==========
文件夹: class127_SegmentTreeDivideAndConquerAlgorithms
==========

[Markdown 文件]
==========
文件: ADDITIONAL_PROBLEMS.md
==========
# 线段树分治补充题目集

## 一、LeetCode 题目

### 1. 动态图连通性 (Dynamic Graph Connectivity)
- **题目链接**: https://leetcode.com/problems/dynamic-graph-connectivity/
- **难度**: Hard
- **标签**: Union Find, Segment Tree, Divide and Conquer
- **题目描述**: 支持动态加边、删边操作，查询两点间连通性
- **解法**: 线段树分治 + 可撤销并查集
- **时间复杂度**: O((n + m) log m)
- **空间复杂度**: O(n + m)

### 2. 不良数对计数 (Count Number of Bad Pairs)
- **题目链接**: https://leetcode.com/problems/count-number-of-bad-pairs/
- **难度**: Medium
- **标签**: Segment Tree, Divide and Conquer, Math
- **题目描述**: 统计满足特定条件的数对数量
- **解法**: 线段树分治 + 数学变换
- **时间复杂度**: O(n log n)
- **空间复杂度**: O(n)

### 3. 带阈值的图连通性 (Graph Connectivity With Threshold)
- **题目链接**: https://leetcode.com/problems/graph-connectivity-with-threshold/
- **难度**: Hard
- **标签**: Union Find, Math, Segment Tree, Divide and Conquer
- **题目描述**: 给定 n 个城市，编号 1 到 n，当两个城市的最大公约数大于 threshold 时它们直接相连，查询任意两个城市是否连通
- **解法**: 线段树分治 + 可撤销并查集
- **时间复杂度**: O(n log n + q log n)
- **空间复杂度**: O(n)

## 二、Codeforces 题目

### 1. 二分图检测 (Bipartite Checking) - 813F

- **题目链接**: https://codeforces.com/contest/813/problem/F
- **难度**: 2400
- **标签**: Segment Tree, Divide and Conquer, Union Find, Bipartite Graph
- **题目描述**: 动态维护图的二分性
- **解法**: 线段树分治 + 扩展域并查集
- **时间复杂度**: O((n + m) log m)
- **空间复杂度**: O(n + m)

### 2. 唯一出现次数 (Unique Occurrences) - 1681F
- **题目链接**: https://codeforces.com/contest/1681/problem/F
- **难度**: 2600
- **标签**: Segment Tree, Divide and Conquer, Union Find, Tree
- **题目描述**: 统计树上路径中唯一出现的颜色数量
- **解法**: 线段树分治 + 可撤销并查集
- **时间复杂度**: O((n + m) log m)
- **空间复杂度**: O(n + m)

### 3. 边着色 (Painting Edges) - 576E
- **题目链接**: https://codeforces.com/contest/576/problem/E
- **难度**: 3300
- **标签**: Segment Tree, Divide and Conquer, Union Find, Graph
- **题目描述**: 给边着色使得每种颜色构成的子图都是二分图
- **解法**: 线段树分治 + 多个扩展域并查集
- **时间复杂度**: O((n + m) log m)
- **空间复杂度**: O(n + m)

### 4. 博物馆劫案 (Museum Robbery) - 601E
- **题目链接**: https://codeforces.com/problemset/problem/601/E
- **难度**: 2800
- **标签**: Segment Tree, Divide and Conquer, Dynamic Programming
- **题目描述**: 维护商品集合，支持添加、删除商品，查询背包问题变形结果
- **解法**: 线段树分治 + 动态规划
- **时间复杂度**: O(qk log q + nk)
- **空间复杂度**: O(qk)

### 5. 线段上的加法 (Addition on Segments) - 981E
- **题目链接**: https://codeforces.com/problemset/problem/981/E
- **难度**: 2200
- **标签**: Segment Tree, Divide and Conquer, Bit Manipulation, Dynamic Programming
- **题目描述**: 给定数组初始全为0，支持区间加法操作，每种操作只能执行一次，查询能通过选择操作得到的所有可能最大值
- **解法**: 线段树分治 + 位运算优化DP
- **时间复杂度**: O(nq log q)

- **空间复杂度**: O(n)

### 6. 异或最短路 (Shortest Path Queries) - 938G
- **题目链接**: https://codeforces.com/problemset/problem/938/G
- **难度**: 2900
- **标签**: Segment Tree, Divide and Conquer, Linear Basis, Union Find
- **题目描述**: 维护图，支持加边、删边操作，查询两点间路径边权异或和的最小值
- **解法**: 线段树分治 + 带权并查集 + 线性基
- **时间复杂度**: O((n+q) log q log V)
- **空间复杂度**: O(n + q)

## 三、洛谷题目

### 1. 二分图 /【模板】线段树分治 - P5787
- **题目链接**: https://www.luogu.com.cn/problem/P5787
- **难度**: 省选/NOI-
- **标签**: 线段树分治, 扩展域并查集, 二分图
- **题目描述**: 维护动态图使其为二分图
- **解法**: 线段树分治 + 扩展域并查集
- **时间复杂度**: O((n + m) log m)
- **空间复杂度**: O(n + m)

### 2. 最小 mex 生成树 - P5631
- **题目链接**: https://www.luogu.com.cn/problem/P5631
- **难度**: 省选/NOI-
- **标签**: 线段树分治, 并查集, 生成树, 二分
- **题目描述**: 求生成树使得边权集合的 mex 最小
- **解法**: 线段树分治 + 可撤销并查集 + 二分答案
- **时间复杂度**: O((n + m) log m log n)
- **空间复杂度**: O(n + m)

### 3. 大融合 - P4219
- **题目链接**: https://www.luogu.com.cn/problem/P4219
- **难度**: 省选/NOI-
- **标签**: 线段树分治, 并查集, 图论
- **题目描述**: 支持加边和查询边负载，边负载定义为删去该边后两个连通块大小的乘积
- **解法**: 线段树分治 + 可撤销并查集
- **时间复杂度**: O((n + m) log m)
- **空间复杂度**: O(n + m)

### 4. 连通图 - P5227
- **题目链接**: https://www.luogu.com.cn/problem/P5227
- **难度**: 省选/NOI-

- **标签**: 线段树分治, 并查集, 图论
- **题目描述**: 给定初始连通图，每次删除一些边，查询是否仍连通
- **解法**: 线段树分治 + 可撤销并查集
- **时间复杂度**: O((n + m) log m)
- **空间复杂度**: O(n + m)

### 5. 八纵八横 - P3733
- **题目链接**: https://www.luogu.com.cn/problem/P3733
- **难度**: 省选/NOI-
- **标签**: 线段树分治, Linear Basis, Union Find
- **题目描述**: 维护图，支持加边、删边、修改边权操作，查询从 1 号点出发回到 1 号点路径边权异或和的最大值
- **解法**: 线段树分治 + 带权并查集 + 线性基
- **时间复杂度**: O((n+q) log q L)
- **空间复杂度**: O(nL + qL)

