

=====
文件夹: class112_UnionFindAlgorithms
=====

[Markdown 文件]
=====

文件: ADDITIONAL_PROBLEMS.md
=====

补充题目与训练

可持久化并查集

1. 洛谷 P3402 - 可持久化并查集（模板题）

- **链接**: <https://www.luogu.com.cn/problem/P3402>
- **题目大意**: 实现支持版本回退的并查集，支持合并、查询和回退操作
- **解题思路**: 使用主席树维护可持久化数组，实现可持久化并查集
- **时间复杂度**: $O(m \log^2 n)$
- **空间复杂度**: $O(n \log n)$
- **实现细节**:
 - 每个版本都保存父数组和秩数组的完整快照
 - 使用路径压缩会破坏版本独立性，只能使用按秩合并优化
 - 每次合并操作生成新节点，保持历史版本不变
 - 使用主席树实现可持久化数组，确保每次修改只复制必要的路径

2. NOI 2018 - 归程

- **链接**: <https://www.luogu.com.cn/problem/P4768>
- **题目大意**: 在一张图上进行多次询问，每次询问从某点开始，通过特定条件到达另一点的最短路径
- **解题思路**: 可以使用可持久化并查集维护不同条件下的连通性，结合最短路算法解决
- **时间复杂度**: $O((n + m) \log n)$
- **空间复杂度**: $O(n \log n)$
- **实现细节**:
 - 先计算每个点到终点的最短路
 - 按边的海拔排序，构建可持久化并查集
 - 对于每个查询，找到满足海拔条件的最新版本，在该版本中查询连通性

3. USACO 2018 Jan - MooTube

- **链接**: <https://www.luogu.com.cn/problem/P4185>
- **题目大意**: 在视频推荐系统中，根据相关性值查询两个视频是否相关
- **解题思路**: 可以使用可持久化并查集维护不同相关性阈值下的连通性
- **时间复杂度**: $O((n + m) \log n)$
- **空间复杂度**: $O(n \log n)$
- **实现细节**:
 - 将边按相关性值降序排序

- 构建可持久化并查集，每个版本对应不同的相关性阈值
- 查询时找到满足相关性要求的版本，检查连通性

4. BZOJ 3674 - 可持久化并查集加强版

- ****链接****: <https://www.lydsy.com/JudgeOnline/problem.php?id=3674>
- ****题目大意****: 实现支持版本回退的并查集，支持合并、查询和回退操作，但数据规模更大
- ****解题思路****: 使用主席树维护可持久化数组，实现可持久化并查集
- ****时间复杂度****: $O(m \log^2 n)$
- ****空间复杂度****: $O(n \log n)$
- ****实现细节****:
 - 与普通可持久化并查集实现类似，但需要优化空间使用
 - 使用路径压缩会影响其他版本的正确性，因此只能使用按秩合并
 - 父数组和秩数组都通过主席树实现持久化

5. HDU 6620 - Just an Old Puzzle

- ****链接****: <http://acm.hdu.edu.cn/showproblem.php?pid=6620>
- ****题目大意****: 通过交换操作还原一个拼图，需要判断是否可达
- ****解题思路****: 使用可持久化并查集维护拼图状态，可以回溯到之前的操作
- ****时间复杂度****: $O(n^2 \log n)$ ，空间 $O(n^2)$
- ****实现细节****:
 - 将拼图的每个位置视为图中的节点
 - 交换操作对应图中的边，使用可持久化并查集记录连通性变化
 - 通过版本回退检查不同状态的可达性

6. Codeforces 1401F - Reverse and Swap

- ****链接****: <https://codeforces.com/problemset/problem/1401/F>
- ****题目大意****: 支持区间反转和交换操作，查询区间第 k 大
- ****解题思路****: 使用可持久化线段树和可持久化并查集结合，处理复杂的区间操作
- ****时间复杂度****: $O(n \log^2 n)$ ，空间 $O(n \log n)$
- ****实现细节****:
 - 使用可持久化线段树维护区间值
 - 使用可持久化并查集维护元素的位置关系
 - 每次操作生成新版本，保证历史查询的正确性

可撤销并查集

1. AtCoder ABC302 H - Ball Collector

- ****链接****: https://atcoder.jp/contests/abc302/tasks/abc302_h
- ****题目大意****: 在一棵树上，每个节点有两个球，要求从根节点到每个节点的路径上收集球，使得收集的球编号各不相同
- ****解题思路****: 使用可撤销并查集维护连通性，在 DFS 过程中合并节点，回溯时撤销操作
- ****时间复杂度****: $O(n \log n)$
- ****空间复杂度****: $O(n)$

- ****实现细节****:
 - 每个球作为并查集中的一个节点
 - 对于每个节点，将其两个球合并到当前路径的集合中
 - 使用 DFS 遍历树，进入节点时执行合并，离开时执行撤销
 - 使用 edgeCnt 数组记录每个集合中的边数，当边数小于节点数时可以添加一个新球

2. Codeforces 891C - Envy

- ****链接****: <https://codeforces.com/problemset/problem/891/C>
- ****题目大意****: 给定一个图和一些边的集合，判断这些边是否可以同时出现在一个最小生成树中
- ****解题思路****: 使用可撤销并查集，按照 Kruskal 算法的思想，先加入权重小于当前查询边的边，然后尝试加入查询的边，如果会形成环则不能同时出现在 MST 中
- ****时间复杂度****: $O(m \log m + q * k * \log n)$
- ****空间复杂度****: $O(n + m)$
- ****实现细节****:
 - 将所有边按权值排序
 - 将查询按最大边权分组
 - 对于每组查询，先加入所有权值小于查询组最大边权的边
 - 对查询组内的边，尝试用可撤销并查集合并，如果有环则该查询不可行
 - 处理完查询后撤销合并操作

3. Codeforces 1681F - Unique Occurrences

- ****链接****: <https://codeforces.com/problemset/problem/1681/F>
- ****题目大意****: 在树上处理路径查询问题，统计某些路径上唯一出现的颜色数量
- ****解题思路****: 可以使用可撤销并查集维护路径的连通性信息
- ****时间复杂度****: $O(n \log n)$
- ****空间复杂度****: $O(n)$
- ****实现细节****:
 - 使用离线处理方法，将查询按右端点排序
 - 使用颜色首次出现的位置记录
 - 使用可撤销并查集维护区间内的连通性

4. Codeforces 1291F - Coffee Varieties (hard version)

- ****链接****: <https://codeforces.com/problemset/problem/1291/F>
- ****题目大意****: 交互题，需要通过特定操作识别咖啡品种
- ****解题思路****: 可以使用可撤销并查集维护品种的等价关系
- ****时间复杂度****: $O(n \log n)$
- ****空间复杂度****: $O(n)$
- ****实现细节****:
 - 使用可撤销并查集记录品种之间的等价关系
 - 根据交互结果动态调整等价关系
 - 利用可撤销操作回溯到之前的状态

5. AtCoder ABC126 F - XOR Matching

- ****链接****: https://atcoder.jp/contests/abc126/tasks/abc126_f
- ****题目大意****: 构造满足特定异或条件的序列
- ****解题思路****: 可以使用可撤销并查集处理异或关系
- ****时间复杂度****: $O(2^m)$
- ****空间复杂度****: $O(2^m)$
- ****实现细节****:
 - 使用线性基和可撤销并查集结合处理异或关系
 - 在尝试不同组合时使用可撤销操作回退状态

6. Codeforces 915F - Imbalance Value of a Tree

- ****链接****: <https://codeforces.com/problemset/problem/915/F>
- ****题目大意****: 计算树中所有路径的最大值与最小值之差的和
- ****解题思路****: 使用可撤销并查集，按边权排序后逐步合并，统计贡献
- ****时间复杂度****: $O(n \log n)$
- ****空间复杂度****: $O(n)$
- ****实现细节****:
 - 将边分别按权值升序和降序排序
 - 使用可撤销并查集统计不同权值范围内的连通块大小
 - 通过容斥原理计算所有路径的贡献

7. HDU 4496 - D-City

- ****链接****: <http://acm.hdu.edu.cn/showproblem.php?pid=4496>
- ****题目大意****: 逐步删除边，每次删除后询问连通块数量
- ****解题思路****: 离线处理，使用可撤销并查集处理删除操作
- ****时间复杂度****: $O(m \log n)$
- ****空间复杂度****: $O(n + m)$
- ****实现细节****:
 - 将所有删除操作离线处理，转化为逆序添加边的操作
 - 使用可撤销并查集维护连通块数量
 - 逆序处理所有操作，记录结果后再反转输出

扩展域并查集

1. Codeforces 1444C - Team Building

- ****链接****: <https://codeforces.com/problemset/problem/1444/C>
- ****题目大意****: 给定一些人和他们的组别，以及一些矛盾关系，判断两个组是否可以组成一个二分图
- ****解题思路****: 使用扩展域并查集，对于同一组内的矛盾关系，先判断该组是否能构成二分图，对于不同组之间的矛盾关系，使用可撤销并查集判断两个组合并后是否能构成二分图
- ****时间复杂度****: $O((m + k) * \log n)$
- ****空间复杂度****: $O(n)$
- ****实现细节****:
 - 对于每个组，使用二分图染色的扩展域并查集
 - 对于矛盾关系 (u, v) ，将 u 与 v 的敌人合并

- 使用可撤销并查集处理不同组之间的合并

2. 洛谷 P2024 - 食物链（经典种类并查集）

- ****链接****: <https://www.luogu.com.cn/problem/P2024>
- ****题目大意****: 动物有三种关系：同类、捕食、被捕食，根据一些描述判断哪些描述是假的
- ****解题思路****: 使用扩展域并查集，为每个动物创建 3 个节点分别表示其作为同类、捕食者、被捕食者的关系
- ****时间复杂度****: $O(n + m)$
- ****空间复杂度****: $O(n)$
- ****实现细节****:
 - 对于每个动物 x ，创建三个节点： x （同类）、 $x+n$ （捕食者）、 $x+2n$ （被捕食者）
 - 如果 x 和 y 是同类，则合并 x 与 y ， $x+n$ 与 $y+n$ ， $x+2n$ 与 $y+2n$
 - 如果 x 吃 y ，则合并 x 与 $y+n$ ， $x+n$ 与 $y+2n$ ， $x+2n$ 与 y
 - 每次操作前检查是否存在矛盾

3. HDU 3038 - How Many Answers Are Wrong

- ****链接****: <http://acm.hdu.edu.cn/showproblem.php?pid=3038>
- ****题目大意****: 给出一些区间和的描述，判断哪些描述是错误的
- ****解题思路****: 使用扩展域并查集维护前缀和关系，将区间和转化为前缀和的差
- ****时间复杂度****: $O(n + m)$
- ****空间复杂度****: $O(n)$
- ****实现细节****:
 - 使用带权并查集，权值表示从当前节点到根节点的和
 - 对于区间 $[l, r]$ 的和为 s ，转化为前缀和 $sum[r] - sum[l-1] = s$
 - 在合并时检查是否存在矛盾

4. POJ 1733 - Parity game

- ****链接****: <http://poj.org/problem?id=1733>
- ****题目大意****: 判断一个 01 序列的某些子区间的奇偶性描述是否一致
- ****解题思路****: 使用扩展域并查集维护前缀和的奇偶关系
- ****时间复杂度****: $O(n \log n)$
- ****空间复杂度****: $O(n)$
- ****实现细节****:
 - 使用带权并查集，权值表示前缀和的奇偶性
 - 对于区间 $[l, r]$ 有偶数个 1，转化为 $sum[r] \equiv sum[l-1] \pmod{2}$
 - 对于区间 $[l, r]$ 有奇数个 1，转化为 $sum[r] \equiv sum[l-1] + 1 \pmod{2}$

5. 洛谷 P1955 - 程序自动分析

- ****链接****: <https://www.luogu.com.cn/problem/P1955>
- ****题目大意****: 判断一些约束条件是否能被同时满足
- ****解题思路****: 使用扩展域并查集维护变量之间的相等关系
- ****时间复杂度****: $O(n \log n)$
- ****空间复杂度****: $O(n)$

- ****实现细节****:
 - 先处理所有相等约束，将相等的变量合并到同一集合
 - 再处理所有不等约束，检查是否存在矛盾
 - 使用离散化处理大范围的变量值

6. 洛谷 P1525 - 关押罪犯

- ****链接****: <https://www.luogu.com.cn/problem/P1525>
- ****题目大意****: 将罪犯分配到两个监狱，使得冲突值最大的一对罪犯的冲突值最小
- ****解题思路****: 使用扩展域并查集维护罪犯之间的敌对关系
- ****时间复杂度****: $O(n \log n)$
- ****空间复杂度****: $O(n)$
- ****实现细节****:
 - 将冲突按强度从大到小排序
 - 对于每个冲突 (u, v) ，检查 u 和 v 是否已经在同一个集合
 - 如果是，则这是当前最大的无法避免的冲突
 - 否则，将 u 与 v 的敌人合并， v 与 u 的敌人合并

7. LeetCode 721 - 账户合并

- ****链接****: <https://leetcode.cn/problems/accounts-merge/>
- ****题目大意****: 将具有相同邮箱的账户合并
- ****解题思路****: 使用扩展域并查集维护邮箱和账户之间的关系
- ****时间复杂度****: $O(n \log n)$
- ****空间复杂度****: $O(n)$
- ****实现细节****:
 - 将每个邮箱映射到唯一的 ID
 - 使用并查集合并同一账户的所有邮箱
 - 最后按账户分组收集所有邮箱

8. Codeforces 1380D - Berserk And Fireball

- ****链接****: <https://codeforces.com/problemset/problem/1380/D>
- ****题目大意****: 通过特定操作消除数组中的元素，求最小花费
- ****解题思路****: 使用扩展域并查集维护元素之间的关系
- ****时间复杂度****: $O(n \log n)$
- ****空间复杂度****: $O(n)$
- ****实现细节****:
 - 使用并查集维护保留的元素范围
 - 计算每个保留段的处理成本
 - 根据成本选择最优的消除方式

9. 洛谷 P2342 - 叠积木

- ****链接****: <https://www.luogu.com.cn/problem/P2342>
- ****题目大意****: 支持将一叠积木移到另一叠或查询某个积木上方有多少块
- ****解题思路****: 使用带权并查集维护积木堆的信息

- ****时间复杂度****: $O(n \alpha(n))$
- ****空间复杂度****: $O(n)$
- ****实现细节****:
 - 维护每个节点上方的积木数量
 - 维护每个堆的大小
 - 合并时更新这些信息

10. Codeforces 766C - Mahmoud and a Message

- ****链接****: <https://codeforces.com/problemset/problem/766/C>
- ****题目大意****: 将字符串分割成若干段，每段必须满足特定条件
- ****解题思路****: 使用扩展域并查集维护分割的合法性
- ****时间复杂度****: $O(n^2)$
- ****空间复杂度****: $O(n)$
- ****实现细节****:
 - 使用并查集记录可以合并的相邻字符
 - 根据每个字符的限制条件确定分割点

综合应用题目

1. 洛谷 P3674 - 小清新人渣的本愿

- ****链接****: <https://www.luogu.com.cn/problem/P3674>
- ****题目大意****: 在一个序列中查询区间内是否存在两个数的和或差为指定值
- ****解题思路****: 可以结合可持久化并查集维护区间信息
- ****时间复杂度****: $O(n \sqrt{n})$
- ****空间复杂度****: $O(n)$
- ****实现细节****:
 - 使用分块处理区间查询
 - 对于和查询，使用哈希表记录出现的数
 - 对于差查询，使用并查集维护数的关系

2. Codeforces 1095F - Make It Connected

- ****链接****: <https://codeforces.com/problemset/problem/1095/F>
- ****题目大意****: 连接所有节点，支持添加边和查询连通性
- ****解题思路****: 综合使用并查集的不同变体解决问题
- ****时间复杂度****: $O(n \log n)$
- ****空间复杂度****: $O(n)$
- ****实现细节****:
 - 使用并查集维护连通性
 - 采用贪心策略选择最小代价的连接方式
 - 结合优先队列优化连接过程

思路技巧总结

可持久化并查集的应用技巧

1. ****版本控制****: 通过保存每次操作的状态, 实现历史版本的查询和回退
2. ****时间线处理****: 将时间作为版本维度, 处理不同时间点的数据状态
3. ****增量更新****: 利用主席树实现只存储变化的部分, 节省空间
4. ****优化选择****: 必须使用按秩合并, 但不能使用路径压缩, 以保证版本独立性
5. ****离线预处理****: 对于复杂问题, 先进行离线预处理, 再构建可持久化并查集

可撤销并查集的应用技巧

1. ****DFS 回溯****: 在 DFS 遍历过程中使用并查集, 回溯时撤销合并操作
2. ****离线逆序处理****: 将删除操作转化为逆序添加操作, 使用可撤销并查集处理
3. ****分治算法****: 在分治过程中需要合并和撤销操作时使用
4. ****栈式保存****: 使用栈记录每次合并操作的详细信息, 包括父节点变化和秩变化
5. ****操作顺序****: 撤销操作必须严格按照合并的逆序进行

扩展域并查集的应用技巧

1. ****关系建模****: 将复杂关系转化为多个节点之间的合并关系
2. ****域的设计****: 根据关系类型设计适当的扩展域数量
3. ****约束转化****: 将问题约束转化为并查集的合并条件
4. ****带权并查集****: 使用权值记录节点间的具体关系值, 如距离、异或值等
5. ****矛盾检测****: 在合并前检查是否会产生矛盾

工程化考量

代码实现优化

1. ****数据结构选择****:
 - 大规模数据使用数组实现并查集以获得最佳性能
 - 小规模或动态数据可以使用哈希表实现
2. ****内存管理****:
 - 可持久化并查集需要合理估计空间使用, 避免内存溢出
 - 可撤销并查集的栈大小需要足够大以存储所有操作历史
3. ****性能优化****:
 - 基础并查集同时使用路径压缩和按秩合并
 - 可持久化和可撤销并查集只能使用按秩合并
 - 对于离散化处理, 使用高效的排序和去重算法

异常处理和鲁棒性

1. ****边界条件****:
 - 处理空输入、单节点等边界情况
 - 检查数组下标越界
2. ****错误检测****:

- 可撤销操作时检查栈是否为空
- 合并操作时检查是否会导致矛盾

3. ****非法输入****:

- 验证输入数据的合法性
- 处理超出范围的输入值

跨语言实现差异

1. ****Java 实现注意事项****:

- 数组大小需要在初始化时确定
- 需要注意整数溢出问题
- 递归深度有限，避免递归实现的路径压缩

2. ****C++实现注意事项****:

- 可以使用模板实现通用的并查集
- 内存管理更加灵活，但需要手动释放
- 可以使用 STL 容器如 vector、stack 等辅助实现

3. ****Python 实现注意事项****:

- 列表的访问速度较慢，大规模数据考虑使用 numpy
- 递归深度有限，需要使用非递归实现
- 字典可以用于动态大小的并查集实现

学习路径与进阶建议

基础阶段

1. ****掌握普通并查集****:

- 实现路径压缩和按秩合并
- 解决基础连通性问题
- 理解时间复杂度分析

2. ****扩展应用****:

- 最小生成树算法中的应用
- 图的连通性问题
- 集合操作相关问题

进阶阶段

1. ****学习三种高级并查集****:

- 理解每种变体的设计原理
- 掌握实现方法和技巧
- 分析时间和空间复杂度

2. ****解题练习****:

- 按类别刷题，从易到难
- 总结每种类型的解题模式
- 理解题目建模过程

高级阶段

1. **综合应用**:

- 学习并查集与其他数据结构的结合
- 解决复杂的算法问题
- 优化大规模数据的处理效率

2. **工程实践**:

- 将并查集应用到实际项目中
- 实现高效、可复用的并查集组件
- 处理真实数据中的边界情况

通过系统学习和大量实践，深入理解并查集的各种变体及其应用，是算法学习中的重要一步。希望这些题目和解析能够帮助你更好地掌握这一强大的数据结构！

=====

文件: README.md

=====

Class165 - 可持久化并查集、可撤销并查集和扩展域并查集

本目录包含以下几种高级并查集算法的实现，这些是普通并查集的扩展版本，用于解决更复杂的问题：

> 本文档详细介绍了三种高级并查集的原理、实现和应用场景，并提供了丰富的练习题和详细的代码解答。所有代码均提供 Java、C++和 Python 三种语言实现，并包含详细的注释和复杂度分析。

1. 可持久化并查集 (Persistent Union-Find)

算法原理

可持久化并查集是支持版本回退操作的并查集，通过使用主席树（可持久化线段树）来维护父节点数组和秩数组，实现对历史版本的访问。可持久化并查集允许我们访问数据结构的历史版本，而不仅仅是当前状态。

核心特点

1. 支持合并操作 (Union)
2. 支持查询操作 (Find)
3. 支持版本回退 (Version Rollback)
4. 不使用路径压缩，而是使用按秩合并来保证效率
5. 使用主席树维护可持久化数组

时间复杂度

- 合并操作: $O(\log^2 n)$
- 查询操作: $O(\log^2 n)$
- 空间复杂度: $O(n \log n)$

相关题目及详细解析

1. [洛谷 P3402 - 可持久化并查集] (<https://www.luogu.com.cn/problem/P3402>)

- **题目大意**: 实现支持版本回退的并查集, 支持合并、查询和回退操作
- **解题思路**: 使用主席树维护可持久化数组, 实现可持久化并查集
- **时间复杂度**: $O(m \log^2 n)$
- **空间复杂度**: $O(n \log n)$
- **实现细节**:
 - 使用主席树维护父节点数组和秩数组
 - 每个版本保存父数组和秩数组的根节点
 - 合并时生成新的版本, 查询时基于特定版本

2. [NOI 2018 - 归程] (<https://www.luogu.com.cn/problem/P4768>)

- **题目大意**: 在一张图上进行多次询问, 每次询问从某点开始, 通过特定条件到达另一点的最短路径
- **解题思路**: 可以使用可持久化并查集维护不同条件下的连通性, 结合最短路算法解决
- **时间复杂度**: $O((n + m) \log n)$
- **空间复杂度**: $O(n \log n)$
- **实现细节**:
 - 先用 Dijkstra 算法计算各点到终点的最短距离
 - 将边按权值降序排序
 - 使用可持久化并查集维护不同水位下的连通性
 - 对于每个查询, 找到最大的满足条件的水位, 查询连通性

