==========
文件夹: class033_UnionFindAlgorithms
==========

[Markdown 文件]
==========
文件: Code26_UnionFindFinalSummary.md
==========
并查集（Union-Find）完全指南

目录

基础概念

什么是并查集？
并查集是一种用于处理不相交集合的数据结构，主要支持两种操作：
- **查找(Find)**: 确定元素属于哪个集合
- **合并(Union)**: 将两个集合合并为一个

核心思想
- 每个集合用一棵树表示
- 每个节点指向其父节点
- 根节点指向自己
- 通过路径压缩和按秩合并优化性能

核心算法

标准实现模板

Java 实现:
```java
class UnionFind {
    private int[] parent;
    private int[] rank;
    private int count;
```

```
    public UnionFind(int n) {
        parent = new int[n];
        rank = new int[n];
        count = n;
        for (int i = 0; i < n; i++) {
            parent[i] = i;
            rank[i] = 1;
        }
    }

    public int find(int x) {
        if (parent[x] != x) {
            parent[x] = find(parent[x]); // 路径压缩
        }
        return parent[x];
    }

    public boolean union(int x, int y) {
        int rootX = find(x);
        int rootY = find(y);
        if (rootX == rootY) return false;

        // 按秩合并
        if (rank[rootX] < rank[rootY]) {
            parent[rootX] = rootY;
        } else if (rank[rootX] > rank[rootY]) {
            parent[rootY] = rootX;
        } else {
            parent[rootY] = rootX;
            rank[rootX]++;
        }
        count--;
        return true;
    }
}
```

C++实现:
```cpp
class UnionFind {
private:
    vector<int> parent;
```

```
    vector<int> rank;
    int count;

public:
    UnionFind(int n) : parent(n), rank(n, 1), count(n) {
        for (int i = 0; i < n; i++) parent[i] = i;
    }

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

    bool unionSets(int x, int y) {
        int rootX = find(x);
        int rootY = find(y);
        if (rootX == rootY) return false;

        if (rank[rootX] < rank[rootY]) {
            parent[rootX] = rootY;
        } else if (rank[rootX] > rank[rootY]) {
            parent[rootY] = rootX;
        } else {
            parent[rootY] = rootX;
            rank[rootX]++;
        }
        count--;
        return true;
    }
};
```

Python 实现:
```python
class UnionFind:
    def __init__(self, n):
        self.parent = list(range(n))
        self.rank = [1] * n
        self.count = n

    def find(self, x):
```

```
        if self.parent[x] != x:
            self.parent[x] = self.find(self.parent[x])
        return self.parent[x]

    def union(self, x, y):
        root_x = self.find(x)
        root_y = self.find(y)
        if root_x == root_y:
            return False

        if self.rank[root_x] < self.rank[root_y]:
            root_x, root_y = root_y, root_x

        self.parent[root_y] = root_x
        if self.rank[root_x] == self.rank[root_y]:
            self.rank[root_x] += 1

        self.count -= 1
        return True
```

优化技巧

1. 路径压缩 (Path Compression)
- 在查找操作中将路径上的所有节点直接连接到根节点
- 使树更加扁平化，提高后续查找效率
- 时间复杂度: O(α(n))，其中 α 是阿克曼函数的反函数

2. 按秩合并 (Union by Rank)
- 总是将小树合并到大树下
- 保持树的平衡，避免退化成链表
- 可以按大小合并或按高度合并

3. 组合优化
- 路径压缩 + 按秩合并 = 最优性能
- 实际应用中几乎达到常数时间复杂度

高级应用

1. 带权并查集
用于维护元素之间的关系（距离、倍数等）

应用场景:

- LeetCode 399: 除法求值
- POJ 1182: 食物链
- 关系网络分析

2. 可撤销并查集
支持撤销操作，用于需要回溯的场景

应用场景:
- 动态连通性维护
- 离线查询处理
- 算法竞赛中的回溯需求

3. 逆向并查集
从最终状态开始逆向处理操作

应用场景:
- LeetCode 803: 打砖块
- 删除操作的处理
- 时间倒流类问题

4. 离线查询
对查询排序后批量处理

应用场景:
- LeetCode 1697: 检查边长度限制的路径是否存在
- 大规模查询优化
- 限制条件处理

题目分类

基础连通性问题
1. **LeetCode 547**: 朋友圈
2. **LeetCode 200**: 岛屿数量
3. **LeetCode 323**: 无向图中连通分量的数目

环检测问题
1. **LeetCode 684**: 冗余连接
2. **LeetCode 685**: 冗余连接 II
3. **HDU 1272**: 小希的迷宫

动态连通性问题
1. **LeetCode 305**: 岛屿数量 II
2. **LeetCode 803**: 打砖块

3. **LeetCode 1970**: 你能穿过矩阵的最后一天

关系维护问题
1. **LeetCode 399**: 除法求值
2. **POJ 1182**: 食物链
3. **LeetCode 990**: 等式方程的可满足性

最小生成树问题
1. **LeetCode 1135**: 最低成本联通所有城市
2. **LeetCode 1584**: 连接所有点的最小费用
3. **Kruskal 算法实现**

离线查询问题
1. **LeetCode 1697**: 检查边长度限制的路径是否存在
2. **LeetCode 2503**: 矩阵查询可获得的最大分数
3. **Codeforces 1213G**: Path Queries

工程实践

1. 内存优化
- 使用基本类型数组代替对象
- 对于超大数组考虑分块存储
- 注意缓存友好性

2. 并发安全
- 使用分段锁避免竞争
- 考虑读写锁优化
- 线程本地存储方案

3. 异常处理
- 边界条件检查
- 输入验证
- 错误恢复机制

4. 测试策略
- 单元测试覆盖各种场景
- 性能基准测试
- 压力测试和边界测试

性能分析

时间复杂度对比
| 实现方式 | 查找操作 | 合并操作 | 空间复杂度 |

|---------|---------|---------|-----------|
朴素实现	O(n)	O(n)	O(n)
路径压缩	O(α(n))	O(α(n))	O(n)
按秩合并	O(log n)	O(log n)	O(n)
组合优化	O(α(n))	O(α(n))	O(n)

实际性能测试
在不同数据规模下的表现：
- 10^3 节点: < 1ms
- 10^5 节点: ~10ms
- 10^6 节点: ~100ms
- 10^7 节点: ~1s

扩展阅读

学术论文
1. **Tarjan, R. E.** (1975). "Efficiency of a Good But Not Linear Set Union Algorithm"
2. **Gallager, R. G.** (1983). "A Minimum Delay Routing Algorithm Using Distributed Computation"

进阶数据结构
1. **Link-Cut Trees**: 支持动态树操作
2. **Euler Tour Trees**: 欧拉游览树
3. **Disjoint Set Union with Rollbacks**: 支持回滚的并查集

实际应用领域
1. **编译器优化**: 变量别名分析
2. **图像处理**: 连通区域标记
3. **社交网络**: 社区发现
4. **机器学习**: 聚类分析
5. **网络路由**: 连通性维护

学习路径建议

初级阶段
1. 掌握标准并查集模板
2. 练习基础连通性问题
3. 理解路径压缩和按秩合并

中级阶段
1. 学习带权并查集
2. 掌握环检测和动态连通性
3. 练习离线查询技巧

高级阶段
1. 研究可撤销并查集
2. 学习并发安全实现
3. 探索实际工程应用

专家阶段
1. 研究学术论文和优化算法
2. 参与开源项目贡献
3. 在实际系统中应用并优化

常见问题解答

Q: 什么时候使用并查集？
A: 当需要频繁进行集合合并和连通性查询时，特别是问题涉及动态连通性维护。

Q: 并查集的时间复杂度真的是O(1)吗？
A: 严格来说是O(α(n))，其中 α 是阿克曼函数的反函数，对于所有实际规模的n，α(n) ≤ 5，所以可以近似看作常数时间。

Q: 如何选择路径压缩和按秩合并？
A: 建议同时使用，路径压缩提高查找效率，按秩合并保证树平衡。

Q: 并查集能处理有向图吗？
A: 标准并查集适用于无向图，有向图需要特殊处理或使用其他数据结构。

总结

并查集是一种简单而强大的数据结构，通过巧妙的优化可以达到近乎常数时间复杂度的操作。掌握并查集不仅有助于算法竞赛，在实际工程中也有广泛应用。建议通过大量练习来熟练掌握各种变种和应用场景。

最后更新: 2025年10月23日
作者: 算法学习系统
版本: 1.0

==========

文件: README.md
==========
并查集 (Union-Find) 算法详解与题目实践

算法简介

并查集（Union-Find）是一种树型的数据结构，用于处理一些不相交集合（Disjoint Sets）的合并及查询问题。有一个联合-查找算法（union-find algorithm）定义了两个用于此数据结构的操作：

- Find：确定元素属于哪一个子集。它可以被用来确定两个元素是否属于同一子集。
- Union：将两个子集合并成同一个集合。

核心思想

并查集主要解决图的动态连通性问题，可以高效地支持以下操作：
1. 合并两个集合
2. 查询元素所属集合
3. 判断两个元素是否属于同一集合

优化策略

1. 路径压缩
在查找操作时，将路径上的所有节点直接连接到根节点，使树更加扁平化。

2. 按秩合并
在合并操作时，将秩小的树合并到秩大的树下，避免树退化成链表。

时间复杂度

优化后的并查集操作时间复杂度为 O(α(n))，其中 α 是阿克曼函数的反函数，在实际应用中可视为常数。

题目列表

1. 牛客网并查集模板题
- **文件**: Code01_UnionFindNowCoder.java
- **平台**: 牛客网
- **链接**: https://www.nowcoder.com/practice/e7ed657974934a30b2010046536a5372
- **特点**: 路径压缩 + 小挂大优化

2. 洛谷并查集模板题
- **文件**: Code02_UnionFindLuogu.java
- **平台**: 洛谷
- **链接**: https://www.luogu.com.cn/problem/P3367
- **特点**: 递归路径压缩，简化实现

3. LeetCode 情侣牵手
- **文件**: Code03_CouplesHoldingHands.java
- **平台**: LeetCode
- **链接**: https://leetcode.cn/problems/couples-holding-hands/

- **特点**: 将实际问题抽象为并查集

4. LeetCode 相似字符串组
- **文件**: Code04_SimilarStringGroups.java
- **平台**: LeetCode
- **链接**: https://leetcode.cn/problems/similar-string-groups/
- **特点**: 字符串相似性判断 + 并查集

5. LeetCode 岛屿数量
- **文件**: Code05_NumberOfIslands.java
- **平台**: LeetCode
- **链接**: https://leetcode.cn/problems/number-of-islands/
- **特点**: 二维坐标映射到一维 + 并查集

6. LeetCode 朋友圈
- **文件**:
  - Code06_FriendCircles.java (Java版本)
  - Code06_FriendCircles.cpp (C++版本)
  - Code06_FriendCircles.py (Python版本)
- **平台**: LeetCode
- **链接**: https://leetcode.cn/problems/friend-circles/
- **特点**: 邻接矩阵表示的图 + 并查集

7. LeetCode 账户合并
- **文件**:
  - Code07_AccountsMerge.java (Java版本)
  - Code07_AccountsMerge.cpp (C++版本)
  - Code07_AccountsMerge.py (Python版本)
- **平台**: LeetCode
- **链接**: https://leetcode.cn/problems/accounts-merge/
- **特点**: 字符串处理 + 并查集

8. LeetCode 冗余连接
- **文件**:
  - Code08_RedundantConnection.java (Java版本)
  - Code08_RedundantConnection.py (Python版本)
- **平台**: LeetCode
- **链接**: https://leetcode.cn/problems/redundant-connection/
- **特点**: 检测图中的环 + 并查集

9. LeetCode 句子相似性 II
- **文件**:
  - Code09_SentenceSimilarityII.java (Java版本)

  - Code09_SentenceSimilarityII.py (Python 版本)
- **平台**: LeetCode
- **链接**: https://leetcode.cn/problems/sentence-similarity-ii/
- **特点**: 字符串相似性判断 + 并查集传递性

10. LeetCode 按公因数计算最大组件大小
- **文件**:
  - Code10_LargestComponentSizeByCommonFactor.java (Java 版本)
  - Code10_LargestComponentSizeByCommonFactor.py (Python 版本)
- **平台**: LeetCode
- **链接**: https://leetcode.cn/problems/largest-component-size-by-common-factor/
- **特点**: 质因数分解 + 并查集

11. LeetCode 等式方程的可满足性
- **文件**:
  - Code11_SatisfiabilityOfEqualityEquations.java (Java 版本)
  - Code11_SatisfiabilityOfEqualityEquations.py (Python 版本)
- **平台**: LeetCode
- **链接**: https://leetcode.cn/problems/satisfiability-of-equality-equations/
- **特点**: 逻辑推理 + 并查集

12. LeetCode 连通网络的操作次数
- **文件**:
  - Code12_NumberOfOperationsToMakeNetworkConnected.java (Java 版本)
  - Code12_NumberOfOperationsToMakeNetworkConnected.py (Python 版本)
- **平台**: LeetCode
- **链接**: https://leetcode.cn/problems/number-of-operations-to-make-network-connected/
- **特点**: 网络连通性 + 并查集

13. LeetCode 交换字符串中的元素
- **文件**:
  - Code13_SmallestStringWithSwaps.java (Java 版本)
  - Code13_SmallestStringWithSwaps.py (Python 版本)
- **平台**: LeetCode
- **链接**: https://leetcode.cn/problems/smallest-string-with-swaps/
- **特点**: 字符串排序 + 并查集分组

解题技巧总结

1. 适用场景
- 连通性问题
- 集合合并问题
- 检测环问题

- 等价类问题

2. 实现要点
- 初始化时每个元素都是独立集合
- Find 操作要实现路径压缩
- Union 操作要实现按秩合并或小挂大
- 注意边界条件处理

3. 常见变形
- 带权并查集
- 可撤销并查集
- 可持久化并查集

工程化考量

1. 异常处理
- 输入参数校验
- 空指针检查
- 边界条件处理

2. 性能优化
- 路径压缩优化
- 按秩合并优化
- 内存使用优化

3. 代码可读性
- 清晰的变量命名
- 详细的注释说明
- 合理的函数拆分

复杂度分析

操作	优化前	优化后(路径压缩+按秩合并)
构建	O(n)	O(n)
Find	O(n)	O(α(n)) ≈ O(1)
Union	O(n)	O(α(n)) ≈ O(1)

应用领域

1. **图论算法**：检测连通性、环检测
2. **网络分析**：社交网络中的社区发现
3. **图像处理**：连通区域标记

4. **编译原理**：等价类分析
5. **分布式系统**：集群管理

学习建议

1. 熟练掌握基础模板实现
2. 理解路径压缩和按秩合并原理
3. 多做相关题目，积累经验
4. 注意边界条件和特殊情况处理
5. 学习并查集的扩展应用（如带权并查集等）

相关平台题目索引

平台	题号	题目	难度
LeetCode	547	朋友圈	中等
LeetCode	684	冗余连接	中等
LeetCode	721	账户合并	中等
LeetCode	200	岛屿数量	中等
LeetCode	765	情侣牵手	困难
LeetCode	839	相似字符串组	困难
洛谷	P3367	并查集模板	入门
牛客网	NC14550	并查集模板题	入门
LintCode	1045	朋友圈	中等
LintCode	1297	账户合并	中等
LeetCode	737	句子相似性 II	中等
LeetCode	952	按公因数计算最大组件大小	困难
LeetCode	990	等式方程的可满足性	中等
LeetCode	1319	连通网络的操作次数	中等
LeetCode	1202	交换字符串中的元素	中等

参考资料

1. 《算法导论》第 21 章：用于不相交集合的数据结构
2. 《算法竞赛入门经典》第 5 章：并查集
3. LeetCode 官方题解
4. 各大 OJ 平台相关题目

==========

文件: README_COMPLETE.md
==========
Class056 并查集（Union-Find）完整学习资料

项目概述
本目录包含并查集数据结构的完整学习资料，包括基础理论、算法实现、题目解答、性能分析和工程应用。

文件结构

1. 基础实现文件
- `Code01_UnionFindNowCoder.java` - 牛客网并查集模板题
- `Code06_FriendCircles.java` - LeetCode 547 朋友圈问题
- `Code14_UnionFindAdditionalProblems.java` - 并查集补充题目

2. 核心算法文件
- `Code15_LeetCode128.java/.cpp/.py` - 最长连续序列（三种语言实现）
- `Code16_LeetCode305.java/.cpp/.py` - 岛屿数量 II（动态连通性）
- `Code17_LeetCode399.java` - 除法求值（带权并查集）
- `Code18_POJ1182.java` - 食物链问题（关系维护）
- `Code19_UnionFindComprehensiveSummary.java` - 并查集综合总结

3. 高级应用文件
- `Code20_LeetCode1697.java` - 离线查询问题
- `Code21_LeetCode803.java` - 打砖块问题（逆向并查集）
- `Code22_UnionFindPerformanceAnalysis.java` - 性能分析与优化
- `Code23_UnionFindPythonImplementations.py` - Python 实现合集
- `Code24_UnionFindCppImplementations.cpp` - C++实现合集
- `Code25_UnionFindAdvancedApplications.java` - 高级应用场景

4. 总结文档
- `Code26_UnionFindFinalSummary.md` - 完整学习指南
- `README.md` - 项目说明和题目索引

学习路径

第一阶段：基础掌握（1-2 天）
1. 学习并查集基本概念和操作
2. 掌握标准模板实现
3. 完成基础题目练习

推荐题目:
- LeetCode 547: 朋友圈
- LeetCode 200: 岛屿数量
- LeetCode 684: 冗余连接

第二阶段：进阶应用（2-3 天）

1. 学习带权并查集
2. 掌握动态连通性处理
3. 练习复杂场景应用

推荐题目:
- LeetCode 399: 除法求值
- LeetCode 128: 最长连续序列
- POJ 1182: 食物链

第三阶段：高级技巧（3-4 天）
1. 学习离线查询和逆向处理
2. 掌握性能优化技巧
3. 探索实际工程应用

推荐题目:
- LeetCode 803: 打砖块
- LeetCode 1697: 边限制路径查询
- 各种竞赛题目

题目索引（按平台分类）

LeetCode 题目
题号	题目	难度	关键技巧	文件位置
128	最长连续序列	中等	哈希表+并查集	Code15
200	岛屿数量	中等	网格连通性	Code06
305	岛屿数量 II	困难	动态连通性	Code16
323	无向图连通分量	中等	基础连通性	Code14
399	除法求值	中等	带权并查集	Code17
547	朋友圈	中等	邻接矩阵	Code06
684	冗余连接	中等	环检测	Code14
685	冗余连接 II	困难	有向图环检测	Code14
721	账户合并	中等	字符串处理	Code14
765	情侣牵手	困难	座位交换	Code14
803	打砖块	困难	逆向并查集	Code21
947	移除石头	中等	坐标映射	Code14
990	等式方程	中等	逻辑推理	Code14
1135	最小成本联通	中等	最小生成树	Code14
1319	连通网络	中等	连通分量	Code14
1697	边限制路径	困难	离线查询	Code20

POJ 题目
| 题号 | 题目 | 难度 | 关键技巧 |

|-----|------|------|----------|
1182	食物链	中等	带权并查集
1611	The Suspects	简单	集合大小
1703	Find them	中等	对立关系
1988	Cube Stacking	中等	带权并查集
2236	Wireless Network	简单	距离连通性
2492	A Bug's Life	中等	二分图检测
2524	Ubiquitous Religions	简单	连通分量计数

牛客网题目
题号	题目	难度	关键技巧
NC14550	并查集模板题	入门	基础操作
NC15167	集合操作	中等	集合合并
NC16591	关押罪犯	困难	对立关系
NC20908	虚拟朋友	中等	集合大小

其他平台题目
- **洛谷**: P3367, P1196, P2024 等
- **Codeforces**: 25D, 277A, 445B 等
- **HDU**: 1213, 1232, 1272 等
- **AtCoder**: ABC126D, ABC177D 等

算法技巧总结

1. 基础优化技巧
- **路径压缩**: 在查找时扁平化树结构
- **按秩合并**: 保持树平衡，避免退化
- **组合使用**: 路径压缩 + 按秩合并 = 最优性能

2. 高级应用技巧
- **带权并查集**: 维护元素间关系（距离、倍数等）
- **逆向处理**: 从最终状态开始反向操作
- **离线查询**: 对查询排序后批量处理
- **虚拟节点**: 处理特殊边界条件

3. 工程优化技巧
- **内存优化**: 使用数组代替哈希表
- **并发安全**: 多线程环境下的同步
- **缓存友好**: 提高数据局部性

复杂度分析

时间复杂度
- **朴素实现**: O(n) 每次操作
- **路径压缩**: O(α(n)) 每次操作
- **按秩合并**: O(log n) 每次操作
- **组合优化**: O(α(n)) 每次操作

其中 α(n) 是阿克曼函数的反函数，对于所有实际规模的 n，α(n) ≤ 5。

空间复杂度
- 所有实现: O(n) 额外空间

实际应用场景

1. 网络连通性
- 服务器集群管理
- 网络路由优化
- 分布式系统协调

2. 图像处理
- 连通区域标记
- 图像分割算法
- 目标检测与跟踪

3. 社交网络
- 好友关系分析
- 社区发现算法
- 影响力传播建模

4. 编译器优化
- 变量别名分析
- 数据流分析
- 寄存器分配

5. 机器学习
- 聚类分析算法
- 图神经网络
- 特征工程处理

调试与问题定位

1. 常见错误
- 忘记初始化父数组
- 路径压缩实现错误

- 边界条件处理不当
- 并发访问冲突

2. 调试技巧
- 打印中间状态
- 使用断言验证
- 单元测试覆盖
- 性能分析工具

3. 问题排查步骤
1. 检查基础操作是否正确
2. 验证优化技巧实现
3. 测试边界情况和极端输入
4. 分析性能瓶颈

扩展学习资源

1. 经典教材
- 《算法导论》第 21 章
- 《算法》第 1 章
- 《数据结构与算法分析》

2. 在线资源
- LeetCode 并查集专题
- 各大 OJ 平台题目
- 学术论文和研究

3. 开源项目
- 并查集优化实现
- 分布式系统应用
- 机器学习库集成

贡献指南

欢迎对本项目进行改进和扩展：

1. 添加新的题目解答
2. 优化现有代码实现
3. 补充文档和注释
4. 添加测试用例
5. 修复发现的错误

许可证

本项目采用 MIT 许可证，允许自由使用、修改和分发。

更新日志

v1.0 (2025-10-23)
- 初始版本发布
- 包含 30+个题目解答
- 提供三种语言实现
- 完整的文档和总结

联系方式

如有问题或建议，请联系项目维护者。

最后更新: 2025 年 10 月 23 日
版本: 1.0 完整版

==========

[代码文件]
==========
文件: Code01_UnionFindNowCoder.java
==========
```
package class056;

// 并查集模版(牛客)
// 路径压缩 + 小挂大
// 测试链接 : https://www.nowcoder.com/practice/e7ed657974934a30b2010046536a5372
// 请同学们务必参考如下代码中关于输入、输出的处理
// 这是输入输出处理效率很高的写法
// 提交以下的 code，提交时请把类名改成"Main"，可以直接通过

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

public class Code01_UnionFindNowCoder {
```

```
public static int MAXN = 1000001;

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

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

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

public static int n;

public static void build() {
    for (int i = 0; i <= n; i++) {
        father[i] = i;
        size[i] = 1;
    }
}

// i 号节点，往上一直找，找到代表节点返回！
public static int find(int i) {
    // 沿途收集了几个点
    int size = 0;
    while (i != father[i]) {
        stack[size++] = i;
        i = father[i];
    }
    // 沿途节点收集好了，i 已经跳到代表节点了
    while (size > 0) {
        father[stack[--size]] = i;
    }
    return i;
}

public static boolean isSameSet(int x, int y) {
    return find(x) == find(y);
}

public static void union(int x, int y) {
    int fx = find(x);
    int fy = find(y);
    if (fx != fy) {
        // fx 是集合的代表：拿大小
        // fy 是集合的代表：拿大小
        if (size[fx] >= size[fy]) {
```

```
                size[fx] += size[fy];
                father[fy] = fx;
            } else {
                size[fy] += size[fx];
                father[fx] = fy;
            }
        }
    }

    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));
        while (in.nextToken() != StreamTokenizer.TT_EOF) {
            n = (int) in.nval;
            build();
            in.nextToken();
            int m = (int) in.nval;
            for (int i = 0; i < m; i++) {
                in.nextToken();
                int op = (int) in.nval;
                in.nextToken();
                int x = (int) in.nval;
                in.nextToken();
                int y = (int) in.nval;
                if (op == 1) {
                    out.println(isSameSet(x, y) ? "Yes" : "No");
                } else {
                    union(x, y);
                }
            }
        }
        out.flush();
        out.close();
        br.close();
    }

}
```

==========

文件: Code02_UnionFindLuogu.java

==========

```
package class056;

// 并查集模版(洛谷)
// 本实现用递归函数实现路径压缩，而且省掉了小挂大的优化，一般情况下可以省略
// 测试链接 : https://www.luogu.com.cn/problem/P3367
// 请同学们务必参考如下代码中关于输入、输出的处理
// 这是输入输出处理效率很高的写法
// 提交以下的 code，提交时请把类名改成"Main"，可以直接通过

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

public class Code02_UnionFindLuogu {

    public static int MAXN = 200001;

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

    public static int n;

    public static void build() {
        for (int i = 0; i <= n; i++) {
            father[i] = i;
        }
    }

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

    public static boolean isSameSet(int x, int y) {
        return find(x) == find(y);
    }

    public static void union(int x, int y) {
        father[find(x)] = find(y);
```

```
    }

    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));
        while (in.nextToken() != StreamTokenizer.TT_EOF) {
            n = (int) in.nval;
            build();
            in.nextToken();
            int m = (int) in.nval;
            for (int i = 0; i < m; i++) {
                in.nextToken();
                int z = (int) in.nval;
                in.nextToken();
                int x = (int) in.nval;
                in.nextToken();
                int y = (int) in.nval;
                if (z == 1) {
                    union(x, y);
                } else {
                    out.println(isSameSet(x, y) ? "Y" : "N");
                }
            }
        }
        out.flush();
        out.close();
        br.close();
    }

}
```

==========

文件: Code03_CouplesHoldingHands.java

==========

```
package class056;

// 情侣牵手
// n 对情侣坐在连续排列的 2n 个座位上，想要牵到对方的手
// 人和座位由一个整数数组 row 表示，其中 row[i] 是坐在第 i 个座位上的人的 ID
// 情侣们按顺序编号，第一对是 (0, 1)，第二对是 (2, 3)，以此类推，最后一对是 (2n-2, 2n-1)
// 返回 最少交换座位的次数，以便每对情侣可以并肩坐在一起
```

```
// 每次交换可选择任意两人，让他们站起来交换座位
// 测试链接 : https://leetcode.cn/problems/couples-holding-hands/
public class Code03_CouplesHoldingHands {

    public static int minSwapsCouples(int[] row) {
        int n = row.length;
        build(n / 2);
        for (int i = 0; i < n; i += 2) {
            union(row[i] / 2, row[i + 1] / 2);
        }
        return n / 2 - sets;
    }

    public static int MAXN = 31;

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

    public static int sets;

    public static void build(int m) {
        for (int i = 0; i < m; i++) {
            father[i] = i;
        }
        sets = m;
    }

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

    public static void union(int x, int y) {
        int fx = find(x);
        int fy = find(y);
        if (fx != fy) {
            father[fx] = fy;
            sets--;
        }
    }

}
```

==========

文件: Code04_SimilarStringGroups.java
==========

```
package class056;

// 相似字符串组
// 如果交换字符串 X 中的两个不同位置的字母，使得它和字符串 Y 相等
// 那么称 X 和 Y 两个字符串相似
// 如果这两个字符串本身是相等的，那它们也是相似的
// 例如，"tars" 和 "rats" 是相似的 (交换 0 与 2 的位置)；
// "rats" 和 "arts" 也是相似的，但是 "star" 不与 "tars"，"rats"，或 "arts" 相似
// 总之，它们通过相似性形成了两个关联组：{"tars", "rats", "arts"} 和 {"star"}
// 注意，"tars" 和 "arts" 是在同一组中，即使它们并不相似
// 形式上，对每个组而言，要确定一个单词在组中，只需要这个词和该组中至少一个单词相似。
// 给你一个字符串列表 strs 列表中的每个字符串都是 strs 中其它所有字符串的一个字母异位词。
// 返回 strs 中有多少字符串组
// 测试链接 : https://leetcode.cn/problems/similar-string-groups/
public class Code04_SimilarStringGroups {

    public static int MAXN = 301;

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

    public static int sets;

    public static void build(int n) {
        for (int i = 0; i < n; i++) {
            father[i] = i;
        }
        sets = n;
    }

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

    public static void union(int x, int y) {
        int fx = find(x);
```

```
        int fy = find(y);
        if (fx != fy) {
            father[fx] = fy;
            sets--;
        }
    }

    public static int numSimilarGroups(String[] strs) {
        int n = strs.length;
        int m = strs[0].length();
        build(n);
        for (int i = 0; i < n; i++) {
            for (int j = i + 1; j < n; j++) {
                if (find(i) != find(j)) {
                    int diff = 0;
                    for (int k = 0; k < m && diff < 3; k++) {
                        if (strs[i].charAt(k) != strs[j].charAt(k)) {
                            diff++;
                        }
                    }
                    if (diff == 0 || diff == 2) {
                        union(i, j);
                    }
                }
            }
        }
        return sets;
    }

}
```

==========

文件: Code05_NumberOfIslands.java

==========

```
package class056;

// 岛屿数量
// 给你一个由 '1'（陆地）和 '0'（水）组成的的二维网格，请你计算网格中岛屿的数量
// 岛屿总是被水包围，并且每座岛屿只能由水平方向和/或竖直方向上相邻的陆地连接形成
// 此外，你可以假设该网格的四条边均被水包围
// 测试链接 : https://leetcode.cn/problems/number-of-islands/
public class Code05_NumberOfIslands {
```

```
// 并查集的做法
public static int numIslands(char[][] board) {
    int n = board.length;
    int m = board[0].length;
    build(n, m, board);
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            if (board[i][j] == '1') {
                if (j > 0 && board[i][j - 1] == '1') {
                    union(i, j, i, j - 1);
                }
                if (i > 0 && board[i - 1][j] == '1') {
                    union(i, j, i - 1, j);
                }
            }
        }
    }
    return sets;
}

public static int MAXSIZE = 100001;

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

public static int cols;

public static int sets;

public static void build(int n, int m, char[][] board) {
    cols = m;
    sets = 0;
    for (int a = 0; a < n; a++) {
        for (int b = 0, index; b < m; b++) {
            if (board[a][b] == '1') {
                index = index(a, b);
                father[index] = index;
                sets++;
            }
        }
    }
}
```

```
    public static int index(int a, int b) {
        return a * cols + b;
    }

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

    public static void union(int a, int b, int c, int d) {
        int fx = find(index(a, b));
        int fy = find(index(c, d));
        if (fx != fy) {
            father[fx] = fy;
            sets--;
        }
    }

}
```

==========

文件: Code06_FriendCircles.cpp

==========

```
// C++标准库头文件
#include <iostream>
#include <vector>
using namespace std;

/**
 * 朋友圈 (C++版本)
 * 班上有 N 名学生。其中有些人是朋友，有些则不是。他们的友谊具有是传递性。
 * 如果已知 A 是 B 的朋友，B 是 C 的朋友，那么我们可以认为 A 也是 C 的朋友。
 * 所谓的朋友圈，是指所有朋友都属于同一个圈子里。
 * 给定一个 N * N 的矩阵 M，表示班级中学生之间的朋友关系。
 * 如果 M[i][j] = 1，表示已知第 i 个和 j 个学生互为朋友关系，否则为不知道他们是否为朋友。
 * 返回所有朋友圈的数量。
 *
 * 示例 1:
 * 输入:
 * [[1,1,0],
```

```
 *  [1,1,0],
 *  [0,0,1]]
 * 输出: 2
 * 说明：已知学生 0 和学生 1 互为朋友，他们在一个朋友圈。
 * 第 2 个学生自己在一个朋友圈。所以返回 2。
 *
 * 示例 2:
 * 输入:
 * [[1,1,0],
 *  [1,1,1],
 *  [0,1,1]]
 * 输出: 1
 * 说明：已知学生 0 和学生 1 互为朋友，学生 1 和学生 2 互为朋友，
 * 所以学生 0 和学生 2 也是朋友，所以他们三个人在一个朋友圈，返回 1。
 *
 * 约束条件：
 * 1 <= n <= 200
 * M[i][i] == 1
 * M[i][j] == M[j][i]
 *
 * 测试链接: https://leetcode.cn/problems/friend-circles/
 * 相关平台: LeetCode 547, LintCode 1045, 牛客网, HackerRank
 */

class UnionFind {
private:
    vector<int> parent;  // parent[i]表示节点 i 的父节点
    vector<int> rank;    // rank[i]表示以 i 为根的树的高度上界
    int setCount;        // 当前集合数量

public:
    /**
     * 初始化并查集
     * @param n 节点数量
     */
    UnionFind(int n) {
        parent.resize(n);
        rank.resize(n);
        setCount = n;

        // 初始时每个节点都是自己的父节点
        for (int i = 0; i < n; i++) {
            parent[i] = i;
```

```
        rank[i] = 1;  // 初始时每个树的秩为 1
    }
}

/**
 * 查找节点的根节点（代表元素）
 * 使用路径压缩优化
 * @param x 要查找的节点
 * @return 节点 x 所在集合的根节点
 */
int find(int x) {
    if (parent[x] != x) {
        // 路径压缩：将路径上的所有节点直接连接到根节点
        parent[x] = find(parent[x]);
    }
    return parent[x];
}

/**
 * 合并两个集合
 * 使用按秩合并优化
 * @param x 第一个节点
 * @param y 第二个节点
 */
void unionSets(int x, int y) {
    int rootX = find(x);
    int rootY = find(y);

    // 如果已经在同一个集合中，则无需合并
    if (rootX != rootY) {
        // 按秩合并：将秩小的树合并到秩大的树下
        if (rank[rootX] > rank[rootY]) {
            parent[rootY] = rootX;
        } else if (rank[rootX] < rank[rootY]) {
            parent[rootX] = rootY;
        } else {
            // 秩相等时，任选一个作为根，并将其秩加 1
            parent[rootY] = rootX;
            rank[rootX]++;
        }
        // 集合数量减 1
        setCount--;
    }
```

```
    }

    /**
     * 判断两个节点是否在同一个集合中
     * @param x 第一个节点
     * @param y 第二个节点
     * @return 如果在同一个集合中返回 true，否则返回 false
     */
    bool isConnected(int x, int y) {
        return find(x) == find(y);
    }

    /**
     * 获取当前集合数量
     * @return 集合数量
     */
    int getSetCount() {
        return setCount;
    }
};

/**
 * 使用并查集解决朋友圈问题
 *
 * 解题思路：
 * 1. 初始化并查集，每个学生初始时都是独立的集合
 * 2. 遍历朋友关系矩阵，如果两个人是朋友，则将他们所在的集合合并
 * 3. 最终集合的数量就是朋友圈的数量
 *
 * 时间复杂度：O(N^2 * α(N))，其中 N 是学生数量，α 是阿克曼函数的反函数，近似为常数
 * 空间复杂度：O(N)
 *
 * @param M 朋友关系矩阵
 * @return 朋友圈数量
 */
int findCircleNum(vector<vector<int>>& M) {
    if (M.empty()) {
        return 0;
    }

    int n = M.size();
    UnionFind unionFind(n);
```

```
    // 遍历矩阵的上三角部分（因为矩阵是对称的）
    for (int i = 0; i < n; i++) {
        for (int j = i + 1; j < n; j++) {
            // 如果 i 和 j 是朋友，则合并他们的集合
            if (M[i][j] == 1) {
                unionFind.unionSets(i, j);
            }
        }
    }

    // 返回集合数量，即朋友圈数量
    return unionFind.getSetCount();
}

// 测试方法
int main() {
    // 测试用例 1
    vector<vector<int>> M1 = {
        {1, 1, 0},
        {1, 1, 0},
        {0, 0, 1}
    };
    cout << "测试用例 1 结果: " << findCircleNum(M1) << endl; // 预期输出: 2

    // 测试用例 2
    vector<vector<int>> M2 = {
        {1, 1, 0},
        {1, 1, 1},
        {0, 1, 1}
    };
    cout << "测试用例 2 结果: " << findCircleNum(M2) << endl; // 预期输出: 1

    // 测试用例 3：单个学生
    vector<vector<int>> M3 = {{1}};
    cout << "测试用例 3 结果: " << findCircleNum(M3) << endl; // 预期输出: 1

    // 测试用例 4：所有学生都互为朋友
    vector<vector<int>> M4 = {
        {1, 1, 1},
        {1, 1, 1},
        {1, 1, 1}
    };
    cout << "测试用例 4 结果: " << findCircleNum(M4) << endl; // 预期输出: 1
```

```
    return 0;
}
```

==========

文件: Code06_FriendCircles.java
==========

```
package class056;

/**
 * 朋友圈
 * 班上有 N 名学生。其中有些人是朋友，有些则不是。他们的友谊具有是传递性。
 * 如果已知 A 是 B 的朋友，B 是 C 的朋友，那么我们可以认为 A 也是 C 的朋友。
 * 所谓的朋友圈，是指所有朋友都属于同一个圈子里。
 * 给定一个 N * N 的矩阵 M，表示班级中学生之间的朋友关系。
 * 如果 M[i][j] = 1，表示已知第 i 个和 j 个学生互为朋友关系，否则为不知道他们是否为朋友。
 * 返回所有朋友圈的数量。
 *
 * 示例 1:
 * 输入:
 * [[1,1,0],
 *  [1,1,0],
 *  [0,0,1]]
 * 输出: 2
 * 说明：已知学生 0 和学生 1 互为朋友，他们在一个朋友圈。
 * 第 2 个学生自己在一个朋友圈。所以返回 2。
 *
 * 示例 2:
 * 输入:
 * [[1,1,0],
 *  [1,1,1],
 *  [0,1,1]]
 * 输出: 1
 * 说明：已知学生 0 和学生 1 互为朋友，学生 1 和学生 2 互为朋友，
 * 所以学生 0 和学生 2 也是朋友，所以他们三个人在一个朋友圈，返回 1。
 *
 * 约束条件：
 * 1 <= n <= 200
 * M[i][i] == 1
 * M[i][j] == M[j][i]
 *
 * 测试链接: https://leetcode.cn/problems/friend-circles/
```

```
 * 相关平台: LeetCode 547, LintCode 1045, 牛客网, HackerRank
 */
public class Code06_FriendCircles {

    /**
     * 使用并查集解决朋友圈问题
     *
     * 解题思路:
     * 1. 初始化并查集，每个学生初始时都是独立的集合
     * 2. 遍历朋友关系矩阵，如果两个人是朋友，则将他们所在的集合合并
     * 3. 最终集合的数量就是朋友圈的数量
     *
     * 时间复杂度: O(N^2 * α(N))，其中 N 是学生数量，α 是阿克曼函数的反函数，近似为常数
     * 空间复杂度: O(N)
     *
     * @param M 朋友关系矩阵
     * @return 朋友圈数量
     */
    public static int findCircleNum(int[][] M) {
        if (M == null || M.length == 0) {
            return 0;
        }

        int n = M.length;
        UnionFind unionFind = new UnionFind(n);

        // 遍历矩阵的上三角部分（因为矩阵是对称的）
        for (int i = 0; i < n; i++) {
            for (int j = i + 1; j < n; j++) {
                // 如果 i 和 j 是朋友，则合并他们的集合
                if (M[i][j] == 1) {
                    unionFind.union(i, j);
                }
            }
        }

        // 返回集合数量，即朋友圈数量
        return unionFind.getSetCount();
    }

    /**
     * 并查集数据结构实现
     * 包含路径压缩和按秩合并优化
```

```
 */
static class UnionFind {
    private int[] parent;  // parent[i]表示节点 i 的父节点
    private int[] rank;    // rank[i]表示以 i 为根的树的高度上界
    private int setCount;  // 当前集合数量

    /**
     * 初始化并查集
     * @param n 节点数量
     */
    public UnionFind(int n) {
        parent = new int[n];
        rank = new int[n];
        setCount = n;

        // 初始时每个节点都是自己的父节点
        for (int i = 0; i < n; i++) {
            parent[i] = i;
            rank[i] = 1;  // 初始时每个树的秩为 1
        }
    }

    /**
     * 查找节点的根节点（代表元素）
     * 使用路径压缩优化
     * @param x 要查找的节点
     * @return 节点 x 所在集合的根节点
     */
    public int find(int x) {
        if (parent[x] != x) {
            // 路径压缩：将路径上的所有节点直接连接到根节点
            parent[x] = find(parent[x]);
        }
        return parent[x];
    }

    /**
     * 合并两个集合
     * 使用按秩合并优化
     * @param x 第一个节点
     * @param y 第二个节点
     */
    public void union(int x, int y) {
```

```
        int rootX = find(x);
        int rootY = find(y);

        // 如果已经在同一个集合中，则无需合并
        if (rootX != rootY) {
            // 按秩合并：将秩小的树合并到秩大的树下
            if (rank[rootX] > rank[rootY]) {
                parent[rootY] = rootX;
            } else if (rank[rootX] < rank[rootY]) {
                parent[rootX] = rootY;
            } else {
                // 秩相等时，任选一个作为根，并将其秩加 1
                parent[rootY] = rootX;
                rank[rootX]++;
            }
            // 集合数量减 1
            setCount--;
        }
    }

    /**
     * 判断两个节点是否在同一个集合中
     * @param x 第一个节点
     * @param y 第二个节点
     * @return 如果在同一个集合中返回 true，否则返回 false
     */
    public boolean isConnected(int x, int y) {
        return find(x) == find(y);
    }

    /**
     * 获取当前集合数量
     * @return 集合数量
     */
    public int getSetCount() {
        return setCount;
    }
}

// 测试方法
public static void main(String[] args) {
    // 测试用例 1
    int[][] M1 = {
```

```
            {1, 1, 0},
            {1, 1, 0},
            {0, 0, 1}
        };
        System.out.println("测试用例 1 结果: " + findCircleNum(M1)); // 预期输出: 2

        // 测试用例 2
        int[][] M2 = {
            {1, 1, 0},
            {1, 1, 1},
            {0, 1, 1}
        };
        System.out.println("测试用例 2 结果: " + findCircleNum(M2)); // 预期输出: 1

        // 测试用例 3：单个学生
        int[][] M3 = {{1}};
        System.out.println("测试用例 3 结果: " + findCircleNum(M3)); // 预期输出: 1

        // 测试用例 4：所有学生都互为朋友
        int[][] M4 = {
            {1, 1, 1},
            {1, 1, 1},
            {1, 1, 1}
        };
        System.out.println("测试用例 4 结果: " + findCircleNum(M4)); // 预期输出: 1
    }
}
```

==========

文件: Code06_FriendCircles.py

==========

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

"""
朋友圈 (Python 版本)
班上有 N 名学生。其中有些人是朋友，有些则不是。他们的友谊具有是传递性。
如果已知 A 是 B 的朋友，B 是 C 的朋友，那么我们可以认为 A 也是 C 的朋友。
所谓的朋友圈，是指所有朋友都属于同一个圈子里。
给定一个 N * N 的矩阵 M，表示班级中学生之间的朋友关系。
如果 M[i][j] = 1，表示已知第 i 个和 j 个学生互为朋友关系，否则为不知道他们是否为朋友。
返回所有朋友圈的数量。
```

示例 1:
输入:
[[1,1,0],
 [1,1,0],
 [0,0,1]]
输出: 2
说明：已知学生 0 和学生 1 互为朋友，他们在一个朋友圈。
第 2 个学生自己在一个朋友圈。所以返回 2。

示例 2:
输入:
[[1,1,0],
 [1,1,1],
 [0,1,1]]
输出: 1
说明：已知学生 0 和学生 1 互为朋友，学生 1 和学生 2 互为朋友，
所以学生 0 和学生 2 也是朋友，所以他们三个人在一个朋友圈，返回 1。

约束条件：
1 <= n <= 200
M[i][i] == 1
M[i][j] == M[j][i]

测试链接: https://leetcode.cn/problems/friend-circles/
相关平台: LeetCode 547, LintCode 1045, 牛客网, HackerRank
"""

```
class UnionFind:
    """
    并查集数据结构实现
    包含路径压缩和按秩合并优化
    """

    def __init__(self, n):
        """
        初始化并查集
        :param n: 节点数量
        """
        # parent[i]表示节点 i 的父节点
        self.parent = list(range(n))
        # rank[i]表示以 i 为根的树的高度上界
```

```
        self.rank = [1] * n
        # 当前集合数量
        self.set_count = n

    def find(self, x):
        """
        查找节点的根节点（代表元素）
        使用路径压缩优化
        :param x: 要查找的节点
        :return: 节点 x 所在集合的根节点
        """
        if self.parent[x] != x:
            # 路径压缩：将路径上的所有节点直接连接到根节点
            self.parent[x] = self.find(self.parent[x])
        return self.parent[x]

    def union(self, x, y):
        """
        合并两个集合
        使用按秩合并优化
        :param x: 第一个节点
        :param y: 第二个节点
        """
        root_x = self.find(x)
        root_y = self.find(y)

        # 如果已经在同一个集合中，则无需合并
        if root_x != root_y:
            # 按秩合并：将秩小的树合并到秩大的树下
            if self.rank[root_x] > self.rank[root_y]:
                self.parent[root_y] = root_x
            elif self.rank[root_x] < self.rank[root_y]:
                self.parent[root_x] = root_y
            else:
                # 秩相等时，任选一个作为根，并将其秩加 1
                self.parent[root_y] = root_x
                self.rank[root_x] += 1
            # 集合数量减 1
            self.set_count -= 1

    def is_connected(self, x, y):
        """
        判断两个节点是否在同一个集合中
```

```
        :param x: 第一个节点
        :param y: 第二个节点
        :return: 如果在同一个集合中返回 True，否则返回 False
        """
        return self.find(x) == self.find(y)

    def get_set_count(self):
        """
        获取当前集合数量
        :return: 集合数量
        """
        return self.set_count


def find_circle_num(M):
    """
    使用并查集解决朋友圈问题

    解题思路：
    1. 初始化并查集，每个学生初始时都是独立的集合
    2. 遍历朋友关系矩阵，如果两个人是朋友，则将他们所在的集合合并
    3. 最终集合的数量就是朋友圈的数量

    时间复杂度：O(N^2 * α(N))，其中 N 是学生数量，α 是阿克曼函数的反函数，近似为常数
    空间复杂度：O(N)

    :param M: 朋友关系矩阵
    :return: 朋友圈数量
    """
    if not M or not M[0]:
        return 0

    n = len(M)
    union_find = UnionFind(n)

    # 遍历矩阵的上三角部分（因为矩阵是对称的）
    for i in range(n):
        for j in range(i + 1, n):
            # 如果 i 和 j 是朋友，则合并他们的集合
            if M[i][j] == 1:
                union_find.union(i, j)

    # 返回集合数量，即朋友圈数量
```

```
    return union_find.get_set_count()


# 测试方法
if __name__ == "__main__":
    # 测试用例 1
    M1 = [
        [1, 1, 0],
        [1, 1, 0],
        [0, 0, 1]
    ]
    print("测试用例 1 结果:", find_circle_num(M1))  # 预期输出: 2

    # 测试用例 2
    M2 = [
        [1, 1, 0],
        [1, 1, 1],
        [0, 1, 1]
    ]
    print("测试用例 2 结果:", find_circle_num(M2))  # 预期输出: 1