### 6. 火星商店 - P4585
- **题目链接**: https://www.luogu.com.cn/problem/P4585
- **难度**: 省选/NOI-
- **标签**: 线段树分治, Persistent Trie
- **题目描述**: 维护 n 个商店，每个商店有商品，支持添加商品、查询操作，查询要求在特定商店范围内和时间范围内找到异或最大值
- **解法**: 线段树分治 + 可持久化 Trie
- **时间复杂度**: O((n+q) log q log V)
- **空间复杂度**: O((n+q) log V)

## 四、AtCoder 题目

### 1. 细胞分裂 (Cell Division) - AGC010C
- **题目链接**: https://atcoder.jp/contests/agc010/tasks/agc010_c
- **难度**: 2300
- **标签**: Union Find, Divide and Conquer
- **题目描述**: 分割矩形并计算每次分割后的连通分量数
- **解法**: 线段树分治 + 可撤销并查集
- **时间复杂度**: O((n + m) log m)
- **空间复杂度**: O(n + m)

### 2. 最小异或对查询 (Minimum Xor Pair Query) - ABC308G
- **题目链接**: https://atcoder.jp/contests/abc308/tasks/abc308_g
- **难度**: 600
- **标签**: Trie, Bit Manipulation
- **题目描述**: 维护集合，支持添加数字、删除数字、查询操作，查询集合中任意两个数的异或最小值
- **解法**: 01Trie + 在线维护

- **时间复杂度**: O(q log V)
- **空间复杂度**: O(q log V)

## 五、其他平台题目

### 1. 动态连通性 (SPOJ DYNACON1)
- **题目链接**: https://www.spoj.com/problems/DYNACON1/
- **难度**: Hard
- **标签**: Segment Tree, Divide and Conquer, Union Find
- **题目描述**: 动态维护图的连通性，支持加边、删边和查询操作
- **解法**: 线段树分治 + 可撤销并查集
- **时间复杂度**: O((n + m) log m)
- **空间复杂度**: O(n + m)

### 2. 动态图的最小生成树 (SPOJ DYNALCA)
- **题目链接**: https://www.spoj.com/problems/DYNALCA/
- **难度**: Hard
- **标签**: Segment Tree, Divide and Conquer, Union Find
- **题目描述**: 动态维护图的最小生成树相关查询
- **解法**: 线段树分治 + 可撤销并查集 + Kruskal 算法
- **时间复杂度**: O((n + m) log m log n)
- **空间复杂度**: O(n + m)

### 3. 动态树 (HackerRank Dynamic Trees)
- **题目链接**: https://www.hackerrank.com/challenges/dynamic-trees
- **难度**: Advanced
- **标签**: Segment Tree, Divide and Conquer, Tree Data Structures
- **题目描述**: 动态维护树的结构和路径查询
- **解法**: 线段树分治 + 可撤销并查集
- **时间复杂度**: O((n + m) log m)
- **空间复杂度**: O(n + m)

## 六、练习建议

1. **从基础开始**：先掌握可撤销并查集的实现，理解如何支持回滚操作
2. **理解线段树分治的核心思想**：将时间轴划分为线段树结构，处理每个时间区间内的操作
3. **掌握扩展域并查集**：用于处理二分图检测等约束问题
4. **练习经典题目**：从简单的动态连通性问题开始，逐步挑战更复杂的题目
5. **注意实现细节**：特别是回滚操作的正确性和时间复杂度分析

==========

文件: COMPREHENSIVE_SEGMENT_TREE_PROBLEMS.md

==========
# 线段树分治算法 comprehensive 题目集

## 一、核心算法概念

线段树分治（Segment Tree Divide and Conquer）是一种离线算法技术，主要用于解决带有时间维度的图论问题和动态维护问题。它将操作序列按照时间轴建立线段树，然后通过 DFS 遍历线段树来处理各个时间区间内的操作。

### 核心思想
1. **离线处理**：将所有操作和查询离线，按照时间建立线段树
2. **区间操作**：将每个操作的影响时间段映射到线段树的节点上
3. **可撤销数据结构**：使用可撤销并查集等支持回滚操作的数据结构
4. **DFS 遍历**：通过 DFS 遍历线段树，处理每个节点的操作并及时回滚

### 关键技术点
1. **可撤销并查集 (Rollback DSU)**：支持合并操作的回滚
2. **扩展域并查集 (Extended Union Find)**：用于二分图检测等约束问题
3. **线性基 (Linear Basis)**：处理异或运算相关问题
4. **可持久化数据结构**：处理多维限制问题

## 二、LeetCode 题目

### 1. 动态图连通性 (Dynamic Graph Connectivity)
- **题目链接**: https://leetcode.com/problems/dynamic-graph-connectivity/
- **难度**: Hard
- **标签**: Union Find, Segment Tree, Divide and Conquer
- **题目描述**: 支持动态加边、删边操作，查询两点间连通性
- **解法**: 线段树分治 + 可撤销并查集
- **时间复杂度**: O((n + m) log m)
- **空间复杂度**: O(n + m)

### 2. 不良数对计数 (Count Number of Bad Pairs)
- **题目链接**: https://leetcode.com/problems/count-number-of-bad-pairs/
- **难度**: Medium
- **标签**: Segment Tree, Divide and Conquer, Math
- **题目描述**: 统计满足特定条件的数对数量
- **解法**: 线段树分治 + 数学变换
- **时间复杂度**: O(n log n)
- **空间复杂度**: O(n)

### 3. 带阈值的图连通性 (Graph Connectivity With Threshold)
- **题目链接**: https://leetcode.com/problems/graph-connectivity-with-threshold/

- **难度**: Hard
- **标签**: Union Find, Math, Segment Tree, Divide and Conquer
- **题目描述**: 给定 n 个城市，编号 1 到 n，当两个城市的最大公约数大于 threshold 时它们直接相连，查询任意两个城市是否连通
- **解法**: 线段树分治 + 可撤销并查集
- **时间复杂度**: O(n log n + q log n)
- **空间复杂度**: O(n)

## 三、Codeforces 题目

### 1. 二分图检测 (Bipartite Checking) - 813F
- **题目链接**: https://codeforces.com/contest/813/problem/F
- **难度**: 2400
- **标签**: Segment Tree, Divide and Conquer, Union Find, Bipartite Graph
- **题目描述**: 动态维护图的二分性
- **解法**: 线段树分治 + 扩展域并查集
- **时间复杂度**: O((n + m) log m)
- **空间复杂度**: O(n + m)