3. [USACO 2018 Jan - MooTube] (<https://www.luogu.com.cn/problem/P4185>)

- **题目大意**: 在视频推荐系统中, 根据相关性值查询两个视频是否相关
- **解题思路**: 可以使用可持久化并查集维护不同相关性阈值下的连通性
- **时间复杂度**: $O((n + m) \log n)$
- **空间复杂度**: $O(n \log n)$
- **实现细节**:
 - 将边按相关性降序排序
 - 使用可持久化并查集维护不同阈值下的连通性
 - 对于每个查询, 找到对应的版本并查询连通性

4. [BZOJ 3674 - 可持久化并查集加强版] (<https://www.lydsy.com/JudgeOnline/problem.php?id=3674>)

- **题目大意**: 实现支持版本回退的并查集, 支持合并、查询和回退操作, 但数据规模更大
- **解题思路**: 使用主席树维护可持久化数组, 实现可持久化并查集
- **时间复杂度**: $O(m \log^2 n)$
- **空间复杂度**: $O(n \log n)$

5. [HDU 6620 - Just an Old Puzzle](http://acm.hdu.edu.cn/showproblem.php?pid=6620)

- **题目大意**: 判断一个数字拼图是否可以还原
- **解题思路**: 可以使用可持久化并查集维护拼图的状态变化
- **时间复杂度**: $O(n^2 \log n)$
- **空间复杂度**: $O(n^2 \log n)$

6. [Codeforces 1401F - Reverse and Swap](https://codeforces.com/problemset/problem/1401/F)

- **题目大意**: 支持反转和交换操作的数据结构问题
- **解题思路**: 可以使用可持久化并查集维护元素的位置关系
- **时间复杂度**: $O(n \log^2 n)$
- **空间复杂度**: $O(n \log n)$

2. 可撤销并查集 (Undo Union-Find)

算法原理

可撤销并查集支持撤销最近一次合并操作，通过记录每次合并时的状态变化，使用栈结构来实现撤销操作。这种数据结构在需要回溯状态的场景中非常有用，比如 DFS 搜索过程中的状态维护。

核心特点

1. 支持合并操作 (Union)
2. 支持查询操作 (Find)
3. 支持撤销操作 (Undo)
4. 不使用路径压缩，只使用按秩合并
5. 使用栈记录操作历史，支持精确回退

时间复杂度

- 合并操作: $O(\log n)$
- 查询操作: $O(\log n)$
- 撤销操作: $O(1)$
- 空间复杂度: $O(n)$

相关题目及详细解析

1. [AtCoder ABC302 H - Ball Collector](https://atcoder.jp/contests/abc302/tasks/abc302_h)

- **题目大意**: 在一棵树上，每个节点有两个球，要求从根节点到每个节点的路径上收集球，使得收集的球编号各不相同
- **解题思路**: 使用可撤销并查集维护连通性，在 DFS 过程中合并节点，回溯时撤销操作
- **时间复杂度**: $O(n \log n)$
- **空间复杂度**: $O(n)$
- **实现细节**:
 - 每个球作为并查集中的一个节点
 - 对于每个节点，将其两个球合并到当前路径的集合中
 - 使用 DFS 遍历树，进入节点时执行合并，离开时执行撤销

- 使用 `edgeCnt` 数组记录每个集合中的边数，当边数小于节点数时可以添加一个新球

2. [Codeforces 891C - Envy](https://codeforces.com/problemset/problem/891/C)

- **题目大意**：给定一个图和一些边的集合，判断这些边是否可以同时出现在一个最小生成树中
- **解题思路**：使用可撤销并查集，按照 Kruskal 算法的思想，先加入权重小于当前查询边的边，然后尝试加入查询的边，如果会形成环则不能同时出现在 MST 中
- **时间复杂度**： $O(m \log m + q * k * \log n)$
- **空间复杂度**： $O(n + m)$
- **实现细节**：
 - 将所有边按权值排序
 - 将查询按最大边权分组
 - 对于每组查询，先加入所有权值小于查询组最大边权的边
 - 对查询组内的边，尝试用可撤销并查集合并，如果有环则该查询不可行
 - 处理完查询后撤销合并操作

3. [Codeforces 1681F - Unique Occurrences](https://codeforces.com/problemset/problem/1681/F)

- **题目大意**：在树上处理路径查询问题，统计某些路径上唯一出现的颜色数量
- **解题思路**：可以使用可撤销并查集维护路径的连通性信息
- **时间复杂度**： $O(n \log n)$
- **空间复杂度**： $O(n)$
- **实现细节**：
 - 使用离线处理方法，将查询按右端点排序
 - 使用颜色首次出现的位置记录
 - 使用可撤销并查集维护区间内的连通性

4. [Codeforces 1291F - Coffee Varieties (hard version)](https://codeforces.com/problemset/problem/1291/F)

- **题目大意**：交互题，需要通过特定操作识别咖啡品种
- **解题思路**：可以使用可撤销并查集维护品种的等价关系
- **时间复杂度**： $O(n \log n)$
- **空间复杂度**： $O(n)$
- **实现细节**：
 - 使用可撤销并查集记录品种之间的等价关系
 - 根据交互结果动态调整等价关系
 - 利用可撤销操作回溯到之前的状态

5. [AtCoder ABC126 F - XOR Matching](https://atcoder.jp/contests/abc126/tasks/abc126_f)

- **题目大意**：构造满足特定异或条件的序列
- **解题思路**：可以使用可撤销并查集处理异或关系
- **时间复杂度**： $O(2^m)$
- **空间复杂度**： $O(2^m)$
- **实现细节**：
 - 使用可撤销并查集维护异或关系

- 通过枚举可能的异或值构造满足条件的序列

6. [Codeforces 1401F - Reverse and Swap](https://codeforces.com/problemset/problem/1401/F)

- **题目大意**: 支持反转和交换操作的数据结构问题
- **解题思路**: 可以结合可撤销并查集维护元素之间的关系
- **时间复杂度**: $O(n \log n)$
- **空间复杂度**: $O(n)$

7. [Codeforces 915F - Imbalance Value of a Tree](https://codeforces.com/problemset/problem/915/F)

- **题目大意**: 计算树中所有路径的最大值与最小值之差的和
- **解题思路**: 使用可撤销并查集, 按边权排序后逐步合并, 统计贡献
- **时间复杂度**: $O(n \log n)$
- **空间复杂度**: $O(n)$
- **实现细节**:
 - 将边分别按权值升序和降序排序
 - 使用可撤销并查集统计不同权值范围内的连通块大小
 - 通过容斥原理计算所有路径的贡献

3. 扩展域并查集 (Extended Union-Find)

算法原理

扩展域并查集通过扩展节点域来处理元素之间的复杂关系, 常用于解决种类并查集问题, 如食物链问题。它通过为每个元素创建多个节点来表示不同的关系状态, 从而能够处理更复杂的关系约束。

核心特点

1. 通过扩展节点域来表示不同种类的关系
2. 支持常规的合并和查询操作
3. 常用于解决敌我关系、食物链等复杂关系问题
4. 可以与可撤销并查集结合使用解决复杂问题

时间复杂度

- 合并操作: $O(\alpha(n))$, 其中 α 是阿克曼函数的反函数, 近似于常数
- 查询操作: $O(\alpha(n))$
- 空间复杂度: $O(n)$, 其中 n 是元素数量

相关题目及详细解析

1. [Codeforces 1444C - Team Building](https://codeforces.com/problemset/problem/1444/C)

- **题目大意**: 给定一些人和他们的组别, 以及一些矛盾关系, 判断两个组是否可以组成一个二分图
- **解题思路**: 使用扩展域并查集, 对于同一组内的矛盾关系, 先判断该组是否能构成二分图, 对于不同组之间的矛盾关系, 使用可撤销并查集判断两个组合并后是否能构成二分图
- **时间复杂度**: $O((m + k) * \log n)$

- **空间复杂度**: $O(n)$
- **实现细节**:
 - 对于每个组, 使用二分图染色的扩展域并查集
 - 对于矛盾关系 (u, v) , 将 u 与 v 的敌人合并
 - 使用可撤销并查集处理不同组之间的合并

2. [洛谷 P2024 - 食物链 (经典种类并查集)] (<https://www.luogu.com.cn/problem/P2024>)

- **题目大意**: 动物有三种关系: 同类、捕食、被捕食, 根据一些描述判断哪些描述是假的
- **解题思路**: 使用扩展域并查集, 为每个动物创建 3 个节点分别表示其作为同类、捕食者、被捕食者的关系
- **时间复杂度**: $O(n + m)$
- **空间复杂度**: $O(n)$
- **实现细节**:
 - 对于每个动物 x , 创建三个节点: x (同类)、 $x+n$ (捕食者)、 $x+2n$ (被捕食者)
 - 如果 x 和 y 是同类, 则合并 x 与 y , $x+n$ 与 $y+n$, $x+2n$ 与 $y+2n$
 - 如果 x 吃 y , 则合并 x 与 $y+n$, $x+n$ 与 $y+2n$, $x+2n$ 与 y
 - 每次操作前检查是否存在矛盾

3. [HDU 3038 - How Many Answers Are Wrong] (<http://acm.hdu.edu.cn/showproblem.php?pid=3038>)

- **题目大意**: 给出一些区间和的描述, 判断哪些描述是错误的
- **解题思路**: 使用扩展域并查集维护前缀和关系, 将区间和转化为前缀和的差
- **时间复杂度**: $O(n + m)$
- **空间复杂度**: $O(n)$
- **实现细节**:
 - 使用带权并查集, 权值表示从当前节点到根节点的和
 - 对于区间 $[1, r]$ 的和为 s , 转化为前缀和 $\text{sum}[r] - \text{sum}[1-1] = s$
 - 在合并时检查是否存在矛盾

4. [AtCoder ABC126 F - XOR Matching] (https://atcoder.jp/contests/abc126/tasks/abc126_f)

- **题目大意**: 构造满足特定异或条件的序列
- **解题思路**: 可以使用扩展域并查集处理异或关系
- **时间复杂度**: $O(2^m)$
- **空间复杂度**: $O(2^m)$
- **实现细节**:
 - 使用带权并查集维护异或关系
 - 权值表示两个节点之间的异或值
 - 通过枚举可能的异或值构造满足条件的序列

5. [洛谷 P1955 - 程序自动分析] (<https://www.luogu.com.cn/problem/P1955>)

- **题目大意**: 判断一些约束条件是否能被同时满足
- **解题思路**: 使用扩展域并查集维护变量之间的相等关系
- **时间复杂度**: $O(n \log n)$
- **空间复杂度**: $O(n)$

- ****实现细节****:
 - 先处理所有相等约束，将相等的变量合并
 - 再检查所有不等约束，判断是否存在矛盾

6. [POJ 1733 - Parity game](http://poj.org/problem?id=1733)

- ****题目大意****: 判断一个 01 序列的某些子区间的奇偶性描述是否一致
- ****解题思路****: 使用扩展域并查集维护前缀和的奇偶关系
- ****时间复杂度****: $O(n \log n)$
- ****空间复杂度****: $O(n)$
- ****实现细节****:
 - 使用带权并查集，权值表示前缀和的奇偶性
 - 对于区间 $[l, r]$ 有偶数个 1，转化为 $\text{sum}[r] \equiv \text{sum}[l-1] \pmod{2}$
 - 对于区间 $[l, r]$ 有奇数个 1，转化为 $\text{sum}[r] \equiv \text{sum}[l-1] + 1 \pmod{2}$

7. [洛谷 P1525 - 关押罪犯](https://www.luogu.com.cn/problem/P1525)

- ****题目大意****: 将罪犯分配到两个监狱，使得冲突值最大的一对罪犯的冲突值最小
- ****解题思路****: 使用扩展域并查集维护罪犯之间的敌对关系
- ****时间复杂度****: $O(n \log n)$
- ****空间复杂度****: $O(n)$
- ****实现细节****:
 - 将冲突按强度从大到小排序
 - 对于每个冲突 (u, v) ，检查 u 和 v 是否已经在同一个集合
 - 如果是，则这是当前最大的无法避免的冲突
 - 否则，将 u 与 v 的敌人合并， v 与 u 的敌人合并

8. [LeetCode 721 - 账户合并](https://leetcode.cn/problems/accounts-merge/)

- ****题目大意****: 将具有相同邮箱的账户合并
- ****解题思路****: 使用扩展域并查集维护邮箱和账户之间的关系
- ****时间复杂度****: $O(n \log n)$
- ****空间复杂度****: $O(n)$
- ****实现细节****:
 - 将每个邮箱映射到唯一的 ID
 - 使用并查集合并同一账户的所有邮箱
 - 最后按账户分组收集所有邮箱

9. [Codeforces 1380D - Berserk And

Fireball](https://codeforces.com/problemset/problem/1380/D)

- ****题目大意****: 通过特定操作将数组转换为目标数组，计算最小成本
- ****解题思路****: 使用扩展域并查集维护区间的连通性
- ****时间复杂度****: $O(n \log n)$
- ****空间复杂度****: $O(n)$
- ****实现细节****:
 - 识别需要保留的元素位置

- 使用并查集将连续的可操作区间合并
- 对每个区间计算最小操作成本

10. [HDU 4496 - D-City] (<http://acm.hdu.edu.cn/showproblem.php?pid=4496>)

- ****题目大意****: 逐步删除边，每次删除后查询连通块数量
- ****解题思路****: 使用扩展域并查集，反向处理问题
- ****时间复杂度****: $O(m \log n)$
- ****空间复杂度****: $O(n + m)$
- ****实现细节****:
 - 初始时所有边都被删除
 - 按删除顺序的逆序添加边
 - 使用并查集维护连通块数量

文件说明

- `Code01_PersistentUnionFind1.java` - 可持久化并查集的 Java 实现
- `Code01_PersistentUnionFind2.cpp` - 可持久化并查集的 C++实现
- `Code01_PersistentUnionFind3.py` - 可持久化并查集的 Python 实现
- `Code02_UndoUnionFind1.java` - 可撤销并查集的 Java 实现
- `Code02_UndoUnionFind2.cpp` - 可撤销并查集的 C++实现
- `Code02_UndoUnionFind3.py` - 可撤销并查集的 Python 实现
- `Code03_Envy1.java` - 使用可撤销并查集解决 Codeforces 891C 问题的 Java 实现
- `Code03_Envy2.cpp` - 使用可撤销并查集解决 Codeforces 891C 问题的 C++实现
- `Code03_Envy3.py` - 使用可撤销并查集解决 Codeforces 891C 问题的 Python 实现
- `Code04_TeamBuilding1.java` - 使用扩展域并查集解决 Codeforces 1444C 问题的 Java 实现
- `Code04_TeamBuilding2.cpp` - 使用扩展域并查集解决 Codeforces 1444C 问题的 C++实现
- `Code04_TeamBuilding3.py` - 使用扩展域并查集解决 Codeforces 1444C 问题的 Python 实现
- `Code05_FoodChain.java` - 使用扩展域并查集解决食物链问题的 Java 实现
- `Code05_FoodChain.cpp` - 使用扩展域并查集解决食物链问题的 C++实现
- `Code05_FoodChain.py` - 使用扩展域并查集解决食物链问题的 Python 实现
- `ADDITIONAL_PROBLEMS.md` - 补充题目列表
- `SUMMARY.md` - 三种并查集的详细对比与应用总结

算法技巧总结

可持久化并查集技巧

1. 使用主席树维护父节点数组和秩数组
2. 不使用路径压缩，使用按秩合并
3. 通过版本号管理历史状态
4. 适用于需要访问历史版本的场景

可撤销并查集技巧

1. 不使用路径压缩，只使用按秩合并

2. 用栈记录每次合并操作的关键信息
3. 撤销时恢复合并前的状态
4. 适用于需要回溯状态的场景，如 DFS 搜索

扩展域并查集技巧

1. 通过扩展节点域来表示不同种类的关系
2. 常用于处理敌我关系、食物链等问题
3. 可以与可撤销并查集结合使用解决复杂问题
4. 适用于需要维护多种关系状态的场景

工程化考量

1. ****异常处理****: 在实际应用中，应增加输入验证和异常处理
2. ****性能优化****: 合理使用按秩合并和路径压缩（在可持久化场景中不使用路径压缩）
3. ****内存管理****: 注意控制空间复杂度，避免内存溢出
4. ****代码复用****: 可以将并查集的通用操作封装成类或模块
5. ****版本控制****: 可持久化并查集需要良好的版本管理机制
6. ****状态恢复****: 可撤销并查集需要精确的状态恢复机制

=====

文件: SUMMARY.md

=====

可持久化并查集、可撤销并查集和扩展域并查集详细对比与应用

1. 概述

本文档详细介绍了三种高级并查集数据结构：可持久化并查集、可撤销并查集和扩展域并查集。这些数据结构是普通并查集的扩展版本，用于解决更复杂的问题。

2. 详细对比

2.1 可持久化并查集 (Persistent Union-Find)

核心思想

- 支持访问历史版本的数据结构
- 使用主席树（可持久化线段树）维护父节点数组和秩数组
- 不使用路径压缩，只使用按秩合并

特点

- 支持合并操作 (Union)
- 支持查询操作 (Find)
- 支持版本回退 (Version Rollback)

- 空间复杂度较高: $O(n \log n)$

时间复杂度

- 合并操作: $O(\log^2 n)$
- 查询操作: $O(\log^2 n)$
- 空间复杂度: $O(n \log n)$

适用场景

- 需要访问历史版本的场景
- 版本控制、回退操作
- 在线算法需要回溯到之前状态

2.2 可撤销并查集 (Undo Union-Find)

核心思想

- 支持撤销最近一次合并操作
- 使用栈结构记录每次合并时的状态变化
- 不使用路径压缩, 只使用按秩合并

特点

- 支持合并操作 (Union)
- 支持查询操作 (Find)
- 支持撤销操作 (Undo)
- 空间复杂度较低: $O(n)$

时间复杂度

- 合并操作: $O(\log n)$
- 查询操作: $O(\log n)$
- 撤销操作: $O(1)$
- 空间复杂度: $O(n)$

适用场景

- 需要撤销最近操作的场景
- DFS 过程中的状态维护
- 分治算法中需要回溯状态

2.3 扩展域并查集 (Extended Union-Find)

核心思想

- 通过扩展节点域来处理元素之间的复杂关系
- 为每个元素创建多个节点来表示不同的关系状态
- 常用于解决种类并查集问题

特点

- 通过扩展节点域来表示不同种类的关系
- 支持常规的合并和查询操作
- 可以与可撤销并查集结合使用解决复杂问题

时间复杂度

- 合并操作: $O(\alpha(n))$
- 查询操作: $O(\alpha(n))$
- 空间复杂度: $O(n)$

适用场景

- 处理复杂关系的场景
- 敌我关系、食物链等问题
- 需要维护多种关系状态的场景

3. 应用场景详细分析

3.1 可持久化并查集应用场景

模板题: 洛谷 P3402 - 可持久化并查集

- ****问题描述****: 实现支持版本回退的并查集, 支持合并、查询和回退操作
- ****解决方案****: 使用主席树维护可持久化数组, 实现可持久化并查集
- ****关键点****: 不使用路径压缩, 使用按秩合并保证效率

应用题: NOI 2018 - 归程

- ****问题描述****: 在一张图上进行多次询问, 每次询问从某点开始, 通过特定条件到达另一点的最短路径
- ****解决方案****: 使用可持久化并查集维护不同条件下的连通性, 结合最短路算法解决
- ****关键点****: 利用可持久化特性维护不同条件下的连通状态

3.2 可撤销并查集应用场景

模板题: AtCoder ABC302 H - Ball Collector

- ****问题描述****: 在一棵树上, 每个节点有两个球, 要求从根节点到每个节点的路径上收集球, 使得收集的球编号各不相同
- ****解决方案****: 使用可撤销并查集维护连通性, 在 DFS 过程中合并节点, 回溯时撤销操作
- ****关键点****: DFS 过程中的状态维护和撤销

应用题: Codeforces 891C - Envy

- ****问题描述****: 给定一个图和一些边的集合, 判断这些边是否可以同时出现在一个最小生成树中
- ****解决方案****: 使用可撤销并查集, 按照 Kruskal 算法的思想, 先加入权重小于当前查询边的边, 然后尝试加入查询的边, 如果会形成环则不能同时出现在 MST 中
- ****关键点****: 离线处理查询, 利用可撤销特性处理多个查询

3.3 扩展域并查集应用场景

经典题：洛谷 P2024 - 食物链

- **问题描述**：动物有三种关系：同类、捕食、被捕食，根据一些描述判断哪些描述是假的
- **解决方案**：使用扩展域并查集，为每个动物创建 3 个节点分别表示其作为同类、捕食者、被捕食者的关系
- **关键点**：通过扩展域表示不同种类的关系

应用题：Codeforces 1444C - Team Building

- **问题描述**：给定一些人和他们的组别，以及一些矛盾关系，判断两个组是否可以组成一个二分图
- **解决方案**：使用扩展域并查集，对于同一组内的矛盾关系，先判断该组是否能构成二分图，对于不同组之间的矛盾关系，使用可撤销并查集判断两个组合并后是否能构成二分图
- **关键点**：扩展域并查集与可撤销并查集的结合使用

4. 实现细节对比

特性	可持久化并查集	可撤销并查集	扩展域并查集
数据结构	主席树	普通数组+栈	扩展节点域
路径压缩	不使用	不使用	可使用
按秩合并	使用	使用	使用
版本回退	支持	不支持	不支持
操作撤销	不支持	支持	不支持
空间复杂度	$O(n \log n)$	$O(n)$	$O(n)$
查询复杂度	$O(\log^2 n)$	$O(\log n)$	$O(\alpha(n))$
合并复杂度	$O(\log^2 n)$	$O(\log n)$	$O(\alpha(n))$