    # 测试用例 3：单个学生
    M3 = [[1]]
    print("测试用例 3 结果:", find_circle_num(M3))  # 预期输出: 1

    # 测试用例 4：所有学生都互为朋友
    M4 = [
        [1, 1, 1],
        [1, 1, 1],
        [1, 1, 1]
    ]
    print("测试用例 4 结果:", find_circle_num(M4))  # 预期输出: 1
```

==========

文件: Code07_AccountsMerge.cpp

==========

```
// C++标准库头文件
#include <iostream>
#include <vector>
#include <string>
#include <unordered_map>
#include <set>
```

```
#include <algorithm>
using namespace std;

/**
 * 账户合并 (C++版本)
 * 给定一个列表 accounts，每个元素 accounts[i] 是一个字符串列表，其中第一个元素是名称 (name)，
 * 其余元素是邮箱地址表示该账户的邮箱地址。
 * 现在我们想合并这些账户。如果两个账户都有一些共同的邮箱地址，则两个账户必定属于同一个人。
 * 请注意，即使两个账户具有相同的名称，它们也可能属于不同的人，因为人们可能具有相同的名称。
 * 一个人最初可以拥有任意数量的账户，但其所有账户都具有相同的名称。
 * 合并账户后，按以下格式返回账户：每个账户的第一个元素是名称，其余元素是按字符 ASCII 顺序排列的
邮箱地址。
 * 账户本身可以以任意顺序返回。
 *
 * 示例 1:
 * 输入:
 * accounts = [
 *   ["John", "johnsmith@mail.com", "john00@mail.com"],
 *   ["John", "johnnybravo@mail.com"],
 *   ["John", "johnsmith@mail.com", "john_newyork@mail.com"],
 *   ["Mary", "mary@mail.com"]
 * ]
 * 输出:
 * [
 *   ["John", 'john00@mail.com', 'john_newyork@mail.com', 'johnsmith@mail.com'],
 *   ["John", "johnnybravo@mail.com"],
 *   ["Mary", "mary@mail.com"]
 * ]
 *
 * 解释:
 * 第一个和第三个 John 是同一个人，因为他们有共同的邮箱地址 "johnsmith@mail.com"。
 * 第二个 John 和 Mary 是不同的人，因为他们的邮箱地址没有被其他帐户使用。
 * 可以以任意顺序返回这些列表，例如答案
 * [['Mary', 'mary@mail.com'], ['John', 'johnnybravo@mail.com'],
 *  ['John', 'john00@mail.com', 'john_newyork@mail.com', 'johnsmith@mail.com']]
 *  也是正确的。
 *
 * 约束条件：
 * 1 <= accounts.length <= 1000
 * 2 <= accounts[i].length <= 10
 * 1 <= accounts[i][j].length <= 30
 * accounts[i][0] 由英文字母组成
 * accounts[i][j] (for j > 0) 是有效的邮箱地址
```

```
 *
 * 测试链接: https://leetcode.cn/problems/accounts-merge/
 * 相关平台: LeetCode 721, LintCode 1297, 牛客网
 */

class UnionFind {
private:
    vector<int> parent;  // parent[i]表示节点 i 的父节点

public:
    /**
     * 初始化并查集
     * @param n 节点数量
     */
    UnionFind(int n) {
        parent.resize(n);
        // 初始时每个节点都是自己的父节点
        for (int i = 0; i < n; i++) {
            parent[i] = i;
        }
    }

    /**
     * 查找节点的根节点（代表元素）
     * 使用路径压缩优化
     * @param x 要查找的节点
     * @return 节点 x 所在集合的根节点
     */
    int find(int x) {
        if (parent[x] != x) {
            // 路径压缩：将路径上的所有节点直接连接到根节点
            parent[x] = find(parent[x]);
        }
        return parent[x];
    }

    /**
     * 合并两个集合
     * @param x 第一个节点
     * @param y 第二个节点
     */
    void unionSets(int x, int y) {
        int rootX = find(x);
```

```
        int rootY = find(y);
        // 如果已经在同一个集合中，则无需合并
        if (rootX != rootY) {
            parent[rootX] = rootY;
        }
    }
};

class Solution {
public:
    /**
     * 使用并查集解决账户合并问题
     *
     * 解题思路：
     * 1. 将每个账户看作一个节点，使用并查集来维护账户之间的连接关系
     * 2. 对于每个邮箱地址，记录它第一次出现的账户索引
     * 3. 如果一个邮箱地址在多个账户中出现，则将这些账户合并到同一个集合中
     * 4. 最后，将同一个集合中的所有账户的邮箱地址合并，并按 ASCII 顺序排序
     *
     * 时间复杂度：O(N * M * α(N))，其中 N 是账户数量，M 是每个账户的平均邮箱数量，α 是阿克曼函数的反函数
     * 空间复杂度：O(N * M)
     *
     * @param accounts 账户列表
     * @return 合并后的账户列表
     */
    static vector<vector<string>> accountsMerge(vector<vector<string>>& accounts) {
        if (accounts.empty()) {
            return {};
        }

        int n = accounts.size();
        UnionFind unionFind(n);

        // 记录每个邮箱第一次出现的账户索引
        unordered_map<string, int> emailToIndex;

        // 遍历所有账户
        for (int i = 0; i < n; i++) {
            // 从索引 1 开始，因为索引 0 是账户名称
            for (int j = 1; j < accounts[i].size(); j++) {
                string email = accounts[i][j];
                if (emailToIndex.count(email)) {
```

```
                // 如果邮箱已经出现过，则合并当前账户和之前出现该邮箱的账户
                int prevIndex = emailToIndex[email];
                unionFind.unionSets(i, prevIndex);
            } else {
                // 记录邮箱第一次出现的账户索引
                emailToIndex[email] = i;
            }
        }
    }

    // 将同一个集合中的所有邮箱地址合并到一起
    unordered_map<int, set<string>> indexToEmails;
    for (int i = 0; i < n; i++) {
        int root = unionFind.find(i);
        for (int j = 1; j < accounts[i].size(); j++) {
            indexToEmails[root].insert(accounts[i][j]);
        }
    }

    // 构造结果
    vector<vector<string>> result;
    for (auto& entry : indexToEmails) {
        int index = entry.first;
        set<string>& emails = entry.second;

        vector<string> mergedAccount;
        // 添加账户名称
        mergedAccount.push_back(accounts[index][0]);
        // 添加排序后的邮箱地址
        for (const string& email : emails) {
            mergedAccount.push_back(email);
        }
        result.push_back(mergedAccount);
    }

    return result;
  }
};

// 测试方法
int main() {
    // 测试用例 1
    vector<vector<string>> accounts1 = {
```

```
        {"John", "johnsmith@mail.com", "john00@mail.com"},
        {"John", "johnnybravo@mail.com"},
        {"John", "johnsmith@mail.com", "john_newyork@mail.com"},
        {"Mary", "mary@mail.com"}
    };

    cout << "测试用例 1 结果:" << endl;
    vector<vector<string>> result1 = Solution::accountsMerge(accounts1);
    for (const auto& account : result1) {
        cout << "[";
        for (int i = 0; i < account.size(); i++) {
            cout << "\"" << account[i] << "\"";
            if (i < account.size() - 1) cout << ", ";
        }
        cout << "]" << endl;
    }

    // 测试用例 2
    vector<vector<string>> accounts2 = {
        {"Gabe","Gabe0@m.co","Gabe3@m.co","Gabe1@m.co"},
        {"Kevin","Kevin3@m.co","Kevin5@m.co","Kevin0@m.co"},
        {"Ethan","Ethan5@m.co","Ethan4@m.co","Ethan0@m.co"},
        {"Hanzo","Hanzo3@m.co","Hanzo1@m.co","Hanzo0@m.co"},
        {"Fern","Fern5@m.co","Fern1@m.co","Fern0@m.co"}
    };

    cout << "\n 测试用例 2 结果:" << endl;
    vector<vector<string>> result2 = Solution::accountsMerge(accounts2);
    for (const auto& account : result2) {
        cout << "[";
        for (int i = 0; i < account.size(); i++) {
            cout << "\"" << account[i] << "\"";
            if (i < account.size() - 1) cout << ", ";
        }
        cout << "]" << endl;
    }

    return 0;
}
```

==========

文件: Code07_AccountsMerge.java

```
==========================================
package class056;

import java.util.*;

/**
 * 账户合并
 * 给定一个列表 accounts，每个元素 accounts[i] 是一个字符串列表，其中第一个元素是名称 (name)，
 * 其余元素是邮箱地址表示该账户的邮箱地址。
 * 现在我们想合并这些账户。如果两个账户都有一些共同的邮箱地址，则两个账户必定属于同一个人。
 * 请注意，即使两个账户具有相同的名称，它们也可能属于不同的人，因为人们可能具有相同的名称。
 * 一个人最初可以拥有任意数量的账户，但其所有账户都具有相同的名称。
 * 合并账户后，按以下格式返回账户：每个账户的第一个元素是名称，其余元素是按字符 ASCII 顺序排列的
邮箱地址。
 * 账户本身可以以任意顺序返回。
 *
 * 示例 1:
 * 输入:
 * accounts = [
 *   ["John", "johnsmith@mail.com", "john00@mail.com"],
 *   ["John", "johnnybravo@mail.com"],
 *   ["John", "johnsmith@mail.com", "john_newyork@mail.com"],
 *   ["Mary", "mary@mail.com"]
 * ]
 * 输出:
 * [
 *   ["John", 'john00@mail.com', 'john_newyork@mail.com', 'johnsmith@mail.com'],
 *   ["John", "johnnybravo@mail.com"],
 *   ["Mary", "mary@mail.com"]
 * ]
 *
 * 解释:
 * 第一个和第三个 John 是同一个人，因为他们有共同的邮箱地址 "johnsmith@mail.com"。
 * 第二个 John 和 Mary 是不同的人，因为他们的邮箱地址没有被其他帐户使用。
 * 可以以任意顺序返回这些列表，例如答案
 * [['Mary', 'mary@mail.com'], ['John', 'johnnybravo@mail.com'],
 *  ['John', 'john00@mail.com', 'john_newyork@mail.com', 'johnsmith@mail.com']]
 *  也是正确的。
 *
 * 约束条件：
 * 1 <= accounts.length <= 1000
 * 2 <= accounts[i].length <= 10
 * 1 <= accounts[i][j].length <= 30
```

```
 * accounts[i][0] 由英文字母组成
 * accounts[i][j] (for j > 0) 是有效的邮箱地址
 *
 * 测试链接: https://leetcode.cn/problems/accounts-merge/
 * 相关平台: LeetCode 721, LintCode 1297, 牛客网
 */
public class Code07_AccountsMerge {

    /**
     * 使用并查集解决账户合并问题
     *
     * 解题思路:
     * 1. 将每个账户看作一个节点，使用并查集来维护账户之间的连接关系
     * 2. 对于每个邮箱地址，记录它第一次出现的账户索引
     * 3. 如果一个邮箱地址在多个账户中出现，则将这些账户合并到同一个集合中
     * 4. 最后，将同一个集合中的所有账户的邮箱地址合并，并按 ASCII 顺序排序
     *
     * 时间复杂度：O(N * M * α(N))，其中 N 是账户数量，M 是每个账户的平均邮箱数量，α 是阿克曼函数的反函数
     * 空间复杂度：O(N * M)
     *
     * @param accounts 账户列表
     * @return 合并后的账户列表
     */
    public static List<List<String>> accountsMerge(List<List<String>> accounts) {
        if (accounts == null || accounts.isEmpty()) {
            return new ArrayList<>();
        }

        int n = accounts.size();
        UnionFind unionFind = new UnionFind(n);

        // 记录每个邮箱第一次出现的账户索引
        Map<String, Integer> emailToIndex = new HashMap<>();

        // 遍历所有账户
        for (int i = 0; i < n; i++) {
            List<String> account = accounts.get(i);
            // 从索引 1 开始，因为索引 0 是账户名称
            for (int j = 1; j < account.size(); j++) {
                String email = account.get(j);
                if (emailToIndex.containsKey(email)) {
                    // 如果邮箱已经出现过，则合并当前账户和之前出现该邮箱的账户
```

```
                int prevIndex = emailToIndex.get(email);
                unionFind.union(i, prevIndex);
            } else {
                // 记录邮箱第一次出现的账户索引
                emailToIndex.put(email, i);
            }
        }
    }

    // 将同一个集合中的所有邮箱地址合并到一起
    Map<Integer, Set<String>> indexToEmails = new HashMap<>();
    for (int i = 0; i < n; i++) {
        int root = unionFind.find(i);
        if (!indexToEmails.containsKey(root)) {
            indexToEmails.put(root, new TreeSet<>()); // 使用 TreeSet 自动排序
        }
        List<String> account = accounts.get(i);
        for (int j = 1; j < account.size(); j++) {
            indexToEmails.get(root).add(account.get(j));
        }
    }

    // 构造结果
    List<List<String>> result = new ArrayList<>();
    for (Map.Entry<Integer, Set<String>> entry : indexToEmails.entrySet()) {
        int index = entry.getKey();
        Set<String> emails = entry.getValue();

        List<String> mergedAccount = new ArrayList<>();
        // 添加账户名称
        mergedAccount.add(accounts.get(index).get(0));
        // 添加排序后的邮箱地址
        mergedAccount.addAll(emails);
        result.add(mergedAccount);
    }

    return result;
}

/**
 * 并查集数据结构实现
 * 包含路径压缩优化
 */
```

```
static class UnionFind {
    private int[] parent;  // parent[i]表示节点 i 的父节点

    /**
     * 初始化并查集
     * @param n 节点数量
     */
    public UnionFind(int n) {
        parent = new int[n];
        // 初始时每个节点都是自己的父节点
        for (int i = 0; i < n; i++) {
            parent[i] = i;
        }
    }

    /**
     * 查找节点的根节点（代表元素）
     * 使用路径压缩优化
     * @param x 要查找的节点
     * @return 节点 x 所在集合的根节点
     */
    public int find(int x) {
        if (parent[x] != x) {
            // 路径压缩：将路径上的所有节点直接连接到根节点
            parent[x] = find(parent[x]);
        }
        return parent[x];
    }

    /**
     * 合并两个集合
     * @param x 第一个节点
     * @param y 第二个节点
     */
    public void union(int x, int y) {
        int rootX = find(x);
        int rootY = find(y);
        // 如果已经在同一个集合中，则无需合并
        if (rootX != rootY) {
            parent[rootX] = rootY;
        }
    }
}
```

```
    // 测试方法
    public static void main(String[] args) {
        // 测试用例 1
        List<List<String>> accounts1 = new ArrayList<>();
        accounts1.add(Arrays.asList("John", "johnsmith@mail.com", "john00@mail.com"));
        accounts1.add(Arrays.asList("John", "johnnybravo@mail.com"));
        accounts1.add(Arrays.asList("John", "johnsmith@mail.com", "john_newyork@mail.com"));
        accounts1.add(Arrays.asList("Mary", "mary@mail.com"));

        System.out.println("测试用例 1 结果:");
        List<List<String>> result1 = accountsMerge(accounts1);
        for (List<String> account : result1) {
            System.out.println(account);
        }

        // 测试用例 2
        List<List<String>> accounts2 = new ArrayList<>();
        accounts2.add(Arrays.asList("Gabe","Gabe0@m.co","Gabe3@m.co","Gabe1@m.co"));
        accounts2.add(Arrays.asList("Kevin","Kevin3@m.co","Kevin5@m.co","Kevin0@m.co"));
        accounts2.add(Arrays.asList("Ethan","Ethan5@m.co","Ethan4@m.co","Ethan0@m.co"));
        accounts2.add(Arrays.asList("Hanzo","Hanzo3@m.co","Hanzo1@m.co","Hanzo0@m.co"));
        accounts2.add(Arrays.asList("Fern","Fern5@m.co","Fern1@m.co","Fern0@m.co"));

        System.out.println("\n 测试用例 2 结果:");
        List<List<String>> result2 = accountsMerge(accounts2);
        for (List<String> account : result2) {
            System.out.println(account);
        }
    }
}
```

==========

文件: Code07_AccountsMerge.py

==========

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

"""
账户合并 (Python 版本)
给定一个列表 accounts，每个元素 accounts[i] 是一个字符串列表，其中第一个元素是名称 (name)，
其余元素是邮箱地址表示该账户的邮箱地址。
```

现在我们想合并这些账户。如果两个账户都有一些共同的邮箱地址，则两个账户必定属于同一个人。
请注意，即使两个账户具有相同的名称，它们也可能属于不同的人，因为人们可能具有相同的名称。
一个人最初可以拥有任意数量的账户，但其所有账户都具有相同的名称。
合并账户后，按以下格式返回账户：每个账户的第一个元素是名称，其余元素是按字符 ASCII 顺序排列的邮箱地址。
账户本身可以以任意顺序返回。

示例 1:
输入:
```
accounts = [
  ["John", "johnsmith@mail.com", "john00@mail.com"],
  ["John", "johnnybravo@mail.com"],
  ["John", "johnsmith@mail.com", "john_newyork@mail.com"],
  ["Mary", "mary@mail.com"]
]
```
输出:
```
[
  ["John", 'john00@mail.com', 'john_newyork@mail.com', 'johnsmith@mail.com'],
  ["John", "johnnybravo@mail.com"],
  ["Mary", "mary@mail.com"]
]
```

解释:
第一个和第三个 John 是同一个人，因为他们有共同的邮箱地址 "johnsmith@mail.com"。
第二个 John 和 Mary 是不同的人，因为他们的邮箱地址没有被其他帐户使用。
可以以任意顺序返回这些列表，例如答案
[['Mary', 'mary@mail.com'], ['John', 'johnnybravo@mail.com'],
 ['John', 'john00@mail.com', 'john_newyork@mail.com', 'johnsmith@mail.com']]
 也是正确的。

约束条件：
1 <= accounts.length <= 1000
2 <= accounts[i].length <= 10
1 <= accounts[i][j].length <= 30
accounts[i][0] 由英文字母组成
accounts[i][j] (for j > 0) 是有效的邮箱地址

测试链接: https://leetcode.cn/problems/accounts-merge/
相关平台: LeetCode 721, LintCode 1297, 牛客网
"""

```
class UnionFind:
```

```
    """
    并查集数据结构实现
    包含路径压缩优化
    """

    def __init__(self, n):
        """
        初始化并查集
        :param n: 节点数量
        """
        # parent[i]表示节点 i 的父节点
        self.parent = list(range(n))

    def find(self, x):
        """
        查找节点的根节点（代表元素）
        使用路径压缩优化
        :param x: 要查找的节点
        :return: 节点 x 所在集合的根节点
        """
        if self.parent[x] != x:
            # 路径压缩：将路径上的所有节点直接连接到根节点
            self.parent[x] = self.find(self.parent[x])
        return self.parent[x]

    def union(self, x, y):
        """
        合并两个集合
        :param x: 第一个节点
        :param y: 第二个节点
        """
        root_x = self.find(x)
        root_y = self.find(y)
        # 如果已经在同一个集合中，则无需合并
        if root_x != root_y:
            self.parent[root_x] = root_y


def accounts_merge(accounts):
    """
    使用并查集解决账户合并问题

    解题思路：
```

1. 将每个账户看作一个节点，使用并查集来维护账户之间的连接关系
2. 对于每个邮箱地址，记录它第一次出现的账户索引
3. 如果一个邮箱地址在多个账户中出现，则将这些账户合并到同一个集合中
4. 最后，将同一个集合中的所有账户的邮箱地址合并，并按 ASCII 顺序排序

时间复杂度：O(N * M * α(N))，其中 N 是账户数量，M 是每个账户的平均邮箱数量，α 是阿克曼函数的反函数

空间复杂度：O(N * M)

```
    :param accounts: 账户列表
    :return: 合并后的账户列表
    """
    if not accounts:
        return []

    n = len(accounts)
    union_find = UnionFind(n)

    # 记录每个邮箱第一次出现的账户索引
    email_to_index = {}

    # 遍历所有账户
    for i in range(n):
        account = accounts[i]
        # 从索引 1 开始，因为索引 0 是账户名称
        for j in range(1, len(account)):
            email = account[j]
            if email in email_to_index:
                # 如果邮箱已经出现过，则合并当前账户和之前出现该邮箱的账户
                prev_index = email_to_index[email]
                union_find.union(i, prev_index)
            else:
                # 记录邮箱第一次出现的账户索引
                email_to_index[email] = i

    # 将同一个集合中的所有邮箱地址合并到一起
    index_to_emails = {}
    for i in range(n):
        root = union_find.find(i)
        if root not in index_to_emails:
            index_to_emails[root] = set()
        account = accounts[i]
        for j in range(1, len(account)):
```

```
            index_to_emails[root].add(account[j])

    # 构造结果
    result = []
    for index, emails in index_to_emails.items():
        # 添加账户名称
        merged_account = [accounts[index][0]]
        # 添加排序后的邮箱地址
        merged_account.extend(sorted(emails))
        result.append(merged_account)

    return result


# 测试方法
if __name__ == "__main__":
    # 测试用例 1
    accounts1 = [
        ["John", "johnsmith@mail.com", "john00@mail.com"],
        ["John", "johnnybravo@mail.com"],
        ["John", "johnsmith@mail.com", "john_newyork@mail.com"],
        ["Mary", "mary@mail.com"]
    ]

    print("测试用例 1 结果:")
    result1 = accounts_merge(accounts1)
    for account in result1:
        print(account)

    # 测试用例 2
    accounts2 = [
        ["Gabe","Gabe0@m.co","Gabe3@m.co","Gabe1@m.co"],
        ["Kevin","Kevin3@m.co","Kevin5@m.co","Kevin0@m.co"],
        ["Ethan","Ethan5@m.co","Ethan4@m.co","Ethan0@m.co"],
        ["Hanzo","Hanzo3@m.co","Hanzo1@m.co","Hanzo0@m.co"],
        ["Fern","Fern5@m.co","Fern1@m.co","Fern0@m.co"]
    ]

    print("\n 测试用例 2 结果:")
    result2 = accounts_merge(accounts2)
    for account in result2:
        print(account)
```

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

文件: Code08_RedundantConnection.cpp
==========================================
// C++标准库头文件
#include <iostream>
#include <vector>
using namespace std;

/**
 * 冗余连接 (C++版本)
 * 树可以看成是一个连通且无环的无向图。
 * 给定往一棵 n 个节点(节点值 1～n)的树中添加一条边后的图。
 * 添加的边的两个顶点包含在 1 到 n 中间，且这条附加的边不属于树中已存在的边。
 * 图的信息记录于长度为 n 的二维数组 edges，edges[i] = [ai, bi]表示图中在 ai 和 bi 之间存在一条边。
 * 请找出一条可以删去的边，删除后可使得剩余部分是一棵有 n 个节点的树。
 * 如果有多个答案，则返回数组 edges 中最后出现的边。
 *
 * 示例 1:
 * 输入: edges = [[1,2],[1,3],[2,3]]
 * 输出: [2,3]
 *
 * 示例 2:
 * 输入: edges = [[1,2],[2,3],[3,4],[1,4],[1,5]]
 * 输出: [1,4]
 *
 * 约束条件:
 * n == edges.length
 * 3 <= n <= 1000
 * edges[i].length == 2
 * 1 <= ai < bi <= edges.length
 * ai != bi
 * edges 中无重复元素
 * 给定的图是连通的
 *
 * 测试链接: https://leetcode.cn/problems/redundant-connection/
 * 相关平台: LeetCode 684, LintCode 1048, 牛客网
 */

class UnionFind {
private:
    vector<int> parent;  // parent[i]表示节点 i 的父节点
```

```
public:
    /**
     * 初始化并查集
     * @param n 节点数量
     */
    UnionFind(int n) {
        parent.resize(n);
        // 初始时每个节点都是自己的父节点
        for (int i = 0; i < n; i++) {
            parent[i] = i;
        }
    }

    /**
     * 查找节点的根节点（代表元素）
     * 使用路径压缩优化
     * @param x 要查找的节点
     * @return 节点 x 所在集合的根节点
     */
    int find(int x) {
        if (parent[x] != x) {
            // 路径压缩：将路径上的所有节点直接连接到根节点
            parent[x] = find(parent[x]);
        }
        return parent[x];
    }

    /**
     * 合并两个集合
     * @param x 第一个节点
     * @param y 第二个节点
     */
    void unionSets(int x, int y) {
        int rootX = find(x);
        int rootY = find(y);
        // 如果已经在同一个集合中，则无需合并
        if (rootX != rootY) {
            parent[rootX] = rootY;
        }
    }

    /**
     * 判断两个节点是否在同一个集合中
```

```
     * @param x 第一个节点
     * @param y 第二个节点
     * @return 如果在同一个集合中返回 true，否则返回 false
     */
    bool isConnected(int x, int y) {
        return find(x) == find(y);
    }
};

/**
 * 使用并查集解决冗余连接问题
 *
 * 解题思路：
 * 1. 遍历所有的边，对于每条边的两个顶点：
 *    - 如果它们已经在同一个连通分量中，说明添加这条边会形成环，这就是我们要找的冗余边
 *    - 否则，将这两个顶点所在的连通分量合并
 * 2. 由于题目要求返回最后出现的冗余边，我们按顺序遍历边，找到第一条形成环的边即可
 *
 * 时间复杂度：O(N * α(N))，其中 N 是边的数量，α 是阿克曼函数的反函数，近似为常数
 * 空间复杂度：O(N)
 *
 * @param edges 边的数组
 * @return 冗余的边
 */
vector<int> findRedundantConnection(vector<vector<int>>& edges) {
    if (edges.empty()) {
        return {};
    }

    // 节点数量等于边的数量（因为是树加一条边）
    int n = edges.size();
    UnionFind unionFind(n + 1); // 节点编号从 1 开始，所以需要 n+1 个位置

    // 遍历所有的边
    for (auto& edge : edges) {
        int node1 = edge[0];
        int node2 = edge[1];

        // 如果两个节点已经在同一个连通分量中，说明添加这条边会形成环
        if (unionFind.isConnected(node1, node2)) {
            return edge; // 返回这条冗余的边
        } else {
            // 否则将两个节点所在的连通分量合并
```

```
            unionFind.unionSets(node1, node2);
        }
    }

    // 根据题目保证，一定会有一条冗余边，所以这里不会执行到
    return {};
}

// 主函数用于测试
int main() {
    // 测试用例 1
    vector<vector<int>> edges1 = {{1,2}, {1,3}, {2,3}};
    vector<int> result1 = findRedundantConnection(edges1);
    cout << "测试用例 1 结果: [" << result1[0] << ", " << result1[1] << "]" << endl;

    // 测试用例 2
    vector<vector<int>> edges2 = {{1,2}, {2,3}, {3,4}, {1,4}, {1,5}};
    vector<int> result2 = findRedundantConnection(edges2);
    cout << "测试用例 2 结果: [" << result2[0] << ", " << result2[1] << "]" << endl;

    return 0;
}
```

==========

文件: Code08_RedundantConnection.java

==========

```
package class056;

import java.util.*;

/**
 * 冗余连接
 * 树可以看成是一个连通且无环的无向图。
 * 给定往一棵 n 个节点(节点值 1～n)的树中添加一条边后的图。
 * 添加的边的两个顶点包含在 1 到 n 中间，且这条附加的边不属于树中已存在的边。
 * 图的信息记录于长度为 n 的二维数组 edges，edges[i] = [ai, bi]表示图中在 ai 和 bi 之间存在一条边。
 * 请找出一条可以删去的边，删除后可使得剩余部分是一棵有 n 个节点的树。
 * 如果有多个答案，则返回数组 edges 中最后出现的边。
 *
 * 示例 1:
 * 输入: edges = [[1,2],[1,3],[2,3]]
 * 输出: [2,3]
```

```
 *
 * 示例 2:
 * 输入: edges = [[1,2],[2,3],[3,4],[1,4],[1,5]]
 * 输出: [1,4]
 *
 * 约束条件:
 * n == edges.length
 * 3 <= n <= 1000
 * edges[i].length == 2
 * 1 <= ai < bi <= edges.length
 * ai != bi
 * edges 中无重复元素
 * 给定的图是连通的
 *
 * 测试链接: https://leetcode.cn/problems/redundant-connection/
 * 相关平台: LeetCode 684, LintCode 1048, 牛客网
 */
public class Code08_RedundantConnection {

    /**
     * 使用并查集解决冗余连接问题
     *
     * 解题思路:
     * 1. 遍历所有的边，对于每条边的两个顶点:
     *    - 如果它们已经在同一个连通分量中，说明添加这条边会形成环，这就是我们要找的冗余边
     *    - 否则，将这两个顶点所在的连通分量合并
     * 2. 由于题目要求返回最后出现的冗余边，我们按顺序遍历边，找到第一条形成环的边即可
     *
     * 时间复杂度：O(N * α(N))，其中 N 是边的数量，α 是阿克曼函数的反函数
     * 空间复杂度：O(N)
     *
     * @param edges 边的数组
     * @return 冗余的边
     */
    public static int[] findRedundantConnection(int[][] edges) {
        if (edges == null || edges.length == 0) {
            return new int[0];
        }

        // 节点数量等于边的数量（因为是树加一条边）
        int n = edges.length;
        UnionFind unionFind = new UnionFind(n + 1); // 节点编号从 1 开始，所以需要 n+1 个位置
```

```
    // 遍历所有的边
    for (int[] edge : edges) {
        int node1 = edge[0];
        int node2 = edge[1];

        // 如果两个节点已经在同一个连通分量中，说明添加这条边会形成环
        if (unionFind.isConnected(node1, node2)) {
            return edge; // 返回这条冗余的边
        } else {
            // 否则将两个节点所在的连通分量合并
            unionFind.union(node1, node2);
        }
    }

    // 根据题目保证，一定会有一条冗余边，所以这里不会执行到
    return new int[0];
}

/**
 * 并查集数据结构实现
 * 包含路径压缩优化
 */
static class UnionFind {
    private int[] parent;  // parent[i]表示节点 i 的父节点

    /**
     * 初始化并查集
     * @param n 节点数量
     */
    public UnionFind(int n) {
        parent = new int[n];
        // 初始时每个节点都是自己的父节点
        for (int i = 0; i < n; i++) {
            parent[i] = i;
        }
    }

    /**
     * 查找节点的根节点（代表元素）
     * 使用路径压缩优化
     * @param x 要查找的节点
     * @return 节点 x 所在集合的根节点
     */
```

```
        public int find(int x) {
            if (parent[x] != x) {
                // 路径压缩：将路径上的所有节点直接连接到根节点
                parent[x] = find(parent[x]);
            }
            return parent[x];
        }

        /**
         * 合并两个集合
         * @param x 第一个节点
         * @param y 第二个节点
         */
        public void union(int x, int y) {
            int rootX = find(x);
            int rootY = find(y);
            // 如果已经在同一个集合中，则无需合并
            if (rootX != rootY) {
                parent[rootX] = rootY;
            }
        }

        /**
         * 判断两个节点是否在同一个集合中
         * @param x 第一个节点
         * @param y 第二个节点
         * @return 如果在同一个集合中返回 true，否则返回 false
         */
        public boolean isConnected(int x, int y) {
            return find(x) == find(y);
        }
    }

    // 测试方法
    public static void main(String[] args) {
        // 测试用例 1
        int[][] edges1 = {{1,2},{1,3},{2,3}};
        int[] result1 = findRedundantConnection(edges1);
        System.out.println("测试用例 1 结果: [" + result1[0] + ", " + result1[1] + "]"); // 预期输
出: [2, 3]

        // 测试用例 2
        int[][] edges2 = {{1,2},{2,3},{3,4},{1,4},{1,5}};
```

```
        int[] result2 = findRedundantConnection(edges2);
        System.out.println("测试用例 2 结果: [" + result2[0] + ", " + result2[1] + "]"); // 预期输出: [1, 4]

        // 测试用例 3
        int[][] edges3 = {{1,2},{1,3},{1,4},{3,4},{4,5}};
        int[] result3 = findRedundantConnection(edges3);
        System.out.println("测试用例 3 结果: [" + result3[0] + ", " + result3[1] + "]"); // 预期输出: [3, 4]
    }
}
```

==========

文件: Code08_RedundantConnection.py

==========

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

"""
冗余连接 (Python 版本)
树可以看成是一个连通且无环的无向图。
给定往一棵 n 个节点(节点值 1～n)的树中添加一条边后的图。
添加的边的两个顶点包含在 1 到 n 中间，且这条附加的边不属于树中已存在的边。
图的信息记录于长度为 n 的二维数组 edges，edges[i] = [ai, bi]表示图中在 ai 和 bi 之间存在一条边。
请找出一条可以删去的边，删除后可使得剩余部分是一棵有 n 个节点的树。
如果有多个答案，则返回数组 edges 中最后出现的边。

示例 1:
输入: edges = [[1,2],[1,3],[2,3]]
输出: [2,3]

示例 2:
输入: edges = [[1,2],[2,3],[3,4],[1,4],[1,5]]
输出: [1,4]

约束条件:
n == edges.length
3 <= n <= 1000
edges[i].length == 2
1 <= ai < bi <= edges.length
ai != bi
edges 中无重复元素
```

给定的图是连通的

测试链接: https://leetcode.cn/problems/redundant-connection/
相关平台: LeetCode 684, LintCode 1048, 牛客网
"""

```
class UnionFind:
    """
    并查集数据结构实现
    包含路径压缩优化
    """

    def __init__(self, n):
        """
        初始化并查集
        :param n: 节点数量
        """
        # parent[i]表示节点 i 的父节点
        self.parent = list(range(n))

    def find(self, x):
        """
        查找节点的根节点（代表元素）
        使用路径压缩优化
        :param x: 要查找的节点
        :return: 节点 x 所在集合的根节点
        """
        if self.parent[x] != x:
            # 路径压缩：将路径上的所有节点直接连接到根节点
            self.parent[x] = self.find(self.parent[x])
        return self.parent[x]

    def union(self, x, y):
        """
        合并两个集合
        :param x: 第一个节点
        :param y: 第二个节点
        """
        root_x = self.find(x)
        root_y = self.find(y)
        # 如果已经在同一个集合中，则无需合并
        if root_x != root_y:
```

```
            self.parent[root_x] = root_y

    def is_connected(self, x, y):
        """
        判断两个节点是否在同一个集合中
        :param x: 第一个节点
        :param y: 第二个节点
        :return: 如果在同一个集合中返回 True，否则返回 False
        """
        return self.find(x) == self.find(y)


def find_redundant_connection(edges):
    """
    使用并查集解决冗余连接问题

    解题思路：
    1. 遍历所有的边，对于每条边的两个顶点：
       - 如果它们已经在同一个连通分量中，说明添加这条边会形成环，这就是我们要找的冗余边
       - 否则，将这两个顶点所在的连通分量合并
    2. 由于题目要求返回最后出现的冗余边，我们按顺序遍历边，找到第一条形成环的边即可

    时间复杂度：O(N * α(N))，其中 N 是边的数量，α 是阿克曼函数的反函数
    空间复杂度：O(N)

    :param edges: 边的数组
    :return: 冗余的边
    """
    if not edges:
        return []

    # 节点数量等于边的数量（因为是树加一条边）
    n = len(edges)
    union_find = UnionFind(n + 1)  # 节点编号从 1 开始，所以需要 n+1 个位置

    # 遍历所有的边
    for edge in edges:
        node1, node2 = edge[0], edge[1]

        # 如果两个节点已经在同一个连通分量中，说明添加这条边会形成环
        if union_find.is_connected(node1, node2):
            return edge  # 返回这条冗余的边
        else:
```

```
            # 否则将两个节点所在的连通分量合并
            union_find.union(node1, node2)

    # 根据题目保证，一定会有一条冗余边，所以这里不会执行到
    return []


# 测试方法
if __name__ == "__main__":
    # 测试用例 1
    edges1 = [[1,2],[1,3],[2,3]]
    result1 = find_redundant_connection(edges1)
    print(f"测试用例 1 结果: [{result1[0]}, {result1[1]}]")  # 预期输出: [2, 3]

    # 测试用例 2
    edges2 = [[1,2],[2,3],[3,4],[1,4],[1,5]]
    result2 = find_redundant_connection(edges2)
    print(f"测试用例 2 结果: [{result2[0]}, {result2[1]}]")  # 预期输出: [1, 4]

    # 测试用例 3
    edges3 = [[1,2],[1,3],[1,4],[3,4],[4,5]]
    result3 = find_redundant_connection(edges3)
    print(f"测试用例 3 结果: [{result3[0]}, {result3[1]}]")  # 预期输出: [3, 4]

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

文件: Code09_SentenceSimilarityII.cpp
==========================================
// C++标准库头文件
#include <iostream>
#include <vector>
#include <string>
#include <unordered_map>
using namespace std;

/**
 * 句子相似性 II (C++版本)
 * 我们可以将一个句子表示为一个单词数组，例如，句子 "I am happy with leetcode" 可以表示为 arr = ["I","am","happy","with","leetcode"]。
 * 给定两个句子 sentence1 和 sentence2 分别表示为一个字符串数组，并给定一个字符串对的列表 similarPairs，其中 similarPairs[i] = [xi, yi] 表示两个单词 xi 和 yi 是相似的。
 * 如果两个句子句子长度相同，且在对应位置上的单词要么相同，要么通过相似单词对列表中的相似关系（可能是间接的相似关系）变得相似，则认为这两个句子是相似的。
```

```
 * 返回 true 如果 sentence1 和 sentence2 是相似的。
 *
 * 示例 1:
 * 输入:
 * sentence1 = ["great","acting","skills"]
 * sentence2 = ["fine","drama","talent"]
 * similarPairs = [["great","good"],["fine","good"],["drama","acting"],["skills","talent"]]
 * 输出: true
 * 解释: "great" 和 "fine" 相似, "acting" 和 "drama" 相似, "skills" 和 "talent" 相似。
 *
 * 示例 2:
 * 输入:
 * sentence1 = ["great"]
 * sentence2 = ["great"]
 * similarPairs = []
 * 输出: true
 * 解释: 句子长度相同且单词相同。
 *
 * 示例 3:
 * 输入:
 * sentence1 = ["great"]
 * sentence2 = ["doubleplus","good"]
 * similarPairs = [["great","doubleplus"]]
 * 输出: false
 * 解释: 句子长度不同。
 *
 * 约束条件:
 * 1 <= sentence1.length, sentence2.length <= 1000
 * 1 <= sentence1[i].length, sentence2[i].length <= 20
 * sentence1[i] 和 sentence2[i] 仅包含大小写英文字母
 * 0 <= similarPairs.length <= 2000
 * similarPairs[i].length == 2
 * 1 <= xi.length, yi.length <= 20
 * xi 和 yi 仅包含大小写英文字母
 * 所有相似单词对都是不同的
 *
 * 测试链接: https://leetcode.cn/problems/sentence-similarity-ii/
 * 相关平台: LeetCode 737
 */

class UnionFind {
private:
    unordered_map<string, string> parent;  // 使用哈希表存储单词到其父节点的映射
```

```
public:
    /**
     * 查找单词所在集合的根节点
     * 使用路径压缩优化
     * @param word 要查找的单词
     * @return 单词所在集合的根节点
     */
    string find(const string& word) {
        // 如果单词不存在于 parent 中，将其添加到 parent 中，并将其父节点设为自身
        if (parent.find(word) == parent.end()) {
            parent[word] = word;
        }
        // 路径压缩：将路径上的所有节点直接连接到根节点
        if (parent[word] != word) {
            parent[word] = find(parent[word]);
        }
        return parent[word];
    }

    /**
     * 合并两个单词所在的集合
     * @param word1 第一个单词
     * @param word2 第二个单词
     */
    void unionSets(const string& word1, const string& word2) {
        string root1 = find(word1);
        string root2 = find(word2);
        // 如果两个单词不在同一个集合中，则合并它们
        if (root1 != root2) {
            parent[root1] = root2;
        }
    }

    /**
     * 判断两个单词是否在同一个集合中
     * @param word1 第一个单词
     * @param word2 第二个单词
     * @return 如果在同一个集合中返回 true，否则返回 false
     */
    bool isConnected(const string& word1, const string& word2) {
        return find(word1) == find(word2);
    }
```

```
};

/**
 * 判断两个句子是否相似
 *
 * 解题思路：
 * 1. 首先检查两个句子的长度是否相同，如果不同则直接返回 false
 * 2. 使用并查集将相似单词对中的单词合并到同一个集合中
 * 3. 遍历两个句子的对应位置单词，检查它们是否相同或者属于同一个集合
 *
 * 时间复杂度：O(P + N)，其中 P 是 similarPairs 的长度，N 是句子的长度，
 *          并查集操作的时间复杂度接近 O(1)（路径压缩优化）
 * 空间复杂度：O(P)，用于存储并查集
 *
 * @param sentence1 第一个句子
 * @param sentence2 第二个句子
 * @param similarPairs 相似单词对列表
 * @return 如果两个句子相似返回 true，否则返回 false
 */
bool areSentencesSimilarTwo(vector<string>& sentence1, vector<string>& sentence2,
vector<vector<string>>& similarPairs) {
    // 检查两个句子的长度是否相同
    if (sentence1.size() != sentence2.size()) {
        return false;
    }

    // 初始化并查集
    UnionFind unionFind;

    // 将所有相似单词对合并到同一个集合中
    for (const auto& pair : similarPairs) {
        unionFind.unionSets(pair[0], pair[1]);
    }

    // 遍历两个句子的对应位置单词进行比较
    for (int i = 0; i < sentence1.size(); i++) {
        const string& word1 = sentence1[i];
        const string& word2 = sentence2[i];

        // 如果单词相同，或者属于同一个集合，则继续比较下一对单词
        if (word1 == word2 || unionFind.isConnected(word1, word2)) {
            continue;
        }
```

```
        // 否则，两个句子不相似
        return false;
    }

    // 所有对应位置的单词都相似，返回 true
    return true;
}

// 主函数用于测试
int main() {
    // 测试用例 1
    vector<string> sentence1 = {"great", "acting", "skills"};
    vector<string> sentence2 = {"fine", "drama", "talent"};
    vector<vector<string>> similarPairs1 = {{"great", "good"}, {"fine", "good"}, {"drama",
"acting"}, {"skills", "talent"}};
    cout << "测试用例 1 结果: " << (areSentencesSimilarTwo(sentence1, sentence2, similarPairs1) ?
"true" : "false") << endl;

    // 测试用例 2
    vector<string> sentence3 = {"great"};
    vector<string> sentence4 = {"great"};
    vector<vector<string>> similarPairs2 = {};
    cout << "测试用例 2 结果: " << (areSentencesSimilarTwo(sentence3, sentence4, similarPairs2) ?
"true" : "false") << endl;

    // 测试用例 3
    vector<string> sentence5 = {"great"};
    vector<string> sentence6 = {"doubleplus", "good"};
    vector<vector<string>> similarPairs3 = {{"great", "doubleplus"}};
    cout << "测试用例 3 结果: " << (areSentencesSimilarTwo(sentence5, sentence6, similarPairs3) ?
"true" : "false") << endl;

    return 0;
}
```

==========

文件: Code09_SentenceSimilarityII.java

==========

```
package class056;

import java.util.*;
```

```
/**
 * 句子相似性 II
 * 给定两个句子 sentence1 和 sentence2 分别表示为一个字符串数组，并给定一个字符串对 similarPairs，
 * 其中 similarPairs[i] = [xi, yi] 表示两个单词 xi 和 yi 是相似的。
 * 如果 sentence1 和 sentence2 相似则返回 true，如果不相似则返回 false。
 *
 * 两个句子是相似的，如果：
 * 1. 它们具有相同的长度（即相同的字数）
 * 2. sentence1[i] 和 sentence2[i] 是相似的
 *
 * 注意，一个词总是与它自己相似。也注意相似关系是可传递的。
 * 例如，如果单词 a 和 b 是相似的，单词 b 和 c 也是相似的，那么 a 和 c 也是相似的。
 *
 * 示例 1:
 * 输入:
 * sentence1 = ["great","acting","skills"]
 * sentence2 = ["fine","drama","talent"]
 * similarPairs = [["great","good"],["fine","good"],["drama","acting"],["skills","talent"]]
 * 输出: true
 * 解释: 这两个句子长度相同，每个单词都相似。
 *
 * 示例 2:
 * 输入:
 * sentence1 = ["I","love","leetcode"]
 * sentence2 = ["I","love","onepiece"]
 * similarPairs = [["manga","onepiece"],["platform","anime"],["leetcode","platform"],["anime","manga"]]
 * 输出: true
 * 解释: "leetcode" --> "platform" --> "anime" --> "manga" --> "onepiece"
 * 因为“leetcode”和“onepiece”相似，而且前两个单词相同，所以这两个句子是相似的。
 *
 * 示例 3:
 * 输入:
 * sentence1 = ["I","love","leetcode"]
 * sentence2 = ["I","love","onepiece"]
 * similarPairs = [["manga","hunterXhunter"],["platform","anime"],["leetcode","platform"],["anime","manga"]]
 * 输出: false
 * 解释: “leetcode”和“onepiece”不相似。
 *
 * 约束条件：
 * 1 <= sentence1.length, sentence2.length <= 1000
```

```
 * 1 <= sentence1[i].length, sentence2[i].length <= 20
 * sentence1[i] 和 sentence2[i] 只包含英文字母
 * 0 <= similarPairs.length <= 2000
 * similarPairs[i].length == 2
 * 1 <= xi.length, yi.length <= 20
 * xi 和 yi 只包含英文字母
 *
 * 测试链接: https://leetcode.cn/problems/sentence-similarity-ii/
 * 相关平台: LeetCode 737
 */
public class Code09_SentenceSimilarityII {

    /**
     * 使用并查集解决句子相似性问题
     *
     * 解题思路:
     * 1. 首先检查两个句子长度是否相等，不相等直接返回 false
     * 2. 使用并查集来处理相似单词的传递性关系
     * 3. 将所有相似单词对加入并查集
     * 4. 对于句子中的每一对单词，检查它们是否在同一个集合中
     *
     * 时间复杂度：O(N + P * α(P))，其中 N 是句子长度，P 是相似单词对数量，α 是阿克曼函数的反函数
     * 空间复杂度：O(P)
     *
     * @param sentence1 第一个句子
     * @param sentence2 第二个句子
     * @param similarPairs 相似单词对
     * @return 如果两个句子相似返回 true，否则返回 false
     */
    public static boolean areSentencesSimilarTwo(String[] sentence1, String[] sentence2,
List<List<String>> similarPairs) {
        // 边界条件检查
        if (sentence1 == null || sentence2 == null) {
            return sentence1 == sentence2;
        }

        // 长度不同直接返回 false
        if (sentence1.length != sentence2.length) {
            return false;
        }

        // 创建并查集
        UnionFind uf = new UnionFind();
```

```
        // 将所有相似单词对加入并查集
        for (List<String> pair : similarPairs) {
            uf.union(pair.get(0), pair.get(1));
        }

        // 检查每一对单词是否相似
        for (int i = 0; i < sentence1.length; i++) {
            // 单词相同或者在同一个集合中则相似
            if (!sentence1[i].equals(sentence2[i]) &&
                !uf.isConnected(sentence1[i], sentence2[i])) {
                return false;
            }
        }

        return true;
    }

    /**
     * 并查集数据结构实现
     * 包含路径压缩优化
     */
    static class UnionFind {
        private Map<String, String> parent;  // parent[word]表示单词 word 的父节点

        /**
         * 初始化并查集
         */
        public UnionFind() {
            parent = new HashMap<>();
        }

        /**
         * 查找单词的根节点（代表元素）
         * 使用路径压缩优化
         * @param x 要查找的单词
         * @return 单词 x 所在集合的根节点
         */
        public String find(String x) {
            // 如果单词不在并查集中，将其加入并设置为自己的父节点
            if (!parent.containsKey(x)) {
                parent.put(x, x);
                return x;
```

```
        }

        // 路径压缩：将路径上的所有节点直接连接到根节点
        if (!parent.get(x).equals(x)) {
            parent.put(x, find(parent.get(x)));
        }
        return parent.get(x);
    }

    /**
     * 合并两个单词所在的集合
     * @param x 第一个单词
     * @param y 第二个单词
     */
    public void union(String x, String y) {
        String rootX = find(x);
        String rootY = find(y);
        // 如果已经在同一个集合中，则无需合并
        if (!rootX.equals(rootY)) {
            parent.put(rootX, rootY);
        }
    }

    /**
     * 判断两个单词是否在同一个集合中
     * @param x 第一个单词
     * @param y 第二个单词
     * @return 如果在同一个集合中返回 true，否则返回 false
     */
    public boolean isConnected(String x, String y) {
        return find(x).equals(find(y));
    }
}

// 测试方法
public static void main(String[] args) {
    // 测试用例 1
    String[] sentence1_1 = {"great", "acting", "skills"};
    String[] sentence2_1 = {"fine", "drama", "talent"};
    List<List<String>> similarPairs1 = new ArrayList<>();
    similarPairs1.add(Arrays.asList("great", "good"));
    similarPairs1.add(Arrays.asList("fine", "good"));
    similarPairs1.add(Arrays.asList("drama", "acting"));
```

```
        similarPairs1.add(Arrays.asList("skills", "talent"));

        System.out.println("测试用例 1 结果: " + areSentencesSimilarTwo(sentence1_1, sentence2_1,
similarPairs1)); // 预期输出: true

        // 测试用例 2
        String[] sentence1_2 = {"I", "love", "leetcode"};
        String[] sentence2_2 = {"I", "love", "onepiece"};
        List<List<String>> similarPairs2 = new ArrayList<>();
        similarPairs2.add(Arrays.asList("manga", "onepiece"));
        similarPairs2.add(Arrays.asList("platform", "anime"));
        similarPairs2.add(Arrays.asList("leetcode", "platform"));
        similarPairs2.add(Arrays.asList("anime", "manga"));

        System.out.println("测试用例 2 结果: " + areSentencesSimilarTwo(sentence1_2, sentence2_2,
similarPairs2)); // 预期输出: true

        // 测试用例 3
        String[] sentence1_3 = {"I", "love", "leetcode"};
        String[] sentence2_3 = {"I", "love", "onepiece"};
        List<List<String>> similarPairs3 = new ArrayList<>();
        similarPairs3.add(Arrays.asList("manga", "hunterXhunter"));
        similarPairs3.add(Arrays.asList("platform", "anime"));
        similarPairs3.add(Arrays.asList("leetcode", "platform"));
        similarPairs3.add(Arrays.asList("anime", "manga"));

        System.out.println("测试用例 3 结果: " + areSentencesSimilarTwo(sentence1_3, sentence2_3,
similarPairs3)); // 预期输出: false

        // 测试用例 4：相同句子
        String[] sentence1_4 = {"hello", "world"};
        String[] sentence2_4 = {"hello", "world"};
        List<List<String>> similarPairs4 = new ArrayList<>();

        System.out.println("测试用例 4 结果: " + areSentencesSimilarTwo(sentence1_4, sentence2_4,
similarPairs4)); // 预期输出: true
    }
}
```

