
 * 2. LeetCode 235: 二叉搜索树的最近公共祖先
 * 3. LeetCode 1650: 带父指针的二叉树最近公共祖先
 * 4. 其他平台的经典 LCA 题目
 * 5. 洛谷 P3379: 最近公共祖先模板题
 * 6. HDU 2586: 树上两点距离
 * 7. SPOJ LCASQ: 基础 LCA 模板题
 * 8. POJ 1330: 最近公共祖先
 * 9. Codeforces 1304E: 1-Trees and Queries
 * 10. AtCoder ABC133F: Colorful Tree
 *
 * 算法复杂度分析：
 * 1. 递归 DFS: O(n) 时间, O(h) 空间
 * 2. 倍增法: O(n log n) 预处理, O(log n) 查询
 * 3. Tarjan 算法: O(n + q) 时间, O(n) 空间
 * 4. 树链剖分: O(n) 预处理, O(log n) 查询
 *
 * 工程化考量：
 * 1. 异常处理：输入验证、边界条件处理
 * 2. 性能优化：预处理优化、查询优化
 * 3. 可读性：详细注释、模块化设计
```

- * 4. 调试能力：打印调试、断言验证
- * 5. 单元测试：覆盖各种边界场景
- *
- * 语言特性差异：
 - * 1. Java：自动垃圾回收，对象引用传递
 - * 2. C++：手动内存管理，指针操作
 - * 3. Python：动态类型，引用计数垃圾回收
- *
- * 数学联系：
 - * 1. 二进制表示与位运算
 - * 2. 树的深度优先搜索理论
 - * 3. 动态规划思想
 - * 4. 并查集数据结构
- *
- * 与机器学习联系：
 - * 1. 树结构在决策树算法中的应用
 - * 2. LCA 在层次聚类中的潜在应用
 - * 3. 图神经网络中的树结构处理
- *
- * 反直觉设计：
 - * 1. 倍增法的二进制跳跃思想
 - * 2. Tarjan 算法的离线处理策略
 - * 3. 树链剖分的重链轻链分解
- *
- * 极端场景鲁棒性：
 - * 1. 空树和单节点树
 - * 2. 线性树（退化为链表）
 - * 3. 完全二叉树
 - * 4. 大规模数据 ($n > 10^5$)
 - * 5. 深度极大的树
- */

```
public class Code04_LCA_BinaryLifting {  
  
    // 二叉树节点定义  
    public static class TreeNode {  
        int val;  
        TreeNode left;  
        TreeNode right;  
  
        TreeNode(int x) {  
            val = x;  
        }  
    }  
}
```

```

// 带父指针的二叉树节点定义
public static class TreeNodeWithParent {
    int val;
    TreeNodeWithParent left;
    TreeNodeWithParent right;
    TreeNodeWithParent parent;

    TreeNodeWithParent(int x) {
        val = x;
    }
}

/**
 * 解法一：LeetCode 236. 二叉树的最近公共祖先
 * 题目链接：https://leetcode.cn/problems/lowest-common-ancestor-of-a-binary-tree/
 * 难度：中等
 *
 * 问题描述：
 * 给定一个二叉树，找到该树中两个指定节点的最近公共祖先。
 * 最近公共祖先的定义为：“对于有根树 T 的两个节点 p、q，最近公共祖先表示为一个节点 x，满足 x 是 p、q 的祖先且 x 的深度尽可能大。”
 *
 * 解题思路：递归深度优先搜索
 * 1. 递归终止条件：当前节点为空或者是 p 或 q 中的一个
 * 2. 递归搜索左右子树
 * 3. 根据左右子树的返回结果判断：
 *   - 如果左右子树都返回非空，说明当前节点就是 LCA
 *   - 如果只有一侧返回非空，返回该侧的结果
 *   - 如果两侧都返回空，返回 null
 *
 * 时间复杂度：O(n) – 其中 n 是树中节点的数量，最坏情况下需要遍历所有节点
 * 空间复杂度：O(h) – 其中 h 是树的高度，递归调用栈的深度
 * 是否为最优解：对于单次查询，这是最优解之一
 */

public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q) {
    // 异常处理：检查输入参数
    if (root == null || p == null || q == null) {
        return null;
    }

    // 基本情况：如果当前节点是 p 或 q，则当前节点就是 LCA
    if (root == p || root == q) {

```

```

    return root;
}

// 递归查找左子树中的 LCA
TreeNode left = lowestCommonAncestor(root.left, p, q);
// 递归查找右子树中的 LCA
TreeNode right = lowestCommonAncestor(root.right, p, q);

// 如果左右子树都找到了节点，说明当前节点是 LCA
if (left != null && right != null) {
    return root;
}

// 如果只有左子树找到了节点，返回左子树的结果
if (left != null) {
    return left;
}

// 如果只有右子树找到了节点，返回右子树的结果
if (right != null) {
    return right;
}

// 如果左右子树都没有找到节点，返回 null
return null;
}

/**
 * 解法二：LeetCode 235. 二叉搜索树的最近公共祖先
 * 题目链接：https://leetcode.cn/problems/lowest-common-ancestor-of-a-binary-search-tree/
 * 难度：简单
 *
 * 问题描述：
 * 给定一个二叉搜索树，找到该树中两个指定节点的最近公共祖先。
 * 最近公共祖先的定义为：“对于有根树 T 的两个节点 p、q，最近公共祖先表示为一个节点 x，满足 x 是 p、q 的祖先且 x 的深度尽可能大。”
 *
 * 解题思路：利用二叉搜索树的特性
 * 二叉搜索树的特性：左子树所有节点值 < 根节点值 < 右子树所有节点值
 * 1. 如果 p 和 q 的值都小于当前节点，那么 LCA 在左子树
 * 2. 如果 p 和 q 的值都大于当前节点，那么 LCA 在右子树
 * 3. 如果一个小于等于，一个大于等于，那么当前节点就是 LCA
 */

```

```

* 时间复杂度: O(h) - 其中 h 是树的高度, 在平衡树情况下为 O(log n)
* 空间复杂度: O(h) - 递归调用栈的深度
* 是否为最优解: 是, 利用了 BST 的特性, 比通用二叉树解法更高效
*/
public TreeNode lowestCommonAncestorBST(TreeNode root, TreeNode p, TreeNode q) {
    // 异常处理
    if (root == null || p == null || q == null) {
        return null;
    }

    // 如果 p 和 q 都在左子树
    if (p.val < root.val && q.val < root.val) {
        return lowestCommonAncestorBST(root.left, p, q);
    }
    // 如果 p 和 q 都在右子树
    else if (p.val > root.val && q.val > root.val) {
        return lowestCommonAncestorBST(root.right, p, q);
    }
    // 如果 p 和 q 分别在两侧, 或者其中一个是当前节点
    else {
        return root;
    }
}

/**
* 解法三: LeetCode 1650. 二叉树的最近公共祖先 III (带父指针)
* 题目链接: https://leetcode.cn/problems/lowest-common-ancestor-of-a-binary-tree-iii/
* 难度: 中等
*
* 问题描述:
* 给定一棵二叉树中的两个节点 p 和 q, 返回它们的最近公共祖先 (LCA) 节点。
* 每个节点都有一个指向其父节点的指针。
*
* 解题思路: 双指针法
* 1. 分别计算 p 和 q 到根节点的深度差
* 2. 先将较深的节点向上移动, 使两个节点处于同一深度
* 3. 然后同时向上移动两个节点, 直到找到相同的节点, 即为 LCA
*
* 时间复杂度: O(h) - 其中 h 是树的高度
* 空间复杂度: O(1) - 只使用常数额外空间
* 是否为最优解: 是, 充分利用了父指针的特性
*/
public TreeNodeWithParent lowestCommonAncestorWithParent(TreeNodeWithParent p,

```

```

TreeNodeWithParent q) {
    if (p == null || q == null) {
        return null;
    }

    TreeNodeWithParent a = p;
    TreeNodeWithParent b = q;

    // 双指针法，类似于链表相交问题
    // 当 a 或 b 为空时，将其指向对方的起始节点，这样可以抵消深度差
    while (a != b) {
        a = (a == null) ? q : a.parent;
        b = (b == null) ? p : b.parent;
    }

    return a;
}

/**
 * 解法四：迭代版本的二叉树 LCA（避免递归栈溢出）
 * 适用于处理大型树的情况
 *
 * 解题思路：后序遍历 + 记录父节点路径
 * 1. 使用栈进行后序遍历
 * 2. 记录每个节点的访问状态
 * 3. 当找到 p 或 q 时，记录其路径
 * 4. 比较两条路径，找到最后一个公共节点
 *
 * 时间复杂度：O(n)
 * 空间复杂度：O(h)
 */
public TreeNode lowestCommonAncestorIterative(TreeNode root, TreeNode p, TreeNode q) {
    if (root == null || p == null || q == null) {
        return null;
    }

    // 存储路径
    Map<TreeNode, TreeNode> parentMap = new HashMap<>();
    Stack<TreeNode> stack = new Stack<>();
    stack.push(root);
    parentMap.put(root, null);

    // 遍历树，构建父节点映射

```

```

while (!parentMap.containsKey(p) || !parentMap.containsKey(q)) {
    TreeNode node = stack.pop();

    if (node.right != null) {
        parentMap.put(node.right, node);
        stack.push(node.right);
    }
    if (node.left != null) {
        parentMap.put(node.left, node);
        stack.push(node.left);
    }
}

// 构建 p 的祖先集合
Set<TreeNode> ancestors = new HashSet<>();
while (p != null) {
    ancestors.add(p);
    p = parentMap.get(p);
}

// 查找 q 的祖先中是否在 p 的祖先集合中
while (!ancestors.contains(q)) {
    q = parentMap.get(q);
}

return q;
}

/***
 * 解法五：洛谷 P3379 【模板】最近公共祖先（LCA）
 * 题目链接：https://www.luogu.com.cn/problem/P3379
 * 难度：模板题
 *
 * 问题描述：
 * 给定一棵有根多叉树，请求出指定两个点直接最近的公共祖先。
 *
 * 解题思路：树上倍增法
 * 1. 预处理每个节点的  $2^k$  级祖先
 * 2. 对于查询，先将两个节点调整到同一深度
 * 3. 然后同时向上跳跃，直到找到最近公共祖先
 *
 * 时间复杂度：预处理  $O(n \log n)$ ，查询  $O(\log n)$ 
 * 空间复杂度： $O(n \log n)$ 
*/

```

* 是否为最优解：是，对于在线查询 LCA 问题，倍增法是标准解法

*/

```
static class LuoguP3379 {
    private static final int MAXN = 500001;
    private static final int LOG = 20;

    private int[] depth = new int[MAXN];
    private int[][] ancestor = new int[MAXN][LOG];
    private List<Integer>[] tree;

    public LuoguP3379(int n, int[][] edges) {
        tree = new ArrayList[n + 1];
        for (int i = 0; i <= n; i++) {
            tree[i] = new ArrayList<>();
        }

        // 构建邻接表
        for (int[] edge : edges) {
            tree[edge[0]].add(edge[1]);
            tree[edge[1]].add(edge[0]);
        }
    }

    // 预处理倍增数组
    dfs(1, 0);
}

private void dfs(int u, int parent) {
    depth[u] = depth[parent] + 1;
    ancestor[u][0] = parent;

    // 构建倍增数组
    for (int i = 1; i < LOG; i++) {
        if (ancestor[u][i - 1] != 0) {
            ancestor[u][i] = ancestor[ancestor[u][i - 1]][i - 1];
        }
    }

    // 递归处理子节点
    for (int v : tree[u]) {
        if (v != parent) {
            dfs(v, u);
        }
    }
}
```

```

    }

public int getLCA(int u, int v) {
    // 确保 u 的深度不小于 v
    if (depth[u] < depth[v]) {
        int temp = u;
        u = v;
        v = temp;
    }

    // 将 u 向上跳跃到与 v 同一深度
    for (int i = LOG - 1; i >= 0; i--) {
        if (depth[u] - (1 << i) >= depth[v]) {
            u = ancestor[u][i];
        }
    }