### 2. 唯一出现次数 (Unique Occurrences) - 1681F
- **题目链接**: https://codeforces.com/contest/1681/problem/F
- **难度**: 2600
- **标签**: Segment Tree, Divide and Conquer, Union Find, Tree
- **题目描述**: 统计树上路径中唯一出现的颜色数量
- **解法**: 线段树分治 + 可撤销并查集
- **时间复杂度**: O((n + m) log m)
- **空间复杂度**: O(n + m)

### 3. 边着色 (Painting Edges) - 576E
- **题目链接**: https://codeforces.com/contest/576/problem/E
- **难度**: 3300
- **标签**: Segment Tree, Divide and Conquer, Union Find, Graph
- **题目描述**: 给边着色使得每种颜色构成的子图都是二分图
- **解法**: 线段树分治 + 多个扩展域并查集
- **时间复杂度**: O((n + m) log m)
- **空间复杂度**: O(n + m)

### 4. 博物馆劫案 (Museum Robbery) - 601E
- **题目链接**: https://codeforces.com/problemset/problem/601/E
- **难度**: 2800
- **标签**: Segment Tree, Divide and Conquer, Dynamic Programming
- **题目描述**: 维护商品集合，支持添加、删除商品，查询背包问题变形结果
- **解法**: 线段树分治 + 动态规划

- **时间复杂度**: O(qk log q + nk)
- **空间复杂度**: O(qk)

### 5. 线段上的加法 (Addition on Segments) - 981E
- **题目链接**: https://codeforces.com/problemset/problem/981/E
- **难度**: 2200
- **标签**: Segment Tree, Divide and Conquer, Bit Manipulation, Dynamic Programming
- **题目描述**: 给定数组初始全为0，支持区间加法操作，每种操作只能执行一次，查询能通过选择操作得到的所有可能最大值
- **解法**: 线段树分治 + 位运算优化DP
- **时间复杂度**: O(nq log q)
- **空间复杂度**: O(n)

### 6. 异或最短路 (Shortest Path Queries) - 938G
- **题目链接**: https://codeforces.com/problemset/problem/938/G
- **难度**: 2900
- **标签**: Segment Tree, Divide and Conquer, Linear Basis, Union Find
- **题目描述**: 维护图，支持加边、删边操作，查询两点间路径边权异或和的最小值
- **解法**: 线段树分治 + 带权并查集 + 线性基
- **时间复杂度**: O((n+q) log q log V)
- **空间复杂度**: O(n + q)

## 四、洛谷题目

### 1. 二分图 /【模板】线段树分治 - P5787
- **题目链接**: https://www.luogu.com.cn/problem/P5787
- **难度**: 省选/NOI-
- **标签**: 线段树分治, 扩展域并查集, 二分图
- **题目描述**: 维护动态图使其为二分图
- **解法**: 线段树分治 + 扩展域并查集
- **时间复杂度**: O((n + m) log m)
- **空间复杂度**: O(n + m)

### 2. 最小mex生成树 - P5631
- **题目链接**: https://www.luogu.com.cn/problem/P5631
- **难度**: 省选/NOI-
- **标签**: 线段树分治, 并查集, 生成树, 二分
- **题目描述**: 求生成树使得边权集合的mex最小
- **解法**: 线段树分治 + 可撤销并查集 + 二分答案
- **时间复杂度**: O((n + m) log m log n)
- **空间复杂度**: O(n + m)

### 3. 大融合 - P4219

- **题目链接**: https://www.luogu.com.cn/problem/P4219
- **难度**: 省选/NOI-
- **标签**: 线段树分治, 并查集, 图论
- **题目描述**: 支持加边和查询边负载，边负载定义为删去该边后两个连通块大小的乘积
- **解法**: 线段树分治 + 可撤销并查集
- **时间复杂度**: O((n + m) log m)
- **空间复杂度**: O(n + m)

### 4. 连通图 - P5227
- **题目链接**: https://www.luogu.com.cn/problem/P5227
- **难度**: 省选/NOI-
- **标签**: 线段树分治, 并查集, 图论
- **题目描述**: 给定初始连通图，每次删除一些边，查询是否仍连通
- **解法**: 线段树分治 + 可撤销并查集
- **时间复杂度**: O((n + m) log m)
- **空间复杂度**: O(n + m)

### 5. 八纵八横 - P3733
- **题目链接**: https://www.luogu.com.cn/problem/P3733
- **难度**: 省选/NOI-
- **标签**: 线段树分治, Linear Basis, Union Find
- **题目描述**: 维护图，支持加边、删边、修改边权操作，查询从 1 号点出发回到 1 号点路径边权异或和的最大值
- **解法**: 线段树分治 + 带权并查集 + 线性基
- **时间复杂度**: O((n+q) log q L)
- **空间复杂度**: O(nL + qL)

### 6. 火星商店 - P4585
- **题目链接**: https://www.luogu.com.cn/problem/P4585
- **难度**: 省选/NOI-
- **标签**: 线段树分治, Persistent Trie
- **题目描述**: 维护 n 个商店，每个商店有商品，支持添加商品、查询操作，查询要求在特定商店范围内和时间范围内找到异或最大值
- **解法**: 线段树分治 + 可持久化 Trie
- **时间复杂度**: O((n+q) log q log V)
- **空间复杂度**: O((n+q) log V)

## 五、AtCoder 题目

### 1. 细胞分裂 (Cell Division) - AGC010C
- **题目链接**: https://atcoder.jp/contests/agc010/tasks/agc010_c
- **难度**: 2300
- **标签**: Union Find, Divide and Conquer

- **题目描述**: 分割矩形并计算每次分割后的连通分量数
- **解法**: 线段树分治 + 可撤销并查集
- **时间复杂度**: O((n + m) log m)
- **空间复杂度**: O(n + m)

### 2. 最小异或对查询 (Minimum Xor Pair Query) - ABC308G
- **题目链接**: https://atcoder.jp/contests/abc308/tasks/abc308_g
- **难度**: 600
- **标签**: Trie, Bit Manipulation
- **题目描述**: 维护集合，支持添加数字、删除数字、查询操作，查询集合中任意两个数的异或最小值
- **解法**: 01Trie + 在线维护
- **时间复杂度**: O(q log V)
- **空间复杂度**: O(q log V)

## 六、SPOJ 题目

### 1. 动态连通性 (DYNACON1)
- **题目链接**: https://www.spoj.com/problems/DYNACON1/
- **难度**: Hard
- **标签**: Segment Tree, Divide and Conquer, Union Find
- **题目描述**: 动态维护图的连通性，支持加边、删边和查询操作
- **解法**: 线段树分治 + 可撤销并查集
- **时间复杂度**: O((n + m) log m)
- **空间复杂度**: O(n + m)