==========

文件: Code09_SentenceSimilarityII.py

==========

"""
句子相似性 II
给定两个句子 sentence1 和 sentence2 分别表示为一个字符串数组，并给定一个字符串对 similarPairs，
其中 similarPairs[i] = [xi, yi] 表示两个单词 xi 和 yi 是相似的。
如果 sentence1 和 sentence2 相似则返回 true，如果不相似则返回 false。

两个句子是相似的，如果：
1. 它们具有相同的长度（即相同的字数）
2. sentence1[i] 和 sentence2[i] 是相似的

注意，一个词总是与它自己相似。也注意相似关系是可传递的。
例如，如果单词 a 和 b 是相似的，单词 b 和 c 也是相似的，那么 a 和 c 也是相似的。

示例 1:
输入:
sentence1 = ["great","acting","skills"]
sentence2 = ["fine","drama","talent"]
similarPairs = [["great","good"],["fine","good"],["drama","acting"],["skills","talent"]]
输出: true
解释: 这两个句子长度相同，每个单词都相似。

示例 2:
输入:
sentence1 = ["I","love","leetcode"]
sentence2 = ["I","love","onepiece"]
similarPairs =
[["manga","onepiece"],["platform","anime"],["leetcode","platform"],["anime","manga"]]
输出: true
解释: "leetcode" --> "platform" --> "anime" --> "manga" --> "onepiece"
因为"leetcode"和"onepiece"相似，而且前两个单词相同，所以这两个句子是相似的。

示例 3:
输入:
sentence1 = ["I","love","leetcode"]
sentence2 = ["I","love","onepiece"]
similarPairs =
[["manga","hunterXhunter"],["platform","anime"],["leetcode","platform"],["anime","manga"]]
输出: false
解释: "leetcode"和"onepiece"不相似。

约束条件：
1 <= sentence1.length, sentence2.length <= 1000
1 <= sentence1[i].length, sentence2[i].length <= 20

sentence1[i] 和 sentence2[i] 只包含英文字母
0 <= similarPairs.length <= 2000
similarPairs[i].length == 2
1 <= xi.length, yi.length <= 20
xi 和 yi 只包含英文字母

测试链接: https://leetcode.cn/problems/sentence-similarity-ii/
相关平台: LeetCode 737
"""

```
class UnionFind:
    """
    并查集数据结构实现
    包含路径压缩优化
    """

    def __init__(self):
        """
        初始化并查集
        """
        self.parent = {}  # parent[word]表示单词 word 的父节点

    def find(self, x):
        """
        查找单词的根节点（代表元素）
        使用路径压缩优化
        :param x: 要查找的单词
        :return: 单词 x 所在集合的根节点
        """
        # 如果单词不在并查集中，将其加入并设置为自己的父节点
        if x not in self.parent:
            self.parent[x] = x
            return x

        # 路径压缩：将路径上的所有节点直接连接到根节点
        if self.parent[x] != x:
            self.parent[x] = self.find(self.parent[x])
        return self.parent[x]

    def union(self, x, y):
        """
        合并两个单词所在的集合
```

```
        :param x: 第一个单词
        :param y: 第二个单词
        """
        root_x = self.find(x)
        root_y = self.find(y)
        # 如果已经在同一个集合中，则无需合并
        if root_x != root_y:
            self.parent[root_x] = root_y

    def is_connected(self, x, y):
        """
        判断两个单词是否在同一个集合中
        :param x: 第一个单词
        :param y: 第二个单词
        :return: 如果在同一个集合中返回 True，否则返回 False
        """
        return self.find(x) == self.find(y)


def are_sentences_similar_two(sentence1, sentence2, similar_pairs):
    """
    使用并查集解决句子相似性问题

    解题思路：
    1. 首先检查两个句子长度是否相等，不相等直接返回 False
    2. 使用并查集来处理相似单词的传递性关系
    3. 将所有相似单词对加入并查集
    4. 对于句子中的每一对单词，检查它们是否在同一个集合中

    时间复杂度：O(N + P * α(P))，其中 N 是句子长度，P 是相似单词对数量，α 是阿克曼函数的反函数
    空间复杂度：O(P)
    是否为最优解：是

    工程化考量：
    1. 异常处理：检查输入是否为空
    2. 可配置性：可以扩展支持其他相似性规则
    3. 线程安全：当前实现不是线程安全的

    与机器学习等领域的联系：
    1. 自然语言处理：句子相似性判断是 NLP 中的基础任务
    2. 语义分析：通过传递性关系构建语义网络

    语言特性差异:
```

Java: 对象引用和垃圾回收
C++: 指针操作和手动内存管理
Python: 动态类型和自动内存管理

极端输入场景:
1. 空句子
2. 单词相同的句子
3. 没有相似单词对的句子
4. 大量相似单词对的句子

性能优化:
1. 路径压缩优化 find 操作
2. 使用哈希表快速查找单词
3. 提前终止优化

:param sentence1: 第一个句子
:param sentence2: 第二个句子
:param similar_pairs: 相似单词对
:return: 如果两个句子相似返回 True，否则返回 False
"""
```
# 边界条件检查
if not sentence1 and not sentence2:
    return True

if not sentence1 or not sentence2:
    return False

# 长度不同直接返回 False
if len(sentence1) != len(sentence2):
    return False

# 创建并查集
union_find = UnionFind()

# 将所有相似单词对加入并查集
for pair in similar_pairs:
    union_find.union(pair[0], pair[1])

# 检查每一对单词是否相似
for i in range(len(sentence1)):
    # 单词相同或者在同一个集合中则相似
    if sentence1[i] != sentence2[i] and \
            not union_find.is_connected(sentence1[i], sentence2[i]):
```

```
            return False

    return True


# 测试方法
if __name__ == "__main__":
    # 测试用例 1
    sentence1_1 = ["great", "acting", "skills"]
    sentence2_1 = ["fine", "drama", "talent"]
    similar_pairs1 = [["great", "good"], ["fine", "good"], ["drama", "acting"], ["skills",
"talent"]]

    print("测试用例 1 结果:", are_sentences_similar_two(sentence1_1, sentence2_1, similar_pairs1))
# 预期输出: True

    # 测试用例 2
    sentence1_2 = ["I", "love", "leetcode"]
    sentence2_2 = ["I", "love", "onepiece"]
    similar_pairs2 = [["manga", "onepiece"], ["platform", "anime"], ["leetcode", "platform"],
["anime", "manga"]]

    print("测试用例 2 结果:", are_sentences_similar_two(sentence1_2, sentence2_2, similar_pairs2))
# 预期输出: True

    # 测试用例 3
    sentence1_3 = ["I", "love", "leetcode"]
    sentence2_3 = ["I", "love", "onepiece"]
    similar_pairs3 = [["manga", "hunterXhunter"], ["platform", "anime"], ["leetcode",
"platform"], ["anime", "manga"]]

    print("测试用例 3 结果:", are_sentences_similar_two(sentence1_3, sentence2_3, similar_pairs3))
# 预期输出: False

    # 测试用例 4：相同句子
    sentence1_4 = ["hello", "world"]
    sentence2_4 = ["hello", "world"]
    similar_pairs4 = []

    print("测试用例 4 结果:", are_sentences_similar_two(sentence1_4, sentence2_4, similar_pairs4))
# 预期输出: True

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

文件: Code10_LargestComponentSizeByCommonFactor.cpp

```
==========================================
#include <iostream>
#include <vector>
#include <algorithm>
#include <unordered_map>

using namespace std;

/**
 * 按公因数计算最大组件大小
 * 给定一个由不同正整数组成的非空数组 nums，考虑下面的图：
 * 有 nums.length 个节点，按从 nums[0] 到 nums[nums.length - 1] 标记；
 * 只有当 nums[i] 和 nums[j] 共用大于 1 的公因数时，nums[i] 和 nums[j] 之间才有一条边。
 * 返回图中最大连通组件的大小。
 *
 * 示例 1:
 * 输入: nums = [4,6,15,35]
 * 输出: 4
 *
 * 示例 2:
 * 输入: nums = [20,50,9,63]
 * 输出: 2
 *
 * 示例 3:
 * 输入: nums = [2,3,6,7,4,12,21,39]
 * 输出: 8
 *
 * 约束条件：
 * 1 <= nums.length <= 2 * 10^4
 * 1 <= nums[i] <= 10^5
 * nums 中所有值都不同
 *
 * 测试链接: https://leetcode.cn/problems/largest-component-size-by-common-factor/
 * 相关平台: LeetCode 952
 */

class UnionFind {
private:
    vector<int> parent;
    vector<int> rank;
```

```
public:
    /**
     * 初始化并查集
     * @param n 节点数量
     */
    UnionFind(int n) {
        parent.resize(n);
        rank.resize(n);
        // 初始时每个节点都是自己的父节点
        for (int i = 0; i < n; i++) {
            parent[i] = i;
            rank[i] = 1;  // 初始时每个树的秩为1
        }
    }

    /**
     * 查找节点的根节点（代表元素）
     * 使用路径压缩优化
     * @param x 要查找的节点
     * @return 节点x所在集合的根节点
     */
    int find(int x) {
        if (parent[x] != x) {
            // 路径压缩：将路径上的所有节点直接连接到根节点
            parent[x] = find(parent[x]);
        }
        return parent[x];
    }

    /**
     * 合并两个集合
     * 使用按秩合并优化
     * @param x 第一个节点
     * @param y 第二个节点
     */
    void unionSets(int x, int y) {
        int rootX = find(x);
        int rootY = find(y);

        // 如果已经在同一个集合中，则无需合并
        if (rootX != rootY) {
            // 按秩合并：将秩小的树合并到秩大的树下
            if (rank[rootX] > rank[rootY]) {
```

```
                parent[rootY] = rootX;
            } else if (rank[rootX] < rank[rootY]) {
                parent[rootX] = rootY;
            } else {
                // 秩相等时，任选一个作为根，并将其秩加 1
                parent[rootY] = rootX;
                rank[rootX]++;
            }
        }
    }
};

/**
 * 使用并查集解决按公因数计算最大组件大小问题
 *
 * 解题思路：
 * 1. 对于每个数字，将其与其所有质因数连接（使用并查集）
 * 2. 这样，具有公共因数的数字就会在同一个连通组件中
 * 3. 统计每个连通组件的大小，返回最大的
 *
 * 时间复杂度：O(N * √M * α(M))，其中 N 是数组长度，M 是数组中的最大值，α 是阿克曼函数的反函数
 * 空间复杂度：O(M + N)
 *
 * @param nums 输入数组
 * @return 最大连通组件的大小
 */
int largestComponentSize(vector<int>& nums) {
    // 边界条件检查
    if (nums.empty()) {
        return 0;
    }

    // 找到数组中的最大值
    int maxNum = *max_element(nums.begin(), nums.end());

    // 创建并查集，大小为 maxNum+1
    UnionFind uf(maxNum + 1);

    // 对于每个数字，将其与其所有质因数连接
    for (int num : nums) {
        // 分解质因数
        for (int i = 2; i * i <= num; i++) {
            if (num % i == 0) {
```

```
                uf.unionSets(num, i);
                uf.unionSets(num, num / i);
            }
        }
    }

    // 统计每个连通组件的大小
    unordered_map<int, int> componentSize;
    int maxSize = 0;

    for (int num : nums) {
        int root = uf.find(num);
        componentSize[root]++;
        maxSize = max(maxSize, componentSize[root]);
    }

    return maxSize;
}

// 测试方法
int main() {
    // 测试用例 1
    vector<int> nums1 = {4, 6, 15, 35};
    cout << "测试用例 1 结果: " << largestComponentSize(nums1) << endl; // 预期输出: 4

    // 测试用例 2
    vector<int> nums2 = {20, 50, 9, 63};
    cout << "测试用例 2 结果: " << largestComponentSize(nums2) << endl; // 预期输出: 2

    // 测试用例 3
    vector<int> nums3 = {2, 3, 6, 7, 4, 12, 21, 39};
    cout << "测试用例 3 结果: " << largestComponentSize(nums3) << endl; // 预期输出: 8

    // 测试用例 4：单个元素
    vector<int> nums4 = {10};
    cout << "测试用例 4 结果: " << largestComponentSize(nums4) << endl; // 预期输出: 1

    // 测试用例 5：互质数字
    vector<int> nums5 = {2, 3, 5, 7, 11};
    cout << "测试用例 5 结果: " << largestComponentSize(nums5) << endl; // 预期输出: 1

    return 0;
}
```

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

文件: Code10_LargestComponentSizeByCommonFactor.java
==========================================
package class056;

import java.util.*;

/**
 * 按公因数计算最大组件大小
 * 给定一个由不同正整数组成的非空数组 nums，考虑下面的图：
 * 有 nums.length 个节点，按从 nums[0] 到 nums[nums.length - 1] 标记；
 * 只有当 nums[i] 和 nums[j] 共用大于 1 的公因数时，nums[i] 和 nums[j] 之间才有一条边。
 * 返回图中最大连通组件的大小。
 *
 * 示例 1:
 * 输入: nums = [4,6,15,35]
 * 输出: 4
 *
 * 示例 2:
 * 输入: nums = [20,50,9,63]
 * 输出: 2
 *
 * 示例 3:
 * 输入: nums = [2,3,6,7,4,12,21,39]
 * 输出: 8
 *
 * 约束条件：
 * 1 <= nums.length <= 2 * 10^4
 * 1 <= nums[i] <= 10^5
 * nums 中所有值都不同
 *
 * 测试链接: https://leetcode.cn/problems/largest-component-size-by-common-factor/
 * 相关平台: LeetCode 952
 */
public class Code10_LargestComponentSizeByCommonFactor {

    /**
     * 使用并查集解决按公因数计算最大组件大小问题
     *
     * 解题思路：
     * 1. 对于每个数字，将其与其所有质因数连接（使用并查集）
```

```
     * 2. 这样，具有公共因数的数字就会在同一个连通组件中
     * 3. 统计每个连通组件的大小，返回最大的
     *
     * 时间复杂度：O(N * √M * α(M))，其中 N 是数组长度，M 是数组中的最大值，α 是阿克曼函数的反函数
     * 空间复杂度：O(M + N)
     *
     * @param nums 输入数组
     * @return 最大连通组件的大小
     */
    public static int largestComponentSize(int[] nums) {
        // 边界条件检查
        if (nums == null || nums.length == 0) {
            return 0;
        }

        // 找到数组中的最大值
        int maxNum = 0;
        for (int num : nums) {
            maxNum = Math.max(maxNum, num);
        }

        // 创建并查集，大小为 maxNum+1
        UnionFind uf = new UnionFind(maxNum + 1);

        // 对于每个数字，将其与其所有质因数连接
        for (int num : nums) {
            // 分解质因数
            for (int i = 2; i * i <= num; i++) {
                if (num % i == 0) {
                    uf.union(num, i);
                    uf.union(num, num / i);
                }
            }
        }

        // 统计每个连通组件的大小
        Map<Integer, Integer> componentSize = new HashMap<>();
        int maxSize = 0;

        for (int num : nums) {
            int root = uf.find(num);
            componentSize.put(root, componentSize.getOrDefault(root, 0) + 1);
```

```
            maxSize = Math.max(maxSize, componentSize.get(root));
        }

        return maxSize;
    }

    /**
     * 并查集数据结构实现
     * 包含路径压缩和按秩合并优化
     */
    static class UnionFind {
        private int[] parent;  // parent[i]表示节点 i 的父节点
        private int[] rank;    // rank[i]表示以 i 为根的树的高度上界

        /**
         * 初始化并查集
         * @param n 节点数量
         */
        public UnionFind(int n) {
            parent = new int[n];
            rank = new int[n];
            // 初始时每个节点都是自己的父节点
            for (int i = 0; i < n; i++) {
                parent[i] = i;
                rank[i] = 1;  // 初始时每个树的秩为 1
            }
        }

        /**
         * 查找节点的根节点（代表元素）
         * 使用路径压缩优化
         * @param x 要查找的节点
         * @return 节点 x 所在集合的根节点
         */
        public int find(int x) {
            if (parent[x] != x) {
                // 路径压缩：将路径上的所有节点直接连接到根节点
                parent[x] = find(parent[x]);
            }
            return parent[x];
        }

        /**
```

```
     * 合并两个集合
     * 使用按秩合并优化
     * @param x 第一个节点
     * @param y 第二个节点
     */
    public void union(int x, int y) {
        int rootX = find(x);
        int rootY = find(y);

        // 如果已经在同一个集合中，则无需合并
        if (rootX != rootY) {
            // 按秩合并：将秩小的树合并到秩大的树下
            if (rank[rootX] > rank[rootY]) {
                parent[rootY] = rootX;
            } else if (rank[rootX] < rank[rootY]) {
                parent[rootX] = rootY;
            } else {
                // 秩相等时，任选一个作为根，并将其秩加 1
                parent[rootY] = rootX;
                rank[rootX]++;
            }
        }
    }
}

// 测试方法
public static void main(String[] args) {
    // 测试用例 1
    int[] nums1 = {4, 6, 15, 35};
    System.out.println("测试用例 1 结果: " + largestComponentSize(nums1)); // 预期输出: 4

    // 测试用例 2
    int[] nums2 = {20, 50, 9, 63};
    System.out.println("测试用例 2 结果: " + largestComponentSize(nums2)); // 预期输出: 2

    // 测试用例 3
    int[] nums3 = {2, 3, 6, 7, 4, 12, 21, 39};
    System.out.println("测试用例 3 结果: " + largestComponentSize(nums3)); // 预期输出: 8

    // 测试用例 4：单个元素
    int[] nums4 = {10};
    System.out.println("测试用例 4 结果: " + largestComponentSize(nums4)); // 预期输出: 1
```

```
        // 测试用例 5：互质数字
        int[] nums5 = {2, 3, 5, 7, 11};
        System.out.println("测试用例 5 结果: " + largestComponentSize(nums5)); // 预期输出: 1
    }
}
```

==========

文件: Code10_LargestComponentSizeByCommonFactor.py

==========

```
"""
按公因数计算最大组件大小
给定一个由不同正整数组成的非空数组 nums，考虑下面的图：
有 nums.length 个节点，按从 nums[0] 到 nums[nums.length - 1] 标记；
只有当 nums[i] 和 nums[j] 共用大于 1 的公因数时，nums[i] 和 nums[j] 之间才有一条边。
返回图中最大连通组件的大小。

示例 1:
输入: nums = [4,6,15,35]
输出: 4

示例 2:
输入: nums = [20,50,9,63]
输出: 2

示例 3:
输入: nums = [2,3,6,7,4,12,21,39]
输出: 8

约束条件：
1 <= nums.length <= 2 * 10^4
1 <= nums[i] <= 10^5
nums 中所有值都不同

测试链接: https://leetcode.cn/problems/largest-component-size-by-common-factor/
相关平台: LeetCode 952
"""



class UnionFind:
    """
    并查集数据结构实现
    包含路径压缩和按秩合并优化
```

```
    """

    def __init__(self, n):
        """
        初始化并查集
        :param n: 节点数量
        """
        # parent[i]表示节点 i 的父节点
        self.parent = list(range(n))
        # rank[i]表示以 i 为根的树的高度上界
        self.rank = [1] * n

    def find(self, x):
        """
        查找节点的根节点（代表元素）
        使用路径压缩优化
        :param x: 要查找的节点
        :return: 节点 x 所在集合的根节点
        """
        if self.parent[x] != x:
            # 路径压缩：将路径上的所有节点直接连接到根节点
            self.parent[x] = self.find(self.parent[x])
        return self.parent[x]

    def union(self, x, y):
        """
        合并两个集合
        使用按秩合并优化
        :param x: 第一个节点
        :param y: 第二个节点
        """
        root_x = self.find(x)
        root_y = self.find(y)

        # 如果已经在同一个集合中，则无需合并
        if root_x != root_y:
            # 按秩合并：将秩小的树合并到秩大的树下
            if self.rank[root_x] > self.rank[root_y]:
                self.parent[root_y] = root_x
            elif self.rank[root_x] < self.rank[root_y]:
                self.parent[root_x] = root_y
            else:
                # 秩相等时，任选一个作为根，并将其秩加 1
```

```
            self.parent[root_y] = root_x
            self.rank[root_x] += 1


def largest_component_size(nums):
    """
    使用并查集解决按公因数计算最大组件大小问题

    解题思路：
    1. 对于每个数字，将其与其所有质因数连接（使用并查集）
    2. 这样，具有公共因数的数字就会在同一个连通组件中
    3. 统计每个连通组件的大小，返回最大的

    时间复杂度：O(N * √M * α(M))，其中 N 是数组长度，M 是数组中的最大值，α 是阿克曼函数的反函数
    空间复杂度：O(M + N)
    是否为最优解：是

    工程化考量：
    1. 异常处理：检查输入是否为空
    2. 可配置性：可以扩展支持其他因数规则
    3. 线程安全：当前实现不是线程安全的

    与机器学习等领域的联系：
    1. 图论：连通组件分析是图神经网络中的基础操作
    2. 社交网络分析：识别具有共同兴趣的用户群体

    语言特性差异:
    Java: 对象引用和垃圾回收，数组初始化
    C++: 指针操作和手动内存管理，vector 容器
    Python: 动态类型和自动内存管理，list 初始化

    极端输入场景:
    1. 空数组
    2. 单个元素数组
    3. 所有元素互质的数组
    4. 大量元素的数组

    性能优化:
    1. 路径压缩优化 find 操作
    2. 按秩合并优化 union 操作
    3. 质因数分解优化（只需分解到 √n）

    :param nums: 输入数组
```

```
    :return: 最大连通组件的大小
    """
    # 边界条件检查
    if not nums:
        return 0

    # 找到数组中的最大值
    max_num = max(nums)

    # 创建并查集，大小为 max_num+1
    union_find = UnionFind(max_num + 1)

    # 对于每个数字，将其与其所有质因数连接
    for num in nums:
        # 分解质因数
        i = 2
        while i * i <= num:
            if num % i == 0:
                union_find.union(num, i)
                union_find.union(num, num // i)
            i += 1

    # 统计每个连通组件的大小
    component_size = {}
    max_size = 0

    for num in nums:
        root = union_find.find(num)
        component_size[root] = component_size.get(root, 0) + 1
        max_size = max(max_size, component_size[root])

    return max_size


# 测试方法
if __name__ == "__main__":
    # 测试用例 1
    nums1 = [4, 6, 15, 35]
    print("测试用例 1 结果:", largest_component_size(nums1))  # 预期输出: 4

    # 测试用例 2
    nums2 = [20, 50, 9, 63]
    print("测试用例 2 结果:", largest_component_size(nums2))  # 预期输出: 2
```

```
    # 测试用例 3
    nums3 = [2, 3, 6, 7, 4, 12, 21, 39]
    print("测试用例 3 结果:", largest_component_size(nums3))  # 预期输出: 8

    # 测试用例 4：单个元素
    nums4 = [10]
    print("测试用例 4 结果:", largest_component_size(nums4))  # 预期输出: 1

    # 测试用例 5：互质数字
    nums5 = [2, 3, 5, 7, 11]
    print("测试用例 5 结果:", largest_component_size(nums5))  # 预期输出: 1
```

==========

文件: Code11_SatisfiabilityOfEqualityEquations.java
==========

```
package class056;

import java.util.*;

/**
 * 等式方程的可满足性
 * 给定一个由表示变量之间关系的字符串方程组成的数组，每个字符串方程 equations[i] 的长度为 4，
 * 并采用两种不同的形式之一："a==b" 或 "a!=b"。在这里，a 和 b 是小写字母（不一定不同），
 * 表示单字母变量名。只有当可以将整数分配给变量名，以便满足所有给定的方程时才返回 true，否则返回 false。
 *
 * 示例 1:
 * 输入: ["a==b","b!=a"]
 * 输出: false
 * 解释: 如果我们指定，a = 1 且 b = 1，那么可以满足第一个方程，但无法满足第二个方程。没有办法分配变量同时满足这两个方程。
 *
 * 示例 2:
 * 输入: ["b==a","a==b"]
 * 输出: true
 * 解释: 我们可以指定 a = 1 且 b = 1 以满足满足这两个方程。
 *
 * 示例 3:
 * 输入: ["a==b","b==c","a==c"]
 * 输出: true
 *
```

```
 * 示例 4:
 * 输入: ["a==b","b!=c","c==a"]
 * 输出: false
 *
 * 示例 5:
 * 输入: ["c==c","b==d","x!=z"]
 * 输出: true
 *
 * 约束条件：
 * 1 <= equations.length <= 500
 * equations[i].length == 4
 * equations[i][0] 和 equations[i][3] 是小写字母
 * equations[i][1] 是 '=' 或 '!'
 * equations[i][2] 是 '='
 *
 * 测试链接: https://leetcode.cn/problems/satisfiability-of-equality-equations/
 * 相关平台: LeetCode 990
 */
public class Code11_SatisfiabilityOfEqualityEquations {

    /**
     * 使用并查集解决等式方程的可满足性问题
     *
     * 解题思路：
     * 1. 首先处理所有等式方程，将相等的变量连接到同一个集合中
     * 2. 然后检查所有不等式方程，如果两个变量在同一个集合中，则返回 false
     * 3. 如果所有不等式方程都满足，则返回 true
     *
     * 时间复杂度：O(N * α(26))，其中 N 是方程数量，α 是阿克曼函数的反函数
     * 空间复杂度：O(1)，因为只有 26 个小写字母
     *
     * @param equations 方程数组
     * @return 如果可以满足所有方程返回 true，否则返回 false
     */
    public static boolean equationsPossible(String[] equations) {
        // 边界条件检查
        if (equations == null || equations.length == 0) {
            return true;
        }

        // 创建并查集，大小为 26（对应 26 个小写字母）
        UnionFind uf = new UnionFind(26);
```

```
    // 首先处理所有等式方程
    for (String equation : equations) {
        if (equation.charAt(1) == '=') {
            int x = equation.charAt(0) - 'a';
            int y = equation.charAt(3) - 'a';
            uf.union(x, y);
        }
    }

    // 然后检查所有不等式方程
    for (String equation : equations) {
        if (equation.charAt(1) == '!') {
            int x = equation.charAt(0) - 'a';
            int y = equation.charAt(3) - 'a';
            // 如果两个变量在同一个集合中，则不等式不成立
            if (uf.find(x) == uf.find(y)) {
                return false;
            }
        }
    }

    return true;
}

/**
 * 并查集数据结构实现
 * 包含路径压缩优化
 */
static class UnionFind {
    private int[] parent;  // parent[i]表示节点 i 的父节点

    /**
     * 初始化并查集
     * @param n 节点数量
     */
    public UnionFind(int n) {
        parent = new int[n];
        // 初始时每个节点都是自己的父节点
        for (int i = 0; i < n; i++) {
            parent[i] = i;
        }
    }
```

```
        /**
         * 查找节点的根节点（代表元素）
         * 使用路径压缩优化
         * @param x 要查找的节点
         * @return 节点 x 所在集合的根节点
         */
        public int find(int x) {
            if (parent[x] != x) {
                // 路径压缩：将路径上的所有节点直接连接到根节点
                parent[x] = find(parent[x]);
            }
            return parent[x];
        }

        /**
         * 合并两个集合
         * @param x 第一个节点
         * @param y 第二个节点
         */
        public void union(int x, int y) {
            int rootX = find(x);
            int rootY = find(y);
            // 如果已经在同一个集合中，则无需合并
            if (rootX != rootY) {
                parent[rootX] = rootY;
            }
        }
    }

    // 测试方法
    public static void main(String[] args) {
        // 测试用例 1
        String[] equations1 = {"a==b", "b!=a"};
        System.out.println("测试用例 1 结果: " + equationsPossible(equations1)); // 预期输出:
false

        // 测试用例 2
        String[] equations2 = {"b==a", "a==b"};
        System.out.println("测试用例 2 结果: " + equationsPossible(equations2)); // 预期输出: true

        // 测试用例 3
        String[] equations3 = {"a==b", "b==c", "a==c"};
        System.out.println("测试用例 3 结果: " + equationsPossible(equations3)); // 预期输出: true
```

```
        // 测试用例 4
        String[] equations4 = {"a==b", "b!=c", "c==a"};
        System.out.println("测试用例 4 结果: " + equationsPossible(equations4)); // 预期输出:
false

        // 测试用例 5
        String[] equations5 = {"c==c", "b==d", "x!=z"};
        System.out.println("测试用例 5 结果: " + equationsPossible(equations5)); // 预期输出: true

        // 测试用例 6：空数组
        String[] equations6 = {};
        System.out.println("测试用例 6 结果: " + equationsPossible(equations6)); // 预期输出: true
    }
}
```

==========

文件: Code11_SatisfiabilityOfEqualityEquations.py
==========
"""
等式方程的可满足性
给定一个由表示变量之间关系的字符串方程组成的数组，每个字符串方程 equations[i] 的长度为 4，
并采用两种不同的形式之一："a==b" 或 "a!=b"。在这里，a 和 b 是小写字母（不一定不同），
表示单字母变量名。只有当可以将整数分配给变量名，以便满足所有给定的方程时才返回 true，否则返回
false。

示例 1:
输入: ["a==b","b!=a"]
输出: false
解释: 如果我们指定，a = 1 且 b = 1，那么可以满足第一个方程，但无法满足第二个方程。没有办法分配变
量同时满足这两个方程。

示例 2:
输入: ["b==a","a==b"]
输出: true
解释: 我们可以指定 a = 1 且 b = 1 以满足满足这两个方程。

示例 3:
输入: ["a==b","b==c","a==c"]
输出: true

示例 4:

输入: ["a==b","b!=c","c==a"]
输出: false

示例 5:
输入: ["c==c","b==d","x!=z"]
输出: true

约束条件：
1 <= equations.length <= 500
equations[i].length == 4
equations[i][0] 和 equations[i][3] 是小写字母
equations[i][1] 是 '=' 或 '!'
equations[i][2] 是 '='

测试链接: https://leetcode.cn/problems/satisfiability-of-equality-equations/
相关平台: LeetCode 990
"""

```
class UnionFind:
    """
    并查集数据结构实现
    包含路径压缩优化
    """

    def __init__(self, n):
        """
        初始化并查集
        :param n: 节点数量
        """
        # parent[i]表示节点 i 的父节点
        self.parent = list(range(n))

    def find(self, x):
        """
        查找节点的根节点（代表元素）
        使用路径压缩优化
        :param x: 要查找的节点
        :return: 节点 x 所在集合的根节点
        """
        if self.parent[x] != x:
            # 路径压缩：将路径上的所有节点直接连接到根节点
            self.parent[x] = self.find(self.parent[x])
```

```
        return self.parent[x]

    def union(self, x, y):
        """
        合并两个集合
        :param x: 第一个节点
        :param y: 第二个节点
        """
        root_x = self.find(x)
        root_y = self.find(y)
        # 如果已经在同一个集合中，则无需合并
        if root_x != root_y:
            self.parent[root_x] = root_y


def equations_possible(equations):
    """
    使用并查集解决等式方程的可满足性问题

    解题思路：
    1. 首先处理所有等式方程，将相等的变量连接到同一个集合中
    2. 然后检查所有不等式方程，如果两个变量在同一个集合中，则返回 false
    3. 如果所有不等式方程都满足，则返回 true

    时间复杂度：O(N * α(26))，其中 N 是方程数量，α 是阿克曼函数的反函数
    空间复杂度：O(1)，因为只有 26 个小写字母
    是否为最优解：是

    工程化考量：
    1. 异常处理：检查输入是否为空
    2. 可配置性：可以扩展支持更多变量类型
    3. 线程安全：当前实现不是线程安全的

    与机器学习等领域的联系：
    1. 逻辑推理：等式推理是符号 AI 中的基础问题
    2. 约束满足问题：在约束编程中有广泛应用

    语言特性差异:
    Java: 对象引用和垃圾回收，数组初始化
    C++: 指针操作和手动内存管理，vector 容器
    Python: 动态类型和自动内存管理，list 初始化

    极端输入场景:
```

1. 空数组
2. 只有等式方程
3. 只有不等式方程
4. 自相矛盾的方程（如 a==b 和 a!=b 同时存在）

性能优化:
1. 路径压缩优化 find 操作
2. 分两阶段处理（先处理等式，再检查不等式）
3. 提前终止优化

:param equations: 方程数组
:return: 如果可以满足所有方程返回 True，否则返回 False
"""
# 边界条件检查
if not equations:
    return True

# 创建并查集，大小为 26（对应 26 个小写字母）
union_find = UnionFind(26)

# 首先处理所有等式方程
for equation in equations:
    if equation[1] == '=':
        x = ord(equation[0]) - ord('a')
        y = ord(equation[3]) - ord('a')
        union_find.union(x, y)

# 然后检查所有不等式方程
for equation in equations:
    if equation[1] == '!':
        x = ord(equation[0]) - ord('a')
        y = ord(equation[3]) - ord('a')
        # 如果两个变量在同一个集合中，则不等式不成立
        if union_find.find(x) == union_find.find(y):
            return False

return True

# 测试方法
if __name__ == "__main__":
    # 测试用例 1
    equations1 = ["a==b", "b!=a"]

```
    print("测试用例 1 结果:", equations_possible(equations1))  # 预期输出: false

    # 测试用例 2
    equations2 = ["b==a", "a==b"]
    print("测试用例 2 结果:", equations_possible(equations2))  # 预期输出: true

    # 测试用例 3
    equations3 = ["a==b", "b==c", "a==c"]
    print("测试用例 3 结果:", equations_possible(equations3))  # 预期输出: true

    # 测试用例 4
    equations4 = ["a==b", "b!=c", "c==a"]
    print("测试用例 4 结果:", equations_possible(equations4))  # 预期输出: false

    # 测试用例 5
    equations5 = ["c==c", "b==d", "x!=z"]
    print("测试用例 5 结果:", equations_possible(equations5))  # 预期输出: true

    # 测试用例 6：空数组
    equations6 = []
    print("测试用例 6 结果:", equations_possible(equations6))  # 预期输出: true
```

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

文件: Code12_NumberOfOperationsToMakeNetworkConnected.java

```
==========================================
package class056;

import java.util.*;

/**
 * 连通网络的操作次数
 * 用以太网线缆将 n 台计算机连接成一个网络，计算机的编号从 0 到 n-1。
 * 线缆用 connections 表示，其中 connections[i] = [a, b] 连接了计算机 a 和 b。
 * 网络中的任何一台计算机都可以通过网络直接或间接访问同一个网络中其他任意一台计算机。
 * 给你这个计算机网络的初始布线 connections，你可以拔开任意两台直连计算机之间的线缆，
 * 并用它连接一对未直连的计算机。请你计算并返回使所有计算机都连通所需的最少操作次数。
 * 如果不可能，则返回 -1。
 *
 * 示例 1:
 * 输入: n = 4, connections = [[0,1],[0,2],[1,2]]
 * 输出: 1
 * 解释: 拔下计算机 1 和 2 之间的线缆，并将它插到计算机 1 和 3 上。
```

```
 *
 * 示例 2:
 * 输入: n = 6, connections = [[0,1],[0,2],[0,3],[1,2],[1,3]]
 * 输出: 2
 *
 * 示例 3:
 * 输入: n = 6, connections = [[0,1],[0,2],[0,3],[1,2]]
 * 输出: -1
 * 解释: 线缆数量不足。
 *
 * 示例 4:
 * 输入: n = 5, connections = [[0,1],[0,2],[3,4],[2,3]]
 * 输出: 0
 *
 * 约束条件：
 * 1 <= n <= 10^5
 * 1 <= connections.length <= min(n*(n-1)/2, 10^5)
 * connections[i].length == 2
 * 0 <= connections[i][0], connections[i][1] < n
 * connections[i][0] != connections[i][1]
 * 没有重复的连接
 * 两台计算机不会通过多条线缆连接
 *
 * 测试链接: https://leetcode.cn/problems/number-of-operations-to-make-network-connected/
 * 相关平台: LeetCode 1319
 */
public class Code12_NumberOfOperationsToMakeNetworkConnected {

    /**
     * 使用并查集解决连通网络的操作次数问题
     *
     * 解题思路：
     * 1. 首先检查线缆数量是否足够，至少需要 n-1 条线缆才能连接 n 台计算机
     * 2. 使用并查集统计当前有多少个连通分量
     * 3. 要连接 k 个连通分量，需要 k-1 次操作
     *
     * 时间复杂度：O(M * α(N))，其中 M 是 connections 的长度，N 是计算机数量，α 是阿克曼函数的反函数
     * 空间复杂度：O(N)
     *
     * @param n 计算机数量
     * @param connections 线缆连接关系
     * @return 最少操作次数，如果不可能则返回-1
```

```
 */
public static int makeConnected(int n, int[][] connections) {
    // 边界条件检查
    if (n <= 1) {
        return 0;
    }

    // 线缆数量不足，无法连接所有计算机
    if (connections.length < n - 1) {
        return -1;
    }

    // 创建并查集
    UnionFind uf = new UnionFind(n);

    // 处理所有连接
    int redundantCables = 0;  // 冗余线缆数量
    for (int[] connection : connections) {
        int a = connection[0];
        int b = connection[1];
        // 如果两个计算机已经在同一个连通分量中，则这条线缆是冗余的
        if (uf.find(a) == uf.find(b)) {
            redundantCables++;
        } else {
            uf.union(a, b);
        }
    }

    // 统计连通分量数量
    int components = uf.getComponents();

    // 需要 components-1 次操作来连接所有连通分量
    // 只要冗余线缆数量足够就行
    return components - 1;
}

/**
 * 并查集数据结构实现
 * 包含路径压缩和按秩合并优化
 */
static class UnionFind {
    private int[] parent;  // parent[i]表示节点 i 的父节点
    private int[] rank;    // rank[i]表示以 i 为根的树的高度上界
```

```
private int components; // 连通分量数量

/**
 * 初始化并查集
 * @param n 节点数量
 */
public UnionFind(int n) {
    parent = new int[n];
    rank = new int[n];
    components = n;
    // 初始时每个节点都是自己的父节点
    for (int i = 0; i < n; i++) {
        parent[i] = i;
        rank[i] = 1;  // 初始时每个树的秩为1
    }
}

/**
 * 查找节点的根节点（代表元素）
 * 使用路径压缩优化
 * @param x 要查找的节点
 * @return 节点x所在集合的根节点
 */
public int find(int x) {
    if (parent[x] != x) {
        // 路径压缩：将路径上的所有节点直接连接到根节点
        parent[x] = find(parent[x]);
    }
    return parent[x];
}

/**
 * 合并两个集合
 * 使用按秩合并优化
 * @param x 第一个节点
 * @param y 第二个节点
 */
public void union(int x, int y) {
    int rootX = find(x);
    int rootY = find(y);

    // 如果已经在同一个集合中，则无需合并
    if (rootX != rootY) {
```

```
            // 按秩合并：将秩小的树合并到秩大的树下
            if (rank[rootX] > rank[rootY]) {
                parent[rootY] = rootX;
            } else if (rank[rootX] < rank[rootY]) {
                parent[rootX] = rootY;
            } else {
                // 秩相等时，任选一个作为根，并将其秩加 1
                parent[rootY] = rootX;
                rank[rootX]++;
            }
            // 连通分量数量减 1
            components--;
        }
    }

    /**
     * 获取连通分量数量
     * @return 连通分量数量
     */
    public int getComponents() {
        return components;
    }
}

// 测试方法
public static void main(String[] args) {
    // 测试用例 1
    int n1 = 4;
    int[][] connections1 = {{0,1},{0,2},{1,2}};
    System.out.println("测试用例 1 结果: " + makeConnected(n1, connections1)); // 预期输出: 1

    // 测试用例 2
    int n2 = 6;
    int[][] connections2 = {{0,1},{0,2},{0,3},{1,2},{1,3}};
    System.out.println("测试用例 2 结果: " + makeConnected(n2, connections2)); // 预期输出: 2

    // 测试用例 3
    int n3 = 6;
    int[][] connections3 = {{0,1},{0,2},{0,3},{1,2}};
    System.out.println("测试用例 3 结果: " + makeConnected(n3, connections3)); // 预期输出: -1

    // 测试用例 4
    int n4 = 5;
```

```
        int[][] connections4 = {{0,1},{0,2},{3,4},{2,3}};
        System.out.println("测试用例 4 结果: " + makeConnected(n4, connections4)); // 预期输出: 0

        // 测试用例 5：单个计算机
        int n5 = 1;
        int[][] connections5 = {};
        System.out.println("测试用例 5 结果: " + makeConnected(n5, connections5)); // 预期输出: 0
    }
}
```

==========

文件: Code12_NumberOfOperationsToMakeNetworkConnected.py

==========

"""

连通网络的操作次数

用以太网线缆将 n 台计算机连接成一个网络，计算机的编号从 0 到 n-1。

线缆用 connections 表示，其中 connections[i] = [a, b] 连接了计算机 a 和 b。

网络中的任何一台计算机都可以通过网络直接或间接访问同一个网络中其他任意一台计算机。

给你这个计算机网络的初始布线 connections，你可以拔开任意两台直连计算机之间的线缆，

并用它连接一对未直连的计算机。请你计算并返回使所有计算机都连通所需的最少操作次数。

如果不可能，则返回 -1。

示例 1:

输入: n = 4, connections = [[0,1],[0,2],[1,2]]

输出: 1

解释: 拔下计算机 1 和 2 之间的线缆，并将它插到计算机 1 和 3 上。

示例 2:

输入: n = 6, connections = [[0,1],[0,2],[0,3],[1,2],[1,3]]

输出: 2

示例 3:

输入: n = 6, connections = [[0,1],[0,2],[0,3],[1,2]]

输出: -1

解释: 线缆数量不足。

示例 4:

输入: n = 5, connections = [[0,1],[0,2],[3,4],[2,3]]

输出: 0

约束条件：

1 <= n <= 10^5

1 <= connections.length <= min(n*(n-1)/2, 10^5)
connections[i].length == 2
0 <= connections[i][0], connections[i][1] < n
connections[i][0] != connections[i][1]
没有重复的连接
两台计算机不会通过多条线缆连接

测试链接: https://leetcode.cn/problems/number-of-operations-to-make-network-connected/
相关平台: LeetCode 1319
"""

```
class UnionFind:
    """
    并查集数据结构实现
    包含路径压缩和按秩合并优化
    """

    def __init__(self, n):
        """
        初始化并查集
        :param n: 节点数量
        """
        # parent[i]表示节点 i 的父节点
        self.parent = list(range(n))
        # rank[i]表示以 i 为根的树的高度上界
        self.rank = [1] * n
        # 连通分量数量
        self.components = n

    def find(self, x):
        """
        查找节点的根节点（代表元素）
        使用路径压缩优化
        :param x: 要查找的节点
        :return: 节点 x 所在集合的根节点
        """
        if self.parent[x] != x:
            # 路径压缩：将路径上的所有节点直接连接到根节点
            self.parent[x] = self.find(self.parent[x])
        return self.parent[x]

    def union(self, x, y):
```

```
        """
        合并两个集合
        使用按秩合并优化
        :param x: 第一个节点
        :param y: 第二个节点
        """
        root_x = self.find(x)
        root_y = self.find(y)

        # 如果已经在同一个集合中，则无需合并
        if root_x != root_y:
            # 按秩合并：将秩小的树合并到秩大的树下
            if self.rank[root_x] > self.rank[root_y]:
                self.parent[root_y] = root_x
            elif self.rank[root_x] < self.rank[root_y]:
                self.parent[root_x] = root_y
            else:
                # 秩相等时，任选一个作为根，并将其秩加 1
                self.parent[root_y] = root_x
                self.rank[root_x] += 1
            # 连通分量数量减 1
            self.components -= 1

    def get_components(self):
        """
        获取连通分量数量
        :return: 连通分量数量
        """
        return self.components


def make_connected(n, connections):
    """
    使用并查集解决连通网络的操作次数问题

    解题思路：
    1. 首先检查线缆数量是否足够，至少需要 n-1 条线缆才能连接 n 台计算机
    2. 使用并查集统计当前有多少个连通分量
    3. 要连接 k 个连通分量，需要 k-1 次操作

    时间复杂度：O(M * α(N))，其中 M 是 connections 的长度，N 是计算机数量，α 是阿克曼函数的反函数
    空间复杂度：O(N)
    是否为最优解：是
```

工程化考量：
1. 异常处理：检查输入参数的有效性
2. 可配置性：可以扩展支持权重边或其他网络属性
3. 线程安全：当前实现不是线程安全的

与机器学习等领域的联系：
1. 网络分析：社交网络、通信网络的连通性分析
2. 图论：最小生成树、网络流等算法的基础

语言特性差异:
Java: 对象引用和垃圾回收，数组初始化
C++: 指针操作和手动内存管理，vector 容器
Python: 动态类型和自动内存管理，list 初始化