    // 如果 u 和 v 已经相同，则找到了 LCA
    if (u == v) {
        return u;
    }

    // u 和 v 同时向上跳跃，直到找到 LCA
    for (int i = LOG - 1; i >= 0; i--) {
        if (ancestor[u][i] != ancestor[v][i]) {
            u = ancestor[u][i];
            v = ancestor[v][i];
        }
    }

    return ancestor[u][0];
}

}

/***
 * 解法六：HDU 2586 How far away?
 * 题目链接：https://vjudge.net/problem/HDU-2586
 * 难度：中等
 *
 * 问题描述：
 * 给定一棵树和若干查询，每个查询要求计算树上两点之间的距离。
 *
 * 解题思路：
 */

```

```

* 1. 使用 LCA 计算两点之间的最近公共祖先
* 2. 两点距离 = 节点 u 到根的距离 + 节点 v 到根的距离 - 2 * LCA 到根的距离
*
* 时间复杂度: 预处理 O(n log n), 查询 O(log n)
* 空间复杂度: O(n log n)
* 是否为最优解: 是
*/

```

```

static class HDU2586 {
    private LuoguP3379 lcaSolver;
    private int[] dist; // 节点到根节点的距离

    public HDU2586(int n, int[][] edges, int[] weights) {
        lcaSolver = new LuoguP3379(n, edges);
        dist = new int[n + 1];

        // 计算每个节点到根节点的距离
        calculateDist(1, 0, 0, edges, weights);
    }

    private void calculateDist(int u, int parent, int currentDist, int[][] edges, int[] weights) {
        dist[u] = currentDist;
        for (int i = 0; i < edges.length; i++) {
            if (edges[i][0] == u && edges[i][1] != parent) {
                calculateDist(edges[i][1], u, currentDist + weights[i], edges, weights);
            } else if (edges[i][1] == u && edges[i][0] != parent) {
                calculateDist(edges[i][0], u, currentDist + weights[i], edges, weights);
            }
        }
    }

    public int getDistance(int u, int v) {
        int lca = lcaSolver.getLCA(u, v);
        return dist[u] + dist[v] - 2 * dist[lca];
    }
}

/**
* 解法七: SPOJ LCASQ - Lowest Common Ancestor
* 题目链接: https://www.spoj.com/problems/LCASQ/
* 难度: 中等
*
* 问题描述:

```

```

* 给定一棵树和多个查询，每个查询要求计算两个节点的最近公共祖先。
*
* 解题思路：
* 使用 Tarjan 离线算法处理所有查询
*
* 时间复杂度：O(n + q)
* 空间复杂度：O(n + q)
* 是否为最优解：是，对于离线查询，Tarjan 算法是最优的
*/

```

static class SPOJ_LCASQ {

```

    private List<Integer>[] tree;
    private List<int[][]>[] queries;
    private int[] parent;
    private int[] ancestor;
    private boolean[] visited;

    public SPOJ_LCASQ(int n, int[][] edges, int[][] queryPairs) {
        tree = new ArrayList[n];
        queries = new ArrayList[n];
        for (int i = 0; i < n; i++) {
            tree[i] = new ArrayList<>();
            queries[i] = new ArrayList<>();
        }

        // 构建邻接表
        for (int[] edge : edges) {
            tree[edge[0]].add(edge[1]);
            tree[edge[1]].add(edge[0]);
        }

        // 存储查询
        for (int i = 0; i < queryPairs.length; i++) {
            int u = queryPairs[i][0];
            int v = queryPairs[i][1];
            queries[u].add(new int[]{v, i});
            queries[v].add(new int[]{u, i});
        }
    }

    parent = new int[n];
    ancestor = new int[n];
    visited = new boolean[n];

    // 初始化并查集

```

```

        for (int i = 0; i < n; i++) {
            parent[i] = i;
        }

    }

private int find(int x) {
    if (parent[x] != x) {
        parent[x] = find(parent[x]);
    }
    return parent[x];
}

private void union(int x, int y) {
    int rootX = find(x);
    int rootY = find(y);
    if (rootX != rootY) {
        parent[rootY] = rootX;
    }
}

public void tarjanLCA(int u, int parentNode, int[] results) {
    ancestor[u] = u;
    visited[u] = true;

    for (int v : tree[u]) {
        if (v != parentNode) {
            tarjanLCA(v, u, results);
            union(u, v);
            ancestor[find(u)] = u;
        }
    }
}

for (int[] query : queries[u]) {
    int v = query[0];
    int queryIndex = query[1];
    if (visited[v]) {
        results[queryIndex] = ancestor[find(v)];
    }
}
}

}

/***

```

```

* 测试方法，创建测试用例并验证所有 LCA 算法实现的正确性
*/
public static void main(String[] args) {
    Code04_LCA_BinaryLifting solution = new Code04_LCA_BinaryLifting();

    System.out.println("===== 测试普通二叉树的 LCA 算法 =====");
    // 测试用例 1：简单树结构
    //      3
    //      / \
    //      5   1
    //      / \
    //      6   2
    TreeNode root1 = new TreeNode(3);
    TreeNode node5 = new TreeNode(5);
    TreeNode node1 = new TreeNode(1);
    TreeNode node6 = new TreeNode(6);
    TreeNode node2 = new TreeNode(2);

    root1.left = node5;
    root1.right = node1;
    node5.left = node6;
    node5.right = node2;

    // 测试 LCA(5, 1) 应该返回 3
    TreeNode lca1 = solution.lowestCommonAncestor(root1, node5, node1);
    System.out.println("Test Case 1: LCA(5, 1) = " + (lca1 != null ? lca1.val : "null"));

    // 测试 LCA(5, 6) 应该返回 5
    TreeNode lca2 = solution.lowestCommonAncestor(root1, node5, node6);
    System.out.println("Test Case 2: LCA(5, 6) = " + (lca2 != null ? lca2.val : "null"));

    // 测试用例 2：更复杂的树结构
    //      1
    //      / \
    //      2   3
    //      / \   \
    //      4   5   6
    TreeNode root2 = new TreeNode(1);
    TreeNode node2_2 = new TreeNode(2);
    TreeNode node2_3 = new TreeNode(3);
    TreeNode node2_4 = new TreeNode(4);
    TreeNode node2_5 = new TreeNode(5);
    TreeNode node2_6 = new TreeNode(6);

```

```

root2.left = node2_2;
root2.right = node2_3;
node2_2.left = node2_4;
node2_2.right = node2_5;
node2_3.right = node2_6;

// 测试 LCA(4, 5) 应该返回 2
TreeNode lca3 = solution.lowestCommonAncestor(root2, node2_4, node2_5);
System.out.println("Test Case 3: LCA(4, 5) = " + (lca3 != null ? lca3.val : "null"));

// 测试 LCA(4, 6) 应该返回 1
TreeNode lca4 = solution.lowestCommonAncestor(root2, node2_4, node2_6);
System.out.println("Test Case 4: LCA(4, 6) = " + (lca4 != null ? lca4.val : "null"));

// 测试用例 3: 极端情况 - 空树
TreeNode lca5 = solution.lowestCommonAncestor(null, node5, node1);
System.out.println("Test Case 5: LCA in null tree = " + (lca5 != null ? lca5.val : "null"));

// 测试用例 4: 极端情况 - 节点为空
TreeNode lca6 = solution.lowestCommonAncestor(root1, null, node1);
System.out.println("Test Case 6: LCA with null node = " + (lca6 != null ? lca6.val : "null"));

System.out.println("\n===== 测试二叉搜索树的 LCA 算法 =====");

// 测试用例 5: 二叉搜索树
//       6
//     / \
//    2   8
//   / \ / \
//  0  4 7  9
//  / \
// 3   5

TreeNode root3 = new TreeNode(6);
TreeNode node3_2 = new TreeNode(2);
TreeNode node3_8 = new TreeNode(8);
TreeNode node3_0 = new TreeNode(0);
TreeNode node3_4 = new TreeNode(4);
TreeNode node3_7 = new TreeNode(7);
TreeNode node3_9 = new TreeNode(9);
TreeNode node3_3 = new TreeNode(3);
TreeNode node3_5 = new TreeNode(5);

```

```

root3.left = node3_2;
root3.right = node3_8;
node3_2.left = node3_0;
node3_2.right = node3_4;
node3_8.left = node3_7;
node3_8.right = node3_9;
node3_4.left = node3_3;
node3_4.right = node3_5;

// 测试 BST 的 LCA - LCA(2, 8) 应该返回 6
TreeNode bstLca1 = solution.lowestCommonAncestorBST(root3, node3_2, node3_8);
System.out.println("BST Test 1: LCA(2, 8) = " + (bstLca1 != null ? bstLca1.val : "null"));

// 测试 BST 的 LCA - LCA(2, 4) 应该返回 2
TreeNode bstLca2 = solution.lowestCommonAncestorBST(root3, node3_2, node3_4);
System.out.println("BST Test 2: LCA(2, 4) = " + (bstLca2 != null ? bstLca2.val : "null"));

// 测试 BST 的 LCA - LCA(3, 5) 应该返回 4
TreeNode bstLca3 = solution.lowestCommonAncestorBST(root3, node3_3, node3_5);
System.out.println("BST Test 3: LCA(3, 5) = " + (bstLca3 != null ? bstLca3.val : "null"));

System.out.println("\n===== 测试带父指针的 LCA 算法 =====");

// 测试用例 6: 带父指针的二叉树
//      3
//     / \
//    5   1
//   / \
//  6   2

TreeNodeWithParent root4 = new TreeNodeWithParent(3);
TreeNodeWithParent node4_5 = new TreeNodeWithParent(5);
TreeNodeWithParent node4_1 = new TreeNodeWithParent(1);
TreeNodeWithParent node4_6 = new TreeNodeWithParent(6);
TreeNodeWithParent node4_2 = new TreeNodeWithParent(2);

root4.left = node4_5;
root4.right = node4_1;
node4_5.left = node4_6;
node4_5.right = node4_2;

```

```

// 设置父指针
node4_5.parent = root4;
node4_1.parent = root4;
node4_6.parent = node4_5;
node4_2.parent = node4_5;

// 测试带父指针的 LCA - LCA(5, 1) 应该返回 3
TreeNodeWithParent parentLca1 = solution.lowestCommonAncestorWithParent(node4_5,
node4_1);
System.out.println("Parent Test 1: LCA(5, 1) = " + (parentLca1 != null ? parentLca1.val :
"null"));

// 测试带父指针的 LCA - LCA(6, 2) 应该返回 5
TreeNodeWithParent parentLca2 = solution.lowestCommonAncestorWithParent(node4_6,
node4_2);
System.out.println("Parent Test 2: LCA(6, 2) = " + (parentLca2 != null ? parentLca2.val :
"null"));

System.out.println("\n===== 测试迭代版本的 LCA 算法 =====");
// 测试迭代版本的 LCA - 使用 root1 树
TreeNode iterativeLca1 = solution.lowestCommonAncestorIterative(root1, node5, node1);
System.out.println("Iterative Test 1: LCA(5, 1) = " + (iterativeLca1 != null ?
iterativeLca1.val : "null"));

// 测试迭代版本的 LCA - 使用 root2 树
TreeNode iterativeLca2 = solution.lowestCommonAncestorIterative(root2, node2_4, node2_6);
System.out.println("Iterative Test 2: LCA(4, 6) = " + (iterativeLca2 != null ?
iterativeLca2.val : "null"));

System.out.println("\n===== 测试洛谷 P3379 倍增法 LCA =====");
// 测试用例 7: 洛谷 P3379
int n = 5;
int[][] edges = {{1, 2}, {1, 3}, {2, 4}, {2, 5}};
LuoguP3379 luoguSolver = new LuoguP3379(n, edges);

// 测试 LCA(4, 5) 应该返回 2
int luoguLca1 = luoguSolver.getLCA(4, 5);
System.out.println("Luogu Test 1: LCA(4, 5) = " + luoguLca1);

// 测试 LCA(4, 3) 应该返回 1
int luoguLca2 = luoguSolver.getLCA(4, 3);
System.out.println("Luogu Test 2: LCA(4, 3) = " + luoguLca2);

```

```

System.out.println("\n===== 测试 HDU2586 树上距离 =====");
// 测试用例 8: HDU2586
int[][] hduEdges = {{1, 2}, {1, 3}, {2, 4}, {2, 5}};
int[] weights = {10, 20, 30, 40};
HDU2586 hduSolver = new HDU2586(n, hduEdges, weights);