### 2. 动态图的最小生成树 (DYNACON2)
- **题目链接**: https://www.spoj.com/problems/DYNACON2/
- **难度**: Hard
- **标签**: Segment Tree, Divide and Conquer, Union Find
- **题目描述**: 动态维护图的最小生成树相关查询
- **解法**: 线段树分治 + 可撤销并查集 + Kruskal 算法
- **时间复杂度**: O((n + m) log m log n)
- **空间复杂度**: O(n + m)

## 七、其他平台题目

### 1. 动态树 (HackerRank Dynamic Trees)
- **题目链接**: https://www.hackerrank.com/challenges/dynamic-trees
- **难度**: Advanced
- **标签**: Segment Tree, Divide and Conquer, Tree Data Structures
- **题目描述**: 动态维护树的结构和路径查询
- **解法**: 线段树分治 + 可撤销并查集
- **时间复杂度**: O((n + m) log m)

- **空间复杂度**: O(n + m)

### 2. USACO 相关题目
- **题目描述**: 在 USACO 竞赛中，线段树分治常用于解决牧场连接、路径查询等动态图问题
- **解法**: 线段树分治 + 可撤销并查集
- **时间复杂度**: O((n + m) log m)
- **空间复杂度**: O(n + m)

## 八、线段树分治典型应用场景

### 1. 动态图问题
- 动态连通性查询
- 二分图维护
- 最小生成树动态维护
- 异或路径查询

### 2. 区间操作问题
- 支持区间添加和删除的数据结构
- 区间内有效操作的维护

### 3. 可撤销操作问题
- 需要回滚操作的场景
- 多版本数据维护

### 4. 离线查询问题
- 所有查询可以预先知道
- 按时间顺序处理的问题

## 九、核心实现模板

### Java 版本核心实现
```java
// 可撤销并查集
class RollbackDSU {
    int[] father, size;
    Stack<int[]> rollbackStack = new Stack<>();

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

    boolean union(int x, int y) {
```

```
        int fx = find(x), fy = find(y);
        if (fx == fy) return false;
        // 按秩合并
        if (size[fx] < size[fy]) {
            int temp = fx; fx = fy; fy = temp;
        }
        father[fy] = fx;
        size[fx] += size[fy];
        rollbackStack.push(new int[]{fx, fy});
        return true;
    }

    void rollback() {
        int[] op = rollbackStack.pop();
        int fx = op[0], fy = op[1];
        father[fy] = fy;
        size[fx] -= size[fy];
    }
}

// 线段树分治 DFS 遍历
public static void dfs(int l, int r, int i) {
    int unionCnt = 0;

    // 处理当前节点上的所有边
    for (int e = head[i]; e > 0; e = next[e]) {
        if (union(tox[e], toy[e])) {
            unionCnt++;
        }
    }

    if (l == r) {
        // 处理叶子节点查询
        if (op[l] == 3) {
            ans[l] = (find(x[l]) == find(y[l]));
        }
    } else {
        // 递归处理左右子树
        int mid = (l + r) >> 1;
        dfs(l, mid, i << 1);
        dfs(mid + 1, r, i << 1 | 1);
    }
```

```
    // 回滚操作
    for (int k = 1; k <= unionCnt; k++) {
        undo();
    }
}
```

### C++版本核心实现
```cpp
// 可撤销并查集
class RollbackDSU {
private:
    vector<int> father;
    vector<int> size;
    vector<pair<int, int>> rollbackStack;

public:
    RollbackDSU(int n) {
        father.resize(n + 1);
        size.resize(n + 1, 1);
        for (int i = 1; i <= n; i++) {
            father[i] = i;
        }
    }

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

    bool unite(int x, int y) {
        int fx = find(x);
        int fy = find(y);
        if (fx == fy) return false;

        if (size[fx] < size[fy]) {
            swap(fx, fy);
        }

        rollbackStack.push_back({fx, fy});
        father[fy] = fx;
```

```
        size[fx] += size[fy];
        return true;
    }

    void rollback(int cnt) {
        while (cnt--) {
            auto [fx, fy] = rollbackStack.back();
            rollbackStack.pop_back();
            father[fy] = fy;
            size[fx] -= size[fy];
        }
    }
};

// 线段树分治 DFS 遍历
void dfs(int l, int r, int i) {
    int unionCnt = 0;

    // 处理当前节点上的所有边
    for (int e = head[i]; e; e = next_[e]) {
        if (unite(tox[e], toy[e])) {
            unionCnt++;
        }
    }

    if (l == r) {
        // 处理叶子节点查询
        if (op[l] == 3) {
            ans[l] = (find(x[l]) == find(y[l]));
        }
    } else {
        // 递归处理左右子树
        int mid = (l + r) >> 1;
        dfs(l, mid, i << 1);
        dfs(mid + 1, r, i << 1 | 1);
    }

    // 回滚操作
    for (int k = 1; k <= unionCnt; k++) {
        undo();
    }
}
```

### Python 版本核心实现

```python
class RollbackDSU:
    def __init__(self, n):
        self.father = list(range(n + 1))
        self.size = [1] * (n + 1)
        self.rollback_stack = []

    def find(self, x):
        # 不使用路径压缩
        while x != self.father[x]:
            x = self.father[x]
        return x

    def union(self, x, y):
        fx = self.find(x)
        fy = self.find(y)
        if fx == fy:
            return False

        if self.size[fx] < self.size[fy]:
            fx, fy = fy, fx

        self.rollback_stack.append((fx, fy))
        self.father[fy] = fx
        self.size[fx] += self.size[fy]
        return True

    def rollback(self, count):
        for _ in range(count):
            fx, fy = self.rollback_stack.pop()
            self.father[fy] = fy
            self.size[fx] -= self.size[fy]

def dfs(l, r, i):
    # 处理当前节点上的所有边
    union_cnt = 0
    e = head[i]
    while e > 0:
        if union(tox[e], toy[e]):
            union_cnt += 1
        e = next_[e]
```

```
    if l == r:
        # 处理叶子节点查询
        if op[l] == 3:
            ans[l] = (find(x[l]) == find(y[l]))
    else:
        # 递归处理左右子树
        mid = (l + r) >> 1
        dfs(l, mid, i << 1)
        dfs(mid + 1, r, i << 1 | 1)

    # 回滚操作
    for _ in range(union_cnt):
        undo()
```

## 十、练习建议

1. **从基础开始**：先掌握可撤销并查集的实现，理解如何支持回滚操作
2. **理解线段树分治的核心思想**：将时间轴划分为线段树结构，处理每个时间区间内的操作
3. **掌握扩展域并查集**：用于处理二分图检测等约束问题
4. **练习经典题目**：从简单的动态连通性问题开始，逐步挑战更复杂的题目
5. **注意实现细节**：特别是回滚操作的正确性和时间复杂度分析
6. **多语言实践**：使用 Java、C++、Python 三种语言实现，理解不同语言的特点
7. **性能优化**：学习各种优化技巧，如位运算优化、内存优化等