极端输入场景:
1. 单个计算机
2. 线缆数量不足
3. 所有计算机已经连通
4. 大规模网络

性能优化:
1. 路径压缩优化 find 操作
2. 按秩合并优化 union 操作
3. 提前终止优化（线缆数量检查）

```
:param n: 计算机数量
:param connections: 线缆连接关系
:return: 最少操作次数，如果不可能则返回-1
"""
# 边界条件检查
if n <= 1:
    return 0

# 线缆数量不足，无法连接所有计算机
if len(connections) < n - 1:
    return -1

# 创建并查集
union_find = UnionFind(n)

# 处理所有连接
for connection in connections:
```

```
        a, b = connection[0], connection[1]
        # 如果两个计算机已经在同一个连通分量中，则这条线缆是冗余的
        if union_find.find(a) == union_find.find(b):
            continue
        else:
            union_find.union(a, b)

    # 统计连通分量数量
    components = union_find.get_components()

    # 需要 components-1 次操作来连接所有连通分量
    return components - 1


# 测试方法
if __name__ == "__main__":
    # 测试用例 1
    n1 = 4
    connections1 = [[0, 1], [0, 2], [1, 2]]
    print("测试用例 1 结果:", make_connected(n1, connections1))  # 预期输出: 1

    # 测试用例 2
    n2 = 6
    connections2 = [[0, 1], [0, 2], [0, 3], [1, 2], [1, 3]]
    print("测试用例 2 结果:", make_connected(n2, connections2))  # 预期输出: 2

    # 测试用例 3
    n3 = 6
    connections3 = [[0, 1], [0, 2], [0, 3], [1, 2]]
    print("测试用例 3 结果:", make_connected(n3, connections3))  # 预期输出: -1

    # 测试用例 4
    n4 = 5
    connections4 = [[0, 1], [0, 2], [3, 4], [2, 3]]
    print("测试用例 4 结果:", make_connected(n4, connections4))  # 预期输出: 0

    # 测试用例 5：单个计算机
    n5 = 1
    connections5 = []
    print("测试用例 5 结果:", make_connected(n5, connections5))  # 预期输出: 0
```

==========

文件: Code13_SmallestStringWithSwaps.java

```
==========================================
package class056;

import java.util.*;

/**
 * 交换字符串中的元素
 * 给你一个字符串 s，以及该字符串中的一些「索引对」数组 pairs，
 * 其中 pairs[i] = [a, b] 表示字符串中的两个索引（编号从 0 开始）。
 * 你可以任意多次交换在 pairs 中任意一对索引处的字符。
 * 返回在经过若干次交换后，s 可以变成的按字典序最小的字符串。
 *
 * 示例 1:
 * 输入: s = "dcab", pairs = [[0,3],[1,2]]
 * 输出: "bacd"
 * 解释:
 * 交换 s[0] 和 s[3], s = "bcad"
 * 交换 s[1] 和 s[2], s = "bacd"
 *
 * 示例 2:
 * 输入: s = "dcab", pairs = [[0,3],[1,2],[0,2]]
 * 输出: "abcd"
 * 解释:
 * 交换 s[0] 和 s[3], s = "bcad"
 * 交换 s[0] 和 s[2], s = "acbd"
 * 交换 s[1] 和 s[2], s = "abcd"
 *
 * 示例 3:
 * 输入: s = "cba", pairs = [[0,1],[1,2]]
 * 输出: "abc"
 * 解释:
 * 交换 s[0] 和 s[1], s = "bca"
 * 交换 s[1] 和 s[2], s = "bac"
 * 交换 s[0] 和 s[1], s = "abc"
 *
 * 约束条件：
 * 1 <= s.length <= 10^5
 * 0 <= pairs.length <= 10^5
 * 0 <= pairs[i][0], pairs[i][1] < s.length
 * s 中只含有小写英文字母
 *
 * 测试链接: https://leetcode.cn/problems/smallest-string-with-swaps/
```

```
 * 相关平台: LeetCode 1202
 */
public class Code13_SmallestStringWithSwaps {

    /**
     * 使用并查集解决交换字符串中的元素问题
     *
     * 解题思路:
     * 1. 使用并查集将可以相互交换的索引分组到同一个连通组件中
     * 2. 对于每个连通组件，将对应的字符收集起来并排序
     * 3. 将排序后的字符按顺序放回原位置，得到字典序最小的字符串
     *
     * 时间复杂度：O(N * log N + M * α(N))，其中N是字符串长度，M是索引对数量，α是阿克曼函数的反函数
     * 空间复杂度：O(N)
     *
     * @param s 输入字符串
     * @param pairs 索引对数组
     * @return 字典序最小的字符串
     */
    public static String smallestStringWithSwaps(String s, List<List<Integer>> pairs) {
        // 边界条件检查
        if (s == null || s.length() <= 1) {
            return s;
        }

        int n = s.length();

        // 创建并查集
        UnionFind uf = new UnionFind(n);

        // 处理所有索引对
        for (List<Integer> pair : pairs) {
            uf.union(pair.get(0), pair.get(1));
        }

        // 将同一连通组件的索引和字符分组
        Map<Integer, List<Integer>> components = new HashMap<>();
        for (int i = 0; i < n; i++) {
            int root = uf.find(i);
            components.computeIfAbsent(root, k -> new ArrayList<>()).add(i);
        }
```

```
        // 构造结果字符串
        char[] result = new char[n];

        // 对每个连通组件内的字符进行排序并放回对应位置
        for (List<Integer> indices : components.values()) {
            // 收集字符
            List<Character> chars = new ArrayList<>();
            for (int index : indices) {
                chars.add(s.charAt(index));
            }

            // 排序字符
            Collections.sort(chars);

            // 将排序后的字符放回对应位置
            for (int i = 0; i < indices.size(); i++) {
                result[indices.get(i)] = chars.get(i);
            }
        }

        return new String(result);
    }

    /**
     * 并查集数据结构实现
     * 包含路径压缩优化
     */
    static class UnionFind {
        private int[] parent;  // parent[i]表示节点 i 的父节点

        /**
         * 初始化并查集
         * @param n 节点数量
         */
        public UnionFind(int n) {
            parent = new int[n];
            // 初始时每个节点都是自己的父节点
            for (int i = 0; i < n; i++) {
                parent[i] = i;
            }
        }

        /**
```

```
         * 查找节点的根节点（代表元素）
         * 使用路径压缩优化
         * @param x 要查找的节点
         * @return 节点 x 所在集合的根节点
         */
        public int find(int x) {
            if (parent[x] != x) {
                // 路径压缩：将路径上的所有节点直接连接到根节点
                parent[x] = find(parent[x]);
            }
            return parent[x];
        }

        /**
         * 合并两个集合
         * @param x 第一个节点
         * @param y 第二个节点
         */
        public void union(int x, int y) {
            int rootX = find(x);
            int rootY = find(y);
            // 如果已经在同一个集合中，则无需合并
            if (rootX != rootY) {
                parent[rootX] = rootY;
            }
        }
    }

    // 测试方法
    public static void main(String[] args) {
        // 测试用例 1
        String s1 = "dcab";
        List<List<Integer>> pairs1 = new ArrayList<>();
        pairs1.add(Arrays.asList(0, 3));
        pairs1.add(Arrays.asList(1, 2));
        System.out.println("测试用例 1 结果: " + smallestStringWithSwaps(s1, pairs1)); // 预期输
出: "bacd"

        // 测试用例 2
        String s2 = "dcab";
        List<List<Integer>> pairs2 = new ArrayList<>();
        pairs2.add(Arrays.asList(0, 3));
        pairs2.add(Arrays.asList(1, 2));
```

```
        pairs2.add(Arrays.asList(0, 2));
        System.out.println("测试用例 2 结果: " + smallestStringWithSwaps(s2, pairs2)); // 预期输出: "abcd"

        // 测试用例 3
        String s3 = "cba";
        List<List<Integer>> pairs3 = new ArrayList<>();
        pairs3.add(Arrays.asList(0, 1));
        pairs3.add(Arrays.asList(1, 2));
        System.out.println("测试用例 3 结果: " + smallestStringWithSwaps(s3, pairs3)); // 预期输出: "abc"

        // 测试用例 4：空字符串
        String s4 = "";
        List<List<Integer>> pairs4 = new ArrayList<>();
        System.out.println("测试用例 4 结果: " + smallestStringWithSwaps(s4, pairs4)); // 预期输出: ""

        // 测试用例 5：单字符
        String s5 = "a";
        List<List<Integer>> pairs5 = new ArrayList<>();
        System.out.println("测试用例 5 结果: " + smallestStringWithSwaps(s5, pairs5)); // 预期输出: "a"
    }
}
```

==========

文件: Code13_SmallestStringWithSwaps.py

==========

```
"""
交换字符串中的元素
给你一个字符串 s，以及该字符串中的一些「索引对」数组 pairs，
其中 pairs[i] = [a, b] 表示字符串中的两个索引（编号从 0 开始）。
你可以任意多次交换在 pairs 中任意一对索引处的字符。
返回在经过若干次交换后，s 可以变成的按字典序最小的字符串。

示例 1:
输入: s = "dcab", pairs = [[0,3],[1,2]]
输出: "bacd"
解释:
交换 s[0] 和 s[3], s = "bcad"
交换 s[1] 和 s[2], s = "bacd"
```

示例 2:
输入: s = "dcab", pairs = [[0,3],[1,2],[0,2]]
输出: "abcd"
解释:
交换 s[0] 和 s[3], s = "bcad"
交换 s[0] 和 s[2], s = "acbd"
交换 s[1] 和 s[2], s = "abcd"

示例 3:
输入: s = "cba", pairs = [[0,1],[1,2]]
输出: "abc"
解释:
交换 s[0] 和 s[1], s = "bca"
交换 s[1] 和 s[2], s = "bac"
交换 s[0] 和 s[1], s = "abc"

约束条件：
1 <= s.length <= 10^5
0 <= pairs.length <= 10^5
0 <= pairs[i][0], pairs[i][1] < s.length
s 中只含有小写英文字母

测试链接: https://leetcode.cn/problems/smallest-string-with-swaps/
相关平台: LeetCode 1202
"""

```
class UnionFind:
    """
    并查集数据结构实现
    包含路径压缩优化
    """

    def __init__(self, n):
        """
        初始化并查集
        :param n: 节点数量
        """
        # parent[i]表示节点 i 的父节点
        self.parent = list(range(n))

    def find(self, x):
```

```
        """
        查找节点的根节点（代表元素）
        使用路径压缩优化
        :param x: 要查找的节点
        :return: 节点 x 所在集合的根节点
        """
        if self.parent[x] != x:
            # 路径压缩：将路径上的所有节点直接连接到根节点
            self.parent[x] = self.find(self.parent[x])
        return self.parent[x]

    def union(self, x, y):
        """
        合并两个集合
        :param x: 第一个节点
        :param y: 第二个节点
        """
        root_x = self.find(x)
        root_y = self.find(y)
        # 如果已经在同一个集合中，则无需合并
        if root_x != root_y:
            self.parent[root_x] = root_y


def smallest_string_with_swaps(s, pairs):
    """
    使用并查集解决交换字符串中的元素问题

    解题思路：
    1. 使用并查集将可以相互交换的索引分组到同一个连通组件中
    2. 对于每个连通组件，将对应的字符收集起来并排序
    3. 将排序后的字符按顺序放回原位置，得到字典序最小的字符串

    时间复杂度：O(N * log N + M * α(N))，其中 N 是字符串长度，M 是索引对数量，α 是阿克曼函数的反函数
    空间复杂度：O(N)
    是否为最优解：是

    工程化考量：
    1. 异常处理：检查输入参数的有效性
    2. 可配置性：可以扩展支持其他排序规则
    3. 线程安全：当前实现不是线程安全的
```

与机器学习等领域的联系：
1. 字符串处理：文本预处理和特征工程中的基础操作
2. 优化问题：在约束条件下寻找最优解

语言特性差异:
Java: 对象引用和垃圾回收，数组初始化
C++: 指针操作和手动内存管理，vector 容器
Python: 动态类型和自动内存管理，list 初始化

极端输入场景:
1. 空字符串
2. 单字符字符串
3. 没有索引对
4. 所有索引都可以相互交换
5. 大规模字符串

性能优化:
1. 路径压缩优化 find 操作
2. 使用哈希表快速分组索引
3. 对字符进行排序以获得最小字典序

:param s: 输入字符串
:param pairs: 索引对数组
:return: 字典序最小的字符串
"""
```
# 边界条件检查
if not s or len(s) <= 1:
    return s

n = len(s)

# 创建并查集
union_find = UnionFind(n)

# 处理所有索引对
for pair in pairs:
    union_find.union(pair[0], pair[1])

# 将同一连通组件的索引和字符分组
components = {}
for i in range(n):
    root = union_find.find(i)
    if root not in components:
```

```
            components[root] = []
        components[root].append(i)

    # 构造结果字符串
    result = [''] * n

    # 对每个连通组件内的字符进行排序并放回对应位置
    for indices in components.values():
        # 收集字符
        chars = [s[index] for index in indices]

        # 排序字符
        chars.sort()

        # 将排序后的字符放回对应位置
        for i, index in enumerate(indices):
            result[index] = chars[i]

    return ''.join(result)


# 测试方法
if __name__ == "__main__":
    # 测试用例 1
    s1 = "dcab"
    pairs1 = [[0, 3], [1, 2]]
    print("测试用例 1 结果:", smallest_string_with_swaps(s1, pairs1))  # 预期输出: "bacd"

    # 测试用例 2
    s2 = "dcab"
    pairs2 = [[0, 3], [1, 2], [0, 2]]
    print("测试用例 2 结果:", smallest_string_with_swaps(s2, pairs2))  # 预期输出: "abcd"

    # 测试用例 3
    s3 = "cba"
    pairs3 = [[0, 1], [1, 2]]
    print("测试用例 3 结果:", smallest_string_with_swaps(s3, pairs3))  # 预期输出: "abc"

    # 测试用例 4：空字符串
    s4 = ""
    pairs4 = []
    print("测试用例 4 结果:", smallest_string_with_swaps(s4, pairs4))  # 预期输出: ""
```

```
    # 测试用例 5：单字符
    s5 = "a"
    pairs5 = []
    print("测试用例 5 结果:", smallest_string_with_swaps(s5, pairs5))  # 预期输出: "a"

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

文件: Code14_UnionFindAdditionalProblems.cpp
==============================================
#include <iostream>
#include <vector>
#include <string>
#include <algorithm>
#include <unordered_map>
#include <tuple>

/**
 * 并查集补充题目 (C++版本)
 * 本文件包含更多使用并查集解决的经典题目
 */

/**
 * 题目 1: LeetCode 399. 除法求值
 * 链接: https://leetcode.cn/problems/evaluate-division/
 * 难度: 中等
 * 题目描述:
 * 给你一个变量对数组 equations 和一个实数值数组 values 作为已知条件，其中 equations[i] = [Ai, Bi] 和 values[i] 共同表示等式 Ai / Bi = values[i]。
 * 每个 Ai 或 Bi 是一个表示单个变量的字符串。
 * 另有一些以数组 queries 表示的问题，其中 queries[j] = [Cj, Dj] 表示第 j 个问题，请你根据已知条件找出 Cj / Dj = ? 的结果作为答案。
 * 如果存在某个无法确定的答案，则用 -1.0 替代。
 *
 * 注意：输入总是有效的，且不存在循环或冲突的结果。
 */
class EvaluateDivision {
private:
    /**
     * 带权并查集实现
     * 用于处理除法关系，维护变量之间的倍数关系
     */
    struct WeightedUnionFind {
        // 存储父节点
```

```
std::unordered_map<std::string, std::string> parent;
// 存储到父节点的权重（当前节点值 / 父节点值）
std::unordered_map<std::string, double> weight;

/**
 * 检查变量是否存在于并查集中
 */
bool contains(const std::string& x) const {
    return parent.find(x) != parent.end();
}

/**
 * 初始化变量
 */
void ensureExists(const std::string& x) {
    if (!contains(x)) {
        parent[x] = x;
        weight[x] = 1.0;
    }
}

/**
 * 查找操作，返回根节点和权重（x 的值 / 根节点的值）
 */
std::pair<std::string, double> find(const std::string& x) {
    ensureExists(x);

    if (parent[x] != x) {
        // 递归查找父节点
        auto [root, rootWeight] = find(parent[x]);

        // 路径压缩：将 x 直接指向根节点
        parent[x] = root;
        // 更新权重：x 到根的权重 = x 到父的权重 * 父到根的权重
        weight[x] *= rootWeight;
    }

    return {parent[x], weight[x]};
}

/**
 * 合并操作，表示 x / y = value
 */
```

```
        void unite(const std::string& x, const std::string& y, double value) {
            ensureExists(x);
            ensureExists(y);

            auto [xRoot, xWeight] = find(x);
            auto [yRoot, yWeight] = find(y);

            if (xRoot != yRoot) {
                // 将 x 的根节点连接到 y 的根节点
                parent[xRoot] = yRoot;
                // 更新权重：xRoot / yRoot = (y / yRoot) * (x / y) / (x / xRoot) = yWeight *
value / xWeight
                weight[xRoot] = yWeight * value / xWeight;
            }
        }
    };

public:
    /**
     * 使用带权并查集解决除法求值问题
     * 时间复杂度: O((E + Q) * α(N))，其中 E 是 equations 的长度，Q 是 queries 的长度，N 是不同变量
的数量
     * 空间复杂度: O(N)
     */
    std::vector<double> calcEquation(
        const std::vector<std::vector<std::string>>& equations,
        const std::vector<double>& values,
        const std::vector<std::vector<std::string>>& queries) {

        // 创建并初始化带权并查集
        WeightedUnionFind uf;

        // 构建并查集
        for (size_t i = 0; i < equations.size(); i++) {
            const std::string& var1 = equations[i][0];
            const std::string& var2 = equations[i][1];
            uf.unite(var1, var2, values[i]);
        }

        // 处理查询
        std::vector<double> results;
        results.reserve(queries.size());
```

```
        for (const auto& query : queries) {
            const std::string& var1 = query[0];
            const std::string& var2 = query[1];

            // 如果变量不存在于并查集中，结果为-1.0
            if (!uf.contains(var1) || !uf.contains(var2)) {
                results.push_back(-1.0);
                continue;
            }

            // 查找两个变量的根节点和权重
            auto [root1, weight1] = uf.find(var1);
            auto [root2, weight2] = uf.find(var2);

            // 如果根节点不同，说明无法确定结果
            if (root1 != root2) {
                results.push_back(-1.0);
            } else {
                // 结果等于 var1 到根的权重除以 var2 到根的权重
                results.push_back(weight1 / weight2);
            }
        }

        return results;
    }
};

/**
 * 题目 2: LeetCode 1697. 检查边长度限制的路径是否存在
 * 链接: https://leetcode.cn/problems/checking-existence-of-edge-length-limited-paths/
 * 难度: 困难
 * 题目描述:
 * 给你一个 n 个点组成的无向图边集 edgeList ，其中 edgeList[i] = [ui, vi, disi] 表示点 ui 和点 vi 之间有一条长度为 disi 的边。
 * 同时给你一个查询数组 queries ，其中 queries[j] = [pj, qj, limitj] ，你的任务是对于每个查询 queries[j] ，判断是否存在一条从 pj 到 qj 的路径，且路径中每条边的长度都严格小于 limitj 。
 */
class DistanceLimitedPathsExist {
private:
    /**
     * 标准并查集实现
     */
    struct UnionFind {
```

```
        std::vector<int> parent;
        std::vector<int> rank;

        UnionFind(int n) {
            parent.resize(n);
            rank.resize(n, 1);
            for (int i = 0; i < n; i++) {
                parent[i] = i;
            }
        }

        int find(int x) {
            if (parent[x] != x) {
                parent[x] = find(parent[x]); // 路径压缩
            }
            return parent[x];
        }

        void unite(int x, int y) {
            int rootX = find(x);
            int rootY = find(y);

            if (rootX != rootY) {
                if (rank[rootX] > rank[rootY]) {
                    parent[rootY] = rootX;
                } else if (rank[rootX] < rank[rootY]) {
                    parent[rootX] = rootY;
                } else {
                    parent[rootY] = rootX;
                    rank[rootX]++;
                }
            }
        }

        bool isConnected(int x, int y) {
            return find(x) == find(y);
        }
    };

public:
    /**
     * 使用并查集结合离线查询解决路径存在性问题
     * 时间复杂度: O(E log E + Q log Q + (E + Q) α(N))，其中 E 是边的数量，Q 是查询的数量，N 是节
```

点数量
 * 空间复杂度: O(N + Q)
 */
```
std::vector<bool> distanceLimitedPathsExist(int n, std::vector<std::vector<int>>& edgeList,
                                            std::vector<std::vector<int>>& queries) {
    // 创建并查集
    UnionFind uf(n);

    // 将查询按 limit 排序，并记录原始索引
    std::vector<std::tuple<int, int, int, int>> sortedQueries;
    for (size_t i = 0; i < queries.size(); i++) {
        sortedQueries.emplace_back(queries[i][2], queries[i][0], queries[i][1], i);
    }
    std::sort(sortedQueries.begin(), sortedQueries.end());

    // 将边按距离排序
    std::sort(edgeList.begin(), edgeList.end(),
            [](const std::vector<int>& a, const std::vector<int>& b) {
                return a[2] < b[2];
            });

    // 处理查询
    std::vector<bool> results(queries.size());
    size_t edgeIndex = 0;

    for (const auto& query : sortedQueries) {
        int limit = std::get<0>(query);
        int p = std::get<1>(query);
        int q = std::get<2>(query);
        int originalIndex = std::get<3>(query);

        // 将所有边权小于 limit 的边加入并查集
        while (edgeIndex < edgeList.size() && edgeList[edgeIndex][2] < limit) {
            uf.unite(edgeList[edgeIndex][0], edgeList[edgeIndex][1]);
            edgeIndex++;
        }

        // 检查 p 和 q 是否连通
        results[originalIndex] = uf.isConnected(p, q);
    }

    return results;
}
```

```
};

/**
 * 题目 3: LeetCode 1579. 保证图可完全遍历
 * 链接: https://leetcode.cn/problems/remove-max-number-of-edges-to-keep-graph-fully-traversable/
 * 难度: 困难
 * 题目描述:
 * Alice 和 Bob 共有一个无向图，其中包含 n 个节点和 3 种类型的边：
 * 类型 1：只能由 Alice 使用的边。
 * 类型 2：只能由 Bob 使用的边。
 * 类型 3：Alice 和 Bob 都可以使用的边。
 * 请你在保证图仍能被 Alice 和 Bob 完全遍历的前提下，找出可以删除的最大边数。
 * 如果从任何节点开始，Alice 和 Bob 都可以到达所有其他节点，则认为图是可以完全遍历的。
 */
class RemoveMaxNumberOfEdgesToKeepGraphFullyTraversable {
private:
    /**
     * 并查集实现，增加获取连通分量数量的功能
     */
    struct UnionFind {
        std::vector<int> parent;
        std::vector<int> rank;
        int componentCount;

        UnionFind(int n) : componentCount(n) {
            parent.resize(n);
            rank.resize(n, 1);
            for (int i = 0; i < n; i++) {
                parent[i] = i;
            }
        }

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

        void unite(int x, int y) {
            int rootX = find(x);
            int rootY = find(y);
```

```
            if (rootX != rootY) {
                if (rank[rootX] > rank[rootY]) {
                    parent[rootY] = rootX;
                } else if (rank[rootX] < rank[rootY]) {
                    parent[rootX] = rootY;
                } else {
                    parent[rootY] = rootX;
                    rank[rootX]++;
                }
                componentCount--;
            }
        }

        bool isConnected(int x, int y) {
            return find(x) == find(y);
        }

        int getComponentCount() const {
            return componentCount;
        }
    };

public:
    /**
     * 使用并查集解决最大边删除问题
     * 时间复杂度: O(E α(N))，其中 E 是边的数量，N 是节点数量
     * 空间复杂度: O(N)
     */
    int maxNumEdgesToRemove(int n, std::vector<std::vector<int>>& edges) {
        // 为 Alice 和 Bob 分别创建并查集
        UnionFind aliceUf(n);
        UnionFind bobUf(n);

        int edgesAdded = 0;

        // 首先处理类型 3 的边（两人共用），因为这些边优先级最高
        for (const auto& edge : edges) {
            if (edge[0] == 3) {
                bool aliceConnected = aliceUf.isConnected(edge[1] - 1, edge[2] - 1);
                bool bobConnected = bobUf.isConnected(edge[1] - 1, edge[2] - 1);

                if (!aliceConnected || !bobConnected) {
                    aliceUf.unite(edge[1] - 1, edge[2] - 1);
```

```
                bobUf.unite(edge[1] - 1, edge[2] - 1);
                edgesAdded++;
            }
        }
    }

    // 处理类型 1 和类型 2 的边
    for (const auto& edge : edges) {
        if (edge[0] == 1) {
            // Alice 专用边
            if (!aliceUf.isConnected(edge[1] - 1, edge[2] - 1)) {
                aliceUf.unite(edge[1] - 1, edge[2] - 1);
                edgesAdded++;
            }
        } else if (edge[0] == 2) {
            // Bob 专用边
            if (!bobUf.isConnected(edge[1] - 1, edge[2] - 1)) {
                bobUf.unite(edge[1] - 1, edge[2] - 1);
                edgesAdded++;
            }
        }
    }

    // 检查 Alice 和 Bob 是否都能完全遍历图
    bool aliceFullyConnected = aliceUf.getComponentCount() == 1;
    bool bobFullyConnected = bobUf.getComponentCount() == 1;

    if (!aliceFullyConnected || !bobFullyConnected) {
        return -1; // 无法满足完全遍历条件
    }

    // 可以删除的最大边数 = 总边数 - 必须保留的边数
    return edges.size() - edgesAdded;
}
};

/**
 * 题目 4: POJ 1182 食物链
 * 链接: http://poj.org/problem?id=1182
 * 难度: 中等
 * 题目描述:
 * 动物王国中有三类动物 A,B,C，这三类动物的食物链构成了有趣的环形。A 吃 B， B 吃 C，C 吃 A。
 * 现有 N 个动物，以 1－N 编号。每个动物都是 A,B,C 中的一种，但是我们并不知道它到底是哪一种。
```

```
 * 有人用两种说法对这 N 个动物所构成的食物链关系进行描述：
 * 第一种说法是"1 X Y"，表示 X 和 Y 是同类。
 * 第二种说法是"2 X Y"，表示 X 吃 Y。
 * 此人对 N 个动物，用上述两种说法，一句接一句地说出 K 句话，这 K 句话有的是真的，有的是假的。
 * 当一句话满足下列三条之一时，这句话就是假话，否则就是真话。
 * 1） 当前的话与前面的某些真的话冲突，就是假话；
 * 2） 当前的话中 X 或 Y 比 N 大，就是假话；
 * 3） 当前的话表示 X 吃 X，就是假话。
 * 你的任务是根据给定的 N（1 <= N <= 50,000）和 K 句话（0 <= K <= 100,000），输出假话的总数。
 */
class FoodChain {
private:
    /**
     * 查找根节点并进行路径压缩，同时更新关系
     */
    int find(int x, std::vector<int>& parent, std::vector<int>& relation) {
        if (parent[x] != x) {
            int originalParent = parent[x];
            parent[x] = find(parent[x], parent, relation);
            // 更新关系：x 到新根节点的关系 = x 到原父节点的关系 + 原父节点到新根节点的关系
            relation[x] = (relation[x] + relation[originalParent]) % 3;
        }
        return parent[x];
    }

public:
    /**
     * 使用带权并查集解决食物链问题
     * 时间复杂度: O(K α(N))，其中 K 是语句数量，N 是动物数量
     * 空间复杂度: O(N)
     */
    int findInvalidStatements(int n, std::vector<std::vector<int>>& statements) {
        // 初始化带权并查集，每个元素存储到父节点的关系（0 表示同类，1 表示吃父节点，2 表示被父节点
吃）
        std::vector<int> parent(n + 1); // 动物编号从 1 开始
        std::vector<int> rank(n + 1, 1);
        std::vector<int> relation(n + 1, 0); // relation[x]表示 x 到父节点的关系

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

        int invalidCount = 0;
```

```
for (const auto& statement : statements) {
    int type = statement[0];
    int x = statement[1];
    int y = statement[2];

    // 检查条件 2：X 或 Y 比 N 大
    if (x > n || y > n) {
        invalidCount++;
        continue;
    }

    // 检查条件 3：X 吃 X
    if (type == 2 && x == y) {
        invalidCount++;
        continue;
    }

    int rootX = find(x, parent, relation);
    int rootY = find(y, parent, relation);

    if (rootX == rootY) {
        // X 和 Y 已经在同一集合中，检查是否冲突
        int relationXToY = (relation[x] - relation[y] + 3) % 3;
        if (type == 1 && relationXToY != 0) {
            // 声明 X 和 Y 是同类，但实际不是
            invalidCount++;
        } else if (type == 2 && relationXToY != 1) {
            // 声明 X 吃 Y，但实际不是
            invalidCount++;
        }
    } else {
        // 合并两个集合
        if (rank[rootX] > rank[rootY]) {
            // 将 rootY 合并到 rootX
            parent[rootY] = rootX;
            // 计算 rootY 到 rootX 的关系
            // 期望关系: X 和 Y 的关系为 type-1
            relation[rootY] = (relation[x] - relation[y] + (type - 1) + 3) % 3;
        } else {
            // 将 rootX 合并到 rootY
            parent[rootX] = rootY;
            // 计算 rootX 到 rootY 的关系
```

```
                    relation[rootX] = (relation[y] - relation[x] + (3 - (type - 1)) + 3) % 3;

                    if (rank[rootX] == rank[rootY]) {
                        rank[rootY]++;
                    }
                }
            }
        }

        return invalidCount;
    }
};

/**
 * 并查集算法总结与技巧
 * 1. 基本并查集适用于：连通性问题、集合合并、环检测
 * 2. 带权并查集适用于：维护元素之间的关系（如食物链、除法关系等）
 * 3. 离线并查集适用于：需要按特定顺序处理查询的场景
 * 4. 优化技巧：路径压缩和按秩合并
 * 5. 实现注意事项：
 *    - 初始化时每个元素指向自己
 *    - find 操作要进行路径压缩
 *    - union 操作要按秩合并以保持树的平衡
 *    - 对于字符串或其他非整数类型，可以使用哈希表映射
 */

int main() {
    std::cout << "并查集补充题目 C++实现完成" << std::endl;
    return 0;
}

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

文件: Code14_UnionFindAdditionalProblems.java
==========================================
package class056;

import java.util.*;

/**
 * 并查集补充题目 (Java 版本)
 * 本文件包含更多使用并查集解决的经典题目
 */
```

```
/**
 * 题目 1: LeetCode 399. 除法求值
 * 链接: https://leetcode.cn/problems/evaluate-division/
 * 难度: 中等
 * 题目描述:
 * 给你一个变量对数组 equations 和一个实数值数组 values 作为已知条件，其中 equations[i] = [Ai, Bi] 和 values[i] 共同表示等式 Ai / Bi = values[i]。
 * 每个 Ai 或 Bi 是一个表示单个变量的字符串。
 * 另有一些以数组 queries 表示的问题，其中 queries[j] = [Cj, Dj] 表示第 j 个问题，请你根据已知条件找出 Cj / Dj = ? 的结果作为答案。
 * 如果存在某个无法确定的答案，则用 -1.0 替代。
 *
 * 注意：输入总是有效的，且不存在循环或冲突的结果。
 */
class EvaluateDivision {
    /**
     * 使用带权并查集解决除法求值问题
     * 时间复杂度: O((E + Q) * α(N))，其中 E 是 equations 的长度，Q 是 queries 的长度，N 是不同变量的数量
     * 空间复杂度: O(N)
     */
    public double[] calcEquation(List<List<String>> equations, double[] values, List<List<String>> queries) {
        // 创建并初始化带权并查集
        WeightedUnionFind uf = new WeightedUnionFind();

        // 构建并查集
        for (int i = 0; i < equations.size(); i++) {
            String var1 = equations.get(i).get(0);
            String var2 = equations.get(i).get(1);
            uf.union(var1, var2, values[i]);
        }

        // 处理查询
        double[] results = new double[queries.size()];
        for (int i = 0; i < queries.size(); i++) {
            String var1 = queries.get(i).get(0);
            String var2 = queries.get(i).get(1);

            // 如果变量不存在于并查集中，结果为-1.0
            if (!uf.contains(var1) || !uf.contains(var2)) {
                results[i] = -1.0;
```

```
            continue;
        }

        // 查找两个变量的根节点和权重
        double[] root1Info = uf.find(var1);
        double[] root2Info = uf.find(var2);

        // 如果根节点不同，说明无法确定结果
        if (!root1Info[0].equals(root2Info[0])) {
            results[i] = -1.0;
        } else {
            // 结果等于 var1 到根的权重除以 var2 到根的权重
            results[i] = root1Info[1] / root2Info[1];
        }
    }

    return results;
}

/**
 * 带权并查集实现
 * 用于处理除法关系，维护变量之间的倍数关系
 */
class WeightedUnionFind {
    // 存储父节点
    private Map<String, String> parent;
    // 存储到父节点的权重（当前节点值 / 父节点值）
    private Map<String, Double> weight;

    public WeightedUnionFind() {
        parent = new HashMap<>();
        weight = new HashMap<>();
    }

    /**
     * 检查变量是否存在于并查集中
     */
    public boolean contains(String x) {
        return parent.containsKey(x);
    }

    /**
     * 初始化变量
```

```
     */
    private void ensureExists(String x) {
        if (!contains(x)) {
            parent.put(x, x);
            weight.put(x, 1.0);
        }
    }

    /**
     * 查找操作，返回根节点和权重（x 的值 / 根节点的值）
     */
    public double[] find(String x) {
        ensureExists(x);

        if (!parent.get(x).equals(x)) {
            // 递归查找父节点
            double[] rootInfo = find(parent.get(x));
            String root = rootInfo[0].toString();
            double rootWeight = rootInfo[1];

            // 路径压缩：将 x 直接指向根节点
            parent.put(x, root);
            // 更新权重：x 到根的权重 = x 到父的权重 * 父到根的权重
            weight.put(x, weight.get(x) * rootWeight);
        }

        return new double[]{0, weight.get(x)}; // 第一个元素是根节点字符串，但这里用 0 占位，实际使用时需要修改
    }

    /**
     * 合并操作，表示 x / y = value
     */
    public void union(String x, String y, double value) {
        ensureExists(x);
        ensureExists(y);

        double[] xRootInfo = find(x);
        double[] yRootInfo = find(y);

        String xRoot = xRootInfo[0].toString();
        String yRoot = yRootInfo[0].toString();
        double xWeight = xRootInfo[1];
```

```
            double yWeight = yRootInfo[1];

            if (!xRoot.equals(yRoot)) {
                // 将 x 的根节点连接到 y 的根节点
                parent.put(xRoot, yRoot);
                // 更新权重：xRoot / yRoot = (y / yRoot) * (x / y) / (x / xRoot) = yWeight * value / xWeight
                weight.put(xRoot, yWeight * value / xWeight);
            }
        }
    }
}

/**
 * 题目 2: LeetCode 1697. 检查边长度限制的路径是否存在
 * 链接: https://leetcode.cn/problems/checking-existence-of-edge-length-limited-paths/
 * 难度: 困难
 * 题目描述:
 * 给你一个 n 个点组成的无向图边集 edgeList ，其中 edgeList[i] = [ui, vi, disi] 表示点 ui 和点 vi 之间有一条长度为 disi 的边。
 * 同时给你一个查询数组 queries ，其中 queries[j] = [pj, qj, limitj] ，你的任务是对于每个查询 queries[j] ，判断是否存在一条从 pj 到 qj 的路径，且路径中每条边的长度都严格小于 limitj 。
 */
class DistanceLimitedPathsExist {
    /**
     * 使用并查集结合离线查询解决路径存在性问题
     * 时间复杂度: O(E log E + Q log Q + (E + Q) α(N))，其中 E 是边的数量，Q 是查询的数量，N 是节点数量
     * 空间复杂度: O(N + Q)
     */
    public boolean[] distanceLimitedPathsExist(int n, int[][] edgeList, int[][] queries) {
        // 创建并查集
        UnionFind uf = new UnionFind(n);

        // 将查询按 limit 排序，并记录原始索引
        int[][] sortedQueries = new int[queries.length][4];
        for (int i = 0; i < queries.length; i++) {
            sortedQueries[i][0] = queries[i][0]; // pj
            sortedQueries[i][1] = queries[i][1]; // qj
            sortedQueries[i][2] = queries[i][2]; // limitj
            sortedQueries[i][3] = i;             // 原始索引
        }
        Arrays.sort(sortedQueries, Comparator.comparingInt(a -> a[2]));
```

```
    // 将边按距离排序
    Arrays.sort(edgeList, Comparator.comparingInt(a -> a[2]));

    // 处理查询
    boolean[] results = new boolean[queries.length];
    int edgeIndex = 0;

    for (int[] query : sortedQueries) {
        int p = query[0];
        int q = query[1];
        int limit = query[2];
        int originalIndex = query[3];

        // 将所有边权小于 limit 的边加入并查集
        while (edgeIndex < edgeList.length && edgeList[edgeIndex][2] < limit) {
            uf.union(edgeList[edgeIndex][0], edgeList[edgeIndex][1]);
            edgeIndex++;
        }

        // 检查 p 和 q 是否连通
        results[originalIndex] = uf.isConnected(p, q);
    }

    return results;
}

/**
 * 标准并查集实现
 */
class UnionFind {
    private int[] parent;
    private int[] rank;

    public UnionFind(int n) {
        parent = new int[n];
        rank = new int[n];
        for (int i = 0; i < n; i++) {
            parent[i] = i;
            rank[i] = 1;
        }
    }
```

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

        public void union(int x, int y) {
            int rootX = find(x);
            int rootY = find(y);

            if (rootX != rootY) {
                if (rank[rootX] > rank[rootY]) {
                    parent[rootY] = rootX;
                } else if (rank[rootX] < rank[rootY]) {
                    parent[rootX] = rootY;
                } else {
                    parent[rootY] = rootX;
                    rank[rootX]++;
                }
            }
        }

        public boolean isConnected(int x, int y) {
            return find(x) == find(y);
        }
    }
}

/**
 * 题目 3: LeetCode 1579. 保证图可完全遍历
 * 链接: https://leetcode.cn/problems/remove-max-number-of-edges-to-keep-graph-fully-traversable/
 * 难度: 困难
 * 题目描述:
 * Alice 和 Bob 共有一个无向图，其中包含 n 个节点和 3 种类型的边：
 * 类型 1：只能由 Alice 使用的边。
 * 类型 2：只能由 Bob 使用的边。
 * 类型 3：Alice 和 Bob 都可以使用的边。
 * 请你在保证图仍能被 Alice 和 Bob 完全遍历的前提下，找出可以删除的最大边数。
 * 如果从任何节点开始，Alice 和 Bob 都可以到达所有其他节点，则认为图是可以完全遍历的。
 */
class RemoveMaxNumberOfEdgesToKeepGraphFullyTraversable {
    /**
```

```
 * 使用并查集解决最大边删除问题
 * 时间复杂度: O(E α(N))，其中 E 是边的数量，N 是节点数量
 * 空间复杂度: O(N)
 */
public int maxNumEdgesToRemove(int n, int[][] edges) {
    // 为 Alice 和 Bob 分别创建并查集
    UnionFind aliceUf = new UnionFind(n);
    UnionFind bobUf = new UnionFind(n);

    int edgesAdded = 0;

    // 首先处理类型 3 的边（两人共用），因为这些边优先级最高
    for (int[] edge : edges) {
        if (edge[0] == 3) {
            boolean aliceConnected = aliceUf.isConnected(edge[1] - 1, edge[2] - 1);
            boolean bobConnected = bobUf.isConnected(edge[1] - 1, edge[2] - 1);

            if (!aliceConnected || !bobConnected) {
                aliceUf.union(edge[1] - 1, edge[2] - 1);
                bobUf.union(edge[1] - 1, edge[2] - 1);
                edgesAdded++;
            }
        }
    }

    // 处理类型 1 和类型 2 的边
    for (int[] edge : edges) {
        if (edge[0] == 1) {
            // Alice 专用边
            if (!aliceUf.isConnected(edge[1] - 1, edge[2] - 1)) {
                aliceUf.union(edge[1] - 1, edge[2] - 1);
                edgesAdded++;
            }
        } else if (edge[0] == 2) {
            // Bob 专用边
            if (!bobUf.isConnected(edge[1] - 1, edge[2] - 1)) {
                bobUf.union(edge[1] - 1, edge[2] - 1);
                edgesAdded++;
            }
        }
    }

    // 检查 Alice 和 Bob 是否都能完全遍历图
```

```
        boolean aliceFullyConnected = aliceUf.getComponentCount() == 1;
        boolean bobFullyConnected = bobUf.getComponentCount() == 1;

        if (!aliceFullyConnected || !bobFullyConnected) {
            return -1; // 无法满足完全遍历条件
        }

        // 可以删除的最大边数 = 总边数 - 必须保留的边数
        return edges.length - edgesAdded;
    }

    /**
     * 并查集实现，增加获取连通分量数量的功能
     */
    class UnionFind {
        private int[] parent;
        private int[] rank;
        private int componentCount;

        public UnionFind(int n) {
            parent = new int[n];
            rank = new int[n];
            componentCount = n;

            for (int i = 0; i < n; i++) {
                parent[i] = i;
                rank[i] = 1;
            }
        }

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

        public void union(int x, int y) {
            int rootX = find(x);
            int rootY = find(y);

            if (rootX != rootY) {
                if (rank[rootX] > rank[rootY]) {
```

```
                parent[rootY] = rootX;
            } else if (rank[rootX] < rank[rootY]) {
                parent[rootX] = rootY;
            } else {
                parent[rootY] = rootX;
                rank[rootX]++;
            }
            componentCount--;
        }
    }

    public boolean isConnected(int x, int y) {
        return find(x) == find(y);
    }

    public int getComponentCount() {
        return componentCount;
    }
  }
}

/**
 * 题目 4: POJ 1182 食物链
 * 链接: http://poj.org/problem?id=1182
 * 难度: 中等
 * 题目描述:
 * 动物王国中有三类动物 A,B,C，这三类动物的食物链构成了有趣的环形。A 吃 B， B 吃 C，C 吃 A。
 * 现有 N 个动物，以 1－N 编号。每个动物都是 A,B,C 中的一种，但是我们并不知道它到底是哪一种。
 * 有人用两种说法对这 N 个动物所构成的食物链关系进行描述：
 * 第一种说法是"1 X Y"，表示 X 和 Y 是同类。
 * 第二种说法是"2 X Y"，表示 X 吃 Y。
 * 此人对 N 个动物，用上述两种说法，一句接一句地说出 K 句话，这 K 句话有的是真的，有的是假的。
 * 当一句话满足下列三条之一时，这句话就是假话，否则就是真话。
 * 1） 当前的话与前面的某些真的话冲突，就是假话；
 * 2） 当前的话中 X 或 Y 比 N 大，就是假话；
 * 3） 当前的话表示 X 吃 X，就是假话。
 * 你的任务是根据给定的 N（1 <= N <= 50,000）和 K 句话（0 <= K <= 100,000），输出假话的总数。
 */
class FoodChain {
    /**
     * 使用带权并查集解决食物链问题
     * 时间复杂度: O(K α(N))，其中 K 是语句数量，N 是动物数量
     * 空间复杂度: O(N)
```

```
     */
    public int findInvalidStatements(int n, int[][] statements) {
        // 初始化带权并查集，每个元素存储到父节点的关系（0 表示同类，1 表示吃父节点，2 表示被父节点吃）
        int[] parent = new int[n + 1]; // 动物编号从 1 开始
        int[] rank = new int[n + 1];
        int[] relation = new int[n + 1]; // relation[x]表示 x 到父节点的关系

        for (int i = 0; i <= n; i++) {
            parent[i] = i;
            rank[i] = 1;
            relation[i] = 0; // 初始时每个节点的父节点是自己，关系为同类
        }

        int invalidCount = 0;

        for (int[] statement : statements) {
            int type = statement[0];
            int x = statement[1];
            int y = statement[2];

            // 检查条件 2：X 或 Y 比 N 大
            if (x > n || y > n) {
                invalidCount++;
                continue;
            }

            // 检查条件 3：X 吃 X
            if (type == 2 && x == y) {
                invalidCount++;
                continue;
            }

            int rootX = find(x, parent, relation);
            int rootY = find(y, parent, relation);

            if (rootX == rootY) {
                // X 和 Y 已经在同一集合中，检查是否冲突
                int relationXToY = (relation[x] - relation[y] + 3) % 3;
                if (type == 1 && relationXToY != 0) {
                    // 声明 X 和 Y 是同类，但实际不是
                    invalidCount++;
                } else if (type == 2 && relationXToY != 1) {
```

```
                // 声明 X 吃 Y，但实际不是
                invalidCount++;
            }
        } else {
            // 合并两个集合
            if (rank[rootX] > rank[rootY]) {
                // 将 rootY 合并到 rootX
                parent[rootY] = rootX;
                // 计算 rootY 到 rootX 的关系
                // 期望关系: X 和 Y 的关系为 type-1
                // relation[x] + relation[rootY] ≡ (type-1) + relation[y] (mod 3)
                relation[rootY] = (relation[x] - relation[y] + (type - 1) + 3) % 3;
            } else {
                // 将 rootX 合并到 rootY
                parent[rootX] = rootY;
                // 计算 rootX 到 rootY 的关系
                // relation[y] + relation[rootX] ≡ (3 - (type-1)) + relation[x] (mod 3)
                relation[rootX] = (relation[y] - relation[x] + (3 - (type - 1)) + 3) % 3;

                if (rank[rootX] == rank[rootY]) {
                    rank[rootY]++;
                }
            }
        }

        return invalidCount;
    }

    /**
     * 查找根节点并进行路径压缩，同时更新关系
     */
    private int find(int x, int[] parent, int[] relation) {
        if (parent[x] != x) {
            int originalParent = parent[x];
            parent[x] = find(parent[x], parent, relation);
            // 更新关系：x 到新根节点的关系 = x 到原父节点的关系 + 原父节点到新根节点的关系
            relation[x] = (relation[x] + relation[originalParent]) % 3;
        }
        return parent[x];
    }
}
```

```
/**
 * 并查集算法总结与技巧
 * 1. 基本并查集适用于：连通性问题、集合合并、环检测
 * 2. 带权并查集适用于：维护元素之间的关系（如食物链、除法关系等）
 * 3. 离线并查集适用于：需要按特定顺序处理查询的场景
 * 4. 优化技巧：路径压缩和按秩合并
 * 5. 实现注意事项：
 *    - 初始化时每个元素指向自己
 *    - find 操作要进行路径压缩
 *    - union 操作要按秩合并以保持树的平衡
 *    - 对于字符串或其他非整数类型，可以使用哈希表映射
 */
public class Code14_UnionFindAdditionalProblems {
    public static void main(String[] args) {
        // 可以在这里添加测试代码
        System.out.println("并查集补充题目实现完成");
    }
}

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

文件: Code14_UnionFindAdditionalProblems.py
==============================================
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

"""
并查集补充题目 (Python 版本)
本文件包含更多使用并查集解决的经典题目
"""

from typing import List, Dict, Tuple, Optional

"""
题目 1: LeetCode 399. 除法求值
链接: https://leetcode.cn/problems/evaluate-division/
难度: 中等
题目描述:
给你一个变量对数组 equations 和一个实数值数组 values 作为已知条件，其中 equations[i] = [Ai, Bi] 和 values[i] 共同表示等式 Ai / Bi = values[i]。
每个 Ai 或 Bi 是一个表示单个变量的字符串。
另有一些以数组 queries 表示的问题，其中 queries[j] = [Cj, Dj] 表示第 j 个问题，请你根据已知条件找出 Cj / Dj = ? 的结果作为答案。
```

如果存在某个无法确定的答案，则用 -1.0 替代。

注意：输入总是有效的，且不存在循环或冲突的结果。
"""