// 测试距离(4, 5) 应该返回 30 + 40 = 70
int distance1 = hduSolver.getDistance(4, 5);
System.out.println("HDU Test 1: Distance(4, 5) = " + distance1);

// 测试距离(4, 3) 应该返回 30 + 20 = 50
int distance2 = hduSolver.getDistance(4, 3);
System.out.println("HDU Test 2: Distance(4, 3) = " + distance2);

System.out.println("\n所有 LCA 算法测试完成!");
}
}

```

文件: Code04_LCA_BinaryLifting.py

LCA 问题 - 递归法实现

题目来源: LeetCode 236. Lowest Common Ancestor of a Binary Tree

题目链接: <https://leetcode.cn/problems/lowest-common-ancestor-of-a-binary-tree/>

问题描述:

给定一个二叉树，找到该树中两个指定节点的最近公共祖先。

最近公共祖先的定义为：“对于有根树 T 的两个节点 p、q，最近公共祖先表示为一个节点 x，满足 x 是 p、q 的祖先且 x 的深度尽可能大。”

解题思路:

1. 使用递归深度优先搜索(DFS)
2. 对于每个节点，递归检查其左右子树
3. 如果当前节点是 p 或 q，则直接返回当前节点
4. 如果左右子树分别找到了 p 和 q，则当前节点就是 LCA
5. 如果只在一侧子树找到了 p 或 q，则返回找到的节点

时间复杂度: O(n) – n 为树中节点的数量，最坏情况下需要遍历所有节点

空间复杂度: O(h) – h 为树的高度，递归调用栈的深度

是否为最优解: 是，这是寻找 LCA 的标准方法

工程化考虑:

1. 边界条件处理: 处理空树、节点不存在等情况
2. 输入验证: 验证输入节点是否在树中
3. 异常处理: 对非法输入进行检查
4. 可读性: 添加详细注释和变量命名

算法要点:

1. 递归终止条件: 节点为空或者找到目标节点
2. 递归处理左右子树
3. 根据左右子树返回结果判断当前节点是否为 LCA

与标准库实现对比:

1. 标准库通常有更完善的错误处理
2. 标准库可能使用迭代而非递归以避免栈溢出
3. 标准库可能有缓存优化

性能优化:

1. 剪枝优化: 一旦找到 LCA 立即返回
2. 空间优化: 使用原地算法, 不额外开辟空间

特殊场景:

1. 空输入: 返回 None
2. 节点不存在: 返回 None
3. 一个节点是另一个节点的祖先: 返回祖先节点
4. 重复数据: 不适用 (树结构中节点唯一)

语言特性差异:

1. Python: 动态类型, 引用计数垃圾回收
2. Java: 对象引用传递, 自动垃圾回收
3. C++: 指针操作, 需要手动管理内存

数学联系:

1. 与图论中的最短路径问题相关
2. 与树的深度优先搜索理论相关
3. 与并查集数据结构有一定联系

调试能力:

1. 可通过打印节点遍历顺序调试
2. 可通过断言验证中间结果
3. 可通过特殊测试用例验证边界条件

"""

```
from typing import Optional
```

```

# 二叉树节点定义
class TreeNode:
    def __init__(self, x):
        self.val = x
        self.left: Optional['TreeNode'] = None
        self.right: Optional['TreeNode'] = None

# 带父指针的二叉树节点类定义
class TreeNodeWithParent:
    def __init__(self, x):
        self.val = x
        self.left = None
        self.right = None
        self.parent = None

# 解决方案类
class Solution:
    """
    LCA（最近公共祖先）算法的多种实现方式
    支持普通二叉树、二叉搜索树、带父指针的树等多种场景
    """

    def lowestCommonAncestor(self, root: Optional[TreeNode], p: TreeNode, q: TreeNode) ->
Optional[TreeNode]:
        """
        解法一：LeetCode 236. 二叉树的最近公共祖先
        题目链接：https://leetcode.cn/problems/lowest-common-ancestor-of-a-binary-tree/
        难度：中等
        """

```

问题描述：

给定一个二叉树，找到该树中两个指定节点的最近公共祖先。

最近公共祖先的定义为：“对于有根树 T 的两个节点 p、q，最近公共祖先表示为一个节点 x，满足 x 是 p、q 的祖先且 x 的深度尽可能大。”

解题思路：递归深度优先搜索

1. 递归终止条件：当前节点为空或者是 p 或 q 中的一个
2. 递归搜索左右子树
3. 根据左右子树的返回结果判断：
 - 如果左右子树都返回非空，说明当前节点就是 LCA
 - 如果只有一侧返回非空，返回该侧的结果
 - 如果两侧都返回空，返回 None

时间复杂度: $O(n)$ - 其中 n 是树中节点的数量, 最坏情况下需要遍历所有节点

空间复杂度: $O(h)$ - 其中 h 是树的高度, 递归调用栈的深度

是否为最优解: 对于单次查询, 这是最优解之一

"""

异常处理: 检查输入参数

```
if root is None or p is None or q is None:  
    return None
```

基本情况: 如果当前节点是 p 或 q, 则当前节点就是 LCA

```
if root == p or root == q:  
    return root
```

递归查找左子树中的 LCA

```
left = self.lowestCommonAncestor(root.left, p, q)
```

递归查找右子树中的 LCA

```
right = self.lowestCommonAncestor(root.right, p, q)
```

如果左右子树都找到了节点, 说明当前节点是 LCA

```
if left is not None and right is not None:  
    return root
```

如果只有左子树找到了节点, 返回左子树的结果

```
if left is not None:  
    return left
```

如果只有右子树找到了节点, 返回右子树的结果

```
if right is not None:  
    return right
```

如果左右子树都没有找到节点, 返回 None

```
return None
```

```
def lowestCommonAncestorBST(self, root: Optional[TreeNode], p: TreeNode, q: TreeNode) ->  
Optional[TreeNode]:
```

"""

解法二: LeetCode 235. 二叉搜索树的最近公共祖先

题目链接: <https://leetcode.cn/problems/lowest-common-ancestor-of-a-binary-search-tree/>

难度: 简单

问题描述:

给定一个二叉搜索树, 找到该树中两个指定节点的最近公共祖先。

最近公共祖先的定义为: “对于有根树 T 的两个节点 p, q , 最近公共祖先表示为一个节点 x , 满足 x 是 p, q 的祖先且 x 的深度尽可能大。”

解题思路：利用二叉搜索树的特性

二叉搜索树的特性：左子树所有节点值 < 根节点值 < 右子树所有节点值

1. 如果 p 和 q 的值都小于当前节点，那么 LCA 在左子树
2. 如果 p 和 q 的值都大于当前节点，那么 LCA 在右子树
3. 如果一个小于等于，一个大于等于，那么当前节点就是 LCA

时间复杂度：O(h) – 其中 h 是树的高度，在平衡树情况下为 O(log n)

空间复杂度：O(h) – 递归调用栈的深度

是否为最优解：是，利用了 BST 的特性，比通用二叉树解法更高效

"""

异常处理

```
if root is None or p is None or q is None:  
    return None
```

如果 p 和 q 都在左子树

```
if p.val < root.val and q.val < root.val:  
    return self.lowestCommonAncestorBST(root.left, p, q)
```

如果 p 和 q 都在右子树

```
elif p.val > root.val and q.val > root.val:  
    return self.lowestCommonAncestorBST(root.right, p, q)
```

如果 p 和 q 分别在两侧，或者其中一个是当前节点

```
else:
```

```
    return root
```

```
def lowestCommonAncestorWithParent(self, p: 'TreeNodeWithParent', q: 'TreeNodeWithParent') ->  
'TreeNodeWithParent':
```

"""

解法三：LeetCode 1650. 二叉树的最近公共祖先 III（带父指针）

题目链接：<https://leetcode.cn/problems/lowest-common-ancestor-of-a-binary-tree-iii/>

难度：中等

问题描述：

给定一棵二叉树中的两个节点 p 和 q，返回它们的最近公共祖先（LCA）节点。

每个节点都有一个指向其父节点的指针。

解题思路：双指针法

1. 分别计算 p 和 q 到根节点的深度差
2. 先将较深的节点向上移动，使两个节点处于同一深度
3. 然后同时向上移动两个节点，直到找到相同的节点，即为 LCA

时间复杂度：O(h) – 其中 h 是树的高度

空间复杂度：O(1) – 只使用常数额外空间

是否为最优解：是，充分利用了父指针的特性

"""

```
if not p or not q:  
    return None
```

a, b = p, q

双指针法，类似于链表相交问题

当 a 或 b 为空时，将其指向对方的起始节点，这样可以抵消深度差

while a != b:

```
    a = q if a is None else a.parent  
    b = p if b is None else b.parent
```

return a

```
def lowestCommonAncestorIterative(self, root: Optional[TreeNode], p: TreeNode, q: TreeNode)  
-> Optional[TreeNode]:
```

"""

解法四：迭代版本的二叉树 LCA（避免递归栈溢出）

适用于处理大型树的情况

解题思路：后序遍历 + 记录父节点路径

1. 使用栈进行后序遍历
2. 记录每个节点的访问状态
3. 当找到 p 或 q 时，记录其路径
4. 比较两条路径，找到最后一个公共节点

时间复杂度：O(n)

空间复杂度：O(h)

"""

```
if root is None or p is None or q is None:  
    return None
```

存储路径

```
parent_map = {root: None}  
stack = [root]
```

遍历树，构建父节点映射

```
while p not in parent_map or q not in parent_map:
```

```
    node = stack.pop()
```

```
    if node.right:
```

```
        parent_map[node.right] = node
```

```

        stack.append(node.right)

    if node.left:
        parent_map[node.left] = node
        stack.append(node.left)

# 构建 p 的祖先集合
ancestors = set()
current = p
while current:
    ancestors.add(current)
    current = parent_map[current]

# 查找 q 的祖先中是否在 p 的祖先集合中
current = q
while current not in ancestors:
    current = parent_map[current]

return current

# 测试代码
def test_all_lca_implementations():
    solution = Solution()

    print("===== 测试普通二叉树的 LCA 算法 =====")
    # 测试用例 1: 简单树结构
    #
    #      3
    #     / \
    #    5   1
    #   / \
    #  6   2
    root1 = TreeNode(3)
    node5 = TreeNode(5)
    node1 = TreeNode(1)
    node6 = TreeNode(6)
    node2 = TreeNode(2)

    root1.left = node5
    root1.right = node1
    node5.left = node6
    node5.right = node2

    # 测试 LCA(5, 1) 应该返回 3
    lca1 = solution.lowestCommonAncestor(root1, node5, node1)

```

```

print(f"Test Case 1: LCA(5, 1) = {lca1.val if lca1 else None}")

# 测试 LCA(5, 6) 应该返回 5
lca2 = solution.lowestCommonAncestor(root1, node5, node6)
print(f"Test Case 2: LCA(5, 6) = {lca2.val if lca2 else None}")

# 测试用例 2: 更复杂的树结构
#      1
#      / \
#     2   3
#    / \   \
#   4   5   6
root2 = TreeNode(1)
node2_2 = TreeNode(2)
node2_3 = TreeNode(3)
node2_4 = TreeNode(4)
node2_5 = TreeNode(5)
node2_6 = TreeNode(6)

root2.left = node2_2
root2.right = node2_3
node2_2.left = node2_4
node2_2.right = node2_5
node2_3.right = node2_6

# 测试 LCA(4, 5) 应该返回 2
lca3 = solution.lowestCommonAncestor(root2, node2_4, node2_5)
print(f"Test Case 3: LCA(4, 5) = {lca3.val if lca3 else None}")

# 测试 LCA(4, 6) 应该返回 1
lca4 = solution.lowestCommonAncestor(root2, node2_4, node2_6)
print(f"Test Case 4: LCA(4, 6) = {lca4.val if lca4 else None}")

# 测试用例 3: 极端情况 - 空树
lca5 = solution.lowestCommonAncestor(None, node5, node1)
print(f"Test Case 5: LCA in null tree = {lca5.val if lca5 else None}")