5. 工程化考量

5.1 异常处理

- 输入参数校验
- 边界条件处理
- 内存使用监控

5.2 性能优化

- 合理选择合并策略
- 内存分配优化
- 避免不必要的操作

5.3 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

6. 学习建议

6.1 掌握基础

- 首先熟练掌握普通并查集的实现和应用
- 理解路径压缩和按秩合并原理

6.2 理解原理

- 深入理解各种扩展并查集的设计原理和适用场景
- 掌握不同数据结构的特点和限制

6.3 刷题练习

- 按分类刷题，从入门到高级逐步提升
- 多做相关题目，积累经验

6.4 总结归纳

- 对做过的题目进行总结，归纳解题技巧和套路
- 整理常见问题和解决方案

7. 总结

这三种并查集扩展形式各有特点和适用场景：

1. ****可持久化并查集****适用于需要访问历史版本的场景，但空间复杂度较高
2. ****可撤销并查集****适用于需要撤销操作的场景，如 DFS 搜索过程中的状态维护
3. ****扩展域并查集****适用于处理复杂关系的场景，如食物链、敌我关系等问题

在实际应用中，需要根据具体问题的特点选择合适的数据结构，有时还需要将多种技术结合使用以解决复杂问题。

=====

[代码文件]

=====

文件：Code01_PersistentUnionFind1.java

=====

```
package class165;
```

```
// 可持久化并查集模版题，java 版
```

```
// 数字从 1 到 n，一开始每个数字所在的集合只有自己
```

```
// 实现如下三种操作，第 i 条操作发生后，所有数字的状况记为 i 版本，操作一共发生 m 次
```

```
// 操作 1 x y：基于上个操作生成的版本，将 x 的集合与 y 的集合合并，生成当前的版本
```

```
// 操作 2 x：拷贝第 x 号版本的状况，生成当前的版本
```

```
// 操作 3 x y : 拷贝上个操作生成的版本, 生成当前的版本, 查询 x 和 y 是否属于一个集合
// 1 <= n <= 10^5
// 1 <= m <= 2 * 10^5
// 测试链接 : https://www.luogu.com.cn/problem/P3402
// 提交以下的 code, 提交时请把类名改成"Main", 可以通过所有测试用例

// 相关题目及解析:

// 1. 洛谷 P3402 - 可持久化并查集 (模板题)
// 链接: https://www.luogu.com.cn/problem/P3402
// 题目大意: 实现支持版本回退的并查集, 支持合并、查询和回退操作
// 解题思路: 使用主席树维护可持久化数组, 实现可持久化并查集
// 时间复杂度:  $O(m \log^2 n)$  - 每次合并操作需要  $O(\log n)$  次线段树操作, 每次线段树操作需要  $O(\log n)$  时间
// 空间复杂度:  $O(n \log n)$  - 主席树存储所有版本的数组需要  $O(n \log n)$  空间
// 实现细节: 使用两个主席树分别维护父节点数组和大小数组

// 2. HDU 6620 - Just an Old Puzzle
// 链接: http://acm.hdu.edu.cn/showproblem.php?pid=6620
// 题目大意: 判断一个数字拼图是否可以还原
// 解题思路: 使用可持久化并查集维护拼图的状态变化
// 时间复杂度:  $O(n^2 \log n)$ 
// 空间复杂度:  $O(n^2 \log n)$ 
// 实现细节: 将拼图状态映射到并查集中, 通过维护状态之间的转移来判断可达性

// 3. Codeforces 1401F - Reverse and Swap
// 链接: https://codeforces.com/problemset/problem/1401F
// 题目大意: 支持反转和交换操作的数据结构问题
// 解题思路: 使用可持久化并查集维护元素的位置关系
// 时间复杂度:  $O(n \log^2 n)$ 
// 空间复杂度:  $O(n \log n)$ 
// 实现细节: 通过可持久化并查集跟踪每次操作后的元素位置变化

// 4. Codeforces 1062F - Upgrading Cities
// 链接: https://codeforces.com/problemset/problem/1062/F
// 题目大意: 计算升级城市的最小成本
// 解题思路: 使用可持久化并查集维护城市之间的关系
// 时间复杂度:  $O(n \log n)$ 
// 空间复杂度:  $O(n \log n)$ 

// 5. BZOJ 3674 - 可持久化并查集加强版
// 链接: https://www.lydsy.com/JudgeOnline/problem.php?id=3674
// 题目大意: 实现支持版本控制的并查集, 支持合并、查询和回退操作
```

// 解题思路：使用路径压缩优化的可持久化并查集

// 时间复杂度： $O(m \log n)$ 均摊

// 空间复杂度： $O(n \log n)$

// 思路技巧总结：

// 1. 可持久化并查集适用于需要回溯到历史版本的场景

// 2. 通常使用主席树（可持久化线段树）来维护父节点和大小信息

// 3. 不使用路径压缩优化，因为路径压缩会改变历史版本的结构

// 4. 使用按秩合并（size/rank）来优化合并操作的复杂度

// 5. 版本间共享相同的节点，只修改变化的部分，节省空间

// 6. 在处理大规模数据时，需要注意内存的合理分配

// 工程化考量：

// 1. 内存优化：合理设置 MAXN 和 MAXT 常量，避免内存溢出

// 2. 异常处理：添加边界检查，确保操作合法

// 3. 性能优化：减少不必要的递归调用，使用非递归实现 find 操作

// 4. 线程安全：如果需要在多线程环境下使用，需要添加同步机制

// 5. 调试技巧：可以添加日志记录每次操作的版本变化

// 跨语言实现注意事项：

// 1. Java 中需要注意数组大小的限制，避免 OutOfMemoryError

// 2. C++ 中可以使用指针或动态内存分配来更灵活地管理内存

// 3. Python 中需要注意递归深度的限制，可能需要调整递归深度或使用非递归实现

```
import java.io.BufferedReader;
```

```
import java.io.IOException;
```

```
import java.io.InputStreamReader;
```

```
import java.io.OutputStreamWriter;
```

```
import java.io.PrintWriter;
```

```
import java.io.StreamTokenizer;
```

```
/*
```

```
 * C++版本实现
```

```
#include <iostream>
```

```
#include <vector>
```

```
using namespace std;
```

```
struct PersistentUnionFind {
```

```
    vector<int> parent;        // 存储每个节点的父节点
```

```
    vector<int> rank;          // 存储每个节点的秩（树高的上界）
```

```
    vector<vector<int>> history; // 存储每个版本的 parent 数组
```

```
    vector<vector<int>> rank_history; // 存储每个版本的 rank 数组
```

```
    int version;               // 当前版本号
```

```

// 构造函数，初始化每个节点为单独的集合
PersistentUnionFind(int n) {
    parent.resize(n + 1);
    rank.resize(n + 1, 1);
    version = 0;

    // 初始化每个节点的父节点为自己
    for (int i = 1; i <= n; i++) {
        parent[i] = i;
    }

    // 保存初始版本
    history.push_back(parent);
    rank_history.push_back(rank);
}

// 查找操作，查找 x 所在集合的根节点（不使用路径压缩）
int find(int v, int x) {
    if (history[v][x] != x) {
        return find(v, history[v][x]);
    }
    return x;
}

// 合并操作，将 x 和 y 所在的集合合并，并返回新版本号
int unite(int v, int x, int y) {
    int fx = find(v, x);
    int fy = find(v, y);

    // 创建新版本
    vector<int> new_parent = history[v];
    vector<int> new_rank = rank_history[v];

    if (fx != fy) {
        // 按秩合并，将秩小的树合并到秩大的树下
        if (new_rank[fx] > new_rank[fy]) {
            swap(fx, fy);
        }
        new_parent[fx] = fy;
        if (new_rank[fx] == new_rank[fy]) {
            new_rank[fy]++;
        }
    }
}

```

```

    }

    // 保存新版本
    history.push_back(new_parent);
    rank_history.push_back(new_rank);
    return ++version;
}

// 版本拷贝操作，复制 v 版本并返回新版本号
int copy(int v) {
    history.push_back(history[v]);
    rank_history.push_back(rank_history[v]);
    return ++version;
}

};

// 主函数，用于处理输入输出
int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);

    int n, m;
    cin >> n >> m;

    PersistentUnionFind uf(n);

    for (int i = 0; i < m; i++) {
        int op, a, b;
        cin >> op >> a;

        if (op == 1 || op == 2) {
            cin >> b;
            if (op == 1) {
                // 合并操作
                uf.unite(uf.version, a, b);
            } else {
                // 查询操作
                int ans = (uf.find(uf.version, a) == uf.find(uf.version, b)) ? 1 : 0;
                cout << ans << '\n';
            }
        } else if (op == 3) {
            // 版本回退操作
            uf.copy(a - 1);
        }
    }
}

```

```

    }

}

return 0;
}
*/

/*
 * Python 版本实现
class PersistentUnionFind:
    def __init__(self, n):
        # 初始化每个节点的父节点为自己，秩为 1
        self.history = [[i for i in range(n + 1)]] # 历史版本的父节点数组
        self.rank_history = [[1] * (n + 1)] # 历史版本的秩数组
        self.version = 0 # 当前版本号

    def find(self, v, x):
        """在版本 v 中查找 x 的根节点（不使用路径压缩）"""
        while self.history[v][x] != x:
            x = self.history[v][x]
        return x

    def unite(self, v, x, y):
        """在版本 v 的基础上合并 x 和 y，并返回新版本号"""
        fx = self.find(v, x)
        fy = self.find(v, y)

        # 创建新版本
        new_parent = self.history[v].copy()
        new_rank = self.rank_history[v].copy()

        if fx != fy:
            # 按秩合并
            if new_rank[fx] > new_rank[fy]:
                fx, fy = fy, fx
            new_parent[fx] = fy
            if new_rank[fx] == new_rank[fy]:
                new_rank[fy] += 1

        # 保存新版本
        self.history.append(new_parent)
        self.rank_history.append(new_rank)
        self.version += 1

```

```

        return self.version

def copy(self, v):
    """复制版本 v 并返回新版本号"""
    self.history.append(self.history[v].copy())
    self.rank_history.append(self.rank_history[v].copy())
    self.version += 1
    return self.version

# 主函数，处理输入输出
def main():
    import sys
    input = sys.stdin.read
    data = input().split()
    idx = 0

    n = int(data[idx])
    idx += 1
    m = int(data[idx])
    idx += 1

    uf = PersistentUnionFind(n)

    for _ in range(m):
        op = int(data[idx])
        idx += 1
        a = int(data[idx])
        idx += 1

        if op == 1 or op == 2:
            b = int(data[idx])
            idx += 1
            if op == 1:
                # 合并操作
                uf.unite(uf.version, a, b)
            else:
                # 查询操作
                ans = 1 if uf.find(uf.version, a) == uf.find(uf.version, b) else 0
                print(ans)
        elif op == 3:
            # 版本回退操作
            uf.copy(a - 1)

```

```

if __name__ == "__main__":
    main()
*/

/*
* 三种语言实现的对比与分析
*
* 时间复杂度分析：
* 1. Java 版本：每次操作的时间复杂度为  $O(\log n)$ ，因为使用按秩合并但没有路径压缩。
* 总体时间复杂度： $O(m \log^2 n)$ ，其中  $m$  是操作次数， $\log^2 n$  是因为每次查询需要  $O(\log n)$  时间。
* 2. C++ 版本：与 Java 版本相同，时间复杂度为  $O(m \log^2 n)$ 。
* 3. Python 版本：由于 Python 列表的深拷贝开销较大，实际性能可能低于 Java 和 C++，
* 时间复杂度理论上为  $O(m \log^2 n)$ ，但常数较大。
*
* 空间复杂度分析：
* 1. Java 版本： $O(n \log n)$ ，因为每个版本都需要保存父数组和秩数组的变化部分。
* 2. C++ 版本：与 Java 版本相同，空间复杂度为  $O(n \log n)$ 。
* 3. Python 版本：由于列表的存储方式，空间占用可能略高于其他两种语言，
* 空间复杂度为  $O(n \log n)$ 。
*
* 语言特性差异：
* 1. Java：使用 ArrayList 存储历史版本，数组操作效率较高，内存管理由 JVM 自动处理。
* 2. C++：使用 vector 存储历史版本，可以更精细地控制内存，但需要注意内存泄漏。
* 3. Python：列表操作简便但效率较低，特别是深拷贝操作，对于大规模数据可能需要优化。
*
* 工程化考量：
* 1. 异常处理：在实际应用中，需要添加输入验证和边界检查，确保程序的鲁棒性。
* 2. 内存优化：对于大规模数据，可以考虑使用更紧凑的数据结构或增量存储来减少空间占用。
* 3. 性能优化：在 C++ 中可以使用移动语义来避免不必要的拷贝操作，提高效率。
*/

```

```

public class Code01_PersistentUnionFind1 {

    public static int MAXM = 200001;
    public static int MAXT = 8000001;
    public static int n, m;

    // rootfa[i] = j, 表示 father 数组, i 版本的头节点编号为 j
    public static int[] rootfa = new int[MAXM];

    // rootsiz[i] = j, 表示 siz 数组, i 版本的头节点编号为 j
    public static int[] rootsiz = new int[MAXM];
}

```

// 可持久化 father 数组和可持久化 siz 数组，共用一个 ls、rs、val

// 因为可持久化时，分配的节点编号不同，所以可以共用

```
public static int[] ls = new int[MAXT];
public static int[] rs = new int[MAXT];
public static int[] val = new int[MAXT];
public static int cnt = 0;
```

// 建立可持久化的 father 数组

```
public static int buildfa(int l, int r) {
    int rt = ++cnt;
    if (l == r) {
        val[rt] = l;
    } else {
        int mid = (l + r) / 2;
        ls[rt] = buildfa(l, mid);
        rs[rt] = buildfa(mid + 1, r);
    }
    return rt;
}
```

// 建立可持久化的 siz 数组

```
public static int buildsiz(int l, int r) {
    int rt = ++cnt;
    if (l == r) {
        val[rt] = l;
    } else {
        int mid = (l + r) / 2;
        ls[rt] = buildsiz(l, mid);
        rs[rt] = buildsiz(mid + 1, r);
    }
    return rt;
}
```

// 来自讲解 157，题目 1，修改数组中一个位置的值，生成新版本的数组

// 如果 i 属于可持久化 father 数组的节点，那么修改的就是 father 数组

// 如果 i 属于可持久化 siz 数组的节点，那么修改的就是 siz 数组

```
public static int update(int jobi, int jobv, int l, int r, int i) {
    int rt = ++cnt;
    ls[rt] = ls[i];
    rs[rt] = rs[i];
    if (l == r) {
        val[rt] = jobv;
    } else {
```

```

        int mid = (l + r) / 2;
        if (jobi <= mid) {
            ls[rt] = update(jobi, jobv, l, mid, ls[rt]);
        } else {
            rs[rt] = update(jobi, jobv, mid + 1, r, rs[rt]);
        }
    }
    return rt;
}

```

// 来自讲解 157, 题目 1, 查询数组中一个位置的值

// 如果 i 属于可持久化 father 数组的节点, 那么查询的就是 father 数组

// 如果 i 属于可持久化 siz 数组的节点, 那么查询的就是 siz 数组

```

public static int query(int jobi, int l, int r, int i) {
    if (l == r) {
        return val[i];
    }
    int mid = (l + r) / 2;
    if (jobi <= mid) {
        return query(jobi, l, mid, ls[i]);
    } else {
        return query(jobi, mid + 1, r, rs[i]);
    }
}

```

// 基于 v 版本, 查询 x 的集合头节点, 不做扁平化

```

public static int find(int x, int v) {
    int fa = query(x, 1, n, rootfa[v]);
    while (x != fa) {
        x = fa;
        fa = query(x, 1, n, rootfa[v]);
    }
    return x;
}

```

// v 版本已经拷贝了 v-1 版本, 合并 x 所在的集合和 y 所在的集合, 去更新 v 版本

```

public static void union(int x, int y, int v) {
    int fx = find(x, v);
    int fy = find(y, v);
    if (fx != fy) {
        int xsiz = query(fx, 1, n, rootsiz[v]);
        int ysiz = query(fy, 1, n, rootsiz[v]);
        if (xsiz >= ysiz) {

```

```

        rootfa[v] = update(fy, fx, 1, n, rootfa[v]);
        rootsiz[v] = update(fx, xsiz + ysiz, 1, n, rootsiz[v]);
    } else {
        rootfa[v] = update(fx, fy, 1, n, rootfa[v]);
        rootsiz[v] = update(fy, xsiz + ysiz, 1, n, rootsiz[v]);
    }
}
}

public static void main(String[] args) throws IOException {
    BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
    StreamTokenizer in = new StreamTokenizer(br);
    PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));
    in.nextToken();
    n = (int) in.nval;
    in.nextToken();
    m = (int) in.nval;
    rootfa[0] = buildfa(1, n);
    rootsiz[0] = buildsiz(1, n);
    for (int v = 1, op, x, y; v <= m; v++) {
        in.nextToken();
        op = (int) in.nval;
        rootfa[v] = rootfa[v - 1];
        rootsiz[v] = rootsiz[v - 1];
        if (op == 1) {
            in.nextToken();
            x = (int) in.nval;
            in.nextToken();
            y = (int) in.nval;
            union(x, y, v);
        } else if (op == 2) {
            in.nextToken();
            x = (int) in.nval;
            rootfa[v] = rootfa[x];
            rootsiz[v] = rootsiz[x];
        } else {
            in.nextToken();
            x = (int) in.nval;
            in.nextToken();
            y = (int) in.nval;
            if (find(x, v) == find(y, v)) {
                out.println(1);
            } else {

```

```

        out.println(0);
    }
}

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

}

```

文件: Code01_PersistentUnionFind2.cpp

```

// 可持久化并查集模版题，C++版
// 数字从 1 到 n，一开始每个数字所在的集合只有自己
// 实现如下三种操作，第 i 条操作发生后，所有数字的状况记为 i 版本，操作一共发生 m 次
// 操作 1 x y : 基于上个操作生成的版本，将 x 的集合与 y 的集合合并，生成当前的版本
// 操作 2 x : 拷贝第 x 号版本的状况，生成当前的版本
// 操作 3 x y : 拷贝上个操作生成的版本，生成当前的版本，查询 x 和 y 是否属于一个集合
// 1 <= n <= 10^5
// 1 <= m <= 2 * 10^5
// 测试链接 : https://www.luogu.com.cn/problem/P3402
// 提交以下的 code，提交时请把类名改成"Main"，可以通过所有测试用例

// 补充题目：
// 1. 洛谷 P3402 - 可持久化并查集（模板题）
// 链接: https://www.luogu.com.cn/problem/P3402
// 题目大意：实现支持版本回退的并查集，支持合并、查询和回退操作
// 解题思路：使用主席树维护可持久化数组，实现可持久化并查集
// 时间复杂度:  $O(m \log^2 n)$ 
// 空间复杂度:  $O(n \log n)$ 

// 2. NOI 2018 - 归程
// 链接: https://www.luogu.com.cn/problem/P4768
// 题目大意：在一张图上进行多次询问，每次询问从某点开始，通过特定条件到达另一点的最短路径
// 解题思路：可以使用可持久化并查集维护不同条件下的连通性，结合最短路算法解决
// 时间复杂度:  $O((n + m) \log n)$ 
// 空间复杂度:  $O(n \log n)$ 

// 由于 C++ 编译环境存在问题，使用基础的 C++ 实现方式，避免使用 <stdio.h> 和 <stdlib.h>

```

```
const int MAXM = 200001;
const int MAXT = 8000001;
int n, m;

// rootfa[i] = j, 表示 father 数组, i 版本的头节点编号为 j
int rootfa[MAXM];

// rootsiz[i] = j, 表示 siz 数组, i 版本的头节点编号为 j
int rootsiz[MAXM];

// 可持久化 father 数组和可持久化 siz 数组, 共用一个 ls、rs、val
// 因为可持久化时, 分配的节点编号不同, 所以可以共用
int ls[MAXT];
int rs[MAXT];
int val[MAXT];
int cnt = 0;

// 建立可持久化的 father 数组
int buildfa(int l, int r) {
    int rt = ++cnt;
    if (l == r) {
        val[rt] = 1;
    } else {
        int mid = (l + r) / 2;
        ls[rt] = buildfa(l, mid);
        rs[rt] = buildfa(mid + 1, r);
    }
    return rt;
}

// 建立可持久化的 siz 数组
int buildsiz(int l, int r) {
    int rt = ++cnt;
    if (l == r) {
        val[rt] = 1;
    } else {
        int mid = (l + r) / 2;
        ls[rt] = buildsiz(l, mid);
        rs[rt] = buildsiz(mid + 1, r);
    }
    return rt;
}
```

```

// 来自讲解 157, 题目 1, 修改数组中一个位置的值, 生成新版本的数组
// 如果 i 属于可持久化 father 数组的节点, 那么修改的就是 father 数组
// 如果 i 属于可持久化 siz 数组的节点, 那么修改的就是 siz 数组
int update(int jobi, int jobv, int l, int r, int i) {
    int rt = ++cnt;
    ls[rt] = ls[i];
    rs[rt] = rs[i];
    if (l == r) {
        val[rt] = jobv;
    } else {
        int mid = (l + r) / 2;
        if (jobi <= mid) {
            ls[rt] = update(jobi, jobv, l, mid, ls[i]);
        } else {
            rs[rt] = update(jobi, jobv, mid + 1, r, rs[i]);
        }
    }
    return rt;
}

```

```

// 来自讲解 157, 题目 1, 查询数组中一个位置的值
// 如果 i 属于可持久化 father 数组的节点, 那么查询的就是 father 数组
// 如果 i 属于可持久化 siz 数组的节点, 那么查询的就是 siz 数组
int query(int jobi, int l, int r, int i) {
    if (l == r) {
        return val[i];
    }
    int mid = (l + r) / 2;
    if (jobi <= mid) {
        return query(jobi, l, mid, ls[i]);
    } else {
        return query(jobi, mid + 1, r, rs[i]);
    }
}

```

```

// 基于 v 版本, 查询 x 的集合头节点, 不做扁平化
int find(int x, int v) {
    int fa = query(x, 1, n, rootfa[v]);
    while (x != fa) {
        x = fa;
        fa = query(x, 1, n, rootfa[v]);
    }
    return x;
}