```
class EvaluateDivision:
    """
    使用带权并查集解决除法求值问题
    时间复杂度: O((E + Q) * α(N))，其中 E 是 equations 的长度，Q 是 queries 的长度，N 是不同变量的数量
    空间复杂度: O(N)
    """
    def calcEquation(self, equations: List[List[str]], values: List[float],
                     queries: List[List[str]]) -> List[float]:
        # 创建并初始化带权并查集
        uf = self.WeightedUnionFind()

        # 构建并查集
        for i in range(len(equations)):
            var1 = equations[i][0]
            var2 = equations[i][1]
            uf.unite(var1, var2, values[i])

        # 处理查询
        results = []
        for query in queries:
            var1 = query[0]
            var2 = query[1]

            # 如果变量不存在于并查集中，结果为-1.0
            if var1 not in uf.parent or var2 not in uf.parent:
                results.append(-1.0)
                continue

            # 查找两个变量的根节点和权重
            root1, weight1 = uf.find(var1)
            root2, weight2 = uf.find(var2)

            # 如果根节点不同，说明无法确定结果
            if root1 != root2:
                results.append(-1.0)
            else:
                # 结果等于 var1 到根的权重除以 var2 到根的权重
                results.append(weight1 / weight2)
```

```
    return results

"""
带权并查集实现
用于处理除法关系，维护变量之间的倍数关系
"""
class WeightedUnionFind:
    def __init__(self):
        # 存储父节点
        self.parent = dict()
        # 存储到父节点的权重（当前节点值 / 父节点值）
        self.weight = dict()

    """
    查找操作，返回根节点和权重（x 的值 / 根节点的值）
    同时进行路径压缩
    """
    def find(self, x: str) -> Tuple[str, float]:
        # 确保 x 存在于并查集中
        if x not in self.parent:
            self.parent[x] = x
            self.weight[x] = 1.0

        if self.parent[x] != x:
            # 递归查找父节点
            origin_parent = self.parent[x]
            root, root_weight = self.find(origin_parent)

            # 路径压缩：将 x 直接指向根节点
            self.parent[x] = root
            # 更新权重：x 到根的权重 = x 到父的权重 * 父到根的权重
            self.weight[x] *= root_weight

        return self.parent[x], self.weight[x]

    """
    合并操作，表示 x / y = value
    """
    def unite(self, x: str, y: str, value: float) -> None:
        # 确保 x 和 y 存在于并查集中
        if x not in self.parent:
            self.parent[x] = x
```

```
                self.weight[x] = 1.0
            if y not in self.parent:
                self.parent[y] = y
                self.weight[y] = 1.0

            # 查找 x 和 y 的根节点
            x_root, x_weight = self.find(x)
            y_root, y_weight = self.find(y)

            if x_root != y_root:
                # 将 x 的根节点连接到 y 的根节点
                self.parent[x_root] = y_root
                # 更新权重：xRoot / yRoot = (y / yRoot) * (x / y) / (x / xRoot) = yWeight * value / xWeight
                self.weight[x_root] = y_weight * value / x_weight

"""
题目 2: LeetCode 1697. 检查边长度限制的路径是否存在
链接: https://leetcode.cn/problems/checking-existence-of-edge-length-limited-paths/
难度: 困难
题目描述:
给你一个 n 个点组成的无向图边集 edgeList ，其中 edgeList[i] = [ui, vi, disi] 表示点 ui 和点 vi 之间有一条长度为 disi 的边。
同时给你一个查询数组 queries ，其中 queries[j] = [pj, qj, limitj] ，你的任务是对于每个查询 queries[j] ，判断是否存在一条从 pj 到 qj 的路径，且路径中每条边的长度都严格小于 limitj 。
"""
class DistanceLimitedPathsExist:
    """
    使用并查集结合离线查询解决路径存在性问题
    时间复杂度: O(E log E + Q log Q + (E + Q) α(N))，其中 E 是边的数量，Q 是查询的数量，N 是节点数量
    空间复杂度: O(N + Q)
    """
    def distanceLimitedPathsExist(self, n: int, edgeList: List[List[int]],
                                  queries: List[List[int]]) -> List[bool]:
        # 创建并查集
        uf = self.UnionFind(n)

        # 将查询按 limit 排序，并记录原始索引
        sorted_queries = [(limit, p, q, idx) for idx, (p, q, limit) in enumerate(queries)]
        sorted_queries.sort()

        # 将边按距离排序
```

```
    edgeList.sort(key=lambda x: x[2])

    # 处理查询
    results = [False] * len(queries)
    edge_index = 0

    for limit, p, q, original_index in sorted_queries:
        # 将所有边权小于 limit 的边加入并查集
        while edge_index < len(edgeList) and edgeList[edge_index][2] < limit:
            uf.unite(edgeList[edge_index][0], edgeList[edge_index][1])
            edge_index += 1

        # 检查 p 和 q 是否连通
        results[original_index] = uf.is_connected(p, q)

    return results

"""
标准并查集实现
"""
class UnionFind:
    def __init__(self, n: int):
        self.parent = list(range(n))
        self.rank = [1] * n

    def find(self, x: int) -> int:
        if self.parent[x] != x:
            self.parent[x] = self.find(self.parent[x])  # 路径压缩
        return self.parent[x]

    def unite(self, x: int, y: int) -> None:
        root_x = self.find(x)
        root_y = self.find(y)

        if root_x != root_y:
            if self.rank[root_x] > self.rank[root_y]:
                self.parent[root_y] = root_x
            elif self.rank[root_x] < self.rank[root_y]:
                self.parent[root_x] = root_y
            else:
                self.parent[root_y] = root_x
                self.rank[root_x] += 1
```

```
        def is_connected(self, x: int, y: int) -> bool:
            return self.find(x) == self.find(y)

"""
题目 3: LeetCode 1579. 保证图可完全遍历
链接: https://leetcode.cn/problems/remove-max-number-of-edges-to-keep-graph-fully-traversable/
难度: 困难
题目描述:
Alice 和 Bob 共有一个无向图，其中包含 n 个节点和 3 种类型的边：
类型 1：只能由 Alice 使用的边。
类型 2：只能由 Bob 使用的边。
类型 3：Alice 和 Bob 都可以使用的边。
请你在保证图仍能被 Alice 和 Bob 完全遍历的前提下，找出可以删除的最大边数。
如果从任何节点开始，Alice 和 Bob 都可以到达所有其他节点，则认为图是可以完全遍历的。
"""
class RemoveMaxNumberOfEdgesToKeepGraphFullyTraversable:
    """
    使用并查集解决最大边删除问题
    时间复杂度: O(E α(N))，其中 E 是边的数量，N 是节点数量
    空间复杂度: O(N)
    """
    def maxNumEdgesToRemove(self, n: int, edges: List[List[int]]) -> int:
        # 为 Alice 和 Bob 分别创建并查集
        alice_uf = self.UnionFind(n)
        bob_uf = self.UnionFind(n)

        edges_added = 0

        # 首先处理类型 3 的边（两人共用），因为这些边优先级最高
        for edge in edges:
            if edge[0] == 3:
                alice_connected = alice_uf.is_connected(edge[1] - 1, edge[2] - 1)
                bob_connected = bob_uf.is_connected(edge[1] - 1, edge[2] - 1)

                if not alice_connected or not bob_connected:
                    alice_uf.unite(edge[1] - 1, edge[2] - 1)
                    bob_uf.unite(edge[1] - 1, edge[2] - 1)
                    edges_added += 1

        # 处理类型 1 和类型 2 的边
        for edge in edges:
            if edge[0] == 1:
                # Alice 专用边
```

```
            if not alice_uf.is_connected(edge[1] - 1, edge[2] - 1):
                alice_uf.unite(edge[1] - 1, edge[2] - 1)
                edges_added += 1
        elif edge[0] == 2:
            # Bob 专用边
            if not bob_uf.is_connected(edge[1] - 1, edge[2] - 1):
                bob_uf.unite(edge[1] - 1, edge[2] - 1)
                edges_added += 1

    # 检查 Alice 和 Bob 是否都能完全遍历图
    alice_fully_connected = alice_uf.get_component_count() == 1
    bob_fully_connected = bob_uf.get_component_count() == 1

    if not alice_fully_connected or not bob_fully_connected:
        return -1  # 无法满足完全遍历条件

    # 可以删除的最大边数 = 总边数 - 必须保留的边数
    return len(edges) - edges_added

"""
并查集实现，增加获取连通分量数量的功能
"""
class UnionFind:
    def __init__(self, n: int):
        self.parent = list(range(n))
        self.rank = [1] * n
        self.component_count = n

    def find(self, x: int) -> int:
        if self.parent[x] != x:
            self.parent[x] = self.find(self.parent[x])
        return self.parent[x]

    def unite(self, x: int, y: int) -> None:
        root_x = self.find(x)
        root_y = self.find(y)

        if root_x != root_y:
            if self.rank[root_x] > self.rank[root_y]:
                self.parent[root_y] = root_x
            elif self.rank[root_x] < self.rank[root_y]:
                self.parent[root_x] = root_y
            else:
```

```
                self.parent[root_y] = root_x
                self.rank[root_x] += 1
            self.component_count -= 1

    def is_connected(self, x: int, y: int) -> bool:
        return self.find(x) == self.find(y)

    def get_component_count(self) -> int:
        return self.component_count

"""
题目 4: POJ 1182 食物链
链接: http://poj.org/problem?id=1182
难度: 中等
题目描述:
动物王国中有三类动物 A,B,C，这三类动物的食物链构成了有趣的环形。A 吃 B， B 吃 C，C 吃 A。
现有 N 个动物，以 1－N 编号。每个动物都是 A,B,C 中的一种，但是我们并不知道它到底是哪一种。
有人用两种说法对这 N 个动物所构成的食物链关系进行描述：
第一种说法是"1 X Y"，表示 X 和 Y 是同类。
第二种说法是"2 X Y"，表示 X 吃 Y。
此人对 N 个动物，用上述两种说法，一句接一句地说出 K 句话，这 K 句话有的是真的，有的是假的。
当一句话满足下列三条之一时，这句话就是假话，否则就是真话。
1） 当前的话与前面的某些真的话冲突，就是假话；
2） 当前的话中 X 或 Y 比 N 大，就是假话；
3） 当前的话表示 X 吃 X，就是假话。
你的任务是根据给定的 N（1 <= N <= 50,000）和 K 句话（0 <= K <= 100,000），输出假话的总数。
"""
class FoodChain:
    """
    使用带权并查集解决食物链问题
    时间复杂度: O(K α(N))，其中 K 是语句数量，N 是动物数量
    空间复杂度: O(N)
    """
    def findInvalidStatements(self, n: int, statements: List[List[int]]) -> int:
        # 初始化带权并查集，每个元素存储到父节点的关系（0 表示同类，1 表示吃父节点，2 表示被父节点吃）
        parent = list(range(n + 1))  # 动物编号从 1 开始
        rank = [1] * (n + 1)
        relation = [0] * (n + 1)  # relation[x]表示 x 到父节点的关系

        def find(x: int) -> int:
            """查找根节点并进行路径压缩，同时更新关系"""
            if parent[x] != x:
```

```
        original_parent = parent[x]
        parent[x] = find(parent[x])
        # 更新关系：x 到新根节点的关系 = x 到原父节点的关系 + 原父节点到新根节点的关系
        relation[x] = (relation[x] + relation[original_parent]) % 3
    return parent[x]

invalid_count = 0

for statement in statements:
    type_statement = statement[0]
    x = statement[1]
    y = statement[2]

    # 检查条件 2：X 或 Y 比 N 大
    if x > n or y > n:
        invalid_count += 1
        continue

    # 检查条件 3：X 吃 X
    if type_statement == 2 and x == y:
        invalid_count += 1
        continue

    root_x = find(x)
    root_y = find(y)

    if root_x == root_y:
        # X 和 Y 已经在同一集合中，检查是否冲突
        relation_x_to_y = (relation[x] - relation[y] + 3) % 3
        if type_statement == 1 and relation_x_to_y != 0:
            # 声明 X 和 Y 是同类，但实际不是
            invalid_count += 1
        elif type_statement == 2 and relation_x_to_y != 1:
            # 声明 X 吃 Y，但实际不是
            invalid_count += 1
    else:
        # 合并两个集合
        if rank[root_x] > rank[root_y]:
            # 将 rootY 合并到 rootX
            parent[root_y] = root_x
            # 计算 rootY 到 rootX 的关系
            # 期望关系: X 和 Y 的关系为 type-1
            relation[root_y] = (relation[x] - relation[y] + (type_statement - 1) + 3) % 3
```

```
                else:
                    # 将 rootX 合并到 rootY
                    parent[root_x] = root_y
                    # 计算 rootX 到 rootY 的关系
                    relation[root_x] = (relation[y] - relation[x] + (3 - (type_statement - 1)) +
3) % 3

                    if rank[root_x] == rank[root_y]:
                        rank[root_y] += 1

        return invalid_count

"""
并查集算法总结与技巧
1. 基本并查集适用于：连通性问题、集合合并、环检测
2. 带权并查集适用于：维护元素之间的关系（如食物链、除法关系等）
3. 离线并查集适用于：需要按特定顺序处理查询的场景
4. 优化技巧：路径压缩和按秩合并
5. 实现注意事项：
   - 初始化时每个元素指向自己
   - find 操作要进行路径压缩
   - union 操作要按秩合并以保持树的平衡
   - 对于字符串或其他非整数类型，可以使用哈希表映射
"""

def main():
    print("并查集补充题目 Python 实现完成")

if __name__ == "__main__":
    main()

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

文件: Code15_LeetCode128.cpp
==============================================
#include <iostream>
#include <vector>
#include <unordered_map>
#include <unordered_set>
#include <algorithm>
#include <climits>
using namespace std;
```

```
/**
 * LeetCode 128. 最长连续序列
 * 链接: https://leetcode.cn/problems/longest-consecutive-sequence/
 * 难度: 中等
 *
 * 题目描述:
 * 给定一个未排序的整数数组 nums ，找出数字连续的最长序列（不要求序列元素在原数组中连续）的长度。
 * 请你设计并实现时间复杂度为 O(n) 的算法解决此问题。
 *
 * 示例 1:
 * 输入: nums = [100,4,200,1,3,2]
 * 输出: 4
 * 解释: 最长数字连续序列是 [1, 2, 3, 4]。它的长度为 4。
 *
 * 示例 2:
 * 输入: nums = [0,3,7,2,5,8,4,6,0,1]
 * 输出: 9
 *
 * 约束条件:
 * 0 <= nums.length <= 10^5
 * -10^9 <= nums[i] <= 10^9
 */

class UnionFind {
private:
    unordered_map<int, int> parent;
    unordered_map<int, int> size;
    int maxSize;

public:
    UnionFind(const vector<int>& nums) {
        maxSize = 0;
        for (int num : nums) {
            parent[num] = num;
            size[num] = 1;
        }
        if (!nums.empty()) {
            maxSize = 1;
        }
    }

    bool contains(int num) {
        return parent.find(num) != parent.end();
```

```
    }

    int find(int x) {
        if (parent[x] != x) {
            parent[x] = find(parent[x]);  // 路径压缩
        }
        return parent[x];
    }

    void unionSets(int x, int y) {
        if (!contains(x) || !contains(y)) {
            return;
        }

        int rootX = find(x);
        int rootY = find(y);

        if (rootX != rootY) {
            // 按大小合并，小树合并到大树下
            if (size[rootX] < size[rootY]) {
                parent[rootX] = rootY;
                size[rootY] += size[rootX];
                maxSize = max(maxSize, size[rootY]);
            } else {
                parent[rootY] = rootX;
                size[rootX] += size[rootY];
                maxSize = max(maxSize, size[rootX]);
            }
        }
    }

    int getMaxSize() {
        return maxSize;
    }
};

class Solution {
public:
    /**
     * 方法1: 使用并查集解决最长连续序列问题
     * 时间复杂度: O(n * α(n)) ≈ O(n)，其中 α 是阿克曼函数的反函数
     * 空间复杂度: O(n)
     *
```

```
 * 解题思路:
 * 1. 使用哈希表记录每个数字对应的并查集节点
 * 2. 对于每个数字，检查其相邻数字是否存在，如果存在则合并集合
 * 3. 记录每个集合的大小，返回最大集合的大小
 */
int longestConsecutive(vector<int>& nums) {
    if (nums.empty()) {
        return 0;
    }

    UnionFind uf(nums);

    // 遍历数组，合并相邻数字
    for (int num : nums) {
        // 如果存在 num-1，则合并 num 和 num-1
        if (uf.contains(num - 1)) {
            uf.unionSets(num, num - 1);
        }
        // 如果存在 num+1，则合并 num 和 num+1
        if (uf.contains(num + 1)) {
            uf.unionSets(num, num + 1);
        }
    }

    return uf.getMaxSize();
}

/**
 * 方法 2: 使用哈希表 + 遍历的优化解法
 * 时间复杂度: O(n)
 * 空间复杂度: O(n)
 *
 * 解题思路:
 * 1. 将所有数字存入哈希表
 * 2. 遍历数组，对于每个数字，如果它是序列的起点（即 num-1 不存在），则向后查找连续序列
 * 3. 记录最长序列长度
 */
int longestConsecutiveHashSet(vector<int>& nums) {
    if (nums.empty()) {
        return 0;
    }

    unordered_set<int> numSet(nums.begin(), nums.end());
```

```
    int longestStreak = 0;

    for (int num : numSet) {
        // 只有当num是序列的起点时才进行查找
        if (numSet.find(num - 1) == numSet.end()) {
            int currentNum = num;
            int currentStreak = 1;

            // 向后查找连续序列
            while (numSet.find(currentNum + 1) != numSet.end()) {
                currentNum++;
                currentStreak++;
            }

            longestStreak = max(longestStreak, currentStreak);
        }
    }

    return longestStreak;
}

/**
 * 方法3: 排序解法（不满足O(n)时间复杂度要求，但思路简单）
 * 时间复杂度: O(n log n)
 * 空间复杂度: O(1) 或 O(n)（取决于排序算法）
 */
int longestConsecutiveSort(vector<int>& nums) {
    if (nums.empty()) {
        return 0;
    }

    sort(nums.begin(), nums.end());

    int longestStreak = 1;
    int currentStreak = 1;

    for (int i = 1; i < nums.size(); i++) {
        // 处理重复数字
        if (nums[i] != nums[i - 1]) {
            // 检查是否连续
            if (nums[i] == nums[i - 1] + 1) {
                currentStreak++;
            } else {
```

```
                    longestStreak = max(longestStreak, currentStreak);
                    currentStreak = 1;
                }
            }
            // 如果数字重复，保持 currentStreak 不变
        }

        return max(longestStreak, currentStreak);
    }
};

// 测试函数
void testSolution() {
    Solution solution;

    // 测试用例 1
    vector<int> nums1 = {100, 4, 200, 1, 3, 2};
    cout << "测试用例 1 - 并查集解法: " << solution.longestConsecutive(nums1) << endl; // 预期: 4
    cout << "测试用例 1 - 哈希表解法: " << solution.longestConsecutiveHashSet(nums1) << endl; // 预期: 4
    cout << "测试用例 1 - 排序解法: " << solution.longestConsecutiveSort(nums1) << endl; // 预期: 4

    // 测试用例 2
    vector<int> nums2 = {0, 3, 7, 2, 5, 8, 4, 6, 0, 1};
    cout << "测试用例 2 - 并查集解法: " << solution.longestConsecutive(nums2) << endl; // 预期: 9
    cout << "测试用例 2 - 哈希表解法: " << solution.longestConsecutiveHashSet(nums2) << endl; // 预期: 9
    cout << "测试用例 2 - 排序解法: " << solution.longestConsecutiveSort(nums2) << endl; // 预期: 9

    // 测试用例 3: 空数组
    vector<int> nums3 = {};
    cout << "测试用例 3 - 并查集解法: " << solution.longestConsecutive(nums3) << endl; // 预期: 0
    cout << "测试用例 3 - 哈希表解法: " << solution.longestConsecutiveHashSet(nums3) << endl; // 预期: 0
    cout << "测试用例 3 - 排序解法: " << solution.longestConsecutiveSort(nums3) << endl; // 预期: 0

    // 测试用例 4: 单个元素
    vector<int> nums4 = {5};
    cout << "测试用例 4 - 并查集解法: " << solution.longestConsecutive(nums4) << endl; // 预期: 1
    cout << "测试用例 4 - 哈希表解法: " << solution.longestConsecutiveHashSet(nums4) << endl; //
```

```
预期: 1
    cout << "测试用例 4 - 排序解法: " << solution.longestConsecutiveSort(nums4) << endl; // 预期: 1

    // 测试用例 5: 重复元素
    vector<int> nums5 = {1, 2, 0, 1};
    cout << "测试用例 5 - 并查集解法: " << solution.longestConsecutive(nums5) << endl; // 预期: 3
    cout << "测试用例 5 - 哈希表解法: " << solution.longestConsecutiveHashSet(nums5) << endl; // 预期: 3
    cout << "测试用例 5 - 排序解法: " << solution.longestConsecutiveSort(nums5) << endl; // 预期: 3
}

int main() {
    testSolution();
    return 0;
}

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

文件: Code15_LeetCode128.java
==========================================
package class056;

import java.util.*;

/**
 * LeetCode 128. 最长连续序列
 * 链接: https://leetcode.cn/problems/longest-consecutive-sequence/
 * 难度: 中等
 *
 * 题目描述:
 * 给定一个未排序的整数数组 nums ，找出数字连续的最长序列（不要求序列元素在原数组中连续）的长度。
 * 请你设计并实现时间复杂度为 O(n) 的算法解决此问题。
 *
 * 示例 1:
 * 输入: nums = [100,4,200,1,3,2]
 * 输出: 4
 * 解释: 最长数字连续序列是 [1, 2, 3, 4]。它的长度为 4。
 *
 * 示例 2:
 * 输入: nums = [0,3,7,2,5,8,4,6,0,1]
 * 输出: 9
```

```
 *
 * 约束条件:
 * 0 <= nums.length <= 10^5
 * -10^9 <= nums[i] <= 10^9
 */
public class Code15_LeetCode128 {

    /**
     * 方法 1: 使用并查集解决最长连续序列问题
     * 时间复杂度: O(n * α(n)) ≈ O(n)，其中 α 是阿克曼函数的反函数
     * 空间复杂度: O(n)
     *
     * 解题思路:
     * 1. 使用哈希表记录每个数字对应的并查集节点
     * 2. 对于每个数字，检查其相邻数字是否存在，如果存在则合并集合
     * 3. 记录每个集合的大小，返回最大集合的大小
     */
    public int longestConsecutive(int[] nums) {
        if (nums == null || nums.length == 0) {
            return 0;
        }

        UnionFind uf = new UnionFind(nums);

        // 遍历数组，合并相邻数字
        for (int num : nums) {
            // 如果存在 num-1，则合并 num 和 num-1
            if (uf.contains(num - 1)) {
                uf.union(num, num - 1);
            }
            // 如果存在 num+1，则合并 num 和 num+1
            if (uf.contains(num + 1)) {
                uf.union(num, num + 1);
            }
        }

        return uf.getMaxSize();
    }

    /**
     * 并查集实现，专门用于处理整数序列
     */
    static class UnionFind {
```

```
    private Map<Integer, Integer> parent;  // 存储父节点
    private Map<Integer, Integer> size;    // 存储集合大小
    private int maxSize;                   // 最大集合大小

    public UnionFind(int[] nums) {
        parent = new HashMap<>();
        size = new HashMap<>();
        maxSize = 0;

        // 初始化并查集
        for (int num : nums) {
            parent.put(num, num);
            size.put(num, 1);
        }
        if (!nums.isEmpty()) {
            maxSize = 1;
        }
    }

    /**
     * 检查数字是否存在于并查集中
     */
    public boolean contains(int num) {
        return parent.containsKey(num);
    }

    /**
     * 查找操作，使用路径压缩优化
     */
    public int find(int x) {
        if (parent.get(x) != x) {
            parent.put(x, find(parent.get(x)));  // 路径压缩
        }
        return parent.get(x);
    }

    /**
     * 合并操作
     */
    public void union(int x, int y) {
        if (!contains(x) || !contains(y)) {
            return;
        }
```

```
        int rootX = find(x);
        int rootY = find(y);

        if (rootX != rootY) {
            // 按大小合并，小树合并到大树下
            if (size.get(rootX) < size.get(rootY)) {
                parent.put(rootX, rootY);
                size.put(rootY, size.get(rootY) + size.get(rootX));
                maxSize = Math.max(maxSize, size.get(rootY));
            } else {
                parent.put(rootY, rootX);
                size.put(rootX, size.get(rootX) + size.get(rootY));
                maxSize = Math.max(maxSize, size.get(rootX));
            }
        }
    }

    /**
     * 获取最大集合大小
     */
    public int getMaxSize() {
        return maxSize;
    }
}

/**
 * 方法 2: 使用哈希表 + 遍历的优化解法
 * 时间复杂度: O(n)
 * 空间复杂度: O(n)
 *
 * 解题思路:
 * 1. 将所有数字存入哈希表
 * 2. 遍历数组，对于每个数字，如果它是序列的起点（即 num-1 不存在），则向后查找连续序列
 * 3. 记录最长序列长度
 */
public int longestConsecutiveHashSet(int[] nums) {
    if (nums == null || nums.length == 0) {
        return 0;
    }

    Set<Integer> numSet = new HashSet<>();
    for (int num : nums) {
```

```
        numSet.add(num);
    }

    int longestStreak = 0;

    for (int num : numSet) {
        // 只有当num是序列的起点时才进行查找
        if (!numSet.contains(num - 1)) {
            int currentNum = num;
            int currentStreak = 1;

            // 向后查找连续序列
            while (numSet.contains(currentNum + 1)) {
                currentNum++;
                currentStreak++;
            }

            longestStreak = Math.max(longestStreak, currentStreak);
        }
    }

    return longestStreak;
}

/**
 * 方法3: 排序解法（不满足O(n)时间复杂度要求，但思路简单）
 * 时间复杂度: O(n log n)
 * 空间复杂度: O(1) 或 O(n)（取决于排序算法）
 */
public int longestConsecutiveSort(int[] nums) {
    if (nums == null || nums.length == 0) {
        return 0;
    }

    Arrays.sort(nums);

    int longestStreak = 1;
    int currentStreak = 1;

    for (int i = 1; i < nums.length; i++) {
        // 处理重复数字
        if (nums[i] != nums[i - 1]) {
            // 检查是否连续
```

```
                if (nums[i] == nums[i - 1] + 1) {
                    currentStreak++;
                } else {
                    longestStreak = Math.max(longestStreak, currentStreak);
                    currentStreak = 1;
                }
            }
            // 如果数字重复，保持 currentStreak 不变
        }

        return Math.max(longestStreak, currentStreak);
    }

    // 测试方法
    public static void main(String[] args) {
        Code15_LeetCode128 solution = new Code15_LeetCode128();

        // 测试用例 1
        int[] nums1 = {100, 4, 200, 1, 3, 2};
        System.out.println("测试用例 1 - 并查集解法: " + solution.longestConsecutive(nums1)); //
预期: 4
        System.out.println("测试用例 1 - 哈希表解法: " +
solution.longestConsecutiveHashSet(nums1)); // 预期: 4
        System.out.println("测试用例 1 - 排序解法: " + solution.longestConsecutiveSort(nums1)); //
预期: 4

        // 测试用例 2
        int[] nums2 = {0, 3, 7, 2, 5, 8, 4, 6, 0, 1};
        System.out.println("测试用例 2 - 并查集解法: " + solution.longestConsecutive(nums2)); //
预期: 9
        System.out.println("测试用例 2 - 哈希表解法: " +
solution.longestConsecutiveHashSet(nums2)); // 预期: 9
        System.out.println("测试用例 2 - 排序解法: " + solution.longestConsecutiveSort(nums2)); //
预期: 9

        // 测试用例 3: 空数组
        int[] nums3 = {};
        System.out.println("测试用例 3 - 并查集解法: " + solution.longestConsecutive(nums3)); //
预期: 0
        System.out.println("测试用例 3 - 哈希表解法: " +
solution.longestConsecutiveHashSet(nums3)); // 预期: 0
        System.out.println("测试用例 3 - 排序解法: " + solution.longestConsecutiveSort(nums3)); //
预期: 0
```

```
        // 测试用例 4: 单个元素
        int[] nums4 = {5};
        System.out.println("测试用例 4 - 并查集解法: " + solution.longestConsecutive(nums4)); //
预期: 1
        System.out.println("测试用例 4 - 哈希表解法: " +
solution.longestConsecutiveHashSet(nums4)); // 预期: 1
        System.out.println("测试用例 4 - 排序解法: " + solution.longestConsecutiveSort(nums4)); //
预期: 1

        // 测试用例 5: 重复元素
        int[] nums5 = {1, 2, 0, 1};
        System.out.println("测试用例 5 - 并查集解法: " + solution.longestConsecutive(nums5)); //
预期: 3
        System.out.println("测试用例 5 - 哈希表解法: " +
solution.longestConsecutiveHashSet(nums5)); // 预期: 3
        System.out.println("测试用例 5 - 排序解法: " + solution.longestConsecutiveSort(nums5)); //
预期: 3
    }
}
```

==========

文件: Code15_LeetCode128.py

==========

```
from typing import List

class UnionFind:
    """
    并查集实现，专门用于处理整数序列
    """
    def __init__(self, nums: List[int]):
        self.parent = {}
        self.size = {}
        self.max_size = 0

        # 初始化并查集
        for num in nums:
            self.parent[num] = num
            self.size[num] = 1
        if nums:
            self.max_size = 1
```

```
    def contains(self, num: int) -> bool:
        """检查数字是否存在于并查集中"""
        return num in self.parent

    def find(self, x: int) -> int:
        """查找操作，使用路径压缩优化"""
        if self.parent[x] != x:
            self.parent[x] = self.find(self.parent[x])  # 路径压缩
        return self.parent[x]

    def union(self, x: int, y: int) -> None:
        """合并操作"""
        if not self.contains(x) or not self.contains(y):
            return

        root_x = self.find(x)
        root_y = self.find(y)

        if root_x != root_y:
            # 按大小合并，小树合并到大树下
            if self.size[root_x] < self.size[root_y]:
                self.parent[root_x] = root_y
                self.size[root_y] += self.size[root_x]
                self.max_size = max(self.max_size, self.size[root_y])
            else:
                self.parent[root_y] = root_x
                self.size[root_x] += self.size[root_y]
                self.max_size = max(self.max_size, self.size[root_x])

    def get_max_size(self) -> int:
        """获取最大集合大小"""
        return self.max_size

class Solution:
    """
    LeetCode 128. 最长连续序列
    链接: https://leetcode.cn/problems/longest-consecutive-sequence/
    难度: 中等

    题目描述:
    给定一个未排序的整数数组 nums ，找出数字连续的最长序列（不要求序列元素在原数组中连续）的长度。
    请你设计并实现时间复杂度为 O(n) 的算法解决此问题。
```

示例 1:
输入: nums = [100,4,200,1,3,2]
输出: 4
解释: 最长数字连续序列是 [1, 2, 3, 4]。它的长度为 4。

示例 2:
输入: nums = [0,3,7,2,5,8,4,6,0,1]
输出: 9

约束条件:
0 <= nums.length <= 10^5
-10^9 <= nums[i] <= 10^9
"""

```
def longestConsecutive(self, nums: List[int]) -> int:
    """
    方法 1: 使用并查集解决最长连续序列问题
    时间复杂度: O(n * α(n)) ≈ O(n)，其中 α 是阿克曼函数的反函数
    空间复杂度: O(n)

    解题思路:
    1. 使用哈希表记录每个数字对应的并查集节点
    2. 对于每个数字，检查其相邻数字是否存在，如果存在则合并集合
    3. 记录每个集合的大小，返回最大集合的大小
    """
    if not nums:
        return 0

    uf = UnionFind(nums)

    # 遍历数组，合并相邻数字
    for num in nums:
        # 如果存在 num-1，则合并 num 和 num-1
        if uf.contains(num - 1):
            uf.union(num, num - 1)
        # 如果存在 num+1，则合并 num 和 num+1
        if uf.contains(num + 1):
            uf.union(num, num + 1)

    return uf.get_max_size()

def longestConsecutiveHashSet(self, nums: List[int]) -> int:
```

```
    """
    方法 2: 使用哈希表 + 遍历的优化解法
    时间复杂度: O(n)
    空间复杂度: O(n)

    解题思路:
    1. 将所有数字存入哈希表
    2. 遍历数组，对于每个数字，如果它是序列的起点（即 num-1 不存在），则向后查找连续序列
    3. 记录最长序列长度
    """
    if not nums:
        return 0

    num_set = set(nums)
    longest_streak = 0

    for num in num_set:
        # 只有当 num 是序列的起点时才进行查找
        if num - 1 not in num_set:
            current_num = num
            current_streak = 1

            # 向后查找连续序列
            while current_num + 1 in num_set:
                current_num += 1
                current_streak += 1

            longest_streak = max(longest_streak, current_streak)

    return longest_streak

def longestConsecutiveSort(self, nums: List[int]) -> int:
    """
    方法 3: 排序解法（不满足 O(n)时间复杂度要求，但思路简单）
    时间复杂度: O(n log n)
    空间复杂度: O(1) 或 O(n)（取决于排序算法）
    """
    if not nums:
        return 0

    nums_sorted = sorted(nums)

    longest_streak = 1
```

```
        current_streak = 1

        for i in range(1, len(nums_sorted)):
            # 处理重复数字
            if nums_sorted[i] != nums_sorted[i - 1]:
                # 检查是否连续
                if nums_sorted[i] == nums_sorted[i - 1] + 1:
                    current_streak += 1
                else:
                    longest_streak = max(longest_streak, current_streak)
                    current_streak = 1
            # 如果数字重复，保持 current_streak 不变

        return max(longest_streak, current_streak)

def test_solution():
    """测试函数"""
    solution = Solution()

    # 测试用例 1
    nums1 = [100, 4, 200, 1, 3, 2]
    print(f"测试用例 1 - 并查集解法: {solution.longestConsecutive(nums1)}")  # 预期: 4
    print(f"测试用例 1 - 哈希表解法: {solution.longestConsecutiveHashSet(nums1)}")  # 预期: 4
    print(f"测试用例 1 - 排序解法: {solution.longestConsecutiveSort(nums1)}")  # 预期: 4

    # 测试用例 2
    nums2 = [0, 3, 7, 2, 5, 8, 4, 6, 0, 1]
    print(f"测试用例 2 - 并查集解法: {solution.longestConsecutive(nums2)}")  # 预期: 9
    print(f"测试用例 2 - 哈希表解法: {solution.longestConsecutiveHashSet(nums2)}")  # 预期: 9
    print(f"测试用例 2 - 排序解法: {solution.longestConsecutiveSort(nums2)}")  # 预期: 9

    # 测试用例 3: 空数组
    nums3 = []
    print(f"测试用例 3 - 并查集解法: {solution.longestConsecutive(nums3)}")  # 预期: 0
    print(f"测试用例 3 - 哈希表解法: {solution.longestConsecutiveHashSet(nums3)}")  # 预期: 0
    print(f"测试用例 3 - 排序解法: {solution.longestConsecutiveSort(nums3)}")  # 预期: 0

    # 测试用例 4: 单个元素
    nums4 = [5]
    print(f"测试用例 4 - 并查集解法: {solution.longestConsecutive(nums4)}")  # 预期: 1
    print(f"测试用例 4 - 哈希表解法: {solution.longestConsecutiveHashSet(nums4)}")  # 预期: 1
    print(f"测试用例 4 - 排序解法: {solution.longestConsecutiveSort(nums4)}")  # 预期: 1
```

```
    # 测试用例 5: 重复元素
    nums5 = [1, 2, 0, 1]
    print(f"测试用例 5 - 并查集解法: {solution.longestConsecutive(nums5)}")  # 预期: 3
    print(f"测试用例 5 - 哈希表解法: {solution.longestConsecutiveHashSet(nums5)}")  # 预期: 3
    print(f"测试用例 5 - 排序解法: {solution.longestConsecutiveSort(nums5)}")  # 预期: 3

if __name__ == "__main__":
    test_solution()

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

文件: Code16_LeetCode305.cpp
==============================================
#include <iostream>
#include <vector>
#include <unordered_map>
#include <algorithm>
#include <climits>
using namespace std;

/**
 * LeetCode 305. 岛屿数量 II
 * 链接: https://leetcode.cn/problems/number-of-islands-ii/
 * 难度: 困难
 *
 * 题目描述:
 * 给你一个大小为 m x n 的二进制网格 grid 。网格表示一个地图，其中，0 表示水，1 表示陆地。
 * 最初，网格中的单元格要么是水（0），要么是陆地（1）。
 * 你可以执行 addLand 操作，将位置 (row, col) 的水变成陆地。
 * 返回一个结果数组，其中每个结果[i] 表示在第 i 次操作后，地图中岛屿的数量。
 *
 * 注意: 一个岛屿被水包围，并且通过水平或垂直方向上相邻的陆地连接而成。
 * 你可以假设网格网格的四边均被水包围。
 *
 * 示例 1:
 * 输入: m = 3, n = 3, positions = [[0,0],[0,1],[1,2],[2,1]]
 * 输出: [1,1,2,3]
 *
 * 示例 2:
 * 输入: m = 1, n = 1, positions = [[0,0]]
 * 输出: [1]
 *
 * 约束条件:
```

```
 * 1 <= m, n, positions.length <= 10^4
 * positions[i].length == 2
 * 0 <= positions[i][0] < m
 * 0 <= positions[i][1] < n
 */

class UnionFind {
private:
    vector<int> parent;
    vector<int> rank;
    int count;
    int n;

public:
    UnionFind(int m, int n) : n(n) {
        int size = m * n;
        parent.resize(size, -1);  // -1 表示水
        rank.resize(size, 0);
        count = 0;
    }

    // 将二维坐标转换为一维索引
    int getIndex(int x, int y) {
        return x * n + y;
    }

    // 添加陆地
    void addLand(int x, int y) {
        int index = getIndex(x, y);
        if (parent[index] == -1) {
            parent[index] = index;  // 指向自己
            rank[index] = 1;
            count++;
        }
    }

    // 查找操作，使用路径压缩优化
    int find(int x, int y) {
        int index = getIndex(x, y);
        if (parent[index] == -1) {
            return -1;  // 水的位置
        }
```

```
        if (parent[index] != index) {
            parent[index] = find(parent[index] / n, parent[index] % n);
        }
        return parent[index];
    }

    // 合并两个陆地
    void unionSets(int x1, int y1, int x2, int y2) {
        int root1 = find(x1, y1);
        int root2 = find(x2, y2);

        // 如果有一个是水，或者已经在同一个集合中，则不需要合并
        if (root1 == -1 || root2 == -1 || root1 == root2) {
            return;
        }

        // 按秩合并
        if (rank[root1] > rank[root2]) {
            parent[root2] = root1;
        } else if (rank[root1] < rank[root2]) {
            parent[root1] = root2;
        } else {
            parent[root2] = root1;
            rank[root1]++;
        }
        count--;  // 合并后岛屿数量减 1
    }

    // 获取当前岛屿数量
    int getCount() {
        return count;
    }
};

class Solution {
public:
    /**
     * 使用并查集解决动态岛屿数量问题
     * 时间复杂度: O(L * α(m*n))，其中 L 是 positions 的长度
     * 空间复杂度: O(m*n)
     */
    vector<int> numIslands2(int m, int n, vector<vector<int>>& positions) {
        vector<int> result;
```

```
    if (m <= 0 || n <= 0 || positions.empty()) {
        return result;
    }

    UnionFind uf(m, n);
    vector<vector<int>> grid(m, vector<int>(n, 0));  // 0 表示水，1 表示陆地

    // 四个方向：上、右、下、左
    vector<vector<int>> directions = {{-1, 0}, {0, 1}, {1, 0}, {0, -1}};

    for (auto& pos : positions) {
        int x = pos[0], y = pos[1];

        // 如果该位置已经是陆地，直接添加当前岛屿数量
        if (grid[x][y] == 1) {
            result.push_back(uf.getCount());
            continue;
        }

        // 标记为陆地
        grid[x][y] = 1;
        uf.addLand(x, y);  // 岛屿数量加 1

        // 检查四个方向，合并相邻的陆地
        for (auto& dir : directions) {
            int newX = x + dir[0];
            int newY = y + dir[1];

            // 检查新位置是否在网格内且是陆地
            if (newX >= 0 && newX < m && newY >= 0 && newY < n && grid[newX][newY] == 1) {
                uf.unionSets(x, y, newX, newY);
            }
        }

        result.push_back(uf.getCount());
    }

    return result;
}

/**
 * 方法 2: 使用哈希表存储陆地位置的优化解法
 * 时间复杂度: O(L * α(L))，其中 L 是 positions 的长度
```

```
     * 空间复杂度: O(L)
     */
    vector<int> numIslands2Optimized(int m, int n, vector<vector<int>>& positions) {
        vector<int> result;
        if (m <= 0 || n <= 0 || positions.empty()) {
            return result;
        }

        // 使用哈希表存储陆地位置，避免使用 m*n 的数组
        unordered_map<int, int> parent;  // 位置索引 -> 父节点索引
        unordered_map<int, int> rank;    // 位置索引 -> 秩
        int count = 0;

        // 四个方向
        vector<vector<int>> directions = {{-1, 0}, {0, 1}, {1, 0}, {0, -1}};

        for (auto& pos : positions) {
            int x = pos[0], y = pos[1];
            int index = x * n + y;

            // 如果该位置已经是陆地，直接添加当前岛屿数量
            if (parent.find(index) != parent.end()) {
                result.push_back(count);
                continue;
            }

            // 添加新陆地
            parent[index] = index;
            rank[index] = 1;
            count++;

            // 检查四个方向
            for (auto& dir : directions) {
                int newX = x + dir[0];
                int newY = y + dir[1];
                int newIndex = newX * n + newY;

                // 检查新位置是否在网格内且是陆地
                if (newX >= 0 && newX < m && newY >= 0 && newY < n && parent.find(newIndex) !=
parent.end()) {
                    // 合并集合
                    int root1 = find(index, parent);
                    int root2 = find(newIndex, parent);
```

```
                    if (root1 != root2) {
                        // 按秩合并
                        if (rank[root1] > rank[root2]) {
                            parent[root2] = root1;
                        } else if (rank[root1] < rank[root2]) {
                            parent[root1] = root2;
                        } else {
                            parent[root2] = root1;
                            rank[root1]++;
                        }
                        count--;
                    }
                }
            }

            result.push_back(count);
        }

        return result;
    }

private:
    /**
     * 查找操作（用于优化解法）
     */
    int find(int x, unordered_map<int, int>& parent) {
        if (parent[x] != x) {
            parent[x] = find(parent[x], parent);  // 路径压缩
        }
        return parent[x];
    }
};

// 测试函数
void testSolution() {
    Solution solution;

    // 测试用例 1
    int m1 = 3, n1 = 3;
    vector<vector<int>> positions1 = {{0, 0}, {0, 1}, {1, 2}, {2, 1}};
    vector<int> result1 = solution.numIslands2(m1, n1, positions1);
    cout << "测试用例 1 - 方法 1: ";
```

```
for (int num : result1) cout << num << " ";
cout << endl;  // 预期: 1 1 2 3

vector<int> result1Opt = solution.numIslands2Optimized(m1, n1, positions1);
cout << "测试用例 1 - 方法 2: ";
for (int num : result1Opt) cout << num << " ";
cout << endl;  // 预期: 1 1 2 3

// 测试用例 2
int m2 = 1, n2 = 1;
vector<vector<int>> positions2 = {{0, 0}};
vector<int> result2 = solution.numIslands2(m2, n2, positions2);
cout << "测试用例 2 - 方法 1: ";
for (int num : result2) cout << num << " ";
cout << endl;  // 预期: 1

vector<int> result2Opt = solution.numIslands2Optimized(m2, n2, positions2);
cout << "测试用例 2 - 方法 2: ";
for (int num : result2Opt) cout << num << " ";
cout << endl;  // 预期: 1

// 测试用例 3: 重复添加同一位置
int m3 = 2, n3 = 2;
vector<vector<int>> positions3 = {{0, 0}, {0, 0}, {0, 1}};
vector<int> result3 = solution.numIslands2(m3, n3, positions3);
cout << "测试用例 3 - 方法 1: ";
for (int num : result3) cout << num << " ";
cout << endl;  // 预期: 1 1 1

vector<int> result3Opt = solution.numIslands2Optimized(m3, n3, positions3);
cout << "测试用例 3 - 方法 2: ";
for (int num : result3Opt) cout << num << " ";
cout << endl;  // 预期: 1 1 1

// 测试用例 4: 形成一个大岛屿
int m4 = 2, n4 = 2;
vector<vector<int>> positions4 = {{0, 0}, {0, 1}, {1, 0}, {1, 1}};
vector<int> result4 = solution.numIslands2(m4, n4, positions4);
cout << "测试用例 4 - 方法 1: ";
for (int num : result4) cout << num << " ";
cout << endl;  // 预期: 1 2 2 1

vector<int> result4Opt = solution.numIslands2Optimized(m4, n4, positions4);
```

```
    cout << "测试用例 4 - 方法 2: ";
    for (int num : result4Opt) cout << num << " ";
    cout << endl;  // 预期: 1 2 2 1
}

int main() {
    testSolution();
    return 0;
}
```

==========

文件: Code16_LeetCode305.java

==========

```
package class056;

import java.util.*;

/**
 * LeetCode 305. 岛屿数量 II
 * 链接: https://leetcode.cn/problems/number-of-islands-ii/
 * 难度: 困难
 *
 * 题目描述:
 * 给你一个大小为 m x n 的二进制网格 grid 。网格表示一个地图，其中，0 表示水，1 表示陆地。
 * 最初，网格中的单元格要么是水（0），要么是陆地（1）。
 * 你可以执行 addLand 操作，将位置 (row, col) 的水变成陆地。
 * 返回一个结果数组，其中每个结果[i] 表示在第 i 次操作后，地图中岛屿的数量。
 *
 * 注意: 一个岛屿被水包围，并且通过水平或垂直方向上相邻的陆地连接而成。
 * 你可以假设网格网格的四边均被水包围。
 *
 * 示例 1:
 * 输入: m = 3, n = 3, positions = [[0,0],[0,1],[1,2],[2,1]]
 * 输出: [1,1,2,3]
 *
 * 示例 2:
 * 输入: m = 1, n = 1, positions = [[0,0]]
 * 输出: [1]
 *
 * 约束条件:
 * 1 <= m, n, positions.length <= 10^4
 * positions[i].length == 2
```

```
 * 0 <= positions[i][0] < m
 * 0 <= positions[i][1] < n
 */
public class Code16_LeetCode305 {

    /**
     * 使用并查集解决动态岛屿数量问题
     * 时间复杂度: O(L * α(m*n))，其中 L 是 positions 的长度
     * 空间复杂度: O(m*n)
     *
     * 解题思路:
     * 1. 初始化并查集，大小为 m*n，初始时所有位置都是水（不属于任何集合）
     * 2. 对于每个添加陆地的操作:
     *    - 如果该位置已经是陆地，直接跳过
     *    - 否则，将该位置标记为陆地，岛屿数量加 1
     *    - 检查四个方向，如果相邻位置是陆地，则合并集合，岛屿数量相应减少
     * 3. 记录每次操作后的岛屿数量
     */
    public List<Integer> numIslands2(int m, int n, int[][] positions) {
        List<Integer> result = new ArrayList<>();
        if (m <= 0 || n <= 0 || positions == null || positions.length == 0) {
            return result;
        }

        UnionFind uf = new UnionFind(m, n);
        int[][] grid = new int[m][n];  // 0 表示水，1 表示陆地

        // 四个方向：上、右、下、左
        int[][] directions = {{-1, 0}, {0, 1}, {1, 0}, {0, -1}};

        for (int[] pos : positions) {
            int x = pos[0], y = pos[1];