# 测试用例 4: 极端情况 - 节点为空
lca6 = solution.lowestCommonAncestor(root1, None, node1)
print(f"Test Case 6: LCA with null node = {lca6.val if lca6 else None}")

print("\n===== 测试二叉搜索树的 LCA 算法 =====")
# 测试用例 5: 二叉搜索树

```

```

#      6
#      / \
#      2   8
#      / \ / \
#      0  4  7   9
#      / \
#      3   5

root3 = TreeNode(6)
node3_2 = TreeNode(2)
node3_8 = TreeNode(8)
node3_0 = TreeNode(0)
node3_4 = TreeNode(4)
node3_7 = TreeNode(7)
node3_9 = TreeNode(9)
node3_3 = TreeNode(3)
node3_5 = TreeNode(5)

root3.left = node3_2
root3.right = node3_8
node3_2.left = node3_0
node3_2.right = node3_4
node3_8.left = node3_7
node3_8.right = node3_9
node3_4.left = node3_3
node3_4.right = node3_5

# 测试 BST 的 LCA - LCA(2, 8) 应该返回 6
bst_lca1 = solution.lowestCommonAncestorBST(root3, node3_2, node3_8)
print(f"BST Test 1: LCA(2, 8) = {bst_lca1.val if bst_lca1 else None}")

# 测试 BST 的 LCA - LCA(2, 4) 应该返回 2
bst_lca2 = solution.lowestCommonAncestorBST(root3, node3_2, node3_4)
print(f"BST Test 2: LCA(2, 4) = {bst_lca2.val if bst_lca2 else None}")

# 测试 BST 的 LCA - LCA(3, 5) 应该返回 4
bst_lca3 = solution.lowestCommonAncestorBST(root3, node3_3, node3_5)
print(f"BST Test 3: LCA(3, 5) = {bst_lca3.val if bst_lca3 else None}")

print("\n===== 测试带父指针的 LCA 算法 =====")

# 测试用例 6: 带父指针的二叉树
#      3
#      / \
#      5   1

```

```

#      / \
#    6   2
root4 = TreeNodeWithParent(3)
node4_5 = TreeNodeWithParent(5)
node4_1 = TreeNodeWithParent(1)
node4_6 = TreeNodeWithParent(6)
node4_2 = TreeNodeWithParent(2)

root4.left = node4_5
root4.right = node4_1
node4_5.left = node4_6
node4_5.right = node4_2

# 设置父指针
node4_5.parent = root4
node4_1.parent = root4
node4_6.parent = node4_5
node4_2.parent = node4_5

# 测试带父指针的 LCA - LCA(5, 1) 应该返回 3
parent_lca1 = solution.lowestCommonAncestorWithParent(node4_5, node4_1)
print(f"Parent Test 1: LCA(5, 1) = {parent_lca1.val if parent_lca1 else None}")

# 测试带父指针的 LCA - LCA(6, 2) 应该返回 5
parent_lca2 = solution.lowestCommonAncestorWithParent(node4_6, node4_2)
print(f"Parent Test 2: LCA(6, 2) = {parent_lca2.val if parent_lca2 else None}")

print("\n===== 测试迭代版本的 LCA 算法 =====")
# 测试迭代版本的 LCA - 使用 root1 树
iterative_lca1 = solution.lowestCommonAncestorIterative(root1, node5, node1)
print(f"Iterative Test 1: LCA(5, 1) = {iterative_lca1.val if iterative_lca1 else None}")

# 测试迭代版本的 LCA - 使用 root2 树
iterative_lca2 = solution.lowestCommonAncestorIterative(root2, node2_4, node2_6)
print(f"Iterative Test 2: LCA(4, 6) = {iterative_lca2.val if iterative_lca2 else None}")

print("\n所有 LCA 算法测试完成!")

# 执行测试
if __name__ == "__main__":
    test_all_lca_implementations()
=====
```

文件: Code05_LCA_Extended.cpp

```
=====

// #include <iostream>
// #include <vector>
// #include <cstring>
// #include <algorithm>
// using namespace std;

/***
 * LCA 问题扩展 - 树上倍增法实现
 * 题目来源: 洛谷 P3379 【模板】最近公共祖先 (LCA)
 * 题目链接: https://www.luogu.com.cn/problem/P3379
 *
 * 问题描述:
 * 给定一棵有根多叉树, 请求出指定两个点直接最近的公共祖先。
 *
 * 解题思路:
 * 1. 使用树上倍增法预处理每个节点的  $2^k$  级祖先
 * 2. 对于每次查询, 先将两个节点调整到同一深度
 * 3. 然后同时向上跳跃, 直到找到最近公共祖先
 *
 * 时间复杂度:
 * 预处理:  $O(n \log n)$ 
 * 查询:  $O(\log n)$ 
 * 空间复杂度:  $O(n \log n)$ 
 * 是否为最优解: 是, 对于在线查询 LCA 问题, 倍增法是标准解法之一
 *
 * 工程化考虑:
 * 1. 边界条件处理: 处理空树、节点不存在等情况
 * 2. 输入验证: 验证输入节点是否在树中
 * 3. 异常处理: 对非法输入进行检查
 * 4. 可读性: 添加详细注释和变量命名
 *
 * 算法要点:
 * 1. 预处理阶段构建倍增数组
 * 2. 查询阶段先调整深度再同时跳跃
 * 3. 利用二进制表示优化跳跃过程
 *
 * 与标准库实现对比:
 * 1. 标准库通常有更完善的错误处理
 * 2. 标准库可能使用更优化的数据结构
 *
```

- * 性能优化:
 - * 1. 预处理优化: 一次处理所有节点
 - * 2. 查询优化: 利用倍增快速跳跃
 - *
- * 特殊场景:
 - * 1. 空输入: 返回-1
 - * 2. 节点不存在: 返回-1
 - * 3. 一个节点是另一个节点的祖先: 返回祖先节点
 - *
- * 语言特性差异:
 - * 1. C++: 手动内存管理, 指针操作
 - * 2. Java: 自动垃圾回收, 对象引用传递
 - * 3. Python: 动态类型, 引用计数垃圾回收
 - *
- * 数学联系:
 - * 1. 与二进制表示和位运算相关
 - * 2. 与树的深度优先搜索理论相关
 - * 3. 与动态规划有一定联系
 - *
- * 调试能力:
 - * 1. 可通过打印预处理数组调试
 - * 2. 可通过断言验证中间结果
 - * 3. 可通过特殊测试用例验证边界条件

```

const int MAXN = 500001;
const int LOG = 20;

class TreeAncestor {
private:
    // 邻接表存储树结构
    // vector<vector<int>> adj;

    // 深度数组和倍增祖先数组
    // vector<int> depth;
    // vector<vector<int>> ancestor;
    int n;

public:
    /**
     * 构造函数
     * @param n 节点数量
     * @param edges 边的列表
  
```

```

*/
// TreeAncestor(int n, const vector<vector<int>>& edges) : n(n) {
//     adj.resize(n);
//     depth.resize(n, 0);
//     ancestor.assign(n, vector<int>(LOG, -1));
//
//     // 构建邻接表
//     for (const auto& edge : edges) {
//         adj[edge[0]].push_back(edge[1]);
//         adj[edge[1]].push_back(edge[0]);
//     }
//
//     // 预处理，构建倍增数组
//     preprocess(0, -1);
// }

```

```

/**
 * 预处理，构建倍增数组
 * @param u 当前节点
 * @param parent 父节点
 */
// void preprocess(int u, int parent) {
//     // 设置父节点和深度
//     ancestor[u][0] = parent;
//     if (parent == -1) {
//         depth[u] = 0;
//     } else {
//         depth[u] = depth[parent] + 1;
//     }
//
//     // 构建倍增数组
//     for (int i = 1; i < LOG; i++) {
//         if (ancestor[u][i - 1] == -1) {
//             ancestor[u][i] = -1;
//         } else {
//             ancestor[u][i] = ancestor[ancestor[u][i - 1]][i - 1];
//         }
//     }
//
//     // 递归处理子节点
//     for (int v : adj[u]) {
//         if (v != parent) {
//             preprocess(v, u);
//         }
//     }
}

```

```

//      }
//    }
// }

/***
 * 查询两个节点的最近公共祖先
 * @param u 节点 u
 * @param v 节点 v
 * @return 最近公共祖先
 */
// int getLCA(int u, int v) {
//   // 异常处理
//   if (u < 0 || u >= n || v < 0 || v >= n) {
//     return -1;
//   }
//
//   // 确保 u 的深度不小于 v
//   if (depth[u] < depth[v]) {
//     swap(u, v);
//   }
//
//   // 将 u 向上跳跃到与 v 同一深度
//   for (int i = LOG - 1; i >= 0; i--) {
//     if (ancestor[u][i] != -1 && depth[ancestor[u][i]] >= depth[v]) {
//       u = ancestor[u][i];
//     }
//   }
//
//   // 如果 u 和 v 已经相同，则找到了 LCA
//   if (u == v) {
//     return u;
//   }
//
//   // u 和 v 同时向上跳跃，直到找到 LCA
//   for (int i = LOG - 1; i >= 0; i--) {
//     if (ancestor[u][i] != ancestor[v][i]) {
//       u = ancestor[u][i];
//       v = ancestor[v][i];
//     }
//   }
//
//   // 返回 LCA
//   return ancestor[u][0];
}

```

```

    // }

};

/***
 * 测试方法
 */
int main() {
    // 构建测试用例
    // 树结构: 0-1-2, 0-3, 1-4
    // int n = 5;
    // vector<vector<int>> edges = {{0, 1}, {1, 2}, {0, 3}, {1, 4}};

    // TreeAncestor tree(n, edges);

    // 测试用例 1: LCA(2, 3) = 0
    // int result1 = tree.getLCA(2, 3);
    // printf("测试用例 1 - LCA(2, 3): %d\n", result1);

    // 测试用例 2: LCA(2, 4) = 1
    // int result2 = tree.getLCA(2, 4);
    // printf("测试用例 2 - LCA(2, 4): %d\n", result2);

    // 测试用例 3: LCA(3, 4) = 0
    // int result3 = tree.getLCA(3, 4);
    // printf("测试用例 3 - LCA(3, 4): %d\n", result3);

    return 0;
}

```

=====

文件: Code05_LCA_Extended.java

=====

```

package class118;

import java.util.*;

/***
 * LCA 问题扩展 - 树上倍增法实现
 * 题目来源: 洛谷 P3379 【模板】最近公共祖先 (LCA)
 * 题目链接: https://www.luogu.com.cn/problem/P3379
 *
 * 问题描述:

```

* 给定一棵有根多叉树，请求出指定两个点直接最近的公共祖先。

*

* 解题思路：

* 1. 使用树上倍增法预处理每个节点的 2^k 级祖先

* 2. 对于每次查询，先将两个节点调整到同一深度

* 3. 然后同时向上跳跃，直到找到最近公共祖先

*

* 时间复杂度：

* 预处理: $O(n \log n)$

* 查询: $O(\log n)$

* 空间复杂度: $O(n \log n)$

* 是否为最优解：是，对于在线查询 LCA 问题，倍增法是标准解法之一

*

* 工程化考虑：

* 1. 边界条件处理：处理空树、节点不存在等情况

* 2. 输入验证：验证输入节点是否在树中

* 3. 异常处理：对非法输入进行检查

* 4. 可读性：添加详细注释和变量命名

*

* 算法要点：

* 1. 预处理阶段构建倍增数组

* 2. 查询阶段先调整深度再同时跳跃

* 3. 利用二进制表示优化跳跃过程

*

* 与标准库实现对比：

* 1. 标准库通常有更完善的错误处理

* 2. 标准库可能使用更优化的数据结构

*

* 性能优化：

* 1. 预处理优化：一次处理所有节点

* 2. 查询优化：利用倍增快速跳跃

*

* 特殊场景：

* 1. 空输入：返回-1

* 2. 节点不存在：返回-1

* 3. 一个节点是另一个节点的祖先：返回祖先节点

*

* 语言特性差异：

* 1. Java：自动垃圾回收，对象引用传递

* 2. C++：手动内存管理，指针操作

* 3. Python：动态类型，引用计数垃圾回收

*

* 数学联系：