==========

文件: README.md
==========
# 线段树分治 (Segment Tree Divide and Conquer) - Class167

## 概述

线段树分治是一种离线算法技术，主要用于解决带有时间维度的图论问题和动态维护问题。它将操作序列按照时间轴建立线段树，然后通过 DFS 遍历线段树来处理各个时间区间内的操作。

## 核心思想

1. **离线处理**：将所有操作和查询离线，按照时间建立线段树
2. **区间操作**：将每个操作的影响时间段映射到线段树的节点上
3. **可撤销数据结构**：使用可撤销并查集等支持回滚操作的数据结构
4. **DFS 遍历**：通过 DFS 遍历线段树，处理每个节点的操作并及时回滚

## 关键技术点

### 1. 可撤销并查集 (Rollback DSU)

```java
class RollbackDSU {
    int[] father, size;
    Stack<int[]> rollbackStack = new Stack<>();

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

    void union(int x, int y) {
        int fx = find(x), fy = find(y);
        if (fx == fy) return;
        // 按秩合并
        if (size[fx] < size[fy]) {
            int temp = fx; fx = fy; fy = temp;
        }
        father[fy] = fx;
        size[fx] += size[fy];
        rollbackStack.push(new int[]{fx, fy});
    }

    void rollback() {
        int[] op = rollbackStack.pop();
        int fx = op[0], fy = op[1];
        father[fy] = fy;
        size[fx] -= size[fy];
    }
}
```

### 2. 扩展域并查集 (Extended Union Find)

用于二分图检测：

```java
// 对于节点 x，其在左侧的编号为 x，右侧的编号为 x+n
void union(int x, int y) {
```

```
    // x 的左侧与 y 的右侧连接
    // y 的左侧与 x 的右侧连接
    union(x, y + n);
    union(y, x + n);
}
```

## 经典题目详解

### 1. 博物馆劫案 (CF601E / Luogu P4585)

**题目描述**：
- 给定 n 件商品，每件商品有价值和重量
- 支持添加商品、删除商品、查询操作
- 查询操作要求计算背包问题的变形结果

**解法**：
- 线段树分治 + 动态规划
- 将每件商品的有效时间映射到线段树上
- 在 DFS 过程中维护背包 DP 状态

**时间复杂度**：O(qk log q + nk)
**空间复杂度**：O(qk)

### 2. 贪玩蓝月 (LOJ #6515)

**题目描述**：
- 双端队列维护装备，支持在两端添加和删除装备
- 查询操作要求在特定特征值范围内选择装备使得战斗力最大

**解法**：
- 线段树分治 + 动态规划
- 将每件装备的有效时间映射到线段树上
- 在 DFS 过程中维护模意义下的 DP 状态

**时间复杂度**：O(mp log m)
**空间复杂度**：O(mp)

### 3. 打印所有合法数 (CF981E)

**题目描述**：
- 给定一个数组，初始全为 0
- 支持区间加法操作，每种操作只能执行一次

- 查询能通过选择操作得到的所有可能最大值

**解法**：
- 线段树分治 + 位运算优化 DP
- 使用位图记录所有可能的状态
- 通过位运算优化状态转移

**时间复杂度**：O(nq log q)
**空间复杂度**：O(n)

### 4. 异或最短路 (CF938G)

**题目描述**：
- 维护一个图，支持加边、删边操作
- 查询两点间路径边权异或和的最小值

**解法**：
- 线段树分治 + 带权并查集 + 线性基
- 利用线性基维护异或运算的性质
- 通过带权并查集维护连通性和路径异或值

**时间复杂度**：O((n+q) log q log V)
**空间复杂度**：O(n + q)

### 5. 八纵八横 (Luogu P3733)

**题目描述**：
- 维护一个图，支持加边、删边、修改边权操作
- 查询从 1 号点出发回到 1 号点路径边权异或和的最大值

**解法**：
- 线段树分治 + 带权并查集 + 线性基
- 使用 BitSet 处理大整数异或运算
- 维护线性基支持异或最大值查询

**时间复杂度**：O((n+q) log q L)
**空间复杂度**：O(nL + qL)，其中 L 为边权长度

### 6. 火星商店 (Luogu P4585)

**题目描述**：
- 维护 n 个商店，每个商店有商品
- 支持添加商品、查询操作

- 查询要求在特定商店范围内和时间范围内找到异或最大值

**解法**：
- 线段树分治 + 可持久化 Trie
- 二维限制（商店编号和时间）的处理
- 使用可持久化 Trie 维护异或最大值查询

**时间复杂度**：O((n+q) log q log V)
**空间复杂度**：O((n+q) log V)

### 7. 最小异或查询 (ABC308G)

**题目描述**：
- 维护一个集合，支持添加数字、删除数字、查询操作
- 查询操作要求找到集合中任意两个数的异或最小值

**解法**：
- 01Trie + 在线维护
- 通过 Trie 维护数字集合
- 实时计算最小异或值

**时间复杂度**：O(q log V)
**空间复杂度**：O(q log V)

### 8. 二分图检测 (CF813F)

**题目描述**：
- 维护一个图，支持加边、删边操作
- 每次操作后检测图是否为二分图

**解法**：
- 线段树分治 + 扩展域并查集
- 使用扩展域并查集维护二分图性质
- 通过线段树分治处理时间维度

**时间复杂度**：O((n+q) log q)
**空间复杂度**：O(n + q)

## 更多线段树分治经典题目

### LeetCode 题目

#### 1. 动态图连通性 (Dynamic Graph Connectivity)

- **题目链接**: https://leetcode.com/problems/dynamic-graph-connectivity/
- **难度**: Hard
- **标签**: Union Find, Segment Tree, Divide and Conquer
- **题目描述**: 支持动态加边、删边操作，查询两点间连通性
- **解法**: 线段树分治 + 可撤销并查集
- **时间复杂度**: O((n + m) log m)
- **空间复杂度**: O(n + m)

#### 2. 不良数对计数 (Count Number of Bad Pairs)
- **题目链接**: https://leetcode.com/problems/count-number-of-bad-pairs/
- **难度**: Medium
- **标签**: Segment Tree, Divide and Conquer, Math
- **题目描述**: 统计满足特定条件的数对数量
- **解法**: 线段树分治 + 数学变换
- **时间复杂度**: O(n log n)
- **空间复杂度**: O(n)

#### 3. 带阈值的图连通性 (Graph Connectivity With Threshold)
- **题目链接**: https://leetcode.com/problems/graph-connectivity-with-threshold/
- **难度**: Hard
- **标签**: Union Find, Math, Segment Tree, Divide and Conquer
- **题目描述**: 给定 n 个城市，编号 1 到 n，当两个城市的最大公约数大于 threshold 时它们直接相连，查询任意两个城市是否连通
- **解法**: 线段树分治 + 可撤销并查集
- **时间复杂度**: O(n log n + q log n)
- **空间复杂度**: O(n)