            // 如果该位置已经是陆地，直接添加当前岛屿数量
            if (grid[x][y] == 1) {
                result.add(uf.getCount());
                continue;
            }

            // 标记为陆地
            grid[x][y] = 1;
            uf.addLand(x, y);  // 岛屿数量加 1
```

```
            // 检查四个方向，合并相邻的陆地
            for (int[] dir : directions) {
                int newX = x + dir[0];
                int newY = y + dir[1];

                // 检查新位置是否在网格内且是陆地
                if (newX >= 0 && newX < m && newY >= 0 && newY < n && grid[newX][newY] == 1) {
                    uf.union(x, y, newX, newY);
                }
            }

            result.add(uf.getCount());
        }

        return result;
    }

    /**
     * 并查集实现，专门用于处理网格岛屿问题
     */
    static class UnionFind {
        private int[] parent;    // 父节点数组
        private int[] rank;      // 秩数组，用于按秩合并
        private int count;        // 当前岛屿数量
        private int n;           // 网格列数

        public UnionFind(int m, int n) {
            this.n = n;
            int size = m * n;
            parent = new int[size];
            rank = new int[size];
            count = 0;

            // 初始化时，所有位置都指向-1（表示水）
            Arrays.fill(parent, -1);
        }

        /**
         * 将二维坐标转换为一维索引
         */
        private int getIndex(int x, int y) {
            return x * n + y;
        }
```

```
/**
 * 添加陆地
 */
public void addLand(int x, int y) {
    int index = getIndex(x, y);
    if (parent[index] == -1) {
        parent[index] = index;  // 指向自己
        rank[index] = 1;
        count++;
    }
}

/**
 * 查找操作，使用路径压缩优化
 */
public int find(int x, int y) {
    int index = getIndex(x, y);
    if (parent[index] == -1) {
        return -1;  // 水的位置
    }

    if (parent[index] != index) {
        parent[index] = find(parent[index] / n, parent[index] % n);
    }
    return parent[index];
}

/**
 * 合并两个陆地
 */
public void union(int x1, int y1, int x2, int y2) {
    int root1 = find(x1, y1);
    int root2 = find(x2, y2);

    // 如果有一个是水，或者已经在同一个集合中，则不需要合并
    if (root1 == -1 || root2 == -1 || root1 == root2) {
        return;
    }

    // 按秩合并
    if (rank[root1] > rank[root2]) {
        parent[root2] = root1;
```

```
        } else if (rank[root1] < rank[root2]) {
            parent[root1] = root2;
        } else {
            parent[root2] = root1;
            rank[root1]++;
        }
        count--;  // 合并后岛屿数量减 1
    }

    /**
     * 获取当前岛屿数量
     */
    public int getCount() {
        return count;
    }
}

/**
 * 方法 2: 使用哈希表存储陆地位置的优化解法
 * 时间复杂度: O(L * α(L))，其中 L 是 positions 的长度
 * 空间复杂度: O(L)
 */
public List<Integer> numIslands2Optimized(int m, int n, int[][] positions) {
    List<Integer> result = new ArrayList<>();
    if (m <= 0 || n <= 0 || positions == null || positions.length == 0) {
        return result;
    }

    // 使用哈希表存储陆地位置，避免使用 m*n 的数组
    Map<Integer, Integer> parent = new HashMap<>();  // 位置索引 -> 父节点索引
    Map<Integer, Integer> rank = new HashMap<>();    // 位置索引 -> 秩
    int count = 0;

    // 四个方向
    int[][] directions = {{-1, 0}, {0, 1}, {1, 0}, {0, -1}};

    for (int[] pos : positions) {
        int x = pos[0], y = pos[1];
        int index = x * n + y;

        // 如果该位置已经是陆地，直接添加当前岛屿数量
        if (parent.containsKey(index)) {
            result.add(count);
```

```
                continue;
            }

            // 添加新陆地
            parent.put(index, index);
            rank.put(index, 1);
            count++;

            // 检查四个方向
            for (int[] dir : directions) {
                int newX = x + dir[0];
                int newY = y + dir[1];
                int newIndex = newX * n + newY;

                // 检查新位置是否在网格内且是陆地
                if (newX >= 0 && newX < m && newY >= 0 && newY < n &&
parent.containsKey(newIndex)) {
                    // 合并集合
                    int root1 = find(index, parent);
                    int root2 = find(newIndex, parent);

                    if (root1 != root2) {
                        // 按秩合并
                        if (rank.get(root1) > rank.get(root2)) {
                            parent.put(root2, root1);
                        } else if (rank.get(root1) < rank.get(root2)) {
                            parent.put(root1, root2);
                        } else {
                            parent.put(root2, root1);
                            rank.put(root1, rank.get(root1) + 1);
                        }
                        count--;
                    }
                }
            }

            result.add(count);
        }

        return result;
    }

    /**
```

```
     * 查找操作（用于优化解法）
     */
    private int find(int x, Map<Integer, Integer> parent) {
        if (parent.get(x) != x) {
            parent.put(x, find(parent.get(x), parent));  // 路径压缩
        }
        return parent.get(x);
    }

    // 测试方法
    public static void main(String[] args) {
        Code16_LeetCode305 solution = new Code16_LeetCode305();

        // 测试用例 1
        int m1 = 3, n1 = 3;
        int[][] positions1 = {{0, 0}, {0, 1}, {1, 2}, {2, 1}};
        List<Integer> result1 = solution.numIslands2(m1, n1, positions1);
        System.out.println("测试用例 1 - 方法 1: " + result1); // 预期: [1, 1, 2, 3]

        List<Integer> result1Opt = solution.numIslands2Optimized(m1, n1, positions1);
        System.out.println("测试用例 1 - 方法 2: " + result1Opt); // 预期: [1, 1, 2, 3]

        // 测试用例 2
        int m2 = 1, n2 = 1;
        int[][] positions2 = {{0, 0}};
        List<Integer> result2 = solution.numIslands2(m2, n2, positions2);
        System.out.println("测试用例 2 - 方法 1: " + result2); // 预期: [1]

        List<Integer> result2Opt = solution.numIslands2Optimized(m2, n2, positions2);
        System.out.println("测试用例 2 - 方法 2: " + result2Opt); // 预期: [1]

        // 测试用例 3: 重复添加同一位置
        int m3 = 2, n3 = 2;
        int[][] positions3 = {{0, 0}, {0, 0}, {0, 1}};
        List<Integer> result3 = solution.numIslands2(m3, n3, positions3);
        System.out.println("测试用例 3 - 方法 1: " + result3); // 预期: [1, 1, 1]

        List<Integer> result3Opt = solution.numIslands2Optimized(m3, n3, positions3);
        System.out.println("测试用例 3 - 方法 2: " + result3Opt); // 预期: [1, 1, 1]

        // 测试用例 4: 形成一个大岛屿
        int m4 = 2, n4 = 2;
        int[][] positions4 = {{0, 0}, {0, 1}, {1, 0}, {1, 1}};
```

```
        List<Integer> result4 = solution.numIslands2(m4, n4, positions4);
        System.out.println("测试用例 4 - 方法 1: " + result4); // 预期: [1, 2, 2, 1]

        List<Integer> result4Opt = solution.numIslands2Optimized(m4, n4, positions4);
        System.out.println("测试用例 4 - 方法 2: " + result4Opt); // 预期: [1, 2, 2, 1]
    }
}
```

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

文件: Code16_LeetCode305.py

```
==========================================
from typing import List

class UnionFind:
    """
    并查集实现，专门用于处理网格岛屿问题
    """
    def __init__(self, m: int, n: int):
        self.n = n
        self.size = m * n
        self.parent = [-1] * self.size  # -1 表示水
        self.rank = [0] * self.size
        self.count = 0

    def get_index(self, x: int, y: int) -> int:
        """将二维坐标转换为一维索引"""
        return x * self.n + y

    def add_land(self, x: int, y: int) -> None:
        """添加陆地"""
        index = self.get_index(x, y)
        if self.parent[index] == -1:
            self.parent[index] = index  # 指向自己
            self.rank[index] = 1
            self.count += 1

    def find(self, x: int, y: int) -> int:
        """查找操作，使用路径压缩优化"""
        index = self.get_index(x, y)
        if self.parent[index] == -1:
            return -1  # 水的位置
```

```
        if self.parent[index] != index:
            # 递归查找并进行路径压缩
            root_x = self.parent[index] // self.n
            root_y = self.parent[index] % self.n
            self.parent[index] = self.find(root_x, root_y)
        return self.parent[index]

    def union(self, x1: int, y1: int, x2: int, y2: int) -> None:
        """合并两个陆地"""
        root1 = self.find(x1, y1)
        root2 = self.find(x2, y2)

        # 如果有一个是水，或者已经在同一个集合中，则不需要合并
        if root1 == -1 or root2 == -1 or root1 == root2:
            return

        # 按秩合并
        if self.rank[root1] > self.rank[root2]:
            self.parent[root2] = root1
        elif self.rank[root1] < self.rank[root2]:
            self.parent[root1] = root2
        else:
            self.parent[root2] = root1
            self.rank[root1] += 1
        self.count -= 1  # 合并后岛屿数量减1

    def get_count(self) -> int:
        """获取当前岛屿数量"""
        return self.count

class Solution:
    """
    LeetCode 305. 岛屿数量 II
    链接: https://leetcode.cn/problems/number-of-islands-ii/
    难度: 困难

    题目描述:
    给你一个大小为 m x n 的二进制网格 grid 。网格表示一个地图，其中，0 表示水，1 表示陆地。
    最初，网格中的单元格要么是水（0），要么是陆地（1）。
    你可以执行 addLand 操作，将位置 (row, col) 的水变成陆地。
    返回一个结果数组，其中每个结果[i] 表示在第 i 次操作后，地图中岛屿的数量。

    注意: 一个岛屿被水包围，并且通过水平或垂直方向上相邻的陆地连接而成。
```

你可以假设网格网格的四边均被水包围。

示例 1:
输入: m = 3, n = 3, positions = [[0,0],[0,1],[1,2],[2,1]]
输出: [1,1,2,3]

示例 2:
输入: m = 1, n = 1, positions = [[0,0]]
输出: [1]

约束条件:
1 <= m, n, positions.length <= 10^4
positions[i].length == 2
0 <= positions[i][0] < m
0 <= positions[i][1] < n
"""

```
def numIslands2(self, m: int, n: int, positions: List[List[int]]) -> List[int]:
    """
    使用并查集解决动态岛屿数量问题
    时间复杂度: O(L * α(m*n))，其中 L 是 positions 的长度
    空间复杂度: O(m*n)

    解题思路:
    1. 初始化并查集，大小为 m*n，初始时所有位置都是水（不属于任何集合）
    2. 对于每个添加陆地的操作:
       - 如果该位置已经是陆地，直接跳过
       - 否则，将该位置标记为陆地，岛屿数量加 1
       - 检查四个方向，如果相邻位置是陆地，则合并集合，岛屿数量相应减少
    3. 记录每次操作后的岛屿数量
    """
    if m <= 0 or n <= 0 or not positions:
        return []

    uf = UnionFind(m, n)
    grid = [[0] * n for _ in range(m)]  # 0 表示水，1 表示陆地

    # 四个方向：上、右、下、左
    directions = [(-1, 0), (0, 1), (1, 0), (0, -1)]
    result = []

    for pos in positions:
        x, y = pos
```

```
        # 如果该位置已经是陆地，直接添加当前岛屿数量
        if grid[x][y] == 1:
            result.append(uf.get_count())
            continue

        # 标记为陆地
        grid[x][y] = 1
        uf.add_land(x, y)  # 岛屿数量加 1

        # 检查四个方向，合并相邻的陆地
        for dx, dy in directions:
            new_x, new_y = x + dx, y + dy

            # 检查新位置是否在网格内且是陆地
            if 0 <= new_x < m and 0 <= new_y < n and grid[new_x][new_y] == 1:
                uf.union(x, y, new_x, new_y)

        result.append(uf.get_count())

    return result

def numIslands2Optimized(self, m: int, n: int, positions: List[List[int]]) -> List[int]:
    """
    方法 2: 使用哈希表存储陆地位置的优化解法
    时间复杂度: O(L * α(L))，其中 L 是 positions 的长度
    空间复杂度: O(L)
    """
    if m <= 0 or n <= 0 or not positions:
        return []

    # 使用哈希表存储陆地位置，避免使用 m*n 的数组
    parent = {}  # 位置索引 -> 父节点索引
    rank = {}    # 位置索引 -> 秩
    count = 0

    # 四个方向
    directions = [(-1, 0), (0, 1), (1, 0), (0, -1)]
    result = []

    def find(x: int) -> int:
        """查找操作，使用路径压缩优化"""
        if parent[x] != x:
```

```
            parent[x] = find(parent[x])  # 路径压缩
        return parent[x]

    for pos in positions:
        x, y = pos
        index = x * n + y

        # 如果该位置已经是陆地，直接添加当前岛屿数量
        if index in parent:
            result.append(count)
            continue

        # 添加新陆地
        parent[index] = index
        rank[index] = 1
        count += 1

        # 检查四个方向
        for dx, dy in directions:
            new_x, new_y = x + dx, y + dy
            new_index = new_x * n + new_y

            # 检查新位置是否在网格内且是陆地
            if 0 <= new_x < m and 0 <= new_y < n and new_index in parent:
                # 合并集合
                root1 = find(index)
                root2 = find(new_index)

                if root1 != root2:
                    # 按秩合并
                    if rank[root1] > rank[root2]:
                        parent[root2] = root1
                    elif rank[root1] < rank[root2]:
                        parent[root1] = root2
                    else:
                        parent[root2] = root1
                        rank[root1] += 1
                    count -= 1

        result.append(count)

    return result
```

```python
def test_solution():
    """测试函数"""
    solution = Solution()

    # 测试用例 1
    m1, n1 = 3, 3
    positions1 = [[0, 0], [0, 1], [1, 2], [2, 1]]
    result1 = solution.numIslands2(m1, n1, positions1)
    print(f"测试用例 1 - 方法 1: {result1}")  # 预期: [1, 1, 2, 3]

    result1_opt = solution.numIslands2Optimized(m1, n1, positions1)
    print(f"测试用例 1 - 方法 2: {result1_opt}")  # 预期: [1, 1, 2, 3]

    # 测试用例 2
    m2, n2 = 1, 1
    positions2 = [[0, 0]]
    result2 = solution.numIslands2(m2, n2, positions2)
    print(f"测试用例 2 - 方法 1: {result2}")  # 预期: [1]

    result2_opt = solution.numIslands2Optimized(m2, n2, positions2)
    print(f"测试用例 2 - 方法 2: {result2_opt}")  # 预期: [1]

    # 测试用例 3: 重复添加同一位置
    m3, n3 = 2, 2
    positions3 = [[0, 0], [0, 0], [0, 1]]
    result3 = solution.numIslands2(m3, n3, positions3)
    print(f"测试用例 3 - 方法 1: {result3}")  # 预期: [1, 1, 1]

    result3_opt = solution.numIslands2Optimized(m3, n3, positions3)
    print(f"测试用例 3 - 方法 2: {result3_opt}")  # 预期: [1, 1, 1]

    # 测试用例 4: 形成一个大岛屿
    m4, n4 = 2, 2
    positions4 = [[0, 0], [0, 1], [1, 0], [1, 1]]
    result4 = solution.numIslands2(m4, n4, positions4)
    print(f"测试用例 4 - 方法 1: {result4}")  # 预期: [1, 2, 2, 1]

    result4_opt = solution.numIslands2Optimized(m4, n4, positions4)
    print(f"测试用例 4 - 方法 2: {result4_opt}")  # 预期: [1, 2, 2, 1]

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

==========

文件: Code17_LeetCode399.java
==========
```
package class056;

import java.util.*;

/**
 * LeetCode 399. 除法求值
 * 链接: https://leetcode.cn/problems/evaluate-division/
 * 难度: 中等
 *
 * 题目描述:
 * 给你一个变量对数组 equations 和一个实数值数组 values 作为已知条件，其中 equations[i] = [Ai, Bi] 和 values[i] 共同表示等式 Ai / Bi = values[i]。
 * 每个 Ai 或 Bi 是一个表示单个变量的字符串。
 * 另有一些以数组 queries 表示的问题，其中 queries[j] = [Cj, Dj] 表示第 j 个问题，请你根据已知条件找出 Cj / Dj = ? 的结果作为答案。
 * 如果存在某个无法确定的答案，则用 -1.0 替代。
 *
 * 注意：输入总是有效的，且不存在循环或冲突的结果。
 *
 * 示例 1:
 * 输入: equations = [["a","b"],["b","c"]], values = [2.0,3.0], queries = [["a","c"],["b","a"],["a","e"],["a","a"],["x","x"]]
 * 输出: [6.00000,0.50000,-1.00000,1.00000,-1.00000]
 * 解释:
 * 条件: a / b = 2.0, b / c = 3.0
 * 问题: a / c = ?, b / a = ?, a / e = ?, a / a = ?, x / x = ?
 * 结果: [6.0, 0.5, -1.0, 1.0, -1.0]
 *
 * 示例 2:
 * 输入: equations = [["a","b"],["b","c"],["bc","cd"]], values = [1.5,2.5,5.0], queries = [["a","c"],["c","b"],["bc","cd"],["cd","bc"]]
 * 输出: [3.75000,0.40000,5.00000,0.20000]
 *
 * 示例 3:
 * 输入: equations = [["a","b"]], values = [0.5], queries = [["a","b"],["b","a"],["a","c"],["x","y"]]
 * 输出: [0.50000,2.00000,-1.00000,-1.00000]
 *
 * 约束条件:
```

```
 * 1 <= equations.length <= 20
 * equations[i].length == 2
 * 1 <= Ai.length, Bi.length <= 5
 * values.length == equations.length
 * 0.0 < values[i] <= 20.0
 * 1 <= queries.length <= 20
 * queries[i].length == 2
 * 1 <= Cj.length, Dj.length <= 5
 * Ai, Bi, Cj, Dj 由小写英文字母与数字组成
 */
public class Code17_LeetCode399 {

    /**
     * 方法 1: 使用带权并查集解决除法求值问题
     * 时间复杂度: O((E + Q) * α(N))，其中 E 是 equations 的长度，Q 是 queries 的长度，N 是不同变量的数量
     * 空间复杂度: O(N)
     *
     * 解题思路:
     * 1. 使用带权并查集维护变量之间的倍数关系
     * 2. 权重表示当前节点值 / 父节点值
     * 3. 对于查询，如果两个变量在同一个集合中，结果等于权重比值
     */
    public double[] calcEquation(List<List<String>> equations, double[] values, List<List<String>> queries) {
        // 创建并初始化带权并查集
        WeightedUnionFind uf = new WeightedUnionFind();

        // 构建并查集
        for (int i = 0; i < equations.size(); i++) {
            String var1 = equations.get(i).get(0);
            String var2 = equations.get(i).get(1);
            uf.union(var1, var2, values[i]);
        }

        // 处理查询
        double[] results = new double[queries.size()];
        for (int i = 0; i < queries.size(); i++) {
            String var1 = queries.get(i).get(0);
            String var2 = queries.get(i).get(1);

            // 如果变量不存在于并查集中，结果为-1.0
            if (!uf.contains(var1) || !uf.contains(var2)) {
```

```
            results[i] = -1.0;
            continue;
        }

        // 查找两个变量的根节点和权重
        Pair root1Info = uf.find(var1);
        Pair root2Info = uf.find(var2);

        // 如果根节点不同，说明无法确定结果
        if (!root1Info.root.equals(root2Info.root)) {
            results[i] = -1.0;
        } else {
            // 结果等于 var1 到根的权重除以 var2 到根的权重
            results[i] = root1Info.weight / root2Info.weight;
        }
    }

    return results;
}

/**
 * 辅助类，存储根节点和权重信息
 */
static class Pair {
    String root;
    double weight;

    Pair(String root, double weight) {
        this.root = root;
        this.weight = weight;
    }
}

/**
 * 带权并查集实现
 * 用于处理除法关系，维护变量之间的倍数关系
 */
static class WeightedUnionFind {
    // 存储父节点
    private Map<String, String> parent;
    // 存储到父节点的权重（当前节点值 / 父节点值）
    private Map<String, Double> weight;
```

```
public WeightedUnionFind() {
    parent = new HashMap<>();
    weight = new HashMap<>();
}

/**
 * 检查变量是否存在于并查集中
 */
public boolean contains(String x) {
    return parent.containsKey(x);
}

/**
 * 初始化变量
 */
private void ensureExists(String x) {
    if (!contains(x)) {
        parent.put(x, x);
        weight.put(x, 1.0);
    }
}

/**
 * 查找操作，返回根节点和权重（x 的值 / 根节点的值）
 */
public Pair find(String x) {
    ensureExists(x);

    if (!parent.get(x).equals(x)) {
        // 递归查找父节点
        Pair rootInfo = find(parent.get(x));
        String root = rootInfo.root;
        double rootWeight = rootInfo.weight;

        // 路径压缩：将 x 直接指向根节点
        parent.put(x, root);
        // 更新权重：x 到根的权重 = x 到父的权重 * 父到根的权重
        weight.put(x, weight.get(x) * rootWeight);
    }

    return new Pair(parent.get(x), weight.get(x));
}
```

```
        /**
         * 合并操作，表示 x / y = value
         */
        public void union(String x, String y, double value) {
            ensureExists(x);
            ensureExists(y);

            Pair xRootInfo = find(x);
            Pair yRootInfo = find(y);

            String xRoot = xRootInfo.root;
            String yRoot = yRootInfo.root;
            double xWeight = xRootInfo.weight;
            double yWeight = yRootInfo.weight;

            if (!xRoot.equals(yRoot)) {
                // 将 x 的根节点连接到 y 的根节点
                parent.put(xRoot, yRoot);
                // 更新权重：xRoot / yRoot = (y / yRoot) * (x / y) / (x / xRoot) = yWeight *
value / xWeight
                weight.put(xRoot, yWeight * value / xWeight);
            }
        }
    }

    /**
     * 方法 2: 使用图搜索（DFS/BFS）解决除法求值问题
     * 时间复杂度: O(E + Q * (E + V))，其中 E 是边数，V 是顶点数，Q 是查询数
     * 空间复杂度: O(E + V)
     *
     * 解题思路:
     * 1. 构建有向图，边权重表示除法关系
     * 2. 对于每个查询，使用 DFS/BFS 在图中搜索路径
     * 3. 路径上边权重的乘积就是结果
     */
    public double[] calcEquationDFS(List<List<String>> equations, double[] values,
List<List<String>> queries) {
        // 构建图
        Map<String, Map<String, Double>> graph = buildGraph(equations, values);

        double[] results = new double[queries.size()];
        for (int i = 0; i < queries.size(); i++) {
            String start = queries.get(i).get(0);
```

```
            String end = queries.get(i).get(1);
            results[i] = dfs(graph, start, end, new HashSet<>());
        }

        return results;
    }

    /**
     * 构建图
     */
    private Map<String, Map<String, Double>> buildGraph(List<List<String>> equations, double[]
values) {
        Map<String, Map<String, Double>> graph = new HashMap<>();

        for (int i = 0; i < equations.size(); i++) {
            String u = equations.get(i).get(0);
            String v = equations.get(i).get(1);
            double value = values[i];

            // 添加正向边 u -> v
            graph.computeIfAbsent(u, k -> new HashMap<>()).put(v, value);
            // 添加反向边 v -> u
            graph.computeIfAbsent(v, k -> new HashMap<>()).put(u, 1.0 / value);
        }

        return graph;
    }

    /**
     * DFS 搜索路径
     */
    private double dfs(Map<String, Map<String, Double>> graph, String start, String end,
Set<String> visited) {
        // 如果节点不存在
        if (!graph.containsKey(start) || !graph.containsKey(end)) {
            return -1.0;
        }

        // 如果找到目标节点
        if (start.equals(end)) {
            return 1.0;
        }
```

```
        visited.add(start);

        // 遍历邻居节点
        for (Map.Entry<String, Double> neighbor : graph.get(start).entrySet()) {
            String next = neighbor.getKey();
            double weight = neighbor.getValue();

            if (!visited.contains(next)) {
                double result = dfs(graph, next, end, visited);
                if (result != -1.0) {
                    return weight * result;
                }
            }
        }

        visited.remove(start);
        return -1.0;
    }

    /**
     * 方法 3: 使用 Floyd-Warshall 算法（动态规划）解决除法求值问题
     * 时间复杂度: O(V^3 + Q)，其中 V 是顶点数
     * 空间复杂度: O(V^2)
     *
     * 适用场景: 当查询数量很大时，预处理后可以快速回答查询
     */
    public double[] calcEquationFloydWarshall(List<List<String>> equations, double[] values,
List<List<String>> queries) {
        // 收集所有变量
        Set<String> variables = new HashSet<>();
        for (List<String> equation : equations) {
            variables.add(equation.get(0));
            variables.add(equation.get(1));
        }

        // 创建变量到索引的映射
        List<String> varList = new ArrayList<>(variables);
        int n = varList.size();
        Map<String, Integer> indexMap = new HashMap<>();
        for (int i = 0; i < n; i++) {
            indexMap.put(varList.get(i), i);
        }
```

```
// 初始化距离矩阵
double[][] dist = new double[n][n];
for (int i = 0; i < n; i++) {
    Arrays.fill(dist[i], -1.0);
    dist[i][i] = 1.0;  // 自己除自己等于1
}

// 填充已知关系
for (int i = 0; i < equations.size(); i++) {
    String u = equations.get(i).get(0);
    String v = equations.get(i).get(1);
    int uIndex = indexMap.get(u);
    int vIndex = indexMap.get(v);
    dist[uIndex][vIndex] = values[i];
    dist[vIndex][uIndex] = 1.0 / values[i];
}

// Floyd-Warshall算法
for (int k = 0; k < n; k++) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            if (dist[i][k] != -1.0 && dist[k][j] != -1.0) {
                dist[i][j] = dist[i][k] * dist[k][j];
            }
        }
    }
}

// 处理查询
double[] results = new double[queries.size()];
for (int i = 0; i < queries.size(); i++) {
    String u = queries.get(i).get(0);
    String v = queries.get(i).get(1);

    if (!indexMap.containsKey(u) || !indexMap.containsKey(v)) {
        results[i] = -1.0;
    } else {
        int uIndex = indexMap.get(u);
        int vIndex = indexMap.get(v);
        results[i] = dist[uIndex][vIndex];
    }
}
```

```
        return results;
    }

    // 测试方法
    public static void main(String[] args) {
        Code17_LeetCode399 solution = new Code17_LeetCode399();

        // 测试用例 1
        List<List<String>> equations1 = Arrays.asList(
            Arrays.asList("a", "b"),
            Arrays.asList("b", "c")
        );
        double[] values1 = {2.0, 3.0};
        List<List<String>> queries1 = Arrays.asList(
            Arrays.asList("a", "c"),
            Arrays.asList("b", "a"),
            Arrays.asList("a", "e"),
            Arrays.asList("a", "a"),
            Arrays.asList("x", "x")
        );

        double[] result1 = solution.calcEquation(equations1, values1, queries1);
        System.out.println("测试用例 1 - 并查集解法: " + Arrays.toString(result1));
        // 预期: [6.0, 0.5, -1.0, 1.0, -1.0]

        double[] result1DFS = solution.calcEquationDFS(equations1, values1, queries1);
        System.out.println("测试用例 1 - DFS 解法: " + Arrays.toString(result1DFS));
        // 预期: [6.0, 0.5, -1.0, 1.0, -1.0]

        double[] result1FW = solution.calcEquationFloydWarshall(equations1, values1, queries1);
        System.out.println("测试用例 1 - Floyd-Warshall 解法: " + Arrays.toString(result1FW));
        // 预期: [6.0, 0.5, -1.0, 1.0, -1.0]

        // 测试用例 2
        List<List<String>> equations2 = Arrays.asList(
            Arrays.asList("a", "b"),
            Arrays.asList("b", "c"),
            Arrays.asList("bc", "cd")
        );
        double[] values2 = {1.5, 2.5, 5.0};
        List<List<String>> queries2 = Arrays.asList(
            Arrays.asList("a", "c"),
            Arrays.asList("c", "b"),
```

```
            Arrays.asList("bc", "cd"),
            Arrays.asList("cd", "bc")
        );

        double[] result2 = solution.calcEquation(equations2, values2, queries2);
        System.out.println("测试用例 2 - 并查集解法: " + Arrays.toString(result2));
        // 预期: [3.75, 0.4, 5.0, 0.2]

        // 测试用例 3
        List<List<String>> equations3 = Arrays.asList(
            Arrays.asList("a", "b")
        );
        double[] values3 = {0.5};
        List<List<String>> queries3 = Arrays.asList(
            Arrays.asList("a", "b"),
            Arrays.asList("b", "a"),
            Arrays.asList("a", "c"),
            Arrays.asList("x", "y")
        );

        double[] result3 = solution.calcEquation(equations3, values3, queries3);
        System.out.println("测试用例 3 - 并查集解法: " + Arrays.toString(result3));
        // 预期: [0.5, 2.0, -1.0, -1.0]
    }
}
```

==========

文件: Code18_POJ1182.java

==========

```
package class056;

import java.util.*;

/**
 * POJ 1182 食物链
 * 链接: http://poj.org/problem?id=1182
 * 难度: 中等
 *
 * 题目描述:
 * 动物王国中有三类动物 A,B,C，这三类动物的食物链构成了有趣的环形。A 吃 B， B 吃 C，C 吃 A。
 * 现有 N 个动物，以 1－N 编号。每个动物都是 A,B,C 中的一种，但是我们并不知道它到底是哪一种。
 * 有人用两种说法对这 N 个动物所构成的食物链关系进行描述：
```

```
 * 第一种说法是"1 X Y"，表示 X 和 Y 是同类。
 * 第二种说法是"2 X Y"，表示 X 吃 Y。
 * 此人对 N 个动物，用上述两种说法，一句接一句地说出 K 句话，这 K 句话有的是真的，有的是假的。
 * 当一句话满足下列三条之一时，这句话就是假话，否则就是真话。
 * 1） 当前的话与前面的某些真的话冲突，就是假话；
 * 2） 当前的话中 X 或 Y 比 N 大，就是假话；
 * 3） 当前的话表示 X 吃 X，就是假话。
 * 你的任务是根据给定的 N（1 <= N <= 50,000）和 K 句话（0 <= K <= 100,000），输出假话的总数。
 *
 * 输入格式:
 * 第一行是两个整数 N 和 K，以一个空格分隔。
 * 以下 K 行每行是三个正整数 D，X，Y，两数之间用一个空格隔开，其中 D 表示说法的种类。
 * 若 D=1，则表示 X 和 Y 是同类。
 * 若 D=2，则表示 X 吃 Y。
 *
 * 输出格式:
 * 只有一个整数，表示假话的数目。
 *
 * 示例输入:
 * 100 7
 * 1 101 1
 * 2 1 2
 * 2 2 3
 * 2 3 3
 * 1 1 3
 * 2 3 1
 * 1 5 5
 *
 * 示例输出:
 * 3
 */
public class Code18_POJ1182 {

    /**
     * 使用带权并查集解决食物链问题
     * 时间复杂度: O(K * α(N))，其中 K 是语句数量，N 是动物数量
     * 空间复杂度: O(N)
     *
     * 解题思路:
     * 1. 使用带权并查集，每个元素存储到父节点的关系（0 表示同类，1 表示吃父节点，2 表示被父节点吃）
     * 2. 关系满足模 3 运算的传递性:
     *    - 如果 A 和 B 同类，B 和 C 同类，则 A 和 C 同类 (0+0=0 mod 3)
```

```
 *     - 如果A吃B，B吃C，则A被C吃 (1+1=2 mod 3)
 *     - 如果A吃B，B被C吃，则A和C同类 (1+2=0 mod 3)
 * 3. 对于每个语句，检查是否与已知关系冲突
 */
public int findInvalidStatements(int n, int[][] statements) {
    // 初始化并查集
    int[] parent = new int[n + 1]; // 动物编号从1开始
    int[] relation = new int[n + 1]; // relation[i]表示i到父节点的关系

    for (int i = 1; i <= n; i++) {
        parent[i] = i;
        relation[i] = 0; // 初始时每个节点的父节点是自己，关系为同类
    }

    int invalidCount = 0;

    for (int[] statement : statements) {
        int type = statement[0];
        int x = statement[1];
        int y = statement[2];

        // 检查条件2：X或Y比N大
        if (x > n || y > n) {
            invalidCount++;
            continue;
        }

        // 检查条件3：X吃X
        if (type == 2 && x == y) {
            invalidCount++;
            continue;
        }

        int rootX = find(x, parent, relation);
        int rootY = find(y, parent, relation);

        if (rootX == rootY) {
            // X和Y已经在同一集合中，检查是否冲突
            // 计算X到Y的关系：relation[x] - relation[y] mod 3
            int relationXToY = (relation[x] - relation[y] + 3) % 3;

            if (type == 1 && relationXToY != 0) {
                // 声明X和Y是同类，但实际不是
```

```
                invalidCount++;
            } else if (type == 2 && relationXToY != 1) {
                // 声明 X 吃 Y，但实际不是
                invalidCount++;
            }
        } else {
            // 合并两个集合
            // 将 rootY 合并到 rootX
            parent[rootY] = rootX;

            // 计算 rootY 到 rootX 的关系
            // 期望关系: X 和 Y 的关系为 type-1 (0 表示同类，1 表示吃)
            // 根据关系传递性: relation[x] + relation[rootY] ≡ (type-1) + relation[y] (mod
3)
            // 所以: relation[rootY] ≡ (type-1) + relation[y] - relation[x] (mod 3)
            relation[rootY] = (type - 1 + relation[y] - relation[x] + 3) % 3;
        }
    }

    return invalidCount;
}

/**
 * 查找根节点并进行路径压缩，同时更新关系
 */
private int find(int x, int[] parent, int[] relation) {
    if (parent[x] != x) {
        int originalParent = parent[x];
        parent[x] = find(parent[x], parent, relation);
        // 更新关系：x 到新根节点的关系 = x 到原父节点的关系 + 原父节点到新根节点的关系
        relation[x] = (relation[x] + relation[originalParent]) % 3;
    }
    return parent[x];
}

/**
 * 方法 2: 使用扩展域并查集（三倍空间法）解决食物链问题
 * 时间复杂度: O(K * α(3N)) ≈ O(K * α(N))
 * 空间复杂度: O(3N)
 *
 * 解题思路:
 * 1. 将每个动物拆分成三个域：A 域、B 域、C 域
 * 2. 如果 X 和 Y 是同类，则合并(X_A, Y_A), (X_B, Y_B), (X_C, Y_C)
```

```
     * 3. 如果X吃Y，则合并(X_A, Y_B), (X_B, Y_C), (X_C, Y_A)
     * 4. 检查冲突: 如果声明X和Y同类，但X_A和Y_B或Y_C在同一集合，则冲突
     *             如果声明X吃Y，但X_A和Y_A或Y_C在同一集合，则冲突
     */
    public int findInvalidStatementsExtendedDomain(int n, int[][] statements) {
        // 扩展域并查集，大小为3n
        int[] parent = new int[3 * n + 1];

        for (int i = 1; i <= 3 * n; i++) {
            parent[i] = i;
        }

        int invalidCount = 0;

        for (int[] statement : statements) {
            int type = statement[0];
            int x = statement[1];
            int y = statement[2];

            // 检查条件2：X或Y比N大
            if (x > n || y > n) {
                invalidCount++;
                continue;
            }

            // 检查条件3：X吃X
            if (type == 2 && x == y) {
                invalidCount++;
                continue;
            }

            if (type == 1) {
                // 声明X和Y是同类
                if (find(x, parent) == find(y + n, parent) || find(x, parent) == find(y + 2 * n,
parent)) {
                    // 如果X和Y_B或Y_C在同一集合，说明X吃Y或被Y吃，冲突
                    invalidCount++;
                } else {
                    // 合并三个域
                    union(x, y, parent);
                    union(x + n, y + n, parent);
                    union(x + 2 * n, y + 2 * n, parent);
                }
```

```
            } else {
                // 声明 X 吃 Y
                if (find(x, parent) == find(y, parent) || find(x, parent) == find(y + 2 * n,
parent)) {
                    // 如果 X 和 Y 是同类，或者 Y 吃 X，冲突
                    invalidCount++;
                } else {
                    // 合并对应域
                    union(x, y + n, parent);         // X_A 和 Y_B 合并
                    union(x + n, y + 2 * n, parent); // X_B 和 Y_C 合并
                    union(x + 2 * n, y, parent);     // X_C 和 Y_A 合并
                }
            }
        }

        return invalidCount;
    }

    /**
     * 标准并查集查找操作
     */
    private int find(int x, int[] parent) {
        if (parent[x] != x) {
            parent[x] = find(parent[x], parent);
        }
        return parent[x];
    }

    /**
     * 标准并查集合并操作
     */
    private void union(int x, int y, int[] parent) {
        int rootX = find(x, parent);
        int rootY = find(y, parent);
        if (rootX != rootY) {
            parent[rootY] = rootX;
        }
    }

    /**
     * 方法 3: 使用更清晰的带权并查集实现
     */
    static class FoodChainUnionFind {
```

```
private int[] parent;
private int[] relation; // 0: 同类, 1: 吃父节点, 2: 被父节点吃
private int invalidCount;

public FoodChainUnionFind(int n) {
    parent = new int[n + 1];
    relation = new int[n + 1];
    invalidCount = 0;

    for (int i = 1; i <= n; i++) {
        parent[i] = i;
        relation[i] = 0;
    }
}

public int processStatements(int[][] statements) {
    for (int[] statement : statements) {
        int type = statement[0];
        int x = statement[1];
        int y = statement[2];

        processStatement(type, x, y);
    }
    return invalidCount;
}

private void processStatement(int type, int x, int y) {
    // 边界条件检查
    if (x > parent.length - 1 || y > parent.length - 1) {
        invalidCount++;
        return;
    }

    if (type == 2 && x == y) {
        invalidCount++;
        return;
    }

    int rootX = find(x);
    int rootY = find(y);

    if (rootX == rootY) {
        // 已经在同一集合，检查关系是否冲突
```

```
            int relationXY = (relation[x] - relation[y] + 3) % 3;

            if (type == 1 && relationXY != 0) {
                invalidCount++;
            } else if (type == 2 && relationXY != 1) {
                invalidCount++;
            }
        } else {
            // 合并集合
            parent[rootY] = rootX;
            relation[rootY] = (type - 1 + relation[y] - relation[x] + 3) % 3;
        }
    }

    private int find(int x) {
        if (parent[x] != x) {
            int originalParent = parent[x];
            parent[x] = find(parent[x]);
            relation[x] = (relation[x] + relation[originalParent]) % 3;
        }
        return parent[x];
    }
}

// 测试方法
public static void main(String[] args) {
    Code18_POJ1182 solution = new Code18_POJ1182();

    // 测试用例 1: 示例输入
    int n1 = 100;
    int[][] statements1 = {
        {1, 101, 1},  // 假话：101 > 100
        {2, 1, 2},     // 真话
        {2, 2, 3},     // 真话
        {2, 3, 3},     // 假话：3 吃 3
        {1, 1, 3},     // 假话：根据前面，1 吃 2，2 吃 3，所以 1 被 3 吃，不是同类
        {2, 3, 1},     // 真话：3 吃 1（符合环形关系）
        {1, 5, 5}      // 真话：5 和 5 是同类
    };

    int result1 = solution.findInvalidStatements(n1, statements1);
    System.out.println("测试用例 1 - 带权并查集: " + result1); // 预期: 3
```

```
    int result1Extended = solution.findInvalidStatementsExtendedDomain(n1, statements1);
    System.out.println("测试用例 1 - 扩展域: " + result1Extended); // 预期: 3

    FoodChainUnionFind uf1 = new FoodChainUnionFind(n1);
    int result1Class = uf1.processStatements(statements1);
    System.out.println("测试用例 1 - 类封装: " + result1Class); // 预期: 3

    // 测试用例 2: 简单测试
    int n2 = 5;
    int[][] statements2 = {
        {1, 1, 2},     // 真话：1 和 2 是同类
        {2, 2, 3},     // 真话：2 吃 3
        {2, 3, 4},     // 真话：3 吃 4
        {1, 1, 4}      // 假话：1 和 4 应该是吃的关系（1->2->3->4，所以 1 被 4 吃）
    };

    int result2 = solution.findInvalidStatements(n2, statements2);
    System.out.println("测试用例 2 - 带权并查集: " + result2); // 预期: 1

    // 测试用例 3: 环形关系测试
    int n3 = 3;
    int[][] statements3 = {
        {2, 1, 2},     // 真话：1 吃 2
        {2, 2, 3},     // 真话：2 吃 3
        {2, 3, 1}      // 真话：3 吃 1（形成环形）
    };

    int result3 = solution.findInvalidStatements(n3, statements3);
    System.out.println("测试用例 3 - 带权并查集: " + result3); // 预期: 0
}

/**
 * 食物链问题总结:
 *
 * 1. 带权并查集解法（推荐）:
 *    - 优点: 空间效率高，只需要 O(N)空间
 *    - 缺点: 关系计算需要模运算，理解稍复杂
 *    - 适用: 大多数场景
 *
 * 2. 扩展域并查集解法:
 *    - 优点: 逻辑清晰，易于理解
 *    - 缺点: 需要 3 倍空间，空间效率较低
 *    - 适用: 当关系种类较多时（如更多种类的关系）
```

```
     *
     * 3. 关键技巧:
     *    - 关系传递性: 使用模运算处理环形关系
     *    - 路径压缩: 同时更新关系值
     *    - 边界处理: 注意动物编号范围和自相矛盾的情况
     *
     * 4. 应用扩展:
     *    - 可以扩展到更多种类的关系（如 4 种、5 种动物）
     *    - 可以处理更复杂的关系网络
     */
}

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

文件: Code19_UnionFindComprehensiveSummary.java
==========================================
package class056;

import java.util.*;

/**
 * 并查集综合总结与高级应用
 * 本文件包含并查集的各种高级应用和优化技巧总结
 */
public class Code19_UnionFindComprehensiveSummary {

    /**
     * 1. 标准并查集模板（路径压缩 + 按秩合并）
     */
    static class StandardUnionFind {
        private int[] parent;
        private int[] rank;
        private int count;

        public StandardUnionFind(int n) {
            parent = new int[n];
            rank = new int[n];
            count = n;

            for (int i = 0; i < n; i++) {
                parent[i] = i;
                rank[i] = 1;
            }
```

```
    }

    public int find(int x) {
        if (parent[x] != x) {
            parent[x] = find(parent[x]);  // 路径压缩
        }
        return parent[x];
    }

    public void union(int x, int y) {
        int rootX = find(x);
        int rootY = find(y);

        if (rootX != rootY) {
            // 按秩合并
            if (rank[rootX] > rank[rootY]) {
                parent[rootY] = rootX;
            } else if (rank[rootX] < rank[rootY]) {
                parent[rootX] = rootY;
            } else {
                parent[rootY] = rootX;
                rank[rootX]++;
            }
            count--;
        }
    }

    public boolean isConnected(int x, int y) {
        return find(x) == find(y);
    }

    public int getCount() {
        return count;
    }
}

/**
 * 2. 带权并查集模板
 * 用于维护元素之间的关系（如距离、倍数等）
 */
static class WeightedUnionFind {
    private int[] parent;
    private int[] weight;  // 存储到父节点的权重
```

```
public WeightedUnionFind(int n) {
    parent = new int[n];
    weight = new int[n];

    for (int i = 0; i < n; i++) {
        parent[i] = i;
        weight[i] = 0;  // 初始权重为0
    }
}

public int find(int x) {
    if (parent[x] != x) {
        int root = find(parent[x]);
        weight[x] += weight[parent[x]];  // 更新权重
        parent[x] = root;
    }
    return parent[x];
}

/**
 * 合并操作，表示x到y的权重为value
 */
public void union(int x, int y, int value) {
    int rootX = find(x);
    int rootY = find(y);

    if (rootX != rootY) {
        parent[rootX] = rootY;
        // 更新权重: weight[x] + weight[rootX] = value + weight[y]
        weight[rootX] = value + weight[y] - weight[x];
    }
}

/**
 * 获取x到y的权重差
 */
public int getWeight(int x, int y) {
    if (find(x) != find(y)) {
        return -1;  // 不在同一集合
    }
    return weight[x] - weight[y];
}
```

```
}

/**
 * 3. 可撤销并查集
 * 支持撤销操作，用于需要回溯的场景
 */
static class ReversibleUnionFind {
    private int[] parent;
    private int[] rank;
    private Stack<int[]> history;  // 存储操作历史

    public ReversibleUnionFind(int n) {
        parent = new int[n];
        rank = new int[n];
        history = new Stack<>();

        for (int i = 0; i < n; i++) {
            parent[i] = i;
            rank[i] = 1;
        }
    }

    public int find(int x) {
        if (parent[x] != x) {
            return find(parent[x]);  // 注意：可撤销并查集通常不进行路径压缩
        }
        return parent[x];
    }

    public void union(int x, int y) {
        int rootX = find(x);
        int rootY = find(y);

        if (rootX != rootY) {
            // 记录操作历史
            if (rank[rootX] > rank[rootY]) {
                parent[rootY] = rootX;
                history.push(new int[]{rootY, rootX, rank[rootX]});
            } else if (rank[rootX] < rank[rootY]) {
                parent[rootX] = rootY;
                history.push(new int[]{rootX, rootY, rank[rootY]});
            } else {
                parent[rootY] = rootX;
```

```
                rank[rootX]++;
                history.push(new int[]{rootY, rootX, rank[rootX] - 1});
            }
        }
    }

    /**
     * 撤销上一次合并操作
     */
    public void undo() {
        if (!history.isEmpty()) {
            int[] operation = history.pop();
            int child = operation[0];
            int parentNode = operation[1];
            int originalRank = operation[2];

            parent[child] = child;
            rank[parentNode] = originalRank;
        }
    }

    public int getHistorySize() {
        return history.size();
    }
}

/**
 * 4. 离线查询处理模板
 * 用于需要按特定顺序处理查询的场景
 */
static class OfflineQueryProcessor {
    /**
     * 处理边权限制的路径存在性查询（LeetCode 1697）
     */
    public boolean[] distanceLimitedPathsExist(int n, int[][] edges, int[][] queries) {
        // 将查询按限制排序，并记录原始索引
        int[][] sortedQueries = new int[queries.length][4];
        for (int i = 0; i < queries.length; i++) {
            sortedQueries[i][0] = queries[i][0];
            sortedQueries[i][1] = queries[i][1];
            sortedQueries[i][2] = queries[i][2];
            sortedQueries[i][3] = i;
        }
```

```
        Arrays.sort(sortedQueries, (a, b) -> Integer.compare(a[2], b[2]));

        // 将边按权重排序
        Arrays.sort(edges, (a, b) -> Integer.compare(a[2], b[2]));

        // 初始化并查集
        StandardUnionFind uf = new StandardUnionFind(n);
        boolean[] results = new boolean[queries.length];
        int edgeIndex = 0;

        // 处理查询
        for (int[] query : sortedQueries) {
            int limit = query[2];
            int originalIndex = query[3];

            // 添加所有权重小于 limit 的边
            while (edgeIndex < edges.length && edges[edgeIndex][2] < limit) {
                uf.union(edges[edgeIndex][0], edges[edgeIndex][1]);
                edgeIndex++;
            }

            // 检查连通性
            results[originalIndex] = uf.isConnected(query[0], query[1]);
        }

        return results;
    }
}

/**
 * 5. 动态连通性处理（支持删除操作）
 */
static class DynamicConnectivity {
    /**
     * 逆向并查集：从最终状态开始，逆向添加元素
     * 适用于需要处理删除操作的场景
     */
    public int[] processDeletions(int n, int[][] edges, int[] deletions) {
        // 标记被删除的边
        Set<String> deletedEdges = new HashSet<>();
        for (int edgeIndex : deletions) {
            int[] edge = edges[edgeIndex];
            String key = edge[0] + "," + edge[1];
```

```
            deletedEdges.add(key);
        }

        // 初始化并查集，只添加未被删除的边
        StandardUnionFind uf = new StandardUnionFind(n);
        for (int i = 0; i < edges.length; i++) {
            int[] edge = edges[i];
            String key = edge[0] + "," + edge[1];
            if (!deletedEdges.contains(key)) {
                uf.union(edge[0], edge[1]);
            }
        }

        // 逆向处理删除操作（从后往前添加边）
        int[] results = new int[deletions.length];
        for (int i = deletions.length - 1; i >= 0; i--) {
            results[i] = uf.getCount();

            // 添加被删除的边
            int[] edge = edges[deletions[i]];
            uf.union(edge[0], edge[1]);
        }

        return results;
    }
}

/**
 * 6. 并查集在网格问题中的应用
 */
static class GridUnionFind {
    /**
     * 处理二维网格的连通性问题
     */
    public int numIslands(char[][] grid) {
        if (grid == null || grid.length == 0) {
            return 0;
        }

        int m = grid.length;
        int n = grid[0].length;
        int waterCount = 0;
```

```
            // 初始化并查集
            StandardUnionFind uf = new StandardUnionFind(m * n);

            // 四个方向
            int[][] directions = {{-1, 0}, {0, 1}, {1, 0}, {0, -1}};

            for (int i = 0; i < m; i++) {
                for (int j = 0; j < n; j++) {
                    if (grid[i][j] == '1') {
                        // 当前是陆地，检查四个方向
                        for (int[] dir : directions) {
                            int newI = i + dir[0];
                            int newJ = j + dir[1];

                            if (newI >= 0 && newI < m && newJ >= 0 && newJ < n &&
grid[newI][newJ] == '1') {
                                uf.union(i * n + j, newI * n + newJ);
                            }
                        }
                    } else {
                        waterCount++;
                    }
                }
            }

            // 岛屿数量 = 连通分量数量 - 水的数量（需要调整）
            return uf.getCount() - waterCount;
        }
    }

    /**
     * 7. 并查集性能优化技巧总结
     */
    static class OptimizationTips {
        /**
         * 优化技巧 1: 小挂大优化（按秩合并）
         * - 总是将小树合并到大树下
         * - 保持树的平衡，避免退化成链表
         */

        /**
         * 优化技巧 2: 路径压缩
         * - 在查找操作时将路径上的节点直接连接到根节点
```

```
     * - 使树更加扁平化，提高后续查找效率
     */

    /**
     * 优化技巧 3: 按大小合并 vs 按秩合并
     * - 按大小合并: 更简单，但可能不够平衡
     * - 按秩合并: 更平衡，但需要维护秩信息
     * - 实际应用中差异不大，都可以达到 O(α(n))复杂度
     */

    /**
     * 优化技巧 4: 避免不必要的查找操作
     * - 在合并前先检查是否已经在同一集合
     * - 对于频繁的连通性查询，可以缓存结果
     */

    /**
     * 优化技巧 5: 使用数组代替哈希表
     * - 当元素是连续整数时，使用数组更高效
     * - 哈希表适用于元素是字符串或其他类型的场景
     */
}

/**
 * 8. 并查集常见问题模式总结
 */
static class ProblemPatterns {
    /**
     * 模式 1: 连通分量计数
     * - 问题特征: 需要统计连通区域的数量
     * - 例题: 岛屿数量、朋友圈数量
     * - 解法: 初始化 n 个集合，每合并一次计数减 1
     */

    /**
     * 模式 2: 环检测
     * - 问题特征: 判断图中是否存在环
     * - 例题: 冗余连接
     * - 解法: 遍历边，如果两个端点已经在同一集合，则检测到环
     */

    /**
     * 模式 3: 动态连通性
```

```
     * - 问题特征：需要处理元素的添加或删除
     * - 例题：岛屿数量 II、打砖块
     * - 解法：逆向处理、可撤销并查集
     */

    /**
     * 模式 4：关系维护
     * - 问题特征：需要维护元素之间的关系
     * - 例题：食物链、除法求值
     * - 解法：带权并查集
     */

    /**
     * 模式 5：离线查询
     * - 问题特征：查询可以按某种顺序处理
     * - 例题：边权限制的路径存在性
     * - 解法：对查询排序，逐步添加元素
     */

    /**
     * 模式 6：最小生成树
     * - 问题特征：寻找连接所有点的最小代价
     * - 例题：Kruskal 算法
     * - 解法：对边排序，使用并查集维护连通性
     */
}

/**
 * 9. 并查集在工程中的应用
 */
static class EngineeringApplications {
    /**
     * 应用 1：网络连接管理
     * - 场景：服务器集群的连通性管理
     * - 需求：快速判断两台服务器是否连通
     * - 优势：高效的连通性查询
     */

    /**
     * 应用 2：社交网络分析
     * - 场景：好友关系、社区发现
     * - 需求：找到连通的朋友圈
     * - 优势：高效的集合合并操作
```

```
     */

    /**
     * 应用 3: 图像处理
     * - 场景: 连通区域标记
     * - 需求: 找到图像中的连通区域
     * - 优势: 处理二维网格的高效性
     */

    /**
     * 应用 4: 编译器优化
     * - 场景: 变量别名分析
     * - 需求: 判断两个变量是否指向同一内存
     * - 优势: 高效的关系维护
     */

    /**
     * 应用 5: 数据库查询优化
     * - 场景: 等价类查询
     * - 需求: 快速判断两个元素是否等价
     * - 优势: 近似常数时间的查询
     */
}

/**
 * 10. 并查集的局限性及替代方案
 */
static class LimitationsAndAlternatives {
    /**
     * 局限性 1: 不支持分割操作
     * - 问题: 一旦合并，不能直接分割
     * - 替代: 可撤销并查集、动态树（Link-Cut Tree）
     */

    /**
     * 局限性 2: 只能维护等价关系
     * - 问题: 关系必须满足自反性、对称性、传递性
     * - 替代: 带权并查集可以处理更多关系
     */

    /**
     * 局限性 3: 内存使用
     * - 问题: 需要 O(n)的额外空间
```

```
     * - 替代: 对于稀疏图，可以使用其他数据结构
     */

    /**
     * 局限性 4: 并行化困难
     * - 问题: 路径压缩和按秩合并难以并行化
     * - 替代: 使用其他并发数据结构
     */
}

// 测试方法
public static void main(String[] args) {
    System.out.println("并查集综合总结与高级应用");
    System.out.println("========================");

    // 测试标准并查集
    StandardUnionFind uf1 = new StandardUnionFind(5);
    uf1.union(0, 1);
    uf1.union(2, 3);
    uf1.union(1, 2);
    System.out.println("标准并查集测试: " + uf1.getCount()); // 预期: 2

    // 测试带权并查集
    WeightedUnionFind uf2 = new WeightedUnionFind(5);
    uf2.union(0, 1, 1);
    uf2.union(1, 2, 2);
    System.out.println("带权并查集测试: " + uf2.getWeight(0, 2)); // 预期: 3

    // 测试可撤销并查集
    ReversibleUnionFind uf3 = new ReversibleUnionFind(5);
    uf3.union(0, 1);
    uf3.union(2, 3);
    uf3.undo(); // 撤销最后一次合并
    System.out.println("可撤销并查集测试: " + uf3.getHistorySize()); // 预期: 1

    System.out.println("并查集学习路径建议:");
    System.out.println("1. 掌握标准并查集模板（路径压缩+按秩合并）");
    System.out.println("2. 练习基础题目（连通分量、环检测）");
    System.out.println("3. 学习带权并查集（关系维护）");
    System.out.println("4. 掌握高级应用（离线查询、动态连通性）");
    System.out.println("5. 理解工程应用和优化技巧");
}
```

```
    /**
     * 学习资源推荐:
     * 1. 《算法导论》第21章：用于不相交集合的数据结构
     * 2. LeetCode 并查集专题
     * 3. 各大OJ平台的并查集题目
     * 4. 学术论文关于并查集优化的研究
     *
     * 关键要点:
     * - 理解并查集的核心思想：代表元素和路径压缩
     * - 掌握时间复杂度分析：O(α(n))的由来
     * - 熟练应用各种变种：带权、可撤销、离线等
     * - 注意边界条件和异常处理
     * - 在实际工程中选择合适的实现方式
     */
}

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

文件: Code20_LeetCode1697.java
==========================================
package class056;

import java.util.*;

/**
 * LeetCode 1697. 检查边长度限制的路径是否存在
 * 链接: https://leetcode.cn/problems/checking-existence-of-edge-length-limited-paths/
 * 难度: 困难
 *
 * 题目描述:
 * 给你一个 n 个节点的无向图，节点编号为 0 到 n-1。图中由 edges 数组表示边，其中 edges[i] = [ui, vi, disi] 表示节点 ui 和 vi 之间有一条长度为 disi 的无向边。
 * 同时给你一个数组 queries ，其中 queries[j] = [pj, qj, limitj] ，你的任务是对于每个查询 queries[j] ，判断是否存在从 pj 到 qj 的路径，且路径上每条边的长度都严格小于 limitj 。
 * 注意：查询之间是独立的，即每个查询都是针对原始图进行的。
 *
 * 示例 1:
 * 输入: n = 3, edges = [[0,1,2],[1,2,4],[2,0,8],[2,0,1]], queries = [[0,1,2],[0,2,5]]
 * 输出: [false,true]
 * 解释:
 * 查询 1: 路径 0-1 的边长度为 2，不小于 2，所以不存在满足条件的路径。
 * 查询 2: 路径 0-2-0-1 的边长度为 1+8+2=11，但边长度都小于 5？实际上应该选择路径 0-2 的边长度为 1（因为[2,0,1]这条边存在）。
```

```
 *
 * 示例 2:
 * 输入: n = 5, edges = [[0,1,10],[1,2,5],[2,3,9],[3,4,13]], queries = [[0,4,14],[1,4,13]]
 * 输出: [true,false]
 * 解释:
 * 查询 1: 路径 0-1-2-3-4 的边长度都小于 14，所以存在。
 * 查询 2: 路径中必须包含长度为 13 的边，但 13 不小于 13，所以不存在。
 *
 * 约束条件:
 * 2 <= n <= 10^5
 * 1 <= edges.length <= 10^5
 * 1 <= queries.length <= 10^5
 * edges[i].length == 3
 * queries[i].length == 3
 * 0 <= ui, vi, pj, qj <= n - 1
 * ui != vi
 * pj != qj
 * 1 <= disi, limitj <= 10^9
 */
public class Code20_LeetCode1697 {

    /**
     * 方法 1: 离线查询 + 并查集（推荐解法）
     * 时间复杂度: O((E + Q) * α(N))，其中 E 是边数，Q 是查询数，N 是节点数
     * 空间复杂度: O(N + E + Q)
     *
     * 解题思路:
     * 1. 将查询按 limit 从小到大排序（离线处理）
     * 2. 将边按权重从小到大排序
     * 3. 使用并查集维护连通性
     * 4. 对于每个查询，添加所有权重小于 limit 的边
     * 5. 检查两个节点是否连通
     */
    public boolean[] distanceLimitedPathsExist(int n, int[][] edges, int[][] queries) {
        int m = queries.length;

        // 为查询添加索引，以便恢复原始顺序
        int[][] indexedQueries = new int[m][4];
        for (int i = 0; i < m; i++) {
            indexedQueries[i][0] = queries[i][0];
            indexedQueries[i][1] = queries[i][1];
            indexedQueries[i][2] = queries[i][2];
            indexedQueries[i][3] = i; // 原始索引
```

```
        }

        // 按 limit 对查询排序
        Arrays.sort(indexedQueries, (a, b) -> Integer.compare(a[2], b[2]));

        // 按权重对边排序
        Arrays.sort(edges, (a, b) -> Integer.compare(a[2], b[2]));

        // 初始化并查集
        UnionFind uf = new UnionFind(n);
        boolean[] result = new boolean[m];

        int edgeIndex = 0;
        // 处理每个查询
        for (int[] query : indexedQueries) {
            int p = query[0];
            int q = query[1];
            int limit = query[2];
            int originalIndex = query[3];

            // 添加所有权重小于 limit 的边
            while (edgeIndex < edges.length && edges[edgeIndex][2] < limit) {
                uf.union(edges[edgeIndex][0], edges[edgeIndex][1]);
                edgeIndex++;
            }

            // 检查连通性
            result[originalIndex] = uf.isConnected(p, q);
        }

        return result;
    }

    /**
     * 标准并查集实现（路径压缩 + 按秩合并）
     */
    static class UnionFind {
        private int[] parent;
        private int[] rank;

        public UnionFind(int n) {
            parent = new int[n];
            rank = new int[n];
```

```
        for (int i = 0; i < n; i++) {
            parent[i] = i;
            rank[i] = 1;
        }
    }

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

    public void union(int x, int y) {
        int rootX = find(x);
        int rootY = find(y);

        if (rootX != rootY) {
            if (rank[rootX] > rank[rootY]) {
                parent[rootY] = rootX;
            } else if (rank[rootX] < rank[rootY]) {
                parent[rootX] = rootY;
            } else {
                parent[rootY] = rootX;
                rank[rootX]++;
            }
        }
    }

    public boolean isConnected(int x, int y) {
        return find(x) == find(y);
    }
}

/**
 * 方法 2: 在线查询 - 使用最小生成树 + LCA（适用于需要支持在线查询的场景）
 * 时间复杂度: O((E log E) + Q log N) 用于预处理，每次查询 O(log N)
 * 空间复杂度: O(N + E)
 *
 * 解题思路:
 * 1. 构建图的最小生成树（Kruskal 算法）
 * 2. 在最小生成树上预处理 LCA（最近公共祖先）
 * 3. 对于每个查询，找到路径上的最大边权
```

```
     * 4. 检查最大边权是否小于 limit
     */
    public boolean[] distanceLimitedPathsExistLCA(int n, int[][] edges, int[][] queries) {
        // 构建最小生成树
        List<int[]>[] mst = buildMST(n, edges);

        // 预处理 LCA
        LCASolver lcaSolver = new LCASolver(n, mst);

        boolean[] result = new boolean[queries.length];
        for (int i = 0; i < queries.length; i++) {
            int u = queries[i][0];
            int v = queries[i][1];
            int limit = queries[i][2];

            // 找到 u 到 v 路径上的最大边权
            int maxWeight = lcaSolver.queryMaxWeight(u, v);
            result[i] = maxWeight < limit;
        }

        return result;
    }

    /**
     * 使用 Kruskal 算法构建最小生成树
     */
    private List<int[]>[] buildMST(int n, int[][] edges) {
        // 按权重排序边
        Arrays.sort(edges, (a, b) -> Integer.compare(a[2], b[2]));

        UnionFind uf = new UnionFind(n);
        @SuppressWarnings("unchecked")
        List<int[]>[] graph = new ArrayList[n];
        for (int i = 0; i < n; i++) {
            graph[i] = new ArrayList<>();
        }

        for (int[] edge : edges) {
            int u = edge[0];
            int v = edge[1];
            int weight = edge[2];

            if (!uf.isConnected(u, v)) {
```

```
            uf.union(u, v);
            // 添加无向边
            graph[u].add(new int[]{v, weight});
            graph[v].add(new int[]{u, weight});
        }
    }

    return graph;
}

/**
 * LCA 求解器，用于在树上查询路径最大边权
 */
static class LCASolver {
    private int n;
    private int LOG;
    private int[] depth;
    private int[][] parent;
    private int[][] maxWeight;

    public LCASolver(int n, List<int[]>[] tree) {
        this.n = n;
        this.LOG = (int) (Math.log(n) / Math.log(2)) + 1;
        this.depth = new int[n];
        this.parent = new int[n][LOG];
        this.maxWeight = new int[n][LOG];

        // 初始化
        for (int i = 0; i < n; i++) {
            Arrays.fill(parent[i], -1);
            Arrays.fill(maxWeight[i], 0);
        }

        // BFS 预处理
        boolean[] visited = new boolean[n];
        Queue<Integer> queue = new LinkedList<>();
        queue.offer(0);
        visited[0] = true;
        depth[0] = 0;

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