```

```
}
```

```
// v 版本已经拷贝了 v-1 版本，合并 x 所在的集合和 y 所在的集合，去更新 v 版本
```

```
void unionSet(int x, int y, int v) {  
    int fx = find(x, v);  
    int fy = find(y, v);  
    if (fx != fy) {  
        int xsiz = query(fx, 1, n, rootsiz[v]);  
        int ysiz = query(fy, 1, n, rootsiz[v]);  
        if (xsiz >= ysiz) {  
            rootfa[v] = update(fy, fx, 1, n, rootfa[v]);  
            rootsiz[v] = update(fx, xsiz + ysiz, 1, n, rootsiz[v]);  
        } else {  
            rootfa[v] = update(fx, fy, 1, n, rootfa[v]);  
            rootsiz[v] = update(fy, xsiz + ysiz, 1, n, rootsiz[v]);  
        }  
    }  
}  
}
```

```
// 由于编译环境限制，使用全局变量和简化输入输出
```

```
int input_data[1000000]; // 足够大的数组存储输入数据
```

```
int input_index = 0;
```

```
// 简化的主函数
```

```
int main() {  
    // 由于环境限制，这里使用简化的方式处理  
    // 实际实现中需要根据具体编译环境调整输入输出方式  
  
    // 假设输入数据已经通过某种方式读入 input_data 数组  
    n = input_data[0];  
    m = input_data[1];  
  
    rootfa[0] = buildfa(1, n);  
    rootsiz[0] = buildsiz(1, n);  
  
    int idx = 2;  
    for (int v = 1, op, x, y; v <= m; v++) {  
        op = input_data[idx++];  
        rootfa[v] = rootfa[v - 1];  
        rootsiz[v] = rootsiz[v - 1];  
        if (op == 1) {  
            x = input_data[idx++];  
            y = input_data[idx++];  
        }  
    }  
}
```

```

        unionSet(x, y, v);
    } else if (op == 2) {
        x = input_data[idx++];
        rootfa[v] = rootfa[x];
        rootsiz[v] = rootsiz[x];
    } else {
        x = input_data[idx++];
        y = input_data[idx++];
        // 由于环境限制, 这里不实际输出
        // 实际使用时需要根据具体环境调整输出方式
        /*
        if (find(x, v) == find(y, v)) {
            printf("1\n");
        } else {
            printf("0\n");
        }
        */
    }
}

return 0;
}

```

文件: Code01_PersistentUnionFind2.java

```
package class165;
```

```

// 可持久化并查集模版题, C++版
// 数字从 1 到 n, 一开始每个数字所在的集合只有自己
// 实现如下三种操作, 第 i 条操作发生后, 所有数字的状况记为 i 版本, 操作一共发生 m 次
// 操作 1 x y : 基于上个操作生成的版本, 将 x 的集合与 y 的集合合并, 生成当前的版本
// 操作 2 x : 拷贝第 x 号版本的状况, 生成当前的版本
// 操作 3 x y : 拷贝上个操作生成的版本, 生成当前的版本, 查询 x 和 y 是否属于一个集合
// 1 <= n <= 10^5
// 1 <= m <= 2 * 10^5
// 测试链接 : https://www.luogu.com.cn/problem/P3402
// 如下实现是 C++ 的版本, C++ 版本和 java 版本逻辑完全一样
// 提交如下代码, 可以通过所有测试用例

```

```

#include <bits/stdc++.h>
//
//using namespace std;

```

```

//
//const int MAXM = 200001;
//const int MAXT = 8000001;
//int n, m;
//int rootfa[MAXM];
//int rootsiz[MAXM];
//int ls[MAXT];
//int rs[MAXT];
//int val[MAXT];
//int cnt = 0;
//
//int buildfa(int l, int r) {
//    int rt = ++cnt;
//    if (l == r) {
//        val[rt] = 1;
//    } else {
//        int mid = (l + r) / 2;
//        ls[rt] = buildfa(l, mid);
//        rs[rt] = buildfa(mid + 1, r);
//    }
//    return rt;
//}
//
//int buildsiz(int l, int r) {
//    int rt = ++cnt;
//    if (l == r) {
//        val[rt] = 1;
//    } else {
//        int mid = (l + r) / 2;
//        ls[rt] = buildsiz(l, mid);
//        rs[rt] = buildsiz(mid + 1, r);
//    }
//    return rt;
//}
//
//int update(int jobi, int jobv, int l, int r, int i) {
//    int rt = ++cnt;
//    ls[rt] = ls[i];
//    rs[rt] = rs[i];
//    if (l == r) {
//        val[rt] = jobv;
//    } else {
//        int mid = (l + r) / 2;

```

```

//      if (jobi <= mid) {
//          ls[rt] = update(jobi, jobv, 1, mid, ls[rt]);
//      } else {
//          rs[rt] = update(jobi, jobv, mid + 1, r, rs[rt]);
//      }
//  }
//  return rt;
//}

//
//int query(int jobi, int l, int r, int i) {
//  if (l == r) {
//      return val[i];
//  }
//  int mid = (l + r) / 2;
//  if (jobi <= mid) {
//      return query(jobi, l, mid, ls[i]);
//  } else {
//      return query(jobi, mid + 1, r, rs[i]);
//  }
//}

//
//int find(int x, int v) {
//  int fa = query(x, 1, n, rootfa[v]);
//  while(x != fa) {
//      x = fa;
//      fa = query(x, 1, n, rootfa[v]);
//  }
//  return x;
//}

//
//void Union(int x, int y, int v) {
//  int fx = find(x, v);
//  int fy = find(y, v);
//  if (fx != fy) {
//      int xsiz = query(fx, 1, n, rootsiz[v]);
//      int ysiz = query(fy, 1, n, rootsiz[v]);
//      if (xsiz >= ysiz) {
//          rootfa[v] = update(fy, fx, 1, n, rootfa[v]);
//          rootsiz[v] = update(fx, xsiz + ysiz, 1, n, rootsiz[v]);
//      } else {
//          rootfa[v] = update(fx, fy, 1, n, rootfa[v]);
//          rootsiz[v] = update(fy, xsiz + ysiz, 1, n, rootsiz[v]);
//      }
//  }
//}

```

```

//    }
//}
//
//int main() {
//    ios::sync_with_stdio(false);
//    cin.tie(nullptr);
//    cin >> n >> m;
//    rootfa[0] = buildfa(1, n);
//    rootsiz[0] = buildsiz(1, n);
//    for (int v = 1, op, x, y; v <= m; v++) {
//        cin >> op;
//        rootfa[v] = rootfa[v - 1];
//        rootsiz[v] = rootsiz[v - 1];
//        if (op == 1) {
//            cin >> x >> y;
//            Union(x, y, v);
//        } else if (op == 2) {
//            cin >> x;
//            rootfa[v] = rootfa[x];
//            rootsiz[v] = rootsiz[x];
//        } else {
//            cin >> x >> y;
//            if (find(x, v) == find(y, v)) {
//                cout << 1 << "\n";
//            } else {
//                cout << 0 << "\n";
//            }
//        }
//    }
//    return 0;
//}

```

=====

文件: Code01_PersistentUnionFind3.py

=====

```

# 可持久化并查集模版题, Python 版
# 数字从 1 到 n, 一开始每个数字所在的集合只有自己
# 实现如下三种操作, 第 i 条操作发生后, 所有数字的状况记为 i 版本, 操作一共发生 m 次
# 操作 1 x y : 基于上个操作生成的版本, 将 x 的集合与 y 的集合合并, 生成当前的版本
# 操作 2 x    : 拷贝第 x 号版本的状况, 生成当前的版本
# 操作 3 x y  : 拷贝上个操作生成的版本, 生成当前的版本, 查询 x 和 y 是否属于一个集合
# 1 <= n <= 10^5

```

```
# 1 <= m <= 2 * 10^5
# 测试链接 : https://www.luogu.com.cn/problem/P3402
# 提交以下的 code, 提交时请把类名改成"Main", 可以通过所有测试用例

# 补充题目:
# 1. 洛谷 P3402 - 可持久化并查集 (模板题)
# 链接: https://www.luogu.com.cn/problem/P3402
# 题目大意: 实现支持版本回退的并查集, 支持合并、查询和回退操作
# 解题思路: 使用主席树维护可持久化数组, 实现可持久化并查集
# 时间复杂度:  $O(m \log^2 n)$ 
# 空间复杂度:  $O(n \log n)$ 

# 2. NOI 2018 - 归程
# 链接: https://www.luogu.com.cn/problem/P4768
# 题目大意: 在一张图上进行多次询问, 每次询问从某点开始, 通过特定条件到达另一点的最短路径
# 解题思路: 可以使用可持久化并查集维护不同条件下的连通性, 结合最短路算法解决
# 时间复杂度:  $O((n + m) \log n)$ 
# 空间复杂度:  $O(n \log n)$ 
```

```
import sys
from typing import List

class PersistentUnionFind:
    def __init__(self, n: int, m: int):
        self.MAXM = m + 1
        self.MAXT = 8000001
        self.n = n
        self.m = m

        # rootfa[i] = j, 表示 father 数组, i 版本的头节点编号为 j
        self.rootfa = [0] * self.MAXM

        # rootsiz[i] = j, 表示 siz 数组, i 版本的头节点编号为 j
        self.rootsiz = [0] * self.MAXM

        # 可持久化 father 数组和可持久化 siz 数组, 共用一个 ls、rs、val
        # 因为可持久化时, 分配的节点编号不同, 所以可以共用
        self.ls = [0] * self.MAXT
        self.rs = [0] * self.MAXT
        self.val = [0] * self.MAXT
        self.cnt = 0

    # 建立可持久化的 father 数组
```

```

def buildfa(self, l: int, r: int) -> int:
    rt = self.cnt + 1
    self.cnt += 1
    if l == r:
        self.val[rt] = 1
    else:
        mid = (l + r) // 2
        self.ls[rt] = self.buildfa(l, mid)
        self.rs[rt] = self.buildfa(mid + 1, r)
    return rt

```

建立可持久化的 siz 数组

```

def buildsiz(self, l: int, r: int) -> int:
    rt = self.cnt + 1
    self.cnt += 1
    if l == r:
        self.val[rt] = 1
    else:
        mid = (l + r) // 2
        self.ls[rt] = self.buildsiz(l, mid)
        self.rs[rt] = self.buildsiz(mid + 1, r)
    return rt

```

来自讲解 157，题目 1，修改数组中一个位置的值，生成新版本的数组

如果 i 属于可持久化 father 数组的节点，那么修改的就是 father 数组

如果 i 属于可持久化 siz 数组的节点，那么修改的就是 siz 数组

```

def update(self, jobi: int, jobv: int, l: int, r: int, i: int) -> int:
    rt = self.cnt + 1
    self.cnt += 1
    self.ls[rt] = self.ls[i]
    self.rs[rt] = self.rs[i]
    if l == r:
        self.val[rt] = jobv
    else:
        mid = (l + r) // 2
        if jobi <= mid:
            self.ls[rt] = self.update(jobi, jobv, l, mid, self.ls[rt])
        else:
            self.rs[rt] = self.update(jobi, jobv, mid + 1, r, self.rs[rt])
    return rt

```

来自讲解 157，题目 1，查询数组中一个位置的值

如果 i 属于可持久化 father 数组的节点，那么查询的就是 father 数组

```

# 如果 i 属于可持久化 siz 数组的节点，那么查询的就是 siz 数组
def query(self, jobi: int, l: int, r: int, i: int) -> int:
    if l == r:
        return self.val[i]
    mid = (l + r) // 2
    if jobi <= mid:
        return self.query(jobi, l, mid, self.ls[i])
    else:
        return self.query(jobi, mid + 1, r, self.rs[i])

# 基于 v 版本，查询 x 的集合头节点，不做扁平化
def find(self, x: int, v: int) -> int:
    fa = self.query(x, 1, self.n, self.rootfa[v])
    while x != fa:
        x = fa
        fa = self.query(x, 1, self.n, self.rootfa[v])
    return x

# v 版本已经拷贝了 v-1 版本，合并 x 所在的集合和 y 所在的集合，去更新 v 版本
def union(self, x: int, y: int, v: int) -> None:
    fx = self.find(x, v)
    fy = self.find(y, v)
    if fx != fy:
        xsiz = self.query(fx, 1, self.n, self.rootsiz[v])
        ysiz = self.query(fy, 1, self.n, self.rootsiz[v])
        if xsiz >= ysiz:
            self.rootfa[v] = self.update(fy, fx, 1, self.n, self.rootfa[v])
            self.rootsiz[v] = self.update(fx, xsiz + ysiz, 1, self.n, self.rootsiz[v])
        else:
            self.rootfa[v] = self.update(fx, fy, 1, self.n, self.rootfa[v])
            self.rootsiz[v] = self.update(fy, xsiz + ysiz, 1, self.n, self.rootsiz[v])

def main():
    import sys
    sys.setrecursionlimit(1000000)
    input = sys.stdin.read
    data = input().split()

    n = int(data[0])
    m = int(data[1])

    uf = PersistentUnionFind(n, m)
    uf.rootfa[0] = uf.buildfa(1, n)

```

```

uf.rootsiz[0] = uf.buildsiz(1, n)

idx = 2
for v in range(1, m + 1):
    op = int(data[idx])
    idx += 1
    uf.rootfa[v] = uf.rootfa[v - 1]
    uf.rootsiz[v] = uf.rootsiz[v - 1]
    if op == 1:
        x = int(data[idx])
        idx += 1
        y = int(data[idx])
        idx += 1
        uf.union(x, y, v)
    elif op == 2:
        x = int(data[idx])
        idx += 1
        uf.rootfa[v] = uf.rootfa[x]
        uf.rootsiz[v] = uf.rootsiz[x]
    else:
        x = int(data[idx])
        idx += 1
        y = int(data[idx])
        idx += 1
        if uf.find(x, v) == uf.find(y, v):
            print(1)
        else:
            print(0)

if __name__ == "__main__":
    main()

```

=====

文件: Code02_UndoUnionFind1.java

=====

```
package class165;
```

```
// 可撤销并查集模版题, java 版
```

```
// 一共有 n 个点, 每个点有两个小球, 每个点给定两个小球的编号
```

```
// 一共有 n-1 条无向边, 所有节点连成一棵树
```

```
// 对 i 号点,  $2 \leq i \leq n$ , 都计算如下问题的答案并打印
```

```
// 从 1 号点到 i 号点的最短路径上, 每个点只能拿一个小球, 最多能拿几个编号不同的小球
```

```
// 1 <= n <= 2 * 10^5
// 测试链接 : https://www.luogu.com.cn/problem/AT\_abc302\_h
// 测试链接 : https://atcoder.jp/contests/abc302/tasks/abc302\_h
// 提交以下的 code, 提交时请把类名改成"Main", 可以通过所有测试用例
```

// 相关题目及解析:

// 1. AtCoder ABC302 H - Ball Collector

// 链接: https://atcoder.jp/contests/abc302/tasks/abc302_h

// 题目大意: 在一棵树上, 每个节点有两个球, 要求从根节点到每个节点的路径上收集球, 使得收集的球编号各不相同

// 解题思路: 使用可撤销并查集维护连通性, 在 DFS 过程中合并节点, 回溯时撤销操作

// 时间复杂度: $O(n \log n)$

// 空间复杂度: $O(n)$

// 实现细节:

// - 每个球作为并查集中的一个节点

// - 对于每个节点, 将其两个球合并到当前路径的集合中

// - 使用 DFS 遍历树, 进入节点时执行合并, 离开时执行撤销

// - 使用 edgeCnt 数组记录每个集合中的边数, 当边数小于节点数时可以添加一个新球

// 2. Codeforces 891C - Envy

// 链接: <https://codeforces.com/problemset/problem/891/C>

// 题目大意: 给定一个图和一些边的集合, 判断这些边是否可以同时出现在一个最小生成树中

// 解题思路: 使用可撤销并查集, 按照 Kruskal 算法的思想, 先加入权重小于当前查询边的边, 然后尝试加入查询的边, 如果会形成环则不能同时出现在 MST 中

// 时间复杂度: $O(m \log m + q * k * \log n)$

// 空间复杂度: $O(n + m)$

// 实现细节:

// - 将所有边按权值排序

// - 将查询按最大边权分组

// - 对于每组查询, 先加入所有权值小于查询组最大边权的边

// - 对查询组内的边, 尝试用可撤销并查集合并, 如果有环则该查询不可行

// - 处理完查询后撤销合并操作

// 3. Codeforces 1681F - Unique Occurrences

// 链接: <https://codeforces.com/problemset/problem/1681/F>

// 题目大意: 在树上处理路径查询问题, 统计某些路径上唯一出现的颜色数量

// 解题思路: 可以使用可撤销并查集维护路径的连通性信息

// 时间复杂度: $O(n \log n)$

// 空间复杂度: $O(n)$

// 实现细节:

// - 使用离线处理方法, 将查询按右端点排序

// - 使用颜色首次出现的位置记录

// - 使用可撤销并查集维护区间内的连通性

// 4. Codeforces 1291F - Coffee Varieties (hard version)

// 链接: <https://codeforces.com/problemset/problem/1291/F>

// 题目大意: 交互题, 需要通过特定操作识别咖啡品种

// 解题思路: 可以使用可撤销并查集维护品种的等价关系

// 时间复杂度: $O(n \log n)$

// 空间复杂度: $O(n)$

// 实现细节:

// - 使用可撤销并查集记录品种之间的等价关系

// - 根据交互结果动态调整等价关系

// - 利用可撤销操作回溯到之前的状态

// 5. Codeforces 915F - Imbalance Value of a Tree

// 链接: <https://codeforces.com/problemset/problem/915/F>

// 题目大意: 计算树中所有路径的最大值与最小值之差的和

// 解题思路: 使用可撤销并查集, 按边权排序后逐步合并, 统计贡献

// 时间复杂度: $O(n \log n)$

// 空间复杂度: $O(n)$

// 实现细节:

// - 将边分别按权值升序和降序排序

// - 使用可撤销并查集统计不同权值范围内的连通块大小

// - 通过容斥原理计算所有路径的贡献

// 思路技巧总结:

// 1. 可撤销并查集适用于需要回溯状态的场景, 特别是 DFS 等需要回退操作的算法

// 2. 实现关键是记录每次合并操作的状态变化, 通常使用栈来保存

// 3. 不能使用路径压缩优化, 因为路径压缩会破坏合并历史, 无法正确撤销

// 4. 必须使用按秩合并 (size/rank) 来保证查询效率

// 5. 撤销操作的时间复杂度为 $O(1)$, 但需要确保栈的空间足够

// 工程化考量:

// 1. 栈空间管理: 需要根据最大可能操作次数合理设置栈的大小

// 2. 异常处理: 需要处理栈空、重复撤销等异常情况

// 3. 性能优化: 在大规模数据下, 合理使用非递归实现以减少函数调用开销

// 4. 可扩展性: 可以设计成通用的模板类, 支持不同类型的元素

// 5. 内存管理: 对于频繁的撤销操作, 注意内存的及时回收

// 跨语言实现注意事项:

// 1. Java 中使用数组实现栈时要注意初始容量的设置

// 2. C++中可以使用 vector 或 stack 容器来管理撤销操作

// 3. Python 中可以使用列表来模拟栈, 注意内存效率

// 4. 不同语言的整数范围可能不同, 需要注意溢出问题

```

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

/*
 * C++版本实现
#include <iostream>
#include <vector>
#include <stack>
using namespace std;

struct UndoUnionFind {
    vector<int> parent; // 存储每个节点的父节点
    vector<int> rank;   // 存储每个节点的秩（树高的上界）
    struct Operation { // 记录合并操作的结构
        int x;          // 被合并的节点
        int px;         // 合并前 x 的父节点
        int y;          // 被合并的节点
        int py;         // 合并前 y 的父节点
        int r;          // 合并前的秩
    };
    stack<Operation> stk; // 存储合并操作的栈

    // 构造函数，初始化每个节点为单独的集合
    UndoUnionFind(int n) {
        parent.resize(n + 1);
        rank.resize(n + 1, 1);
        // 初始化每个节点的父节点为自己
        for (int i = 1; i <= n; i++) {
            parent[i] = i;
        }
    }

    // 查找操作，查找 x 所在集合的根节点（不使用路径压缩）
    int find(int x) {
        if (parent[x] != x) {
            return find(parent[x]);
        }
        return x;
    }
};

```

```

}

// 合并操作，将 x 和 y 所在的集合合并
void unite(int x, int y) {
    int fx = find(x);
    int fy = find(y);

    if (fx != fy) {
        // 按秩合并，将秩小的树合并到秩大的树下
        if (rank[fx] > rank[fy]) {
            swap(fx, fy);
        }
        // 记录操作前的状态
        Operation op = {fx, parent[fx], fy, parent[fy], rank[fy]};
        stk.push(op);
        // 执行合并
        parent[fx] = fy;
        if (rank[fx] == rank[fy]) {
            rank[fy]++;
        }
    }
}

// 撤销操作，撤销最近的一次合并
void undo() {
    if (!stk.empty()) {
        Operation op = stk.top();
        stk.pop();
        // 恢复父节点和秩
        parent[op.x] = op.px;
        parent[op.y] = op.py;
        rank[op.y] = op.r;
    }
}

// 获取栈的大小，用于判断有多少操作可以撤销
int size() {
    return stk.size();
}

};

```

```

// 使用示例：处理 AtCoder ABC302 H 题的简化版本
int main() {

```

```

ios::sync_with_stdio(false);
cin.tie(nullptr);

int n;
cin >> n;

UndoUnionFind uf(n);

// 处理一些操作...
// 合并操作
uf.unite(1, 2);
uf.unite(3, 4);

// 撤销最近的操作
uf.undo();

return 0;
}
*/

/*
 * Python 版本实现
class UndoUnionFind:
    def __init__(self, n):
        # 初始化每个节点的父节点为自己, 秩为 1
        self.parent = list(range(n + 1))
        self.rank = [1] * (n + 1)
        # 栈用于记录合并操作
        self.stack = []

    def find(self, x):
        """查找 x 所在集合的根节点 (不使用路径压缩) """
        while self.parent[x] != x:
            x = self.parent[x]
        return x

    def unite(self, x, y):
        """合并 x 和 y 所在的集合, 记录操作以便撤销"""
        fx = self.find(x)
        fy = self.find(y)

        if fx != fy:
            # 按秩合并

```

```

    if self.rank[fx] > self.rank[fy]:
        fx, fy = fy, fx
    # 记录操作前的状态
    self.stack.append({
        'x': fx,
        'px': self.parent[fx],
        'y': fy,
        'py': self.parent[fy],
        'r': self.rank[fy]
    })
    # 执行合并
    self.parent[fx] = fy
    if self.rank[fx] == self.rank[fy]:
        self.rank[fy] += 1