### Codeforces 题目

#### 1. 唯一出现次数 (Unique Occurrences) - 1681F
- **题目链接**: https://codeforces.com/contest/1681/problem/F
- **难度**: 2600
- **标签**: Segment Tree, Divide and Conquer, Union Find, Tree
- **题目描述**: 统计树上路径中唯一出现的颜色数量
- **解法**: 线段树分治 + 可撤销并查集
- **时间复杂度**: O((n + m) log m)
- **空间复杂度**: O(n + m)

#### 2. 边着色 (Painting Edges) - 576E
- **题目链接**: https://codeforces.com/contest/576/problem/E
- **难度**: 3300
- **标签**: Segment Tree, Divide and Conquer, Union Find, Graph
- **题目描述**: 给边着色使得每种颜色构成的子图都是二分图

- **解法**: 线段树分治 + 多个扩展域并查集
- **时间复杂度**: O((n + m) log m)
- **空间复杂度**: O(n + m)

### 洛谷题目

#### 1. 二分图 /【模板】线段树分治 - P5787
- **题目链接**: https://www.luogu.com.cn/problem/P5787
- **难度**: 省选/NOI-
- **标签**: 线段树分治, 扩展域并查集, 二分图
- **题目描述**: 维护动态图使其为二分图
- **解法**: 线段树分治 + 扩展域并查集
- **时间复杂度**: O((n + m) log m)
- **空间复杂度**: O(n + m)

#### 2. 最小 mex 生成树 - P5631
- **题目链接**: https://www.luogu.com.cn/problem/P5631
- **难度**: 省选/NOI-
- **标签**: 线段树分治, 并查集, 生成树, 二分
- **题目描述**: 求生成树使得边权集合的 mex 最小
- **解法**: 线段树分治 + 可撤销并查集 + 二分答案
- **时间复杂度**: O((n + m) log m log n)
- **空间复杂度**: O(n + m)

#### 3. 大融合 - P4219
- **题目链接**: https://www.luogu.com.cn/problem/P4219
- **难度**: 省选/NOI-
- **标签**: 线段树分治, 并查集, 图论
- **题目描述**: 支持加边和查询边负载，边负载定义为删去该边后两个连通块大小的乘积
- **解法**: 线段树分治 + 可撤销并查集
- **时间复杂度**: O((n + m) log m)
- **空间复杂度**: O(n + m)

#### 4. 连通图 - P5227
- **题目链接**: https://www.luogu.com.cn/problem/P5227
- **难度**: 省选/NOI-
- **标签**: 线段树分治, 并查集, 图论
- **题目描述**: 给定初始连通图，每次删除一些边，查询是否仍连通
- **解法**: 线段树分治 + 可撤销并查集
- **时间复杂度**: O((n + m) log m)
- **空间复杂度**: O(n + m)

### AtCoder 题目

#### 1. 细胞分裂 (Cell Division) - AGC010C
- **题目链接**: https://atcoder.jp/contests/agc010/tasks/agc010_c
- **难度**: 2300
- **标签**: Union Find, Divide and Conquer
- **题目描述**: 分割矩形并计算每次分割后的连通分量数
- **解法**: 线段树分治 + 可撤销并查集
- **时间复杂度**: O((n + m) log m)
- **空间复杂度**: O(n + m)

## 算法复杂度分析

线段树分治的典型复杂度：
- 时间复杂度：O((n + m) log m * k)
- 空间复杂度：O((n + m) log m)

其中：
- n 为数据规模（点数、商品数等）
- m 为操作数
- k 为单次操作的复杂度
- log m 来自于线段树的深度

## 实现要点

1. **不能路径压缩**：为了支持撤销操作，只能使用按秩合并
2. **离线处理**：所有操作必须预先知道
3. **精确回滚**：每次操作后必须准确回滚到操作前状态
4. **时间区间映射**：将操作的生效时间区间正确映射到线段树节点
5. **合理选择数据结构**：根据具体问题选择合适的可撤销数据结构

## 应用场景

线段树分治主要适用于以下场景：

1. **动态图问题**：支持加边、删边操作的图论问题
2. **区间维护问题**：需要维护某个区间内有效操作的问题
3. **可撤销操作**：需要支持操作回滚的数据结构问题
4. **离线查询**：所有查询可以预先知道的情况

## 解题思路与技巧总结

1. **识别问题特征**：
   - 问题中存在时间维度或区间有效性

- 支持添加和删除操作
- 可以离线处理所有操作和查询

2. **关键步骤**：
   - 离线处理所有操作，确定每个操作的有效时间区间
   - 构建时间轴线段树，将操作映射到对应的区间
   - 选择合适的可撤销数据结构（并查集、线性基等）
   - DFS 遍历线段树，处理区间操作并在回溯时回滚

3. **优化技巧**：
   - 使用按秩合并代替路径压缩以支持撤销
   - 合理设计数据结构以提高回滚效率
   - 对于大规模数据，考虑位运算和空间优化

4. **常见误区**：
   - 忽略路径压缩对撤销的影响
   - 时间区间映射错误
   - 回滚操作不完整导致状态错误

5. **工程化考量**：
   - 异常处理：处理无效操作和边界情况
   - 内存管理：避免过大的数据结构导致内存溢出
   - 性能优化：针对特定问题调整数据结构实现

通过掌握线段树分治这一强大的离线算法技术，可以高效解决许多动态维护问题，特别是在图论和数据结构领域有着广泛的应用。

1. **动态图问题**：加边、删边操作下的图性质维护
2. **二分图维护**：动态维护图的二分性
3. **连通性查询**：动态图的连通性相关查询
4. **生成树问题**：动态维护生成树相关性质
5. **异或相关问题**：维护异或运算的性质和查询
6. **背包问题**：动态维护背包状态

## 注意事项

1. 可撤销并查集不能使用路径压缩，只能按秩合并
2. 线段树分治是离线算法，不支持在线查询
3. 每个操作的影响时间区间要正确计算
4. 回滚操作必须与合并操作一一对应
5. 注意空间复杂度，线段树分治通常空间消耗较大

==========

文件: SegmentTreeDivideAndConquerProblems.md
==========
# 线段树分治经典题目汇总

## 一、LeetCode 题目

### 1. 动态图连通性 (Dynamic Graph Connectivity)
- **题目链接**: https://leetcode.com/problems/dynamic-graph-connectivity/
- **难度**: Hard
- **标签**: Union Find, Segment Tree, Divide and Conquer
- **题目描述**: 支持动态加边、删边操作，查询两点间连通性
- **解法**: 线段树分治 + 可撤销并查集
- **时间复杂度**: O((n + m) log m)
- **空间复杂度**: O(n + m)
- **核心思想**:
  ```java
  // 可撤销并查集实现
  public boolean union(int x, int y) {
      int fx = find(x);
      int fy = find(y);
      if (fx == fy) return false;
      // 按秩合并
      if (size[fx] < size[fy]) {
          int tmp = fx;
          fx = fy;
          fy = tmp;
      }
      father[fy] = fx;
      size[fx] += size[fy];
      // 记录操作以便撤销
      rollbackStack.push(new int[]{fx, fy});
      return true;
  }
  ```