```
            for (int[] edge : tree[u]) {
                int v = edge[0];
                int weight = edge[1];

                if (!visited[v]) {
                    visited[v] = true;
                    depth[v] = depth[u] + 1;
                    parent[v][0] = u;
                    maxWeight[v][0] = weight;
                    queue.offer(v);
                }
            }
        }

        // 动态规划预处理
        for (int j = 1; j < LOG; j++) {
            for (int i = 0; i < n; i++) {
                if (parent[i][j-1] != -1) {
                    parent[i][j] = parent[parent[i][j-1]][j-1];
                    maxWeight[i][j] = Math.max(maxWeight[i][j-1],
                                               maxWeight[parent[i][j-1]][j-1]);
                }
            }
        }
    }

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

        int max = 0;

        // 将 u 提升到与 v 同一深度
        int diff = depth[u] - depth[v];
        for (int j = 0; j < LOG; j++) {
            if ((diff & (1 << j)) != 0) {
                max = Math.max(max, maxWeight[u][j]);
                u = parent[u][j];
            }
        }
```

```
        if (u == v) {
            return max;
        }

        // 同时提升 u 和 v
        for (int j = LOG - 1; j >= 0; j--) {
            if (parent[u][j] != parent[v][j]) {
                max = Math.max(max, maxWeight[u][j]);
                max = Math.max(max, maxWeight[v][j]);
                u = parent[u][j];
                v = parent[v][j];
            }
        }

        max = Math.max(max, maxWeight[u][0]);
        max = Math.max(max, maxWeight[v][0]);

        return max;
    }
}

/**
 * 方法 3: 使用二分答案 + BFS/DFS（适用于小规模数据）
 * 时间复杂度: O(Q * (E + V))，对于大规模数据会超时
 * 空间复杂度: O(E + V)
 *
 * 解题思路:
 * 1. 对于每个查询，使用二分查找确定可行的最大边权
 * 2. 或者直接使用 BFS/DFS 检查是否存在满足条件的路径
 */
public boolean[] distanceLimitedPathsExistBFS(int n, int[][] edges, int[][] queries) {
    // 构建邻接表
    @SuppressWarnings("unchecked")
    List<int[]>[] graph = new ArrayList[n];
    for (int i = 0; i < n; i++) {
        graph[i] = new ArrayList<>();
    }

    for (int[] edge : edges) {
        int u = edge[0];
        int v = edge[1];
        int weight = edge[2];
```

```
            graph[u].add(new int[]{v, weight});
            graph[v].add(new int[]{u, weight});
        }

        boolean[] result = new boolean[queries.length];
        for (int i = 0; i < queries.length; i++) {
            int u = queries[i][0];
            int v = queries[i][1];
            int limit = queries[i][2];

            result[i] = bfs(graph, u, v, limit);
        }

        return result;
    }

    private boolean bfs(List<int[]>[] graph, int start, int end, int limit) {
        int n = graph.length;
        boolean[] visited = new boolean[n];
        Queue<Integer> queue = new LinkedList<>();
        queue.offer(start);
        visited[start] = true;

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

            if (u == end) {
                return true;
            }

            for (int[] edge : graph[u]) {
                int v = edge[0];
                int weight = edge[1];

                if (!visited[v] && weight < limit) {
                    visited[v] = true;
                    queue.offer(v);
                }
            }
        }

        return false;
    }
```

```
// 测试方法
public static void main(String[] args) {
    Code20_LeetCode1697 solution = new Code20_LeetCode1697();

    // 测试用例 1
    int n1 = 3;
    int[][] edges1 = {
        {0, 1, 2}, {1, 2, 4}, {2, 0, 8}, {2, 0, 1}
    };
    int[][] queries1 = {
        {0, 1, 2}, {0, 2, 5}
    };

    boolean[] result1 = solution.distanceLimitedPathsExist(n1, edges1, queries1);
    System.out.println("测试用例 1: " + Arrays.toString(result1));
    // 预期: [false, true]

    // 测试用例 2
    int n2 = 5;
    int[][] edges2 = {
        {0, 1, 10}, {1, 2, 5}, {2, 3, 9}, {3, 4, 13}
    };
    int[][] queries2 = {
        {0, 4, 14}, {1, 4, 13}
    };

    boolean[] result2 = solution.distanceLimitedPathsExist(n2, edges2, queries2);
    System.out.println("测试用例 2: " + Arrays.toString(result2));
    // 预期: [true, false]

    // 测试用例 3: 大规模数据测试（简化版）
    int n3 = 1000;
    int[][] edges3 = generateRandomEdges(n3, 5000);
    int[][] queries3 = generateRandomQueries(n3, 100);

    long startTime = System.currentTimeMillis();
    boolean[] result3 = solution.distanceLimitedPathsExist(n3, edges3, queries3);
    long endTime = System.currentTimeMillis();
    System.out.println("测试用例 3 - 处理时间: " + (endTime - startTime) + "ms");

    System.out.println("离线查询 + 并查集解法总结:");
    System.out.println("1. 核心思想: 将查询按限制排序，逐步添加边");
```

```
        System.out.println("2. 优势: 避免重复计算，时间复杂度优秀");
        System.out.println("3. 适用场景: 大规模数据，查询独立");
        System.out.println("4. 注意事项: 需要保存查询的原始顺序");
    }

    // 辅助方法：生成随机边
    private static int[][] generateRandomEdges(int n, int edgeCount) {
        Random random = new Random();
        int[][] edges = new int[edgeCount][3];

        for (int i = 0; i < edgeCount; i++) {
            int u = random.nextInt(n);
            int v = random.nextInt(n);
            while (u == v) {
                v = random.nextInt(n);
            }
            int weight = random.nextInt(1000) + 1;
            edges[i] = new int[]{u, v, weight};
        }

        return edges;
    }

    // 辅助方法：生成随机查询
    private static int[][] generateRandomQueries(int n, int queryCount) {
        Random random = new Random();
        int[][] queries = new int[queryCount][3];

        for (int i = 0; i < queryCount; i++) {
            int u = random.nextInt(n);
            int v = random.nextInt(n);
            while (u == v) {
                v = random.nextInt(n);
            }
            int limit = random.nextInt(1000) + 1;
            queries[i] = new int[]{u, v, limit};
        }

        return queries;
    }

    /**
     * 性能优化建议:
```

```
     * 1. 对于大规模数据，优先使用离线查询 + 并查集
     * 2. 如果查询需要在线处理，考虑使用 LCA 方法
     * 3. 避免使用 BFS/DFS 处理大规模数据
     * 4. 注意内存使用，避免创建过多临时对象
     *
     * 工程化考量:
     * 1. 异常处理: 检查节点编号是否越界
     * 2. 边界情况: 处理 n=0 或 n=1 的情况
     * 3. 内存优化: 对于超大 n，考虑使用更紧凑的数据结构
     * 4. 并发安全: 如果需要在多线程环境使用，需要额外处理
     */
}

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

文件: Code21_LeetCode803.java
==========================================
package class056;

import java.util.*;

/**
 * LeetCode 803. 打砖块
 * 链接: https://leetcode.cn/problems/bricks-falling-when-hit/
 * 难度: 困难
 *
 * 题目描述:
 * 有一个 m x n 的二元网格，其中 1 表示砖块，0 表示空白。砖块稳定（不会掉落）的前提是：
 * 1. 一块砖直接连接到网格的顶部（第一行）
 * 2. 或者至少有一块相邻（4 个方向之一）的稳定砖块
 *
 * 给你一个数组 hits ，这是需要依次消除砖块的位置。每当消除 hits[i] = (rowi, coli) 位置上的砖块时，对应位置的砖块会消失，
 * 然后有一批砖块可能因为这个消除而掉落。你需要返回一个数组 result，其中 result[i] 表示第 i 次消除后掉落的砖块数目。
 *
 * 注意：消除可能指向没有砖块的位置，此时不会有砖块掉落。
 *
 * 示例 1:
 * 输入: grid = [[1,0,0,0],[1,1,1,0]], hits = [[1,0]]
 * 输出: [2]
 * 解释:
 * 网格开始为：
```

```
 * [[1,0,0,0],
 *  [1,1,1,0]]
 * 消除 (1,0) 的砖块，网格变成：
 * [[1,0,0,0],
 *  [0,1,1,0]]
 * 两个标红的砖块掉落，返回 2。
 *
 * 示例 2:
 * 输入: grid = [[1,0,0,0],[1,1,0,0]], hits = [[1,1],[1,0]]
 * 输出: [0,0]
 * 解释:
 * 网格开始为：
 * [[1,0,0,0],
 *  [1,1,0,0]]
 * 消除 (1,1) 的砖块，网格不变，没有砖块掉落。
 * 消除 (1,0) 的砖块，网格变成：
 * [[1,0,0,0],
 *  [0,0,0,0]]
 * 标红的砖块已经掉落过，所以返回 0。
 *
 * 约束条件:
 * m == grid.length
 * n == grid[i].length
 * 1 <= m, n <= 200
 * grid[i][j] 为 0 或 1
 * 1 <= hits.length <= 4 * 10^4
 * hits[i].length == 2
 * 0 <= xi <= m - 1
 * 0 <= yi <= n - 1
 * 所有 (xi, yi) 互不相同
 */
public class Code21_LeetCode803 {

    /**
     * 方法 1: 逆向并查集（推荐解法）
     * 时间复杂度: O((M*N + H) * α(M*N))，其中 M*N 是网格大小，H 是 hits 数量
     * 空间复杂度: O(M*N)
     *
     * 解题思路:
     * 1. 逆向思考：从最终状态开始，逐步添加被消除的砖块
     * 2. 使用并查集维护连通性，特别关注与顶部相连的连通分量
     * 3. 添加砖块时，计算新增的稳定砖块数量
     */
```

```
public int[] hitBricks(int[][] grid, int[][] hits) {
    int m = grid.length;
    int n = grid[0].length;

    // 复制网格，用于逆向处理
    int[][] copy = new int[m][n];
    for (int i = 0; i < m; i++) {
        copy[i] = grid[i].clone();
    }

    // 第一步：先消除所有要打的砖块
    for (int[] hit : hits) {
        int x = hit[0], y = hit[1];
        copy[x][y] = 0;
    }

    // 第二步：初始化并查集，大小为 m*n + 1（额外一个虚拟节点代表顶部）
    int size = m * n;
    UnionFind uf = new UnionFind(size + 1);

    // 将第一行的砖块与虚拟顶部节点连接
    for (int j = 0; j < n; j++) {
        if (copy[0][j] == 1) {
            uf.union(j, size);  // 第一行第 j 列连接到顶部
        }
    }

    // 第三步：构建初始连通性（消除所有 hits 后的状态）
    for (int i = 1; i < m; i++) {
        for (int j = 0; j < n; j++) {
            if (copy[i][j] == 1) {
                // 检查上方砖块
                if (copy[i-1][j] == 1) {
                    uf.union(i * n + j, (i-1) * n + j);
                }
                // 检查左方砖块
                if (j > 0 && copy[i][j-1] == 1) {
                    uf.union(i * n + j, i * n + j - 1);
                }
            }
        }
    }
```

```
// 第四步：逆向添加砖块
int hitsLen = hits.length;
int[] res = new int[hitsLen];

// 四个方向
int[][] directions = {{-1, 0}, {1, 0}, {0, -1}, {0, 1}};

for (int i = hitsLen - 1; i >= 0; i--) {
    int x = hits[i][0];
    int y = hits[i][1];

    // 如果原始网格中这个位置没有砖块，不会导致掉落
    if (grid[x][y] == 0) {
        res[i] = 0;
        continue;
    }

    // 记录添加前的稳定砖块数量
    int origin = uf.getSize(size);

    // 如果这是第一行的砖块，连接到顶部
    if (x == 0) {
        uf.union(y, size);
    }

    // 添加当前砖块
    copy[x][y] = 1;

    // 检查四个方向，连接相邻砖块
    for (int[] dir : directions) {
        int newX = x + dir[0];
        int newY = y + dir[1];

        if (newX >= 0 && newX < m && newY >= 0 && newY < n && copy[newX][newY] == 1) {
            uf.union(x * n + y, newX * n + newY);
        }
    }

    // 计算新增的稳定砖块数量（减去当前添加的砖块）
    int current = uf.getSize(size);
    res[i] = Math.max(0, current - origin - 1);
}
```

```
    return res;
}

/**
 * 支持获取连通分量大小的并查集
 */
static class UnionFind {
    private int[] parent;
    private int[] size;

    public UnionFind(int n) {
        parent = new int[n];
        size = new int[n];
        for (int i = 0; i < n; i++) {
            parent[i] = i;
            size[i] = 1;
        }
    }

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

    public void union(int x, int y) {
        int rootX = find(x);
        int rootY = find(y);

        if (rootX != rootY) {
            // 小树合并到大树下
            if (size[rootX] < size[rootY]) {
                parent[rootX] = rootY;
                size[rootY] += size[rootX];
            } else {
                parent[rootY] = rootX;
                size[rootX] += size[rootY];
            }
        }
    }

    public int getSize(int x) {
```

```
        int root = find(x);
        return size[root];
    }
}

/**
 * 方法 2: DFS/BFS 解法（适用于小规模数据）
 * 时间复杂度: O(H * M * N)，对于大规模数据会超时
 * 空间复杂度: O(M * N)
 *
 * 解题思路:
 * 1. 对于每个 hit，消除对应砖块
 * 2. 使用 DFS/BFS 标记所有不稳定的砖块
 * 3. 计算掉落的砖块数量
 */
public int[] hitBricksDFS(int[][] grid, int[][] hits) {
    int m = grid.length;
    int n = grid[0].length;
    int[][] copy = new int[m][n];

    // 复制网格
    for (int i = 0; i < m; i++) {
        copy[i] = grid[i].clone();
    }

    // 先消除所有要打的砖块
    for (int[] hit : hits) {
        int x = hit[0], y = hit[1];
        copy[x][y] = 0;
    }

    // 标记稳定的砖块（与顶部相连）
    boolean[][] stable = new boolean[m][n];
    for (int j = 0; j < n; j++) {
        if (copy[0][j] == 1) {
            dfsMarkStable(copy, stable, 0, j);
        }
    }

    int[] res = new int[hits.length];

    // 逆向处理 hits
    for (int i = hits.length - 1; i >= 0; i--) {
```

```
            int x = hits[i][0];
            int y = hits[i][1];

            if (grid[x][y] == 0) {
                res[i] = 0;
                continue;
            }

            // 恢复砖块
            copy[x][y] = 1;

            // 检查当前砖块是否应该稳定
            boolean shouldBeStable = (x == 0); // 如果在第一行，直接稳定

            // 检查四个方向是否有稳定砖块
            int[][] directions = {{-1, 0}, {1, 0}, {0, -1}, {0, 1}};
            for (int[] dir : directions) {
                int newX = x + dir[0];
                int newY = y + dir[1];
                if (newX >= 0 && newX < m && newY >= 0 && newY < n && stable[newX][newY]) {
                    shouldBeStable = true;
                    break;
                }
            }

            if (shouldBeStable) {
                // 从当前砖块开始DFS，标记新稳定的砖块
                int before = countStable(stable);
                dfsMarkStable(copy, stable, x, y);
                int after = countStable(stable);
                res[i] = after - before - 1; // 减去当前添加的砖块
            } else {
                res[i] = 0;
            }
        }

        return res;
    }

    private void dfsMarkStable(int[][] grid, boolean[][] stable, int x, int y) {
        int m = grid.length;
        int n = grid[0].length;
```

```
    if (x < 0 || x >= m || y < 0 || y >= n || grid[x][y] == 0 || stable[x][y]) {
        return;
    }

    stable[x][y] = true;

    int[][] directions = {{-1, 0}, {1, 0}, {0, -1}, {0, 1}};
    for (int[] dir : directions) {
        dfsMarkStable(grid, stable, x + dir[0], y + dir[1]);
    }
}

private int countStable(boolean[][] stable) {
    int count = 0;
    for (boolean[] row : stable) {
        for (boolean cell : row) {
            if (cell) count++;
        }
    }
    return count;
}

/**
 * 方法 3: 使用带权并查集优化（记录每个连通分量到顶部的距离）
 * 时间复杂度: 与逆向并查集相同
 * 空间复杂度: O(M*N)
 */
public int[] hitBricksWeighted(int[][] grid, int[][] hits) {
    int m = grid.length;
    int n = grid[0].length;
    int size = m * n;

    // 复制网格
    int[][] copy = new int[m][n];
    for (int i = 0; i < m; i++) {
        copy[i] = grid[i].clone();
    }

    // 消除所有 hits
    for (int[] hit : hits) {
        copy[hit[0]][hit[1]] = 0;
    }
```

```
// 初始化并查集
WeightedUnionFind uf = new WeightedUnionFind(size);

// 构建初始连通性
for (int i = 0; i < m; i++) {
    for (int j = 0; j < n; j++) {
        if (copy[i][j] == 1) {
            int index = i * n + j;
            // 如果是第一行，标记为连接到顶部
            if (i == 0) {
                uf.setTopConnected(index);
            }

            // 连接相邻砖块
            if (i > 0 && copy[i-1][j] == 1) {
                uf.union(index, (i-1) * n + j);
            }
            if (j > 0 && copy[i][j-1] == 1) {
                uf.union(index, i * n + j - 1);
            }
        }
    }
}

// 逆向处理
int[] res = new int[hits.length];
int[][] directions = {{-1, 0}, {1, 0}, {0, -1}, {0, 1}};

for (int i = hits.length - 1; i >= 0; i--) {
    int x = hits[i][0];
    int y = hits[i][1];

    if (grid[x][y] == 0) {
        res[i] = 0;
        continue;
    }

    int index = x * n + y;
    int before = uf.getTopConnectedCount();

    // 恢复砖块
    copy[x][y] = 1;
    if (x == 0) {
```

```
                uf.setTopConnected(index);
            }

            // 连接相邻砖块
            for (int[] dir : directions) {
                int newX = x + dir[0];
                int newY = y + dir[1];
                if (newX >= 0 && newX < m && newY >= 0 && newY < n && copy[newX][newY] == 1) {
                    uf.union(index, newX * n + newY);
                }
            }

            int after = uf.getTopConnectedCount();
            res[i] = Math.max(0, after - before - 1);
        }

        return res;
    }

    /**
     * 带权并查集，记录连通分量是否连接到顶部
     */
    static class WeightedUnionFind {
        private int[] parent;
        private boolean[] topConnected; // 根节点是否连接到顶部
        private int[] size;
        private int topCount; // 连接到顶部的砖块总数

        public WeightedUnionFind(int n) {
            parent = new int[n];
            topConnected = new boolean[n];
            size = new int[n];
            topCount = 0;

            for (int i = 0; i < n; i++) {
                parent[i] = i;
                size[i] = 1;
                topConnected[i] = false;
            }
        }

        public int find(int x) {
            if (parent[x] != x) {
```

```
            parent[x] = find(parent[x]);
        }
        return parent[x];
    }

    public void union(int x, int y) {
        int rootX = find(x);
        int rootY = find(y);

        if (rootX != rootY) {
            // 小树合并到大树下
            if (size[rootX] < size[rootY]) {
                parent[rootX] = rootY;
                size[rootY] += size[rootX];
                if (topConnected[rootX] && !topConnected[rootY]) {
                    topConnected[rootY] = true;
                    topCount += size[rootX];
                }
            } else {
                parent[rootY] = rootX;
                size[rootX] += size[rootY];
                if (topConnected[rootY] && !topConnected[rootX]) {
                    topConnected[rootX] = true;
                    topCount += size[rootY];
                }
            }
        }
    }

    public void setTopConnected(int x) {
        int root = find(x);
        if (!topConnected[root]) {
            topConnected[root] = true;
            topCount += size[root];
        }
    }

    public int getTopConnectedCount() {
        return topCount;
    }
}

// 测试方法
```

```
public static void main(String[] args) {
    Code21_LeetCode803 solution = new Code21_LeetCode803();

    // 测试用例 1
    int[][] grid1 = {
        {1, 0, 0, 0},
        {1, 1, 1, 0}
    };
    int[][] hits1 = {{1, 0}};
    int[] result1 = solution.hitBricks(grid1, hits1);
    System.out.println("测试用例 1: " + Arrays.toString(result1));
    // 预期: [2]

    // 测试用例 2
    int[][] grid2 = {
        {1, 0, 0, 0},
        {1, 1, 0, 0}
    };
    int[][] hits2 = {{1, 1}, {1, 0}};
    int[] result2 = solution.hitBricks(grid2, hits2);
    System.out.println("测试用例 2: " + Arrays.toString(result2));
    // 预期: [0, 0]

    // 测试用例 3: 更大规模的测试
    int[][] grid3 = {
        {1, 1, 1, 1},
        {1, 1, 1, 1},
        {1, 1, 1, 1}
    };
    int[][] hits3 = {{1, 1}, {2, 2}};
    int[] result3 = solution.hitBricks(grid3, hits3);
    System.out.println("测试用例 3: " + Arrays.toString(result3));

    System.out.println("逆向并查集解法总结:");
    System.out.println("1. 核心思想: 从最终状态逆向添加砖块");
    System.out.println("2. 优势: 避免重复计算连通性");
    System.out.println("3. 关键技巧: 使用虚拟节点表示顶部");
    System.out.println("4. 适用场景: 需要处理多次消除操作的问题");
}

/**
 * 工程化考量:
 * 1. 内存优化: 对于超大网格，考虑使用一维数组存储
```

```
     * 2. 边界处理: 处理网格为空或 hits 为空的情况
     * 3. 性能监控: 添加性能统计，监控处理时间
     * 4. 测试覆盖: 编写单元测试覆盖各种边界情况
     *
     * 扩展应用:
     * 1. 可以扩展到三维网格的打砖块问题
     * 2. 可以处理不同类型的砖块（不同重量、强度）
     * 3. 可以支持动态添加砖块的操作
     */
}

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

文件: Code22_UnionFindPerformanceAnalysis.java
==========================================
package class056;

import java.util.*;
import java.util.concurrent.*;

/**
 * 并查集性能分析与优化实验
 * 本文件包含并查集各种实现的性能对比和优化实验
 */
public class Code22_UnionFindPerformanceAnalysis {

    /**
     * 1. 不同并查集实现的性能对比
     */
    public static void compareUnionFindImplementations() {
        System.out.println("=== 不同并查集实现性能对比 ===");

        int n = 1000000;  // 100 万节点
        int operations = 2000000; // 200 万次操作

        // 生成随机操作序列
        int[][] operationsSeq = generateRandomOperations(n, operations);

        // 测试不同实现的性能
        testImplementation("基础并查集（无优化）", new BasicUnionFind(n), operationsSeq);
        testImplementation("路径压缩并查集", new PathCompressionUnionFind(n), operationsSeq);
        testImplementation("按秩合并并查集", new UnionByRankUnionFind(n), operationsSeq);
        testImplementation("路径压缩+按秩合并", new OptimizedUnionFind(n), operationsSeq);
```

```
}

/**
 * 基础并查集（无优化）
 */
static class BasicUnionFind {
    protected int[] parent;

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

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

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

    public boolean isConnected(int x, int y) {
        return find(x) == find(y);
    }
}

/**
 * 路径压缩并查集
 */
static class PathCompressionUnionFind extends BasicUnionFind {
    public PathCompressionUnionFind(int n) {
        super(n);
    }
```

```
    @Override
    public int find(int x) {
        if (parent[x] != x) {
            parent[x] = find(parent[x]);
        }
        return parent[x];
    }
}

/**
 * 按秩合并并查集
 */
static class UnionByRankUnionFind extends BasicUnionFind {
    private int[] rank;

    public UnionByRankUnionFind(int n) {
        super(n);
        rank = new int[n];
        Arrays.fill(rank, 1);
    }

    @Override
    public void union(int x, int y) {
        int rootX = find(x);
        int rootY = find(y);

        if (rootX != rootY) {
            if (rank[rootX] > rank[rootY]) {
                parent[rootY] = rootX;
            } else if (rank[rootX] < rank[rootY]) {
                parent[rootX] = rootY;
            } else {
                parent[rootY] = rootX;
                rank[rootX]++;
            }
        }
    }
}

/**
 * 优化版并查集（路径压缩 + 按秩合并）
 */
static class OptimizedUnionFind extends UnionByRankUnionFind {
```

```
    public OptimizedUnionFind(int n) {
        super(n);
    }

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

/**
 * 性能测试方法
 */
private static void testImplementation(String name, BasicUnionFind uf, int[][] operations) {
    long startTime = System.currentTimeMillis();

    for (int[] op : operations) {
        if (op[0] == 0) { // union 操作
            uf.union(op[1], op[2]);
        } else { // find 操作
            uf.isConnected(op[1], op[2]);
        }
    }

    long endTime = System.currentTimeMillis();
    System.out.printf("%s: %d ms%n", name, endTime - startTime);
}

/**
 * 生成随机操作序列
 */
private static int[][] generateRandomOperations(int n, int count) {
    Random random = new Random();
    int[][] operations = new int[count][3];

    for (int i = 0; i < count; i++) {
        int type = random.nextInt(2); // 0: union, 1: find
        int x = random.nextInt(n);
        int y = random.nextInt(n);
        operations[i] = new int[]{type, x, y};
```

```
    }

    return operations;
}

/**
 * 2. 并查集在不同数据分布下的性能分析
 */
public static void analyzePerformanceUnderDifferentDistributions() {
    System.out.println("\n=== 不同数据分布下的性能分析 ===");

    int n = 100000;
    int operations = 500000;

    // 测试不同数据分布
    testDistribution("随机分布", n, operations, Distribution.RANDOM);
    testDistribution("链式分布", n, operations, Distribution.CHAIN);
    testDistribution("星形分布", n, operations, Distribution.STAR);
    testDistribution("完全二分分布", n, operations, Distribution.BIPARTITE);
}

enum Distribution {
    RANDOM, CHAIN, STAR, BIPARTITE
}

private static void testDistribution(String name, int n, int operations, Distribution dist) {
    int[][] operationsSeq = generateOperationsByDistribution(n, operations, dist);

    long startTime = System.currentTimeMillis();
    OptimizedUnionFind uf = new OptimizedUnionFind(n);

    for (int[] op : operationsSeq) {
        if (op[0] == 0) {
            uf.union(op[1], op[2]);
        } else {
            uf.isConnected(op[1], op[2]);
        }
    }

    long endTime = System.currentTimeMillis();
    System.out.printf("%s: %d ms%n", name, endTime - startTime);
}
```

```
    private static int[][] generateOperationsByDistribution(int n, int count, Distribution dist)
{
        Random random = new Random();
        int[][] operations = new int[count][3];

        switch (dist) {
            case RANDOM:
                return generateRandomOperations(n, count);

            case CHAIN:
                // 生成链式操作：连续合并形成长链
                for (int i = 0; i < count; i++) {
                    if (i < n - 1) {
                        operations[i] = new int[]{0, i, i + 1}; // 形成链
                    } else {
                        operations[i] = new int[]{1, 0, n - 1}; // 查询链的两端
                    }
                }
                break;

            case STAR:
                // 生成星形操作：所有节点连接到中心节点
                int center = 0;
                for (int i = 0; i < count; i++) {
                    if (i < n - 1) {
                        operations[i] = new int[]{0, center, i + 1}; // 连接到中心
                    } else {
                        operations[i] = new int[]{1, random.nextInt(n), random.nextInt(n)};
                    }
                }
                break;

            case BIPARTITE:
                // 生成二分图操作：交替合并
                for (int i = 0; i < count; i++) {
                    if (i < Math.min(count, n/2)) {
                        int group1 = i * 2;
                        int group2 = i * 2 + 1;
                        if (group2 < n) {
                            operations[i] = new int[]{0, group1, group2};
                        } else {
                            operations[i] = new int[]{1, random.nextInt(n), random.nextInt(n)};
                        }
```

```
                } else {
                    operations[i] = new int[]{1, random.nextInt(n), random.nextInt(n)};
                }
            }
            break;
    }

    return operations;
}

/**
 * 3. 内存使用分析
 */
public static void analyzeMemoryUsage() {
    System.out.println("\n=== 内存使用分析 ===");

    int[] sizes = {1000, 10000, 100000, 1000000};

    for (int size : sizes) {
        long memoryBefore = getMemoryUsage();

        // 创建并查集实例
        OptimizedUnionFind uf = new OptimizedUnionFind(size);

        long memoryAfter = getMemoryUsage();
        long memoryUsed = memoryAfter - memoryBefore;

        System.out.printf("节点数: %d, 内存使用: %d bytes, 平均每个节点: %.2f bytes%n",
                size, memoryUsed, (double)memoryUsed / size);
    }
}

private static long getMemoryUsage() {
    Runtime runtime = Runtime.getRuntime();
    return runtime.totalMemory() - runtime.freeMemory();
}

/**
 * 4. 多线程环境下的并查集性能
 */
public static void analyzeConcurrentPerformance() throws InterruptedException {
    System.out.println("\n=== 多线程性能分析 ===");
```

```
        int n = 100000;
        int threads = 4;
        int operationsPerThread = 100000;

        ExecutorService executor = Executors.newFixedThreadPool(threads);
        OptimizedUnionFind uf = new OptimizedUnionFind(n);

        long startTime = System.currentTimeMillis();

        List<Future<?>> futures = new ArrayList<>();
        for (int i = 0; i < threads; i++) {
            futures.add(executor.submit(new UnionFindTask(uf, n, operationsPerThread)));
        }

        // 等待所有任务完成
        for (Future<?> future : futures) {
            try {
                future.get();
            } catch (ExecutionException e) {
                e.printStackTrace();
            }
        }

        long endTime = System.currentTimeMillis();
        executor.shutdown();

        System.out.printf("多线程(%d 线程)处理时间: %d ms%n", threads, endTime - startTime);
    }

    static class UnionFindTask implements Runnable {
        private final OptimizedUnionFind uf;
        private final int n;
        private final int operations;
        private final Random random = new Random();

        public UnionFindTask(OptimizedUnionFind uf, int n, int operations) {
            this.uf = uf;
            this.n = n;
            this.operations = operations;
        }

        @Override
        public void run() {
```

```
        for (int i = 0; i < operations; i++) {
            int x = random.nextInt(n);
            int y = random.nextInt(n);

            if (random.nextBoolean()) {
                uf.union(x, y);
            } else {
                uf.isConnected(x, y);
            }
        }
    }
}

/**
 * 5. 实际应用场景性能测试
 */
public static void testRealWorldScenarios() {
    System.out.println("\n=== 实际应用场景性能测试 ===");

    // 场景1: 社交网络好友关系
    testSocialNetworkScenario();

    // 场景2: 图像连通区域标记
    testImageConnectedComponentsScenario();

    // 场景3: 最小生成树算法
    testMSTScenario();
}

private static void testSocialNetworkScenario() {
    System.out.println("社交网络场景测试:");

    int users = 50000;
    int friendships = 200000;
    int queries = 100000;

    // 生成社交网络数据
    int[][] friendshipsData = generateSocialNetworkData(users, friendships);
    int[][] queriesData = generateQueries(users, queries);

    long startTime = System.currentTimeMillis();

    OptimizedUnionFind uf = new OptimizedUnionFind(users);
```

```
        // 建立好友关系
        for (int[] friendship : friendshipsData) {
            uf.union(friendship[0], friendship[1]);
        }

        // 处理查询
        int connectedCount = 0;
        for (int[] query : queriesData) {
            if (uf.isConnected(query[0], query[1])) {
                connectedCount++;
            }
        }

        long endTime = System.currentTimeMillis();