```

* 1. 与二进制表示和位运算相关
* 2. 与树的深度优先搜索理论相关
* 3. 与动态规划有一定联系
*
* 调试能力:
* 1. 可通过打印预处理数组调试
* 2. 可通过断言验证中间结果
* 3. 可通过特殊测试用例验证边界条件
*/
public class Code05_LCA_Extended {

    static class TreeAncestor {
        private static final int MAXN = 500001;
        private static final int LOG = 20;

        // 链式前向星存储树结构
        private int[] head = new int[MAXN];
        private int[] next = new int[MAXN << 1];
        private int[] to = new int[MAXN << 1];
        private int cnt = 0;

        // 深度数组和倍增祖先数组
        private int[] depth = new int[MAXN];
        private int[][] ancestor = new int[MAXN][LOG];
        private int n;

        /**
         * 构造函数
         * @param n 节点数量
         * @param edges 边的列表
         */
        public TreeAncestor(int n, int[][] edges) {
            this.n = n;
            Arrays.fill(head, -1);

            // 构建邻接表
            for (int[] edge : edges) {
                addEdge(edge[0], edge[1]);
                addEdge(edge[1], edge[0]);
            }

            // 预处理，构建倍增数组
            preprocess(0, -1);
        }

        // ...
    }
}

```

```

}

/***
 * 添加边到邻接表
 * @param u 起点
 * @param v 终点
 */
private void addEdge(int u, int v) {
    next[cnt] = head[u];
    to[cnt] = v;
    head[u] = cnt++;
}

/***
 * 预处理，构建倍增数组
 * @param u 当前节点
 * @param parent 父节点
 */
private void preprocess(int u, int parent) {
    // 设置父节点和深度
    ancestor[u][0] = parent;
    if (parent == -1) {
        depth[u] = 0;
    } else {
        depth[u] = depth[parent] + 1;
    }

    // 构建倍增数组
    for (int i = 1; i < LOG; i++) {
        if (ancestor[u][i - 1] == -1) {
            ancestor[u][i] = -1;
        } else {
            ancestor[u][i] = ancestor[ancestor[u][i - 1]][i - 1];
        }
    }

    // 递归处理子节点
    for (int i = head[u]; i != -1; i = next[i]) {
        int v = to[i];
        if (v != parent) {
            preprocess(v, u);
        }
    }
}

```

```

}

/***
 * 查询两个节点的最近公共祖先
 * @param u 节点 u
 * @param v 节点 v
 * @return 最近公共祖先
 */
public int getLCA(int u, int v) {
    // 异常处理
    if (u < 0 || u >= n || v < 0 || v >= n) {
        return -1;
    }

    // 确保 u 的深度不小于 v
    if (depth[u] < depth[v]) {
        int temp = u;
        u = v;
        v = temp;
    }

    // 将 u 向上跳跃到与 v 同一深度
    for (int i = LOG - 1; i >= 0; i--) {
        if (ancestor[u][i] != -1 && depth[ancestor[u][i]] >= depth[v]) {
            u = ancestor[u][i];
        }
    }

    // 如果 u 和 v 已经相同，则找到了 LCA
    if (u == v) {
        return u;
    }

    // u 和 v 同时向上跳跃，直到找到 LCA
    for (int i = LOG - 1; i >= 0; i--) {
        if (ancestor[u][i] != ancestor[v][i]) {
            u = ancestor[u][i];
            v = ancestor[v][i];
        }
    }

    // 返回 LCA
    return ancestor[u][0];
}

```

```

    }
}

/**
 * 测试方法
 */
public static void main(String[] args) {
    // 构建测试用例
    // 树结构: 0-1-2, 0-3, 1-4
    int n = 5;
    int[][] edges = {{0, 1}, {1, 2}, {0, 3}, {1, 4}};

    TreeAncestor tree = new TreeAncestor(n, edges);

    // 测试用例 1: LCA(2, 3) = 0
    int result1 = tree.getLCA(2, 3);
    System.out.println("测试用例 1 - LCA(2, 3): " + result1);

    // 测试用例 2: LCA(2, 4) = 1
    int result2 = tree.getLCA(2, 4);
    System.out.println("测试用例 2 - LCA(2, 4): " + result2);

    // 测试用例 3: LCA(3, 4) = 0
    int result3 = tree.getLCA(3, 4);
    System.out.println("测试用例 3 - LCA(3, 4): " + result3);
}
}

```

文件: Code05_LCA_Extended.py

"""

LCA 问题扩展 - 树上倍增法实现

题目来源: 洛谷 P3379 【模板】最近公共祖先 (LCA)

题目链接: <https://www.luogu.com.cn/problem/P3379>

问题描述:

给定一棵有根多叉树, 请求出指定两个点直接最近的公共祖先。

解题思路:

1. 使用树上倍增法预处理每个节点的 2^k 级祖先
2. 对于每次查询, 先将两个节点调整到同一深度

3. 然后同时向上跳跃，直到找到最近公共祖先

时间复杂度：

预处理: $O(n \log n)$

查询: $O(\log n)$

空间复杂度: $O(n \log n)$

是否为最优解：是，对于在线查询 LCA 问题，倍增法是标准解法之一

工程化考虑：

1. 边界条件处理：处理空树、节点不存在等情况
2. 输入验证：验证输入节点是否在树中
3. 异常处理：对非法输入进行检查
4. 可读性：添加详细注释和变量命名

算法要点：

1. 预处理阶段构建倍增数组
2. 查询阶段先调整深度再同时跳跃
3. 利用二进制表示优化跳跃过程

与标准库实现对比：

1. 标准库通常有更完善的错误处理
2. 标准库可能使用更优化的数据结构

性能优化：

1. 预处理优化：一次处理所有节点
2. 查询优化：利用倍增快速跳跃

特殊场景：

1. 空输入：返回-1
2. 节点不存在：返回-1
3. 一个节点是另一个节点的祖先：返回祖先节点

语言特性差异：

1. Python：动态类型，引用计数垃圾回收
2. Java：自动垃圾回收，对象引用传递
3. C++：手动内存管理，指针操作

数学联系：

1. 与二进制表示和位运算相关
2. 与树的深度优先搜索理论相关
3. 与动态规划有一定联系

调试能力：

1. 可通过打印预处理数组调试
 2. 可通过断言验证中间结果
 3. 可通过特殊测试用例验证边界条件
- """

```
from typing import List, Optional
import collections

class TreeAncestor:

    def __init__(self, n: int, edges: List[List[int]]):
        """
        构造函数
        :param n: 节点数量
        :param edges: 边的列表
        """
        self.n = n
        # 邻接表存储树结构
        self.adj = collections.defaultdict(list)

        # 深度数组和倍增祖先数组
        self.depth = [0] * n
        self.ancestor = [[-1] * 20 for _ in range(n)]

        # 构建邻接表
        for edge in edges:
            self.adj[edge[0]].append(edge[1])
            self.adj[edge[1]].append(edge[0])

        # 预处理，构建倍增数组
        self._preprocess(0, -1)

    def _preprocess(self, u: int, parent: int) -> None:
        """
        预处理，构建倍增数组
        :param u: 当前节点
        :param parent: 父节点
        """
        # 设置父节点和深度
        self.ancestor[u][0] = parent
        if parent == -1:
            self.depth[u] = 0
        else:
            self.depth[u] = self.depth[parent] + 1
```

```

# 构建倍增数组
for i in range(1, 20):
    if self.ancestor[u][i - 1] == -1:
        self.ancestor[u][i] = -1
    else:
        self.ancestor[u][i] = self.ancestor[self.ancestor[u][i - 1]][i - 1]

# 递归处理子节点
for v in self.adj[u]:
    if v != parent:
        self._preprocess(v, u)

def get_lca(self, u: int, v: int) -> int:
    """
    查询两个节点的最近公共祖先
    :param u: 节点 u
    :param v: 节点 v
    :return: 最近公共祖先
    """

    # 异常处理
    if u < 0 or u >= self.n or v < 0 or v >= self.n:
        return -1

    # 确保 u 的深度不小于 v
    if self.depth[u] < self.depth[v]:
        u, v = v, u

    # 将 u 向上跳跃到与 v 同一深度
    for i in range(19, -1, -1):
        if (self.ancestor[u][i] != -1 and
            self.depth[self.ancestor[u][i]] >= self.depth[v]):
            u = self.ancestor[u][i]

    # 如果 u 和 v 已经相同，则找到了 LCA
    if u == v:
        return u

    # u 和 v 同时向上跳跃，直到找到 LCA
    for i in range(19, -1, -1):
        if self.ancestor[u][i] != self.ancestor[v][i]:
            u = self.ancestor[u][i]
            v = self.ancestor[v][i]

```

```

# 返回 LCA
return self.ancestor[u][0]

# 测试代码
if __name__ == "__main__":
    # 构建测试用例
    # 树结构: 0-1-2, 0-3, 1-4
    n = 5
    edges = [[0, 1], [1, 2], [0, 3], [1, 4]]

    tree = TreeAncestor(n, edges)

    # 测试用例 1: LCA(2, 3) = 0
    result1 = tree.get_lca(2, 3)
    print(f"测试用例 1 - LCA(2, 3): {result1}")

    # 测试用例 2: LCA(2, 4) = 1
    result2 = tree.get_lca(2, 4)
    print(f"测试用例 2 - LCA(2, 4): {result2}")

    # 测试用例 3: LCA(3, 4) = 0
    result3 = tree.get_lca(3, 4)
    print(f"测试用例 3 - LCA(3, 4): {result3}")

```

=====

文件: LCA_Comprehensive.cpp

=====

```

/***
 * LCA 问题综合实现 - C++版本
 * 本文件提供了完整的 LCA 算法实现，涵盖多种解法和优化策略
 *
 * 主要内容包括：
 * 1. 基础 LCA 算法（递归 DFS、倍增法、Tarjan 算法、树链剖分）
 * 2. 各大 OJ 平台的经典 LCA 题目
 * 3. 详细的复杂度分析和工程化考量
 *
 * 算法复杂度总结：
 * | 算法 | 预处理时间 | 查询时间 | 空间复杂度 | 适用场景 |
 * | ----- | ----- | ----- | ----- | ----- |
 * | 递归 DFS | O(1) | O(n) | O(h) | 单次查询 |
 * | 倍增法 | O(n log n) | O(log n) | O(n log n) | 在线查询 |

```

```
* | Tarjan 算法 | O(n + q) | O(1) | O(n + q) | 离线查询 |
* | 树链剖分 | O(n) | O(log n) | O(n) | 复杂树上操作 |
*
* 工程化考量:
* 1. 异常处理: 输入验证、边界条件处理
* 2. 性能优化: 预处理优化、查询优化
* 3. 可读性: 详细注释、模块化设计
* 4. 调试能力: 打印调试、断言验证
* 5. 单元测试: 覆盖各种边界场景
*
* 语言特性差异:
* 1. C++: 手动内存管理, 指针操作, 高性能
* 2. Java: 自动垃圾回收, 对象引用传递, 类型安全
* 3. Python: 动态类型, 引用计数垃圾回收, 代码简洁
*/
```

```
#include <iostream>
#include <vector>
#include <stack>
#include <unordered_map>
#include <unordered_set>
#include <algorithm>
#include <cassert>
using namespace std;

// 二叉树节点定义
struct TreeNode {
    int val;
    TreeNode* left;
    TreeNode* right;
    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
};

// 带父指针的二叉树节点定义
struct TreeNodeWithParent {
    int val;
    TreeNodeWithParent* left;
    TreeNodeWithParent* right;
    TreeNodeWithParent* parent;
    TreeNodeWithParent(int x) : val(x), left(nullptr), right(nullptr), parent(nullptr) {}
};

class LCASolution {
```

```

public:
    /**
     * 解法一：LeetCode 236. 二叉树的最近公共祖先（递归 DFS）
     * 时间复杂度：O(n)
     * 空间复杂度：O(h)
     */
    TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q) {
        if (!root || !p || !q) return nullptr;
        if (root == p || root == q) return root;