### 2. 不良数对计数 (Count Number of Bad Pairs)
- **题目链接**: https://leetcode.com/problems/count-number-of-bad-pairs/
- **难度**: Medium
- **标签**: Segment Tree, Divide and Conquer, Math
- **题目描述**: 统计满足特定条件的数对数量
- **解法**: 线段树分治 + 数学变换
- **时间复杂度**: O(n log n)

- **空间复杂度**: O(n)
- **优化技巧**:
  - 通过数学变换将问题转化为统计符合条件的数对
  - 使用线段树维护区间信息

### 3. 带阈值的图连通性 (Graph Connectivity With Threshold)
- **题目链接**: https://leetcode.com/problems/graph-connectivity-with-threshold/
- **难度**: Hard
- **标签**: Union Find, Math, Segment Tree, Divide and Conquer
- **题目描述**: 给定 n 个城市，编号 1 到 n，当两个城市的最大公约数大于 threshold 时它们直接相连，查询任意两个城市是否连通
- **解法**: 线段树分治 + 可撤销并查集
- **时间复杂度**: O(n log n + q log n)
- **空间复杂度**: O(n)
- **算法思路**:
  - 对于每个阈值，使用线段树分治处理不同阈值范围内的连通性
  - 利用数学性质优化连通性判断

## 二、Codeforces 题目

### 1. 二分图检测 (Bipartite Checking) - 813F
- **题目链接**: https://codeforces.com/contest/813/problem/F
- **难度**: 2400
- **标签**: Segment Tree, Divide and Conquer, Union Find, Bipartite Graph
- **题目描述**: 动态维护图的二分性
- **解法**: 线段树分治 + 扩展域并查集
- **时间复杂度**: O((n + m) log m)
- **空间复杂度**: O(n + m)
- **核心实现**:
  ```python
  # Python 版本的扩展域并查集实现
  def union(x, y):
      nonlocal opsize
      fx1 = find(x)
      fy1 = find(y)
      if fx1 == fy1:  # 如果 x 和 y 在同一集合，说明不是二分图
          return False
      fx2 = find(x + n)
      fy2 = find(y + n)
      # 合并 x 的左侧与 y 的右侧
      if fx1 != fy2:
          if siz[fx1] < siz[fy2]:
              fx1, fy2 = fy2, fx1
```

```
            father[fy2] = fx1
            siz[fx1] += siz[fy2]
            opsize += 1
            rollback[opsize][0] = fx1
            rollback[opsize][1] = fy2
        # 合并 y 的左侧与 x 的右侧
        if fx2 != fy1:
            if siz[fx2] < siz[fy1]:
                fx2, fy1 = fy1, fx2
            father[fy1] = fx2
            siz[fx2] += siz[fy1]
            opsize += 1
            rollback[opsize][0] = fx2
            rollback[opsize][1] = fy1
        return True
```

### 2. 唯一出现次数 (Unique Occurrences) - 1681F
- **题目链接**: https://codeforces.com/contest/1681/problem/F
- **难度**: 2600
- **标签**: Segment Tree, Divide and Conquer, Union Find, Tree
- **题目描述**: 统计树上路径中唯一出现的颜色数量
- **解法**: 线段树分治 + 可撤销并查集
- **时间复杂度**: O((n + m) log m)
- **空间复杂度**: O(n + m)
- **关键点**:
  - 离线处理所有颜色查询
  - 使用线段树分治维护颜色出现的时间区间
  - 利用并查集合并相同颜色的节点

### 3. 边着色 (Painting Edges) - 576E
- **题目链接**: https://codeforces.com/contest/576/problem/E
- **难度**: 3300
- **标签**: Segment Tree, Divide and Conquer, Union Find, Graph
- **题目描述**: 给边着色使得每种颜色构成的子图都是二分图
- **解法**: 线段树分治 + 多个扩展域并查集
- **时间复杂度**: O((n + m) log m)
- **空间复杂度**: O(n + m)
- **算法思路**:
  - 对每种颜色维护一个扩展域并查集
  - 使用线段树分治处理颜色变更操作
  - 实时检测二分图性质

### 4. 博物馆劫案 (Museum Robbery) - 601E
- **题目链接**: https://codeforces.com/problemset/problem/601/E
- **难度**: 2800
- **标签**: Segment Tree, Divide and Conquer, Dynamic Programming
- **题目描述**: 维护商品集合，支持添加、删除商品，查询背包问题变形结果
- **解法**: 线段树分治 + 动态规划
- **时间复杂度**: O(qk log q + nk)
- **空间复杂度**: O(qk)
- **核心实现**:
  ```java
  // 线段树分治的DFS过程
  public static void dfs(int l, int r, int i, int dep) {
      // 备份当前DP状态
      clone(backup[dep], dp);
      // 处理当前区间的所有商品
      for (int e = head[i]; e > 0; e = next[e]) {
          int v = tov[e];
          int w = tow[e];
          // 0-1背包的逆序更新
          for (int j = k; j >= w; j--) {
              dp[j] = Math.max(dp[j], dp[j - w] + v);
          }
      }
      // 处理叶子节点（查询操作）
      if (l == r) {
          if (op[l] == 3) {
              long ret = 0;
              long base = 1;
              for (int j = 1; j <= k; j++) {
                  ret = (ret + dp[j] * base) % MOD;
                  base = (base * BAS) % MOD;
              }
              ans[l] = ret;
          }
      } else {
          // 递归处理左右子树
          int mid = (l + r) >> 1;
          dfs(l, mid, i << 1, dep + 1);
          dfs(mid + 1, r, i << 1 | 1, dep + 1);
      }
      // 恢复DP状态
      clone(dp, backup[dep]);
  }
```

```

### 5. 线段上的加法 (Addition on Segments) - 981E
- **题目链接**: https://codeforces.com/problemset/problem/981/E
- **难度**: 2200
- **标签**: Segment Tree, Divide and Conquer, Bit Manipulation, Dynamic Programming
- **题目描述**: 给定数组初始全为0，支持区间加法操作，每种操作只能执行一次，查询能通过选择操作得到的所有可能最大值
- **解法**: 线段树分治 + 位运算优化DP
- **时间复杂度**: O(nq log q)
- **空间复杂度**: O(n)
- **优化技巧**:
  - 使用位运算表示所有可能的状态
  - 线段树分治处理区间操作
  - 利用位运算加速状态转移

### 6. 异或最短路 (Shortest Path Queries) - 938G
- **题目链接**: https://codeforces.com/problemset/problem/938/G
- **难度**: 2900
- **标签**: Segment Tree, Divide and Conquer, Linear Basis, Union Find
- **题目描述**: 维护图，支持加边、删边操作，查询两点间路径边权异或和的最小值
- **解法**: 线段树分治 + 带权并查集 + 线性基
- **时间复杂度**: O((n+q) log q log V)
- **空间复杂度**: O(n + q)
- **核心实现**:
  ```java
  // 插入线性基
  public static void insert(int num) {
      for (int i = BIT; i >= 0; i--) {
          if ((num >> i & 1) == 1) {
              if (basis[i] == 0) {
                  basis[i] = num;
                  inspos[basiz++] = i;
                  return;
              }
              num ^= basis[i];
          }
      }
  }