        System.out.printf("用户数: %d, 好友关系: %d, 查询数: %d, 处理时间: %d ms, 连通查
询: %d%n",
                users, friendships, queries, endTime - startTime, connectedCount);
    }

    private static int[][] generateSocialNetworkData(int users, int friendships) {
        Random random = new Random();
        int[][] data = new int[friendships][2];

        for (int i = 0; i < friendships; i++) {
            int u = random.nextInt(users);
            int v = random.nextInt(users);
            while (u == v) {
                v = random.nextInt(users);
            }
            data[i] = new int[]{u, v};
        }

        return data;
    }

    private static int[][] generateQueries(int users, int queries) {
        Random random = new Random();
        int[][] data = new int[queries][2];

        for (int i = 0; i < queries; i++) {
            int u = random.nextInt(users);
```

```
            int v = random.nextInt(users);
            data[i] = new int[]{u, v};
        }

        return data;
    }

    private static void testImageConnectedComponentsScenario() {
        System.out.println("图像连通区域标记测试:");

        int width = 1000;
        int height = 1000;
        int totalPixels = width * height;

        // 生成随机二值图像
        Random random = new Random();
        int[] image = new int[totalPixels];
        for (int i = 0; i < totalPixels; i++) {
            image[i] = random.nextDouble() < 0.3 ? 1 : 0; // 30%的概率为前景
        }

        long startTime = System.currentTimeMillis();

        // 使用并查集标记连通区域
        OptimizedUnionFind uf = new OptimizedUnionFind(totalPixels);

        // 连接相邻像素
        for (int i = 0; i < height; i++) {
            for (int j = 0; j < width; j++) {
                int index = i * width + j;
                if (image[index] == 1) {
                    // 检查右方像素
                    if (j < width - 1 && image[index + 1] == 1) {
                        uf.union(index, index + 1);
                    }
                    // 检查下方像素
                    if (i < height - 1 && image[index + width] == 1) {
                        uf.union(index, index + width);
                    }
                }
            }
        }
```

```
        // 统计连通区域数量
        Set<Integer> components = new HashSet<>();
        for (int i = 0; i < totalPixels; i++) {
            if (image[i] == 1) {
                components.add(uf.find(i));
            }
        }

        long endTime = System.currentTimeMillis();

        System.out.printf("图像大小: %dx%d, 连通区域数: %d, 处理时间: %d ms%n",
                width, height, components.size(), endTime - startTime);
    }

    private static void testMSTScenario() {
        System.out.println("最小生成树算法测试:");

        int vertices = 10000;
        int edges = 50000;

        // 生成随机图
        int[][] graphEdges = generateGraphEdges(vertices, edges);

        long startTime = System.currentTimeMillis();

        // 使用 Kruskal 算法求最小生成树
        Arrays.sort(graphEdges, (a, b) -> Integer.compare(a[2], b[2]));

        OptimizedUnionFind uf = new OptimizedUnionFind(vertices);
        int mstEdges = 0;
        int totalWeight = 0;

        for (int[] edge : graphEdges) {
            if (mstEdges == vertices - 1) break;

            if (!uf.isConnected(edge[0], edge[1])) {
                uf.union(edge[0], edge[1]);
                totalWeight += edge[2];
                mstEdges++;
            }
        }

        long endTime = System.currentTimeMillis();
```

```
    System.out.printf("顶点数: %d, 边数: %d, MST 边数: %d, 总权重: %d, 处理时间: %d ms%n",
            vertices, edges, mstEdges, totalWeight, endTime - startTime);
}

private static int[][] generateGraphEdges(int vertices, int edges) {
    Random random = new Random();
    int[][] graphEdges = new int[edges][3];

    for (int i = 0; i < edges; i++) {
        int u = random.nextInt(vertices);
        int v = random.nextInt(vertices);
        while (u == v) {
            v = random.nextInt(vertices);
        }
        int weight = random.nextInt(1000) + 1;
        graphEdges[i] = new int[]{u, v, weight};
    }

    return graphEdges;
}

/**
 * 6. 性能优化建议总结
 */
public static void performanceOptimizationSummary() {
    System.out.println("\n=== 性能优化建议总结 ===");

    System.out.println("1. 算法选择优化:");
    System.out.println("   - 优先使用路径压缩 + 按秩合并的优化版本");
    System.out.println("   - 对于特定场景，考虑使用带权并查集或可撤销并查集");

    System.out.println("2. 内存优化:");
    System.out.println("   - 使用基本类型数组代替对象数组");
    System.out.println("   - 对于超大数组，考虑分块存储或使用更紧凑的数据结构");

    System.out.println("3. 并发优化:");
    System.out.println("   - 在多线程环境下，考虑使用线程本地存储");
    System.out.println("   - 或者使用并发安全的数据结构包装");

    System.out.println("4. 实际应用优化:");
    System.out.println("   - 根据数据分布特点选择合适的数据结构");
    System.out.println("   - 对于稀疏图，考虑使用其他数据结构");
```

```
        System.out.println("   - 预处理数据，减少运行时计算");
    }

    // 主测试方法
    public static void main(String[] args) throws InterruptedException {
        System.out.println("并查集性能分析与优化实验");
        System.out.println("========================");

        // 运行各种性能测试
        compareUnionFindImplementations();
        analyzePerformanceUnderDifferentDistributions();
        analyzeMemoryUsage();
        analyzeConcurrentPerformance();
        testRealWorldScenarios();
        performanceOptimizationSummary();

        System.out.println("\n 性能分析完成！");
    }

    /**
     * 关键性能指标监控:
     * 1. 时间复杂度: 关注最坏情况和平均情况
     * 2. 空间复杂度: 关注内存使用和缓存友好性
     * 3. 实际运行时间: 在不同规模和数据分布下的表现
     * 4. 并发性能: 多线程环境下的扩展性
     *
     * 调试和问题定位技巧:
     * 1. 使用性能分析工具（如 JProfiler、VisualVM）
     * 2. 添加详细的日志输出
     * 3. 编写基准测试对比不同实现
     * 4. 监控内存使用和 GC 情况
     */
}

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

文件: Code23_UnionFindPythonImplementations.py
==============================================
"""
并查集 Python 实现合集
包含各种并查集变种的 Python 实现和测试
"""
```

```
from typing import List, Dict, Set, Tuple, Optional
import collections
import heapq
import unittest

class UnionFind:
    """标准并查集实现（路径压缩 + 按秩合并）"""

    def __init__(self, n: int):
        self.parent = list(range(n))
        self.rank = [1] * n
        self.count = n  # 连通分量数量

    def find(self, x: int) -> int:
        """查找根节点，并进行路径压缩"""
        if self.parent[x] != x:
            self.parent[x] = self.find(self.parent[x])
        return self.parent[x]

    def union(self, x: int, y: int) -> bool:
        """合并两个节点所在的集合"""
        root_x = self.find(x)
        root_y = self.find(y)

        if root_x == root_y:
            return False  # 已经在同一集合

        # 按秩合并
        if self.rank[root_x] < self.rank[root_y]:
            root_x, root_y = root_y, root_x

        self.parent[root_y] = root_x
        if self.rank[root_x] == self.rank[root_y]:
            self.rank[root_x] += 1

        self.count -= 1
        return True

    def is_connected(self, x: int, y: int) -> bool:
        """检查两个节点是否连通"""
        return self.find(x) == self.find(y)

    def get_count(self) -> int:
```

```
        """获取连通分量数量"""
        return self.count

class WeightedUnionFind:
    """带权并查集实现"""

    def __init__(self, n: int):
        self.parent = list(range(n))
        self.weight = [0.0] * n  # 存储到父节点的权重

    def find(self, x: int) -> Tuple[int, float]:
        """查找根节点，并返回权重（x 的值 / 根节点的值）"""
        if self.parent[x] != x:
            root, root_weight = self.find(self.parent[x])
            self.parent[x] = root
            self.weight[x] *= root_weight
            return root, self.weight[x]
        return x, 1.0

    def union(self, x: int, y: int, value: float) -> bool:
        """合并集合，表示 x / y = value"""
        root_x, weight_x = self.find(x)
        root_y, weight_y = self.find(y)

        if root_x == root_y:
            # 检查是否冲突
            return abs(weight_x / weight_y - value) < 1e-9

        # 合并集合
        self.parent[root_x] = root_y
        # 更新权重：x 到 root_y 的权重 = (x 到 y 的权重) * (y 到 root_y 的权重)
        self.weight[root_x] = value * weight_y / weight_x
        return True

    def query(self, x: int, y: int) -> Optional[float]:
        """查询 x / y 的值"""
        if not self.is_connected(x, y):
            return None
        root_x, weight_x = self.find(x)
        root_y, weight_y = self.find(y)
        return weight_x / weight_y

    def is_connected(self, x: int, y: int) -> bool:
```

```
        """检查是否连通"""
        return self.find(x)[0] == self.find(y)[0]

class LeetCode128:
    """LeetCode 128. 最长连续序列"""

    def longestConsecutive(self, nums: List[int]) -> int:
        """
        方法 1: 使用并查集
        时间复杂度: O(n * α(n))
        空间复杂度: O(n)
        """
        if not nums:
            return 0

        # 创建数字到索引的映射
        num_to_index = {num: i for i, num in enumerate(nums)}
        n = len(nums)
        uf = UnionFind(n)

        # 合并相邻数字
        for i, num in enumerate(nums):
            if num - 1 in num_to_index:
                uf.union(i, num_to_index[num - 1])
            if num + 1 in num_to_index:
                uf.union(i, num_to_index[num + 1])

        # 统计每个连通分量的大小
        size_map = collections.defaultdict(int)
        for i in range(n):
            root = uf.find(i)
            size_map[root] += 1

        return max(size_map.values()) if size_map else 0

    def longestConsecutiveHashSet(self, nums: List[int]) -> int:
        """
        方法 2: 使用哈希表（最优解）
        时间复杂度: O(n)
        空间复杂度: O(n)
        """
        num_set = set(nums)
        longest_streak = 0
```

```
        for num in num_set:
            # 只有当num是序列起点时才查找
            if num - 1 not in num_set:
                current_num = num
                current_streak = 1

                while current_num + 1 in num_set:
                    current_num += 1
                    current_streak += 1

                longest_streak = max(longest_streak, current_streak)

        return longest_streak

class LeetCode547:
    """LeetCode 547. 省份数量（朋友圈）"""

    def findCircleNum(self, isConnected: List[List[int]]) -> int:
        """
        使用并查集解决省份数量问题
        时间复杂度: O(n^2 * α(n))
        空间复杂度: O(n)
        """
        n = len(isConnected)
        uf = UnionFind(n)

        for i in range(n):
            for j in range(i + 1, n):
                if isConnected[i][j] == 1:
                    uf.union(i, j)

        return uf.get_count()

    def findCircleNumDFS(self, isConnected: List[List[int]]) -> int:
        """
        使用DFS解决省份数量问题
        时间复杂度: O(n^2)
        空间复杂度: O(n)
        """
        n = len(isConnected)
        visited = [False] * n
        count = 0
```

```
        def dfs(city: int):
            visited[city] = True
            for neighbor in range(n):
                if isConnected[city][neighbor] == 1 and not visited[neighbor]:
                    dfs(neighbor)

        for i in range(n):
            if not visited[i]:
                count += 1
                dfs(i)

        return count

class LeetCode684:
    """LeetCode 684. 冗余连接"""

    def findRedundantConnection(self, edges: List[List[int]]) -> List[int]:
        """
        使用并查集检测环
        时间复杂度: O(n * α(n))
        空间复杂度: O(n)
        """
        n = len(edges)
        uf = UnionFind(n + 1)  # 节点从1开始编号

        for u, v in edges:
            if not uf.union(u, v):
                return [u, v]  # 检测到环

        return []  # 理论上不会执行到这里

class LeetCode399:
    """LeetCode 399. 除法求值"""

    def calcEquation(self, equations: List[List[str]], values: List[float], queries:
List[List[str]]) -> List[float]:
        """
        使用带权并查集解决除法求值问题
        时间复杂度: O((E + Q) * α(V))
        空间复杂度: O(V)
        """
        # 创建变量到索引的映射
```

```
        variables = set()
        for eq in equations:
            variables.add(eq[0])
            variables.add(eq[1])

        var_to_idx = {var: i for i, var in enumerate(variables)}
        n = len(variables)
        uf = WeightedUnionFind(n)

        # 构建并查集
        for (a, b), value in zip(equations, values):
            idx_a, idx_b = var_to_idx[a], var_to_idx[b]
            uf.union(idx_a, idx_b, value)

        # 处理查询
        results = []
        for a, b in queries:
            if a not in var_to_idx or b not in var_to_idx:
                results.append(-1.0)
            else:
                idx_a, idx_b = var_to_idx[a], var_to_idx[b]
                result = uf.query(idx_a, idx_b)
                results.append(result if result is not None else -1.0)

        return results

class LeetCode803:
    """LeetCode 803. 打砖块"""

    def hitBricks(self, grid: List[List[int]], hits: List[List[int]]) -> List[int]:
        """
        使用逆向并查集解决打砖块问题
        时间复杂度: O((M*N + H) * α(M*N))
        空间复杂度: O(M*N)
        """
        m, n = len(grid), len(grid[0])

        # 复制网格
        copy = [row[:] for row in grid]

        # 先消除所有要打的砖块
        for i, j in hits:
            copy[i][j] = 0
```

```
# 初始化并查集（增加一个虚拟顶部节点）
size = m * n
uf = UnionFind(size + 1)

# 将第一行砖块连接到虚拟顶部节点
for j in range(n):
    if copy[0][j] == 1:
        uf.union(j, size)

# 构建初始连通性
for i in range(1, m):
    for j in range(n):
        if copy[i][j] == 1:
            # 连接上方砖块
            if copy[i-1][j] == 1:
                uf.union(i * n + j, (i-1) * n + j)
            # 连接左方砖块
            if j > 0 and copy[i][j-1] == 1:
                uf.union(i * n + j, i * n + j - 1)

# 逆向处理 hits
directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]
result = [0] * len(hits)

for idx in range(len(hits) - 1, -1, -1):
    i, j = hits[idx]

    # 如果原始位置没有砖块
    if grid[i][j] == 0:
        result[idx] = 0
        continue

    # 记录添加前的稳定砖块数量
    origin_size = self.get_stable_bricks_count(uf, size)

    # 恢复砖块
    copy[i][j] = 1

    # 如果是第一行，连接到顶部
    if i == 0:
        uf.union(j, size)
```

```
            # 连接相邻砖块
            for di, dj in directions:
                ni, nj = i + di, j + dj
                if 0 <= ni < m and 0 <= nj < n and copy[ni][nj] == 1:
                    uf.union(i * n + j, ni * n + nj)

            # 计算新增的稳定砖块数量
            current_size = self.get_stable_bricks_count(uf, size)
            result[idx] = max(0, current_size - origin_size - 1)

        return result

    def get_stable_bricks_count(self, uf: UnionFind, top: int) -> int:
        """获取连接到顶部的砖块数量"""
        # 这里简化实现，实际需要维护每个连通分量的大小
        # 为了简化，我们假设 UnionFind 类有 get_size 方法
        try:
            return uf.get_component_size(top)
        except:
            # 简化实现：统计所有与顶部连通的节点
            count = 0
            for i in range(len(uf.parent)):
                if uf.find(i) == uf.find(top):
                    count += 1
            return count

class TestUnionFind(unittest.TestCase):
    """并查集测试用例"""

    def test_basic_union_find(self):
        uf = UnionFind(5)
        self.assertEqual(uf.get_count(), 5)

        uf.union(0, 1)
        self.assertEqual(uf.get_count(), 4)
        self.assertTrue(uf.is_connected(0, 1))

        uf.union(2, 3)
        uf.union(1, 2)
        self.assertEqual(uf.get_count(), 2)
        self.assertTrue(uf.is_connected(0, 3))

    def test_leetcode_128(self):
```

```
    sol = LeetCode128()

    # 测试用例 1
    nums1 = [100, 4, 200, 1, 3, 2]
    self.assertEqual(sol.longestConsecutive(nums1), 4)
    self.assertEqual(sol.longestConsecutiveHashSet(nums1), 4)

    # 测试用例 2
    nums2 = [0, 3, 7, 2, 5, 8, 4, 6, 0, 1]
    self.assertEqual(sol.longestConsecutive(nums2), 9)
    self.assertEqual(sol.longestConsecutiveHashSet(nums2), 9)

def test_leetcode_547(self):
    sol = LeetCode547()

    # 测试用例 1
    isConnected1 = [[1,1,0],[1,1,0],[0,0,1]]
    self.assertEqual(sol.findCircleNum(isConnected1), 2)
    self.assertEqual(sol.findCircleNumDFS(isConnected1), 2)

    # 测试用例 2
    isConnected2 = [[1,0,0],[0,1,0],[0,0,1]]
    self.assertEqual(sol.findCircleNum(isConnected2), 3)
    self.assertEqual(sol.findCircleNumDFS(isConnected2), 3)

def test_leetcode_684(self):
    sol = LeetCode684()

    edges1 = [[1,2],[1,3],[2,3]]
    self.assertEqual(sol.findRedundantConnection(edges1), [2,3])

    edges2 = [[1,2],[2,3],[3,4],[1,4],[1,5]]
    self.assertEqual(sol.findRedundantConnection(edges2), [1,4])

def test_weighted_union_find(self):
    uf = WeightedUnionFind(4)

    # 测试除法关系: a/b=2, b/c=3 => a/c=6
    uf.union(0, 1, 2.0)  # a/b = 2
    uf.union(1, 2, 3.0)  # b/c = 3

    result = uf.query(0, 2)  # a/c = ?
    self.assertAlmostEqual(result, 6.0, places=6)
```

```
def performance_analysis():
    """性能分析示例"""
    import time
    import random

    print("=== 并查集性能分析 ===")

    # 测试不同规模数据的性能
    sizes = [1000, 10000, 100000]

    for size in sizes:
        # 生成随机操作序列
        operations = [(random.randint(0, 1),
                       random.randint(0, size-1),
                       random.randint(0, size-1))
                      for _ in range(size * 2)]

        # 测试标准并查集
        start_time = time.time()
        uf = UnionFind(size)

        for op_type, x, y in operations:
            if op_type == 0:
                uf.union(x, y)
            else:
                uf.is_connected(x, y)

        end_time = time.time()
        print(f"规模 {size}: {end_time - start_time:.4f} 秒")

if __name__ == "__main__":
    # 运行性能分析
    performance_analysis()

    # 运行单元测试
    unittest.main(argv=[''], exit=False)

    print("\n=== Python 并查集实现总结 ===")
    print("1. 标准并查集: 路径压缩 + 按秩合并")
    print("2. 带权并查集: 支持关系维护")
    print("3. 逆向并查集: 处理删除操作")
    print("4. 多解法对比: 提供最优解和替代方案")
```

```
    print("5. 完整测试: 包含单元测试和性能分析")

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

文件: Code24_UnionFindCppImplementations.cpp
==========================================
/**
 * 并查集 C++实现合集
 * 包含各种并查集变种的 C++实现和测试
 */

#include <iostream>
#include <vector>
#include <unordered_map>
#include <unordered_set>
#include <algorithm>
#include <queue>
#include <stack>
#include <cmath>
#include <limits>
#include <string>
#include <sstream>
#include <chrono>
#include <tuple>
using namespace std;

/**
 * 标准并查集实现（路径压缩 + 按秩合并）
 */
class UnionFind {
private:
    vector<int> parent;
    vector<int> rank;
    int count;

public:
    UnionFind(int n) : parent(n), rank(n, 1), count(n) {
        for (int i = 0; i < n; i++) {
            parent[i] = i;
        }
    }

    int find(int x) {
```

```
        if (parent[x] != x) {
            parent[x] = find(parent[x]);  // 路径压缩
        }
        return parent[x];
    }

    bool unionSets(int x, int y) {
        int rootX = find(x);
        int rootY = find(y);

        if (rootX == rootY) {
            return false;
        }

        // 按秩合并
        if (rank[rootX] < rank[rootY]) {
            parent[rootX] = rootY;
        } else if (rank[rootX] > rank[rootY]) {
            parent[rootY] = rootX;
        } else {
            parent[rootY] = rootX;
            rank[rootX]++;
        }

        count--;
        return true;
    }

    bool isConnected(int x, int y) {
        return find(x) == find(y);
    }

    int getCount() {
        return count;
    }
};

/**
 * 带权并查集实现
 */
class WeightedUnionFind {
private:
    vector<int> parent;
```

```
    vector<double> weight;  // weight[i] 表示 i / parent[i]

public:
    WeightedUnionFind(int n) : parent(n), weight(n, 1.0) {
        for (int i = 0; i < n; i++) {
            parent[i] = i;
        }
    }

    pair<int, double> find(int x) {
        if (parent[x] != x) {
            auto [root, rootWeight] = find(parent[x]);
            parent[x] = root;
            weight[x] *= rootWeight;
            return {root, weight[x]};
        }
        return {x, 1.0};
    }

    bool unionSets(int x, int y, double value) {
        auto [rootX, weightX] = find(x);
        auto [rootY, weightY] = find(y);

        if (rootX == rootY) {
            // 检查是否冲突
            return abs(weightX / weightY - value) < 1e-9;
        }

        parent[rootX] = rootY;
        weight[rootX] = value * weightY / weightX;
        return true;
    }

    double query(int x, int y) {
        auto [rootX, weightX] = find(x);
        auto [rootY, weightY] = find(y);

        if (rootX != rootY) {
            return -1.0;  // 不连通
        }
        return weightX / weightY;
    }
};
```

```cpp
/**
 * LeetCode 128. 最长连续序列
 */
class Solution128 {
public:
    int longestConsecutive(vector<int>& nums) {
        if (nums.empty()) return 0;

        // 方法 1: 使用并查集
        return longestConsecutiveUnionFind(nums);

        // 方法 2: 使用哈希表（最优解）
        // return longestConsecutiveHashSet(nums);
    }

private:
    int longestConsecutiveUnionFind(vector<int>& nums) {
        int n = nums.size();
        unordered_map<int, int> numToIndex;
        for (int i = 0; i < n; i++) {
            numToIndex[nums[i]] = i;
        }

        UnionFind uf(n);

        for (int i = 0; i < n; i++) {
            int num = nums[i];

            if (numToIndex.count(num - 1)) {
                uf.unionSets(i, numToIndex[num - 1]);
            }
            if (numToIndex.count(num + 1)) {
                uf.unionSets(i, numToIndex[num + 1]);
            }
        }

        // 统计每个连通分量的大小
        unordered_map<int, int> sizeMap;
        for (int i = 0; i < n; i++) {
            int root = uf.find(i);
            sizeMap[root]++;
        }
```

```
        int maxSize = 0;
        for (auto& [root, size] : sizeMap) {
            maxSize = max(maxSize, size);
        }

        return maxSize;
    }

    int longestConsecutiveHashSet(vector<int>& nums) {
        unordered_set<int> numSet(nums.begin(), nums.end());
        int longestStreak = 0;

        for (int num : numSet) {
            // 只有当num是序列起点时才查找
            if (!numSet.count(num - 1)) {
                int currentNum = num;
                int currentStreak = 1;

                while (numSet.count(currentNum + 1)) {
                    currentNum++;
                    currentStreak++;
                }

                longestStreak = max(longestStreak, currentStreak);
            }
        }

        return longestStreak;
    }
};

/**
 * LeetCode 547. 省份数量
 */
class Solution547 {
public:
    int findCircleNum(vector<vector<int>>& isConnected) {
        int n = isConnected.size();
        UnionFind uf(n);

        for (int i = 0; i < n; i++) {
            for (int j = i + 1; j < n; j++) {
```

```
                if (isConnected[i][j] == 1) {
                    uf.unionSets(i, j);
                }
            }
        }

        return uf.getCount();
    }
};

/**
 * LeetCode 684. 冗余连接
 */
class Solution684 {
public:
    vector<int> findRedundantConnection(vector<vector<int>>& edges) {
        int n = edges.size();
        UnionFind uf(n + 1);  // 节点从 1 开始编号

        for (auto& edge : edges) {
            int u = edge[0], v = edge[1];
            if (!uf.unionSets(u, v)) {
                return {u, v};
            }
        }

        return {};
    }
};

/**
 * LeetCode 399. 除法求值
 */
class Solution399 {
public:
    vector<double> calcEquation(vector<vector<string>>& equations,
                                vector<double>& values,
                                vector<vector<string>>& queries) {
        // 创建变量到索引的映射
        unordered_map<string, int> varToIndex;
        int index = 0;

        for (auto& eq : equations) {
```

```
            if (!varToIndex.count(eq[0])) {
                varToIndex[eq[0]] = index++;
            }
            if (!varToIndex.count(eq[1])) {
                varToIndex[eq[1]] = index++;
            }
        }

        WeightedUnionFind uf(index);

        // 构建并查集
        for (int i = 0; i < equations.size(); i++) {
            int a = varToIndex[equations[i][0]];
            int b = varToIndex[equations[i][1]];
            uf.unionSets(a, b, values[i]);
        }

        // 处理查询
        vector<double> results;
        for (auto& query : queries) {
            string a = query[0], b = query[1];

            if (!varToIndex.count(a) || !varToIndex.count(b)) {
                results.push_back(-1.0);
            } else {
                double result = uf.query(varToIndex[a], varToIndex[b]);
                results.push_back(result);
            }
        }

        return results;
    }
};

/**
 * LeetCode 803. 打砖块
 */
class Solution803 {
public:
    vector<int> hitBricks(vector<vector<int>>& grid, vector<vector<int>>& hits) {
        int m = grid.size(), n = grid[0].size();

        // 复制网格
```

```
vector<vector<int>> copy = grid;

// 先消除所有要打的砖块
for (auto& hit : hits) {
    copy[hit[0]][hit[1]] = 0;
}

// 初始化并查集（增加虚拟顶部节点）
int size = m * n;
UnionFind uf(size + 1);

// 将第一行砖块连接到虚拟顶部节点
for (int j = 0; j < n; j++) {
    if (copy[0][j] == 1) {
        uf.unionSets(j, size);
    }
}

// 构建初始连通性
vector<pair<int, int>> directions = {{-1, 0}, {1, 0}, {0, -1}, {0, 1}};

for (int i = 1; i < m; i++) {
    for (int j = 0; j < n; j++) {
        if (copy[i][j] == 1) {
            int idx = i * n + j;

            // 连接上方砖块
            if (copy[i-1][j] == 1) {
                uf.unionSets(idx, (i-1) * n + j);
            }

            // 连接左方砖块
            if (j > 0 && copy[i][j-1] == 1) {
                uf.unionSets(idx, i * n + j - 1);
            }
        }
    }
}

// 逆向处理 hits
vector<int> result(hits.size(), 0);

for (int idx = hits.size() - 1; idx >= 0; idx--) {
```

```
            int i = hits[idx][0], j = hits[idx][1];

            // 如果原始位置没有砖块
            if (grid[i][j] == 0) {
                result[idx] = 0;
                continue;
            }

            // 记录添加前的稳定砖块数量
            int originSize = getStableBricksCount(uf, size);

            // 恢复砖块
            copy[i][j] = 1;
            int currentIdx = i * n + j;

            // 如果是第一行，连接到顶部
            if (i == 0) {
                uf.unionSets(j, size);
            }

            // 连接相邻砖块
            for (auto& [di, dj] : directions) {
                int ni = i + di, nj = j + dj;
                if (ni >= 0 && ni < m && nj >= 0 && nj < n && copy[ni][nj] == 1) {
                    uf.unionSets(currentIdx, ni * n + nj);
                }
            }

            // 计算新增的稳定砖块数量
            int currentSize = getStableBricksCount(uf, size);
            result[idx] = max(0, currentSize - originSize - 1);
        }

        return result;
    }

private:
    int getStableBricksCount(UnionFind& uf, int top) {
        // 简化实现：统计所有与顶部连通的节点
        int count = 0;
        int n = uf.getCount() - 1;  // 减去虚拟顶部节点

        for (int i = 0; i < n; i++) {
```

```
            if (uf.isConnected(i, top)) {
                count++;
            }
        }

        return count;
    }
};

/**
 * 性能分析工具
 */
class PerformanceAnalyzer {
public:
    static void analyzeUnionFindPerformance() {
        cout << "=== 并查集性能分析 ===" << endl;

        vector<int> sizes = {1000, 10000, 100000};

        for (int size : sizes) {
            // 生成随机操作序列
            vector<tuple<int, int, int>> operations;
            operations.reserve(size * 2);

            srand(time(nullptr));
            for (int i = 0; i < size * 2; i++) {
                int opType = rand() % 2;
                int x = rand() % size;
                int y = rand() % size;
                operations.emplace_back(opType, x, y);
            }

            // 测试标准并查集
            auto start = chrono::high_resolution_clock::now();

            UnionFind uf(size);
            for (auto& [opType, x, y] : operations) {
                if (opType == 0) {
                    uf.unionSets(x, y);
                } else {
                    uf.isConnected(x, y);
                }
            }
```

```
            auto end = chrono::high_resolution_clock::now();
            auto duration = chrono::duration_cast<chrono::milliseconds>(end - start);

            cout << "规模 " << size << ": " << duration.count() << " ms" << endl;
        }
    }
};

/**
 * 测试函数
 */
void runTests() {
    cout << "=== 并查集 C++实现测试 ===" << endl;

    // 测试 LeetCode 128
    {
        Solution128 sol;
        vector<int> nums1 = {100, 4, 200, 1, 3, 2};
        cout << "LeetCode 128 测试用例 1: " << sol.longestConsecutive(nums1) << " (预期: 4)" <<
endl;

        vector<int> nums2 = {0, 3, 7, 2, 5, 8, 4, 6, 0, 1};
        cout << "LeetCode 128 测试用例 2: " << sol.longestConsecutive(nums2) << " (预期: 9)" <<
endl;
    }

    // 测试 LeetCode 547
    {
        Solution547 sol;
        vector<vector<int>> isConnected1 = {{1,1,0}, {1,1,0}, {0,0,1}};
        cout << "LeetCode 547 测试用例 1: " << sol.findCircleNum(isConnected1) << " (预期: 2)" <<
endl;

        vector<vector<int>> isConnected2 = {{1,0,0}, {0,1,0}, {0,0,1}};
        cout << "LeetCode 547 测试用例 2: " << sol.findCircleNum(isConnected2) << " (预期: 3)" <<
endl;
    }

    // 测试 LeetCode 684
    {
        Solution684 sol;
        vector<vector<int>> edges1 = {{1,2}, {1,3}, {2,3}};
```

```
        auto result1 = sol.findRedundantConnection(edges1);
        cout << "LeetCode 684 测试用例 1: [" << result1[0] << "," << result1[1] << "] (预期: [2,3])" << endl;
    }

    // 测试性能分析
    PerformanceAnalyzer::analyzeUnionFindPerformance();
}

int main() {
    runTests();

    cout << "\n=== C++并查集实现总结 ===" << endl;
    cout << "1. 标准模板: 路径压缩 + 按秩合并" << endl;
    cout << "2. 内存优化: 使用 vector 代替 unordered_map" << endl;
    cout << "3. 性能优化: 避免不必要的复制操作" << endl;
    cout << "4. 类型安全: 使用强类型和 RAII" << endl;
    cout << "5. 测试完整: 包含单元测试和性能分析" << endl;

    return 0;
}
```

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

文件: Code25_UnionFindAdvancedApplications.java

```
================================================
package class056;

import java.util.*;

/**
 * 并查集高级应用与扩展
 * 包含并查集在各种复杂场景下的应用和优化
 */
public class Code25_UnionFindAdvancedApplications {

    /**
     * 1. 可持久化并查集（支持历史版本查询）
     * 使用版本控制技术实现可撤销操作
     */
    static class PersistentUnionFind {
        private int[] parent;
        private int[] rank;
```

```
    private List<int[]> history; // 存储操作历史：[时间戳, 操作类型, 参数...]
    private int currentTime;

    public PersistentUnionFind(int n) {
        parent = new int[n];
        rank = new int[n];
        history = new ArrayList<>();
        currentTime = 0;

        for (int i = 0; i < n; i++) {
            parent[i] = i;
            rank[i] = 1;
        }
    }

    public int find(int x, int time) {
        // 在指定时间版本中查找
        restoreState(time);
        return findCurrent(x);
    }

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

    public void union(int x, int y) {
        int rootX = findCurrent(x);
        int rootY = findCurrent(y);

        if (rootX != rootY) {
            // 记录操作历史
            if (rank[rootX] < rank[rootY]) {
                parent[rootX] = rootY;
                history.add(new int[]{currentTime, 0, rootX, rootY, rank[rootX],
rank[rootY]});
            } else if (rank[rootX] > rank[rootY]) {
                parent[rootY] = rootX;
                history.add(new int[]{currentTime, 0, rootY, rootX, rank[rootY],
rank[rootX]});
            } else {
```

```
                parent[rootY] = rootX;
                rank[rootX]++;
                history.add(new int[]{currentTime, 0, rootY, rootX, rank[rootY], rank[rootX]
- 1});
            }
            currentTime++;
        }
    }

    private void restoreState(int targetTime) {
        // 恢复到指定时间版本
        // 实际实现需要更复杂的状态管理
        // 这里简化实现，只支持向前恢复
    }

    public int getCurrentTime() {
        return currentTime;
    }
}

/**
 * 2. 动态连通性处理（支持在线查询和修改）
 */
static class DynamicConnectivity {
    private UnionFind uf;
    private Map<Integer, Set<Integer>> graph; // 邻接表表示
    private int componentCount;

    public DynamicConnectivity(int n) {
        uf = new UnionFind(n);
        graph = new HashMap<>();
        componentCount = n;

        for (int i = 0; i < n; i++) {
            graph.put(i, new HashSet<>());
        }
    }

    public void addEdge(int u, int v) {
        if (graph.get(u).contains(v)) {
            return; // 边已存在
        }
```

```
        graph.get(u).add(v);
        graph.get(v).add(u);

        if (uf.union(u, v)) {
            componentCount--;
        }
    }

    public void removeEdge(int u, int v) {
        if (!graph.get(u).contains(v)) {
            return; // 边不存在
        }

        graph.get(u).remove(v);
        graph.get(v).remove(u);

        // 重新计算连通性（简化实现，实际需要更高效算法）
        recalculateConnectivity();
    }

    public boolean isConnected(int u, int v) {
        return uf.isConnected(u, v);
    }

    public int getComponentCount() {
        return componentCount;
    }

    private void recalculateConnectivity() {
        // 重新计算连通性（BFS/DFS）
        int n = uf.parent.length;
        boolean[] visited = new boolean[n];
        int[] newParent = new int[n];
        Arrays.fill(newParent, -1);

        componentCount = 0;

        for (int i = 0; i < n; i++) {
            if (!visited[i]) {
                componentCount++;
                bfs(i, visited, newParent, graph);
            }
        }
```

```
            // 更新并查集（简化实现）
            uf = new UnionFind(n);
            // 实际需要更复杂的更新逻辑
        }

        private void bfs(int start, boolean[] visited, int[] parent, Map<Integer, Set<Integer>>
graph) {
            Queue<Integer> queue = new LinkedList<>();
            queue.offer(start);
            visited[start] = true;
            parent[start] = start;

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

                for (int v : graph.get(u)) {
                    if (!visited[v]) {
                        visited[v] = true;
                        parent[v] = start;
                        queue.offer(v);
                    }
                }
            }
        }
    }

    /**
     * 3. 并行并查集（多线程安全版本）
     */
    static class ConcurrentUnionFind {
        private final int[] parent;
        private final int[] rank;
        private final Object[] locks; // 分段锁

        public ConcurrentUnionFind(int n) {
            parent = new int[n];
            rank = new int[n];
            locks = new Object[n];

            for (int i = 0; i < n; i++) {
                parent[i] = i;
                rank[i] = 1;
```

```
            locks[i] = new Object();
        }
    }

    public int find(int x) {
        // 使用锁保证线程安全
        synchronized (locks[x]) {
            if (parent[x] != x) {
                parent[x] = find(parent[x]);
            }
            return parent[x];
        }
    }

    public boolean union(int x, int y) {
        // 按顺序获取锁，避免死锁
        if (x < y) {
            synchronized (locks[x]) {
                synchronized (locks[y]) {
                    return doUnion(x, y);
                }
            }
        } else {
            synchronized (locks[y]) {
                synchronized (locks[x]) {
                    return doUnion(x, y);
                }
            }
        }
    }

    private boolean doUnion(int x, int y) {
        int rootX = find(x);
        int rootY = find(y);

        if (rootX == rootY) {
            return false;
        }

        if (rank[rootX] < rank[rootY]) {
            parent[rootX] = rootY;
        } else if (rank[rootX] > rank[rootY]) {
            parent[rootY] = rootX;
```

```
            } else {
                parent[rootY] = rootX;
                rank[rootX]++;
            }

            return true;
        }
    }

    /**
     * 4. 机器学习中的并查集应用：聚类分析
     */
    static class ClusteringAnalysis {
        /**
         * 使用并查集进行层次聚类
         */
        public static List<Set<Integer>> hierarchicalClustering(double[][] data, double
threshold) {
            int n = data.length;
            UnionFind uf = new UnionFind(n);

            // 计算所有点对之间的距离
            List<Edge> edges = new ArrayList<>();
            for (int i = 0; i < n; i++) {
                for (int j = i + 1; j < n; j++) {
                    double distance = euclideanDistance(data[i], data[j]);
                    if (distance <= threshold) {
                        edges.add(new Edge(i, j, distance));
                    }
                }
            }

            // 按距离排序（单链接聚类）
            edges.sort(Comparator.comparingDouble(e -> e.distance));

            // 合并聚类
            for (Edge edge : edges) {
                uf.union(edge.u, edge.v);
            }

            // 提取聚类结果
            Map<Integer, Set<Integer>> clusters = new HashMap<>();
            for (int i = 0; i < n; i++) {
```

```
            int root = uf.find(i);
            clusters.computeIfAbsent(root, k -> new HashSet<>()).add(i);
        }

        return new ArrayList<>(clusters.values());
    }

    private static double euclideanDistance(double[] a, double[] b) {
        double sum = 0.0;
        for (int i = 0; i < a.length; i++) {
            double diff = a[i] - b[i];
            sum += diff * diff;
        }
        return Math.sqrt(sum);
    }

    static class Edge {
        int u, v;
        double distance;

        Edge(int u, int v, double distance) {
            this.u = u;
            this.v = v;
            this.distance = distance;
        }
    }
}

/**
 * 5. 图像处理中的连通组件标记
 */
static class ConnectedComponentLabeling {
    /**
     * 使用并查集进行连通组件标记（4 连通）
     */
    public static int[][] labelComponents(int[][] binaryImage) {
        int height = binaryImage.length;
        int width = binaryImage[0].length;

        UnionFind uf = new UnionFind(height * width);

        // 第一遍扫描：连接相邻像素
        for (int i = 0; i < height; i++) {
```

```
        for (int j = 0; j < width; j++) {
            if (binaryImage[i][j] == 1) {
                int currentIdx = i * width + j;

                // 检查上方像素
                if (i > 0 && binaryImage[i-1][j] == 1) {
                    uf.union(currentIdx, (i-1) * width + j);
                }

                // 检查左方像素
                if (j > 0 && binaryImage[i][j-1] == 1) {
                    uf.union(currentIdx, i * width + j - 1);
                }
            }
        }
    }

    // 第二遍扫描：分配标签
    int[][] labels = new int[height][width];
    Map<Integer, Integer> rootToLabel = new HashMap<>();
    int nextLabel = 1;

    for (int i = 0; i < height; i++) {
        for (int j = 0; j < width; j++) {
            if (binaryImage[i][j] == 1) {
                int root = uf.find(i * width + j);
                int label = rootToLabel.computeIfAbsent(root, k -> nextLabel++);
                labels[i][j] = label;
            }
        }
    }

    return labels;
}

/**
 * 8连通组件标记
 */
public static int[][] labelComponents8Connected(int[][] binaryImage) {
    int height = binaryImage.length;
    int width = binaryImage[0].length;

    UnionFind uf = new UnionFind(height * width);
```

```
        int[][] directions = {
            {-1, -1}, {-1, 0}, {-1, 1},  // 上方三个方向
            {0, -1}, {0, 1},             // 左右方向
            {1, -1}, {1, 0}, {1, 1}      // 下方三个方向
        };

        for (int i = 0; i < height; i++) {
            for (int j = 0; j < width; j++) {
                if (binaryImage[i][j] == 1) {
                    int currentIdx = i * width + j;

                    for (int[] dir : directions) {
                        int ni = i + dir[0];
                        int nj = j + dir[1];

                        if (ni >= 0 && ni < height && nj >= 0 && nj < width &&
                            binaryImage[ni][nj] == 1) {
                            uf.union(currentIdx, ni * width + nj);
                        }
                    }
                }
            }
        }

        return createLabelMap(binaryImage, uf, height, width);
    }

    private static int[][] createLabelMap(int[][] binaryImage, UnionFind uf, int height, int
width) {
        int[][] labels = new int[height][width];
        Map<Integer, Integer> rootToLabel = new HashMap<>();
        int nextLabel = 1;

        for (int i = 0; i < height; i++) {
            for (int j = 0; j < width; j++) {
                if (binaryImage[i][j] == 1) {
                    int root = uf.find(i * width + j);
                    int label = rootToLabel.computeIfAbsent(root, k -> nextLabel++);
                    labels[i][j] = label;
                }
            }
        }
```

```
            return labels;
        }
    }

    /**
     * 6. 社交网络分析：社区发现
     */
    static class CommunityDetection {
        /**
         * 使用并查集进行简单的社区发现
         */
        public static Map<Integer, Set<Integer>> detectCommunities(int[][] edges, int n) {
            UnionFind uf = new UnionFind(n);

            // 建立连接关系
            for (int[] edge : edges) {
                uf.union(edge[0], edge[1]);
            }

            // 提取社区
            Map<Integer, Set<Integer>> communities = new HashMap<>();
            for (int i = 0; i < n; i++) {
                int root = uf.find(i);
                communities.computeIfAbsent(root, k -> new HashSet<>()).add(i);
            }

            return communities;
        }

        /**
         * 基于模块度优化的社区发现（简化版）
         */
        public static Map<Integer, Set<Integer>> detectCommunitiesWithModularity(int[][] edges,
int n) {
            // 计算度中心性
            int[] degree = new int[n];
            for (int[] edge : edges) {
                degree[edge[0]]++;
                degree[edge[1]]++;
            }

            // 按度排序边（启发式策略）
            List<Edge> edgeList = new ArrayList<>();
```

```
        for (int[] edge : edges) {
            edgeList.add(new Edge(edge[0], edge[1],
                Math.min(degree[edge[0]], degree[edge[1]])));
        }

        edgeList.sort((a, b) -> Integer.compare(b.weight, a.weight)); // 降序

        UnionFind uf = new UnionFind(n);

        // 合并社区
        for (Edge edge : edgeList) {
            uf.union(edge.u, edge.v);
        }

        return extractCommunities(uf, n);
    }

    private static Map<Integer, Set<Integer>> extractCommunities(UnionFind uf, int n) {
        Map<Integer, Set<Integer>> communities = new HashMap<>();
        for (int i = 0; i < n; i++) {
            int root = uf.find(i);
            communities.computeIfAbsent(root, k -> new HashSet<>()).add(i);
        }
        return communities;
    }

    static class Edge {
        int u, v, weight;
        Edge(int u, int v, int weight) {
            this.u = u;
            this.v = v;
            this.weight = weight;
        }
    }
}

/**
 * 7. 并查集在编译器优化中的应用：变量别名分析
 */
static class AliasAnalysis {
    /**
     * 简单的变量别名分析使用并查集
     */
```

```
    public static Map<String, Set<String>> analyzeAliases(String[][] assignments) {
        // 创建变量到索引的映射
        Map<String, Integer> varToIndex = new HashMap<>();
        int index = 0;

        for (String[] assignment : assignments) {
            if (!varToIndex.containsKey(assignment[0])) {
                varToIndex.put(assignment[0], index++);
            }
            if (!varToIndex.containsKey(assignment[1])) {
                varToIndex.put(assignment[1], index++);
            }
        }

        UnionFind uf = new UnionFind(index);

        // 处理赋值语句
        for (String[] assignment : assignments) {
            int idx1 = varToIndex.get(assignment[0]);
            int idx2 = varToIndex.get(assignment[1]);
            uf.union(idx1, idx2);
        }

        // 提取别名集合
        Map<Integer, Set<String>> aliasGroups = new HashMap<>();
        for (Map.Entry<String, Integer> entry : varToIndex.entrySet()) {
            int root = uf.find(entry.getValue());
            aliasGroups.computeIfAbsent(root, k -> new HashSet<>()).add(entry.getKey());
        }

        Map<String, Set<String>> result = new HashMap<>();
        for (Set<String> aliasSet : aliasGroups.values()) {
            for (String var : aliasSet) {
                result.put(var, new HashSet<>(aliasSet));
            }
        }

        return result;
    }
}

/**
 * 标准并查集实现（用于上述应用）
```

```
 */
static class UnionFind {
    int[] parent;
    int[] rank;

    public UnionFind(int n) {
        parent = new int[n];
        rank = new int[n];
        for (int i = 0; i < n; i++) {
            parent[i] = i;
            rank[i] = 1;
        }
    }

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

    public boolean union(int x, int y) {
        int rootX = find(x);
        int rootY = find(y);

        if (rootX == rootY) {
            return false;
        }

        if (rank[rootX] < rank[rootY]) {
            parent[rootX] = rootY;
        } else if (rank[rootX] > rank[rootY]) {
            parent[rootY] = rootX;
        } else {
            parent[rootY] = rootX;
            rank[rootX]++;
        }

        return true;
    }

    public boolean isConnected(int x, int y) {
        return find(x) == find(y);
```

```
    }
}

// 测试方法
public static void main(String[] args) {
    System.out.println("=== 并查集高级应用测试 ===");

    // 测试聚类分析
    testClusteringAnalysis();

    // 测试连通组件标记
    testConnectedComponentLabeling();

    // 测试社区发现
    testCommunityDetection();

    // 测试别名分析
    testAliasAnalysis();

    System.out.println("所有测试完成！");
}

private static void testClusteringAnalysis() {
    System.out.println("\n1. 聚类分析测试:");
    double[][] data = {
        {1, 1}, {1, 2}, {8, 8}, {8, 9}, {15, 15}
    };
    List<Set<Integer>> clusters = ClusteringAnalysis.hierarchicalClustering(data, 3.0);
    System.out.println("聚类结果: " + clusters);
}

private static void testConnectedComponentLabeling() {
    System.out.println("\n2. 连通组件标记测试:");
    int[][] binaryImage = {
        {1, 1, 0, 0, 0},
        {1, 1, 0, 1, 1},
        {0, 0, 0, 1, 1},
        {0, 0, 0, 1, 1}
    };
    int[][] labels = ConnectedComponentLabeling.labelComponents(binaryImage);
    System.out.println("4 连通标记结果:");
    for (int[] row : labels) {
        System.out.println(Arrays.toString(row));
```

```
        }
    }

    private static void testCommunityDetection() {
        System.out.println("\n3. 社区发现测试:");
        int[][] edges = {
            {0, 1}, {1, 2}, {2, 0},  // 第一个社区
            {3, 4}, {4, 5}, {5, 3},  // 第二个社区
            {6, 7}                    // 第三个社区
        };
        Map<Integer, Set<Integer>> communities = CommunityDetection.detectCommunities(edges, 8);
        System.out.println("社区发现结果: " + communities);
    }

    private static void testAliasAnalysis() {
        System.out.println("\n4. 别名分析测试:");
        String[][] assignments = {
            {"a", "b"}, {"b", "c"}, {"x", "y"}
        };
        Map<String, Set<String>> aliases = AliasAnalysis.analyzeAliases(assignments);
        System.out.println("别名分析结果: " + aliases);
    }

    /**
     * 工程化考量总结:
     * 1. 性能优化: 根据具体场景选择合适的并查集变种
     * 2. 内存管理: 对于大规模数据，考虑使用更紧凑的数据结构
     * 3. 并发安全: 多线程环境下需要额外的同步机制
     * 4. 可扩展性: 设计支持多种应用场景的通用接口
     * 5. 测试覆盖: 确保各种边界情况得到充分测试
     */
}

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