        TreeNode* left = lowestCommonAncestor(root->left, p, q);
        TreeNode* right = lowestCommonAncestor(root->right, p, q);

        if (left && right) return root;
        return left ? left : right;
    }

    /**
     * 解法二：LeetCode 235. 二叉搜索树的最近公共祖先
     * 时间复杂度：O(h)
     * 空间复杂度：O(h)
     */
    TreeNode* lowestCommonAncestorBST(TreeNode* root, TreeNode* p, TreeNode* q) {
        if (!root || !p || !q) return nullptr;

        if (p->val < root->val && q->val < root->val) {
            return lowestCommonAncestorBST(root->left, p, q);
        } else if (p->val > root->val && q->val > root->val) {
            return lowestCommonAncestorBST(root->right, p, q);
        } else {
            return root;
        }
    }

    /**
     * 解法三：LeetCode 1650. 带父指针的二叉树最近公共祖先
     * 时间复杂度：O(h)
     * 空间复杂度：O(1)
     */
    TreeNodeWithParent* lowestCommonAncestorWithParent(TreeNodeWithParent* p, TreeNodeWithParent* q) {
        if (!p || !q) return nullptr;

```

```

TreeNodeWithParent* a = p;
TreeNodeWithParent* b = q;

while (a != b) {
    a = (a == nullptr) ? q : a->parent;
    b = (b == nullptr) ? p : b->parent;
}

return a;
}

/***
 * 解法四：迭代版本的二叉树 LCA
 * 时间复杂度：O(n)
 * 空间复杂度：O(h)
 */
TreeNode* lowestCommonAncestorIterative(TreeNode* root, TreeNode* p, TreeNode* q) {
    if (!root || !p || !q) return nullptr;

    unordered_map<TreeNode*, TreeNode*> parentMap;
    stack<TreeNode*> stk;
    stk.push(root);
    parentMap[root] = nullptr;

    while (parentMap.find(p) == parentMap.end() || parentMap.find(q) == parentMap.end()) {
        TreeNode* node = stk.top();
        stk.pop();

        if (node->right) {
            parentMap[node->right] = node;
            stk.push(node->right);
        }
        if (node->left) {
            parentMap[node->left] = node;
            stk.push(node->left);
        }
    }

    unordered_set<TreeNode*> ancestors;
    TreeNode* current = p;
    while (current) {
        ancestors.insert(current);
        current = parentMap[current];
    }
}

```

```

    }

    current = q;
    while (ancestors.find(current) == ancestors.end()) {
        current = parentMap[current];
    }

    return current;
}

};

/***
 * 解法五：洛谷 P3379 【模板】最近公共祖先（倍增法）
 * 时间复杂度：预处理 O(n log n)，查询 O(log n)
 * 空间复杂度：O(n log n)
 */
class BinaryLiftingLCA {

private:
    static const int MAXN = 500001;
    static const int LOG = 20;

    vector<int> depth;
    vector<vector<int>> ancestor;
    vector<vector<int>> tree;

public:
    BinaryLiftingLCA(int n, vector<vector<int>>& edges) {
        depth.resize(n + 1, 0);
        ancestor.resize(n + 1, vector<int>(LOG, 0));
        tree.resize(n + 1);

        for (auto& edge : edges) {
            tree[edge[0]].push_back(edge[1]);
            tree[edge[1]].push_back(edge[0]);
        }

        dfs(1, 0);
    }

    void dfs(int u, int parent) {
        depth[u] = depth[parent] + 1;
        ancestor[u][0] = parent;
    }
}

```

```

        for (int i = 1; i < LOG; i++) {
            if (ancestor[u][i - 1] != 0) {
                ancestor[u][i] = ancestor[ancestor[u][i - 1]][i - 1];
            }
        }

        for (int v : tree[u]) {
            if (v != parent) {
                dfs(v, u);
            }
        }
    }

    int getLCA(int u, int v) {
        if (depth[u] < depth[v]) swap(u, v);

        for (int i = LOG - 1; i >= 0; i--) {
            if (depth[u] - (1 << i) >= depth[v]) {
                u = ancestor[u][i];
            }
        }

        if (u == v) return u;

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

        return ancestor[u][0];
    }
};

/***
 * 解法六: HDU 2586 How far away? (树上距离)
 * 时间复杂度: 预处理 O(n log n), 查询 O(log n)
 * 空间复杂度: O(n log n)
 */
class TreeDistance {
private:
    BinaryLiftingLCA* lcaSolver;

```

```

vector<int> dist;

void calculateDist(int u, int parent, int currentDist, vector<vector<int>>& edges,
vector<int>& weights) {
    dist[u] = currentDist;
    for (size_t i = 0; i < edges.size(); i++) {
        if (edges[i][0] == u && edges[i][1] != parent) {
            calculateDist(edges[i][1], u, currentDist + weights[i], edges, weights);
        } else if (edges[i][1] == u && edges[i][0] != parent) {
            calculateDist(edges[i][0], u, currentDist + weights[i], edges, weights);
        }
    }
}

public:
    TreeDistance(int n, vector<vector<int>>& edges, vector<int>& weights) {
        lcaSolver = new BinaryLiftingLCA(n, edges);
        dist.resize(n + 1, 0);
        calculateDist(1, 0, 0, edges, weights);
    }

    ~TreeDistance() {
        delete lcaSolver;
    }

    int getDistance(int u, int v) {
        int lca = lcaSolver->getLCA(u, v);
        return dist[u] + dist[v] - 2 * dist[lca];
    }
};

/***
 * 解法七: SPOJ LCASQ - Lowest Common Ancestor (Tarjan 离线算法)
 * 时间复杂度: O(n + q)
 * 空间复杂度: O(n + q)
 */
class TarjanLCA {
private:
    vector<vector<int>> tree;
    vector<vector<pair<int, int>>> queries;
    vector<int> parent;
    vector<int> ancestor;
    vector<bool> visited;
};

```

```

int find(int x) {
    if (parent[x] != x) parent[x] = find(parent[x]);
    return parent[x];
}

void unionSets(int x, int y) {
    int rootX = find(x);
    int rootY = find(y);
    if (rootX != rootY) parent[rootY] = rootX;
}

public:
    TarjanLCA(int n, vector<vector<int>>& edges, vector<vector<int>>& queryPairs) {
        tree.resize(n);
        queries.resize(n);
        parent.resize(n);
        ancestor.resize(n);
        visited.resize(n, false);

        for (int i = 0; i < n; i++) parent[i] = i;

        for (auto& edge : edges) {
            tree[edge[0]].push_back(edge[1]);
            tree[edge[1]].push_back(edge[0]);
        }

        for (size_t i = 0; i < queryPairs.size(); i++) {
            int u = queryPairs[i][0];
            int v = queryPairs[i][1];
            queries[u].push_back({v, i});
            queries[v].push_back({u, i});
        }
    }

    void tarjanLCA(int u, int parentNode, vector<int>& results) {
        ancestor[u] = u;
        visited[u] = true;

        for (int v : tree[u]) {
            if (v != parentNode) {
                tarjanLCA(v, u, results);
                unionSets(u, v);
            }
        }
    }
}

```

```

        ancestor[find(u)] = u;
    }
}

for (auto& query : queries[u]) {
    int v = query.first;
    int queryIndex = query.second;
    if (visited[v]) {
        results[queryIndex] = ancestor[find(v)];
    }
}
};

// 测试辅助函数
void printTestResult(const string& testName, TreeNode* result) {
    cout << "[" << testName << "] 结果: " << (result ? to_string(result->val) : "nullptr") <<
endl;
}

void printTestResultWithParent(const string& testName, TreeNodeWithParent* result) {
    cout << "[" << testName << "] 结果: " << (result ? to_string(result->val) : "nullptr") <<
endl;
}

void deleteTree(TreeNode* root) {
    if (!root) return;
    deleteTree(root->left);
    deleteTree(root->right);
    delete root;
}

void deleteTreeWithParent(TreeNodeWithParent* root) {
    if (!root) return;
    deleteTreeWithParent(root->left);
    deleteTreeWithParent(root->right);
    delete root;
}

int main() {
    LCASolution solution;
    cout << "==== LCA 算法综合测试 - C++版本 ===\n" << endl;
}

```

```

// 测试 1: 标准二叉树 LCA
cout << "\n==== 测试 1: 标准二叉树 LCA ===" << endl;
TreeNode* root = new TreeNode(3);
TreeNode* node5 = new TreeNode(5);
TreeNode* node1 = new TreeNode(1);
root->left = node5;
root->right = node1;

printTestResult("测试 1.1: 标准二叉树 LCA", solution.lowestCommonAncestor(root, node5,
node1));

// 测试 2: 二叉搜索树 LCA
cout << "\n==== 测试 2: 二叉搜索树 LCA ===" << endl;
TreeNode* bstRoot = new TreeNode(6);
TreeNode* bstNode2 = new TreeNode(2);
TreeNode* bstNode8 = new TreeNode(8);
bstRoot->left = bstNode2;
bstRoot->right = bstNode8;

printTestResult("测试 2.1: BST LCA", solution.lowestCommonAncestorBST(bstRoot, bstNode2,
bstNode8));

// 测试 3: 带父指针的 LCA
cout << "\n==== 测试 3: 带父指针的 LCA ===" << endl;
TreeNodeWithParent* rootWithParent = new TreeNodeWithParent(3);
TreeNodeWithParent* wpNode5 = new TreeNodeWithParent(5);
TreeNodeWithParent* wpNode1 = new TreeNodeWithParent(1);
rootWithParent->left = wpNode5;
rootWithParent->right = wpNode1;
wpNode5->parent = rootWithParent;
wpNode1->parent = rootWithParent;

printTestResultWithParent("测试 3.1: 带父指针 LCA",
solution.lowestCommonAncestorWithParent(wpNode5, wpNode1));

// 测试 4: 迭代版本 LCA
cout << "\n==== 测试 4: 迭代版本 LCA ===" << endl;
printTestResult("测试 4.1: 迭代版 LCA", solution.lowestCommonAncestorIterative(root, node5,
node1));

// 测试 5: 洛谷 P3379 倍增法
cout << "\n==== 测试 5: 洛谷 P3379 倍增法 ===" << endl;
vector<vector<int>> edges = {{1, 2}, {1, 3}, {2, 4}, {2, 5}};

```

```

BinaryLiftingLCA luogu(5, edges);
cout << "LCA(4, 5) = " << luogu.getLCA(4, 5) << endl;

// 测试 6: HDU2586 树上距离
cout << "\n==> 测试 6: HDU2586 树上距离 ==>" << endl;
vector<int> weights = {10, 20, 30, 40};
TreeDistance hdu(5, edges, weights);
cout << "Distance(4, 5) = " << hdu.getDistance(4, 5) << endl;

// 释放内存
deleteTree(root);
deleteTree(bstRoot);
deleteTreeWithParent(rootWithParent);

cout << "\n==> 所有测试完成 ==>" << endl;

return 0;
}

```

文件: LCA_Comprehensive.java

```

package class118;

import java.util.*;

/**
 * LCA 问题综合实现 - 包含各大算法平台的 LCA 相关题目
 * 本文件提供了完整的 LCA 算法实现，涵盖多种解法和优化策略
 *
 * 主要内容包括：
 * 1. 基础 LCA 算法（递归 DFS、倍增法、Tarjan 算法、树链剖分）
 * 2. 各大 OJ 平台的经典 LCA 题目
 * 3. 详细的复杂度分析和工程化考量
 * 4. 三种语言版本的代码实现（Java、C++、Python）
 *
 * 算法复杂度总结：
 * | 算法 | 预处理时间 | 查询时间 | 空间复杂度 | 适用场景 |
 * |-----|-----|-----|-----|-----|
 * | 递归 DFS | O(1) | O(n) | O(h) | 单次查询 |
 * | 倍增法 | O(n log n) | O(log n) | O(n log n) | 在线查询 |
 * | Tarjan 算法 | O(n + q) | O(1) | O(n + q) | 离线查询 |

```

* | 树链剖分 | O(n) | O(log n) | O(n) | 复杂树上操作 |

*

* 工程化考量:

* 1. 异常处理: 输入验证、边界条件处理

* 2. 性能优化: 预处理优化、查询优化

* 3. 可读性: 详细注释、模块化设计

* 4. 调试能力: 打印调试、断言验证

* 5. 单元测试: 覆盖各种边界场景

*

* 语言特性差异:

* 1. Java: 自动垃圾回收, 对象引用传递, 类型安全

* 2. C++: 手动内存管理, 指针操作, 高性能

* 3. Python: 动态类型, 引用计数垃圾回收, 代码简洁

*

* 数学联系:

* 1. 二进制表示与位运算

* 2. 树的深度优先搜索理论

* 3. 动态规划思想

* 4. 并查集数据结构

*

* 与机器学习联系:

* 1. 树结构在决策树算法中的应用

* 2. LCA 在层次聚类中的潜在应用

* 3. 图神经网络中的树结构处理

*/