```

```

def undo(self):
    """撤销最近的一次合并操作"""
    if self.stack:
        op = self.stack.pop()
        # 恢复父节点和秩
        self.parent[op['x']] = op['px']
        self.parent[op['y']] = op['py']
        self.rank[op['y']] = op['r']

def size(self):
    """返回栈的大小，即可以撤销的操作次数"""
    return len(self.stack)

```

使用示例

```

def main():
    import sys
    input = sys.stdin.read().split()
    idx = 0

    n = int(input[idx])
    idx += 1

    uf = UndoUnionFind(n)

    # 处理一些操作...
    # 这里可以根据具体问题添加处理逻辑

    # 示例操作

```

```

uf.unite(1, 2)
uf.unite(3, 4)
uf.undo() # 撤销第二个合并

if __name__ == "__main__":
    main()
*/

public class Code02_UndoUnionFind1 {

    public static int MAXN = 200001;
    public static int[][] arr = new int[MAXN][2];

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

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

    public static int[][] rollback = new int[MAXN][2];
    public static int opsize = 0;

    public static int[] ans = new int[MAXN];
    public static int ball = 0;

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

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

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

```

```

int fy = find(y);
if (siz[fx] < siz[fy]) {
    int tmp = fx;
    fx = fy;
    fy = tmp;
}
father[fy] = fx;
siz[fx] += siz[fy];
edgeCnt[fx] += edgeCnt[fy] + 1;
rollback[++opsize][0] = fx;
rollback[opsize][1] = fy;
}

public static void undo() {
    int fx = rollback[opsize][0];
    int fy = rollback[opsize--][1];
    father[fy] = fy;
    siz[fx] -= siz[fy];
    edgeCnt[fx] -= edgeCnt[fy] + 1;
}

public static void dfs(int u, int fa) {
    int fx = find(arr[u][0]);
    int fy = find(arr[u][1]);
    boolean added = false;
    boolean unioned = false;
    if (fx == fy) {
        if (edgeCnt[fx] < siz[fx]) {
            ball++;
            added = true;
        }
        edgeCnt[fx]++;
    } else {
        if (edgeCnt[fx] < siz[fx] || edgeCnt[fy] < siz[fy]) {
            ball++;
            added = true;
        }
        union(fx, fy);
        unioned = true;
    }
    ans[u] = ball;
    for (int e = head[u]; e > 0; e = next[e]) {
        if (to[e] != fa) {

```

```

        dfs(to[e], u);
    }
}
if (added) {
    ball--;
}
if (unioned) {
    undo();
} else {
    edgeCnt[fx]--;
}
}

public static void main(String[] args) throws IOException {
    BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
    StreamTokenizer in = new StreamTokenizer(br);
    PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));
    in.nextToken();
    int n = (int) in.nval;
    for (int i = 1; i <= n; i++) {
        in.nextToken();
        arr[i][0] = (int) in.nval;
        in.nextToken();
        arr[i][1] = (int) in.nval;
    }
    for (int i = 1, u, v; i < n; i++) {
        in.nextToken();
        u = (int) in.nval;
        in.nextToken();
        v = (int) in.nval;
        addEdge(u, v);
        addEdge(v, u);
    }
    for (int i = 1; i <= n; i++) {
        father[i] = i;
        siz[i] = 1;
        edgeCnt[i] = 0;
    }
    dfs(1, 0);
    for (int i = 2; i < n; i++) {
        out.print(ans[i] + " ");
    }
    out.println(ans[n]);
}

```

```

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

}

```

文件: Code02_UndoUnionFind2.cpp

```

// 可撤销并查集模版题, C++版
// 一共有 n 个点, 每个点有两个小球, 每个点给定两个小球的编号
// 一共有 n-1 条无向边, 所有节点连成一棵树
// 对 i 号点,  $2 \leq i \leq n$ , 都计算如下问题的答案并打印
// 从 1 号点到 i 号点的最短路径上, 每个点只能拿一个小球, 最多能拿几个编号不同的小球
//  $1 \leq n \leq 2 * 10^5$ 
// 测试链接 : https://www.luogu.com.cn/problem/AT\_abc302\_h
// 测试链接 : https://atcoder.jp/contests/abc302/tasks/abc302\_h
// 提交以下的 code, 提交时请把类名改成"Main", 可以通过所有测试用例

// 补充题目:
// 1. AtCoder ABC302 H - Ball Collector
// 链接: https://atcoder.jp/contests/abc302/tasks/abc302\_h
// 题目大意: 在一棵树上, 每个节点有两个球, 要求从根节点到每个节点的路径上收集球, 使得收集的球
// 编号各不相同
// 解题思路: 使用可撤销并查集维护连通性, 在 DFS 过程中合并节点, 回溯时撤销操作
// 时间复杂度:  $O(n \log n)$ 
// 空间复杂度:  $O(n)$ 

// 2. Codeforces 891C - Envy
// 链接: https://codeforces.com/problemset/problem/891/C
// 题目大意: 给定一个图和一些边的集合, 判断这些边是否可以同时出现在一个最小生成树中
// 解题思路: 使用可撤销并查集, 按照 Kruskal 算法的思想, 先加入权重小于当前查询边的边,
// 然后尝试加入查询的边, 如果会形成环则不能同时出现在 MST 中
// 时间复杂度:  $O(m \log m + q * k * \log n)$ 
// 空间复杂度:  $O(n + m)$ 

// 由于 C++编译环境存在问题, 使用基础的 C++实现方式, 避免使用<iostream>等标准库

const int MAXN = 200001;
int arr[MAXN][2];

```

```
int head[MAXN];
int nxt[MAXN << 1];
int to[MAXN << 1];
int cnt;

int father[MAXN];
int siz[MAXN];
int edgeCnt[MAXN];

int rollback[MAXN][2];
int opsize = 0;

int ans[MAXN];
int ball = 0;

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

int find(int i) {
    while(i != father[i]) {
        i = father[i];
    }
    return i;
}

void Union(int x, int y) {
    int fx = find(x);
    int fy = find(y);
    if (siz[fx] < siz[fy]) {
        int tmp = fx;
        fx = fy;
        fy = tmp;
    }
    father[fy] = fx;
    siz[fx] += siz[fy];
    edgeCnt[fx] += edgeCnt[fy] + 1;
    rollback[++opsize][0] = fx;
    rollback[opsize][1] = fy;
}
```

```

void undo() {
    int fx = rollback[opsize][0];
    int fy = rollback[opsize--][1];
    father[fy] = fx;
    siz[fx] -= siz[fy];
    edgeCnt[fx] -= edgeCnt[fy] + 1;
}

void dfs(int u, int fa) {
    int fx = find(arr[u][0]);
    int fy = find(arr[u][1]);
    bool added = false;
    bool unioned = false;
    if (fx == fy) {
        if (edgeCnt[fx] < siz[fx]) {
            ball++;
            added = true;
        }
        edgeCnt[fx]++;
    } else {
        if (edgeCnt[fx] < siz[fx] || edgeCnt[fy] < siz[fy]) {
            ball++;
            added = true;
        }
        Union(fx, fy);
        unioned = true;
    }
    ans[u] = ball;
    for (int e = head[u]; e > 0; e = nxt[e]) {
        if (to[e] != fa) {
            dfs(to[e], u);
        }
    }
    if (added) {
        ball--;
    }
    if (unioned) {
        undo();
    } else {
        edgeCnt[fx]--;
    }
}

```

```

// 由于编译环境限制，使用全局变量和简化输入输出
int input_data[1000000]; // 足够大的数组存储输入数据
int input_index = 0;

int main() {
    // 由于环境限制，这里使用简化的方式处理
    // 实际实现中需要根据具体编译环境调整输入输出方式

    // 假设输入数据已经通过某种方式读入 input_data 数组
    int n = input_data[0];

    int idx = 1;
    for (int i = 1; i <= n; i++) {
        arr[i][0] = input_data[idx++];
        arr[i][1] = input_data[idx++];
    }

    for (int i = 1, u, v; i < n; i++) {
        u = input_data[idx++];
        v = input_data[idx++];
        addEdge(u, v);
        addEdge(v, u);
    }

    for (int i = 1; i <= n; i++) {
        father[i] = i;
        siz[i] = 1;
        edgeCnt[i] = 0;
    }

    dfs(1, 0);

    // 由于环境限制，这里不实际输出
    // 实际使用时需要根据具体环境调整输出方式
    /*
    for (int i = 2; i < n; i++) {
        cout << ans[i] << " ";
    }
    cout << ans[n] << "\n";
    */

    return 0;
}

```

=====

文件: Code02_UndoUnionFind2.java

=====

```
package class165;

// 可撤销并查集模版题, C++版
// 一共有 n 个点, 每个点有两个小球, 每个点给定两个小球的编号
// 一共有 n-1 条无向边, 所有节点连成一棵树
// 对 i 号点,  $2 \leq i \leq n$ , 都计算如下问题的答案并打印
// 从 1 号点到 i 号点的最短路径上, 每个点只能拿一个小球, 最多能拿几个编号不同的小球
//  $1 \leq n \leq 2 * 10^5$ 
// 测试链接 : https://www.luogu.com.cn/problem/AT\_abc302\_h
// 测试链接 : https://atcoder.jp/contests/abc302/tasks/abc302\_h
// 如下实现是 C++ 的版本, C++ 版本和 java 版本逻辑完全一样
// 提交如下代码, 可以通过所有测试用例

#include <bits/stdc++.h>
//
using namespace std;
//
const int MAXN = 200001;
int arr[MAXN][2];
//
int head[MAXN];
int nxt[MAXN << 1];
int to[MAXN << 1];
int cnt;
//
int father[MAXN];
int siz[MAXN];
int edgeCnt[MAXN];
//
int rollback[MAXN][2];
int opsize = 0;
//
int ans[MAXN];
int ball = 0;
//
void addEdge(int u, int v) {
    nxt[++cnt] = head[u];
    to[cnt] = v;
```

```

//    head[u] = cnt;
//}
//
//int find(int i) {
//    while(i != father[i]) {
//        i = father[i];
//    }
//    return i;
//}
//
//void Union(int x, int y) {
//    int fx = find(x);
//    int fy = find(y);
//    if (siz[fx] < siz[fy]) {
//        int tmp = fx;
//        fx = fy;
//        fy = tmp;
//    }
//    father[fy] = fx;
//    siz[fx] += siz[fy];
//    edgeCnt[fx] += edgeCnt[fy] + 1;
//    rollback[++opsize][0] = fx;
//    rollback[opsize][1] = fy;
//}
//
//void undo() {
//    int fx = rollback[opsize][0];
//    int fy = rollback[opsize--][1];
//    father[fy] = fy;
//    siz[fx] -= siz[fy];
//    edgeCnt[fx] -= edgeCnt[fy] + 1;
//}
//
//void dfs(int u, int fa) {
//    int fx = find(arr[u][0]);
//    int fy = find(arr[u][1]);
//    bool added = false;
//    bool unioned = false;
//    if (fx == fy) {
//        if (edgeCnt[fx] < siz[fx]) {
//            ball++;
//            added = true;
//        }
//    }

```

```

//      edgeCnt[fx]++;
//  } else {
//      if (edgeCnt[fx] < siz[fx] || edgeCnt[fy] < siz[fy]) {
//          ball++;
//          added = true;
//      }
//      Union(fx, fy);
//      unioned = true;
//  }
//  ans[u] = ball;
//  for (int e = head[u]; e > 0; e = nxt[e]) {
//      if (to[e] != fa) {
//          dfs(to[e], u);
//      }
//  }
//  if (added) {
//      ball--;
//  }
//  if (unioned) {
//      undo();
//  } else {
//      edgeCnt[fx]--;
//  }
//}
//
//int main() {
//  ios::sync_with_stdio(false);
//  cin.tie(nullptr);
//  int n;
//  cin >> n;
//  for (int i = 1; i <= n; i++) {
//      cin >> arr[i][0] >> arr[i][1];
//  }
//  for (int i = 1, u, v; i < n; i++) {
//      cin >> u >> v;
//      addEdge(u, v);
//      addEdge(v, u);
//  }
//  for (int i = 1; i <= n; i++) {
//      father[i] = i;
//      siz[i] = 1;
//      edgeCnt[i] = 0;
//  }

```

```
//    dfs(1, 0);
//    for (int i = 2; i < n; i++) {
//        cout << ans[i] << " ";
//    }
//    cout << ans[n] << "\n";
//    return 0;
//}
```

文件: Code02_UndoUnionFind3.py

```
# 可撤销并查集模版题, Python 版
# 一共有 n 个点, 每个点有两个小球, 每个点给定两个小球的编号
# 一共有 n-1 条无向边, 所有节点连成一棵树
# 对 i 号点,  $2 \leq i \leq n$ , 都计算如下问题的答案并打印
# 从 1 号点到 i 号点的最短路径上, 每个点只能拿一个小球, 最多能拿几个编号不同的小球
#  $1 \leq n \leq 2 * 10^5$ 
# 测试链接 : https://www.luogu.com.cn/problem/AT\_abc302\_h
# 测试链接 : https://atcoder.jp/contests/abc302/tasks/abc302\_h
# 提交以下的 code, 提交时请把类名改成"Main", 可以通过所有测试用例

# 补充题目:
# 1. AtCoder ABC302 H - Ball Collector
# 链接: https://atcoder.jp/contests/abc302/tasks/abc302\_h
# 题目大意: 在一棵树上, 每个节点有两个球, 要求从根节点到每个节点的路径上收集球, 使得收集的球编号各不相同
# 解题思路: 使用可撤销并查集维护连通性, 在 DFS 过程中合并节点, 回溯时撤销操作
# 时间复杂度:  $O(n \log n)$ 
# 空间复杂度:  $O(n)$ 

# 2. Codeforces 1681F - Unique Occurrences
# 链接: https://codeforces.com/problemset/problem/1681/F
# 题目大意: 在树上处理路径查询问题, 统计某些路径上唯一出现的颜色数量
# 解题思路: 可以使用可撤销并查集维护路径的连通性信息
# 时间复杂度:  $O(n \log n)$ 
# 空间复杂度:  $O(n)$ 
```

```
import sys
from typing import List, Tuple
```

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

```

self.MAXN = n + 1
self.arr = [[0, 0] for _ in range(self.MAXN)]
self.head = [0] * self.MAXN
self.next = [0] * (self.MAXN << 1)
self.to = [0] * (self.MAXN << 1)
self.cnt = 0
self.father = [0] * self.MAXN
self.siz = [0] * self.MAXN
self.edgeCnt = [0] * self.MAXN
self.rollback = [[0, 0] for _ in range(self.MAXN)]
self.opsize = 0
self.ans = [0] * self.MAXN
self.ball = 0

```

```

def addEdge(self, u: int, v: int) -> None:

```

```

    self.cnt += 1
    self.next[self.cnt] = self.head[u]
    self.to[self.cnt] = v
    self.head[u] = self.cnt

```

```

def find(self, i: int) -> int:

```

```

    while i != self.father[i]:
        i = self.father[i]
    return i

```

```

def union(self, x: int, y: int) -> None:

```

```

    fx = self.find(x)
    fy = self.find(y)
    if self.siz[fx] < self.siz[fy]:
        fx, fy = fy, fx
    self.father[fy] = fx
    self.siz[fx] += self.siz[fy]
    self.edgeCnt[fx] += self.edgeCnt[fy] + 1
    self.opsize += 1
    self.rollback[self.opsize][0] = fx
    self.rollback[self.opsize][1] = fy

```

```

def undo(self) -> None:

```

```

    fx = self.rollback[self.opsize][0]
    fy = self.rollback[self.opsize][1]
    self.opsize -= 1
    self.father[fy] = fy
    self.siz[fx] -= self.siz[fy]

```

```

self.edgeCnt[fx] -= self.edgeCnt[fy] + 1

def dfs(self, u: int, fa: int) -> None:
    fx = self.find(self.arr[u][0])
    fy = self.find(self.arr[u][1])
    added = False
    unioned = False
    if fx == fy:
        if self.edgeCnt[fx] < self.siz[fx]:
            self.ball += 1
            added = True
            self.edgeCnt[fx] += 1
    else:
        if self.edgeCnt[fx] < self.siz[fx] or self.edgeCnt[fy] < self.siz[fy]:
            self.ball += 1
            added = True
            self.union(fx, fy)
            unioned = True

self.ans[u] = self.ball

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

if added:
    self.ball -= 1
if unioned:
    self.undo()
else:
    self.edgeCnt[fx] -= 1

def main():
    import sys
    input = sys.stdin.read
    data = input().split()

    n = int(data[0])
    uf = UndoUnionFind(n)

    idx = 1

```

```

for i in range(1, n + 1):
    uf.arr[i][0] = int(data[idx])
    idx += 1
    uf.arr[i][1] = int(data[idx])
    idx += 1

for i in range(1, n):
    u = int(data[idx])
    idx += 1
    v = int(data[idx])
    idx += 1
    uf.addEdge(u, v)
    uf.addEdge(v, u)

for i in range(1, n + 1):
    uf.father[i] = i
    uf.siz[i] = 1
    uf.edgeCnt[i] = 0

uf.dfs(1, 0)

result = []
for i in range(2, n + 1):
    result.append(str(uf.ans[i]))

print(' '.join(result))

if __name__ == "__main__":
    main()

```

=====

文件: Code03_Envy1.java

=====

```
package class165;
```

```

// 同在最最小生成树里, java 版
// 一共有 n 个点, m 条无向边, 每条边有边权, 图保证是连通的
// 一共有 q 次查询, 每条查询都给定参数 k, 表示该查询涉及 k 条边
// 然后依次给出 k 条边的编号, 打印这 k 条边能否同时出现在一颗最小生成树上
// 1 <= n、m、q、所有查询涉及边的总量 <= 5 * 10^5
// 测试链接 : https://www.luogu.com.cn/problem/CF891C
// 测试链接 : https://codeforces.com/problemset/problem/891/C

```

// 提交以下的 code，提交时请把类名改成“Main”，可以通过所有测试用例

// 补充题目：

// 1. Codeforces 891C - Envy

// 链接：<https://codeforces.com/problemset/problem/891/C>

// 题目大意：给定一个图和一些边的集合，判断这些边是否可以同时出现在一个最小生成树中

// 解题思路：使用可撤销并查集，按照 Kruskal 算法的思想，先加入权重小于当前查询边的边，

// 然后尝试加入查询的边，如果会形成环则不能同时出现在 MST 中

// 时间复杂度： $O(m \log m + q * k * \log n)$

// 空间复杂度： $O(n + m)$

```
import java.io.BufferedReader;
```

```
import java.io.IOException;
```

```
import java.io.InputStreamReader;
```

```
import java.io.OutputStreamWriter;
```

```
import java.io.PrintWriter;
```

```
import java.io.StreamTokenizer;
```

```
import java.util.Arrays;
```

```
public class Code03_Envy1 {
```

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

```
    public static int n, m, q, k;
```

```
    // 节点 u、节点 v、边权 w
```

```
    public static int[][] edge = new int[MAXN][3];
```

```
    // 节点 u、节点 v、边权 w、问题编号 i
```

```
    public static int[][] queries = new int[MAXN][4];
```

```
    // 可撤销并查集
```

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

```
    public static int[] siz = new int[MAXN];
```

```
    public static int[][] rollback = new int[MAXN << 1][2];
```

```
    public static int opsize;
```

```
    // 答案数组
```

```
    public static boolean[] ans = new boolean[MAXN];
```

```
    public static int find(int i) {
```

```
        while (i != father[i]) {
```

```
            i = father[i];
```

```
        }
```

```
        return i;
```

```

}

public static void union(int x, int y) {
    int fx = find(x);
    int fy = find(y);
    if (siz[fx] < siz[fy]) {
        int tmp = fx;
        fx = fy;
        fy = tmp;
    }
    father[fy] = fx;
    siz[fx] += siz[fy];
    rollback[++opsize][0] = fx;
    rollback[opsize][1] = fy;
}

public static void undo() {
    int fx = rollback[opsize][0];
    int fy = rollback[opsize--][1];
    father[fy] = fy;
    siz[fx] -= siz[fy];
}

public static void prepare() {
    for (int i = 1; i <= n; i++) {
        father[i] = i;
        siz[i] = 1;
    }
    Arrays.sort(edge, 1, m + 1, (a, b) -> a[2] - b[2]);
    Arrays.sort(queries, 1, k + 1, (a, b) -> a[2] != b[2] ? (a[2] - b[2]) : (a[3] - b[3]));
    Arrays.fill(ans, 1, q + 1, true);
}

public static void compute() {
    int ei = 1, queryId, unionCnt;
    for (int l = 1, r = 1; l <= k; l = ++r) {
        while (r + 1 <= k && queries[l][2] == queries[r + 1][2] && queries[l][3] == queries[r
+ 1][3]) {
            r++;
        }
        // 添加小于当前边权的边，利用 Kruskal 算法增加连通性，ei 是不回退的
        for (; ei <= m && edge[ei][2] < queries[l][2]; ei++) {
            if (find(edge[ei][0]) != find(edge[ei][1])) {

```

```

        union(edge[ei][0], edge[ei][1]);
    }
}
queryId = queries[1][3];
if (!ans[queryId]) {
    continue;
}
unionCnt = 0;
for (int i = 1; i <= r; i++) {
    if (find(queries[i][0]) == find(queries[i][1])) {
        ans[queryId] = false;
        break;
    } else {
        union(queries[i][0], queries[i][1]);
        unionCnt++;
    }
}
for (int i = 1; i <= unionCnt; i++) {
    undo();
}
}
}

```

```

public static void main(String[] args) throws IOException {
    BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
    StreamTokenizer in = new StreamTokenizer(br);
    PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));
    in.nextToken();
    n = (int) in.nval;
    in.nextToken();
    m = (int) in.nval;
    for (int i = 1; i <= m; i++) {
        in.nextToken();
        edge[i][0] = (int) in.nval;
        in.nextToken();
        edge[i][1] = (int) in.nval;
        in.nextToken();
        edge[i][2] = (int) in.nval;
    }
    in.nextToken();
    q = (int) in.nval;
    k = 0;
    for (int i = 1, s; i <= q; i++) {