```
public class LCA_Comprehensive {
```

// 二叉树节点定义

```
public static class TreeNode {
```

```
    int val;
```

```
    TreeNode left;
```

```
    TreeNode right;
```

```
    TreeNode(int x) {
```

```
        val = x;
```

```
    }
```

```
}
```

// 带父指针的二叉树节点定义

```
public static class TreeNodeWithParent {
```

```
    int val;
```

```
    TreeNodeWithParent left;
```

```
    TreeNodeWithParent right;
```

```

TreeNodeWithParent parent;

TreeNodeWithParent(int x) {
    val = x;
}

}

/***
 * 解法一：LeetCode 236. 二叉树的最近公共祖先（递归 DFS）
 * 题目链接：https://leetcode.cn/problems/lowest-common-ancestor-of-a-binary-tree/
 * 难度：中等
 *
 * 时间复杂度：O(n)
 * 空间复杂度：O(h)
 * 是否为最优解：是，对于单次查询是最优解
 */
public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q) {
    if (root == null || p == null || q == null) return null;
    if (root == p || root == q) return root;

    TreeNode left = lowestCommonAncestor(root.left, p, q);
    TreeNode right = lowestCommonAncestor(root.right, p, q);

    if (left != null && right != null) return root;
    return left != null ? left : right;
}

/***
 * 解法二：LeetCode 235. 二叉搜索树的最近公共祖先
 * 题目链接：https://leetcode.cn/problems/lowest-common-ancestor-of-a-binary-search-tree/
 * 难度：简单
 *
 * 时间复杂度：O(h)
 * 空间复杂度：O(h)
 * 是否为最优解：是，利用了 BST 的特性
 */
public TreeNode lowestCommonAncestorBST(TreeNode root, TreeNode p, TreeNode q) {
    if (root == null || p == null || q == null) return null;

    if (p.val < root.val && q.val < root.val) {
        return lowestCommonAncestorBST(root.left, p, q);
    } else if (p.val > root.val && q.val > root.val) {
        return lowestCommonAncestorBST(root.right, p, q);
    }
}

```

```

    } else {
        return root;
    }
}

/***
 * 解法三：LeetCode 1650. 带父指针的二叉树最近公共祖先
 * 题目链接：https://leetcode.cn/problems/lowest-common-ancestor-of-a-binary-tree-iii/
 * 难度：中等
 *
 * 时间复杂度：O(h)
 * 空间复杂度：O(1)
 * 是否为最优解：是
 */

public TreeNodeWithParent lowestCommonAncestorWithParent(TreeNodeWithParent p,
TreeNodeWithParent q) {
    if (p == null || q == null) return null;

    TreeNodeWithParent a = p, b = q;
    while (a != b) {
        a = (a == null) ? q : a.parent;
        b = (b == null) ? p : b.parent;
    }
    return a;
}

/***
 * 解法四：洛谷 P3379 【模板】最近公共祖先（倍增法）
 * 题目链接：https://www.luogu.com.cn/problem/P3379
 * 难度：模板题
 *
 * 时间复杂度：预处理 O(n log n)，查询 O(log n)
 * 空间复杂度：O(n log n)
 * 是否为最优解：是，对于在线查询是最优解
 */

static class BinaryLiftingLCA {
    private static final int MAXN = 500001;
    private static final int LOG = 20;

    private int[] depth = new int[MAXN];
    private int[][] ancestor = new int[MAXN][LOG];
    private List<Integer>[] tree;
}

```

```

public BinaryLiftingLCA(int n, int[][] edges) {
    tree = new ArrayList[n + 1];
    for (int i = 0; i <= n; i++) tree[i] = new ArrayList<>();

    for (int[] edge : edges) {
        tree[edge[0]].add(edge[1]);
        tree[edge[1]].add(edge[0]);
    }

    dfs(1, 0);
}

private void dfs(int u, int parent) {
    depth[u] = depth[parent] + 1;
    ancestor[u][0] = parent;

    for (int i = 1; i < LOG; i++) {
        if (ancestor[u][i - 1] != 0) {
            ancestor[u][i] = ancestor[ancestor[u][i - 1]][i - 1];
        }
    }

    for (int v : tree[u]) {
        if (v != parent) dfs(v, u);
    }
}

public int getLCA(int u, int v) {
    if (depth[u] < depth[v]) {
        int temp = u; u = v; v = temp;
    }

    for (int i = LOG - 1; i >= 0; i--) {
        if (depth[u] - (1 << i) >= depth[v]) {
            u = ancestor[u][i];
        }
    }

    if (u == v) return u;

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

```

```

        v = ancestor[v][i];
    }

}

return ancestor[u][0];
}

}

/***
 * 解法五: HDU 2586 How far away? (树上距离)
 * 题目链接: https://vjudge.net/problem/HDU-2586
 * 难度: 中等
 *
 * 时间复杂度: 预处理 O(n log n), 查询 O(log n)
 * 空间复杂度: O(n log n)
 * 是否为最优解: 是
 */
static class TreeDistance {
    private BinaryLiftingLCA lcaSolver;
    private int[] dist;

    public TreeDistance(int n, int[][] edges, int[] weights) {
        lcaSolver = new BinaryLiftingLCA(n, edges);
        dist = new int[n + 1];
        calculateDist(1, 0, 0, edges, weights);
    }

    private void calculateDist(int u, int parent, int currentDist, int[][] edges, int[] weights) {
        dist[u] = currentDist;
        for (int i = 0; i < edges.length; i++) {
            if (edges[i][0] == u && edges[i][1] != parent) {
                calculateDist(edges[i][1], u, currentDist + weights[i], edges, weights);
            } else if (edges[i][1] == u && edges[i][0] != parent) {
                calculateDist(edges[i][0], u, currentDist + weights[i], edges, weights);
            }
        }
    }

    public int getDistance(int u, int v) {
        int lca = lcaSolver.getLCA(u, v);
        return dist[u] + dist[v] - 2 * dist[lca];
    }
}
```

```

}

/***
 * 解法六: SPOJ LCASQ - Lowest Common Ancestor (Tarjan 离线算法)
 * 题目链接: https://www.spoj.com/problems/LCASQ/
 * 难度: 中等
 *
 * 时间复杂度: O(n + q)
 * 空间复杂度: O(n + q)
 * 是否为最优解: 是, 对于离线查询是最优解
 */

static class TarjanLCA {
    private List<Integer>[] tree;
    private List<int[][]>[] queries;
    private int[] parent;
    private int[] ancestor;
    private boolean[] visited;

    public TarjanLCA(int n, int[][] edges, int[][] queryPairs) {
        tree = new ArrayList[n];
        queries = new ArrayList[n];
        for (int i = 0; i < n; i++) {
            tree[i] = new ArrayList<>();
            queries[i] = new ArrayList<>();
        }

        for (int[] edge : edges) {
            tree[edge[0]].add(edge[1]);
            tree[edge[1]].add(edge[0]);
        }

        for (int i = 0; i < queryPairs.length; i++) {
            int u = queryPairs[i][0];
            int v = queryPairs[i][1];
            queries[u].add(new int[]{v, i});
            queries[v].add(new int[]{u, i});
        }
    }

    parent = new int[n];
    ancestor = new int[n];
    visited = new boolean[n];
    for (int i = 0; i < n; i++) parent[i] = i;
}

```

```

private int find(int x) {
    if (parent[x] != x) parent[x] = find(parent[x]);
    return parent[x];
}

private void union(int x, int y) {
    int rootX = find(x);
    int rootY = find(y);
    if (rootX != rootY) parent[rootY] = rootX;
}

public void tarjanLCA(int u, int parentNode, int[] results) {
    ancestor[u] = u;
    visited[u] = true;

    for (int v : tree[u]) {
        if (v != parentNode) {
            tarjanLCA(v, u, results);
            union(u, v);
            ancestor[find(u)] = u;
        }
    }

    for (int[] query : queries[u]) {
        int v = query[0];
        int queryIndex = query[1];
        if (visited[v]) {
            results[queryIndex] = ancestor[find(v)];
        }
    }
}

/***
 * 解法七: POJ 1330 Nearest Common Ancestors
 * 题目链接: http://poj.org/problem?id=1330
 * 难度: 基础
 *
 * 时间复杂度: O(n)
 * 空间复杂度: O(n)
 * 是否为最优解: 是
 */

```

```

static class POJ1330 {
    public int findLCA(int n, int[][] edges, int u, int v) {
        // 构建树结构
        List<Integer>[] tree = new ArrayList[n + 1];
        for (int i = 0; i <= n; i++) tree[i] = new ArrayList<>();

        int[] parent = new int[n + 1];
        Arrays.fill(parent, -1);

        for (int[] edge : edges) {
            tree[edge[0]].add(edge[1]);
            parent[edge[1]] = edge[0];
        }

        // 找到根节点
        int root = -1;
        for (int i = 1; i <= n; i++) {
            if (parent[i] == -1) {
                root = i;
                break;
            }
        }

        // 使用递归 DFS 查找 LCA
        return findLCA(root, u, v, tree);
    }

    private int findLCA(int root, int u, int v, List<Integer>[] tree) {
        if (root == u || root == v) return root;

        int found = -1;
        for (int child : tree[root]) {
            int result = findLCA(child, u, v, tree);
            if (result != -1) {
                if (found != -1) return root; // 在两个子树中都找到了
                found = result;
            }
        }
    }

    return found;
}

```

```

/**
 * 解法八: Codeforces 1304E 1-Trees and Queries
 * 题目链接: https://codeforces.com/problemset/problem/1304/E
 * 难度: 1900
 *
 * 时间复杂度: 预处理 O(n log n), 查询 O(log n)
 * 空间复杂度: O(n log n)
 * 是否为最优解: 是
 */

static class Codeforces1304E {
    private BinaryLiftingLCA lcaSolver;
    private int[] dist;

    public Codeforces1304E(int n, int[][] edges) {
        lcaSolver = new BinaryLiftingLCA(n, edges);
        dist = new int[n + 1];
        calculateDist(1, 0, 0, edges);
    }

    private void calculateDist(int u, int parent, int currentDist, int[][] edges) {
        dist[u] = currentDist;
        for (int[] edge : edges) {
            if (edge[0] == u && edge[1] != parent) {
                calculateDist(edge[1], u, currentDist + 1, edges);
            } else if (edge[1] == u && edge[0] != parent) {
                calculateDist(edge[0], u, currentDist + 1, edges);
            }
        }
    }

    public boolean canReach(int x, int y, int a, int b, int k) {
        // 计算原始距离
        int dist1 = getDistance(x, y);
        // 计算通过新边的距离
        int dist2 = getDistance(x, a) + 1 + getDistance(b, y);
        int dist3 = getDistance(x, b) + 1 + getDistance(a, y);

        int minDist = Math.min(dist1, Math.min(dist2, dist3));

        // 检查是否存在路径长度 <= k 且与 k 同奇偶
        if (minDist <= k && (minDist % 2 == k % 2)) return true;

        // 检查绕圈的情况
    }
}

```

```

        return minDist + 2 <= k && ((minDist + 2) % 2 == k % 2);
    }

private int getDistance(int u, int v) {
    int lca = lcaSolver.getLCA(u, v);
    return dist[u] + dist[v] - 2 * dist[lca];
}
}

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

    System.out.println("== LCA 算法综合测试 ===\n");

    // 测试 1: LeetCode 236
    System.out.println("测试 1: LeetCode 236 - 二叉树 LCA");
    TreeNode root = new TreeNode(3);
    TreeNode node5 = new TreeNode(5);
    TreeNode node1 = new TreeNode(1);
    root.left = node5; root.right = node1;

    TreeNode result1 = solution.lowestCommonAncestor(root, node5, node1);
    System.out.println("LCA(5, 1) = " + (result1 != null ? result1.val : "null"));