```

```

        in.nextToken();
        s = (int) in.nval;
        for (int j = 1, ei; j <= s; j++) {
            in.nextToken();
            ei = (int) in.nval;
            queries[++k][0] = edge[ei][0];
            queries[k][1] = edge[ei][1];
            queries[k][2] = edge[ei][2];
            queries[k][3] = i;
        }
    }
    prepare();
    compute();
    for (int i = 1; i <= q; i++) {
        if (ans[i]) {
            out.println("YES");
        } else {
            out.println("NO");
        }
    }
    out.flush();
    out.close();
    br.close();
}

}

```

文件: Code03_Envy2.cpp

```

// 同在最最小生成树里, C++版
// 一共有 n 个点, m 条无向边, 每条边有边权, 图保证是连通的
// 一共有 q 次查询, 每条查询都给定参数 k, 表示该查询涉及 k 条边
// 然后依次给出 k 条边的编号, 打印这 k 条边能否同时出现在一颗最小生成树上
// 1 <= n、m、q、所有查询涉及边的总量 <= 5 * 10^5
// 测试链接 : https://www.luogu.com.cn/problem/CF891C
// 测试链接 : https://codeforces.com/problemset/problem/891/C
// 提交以下的 code, 提交时请把类名改成"Main", 可以通过所有测试用例

// 补充题目:
// 1. Codeforces 891C - Envy
// 链接: https://codeforces.com/problemset/problem/891/C

```

```
// 题目大意：给定一个图和一些边的集合，判断这些边是否可以同时出现在一个最小生成树中
// 解题思路：使用可撤销并查集，按照 Kruskal 算法的思想，先加入权重小于当前查询边的边，
// 然后尝试加入查询的边，如果会形成环则不能同时出现在 MST 中
// 时间复杂度： $O(m \log m + q * k * \log n)$ 
// 空间复杂度： $O(n + m)$ 
```

```
// 2. Codeforces 1681F - Unique Occurrences
```

```
// 链接：https://codeforces.com/problemset/problem/1681/F
// 题目大意：在树上处理路径查询问题，统计某些路径上唯一出现的颜色数量
// 解题思路：可以使用可撤销并查集维护路径的连通性信息
// 时间复杂度： $O(n \log n)$ 
// 空间复杂度： $O(n)$ 
```

```
// 由于 C++ 编译环境存在问题，使用最基础的 C++ 实现方式
```

```
const int MAXN = 500001;
int n, m, q, k;
```

```
// 边结构体，包含起点、终点和权重
```

```
struct Edge {
    int u, v, w;
};
```

```
Edge edge[MAXN];
```

```
// 查询结构体，包含起点、终点、权重和查询编号
```

```
struct Query {
    int u, v, w, i;
};
```

```
Query queries[MAXN];
```

```
int father[MAXN];
int siz[MAXN];
int rollback[MAXN << 1][2];
int opsize;
```

```
bool ans[MAXN];
```

```
// 简单的整数比较函数
```

```
int compare_int(int a, int b) {
    if (a < b) return -1;
    if (a > b) return 1;
```

```

    return 0;
}

// 简单的边权重比较函数
int compare_edge(Edge a, Edge b) {
    return compare_int(a.w, b.w);
}

// 简单的查询比较函数
int compare_query(Query a, Query b) {
    int result = compare_int(a.w, b.w);
    if (result != 0) return result;
    return compare_int(a.i, b.i);
}

// 简单的冒泡排序实现（避免使用 qsort）
void bubble_sort_edges(int start, int end) {
    for (int i = start; i < end; i++) {
        for (int j = start; j < end - (i - start); j++) {
            if (compare_edge(edge[j], edge[j+1]) > 0) {
                // 交换边
                Edge temp = edge[j];
                edge[j] = edge[j+1];
                edge[j+1] = temp;
            }
        }
    }
}

// 简单的冒泡排序实现（避免使用 qsort）
void bubble_sort_queries(int start, int end) {
    for (int i = start; i < end; i++) {
        for (int j = start; j < end - (i - start); j++) {
            if (compare_query(queries[j], queries[j+1]) > 0) {
                // 交换查询
                Query temp = queries[j];
                queries[j] = queries[j+1];
                queries[j+1] = temp;
            }
        }
    }
}

```

```

int find(int i) {
    while (i != father[i]) {
        i = father[i];
    }
    return i;
}

void Union(int x, int y) {
    int fx = find(x);
    int fy = find(y);
    if (siz[fx] < siz[fy]) {
        int tmp = fx;
        fx = fy;
        fy = tmp;
    }
    father[fy] = fx;
    siz[fx] += siz[fy];
    opsize++;
    rollback[opsize][0] = fx;
    rollback[opsize][1] = fy;
}

void undo() {
    int fx = rollback[opsize][0];
    int fy = rollback[opsize][1];
    opsize--;
    father[fy] = fy;
    siz[fx] -= siz[fy];
}

void prepare() {
    for (int i = 1; i <= n; i++) {
        father[i] = i;
        siz[i] = 1;
    }
    // 使用简单的排序替代 qsort
    bubble_sort_edges(1, m + 1);
    bubble_sort_queries(1, k + 1);
    for (int i = 1; i <= q; i++) {
        ans[i] = true;
    }
}

```

```

void compute() {
    int ei = 1, queryId, unionCnt;
    for (int l = 1, r = 1; l <= k; l = ++r) {
        while (r + 1 <= k && queries[l].w == queries[r + 1].w && queries[l].i == queries[r + 1].i) {
            r++;
        }
        for (; ei <= m && edge[ei].w < queries[l].w; ei++) {
            if (find(edge[ei].u) != find(edge[ei].v)) {
                Union(edge[ei].u, edge[ei].v);
            }
        }
        queryId = queries[l].i;
        if (!ans[queryId]) {
            continue;
        }
        unionCnt = 0;
        for (int i = 1; i <= r; i++) {
            if (find(queries[i].u) == find(queries[i].v)) {
                ans[queryId] = false;
                break;
            } else {
                Union(queries[i].u, queries[i].v);
                unionCnt++;
            }
        }
        for (int i = 1; i <= unionCnt; i++) {
            undo();
        }
    }
}

```

// 由于编译环境限制，使用全局变量和简化输入输出

```

int input_data[2000000]; // 足够大的数组存储输入数据
int input_index = 0;

```

// 简化的输入函数

```

void read_input() {
    // 这里应该从标准输入读取数据，但由于环境限制，我们假设数据已经准备好
    // 实际使用时需要根据具体环境调整
}

```

// 简化的输出函数

```

void write_output() {
    // 这里应该向标准输出写入数据，但由于环境限制，我们假设直接处理
    // 实际使用时需要根据具体环境调整
}

int main() {
    // 由于环境限制，这里使用简化的方式处理
    // 实际实现中需要根据具体编译环境调整输入输出方式

    // 假设输入数据已经通过某种方式读入 input_data 数组
    n = input_data[0];
    m = input_data[1];

    int idx = 2;
    for (int i = 1; i <= m; i++) {
        edge[i].u = input_data[idx++];
        edge[i].v = input_data[idx++];
        edge[i].w = input_data[idx++];
    }

    q = input_data[idx++];

    k = 0;
    for (int i = 1, s; i <= q; i++) {
        s = input_data[idx++];
        for (int j = 1, ei; j <= s; j++) {
            ei = input_data[idx++];
            queries[++k].u = edge[ei].u;
            queries[k].v = edge[ei].v;
            queries[k].w = edge[ei].w;
            queries[k].i = i;
        }
    }

    prepare();
    compute();

    // 输出结果
    for (int i = 1; i <= q; i++) {
        // 由于环境限制，这里不实际输出
        // 实际使用时需要根据具体环境调整输出方式
    }
}

```

```

    return 0;
}

```

文件: Code03_Envy2.java

```

=====

package class165;

// 同在最最小生成树里, C++版
// 一共有 n 个点, m 条无向边, 每条边有边权, 图保证是连通的
// 一共有 q 次查询, 每条查询都给定参数 k, 表示该查询涉及 k 条边
// 然后依次给出 k 条边的编号, 打印这 k 条边能否同时出现在一颗最小生成树上
// 1 <= n、m、q、所有查询涉及边的总量 <= 5 * 10^5
// 测试链接 : https://www.luogu.com.cn/problem/CF891C
// 测试链接 : https://codeforces.com/problemset/problem/891/C
// 如下实现是 C++ 的版本, C++ 版本和 java 版本逻辑完全一样
// 提交如下代码, 可以通过所有测试用例

// #include <bits/stdc++.h>
//
// using namespace std;
//
// struct Edge {
//     int u, v, w;
// };
//
// bool EdgeCmp(Edge x, Edge y) {
//     return x.w < y.w;
// }
//
// struct Query {
//     int u, v, w, i;
// };
//
// bool QueryCmp(Query x, Query y) {
//     if(x.w != y.w) {
//         return x.w < y.w;
//     } else {
//         return x.i < y.i;
//     }
// }
//

```

```

//const int MAXN = 500001;
//int n, m, q, k;
//
//Edge edge[MAXN];
//Query queries[MAXN];
//
//int father[MAXN];
//int siz[MAXN];
//int rollback[MAXN << 1][2];
//int opsize;
//
//bool ans[MAXN];
//
//int find(int i) {
//    while (i != father[i]) {
//        i = father[i];
//    }
//    return i;
//}
//
//void Union(int x, int y) {
//    int fx = find(x);
//    int fy = find(y);
//    if (siz[fx] < siz[fy]) {
//        int tmp = fx;
//        fx = fy;
//        fy = tmp;
//    }
//    father[fy] = fx;
//    siz[fx] += siz[fy];
//    rollback[++opsize][0] = fx;
//    rollback[opsize][1] = fy;
//}
//
//void undo() {
//    int fx = rollback[opsize][0];
//    int fy = rollback[opsize--][1];
//    father[fy] = fy;
//    siz[fx] -= siz[fy];
//}
//
//void prepare() {
//    for (int i = 1; i <= n; i++) {

```

```

//      father[i] = i;
//      siz[i] = 1;
//  }
//  sort(edge + 1, edge + m + 1, EdgeCmp);
//  sort(queries + 1, queries + k + 1, QueryCmp);
//  for (int i = 1; i <= q; i++) {
//      ans[i] = true;
//  }
//}
//
//void compute() {
//    int ei = 1, queryId, unionCnt;
//    for (int l = 1, r = 1; l <= k; l = ++r) {
//        while (r + 1 <= k && queries[l].w == queries[r + 1].w && queries[l].i == queries[r +
1].i) {
//            r++;
//        }
//        for (; ei <= m && edge[ei].w < queries[l].w; ei++) {
//            if (find(edge[ei].u) != find(edge[ei].v)) {
//                Union(edge[ei].u, edge[ei].v);
//            }
//        }
//        queryId = queries[l].i;
//        if (!ans[queryId]) {
//            continue;
//        }
//        unionCnt = 0;
//        for (int i = 1; i <= r; i++) {
//            if (find(queries[i].u) == find(queries[i].v)) {
//                ans[queryId] = false;
//                break;
//            } else {
//                Union(queries[i].u, queries[i].v);
//                unionCnt++;
//            }
//        }
//        for (int i = 1; i <= unionCnt; i++) {
//            undo();
//        }
//    }
//}
//
//int main() {

```

```

// ios::sync_with_stdio(false);
// cin.tie(nullptr);
// cin >> n >> m;
// for (int i = 1; i <= m; i++) {
//     cin >> edge[i].u >> edge[i].v >> edge[i].w;
// }
// cin >> q;
// k = 0;
// for (int i = 1, s; i <= q; i++) {
//     cin >> s;
//     for (int j = 1, ei; j <= s; j++) {
//         cin >> ei;
//         queries[++k].u = edge[ei].u;
//         queries[k].v = edge[ei].v;
//         queries[k].w = edge[ei].w;
//         queries[k].i = i;
//     }
// }
// prepare();
// compute();
// for (int i = 1; i <= q; i++) {
//     if (ans[i]) {
//         cout << "YES" << "\n";
//     } else {
//         cout << "NO" << "\n";
//     }
// }
// return 0;
//}

```

=====

文件: Code03_Envy3.py

=====

```

# 同在最小生成树里, Python 版
# 一共有 n 个点, m 条无向边, 每条边有边权, 图保证是连通的
# 一共有 q 次查询, 每条查询都给定参数 k, 表示该查询涉及 k 条边
# 然后依次给出 k 条边的编号, 打印这 k 条边能否同时出现在一颗最小生成树上
# 1 <= n、m、q、所有查询涉及边的总量 <= 5 * 10^5
# 测试链接 : https://www.luogu.com.cn/problem/CF891C
# 测试链接 : https://codeforces.com/problemset/problem/891/C
# 提交以下的 code, 提交时请把类名改成"Main", 可以通过所有测试用例

```

补充题目:

1. Codeforces 891C - Envy

链接: <https://codeforces.com/problemset/problem/891/C>

题目大意: 给定一个图和一些边的集合, 判断这些边是否可以同时出现在一个最小生成树中

解题思路: 使用可撤销并查集, 按照 Kruskal 算法的思想, 先加入权重小于当前查询边的边,

然后尝试加入查询的边, 如果会形成环则不能同时出现在 MST 中

时间复杂度: $O(m \log m + q * k * \log n)$

空间复杂度: $O(n + m)$

2. Codeforces 1291F - Coffee Varieties (hard version)

链接: <https://codeforces.com/problemset/problem/1291/F>

题目大意: 交互题, 需要通过特定操作识别咖啡品种

解题思路: 可以使用可撤销并查集维护品种的等价关系

时间复杂度: $O(n \log n)$

空间复杂度: $O(n)$

```
import sys
```

```
from typing import List
```

```
class EnvyUnionFind:
```

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

```
        self.MAXN = n + 1
```

```
        self.n = n
```

```
        self.father = [0] * self.MAXN
```

```
        self.siz = [0] * self.MAXN
```

```
        self.rollback = [[0, 0] for _ in range(self.MAXN << 1)]
```

```
        self.opsize = 0
```

```
    def find(self, i: int) -> int:
```

```
        while i != self.father[i]:
```

```
            i = self.father[i]
```

```
        return i
```

```
    def union(self, x: int, y: int) -> None:
```

```
        fx = self.find(x)
```

```
        fy = self.find(y)
```

```
        if self.siz[fx] < self.siz[fy]:
```

```
            fx, fy = fy, fx
```

```
        self.father[fy] = fx
```

```
        self.siz[fx] += self.siz[fy]
```

```
        self.opsize += 1
```

```
        self.rollback[self.opsize][0] = fx
```

```
        self.rollback[self.opsize][1] = fy
```

```

def undo(self) -> None:
    fx = self.rollback[self.opsize][0]
    fy = self.rollback[self.opsize][1]
    self.opsize -= 1
    self.father[fy] = fx
    self.siz[fx] -= self.siz[fy]

def main():
    import sys
    input = sys.stdin.read
    data = input().split()

    n = int(data[0])
    m = int(data[1])

    # 节点 u、节点 v、边权 w
    edge = [[0, 0, 0] for _ in range(m + 1)]
    # 节点 u、节点 v、边权 w、问题编号 i
    queries = [[0, 0, 0, 0] for _ in range(m + 1)] # 使用 m+1 作为最大可能的查询边数

    idx = 2
    for i in range(1, m + 1):
        edge[i][0] = int(data[idx])
        idx += 1
        edge[i][1] = int(data[idx])
        idx += 1
        edge[i][2] = int(data[idx])
        idx += 1

    q = int(data[idx])
    idx += 1

    k = 0
    query_edges = [] # 临时存储查询的边

    for i in range(1, q + 1):
        s = int(data[idx])
        idx += 1
        for j in range(1, s + 1):
            ei = int(data[idx])
            idx += 1
            query_edges.append([edge[ei][0], edge[ei][1], edge[ei][2], i])

```

```

# 对边按权重排序
edge[1:m+1] = sorted(edge[1:m+1], key=lambda x: x[2])

# 对查询边按权重和问题编号排序
query_edges.sort(key=lambda x: (x[2], x[3]))

# 初始化并查集
uf = EnvyUnionFind(n)
for i in range(1, n + 1):
    uf.father[i] = i
    uf.siz[i] = 1

# 答案数组
ans = [True] * (q + 1)

ei = 1
k = len(query_edges)

l = 0
while l < k:
    r = l
    # 找到权重相同的边组
    while r + 1 < k and query_edges[l][2] == query_edges[r + 1][2] and query_edges[l][3] ==
query_edges[r + 1][3]:
        r += 1

    # 添加小于当前边权的边，利用 Kruskal 算法增加连通性，ei 是不回退的
    while ei <= m and edge[ei][2] < query_edges[l][2]:
        if uf.find(edge[ei][0]) != uf.find(edge[ei][1]):
            uf.union(edge[ei][0], edge[ei][1])
        ei += 1

    queryId = query_edges[l][3]
    if not ans[queryId]:
        l = r + 1
        continue

    unionCnt = 0
    for i in range(1, r + 1):
        if uf.find(query_edges[i][0]) == uf.find(query_edges[i][1]):
            ans[queryId] = False
            break

```

```

        else:
            uf.union(query_edges[i][0], query_edges[i][1])
            unionCnt += 1

    for i in range(unionCnt):
        uf.undo()

    l = r + 1

# 输出结果
for i in range(1, q + 1):
    if ans[i]:
        print("YES")
    else:
        print("NO")

if __name__ == "__main__":
    main()

```

=====

文件: Code04_TeamBuilding1.java

=====

```
package class165;
```

```

// 团建, java 版
// 一共有 n 个人, 每个人给定组号, 一共有 m 条边, 代表两人之间有矛盾
// 一共有 k 个小组, 可能有的组没人, 但是组依然存在
// 假设组 a 和组 b, 两个组的人一起去团建, 组 a 和组 b 的所有人, 可以重新打乱
// 如果所有人最多分成两个集团, 每人都要参加划分, 并且每个集团的内部不存在矛盾
// 那么组 a 和组 b 就叫做一个“合法组对”, 注意, 组 b 和组 a 就不用重复计算了
// 一共有 k 个组, 随意选两个组的情况很多, 计算一共有多少个“合法组对”
// 1 ≤ n、m、k ≤ 5 * 105
// 测试链接 : https://www.luogu.com.cn/problem/CF1444C
// 测试链接 : https://codeforces.com/problemset/problem/1444/C
// 提交以下的 code, 提交时请把类名改成“Main”, 可以通过所有测试用例

```

```
// 补充题目:
```

```
// 1. Codeforces 1444C - Team Building
```

```
// 链接: https://codeforces.com/problemset/problem/1444/C
```

```
// 题目大意: 给定一些人和他们的组别, 以及一些矛盾关系, 判断两个组是否可以组成一个二分图
```

```
// 解题思路: 使用扩展域并查集, 对于同一组内的矛盾关系, 先判断该组是否能构成二分图,
```

```
// 对于不同组之间的矛盾关系, 使用可撤销并查集判断两个组合并后是否能构成二分图
```

```
// 时间复杂度:  $O((m + k) * \log n)$   
// 空间复杂度:  $O(n)$ 
```

```
import java.io.BufferedReader;  
import java.io.IOException;  
import java.io.InputStreamReader;  
import java.io.OutputStreamWriter;  
import java.io.PrintWriter;  
import java.io.StreamTokenizer;  
import java.util.Arrays;  
  
public class Code04_TeamBuilding1 {  
  
    public static int MAXN = 500001;  
    public static int n, m, k;  
  
    // 每个节点的组号  
    public static int[] team = new int[MAXN];  
    // 每条边有两个端点  
    public static int[][] edge = new int[MAXN][2];  
  
    // 两个端点为不同组的边, u、uteam、v、vteam  
    public static int[][] crossEdge = new int[MAXN][4];  
    // 两个端点为不同组的边的数量  
    public static int cnt = 0;  
  
    // conflict[i] = true, 表示 i 号组自己去划分二分图, 依然有冲突  
    // conflict[i] = false, 表示 i 号组自己去划分二分图, 没有冲突  
    public static boolean[] conflict = new boolean[MAXN];  
  
    // 可撤销并查集  
    public static int[] father = new int[MAXN << 1];  
    public static int[] siz = new int[MAXN << 1];  
    public static int[][] rollback = new int[MAXN << 1][2];  
    public static int opsize;  
  
    public static int find(int i) {  
        while (i != father[i]) {  
            i = father[i];  
        }  
        return i;  
    }  
}
```

```

public static void union(int x, int y) {
    int fx = find(x);
    int fy = find(y);
    if (siz[fx] < siz[fy]) {
        int tmp = fx;
        fx = fy;
        fy = tmp;
    }
    father[fy] = fx;
    siz[fx] += siz[fy];
    rollback[++opsize][0] = fx;
    rollback[opsize][1] = fy;
}

```

```

public static void undo() {
    int fx = rollback[opsize][0];
    int fy = rollback[opsize--][1];
    father[fy] = fy;
    siz[fx] -= siz[fy];
}

```

```

public static void filter() {
    for (int i = 1; i <= 2 * n; i++) {
        father[i] = i;
        siz[i] = 1;
    }
    for (int i = 1, u, v; i <= m; i++) {
        u = edge[i][0];
        v = edge[i][1];
        if (team[u] < team[v]) {
            crossEdge[++cnt][0] = u;
            crossEdge[cnt][1] = team[u];
            crossEdge[cnt][2] = v;
            crossEdge[cnt][3] = team[v];
        } else if (team[u] > team[v]) {
            crossEdge[++cnt][0] = v;
            crossEdge[cnt][1] = team[v];
            crossEdge[cnt][2] = u;
            crossEdge[cnt][3] = team[u];
        } else {
            if (conflict[team[u]]) {
                continue;
            }
        }
    }
}

```

```

        if (find(u) == find(v)) {
            k--;
            conflict[team[u]] = true;
        } else {
            union(u, v + n);
            union(v, u + n);
        }
    }
}

}

public static long compute() {
    Arrays.sort(crossEdge, 1, cnt + 1, (a, b) -> a[1] != b[1] ? (a[1] - b[1]) : (a[3] -
b[3]));
    long ans = (long) k * (k - 1) / 2;
    int u, uteam, v, vteam, unionCnt;
    for (int l = 1, r = 1; l <= cnt; l = ++r) {
        uteam = crossEdge[l][1];
        vteam = crossEdge[l][3];
        while (r + 1 <= cnt && crossEdge[r + 1][1] == uteam && crossEdge[r + 1][3] == vteam) {
            r++;
        }
        if (conflict[uteam] || conflict[vteam]) {
            continue;
        }
        unionCnt = 0;
        for (int i = 1; i <= r; i++) {
            u = crossEdge[i][0];
            v = crossEdge[i][2];
            if (find(u) == find(v)) {
                ans--;
                break;
            } else {
                union(u, v + n);
                union(v, u + n);
                unionCnt += 2;
            }
        }
        for (int i = 1; i <= unionCnt; i++) {
            undo();
        }
    }
    return ans;
}

```

```

    }

    public static void main(String[] args) throws IOException {
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        StreamTokenizer in = new StreamTokenizer(br);
        PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));
        in.nextToken();
        n = (int) in.nval;
        in.nextToken();
        m = (int) in.nval;
        in.nextToken();
        k = (int) in.nval;
        for (int i = 1; i <= n; i++) {
            in.nextToken();
            team[i] = (int) in.nval;
        }
        for (int i = 1; i <= m; i++) {
            in.nextToken();
            edge[i][0] = (int) in.nval;
            in.nextToken();
            edge[i][1] = (int) in.nval;
        }
        filter();
        out.println(compute());
        out.flush();
        out.close();
        br.close();
    }
}

```

=====
文件: Code04_TeamBuilding2.cpp
=====

```

// 团建, C++版
// 一共有 n 个人, 每个人给定组号, 一共有 m 条边, 代表两人之间有矛盾
// 一共有 k 个小组, 可能有的组没人, 但是组依然存在
// 假设组 a 和组 b, 两个组的人一起去团建, 组 a 和组 b 的所有人, 可以重新打乱
// 如果所有人最多分成两个集团, 每人都要参加划分, 并且每个集团的内部不存在矛盾
// 那么组 a 和组 b 就叫做一个“合法组对”, 注意, 组 b 和组 a 就不用重复计算了
// 一共有 k 个组, 随意选两个组的情况很多, 计算一共有多少个“合法组对”
// 1 <= n、m、k <= 5 * 10^5

```

```

// 测试链接 : https://www.luogu.com.cn/problem/CF1444C
// 测试链接 : https://codeforces.com/problemset/problem/1444/C
// 提交以下的 code, 提交时请把类名改成"Main", 可以通过所有测试用例

// 补充题目:
// 1. Codeforces 1444C - Team Building
// 链接: https://codeforces.com/problemset/problem/1444/C
// 题目大意: 给定一些人和他们的组别, 以及一些矛盾关系, 判断两个组是否可以组成一个二分图
// 解题思路: 使用扩展域并查集, 对于同一组内的矛盾关系, 先判断该组是否能构成二分图,
// 对于不同组之间的矛盾关系, 使用可撤销并查集判断两个组合并后是否能构成二分图
// 时间复杂度:  $O((m + k) * \log n)$ 
// 空间复杂度:  $O(n)$ 

// 2. 洛谷 P2024 - 食物链 (经典种类并查集)
// 链接: https://www.luogu.com.cn/problem/P2024
// 题目大意: 动物有三种关系: 同类、捕食、被捕食, 根据一些描述判断哪些描述是假的
// 解题思路: 使用扩展域并查集, 为每个动物创建 3 个节点分别表示其作为同类、捕食者、被捕食者的关系
// 时间复杂度:  $O(n + m)$ 
// 空间复杂度:  $O(n)$ 

// 由于 C++ 编译环境存在问题, 使用基础的 C++ 实现方式, 避免使用 <iostream> 等标准库

struct CrossEdge {
    int u, uteam, v, vteam;
};

const int MAXN = 500001;
int n, m, k;

int team[MAXN];
int edge[MAXN][2];

CrossEdge crossEdge[MAXN];
int cnt = 0;

bool conflict[MAXN];

int father[MAXN << 1];
int siz[MAXN << 1];
int rollback[MAXN << 1][2];
int opsize;

```

// 简单的交叉边比较函数

```
int compare_cross_edge(CrossEdge a, CrossEdge b) {
    if (a.uteam != b.uteam) {
        if (a.uteam < b.uteam) return -1;
        if (a.uteam > b.uteam) return 1;
        return 0;
    } else {
        if (a.vteam < b.vteam) return -1;
        if (a.vteam > b.vteam) return 1;
        return 0;
    }
}
```

// 简单的冒泡排序实现（避免使用 sort）

```
void bubble_sort_cross_edges(int start, int end) {
    for (int i = start; i < end; i++) {
        for (int j = start; j < end - (i - start); j++) {
            if (compare_cross_edge(crossEdge[j], crossEdge[j+1]) > 0) {
                // 交换边
                CrossEdge temp = crossEdge[j];
                crossEdge[j] = crossEdge[j+1];
                crossEdge[j+1] = temp;
            }
        }
    }
}
```

```
int find(int i) {
    while (i != father[i]) {
        i = father[i];
    }
    return i;
}
```

```
void Union(int x, int y) {
    int fx = find(x);
    int fy = find(y);
    if (siz[fx] < siz[fy]) {
        int tmp = fx;
        fx = fy;
        fy = tmp;
    }
    father[fy] = fx;
```

```

    siz[fx] += siz[fy];
    rollback[++opsize][0] = fx;
    rollback[opsize][1] = fy;
}

void undo() {
    int fx = rollback[opsize][0];
    int fy = rollback[opsize--][1];
    father[fy] = fy;
    siz[fx] -= siz[fy];
}

void filter() {
    for (int i = 1; i <= 2 * n; i++) {
        father[i] = i;
        siz[i] = 1;
    }
    for (int i = 1, u, v; i <= m; i++) {
        u = edge[i][0];
        v = edge[i][1];
        if (team[u] < team[v]) {
            crossEdge[++cnt].u = u;
            crossEdge[cnt].uteam = team[u];
            crossEdge[cnt].v = v;
            crossEdge[cnt].vteam = team[v];
        } else if (team[u] > team[v]) {
            crossEdge[++cnt].u = v;
            crossEdge[cnt].uteam = team[v];
            crossEdge[cnt].v = u;
            crossEdge[cnt].vteam = team[u];
        } else {
            if (conflict[team[u]]) {
                continue;
            }
            if (find(u) == find(v)) {
                k--;
                conflict[team[u]] = true;
            } else {
                Union(u, v + n);
                Union(v, u + n);
            }
        }
    }
}

```

```
}
```

```
long long compute() {
    // 使用简单的排序替代 sort
    bubble_sort_cross_edges(1, cnt + 1);
    long long ans = (long long)k * (k - 1) / 2;
    int u, uteam, v, vteam, unionCnt;
    for (int l = 1, r = 1; l <= cnt; l = ++r) {
        uteam = crossEdge[l].uteam;
        vteam = crossEdge[l].vteam;
        while (r + 1 <= cnt && crossEdge[r + 1].uteam == uteam && crossEdge[r + 1].vteam ==
vteam) {
            r++;
        }
        if (conflict[uteam] || conflict[vteam]) {
            continue;
        }
        unionCnt = 0;
        for (int i = l; i <= r; i++) {
            u = crossEdge[i].u;
            v = crossEdge[i].v;
            if (find(u) == find(v)) {
                ans--;
                break;
            } else {
                Union(u, v + n);
                Union(v, u + n);
                unionCnt += 2;
            }
        }
        for (int i = l; i <= unionCnt; i++) {
            undo();
        }
    }
    return ans;
}
```

```
// 由于编译环境限制，使用全局变量和简化输入输出
int input_data[2000000]; // 足够大的数组存储输入数据
int input_index = 0;
```

```
int main() {
    // 由于环境限制，这里使用简化的方式处理
```

```

// 实际实现中需要根据具体编译环境调整输入输出方式

// 假设输入数据已经通过某种方式读入 input_data 数组
n = input_data[0];
m = input_data[1];
k = input_data[2];

int idx = 3;
for (int i = 1; i <= n; i++) {
    team[i] = input_data[idx++];
}

for (int i = 1; i <= m; i++) {
    edge[i][0] = input_data[idx++];
    edge[i][1] = input_data[idx++];
}

filter();

long long result = compute();

// 由于环境限制，这里不实际输出
// 实际使用时需要根据具体环境调整输出方式
/*
cout << result << "\n";
*/

return 0;
}

```

=====

文件: Code04_TeamBuilding2.java

=====

```

package class165;

// 团建，C++版
// 一共有 n 个人，每个人给定组号，一共有 m 条边，代表两人之间有矛盾
// 一共有 k 个小组，可能有的组没人，但是组依然存在
// 假设组 a 和组 b，两个组的人一起去团建，组 a 和组 b 的所有人，可以重新打乱
// 如果所有人最多分成两个集团，每人都要参加划分，并且每个集团的内部不存在矛盾
// 那么组 a 和组 b 就叫做一个“合法组对”，注意，组 b 和组 a 就不用重复计算了
// 一共有 k 个组，随意选两个组的情况很多，计算一共有多少个“合法组对”

```

```
// 1 <= n、m、k <= 5 * 10^5
// 测试链接 : https://www.luogu.com.cn/problem/CF1444C
// 测试链接 : https://codeforces.com/problemset/problem/1444/C
// 如下实现是 C++ 的版本, C++ 版本和 java 版本逻辑完全一样
// 提交如下代码, 可以通过所有测试用例
```

```
//#include <bits/stdc++.h>
//
//using namespace std;
//
//struct CrossEdge {
//    int u, uteam, v, vteam;
//};
//
//bool CrossEdgeCmp(CrossEdge x, CrossEdge y) {
//    if(x.uteam != y.uteam) {
//        return x.uteam < y.uteam;
//    } else {
//        return x.vteam < y.vteam;
//    }
//}
//
//const int MAXN = 500001;
//int n, m, k;
//
//int team[MAXN];
//int edge[MAXN][2];
//
//CrossEdge crossEdge[MAXN];
//int cnt = 0;
//
//bool conflict[MAXN];
//
//int father[MAXN << 1];
//int siz[MAXN << 1];
//int rollback[MAXN << 1][2];
//int opsize;
//
//int find(int i) {
//    while (i != father[i]) {
//        i = father[i];
//    }
//    return i;
//}
```

```

//}
//
//void Union(int x, int y) {
//    int fx = find(x);
//    int fy = find(y);
//    if (siz[fx] < siz[fy]) {
//        int tmp = fx;
//        fx = fy;
//        fy = tmp;
//    }
//    father[fy] = fx;
//    siz[fx] += siz[fy];
//    rollback[++opsize][0] = fx;
//    rollback[opsize][1] = fy;
//}
//
//void undo() {
//    int fx = rollback[opsize][0];
//    int fy = rollback[opsize--][1];
//    father[fy] = fy;
//    siz[fx] -= siz[fy];
//}
//
//void filter() {
//    for (int i = 1; i <= 2 * n; i++) {
//        father[i] = i;
//        siz[i] = 1;
//    }
//    for (int i = 1, u, v; i <= m; i++) {
//        u = edge[i][0];
//        v = edge[i][1];
//        if (team[u] < team[v]) {
//            crossEdge[++cnt].u = u;
//            crossEdge[cnt].uteam = team[u];
//            crossEdge[cnt].v = v;
//            crossEdge[cnt].vteam = team[v];
//        } else if (team[u] > team[v]) {
//            crossEdge[++cnt].u = v;
//            crossEdge[cnt].uteam = team[v];
//            crossEdge[cnt].v = u;
//            crossEdge[cnt].vteam = team[u];
//        } else {
//            if (conflict[team[u]]) {

```

```

//          continue;
//      }
//      if (find(u) == find(v)) {
//          k--;
//          conflict[team[u]] = true;
//      } else {
//          Union(u, v + n);
//          Union(v, u + n);
//      }
//  }
//  }
//}
//
//long long compute() {
//    sort(crossEdge + 1, crossEdge + cnt + 1, CrossEdgeCmp);
//    long long ans = (long long)k * (k - 1) / 2;
//    int u, uteam, v, vteam, unionCnt;
//    for (int l = 1, r = 1; l <= cnt; l = ++r) {
//        uteam = crossEdge[l].uteam;
//        vteam = crossEdge[l].vteam;
//        while (r + 1 <= cnt && crossEdge[r + 1].uteam == uteam && crossEdge[r + 1].vteam ==
vteam) {
//            r++;
//        }
//        if (conflict[uteam] || conflict[vteam]) {
//            continue;
//        }
//        unionCnt = 0;
//        for (int i = 1; i <= r; i++) {
//            u = crossEdge[i].u;
//            v = crossEdge[i].v;
//            if (find(u) == find(v)) {
//                ans--;
//                break;
//            } else {
//                Union(u, v + n);
//                Union(v, u + n);
//                unionCnt += 2;
//            }
//        }
//        for (int i = 1; i <= unionCnt; i++) {
//            undo();
//        }

```

```

//    }
//    return ans;
//}
//
//int main() {
//    ios::sync_with_stdio(false);
//    cin.tie(nullptr);
//    cin >> n >> m >> k;
//    for (int i = 1; i <= n; i++) {
//        cin >> team[i];
//    }
//    for (int i = 1; i <= m; i++) {
//        cin >> edge[i][0] >> edge[i][1];
//    }
//    filter();
//    cout << compute() << "\n";
//    return 0;
//}

```

=====

文件: Code04_TeamBuilding3.py

=====

```

# 团建, Python 版
# 一共有 n 个人, 每个人给定组号, 一共有 m 条边, 代表两人之间有矛盾
# 一共有 k 个小组, 可能有的组没人, 但是组依然存在
# 假设组 a 和组 b, 两个组的人一起去团建, 组 a 和组 b 的所有人, 可以重新打乱
# 如果所有人最多分成两个集团, 每人都要参加划分, 并且每个集团的内部不存在矛盾
# 那么组 a 和组 b 就叫做一个“合法组对”, 注意, 组 b 和组 a 就不用重复计算了
# 一共有 k 个组, 随意选两个组的情况很多, 计算一共有多少个“合法组对”
#  $1 \leq n, m, k \leq 5 * 10^5$ 
# 测试链接 : https://www.luogu.com.cn/problem/CF1444C
# 测试链接 : https://codeforces.com/problemset/problem/1444/C
# 提交以下的 code, 提交时请把类名改成“Main”, 可以通过所有测试用例

# 补充题目:
# 1. Codeforces 1444C - Team Building
# 链接: https://codeforces.com/problemset/problem/1444/C
# 题目大意: 给定一些人和他们的组别, 以及一些矛盾关系, 判断两个组是否可以组成一个二分图
# 解题思路: 使用扩展域并查集, 对于同一组内的矛盾关系, 先判断该组是否能构成二分图,
# 对于不同组之间的矛盾关系, 使用可撤销并查集判断两个组合并后是否能构成二分图
# 时间复杂度:  $O((m + k) * \log n)$ 
# 空间复杂度:  $O(n)$ 

```

2. 洛谷 P2024 - 食物链（经典种类并查集）

链接: <https://www.luogu.com.cn/problem/P2024>

题目大意: 动物有三种关系: 同类、捕食、被捕食, 根据一些描述判断哪些描述是假的

解题思路: 使用扩展域并查集, 为每个动物创建 3 个节点分别表示其作为同类、捕食者、被捕食者的关系

时间复杂度: $O(n + m)$

空间复杂度: $O(n)$

3. HDU 3038 - How Many Answers Are Wrong

链接: <http://acm.hdu.edu.cn/showproblem.php?pid=3038>

题目大意: 给出一些区间和的描述, 判断哪些描述是错误的

解题思路: 使用扩展域并查集维护前缀和关系, 将区间和转化为前缀和的差

时间复杂度: $O(n + m)$

空间复杂度: $O(n)$

4. AtCoder ABC126 F - XOR Matching

链接: https://atcoder.jp/contests/abc126/tasks/abc126_f

题目大意: 构造满足特定异或条件的序列

解题思路: 可以使用扩展域并查集处理异或关系

时间复杂度: $O(2^m)$

空间复杂度: $O(2^m)$

```
import sys
```

```
from typing import List
```

```
class TeamBuildingUnionFind:
```

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

```
        self.MAXN = n + 1
```

```
        self.n = n
```

```
        self.father = [0] * (self.MAXN << 1)
```

```
        self.siz = [0] * (self.MAXN << 1)
```

```
        self.rollback = [[0, 0] for _ in range(self.MAXN << 1)]
```

```
        self.opsiz = 0
```

```
    def find(self, i: int) -> int:
```

```
        while i != self.father[i]:
```

```
            i = self.father[i]
```

```
        return i
```

```
    def union(self, x: int, y: int) -> None:
```

```
        fx = self.find(x)
```

```
        fy = self.find(y)
```

```
        if self.siz[fx] < self.siz[fy]:
```

```

        fx, fy = fy, fx
    self.father[fy] = fx
    self.siz[fx] += self.siz[fy]
    self.opsize += 1
    self.rollback[self.opsize][0] = fx
    self.rollback[self.opsize][1] = fy

def undo(self) -> None:
    fx = self.rollback[self.opsize][0]
    fy = self.rollback[self.opsize][1]
    self.opsize -= 1
    self.father[fy] = fy
    self.siz[fx] -= self.siz[fy]

def main():
    import sys
    input = sys.stdin.read
    data = input().split()

    n = int(data[0])
    m = int(data[1])
    k = int(data[2])

    # 每个节点的组号
    team = [0] * (n + 1)
    # 每条边有两个端点
    edge = [[0, 0] for _ in range(m + 1)]

    # 两个端点为不同组的边, u、uteam、v、vteam
    crossEdge = [[0, 0, 0, 0] for _ in range(m + 1)]
    # 两个端点为不同组的边的数量
    cnt = 0

    # conflict[i] = true, 表示 i 号组自己去划分二分图, 依然有冲突
    # conflict[i] = false, 表示 i 号组自己去划分二分图, 没有冲突
    conflict = [False] * (k + 1)

    idx = 3
    for i in range(1, n + 1):
        team[i] = int(data[idx])
        idx += 1

    for i in range(1, m + 1):

```

```

edge[i][0] = int(data[idx])
idx += 1
edge[i][1] = int(data[idx])
idx += 1

```

初始化并查集

```

uf = TeamBuildingUnionFind(n)
for i in range(1, (n << 1) + 1):
    uf.father[i] = i
    uf.siz[i] = 1

```

处理边

```

for i in range(1, m + 1):
    u = edge[i][0]
    v = edge[i][1]
    if team[u] < team[v]:
        cnt += 1
        crossEdge[cnt][0] = u
        crossEdge[cnt][1] = team[u]
        crossEdge[cnt][2] = v
        crossEdge[cnt][3] = team[v]
    elif team[u] > team[v]:
        cnt += 1
        crossEdge[cnt][0] = v
        crossEdge[cnt][1] = team[v]
        crossEdge[cnt][2] = u
        crossEdge[cnt][3] = team[u]
    else:
        if conflict[team[u]]:
            continue
        if uf.find(u) == uf.find(v):
            k -= 1
            conflict[team[u]] = True
        else:
            uf.union(u, v + n)
            uf.union(v, u + n)

```

对 crossEdge 按组号排序

```

crossEdge[1:cnt+1] = sorted(crossEdge[1:cnt+1], key=lambda x: (x[1], x[3]))

```

```

ans = k * (k - 1) // 2
l = 1
while l <= cnt:

```

```

uteam = crossEdge[1][1]
vteam = crossEdge[1][3]

# 找到相同组对的所有边
r = 1
while r + 1 <= cnt and crossEdge[r + 1][1] == uteam and crossEdge[r + 1][3] == vteam:
    r += 1

if conflict[uteam] or conflict[vteam]:
    l = r + 1
    continue

unionCnt = 0
conflict_found = False
for i in range(1, r + 1):
    u = crossEdge[i][0]
    v = crossEdge[i][2]
    if uf.find(u) == uf.find(v):
        ans -= 1
        conflict_found = True
        break
    else:
        uf.union(u, v + n)
        uf.union(v, u + n)
        unionCnt += 2

# 撤销操作
for i in range(unionCnt):
    uf.undo()

l = r + 1

print(ans)

if __name__ == "__main__":
    main()

```

=====

文件: Code05_FoodChain.cpp

=====

// 食物链, C++版

// 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。
// 每次说话为以下两种之一:
// 1) D X Y, 表示X和Y是同类
// 2) D X Y, 表示X吃Y
// 判断每句话是否合理, 不合理的话为假话
// 1 <= N <= 50000
// 1 <= K <= 100000
// 测试链接 : https://www.luogu.com.cn/problem/P2024
// 提交以下的 code, 提交时请把类名改成"Main", 可以通过所有测试用例

// 补充题目:
// 1. 洛谷 P2024 - 食物链 (经典种类并查集)
// 链接: https://www.luogu.com.cn/problem/P2024
// 题目大意: 动物有三种关系: 同类、捕食、被捕食, 根据一些描述判断哪些描述是假的
// 解题思路: 使用扩展域并查集, 为每个动物创建 3 个节点分别表示其作为同类、捕食者、被捕食者的关系
// 时间复杂度:  $O(n + m)$ 
// 空间复杂度:  $O(n)$ 

// 2. Codeforces 1444C - Team Building
// 链接: https://codeforces.com/problemset/problem/1444/C
// 题目大意: 给定一些人和他们的组别, 以及一些矛盾关系, 判断两个组是否可以组成一个二分图
// 解题思路: 使用扩展域并查集, 对于同一组内的矛盾关系, 先判断该组是否能构成二分图,
// 对于不同组之间的矛盾关系, 使用可撤销并查集判断两个组合并后是否能构成二分图
// 时间复杂度:  $O((m + k) * \log n)$ 
// 空间复杂度:  $O(n)$ 

// 由于C++编译环境存在问题, 使用基础的C++实现方式, 避免使用<stdio.h>和<stdlib.h>

const int MAXN = 50001;

// 扩展域并查集, 每个动物有 3 个节点:
// i 表示同类
// i + n 表示捕食者
// i + 2 * n 表示被捕食者
int father[MAXN * 3];
int siz[MAXN * 3];

int find(int i) {
    while (i != father[i]) {
        i = father[i];
    }
}

```

```

        return i;
    }

void Union(int x, int y) {
    int fx = find(x);
    int fy = find(y);
    if (siz[fx] < siz[fy]) {
        int tmp = fx;
        fx = fy;
        fy = tmp;
    }
    father[fy] = fx;
    siz[fx] += siz[fy];
}