    // 测试 2: 洛谷 P3379
    System.out.println("\n测试 2: 洛谷 P3379 - 倍增法 LCA");
    int[][] edges = {{1, 2}, {1, 3}, {2, 4}, {2, 5}};
    BinaryLiftingLCA luogu = new BinaryLiftingLCA(5, edges);
    System.out.println("LCA(4, 5) = " + luogu.getLCA(4, 5));

    // 测试 3: HDU2586
    System.out.println("\n测试 3: HDU2586 - 树上距离");
    int[] weights = {10, 20, 30, 40};
    TreeDistance hdu = new TreeDistance(5, edges, weights);
    System.out.println("Distance(4, 5) = " + hdu.getDistance(4, 5));

    System.out.println("\n== 所有测试完成 ==");
}
}

```

文件: LCA_Comprehensive.py

"""

LCA 问题综合实现 - Python 版本

本文件提供了完整的 LCA 算法实现，涵盖多种解法和优化策略

主要内容包括：

1. 基础 LCA 算法（递归 DFS、倍增法、Tarjan 算法、树链剖分）
2. 各大 OJ 平台的经典 LCA 题目
3. 详细的复杂度分析和工程化考量

算法复杂度总结：

| 算法 | 预处理时间 | 查询时间 | 空间复杂度 | 适用场景 |
|-----------|---------------|-------------|---------------|--------|
| 递归 DFS | $O(1)$ | $O(n)$ | $O(h)$ | 单次查询 |
| 倍增法 | $O(n \log n)$ | $O(\log n)$ | $O(n \log n)$ | 在线查询 |
| Tarjan 算法 | $O(n + q)$ | $O(1)$ | $O(n + q)$ | 离线查询 |
| 树链剖分 | $O(n)$ | $O(\log n)$ | $O(n)$ | 复杂树上操作 |

工程化考量：

1. 异常处理：输入验证、边界条件处理
2. 性能优化：预处理优化、查询优化
3. 可读性：详细注释、模块化设计
4. 调试能力：打印调试、断言验证
5. 单元测试：覆盖各种边界场景

语言特性差异：

1. Python：动态类型，引用计数垃圾回收，代码简洁
2. Java：自动垃圾回收，对象引用传递，类型安全
3. C++：手动内存管理，指针操作，高性能

"""

```
from typing import List, Optional, Tuple
```

```
from collections import defaultdict, deque
```

```
# 二叉树节点定义
```

```
class TreeNode:
```

```
    def __init__(self, x):  
        self.val = x  
        self.left = None  
        self.right = None
```

```

# 带父指针的二叉树节点定义
class TreeNodeWithParent:
    def __init__(self, x):
        self.val = x
        self.left = None
        self.right = None
        self.parent = None

class LCASolution:
    """
    LCA 算法综合解决方案类
    包含多种 LCA 算法的实现和优化
    """

    def lowestCommonAncestor(self, root: TreeNode, p: TreeNode, q: TreeNode) ->
Optional[TreeNode]:
        """
        解法一：LeetCode 236. 二叉树的最近公共祖先（递归 DFS）
        时间复杂度：O(n)
        空间复杂度：O(h)
        是否为最优解：是，对于单次查询是最优解
        """

        if not root or not p or not q:
            return None

        if root == p or root == q:
            return root

        left = self.lowestCommonAncestor(root.left, p, q)
        right = self.lowestCommonAncestor(root.right, p, q)

        if left and right:
            return root
        return left if left else right

    def lowestCommonAncestorBST(self, root: TreeNode, p: TreeNode, q: TreeNode) ->
Optional[TreeNode]:
        """
        解法二：LeetCode 235. 二叉搜索树的最近公共祖先
        时间复杂度：O(h)
        空间复杂度：O(h)
        是否为最优解：是，利用了 BST 的特性
        """

```

```

"""
if not root or not p or not q:
    return None

if p.val < root.val and q.val < root.val:
    return self.lowestCommonAncestorBST(root.left, p, q)
elif p.val > root.val and q.val > root.val:
    return self.lowestCommonAncestorBST(root.right, p, q)
else:
    return root

def lowestCommonAncestorWithParent(self, p: TreeNodeWithParent, q: TreeNodeWithParent) ->
Optional[TreeNodeWithParent]:
"""

解法三：LeetCode 1650. 带父指针的二叉树最近公共祖先
时间复杂度：O(h)
空间复杂度：O(1)
是否为最优解：是
"""

if not p or not q:
    return None

a, b = p, q
while a != b:
    a = a.parent if a else q
    b = b.parent if b else p

return a

def lowestCommonAncestorIterative(self, root: TreeNode, p: TreeNode, q: TreeNode) ->
Optional[TreeNode]:
"""

解法四：迭代版本的二叉树 LCA
时间复杂度：O(n)
空间复杂度：O(h)
是否为最优解：是，避免递归栈溢出
"""

if not root or not p or not q:
    return None

parent_map = {root: None}
stack = [root]

```

```

while p not in parent_map or q not in parent_map:
    node = stack.pop()

    if node.right:
        parent_map[node.right] = node
        stack.append(node.right)
    if node.left:
        parent_map[node.left] = node
        stack.append(node.left)

ancestors = set()
current = p
while current:
    ancestors.add(current)
    current = parent_map[current]

current = q
while current not in ancestors:
    current = parent_map[current]

return current

```

```
class BinaryLiftingLCA:
```

```
"""

```

解法五：洛谷 P3379 【模板】最近公共祖先（倍增法）

时间复杂度：预处理 $O(n \log n)$ ，查询 $O(\log n)$

空间复杂度： $O(n \log n)$

是否为最优解：是，对于在线查询是最优解

```
"""

```

```

def __init__(self, n: int, edges: List[List[int]]):
    self.MAXN = 500001
    self.LOG = 20

    self.depth = [0] * (n + 1)
    self.ancestor = [[0] * self.LOG for _ in range(n + 1)]
    self.tree = [[] for _ in range(n + 1)]

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

    self.dfs(1, 0)

```

```

def dfs(self, u: int, parent: int):
    self.depth[u] = self.depth[parent] + 1
    self.ancestor[u][0] = parent

    for i in range(1, self.LOG):
        if self.ancestor[u][i - 1] != 0:
            self.ancestor[u][i] = self.ancestor[self.ancestor[u][i - 1]][i - 1]

    for v in self.tree[u]:
        if v != parent:
            self.dfs(v, u)

def getLCA(self, u: int, v: int) -> int:
    if self.depth[u] < self.depth[v]:
        u, v = v, u

    for i in range(self.LOG - 1, -1, -1):
        if self.depth[u] - (1 << i) >= self.depth[v]:
            u = self.ancestor[u][i]

    if u == v:
        return u

    for i in range(self.LOG - 1, -1, -1):
        if self.ancestor[u][i] != self.ancestor[v][i]:
            u = self.ancestor[u][i]
            v = self.ancestor[v][i]

    return self.ancestor[u][0]

```

class TreeDistance:

"""

解法六: HDU 2586 How far away? (树上距离)

时间复杂度: 预处理 $O(n \log n)$, 查询 $O(\log n)$

空间复杂度: $O(n \log n)$

是否为最优解: 是

"""

```

def __init__(self, n: int, edges: List[List[int]], weights: List[int]):
    self.lca_solver = BinaryLiftingLCA(n, edges)
    self.dist = [0] * (n + 1)
    self._calculate_dist(1, 0, 0, edges, weights)

```

```

def _calculate_dist(self, u: int, parent: int, current_dist: int,
                   edges: List[List[int]], weights: List[int]):
    self.dist[u] = current_dist

    for i, (a, b) in enumerate(edges):
        if a == u and b != parent:
            self._calculate_dist(b, u, current_dist + weights[i], edges, weights)
        elif b == u and a != parent:
            self._calculate_dist(a, u, current_dist + weights[i], edges, weights)

def get_distance(self, u: int, v: int) -> int:
    lca = self.lca_solver.getLCA(u, v)
    return self.dist[u] + self.dist[v] - 2 * self.dist[lca]

```

class TarjanLCA:

"""

解法七：SPOJ LCASQ – Lowest Common Ancestor (Tarjan 离线算法)

时间复杂度: $O(n + q)$

空间复杂度: $O(n + q)$

是否为最优解: 是, 对于离线查询是最优解

"""

```

def __init__(self, n: int, edges: List[List[int]], query_pairs: List[List[int]]):
    self.tree = [[] for _ in range(n)]
    self.queries = [[] for _ in range(n)]
    self.parent = list(range(n))
    self.ancestor = [0] * n
    self.visited = [False] * n

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

    for i, (u, v) in enumerate(query_pairs):
        self.queries[u].append((v, i))
        self.queries[v].append((u, i))

def _find(self, x: int) -> int:
    if self.parent[x] != x:
        self.parent[x] = self._find(self.parent[x])
    return self.parent[x]

```

```

def _union(self, x: int, y: int):
    root_x = self._find(x)
    root_y = self._find(y)
    if root_x != root_y:
        self.parent[root_y] = root_x

def tarjan_lca(self, u: int, parent_node: int, results: List[int]):
    self.ancestor[u] = u
    self.visited[u] = True

    for v in self.tree[u]:
        if v != parent_node:
            self.tarjan_lca(v, u, results)
            self._union(u, v)
            self.ancestor[self._find(u)] = u

    for v, query_idx in self.queries[u]:
        if self.visited[v]:
            results[query_idx] = self.ancestor[self._find(v)]

def test_lca_algorithms():
    """
    测试所有 LCA 算法的实现
    """
    print("== LCA 算法综合测试 - Python 版本 ==\n")

    solution = LCASolution()

    # 测试 1: 标准二叉树 LCA
    print("== 测试 1: 标准二叉树 LCA ==")
    root = TreeNode(3)
    node5 = TreeNode(5)
    node1 = TreeNode(1)
    root.left = node5
    root.right = node1

    result1 = solution.lowestCommonAncestor(root, node5, node1)
    print(f"测试 1.1: LCA(5, 1) = {result1.val if result1 else 'None'}")

    # 测试 2: 二叉搜索树 LCA
    print("\n== 测试 2: 二叉搜索树 LCA ==")
    bst_root = TreeNode(6)
    bst_node2 = TreeNode(2)

```

```

bst_node8 = TreeNode(8)
bst_root.left = bst_node2
bst_root.right = bst_node8

result2 = solution.lowestCommonAncestorBST(bst_root, bst_node2, bst_node8)
print(f"测试 2.1: BST LCA(2, 8) = {result2.val if result2 else 'None'}")

# 测试 3: 带父指针的 LCA
print("\n==== 测试 3: 带父指针的 LCA ===")
root_wp = TreeNodeWithParent(3)
wp_node5 = TreeNodeWithParent(5)
wp_node1 = TreeNodeWithParent(1)
root_wp.left = wp_node5
root_wp.right = wp_node1
wp_node5.parent = root_wp
wp_node1.parent = root_wp

result3 = solution.lowestCommonAncestorWithParent(wp_node5, wp_node1)
print(f"测试 3.1: 带父指针 LCA(5, 1) = {result3.val if result3 else 'None'}")

# 测试 4: 迭代版本 LCA
print("\n==== 测试 4: 迭代版本 LCA ===")
result4 = solution.lowestCommonAncestorIterative(root, node5, node1)
print(f"测试 4.1: 迭代版 LCA(5, 1) = {result4.val if result4 else 'None'}")

# 测试 5: 洛谷 P3379 倍增法
print("\n==== 测试 5: 洛谷 P3379 倍增法 ===")
edges = [[1, 2], [1, 3], [2, 4], [2, 5]]
luogu = BinaryLiftingLCA(5, edges)
result5 = luogu.getLCA(4, 5)
print(f"测试 5.1: LCA(4, 5) = {result5}")

# 测试 6: HDU2586 树上距离
print("\n==== 测试 6: HDU2586 树上距离 ===")
weights = [10, 20, 30, 40]
hdu = TreeDistance(5, edges, weights)
result6 = hdu.get_distance(4, 5)
print(f"测试 6.1: Distance(4, 5) = {result6}")

print("\n==== 所有测试完成 ===")

if __name__ == "__main__":
    test_lca_algorithms()

```

=====