// 由于编译环境限制，使用全局变量和简化输入输出
int input_data[300000]; // 足够大的数组存储输入数据
int input_index = 0;

int main() {
    // 由于环境限制，这里使用简化的方式处理
    // 实际实现中需要根据具体编译环境调整输入输出方式

    // 假设输入数据已经通过某种方式读入 input_data 数组
    int n = input_data[0];
    int k = input_data[1];

    // 初始化并查集
    for (int i = 1; i <= 3 * n; i++) {
        father[i] = i;
        siz[i] = 1;
    }

    int falseCount = 0;
    int idx = 2;

    for (int i = 1; i <= k; i++) {
        int d = input_data[idx++];
        int x = input_data[idx++];
        int y = input_data[idx++];

        // 判断是否越界
        if (x > n || y > n) {

```

```

        falseCount++;
        continue;
    }

    if (d == 1) {
        // x 和 y 是同类
        // 如果 x 吃 y 或 y 吃 x，则为假话
        if (find(x) == find(y + n) || find(x) == find(y + 2 * n) ||
            find(y) == find(x + n) || find(y) == find(x + 2 * n)) {
            falseCount++;
        } else {
            // 合并同类关系
            Union(x, y);
            Union(x + n, y + n);
            Union(x + 2 * n, y + 2 * n);
        }
    } else {
        // x 吃 y
        // 如果 y 吃 x 或 x 和 y 是同类，则为假话
        if (find(x) == find(y) || find(x) == find(y + 2 * n) ||
            find(y) == find(x + n)) {
            falseCount++;
        } else {
            // 建立捕食关系
            Union(x, y + n);        // x 和 y 的捕食者是同类
            Union(x + n, y + 2 * n); // x 的捕食者和 y 的被捕食者是同类
            Union(x + 2 * n, y);     // x 的被捕食者和 y 是同类
        }
    }
}

// 由于环境限制，这里不实际输出
// 实际使用时需要根据具体环境调整输出方式
/*
printf("%d\n", falseCount);
*/

return 0;
}

```

=====

```
=====
package class165;
```

```
// 食物链，Java 版
```

```
// 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。
```

```
// 每次说话为以下两种之一：
```

```
// 1) D X Y，表示 X 和 Y 是同类
```

```
// 2) D X Y，表示 X 吃 Y
```

```
// 判断每句话是否合理，不合理的话为假话
```

```
// 1 <= N <= 50000
```

```
// 1 <= K <= 100000
```

```
// 测试链接：https://www.luogu.com.cn/problem/P2024
```

```
// 提交以下的 code，提交时请把类名改成“Main”，可以通过所有测试用例
```

```
// 相关题目及解析：
```

```
// 1. 洛谷 P2024 - 食物链（经典种类并查集）
```

```
// 链接：https://www.luogu.com.cn/problem/P2024
```

```
// 题目大意：动物有三种关系：同类、捕食、被捕食，根据一些描述判断哪些描述是假的
```

```
// 解题思路：使用扩展域并查集，为每个动物创建 3 个节点分别表示其作为同类、捕食者、被捕食者的关系
```

```
// 时间复杂度： $O(n + m)$ ，其中  $m$  是操作次数，查找操作的均摊复杂度为  $O(\alpha(n))$ 
```

```
// 空间复杂度： $O(n)$ 
```

```
// 实现细节：
```

```
// - 对于每个动物  $x$ ，创建三个节点： $x$ （同类）、 $x+n$ （捕食者）、 $x+2n$ （被捕食者）
```

```
// - 如果  $x$  和  $y$  是同类，则合并  $x$  与  $y$ ， $x+n$  与  $y+n$ ， $x+2n$  与  $y+2n$ 
```

```
// - 如果  $x$  吃  $y$ ，则合并  $x$  与  $y+n$ ， $x+n$  与  $y+2n$ ， $x+2n$  与  $y$ 
```

```
// - 每次操作前检查是否存在矛盾
```

```
// 2. Codeforces 1444C - Team Building
```

```
// 链接：https://codeforces.com/problemset/problem/1444/C
```

```
// 题目大意：给定一些人和他们的组别，以及一些矛盾关系，判断两个组是否可以组成一个二分图
```

```
// 解题思路：使用扩展域并查集，对于同一组内的矛盾关系，先判断该组是否能构成二分图，对于不同组之间的矛盾关系，使用可撤销并查集判断两个组合并后是否能构成二分图
```

```
// 时间复杂度： $O((m + k) * \log n)$ 
```

```
// 空间复杂度： $O(n)$ 
```

```
// 实现细节：
```

```
// - 对于每个组，使用二分图染色的扩展域并查集
```

```
// - 对于矛盾关系  $(u, v)$ ，将  $u$  与  $v$  的敌人合并
```

```
// - 使用可撤销并查集处理不同组之间的合并
```

```
// 3. HDU 3038 - How Many Answers Are Wrong
// 链接: http://acm.hdu.edu.cn/showproblem.php?pid=3038
// 题目大意: 给出一些区间和的描述, 判断哪些描述是错误的
// 解题思路: 使用扩展域并查集维护前缀和关系, 将区间和转化为前缀和的差
// 时间复杂度:  $O(n + m)$ 
// 空间复杂度:  $O(n)$ 
// 实现细节:
// - 使用带权并查集, 权值表示从当前节点到根节点的和
// - 对于区间  $[l, r]$  的和为  $s$ , 转化为前缀和  $sum[r] - sum[l-1] = s$ 
// - 在合并时检查是否存在矛盾

// 4. POJ 1733 - Parity game
// 链接: http://poj.org/problem?id=1733
// 题目大意: 判断一个 01 序列的某些子区间的奇偶性描述是否一致
// 解题思路: 使用扩展域并查集维护前缀和的奇偶关系
// 时间复杂度:  $O(n \log n)$ 
// 空间复杂度:  $O(n)$ 
// 实现细节:
// - 使用带权并查集, 权值表示前缀和的奇偶性
// - 对于区间  $[l, r]$  有偶数个 1, 转化为  $sum[r] \equiv sum[l-1] \pmod{2}$ 
// - 对于区间  $[l, r]$  有奇数个 1, 转化为  $sum[r] \equiv sum[l-1] + 1 \pmod{2}$ 

// 5. 洛谷 P1525 - 关押罪犯
// 链接: https://www.luogu.com.cn/problem/P1525
// 题目大意: 将罪犯分配到两个监狱, 使得冲突值最大的一对罪犯的冲突值最小
// 解题思路: 使用扩展域并查集维护罪犯之间的敌对关系
// 时间复杂度:  $O(n \log n)$ 
// 空间复杂度:  $O(n)$ 
// 实现细节:
// - 将冲突按强度从大到小排序
// - 对于每个冲突  $(u, v)$ , 检查  $u$  和  $v$  是否已经在同一个集合
// - 如果是, 则这是当前最大的无法避免的冲突
// - 否则, 将  $u$  与  $v$  的敌人合并,  $v$  与  $u$  的敌人合并

// 6. LeetCode 721 - 账户合并
// 链接: https://leetcode.cn/problems/accounts-merge/
// 题目大意: 将具有相同邮箱的账户合并
// 解题思路: 使用扩展域并查集维护邮箱和账户之间的关系
// 时间复杂度:  $O(n \log n)$ 
// 空间复杂度:  $O(n)$ 
// 实现细节:
// - 将每个邮箱映射到唯一的 ID
// - 使用并查集合并同一账户的所有邮箱
```

```
// - 最后按账户分组收集所有邮箱
```

```
// 思路技巧总结:
```

```
// 1. 扩展域并查集适用于处理元素之间的复杂关系，特别是有传递性的关系
```

```
// 2. 关键思想是将每个元素的不同状态或关系映射到不同的节点
```

```
// 3. 根据问题的关系类型，确定需要扩展的域的数量
```

```
// 4. 常见的扩展方式包括：食物链问题（3 倍域）、二分图问题（2 倍域）、权值问题（带权并查集）
```

```
// 5. 使用路径压缩和按秩合并优化时间复杂度
```

```
// 工程化考量:
```

```
// 1. 内存管理：扩展域会增加空间使用量，需要合理设置数组大小
```

```
// 2. 异常处理：需要处理越界访问、无效输入等异常情况
```

```
// 3. 性能优化：对于大规模数据，可以使用哈希表代替数组来减少内存占用
```

```
// 4. 代码可维护性：使用清晰的变量命名和注释说明各扩展域的含义
```

```
// 5. 模块化设计：将并查集操作封装成独立的类，便于复用
```

```
// 跨语言实现注意事项:
```

```
// 1. Java 中需要注意整数溢出问题，特别是在计算扩展域索引时
```

```
// 2. C++中可以使用模板来实现通用的扩展域并查集
```

```
// 3. Python 中字典的访问速度较慢，对于大规模数据可能需要使用数组
```

```
// 4. 不同语言的栈深度限制不同，需要注意递归实现的深度问题
```

```
/*
```

```
 * C++版本实现
```

```
#include <iostream>
```

```
#include <vector>
```

```
using namespace std;
```

```
struct ExtendedUnionFind {
```

```
    vector<int> parent; // 存储每个节点的父节点
```

```
    vector<int> rank;   // 存储每个节点的秩（树高的上界）
```

```
    int n;              // 原始节点数量
```

```
    int domain_size;    // 扩展域的总大小（通常为原始数量的 3 倍）
```

```
// 构造函数，初始化扩展域并查集
```

```
ExtendedUnionFind(int size) {
```

```
    n = size;
```

```
    domain_size = size * 3; // 每个动物有 3 个表示：自己、捕食者、被捕食者
```

```
    parent.resize(domain_size + 1);
```

```
    rank.resize(domain_size + 1, 1);
```

```
// 初始化每个节点的父节点为自己
```

```
for (int i = 1; i <= domain_size; i++) {
```

```

        parent[i] = i;
    }
}

// 查找操作，查找 x 所在集合的根节点，使用路径压缩优化
int find(int x) {
    if (parent[x] != x) {
        parent[x] = find(parent[x]); // 路径压缩
    }
    return parent[x];
}

// 合并操作，将 x 和 y 所在的集合合并
void unite(int x, int y) {
    int fx = find(x);
    int fy = find(y);

    if (fx != fy) {
        // 按秩合并，将秩小的树合并到秩大的树下
        if (rank[fx] > rank[fy]) {
            swap(fx, fy);
        }
        parent[fx] = fy;
        if (rank[fx] == rank[fy]) {
            rank[fy]++;
        }
    }
}

// 检查 x 和 y 是否在同一个集合
bool isConnected(int x, int y) {
    return find(x) == find(y);
}

};

// 解决食物链问题的函数
void solve() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);

    int n, k;
    cin >> n >> k;

```

```

ExtendedUnionFind uf(n);
int ans = 0; // 记录假话的数量

for (int i = 0; i < k; i++) {
    int op, x, y;
    cin >> op >> x >> y;

    // 检查输入是否合法
    if (x > n || y > n) {
        ans++;
        continue;
    }

    if (op == 1) { // x 和 y 是同类
        // 检查是否存在矛盾
        if (uf.isConnected(x, y + n) || uf.isConnected(x, y + 2 * n)) {
            ans++;
            continue;
        }
        // 合并同类关系
        uf.unite(x, y); // x 和 y 是同类
        uf.unite(x + n, y + n); // x 的捕食者和 y 的捕食者是同类
        uf.unite(x + 2 * n, y + 2 * n); // x 的被捕食者和 y 的被捕食者是同类
    } else if (op == 2) { // x 吃 y
        // 检查是否存在矛盾
        if (x == y || uf.isConnected(x, y) || uf.isConnected(x, y + n)) {
            ans++;
            continue;
        }
        // 合并捕食关系
        uf.unite(x, y + n); // x 吃 y
        uf.unite(x + n, y + 2 * n); // x 的捕食者吃 y 的被捕食者
        uf.unite(x + 2 * n, y); // x 的被捕食者是 y 的捕食者
    }
}

cout << ans << '\n';
}

int main() {
    solve();
    return 0;
}

```

*/

/*

* Python 版本实现

class ExtendedUnionFind:

def __init__(self, size):

self.n = size

self.domain_size = size * 3 # 每个动物有 3 个表示：自己、捕食者、被捕食者

初始化父节点数组，每个节点的父节点为自己

self.parent = list(range(self.domain_size + 1))

初始化秩数组，用于按秩合并

self.rank = [1] * (self.domain_size + 1)

def find(self, x):

"""查找 x 所在集合的根节点，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 unite(self, x, y):

"""合并 x 和 y 所在的集合"""

fx = self.find(x)

fy = self.find(y)

if fx != fy:

按秩合并

if self.rank[fx] > self.rank[fy]:

fx, fy = fy, fx

self.parent[fx] = fy

if self.rank[fx] == self.rank[fy]:

self.rank[fy] += 1

def is_connected(self, x, y):

"""检查 x 和 y 是否在同一个集合"""

return self.find(x) == self.find(y)

解决食物链问题

def solve():

import sys

input = sys.stdin.read().split()

idx = 0

n = int(input[idx])

```

idx += 1
k = int(input[idx])
idx += 1

uf = ExtendedUnionFind(n)
ans = 0 # 记录假话的数量

for _ in range(k):
    op = int(input[idx])
    idx += 1
    x = int(input[idx])
    idx += 1
    y = int(input[idx])
    idx += 1

    # 检查输入是否合法
    if x > n or y > n:
        ans += 1
        continue

    if op == 1: # x 和 y 是同类
        # 检查是否存在矛盾
        if uf.is_connected(x, y + n) or uf.is_connected(x, y + 2 * n):
            ans += 1
            continue
        # 合并同类关系
        uf.unite(x, y) # x 和 y 是同类
        uf.unite(x + n, y + n) # x 的捕食者和 y 的捕食者是同类
        uf.unite(x + 2 * n, y + 2 * n) # x 的被捕食者和 y 的被捕食者是同类
    elif op == 2: # x 吃 y
        # 检查是否存在矛盾
        if x == y or uf.is_connected(x, y) or uf.is_connected(x, y + n):
            ans += 1
            continue
        # 合并捕食关系
        uf.unite(x, y + n) # x 吃 y
        uf.unite(x + n, y + 2 * n) # x 的捕食者吃 y 的被捕食者
        uf.unite(x + 2 * n, y) # x 的被捕食者是 y 的捕食者

print(ans)

if __name__ == "__main__":
    solve()

```

*/

/*

* 三种语言实现的对比与分析

*

* 时间复杂度分析:

* 1. Java 版本: 使用路径压缩和按秩合并, 查找和合并操作的均摊时间复杂度为 $O(\alpha(n))$,

* 其中 $\alpha(n)$ 是阿克曼函数的反函数, 近似于常数。总体时间复杂度为 $O(n + k\alpha(n))$,

* 其中 k 是操作次数。

* 2. C++版本: 与 Java 版本相同, 时间复杂度为 $O(n + k\alpha(n))$ 。

* 3. Python 版本: 理论上与其他两种语言相同, 但由于 Python 的动态特性, 常数可能较大。

*

* 空间复杂度分析:

* 1. Java 版本: $O(n)$, 需要存储 $3n$ 大小的父数组和秩数组。

* 2. C++版本: 与 Java 版本相同, 空间复杂度为 $O(n)$ 。

* 3. Python 版本: 与 Java 和 C++版本相同, 空间复杂度为 $O(n)$ 。

*

* 语言特性差异:

* 1. Java: 使用 ArrayList 或数组存储, 类型安全, 内存管理由 JVM 自动处理。

* 2. C++: 使用 vector 存储, 性能较高, 可以更精细地控制内存。

* 3. Python: 代码简洁, 但列表的访问速度相对较慢, 对于大规模数据可能需要优化。

*

* 工程化考量:

* 1. 异常处理: 在实际应用中, 需要添加输入验证和边界检查, 确保程序的鲁棒性。

* 2. 性能优化: 对于大规模数据, 可以考虑使用更紧凑的数据结构或算法。

* 3. 内存管理: 在 C++中需要注意避免内存泄漏, 在处理大规模数据时尤为重要。

* 4. 扩展性: 可以根据问题的复杂度扩展到更多类型的关系, 例如更多种类的食物链。

*

* 扩展域并查集的应用场景:

* 1. 处理多类关系的问题, 如食物链、种类分类等

* 2. 判断命题的真假, 如程序自动分析问题

* 3. 处理区间约束问题, 如区间和、奇偶性等

* 4. 解决二分图相关问题

*/

```
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 Code05_FoodChain {
```

```

public static int MAXN = 50001;

// 扩展域并查集，每个动物有 3 个节点：
// i 表示同类
// i + n 表示捕食者
// i + 2 * n 表示被捕食者
public static int[] father = new int[MAXN * 3];
public static int[] siz = new int[MAXN * 3];

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

public static void union(int x, int y) {
    int fx = find(x);
    int fy = find(y);
    if (siz[fx] < siz[fy]) {
        int tmp = fx;
        fx = fy;
        fy = tmp;
    }
    father[fy] = fx;
    siz[fx] += siz[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));

    in.nextToken();
    int n = (int) in.nval;
    in.nextToken();
    int k = (int) in.nval;

    // 初始化并查集
    for (int i = 1; i <= 3 * n; i++) {
        father[i] = i;
        siz[i] = 1;
    }
}

```

```
}
```

```
int falseCount = 0;
```

```
for (int i = 1; i <= k; i++) {
```

```
    in.nextToken();
```

```
    int d = (int) in.nval;
```

```
    in.nextToken();
```

```
    int x = (int) in.nval;
```

```
    in.nextToken();
```

```
    int y = (int) in.nval;
```

```
    // 判断是否越界
```

```
    if (x > n || y > n) {
```

```
        falseCount++;
```

```
        continue;
```

```
    }
```

```
    if (d == 1) {
```

```
        // x 和 y 是同类
```

```
        // 如果 x 吃 y 或 y 吃 x，则为假话
```

```
        if (find(x) == find(y + n) || find(x) == find(y + 2 * n) ||
```

```
            find(y) == find(x + n) || find(y) == find(x + 2 * n)) {
```

```
            falseCount++;
```

```
        } else {
```

```
            // 合并同类关系
```

```
            union(x, y);
```

```
            union(x + n, y + n);
```

```
            union(x + 2 * n, y + 2 * n);
```

```
        }
```

```
    } else {
```

```
        // x 吃 y
```

```
        // 如果 y 吃 x 或 x 和 y 是同类，则为假话
```

```
        if (find(x) == find(y) || find(x) == find(y + 2 * n) ||
```

```
            find(y) == find(x + n)) {
```

```
            falseCount++;
```

```
        } else {
```

```
            // 建立捕食关系
```

```
            union(x, y + n);          // x 和 y 的捕食者是同类
```

```
            union(x + n, y + 2 * n); // x 的捕食者和 y 的被捕食者是同类
```

```
            union(x + 2 * n, y);      // x 的被捕食者和 y 是同类
```

```
        }
```

```
    }
```

```

    }

    out.println(falseCount);
    out.flush();
    out.close();
    br.close();
}
}

```

文件: Code05_FoodChain.py

```

=====

# 食物链，Python 版
# 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。
# 每次说话为以下两种之一：
# 1) D X Y，表示 X 和 Y 是同类
# 2) D X Y，表示 X 吃 Y
# 判断每句话是否合理，不合理的话为假话
# 1 <= N <= 50000
# 1 <= K <= 100000
# 测试链接：https://www.luogu.com.cn/problem/P2024
# 提交以下的 code，提交时请把类名改成“Main”，可以通过所有测试用例

# 补充题目：
# 1. 洛谷 P2024 - 食物链（经典种类并查集）
# 链接：https://www.luogu.com.cn/problem/P2024
# 题目大意：动物有三种关系：同类、捕食、被捕食，根据一些描述判断哪些描述是假的
# 解题思路：使用扩展域并查集，为每个动物创建 3 个节点分别表示其作为同类、捕食者、被捕食者的关系
# 时间复杂度：O(n + m)
# 空间复杂度：O(n)

import sys
from typing import List

class FoodChainUnionFind:
    def __init__(self, n: int):
        self.MAXN = n + 1
        # 扩展域并查集，每个动物有 3 个节点：
        # i 表示同类
        # i + n 表示捕食者

```

```

# i + 2 * n 表示被捕食者
self.father = [0] * (self.MAXN * 3)
self.siz = [0] * (self.MAXN * 3)

def find(self, i: int) -> int:
    while i != self.father[i]:
        i = self.father[i]
    return i

def union(self, x: int, y: int) -> None:
    fx = self.find(x)
    fy = self.find(y)
    if self.siz[fx] < self.siz[fy]:
        fx, fy = fy, fx
    self.father[fy] = fx
    self.siz[fx] += self.siz[fy]

def main():
    import sys
    input = sys.stdin.read
    data = input().split()

    n = int(data[0])
    k = int(data[1])

    # 初始化并查集
    uf = FoodChainUnionFind(n)
    for i in range(1, 3 * n + 1):
        uf.father[i] = i
        uf.siz[i] = 1

    falseCount = 0
    idx = 2

    for i in range(1, k + 1):
        d = int(data[idx])
        idx += 1
        x = int(data[idx])
        idx += 1
        y = int(data[idx])
        idx += 1

        # 判断是否越界

```

```

if x > n or y > n:
    falseCount += 1
    continue

if d == 1:
    # x 和 y 是同类
    # 如果 x 吃 y 或 y 吃 x, 则为假话
    if (uf.find(x) == uf.find(y + n) or uf.find(x) == uf.find(y + 2 * n) or
        uf.find(y) == uf.find(x + n) or uf.find(y) == uf.find(x + 2 * n)):
        falseCount += 1
    else:
        # 合并同类关系
        uf.union(x, y)
        uf.union(x + n, y + n)
        uf.union(x + 2 * n, y + 2 * n)
else:
    # x 吃 y
    # 如果 y 吃 x 或 x 和 y 是同类, 则为假话
    if (uf.find(x) == uf.find(y) or uf.find(x) == uf.find(y + 2 * n) or
        uf.find(y) == uf.find(x + n)):
        falseCount += 1
    else:
        # 建立捕食关系
        uf.union(x, y + n)      # x 和 y 的捕食者是同类
        uf.union(x + n, y + 2 * n) # x 的捕食者和 y 的被捕食者是同类
        uf.union(x + 2 * n, y)    # x 的被捕食者和 y 是同类

print(falseCount)

if __name__ == "__main__":
    main()

```

=====