  // 计算最小异或值
  public static int minEor(int num) {
      for (int i = BIT; i >= 0; i--) {

```
            num = Math.min(num, num ^ basis[i]);
        }
        return num;
    }
```

## 三、洛谷题目

### 1. 二分图 /【模板】线段树分治 - P5787
- **题目链接**: https://www.luogu.com.cn/problem/P5787
- **难度**: 省选/NOI-
- **标签**: 线段树分治, 扩展域并查集, 二分图
- **题目描述**: 维护动态图使其为二分图
- **解法**: 线段树分治 + 扩展域并查集
- **时间复杂度**: O((n + m) log m)
- **空间复杂度**: O(n + m)
- **实现要点**:
  - 扩展域并查集的正确实现
  - 时间区间的准确映射
  - 回滚操作的正确性

### 2. 最小 mex 生成树 - P5631
- **题目链接**: https://www.luogu.com.cn/problem/P5631
- **难度**: 省选/NOI-
- **标签**: 线段树分治, 并查集, 生成树, 二分
- **题目描述**: 求生成树使得边权集合的 mex 最小
- **解法**: 线段树分治 + 可撤销并查集 + 二分答案
- **时间复杂度**: O((n + m) log m log n)
- **空间复杂度**: O(n + m)
- **算法思路**:
  - 二分可能的 mex 值
  - 对每个候选值，使用线段树分治判断是否存在生成树
  - 利用 Kruskal 算法和可撤销并查集维护生成树

### 3. 大融合 - P4219
- **题目链接**: https://www.luogu.com.cn/problem/P4219
- **难度**: 省选/NOI-
- **标签**: 线段树分治, 并查集, 图论
- **题目描述**: 支持加边和查询边负载，边负载定义为删去该边后两个连通块大小的乘积
- **解法**: 线段树分治 + 可撤销并查集
- **时间复杂度**: O((n + m) log m)
- **空间复杂度**: O(n + m)
- **关键点**:

  - 离线处理所有操作
  - 使用并查集维护连通块大小
  - 线段树分治处理时间区间

### 4. 连通图 - P5227
- **题目链接**: https://www.luogu.com.cn/problem/P5227
- **难度**: 省选/NOI-
- **标签**: 线段树分治, 并查集, 图论
- **题目描述**: 给定初始连通图，每次删除一些边，查询是否仍连通
- **解法**: 线段树分治 + 可撤销并查集
- **时间复杂度**: O((n + m) log m)
- **空间复杂度**: O(n + m)
- **逆向思维**:
  - 将删除操作转换为添加操作
  - 从最终状态逆向构建连通性
  - 使用线段树分治处理时间区间

### 5. 八纵八横 - P3733
- **题目链接**: https://www.luogu.com.cn/problem/P3733
- **难度**: 省选/NOI-
- **标签**: 线段树分治, Linear Basis, Union Find
- **题目描述**: 维护图，支持加边、删边、修改边权操作，查询从 1 号点出发回到 1 号点路径边权异或和的最大值
- **解法**: 线段树分治 + 带权并查集 + 线性基
- **时间复杂度**: O((n+q) log q L)
- **空间复杂度**: O(nL + qL)
- **核心实现**:
  ```java
  // 自定义 BitSet 处理大整数异或
  static class BitSet {
      public int len;
      public int[] arr;

      public BitSet() {
          len = BIT / INT_BIT + 1;
          arr = new int[len];
      }

      public void eor(BitSet other) {
          for (int i = 0; i < len; i++) {
              arr[i] ^= other.arr[i];
          }
      }
  ```

```
}
```

### 6. 火星商店 - P4585
- **题目链接**: https://www.luogu.com.cn/problem/P4585
- **难度**: 省选/NOI-
- **标签**: 线段树分治, Persistent Trie
- **题目描述**: 维护 n 个商店，每个商店有商品，支持添加商品、查询操作，查询要求在特定商店范围内和时间范围内找到异或最大值
- **解法**: 线段树分治 + 可持久化 Trie
- **时间复杂度**: O((n+q) log q log V)
- **空间复杂度**: O((n+q) log V)
- **算法思路**:
  - 二维限制（商店编号和时间）的处理
  - 线段树分治维护时间维度
  - 可持久化 Trie 维护异或最大值查询

## 四、AtCoder 题目

### 1. 细胞分裂 (Cell Division) - AGC010C
- **题目链接**: https://atcoder.jp/contests/agc010/tasks/agc010_c
- **难度**: 2300
- **标签**: Union Find, Divide and Conquer
- **题目描述**: 分割矩形并计算每次分割后的连通分量数
- **解法**: 线段树分治 + 可撤销并查集
- **时间复杂度**: O((n + m) log m)
- **空间复杂度**: O(n + m)
- **实现要点**:
  - 离线处理所有分割操作
  - 使用并查集维护连通分量
  - 线段树分治处理时间区间

### 2. 最小异或对查询 (Minimum Xor Pair Query) - ABC308G
- **题目链接**: https://atcoder.jp/contests/abc308/tasks/abc308_g
- **难度**: 600
- **标签**: Trie, Bit Manipulation
- **题目描述**: 维护集合，支持添加数字、删除数字、查询操作，查询集合中任意两个数的异或最小值
- **解法**: 01Trie + 在线维护
- **时间复杂度**: O(q log V)
- **空间复杂度**: O(q log V)
- **算法思路**:
  - 使用 01Trie 树维护数字集合
  - 对于每个数字，在 Trie 中查找异或最小的数

- 支持动态插入和删除操作

## 五、其他平台题目

### 1. 动态连通性 (SPOJ DYNACON1)
- **题目链接**: https://www.spoj.com/problems/DYNACON1/
- **难度**: Hard
- **标签**: Segment Tree, Divide and Conquer, Union Find
- **题目描述**: 动态维护图的连通性，支持加边、删边和查询操作
- **解法**: 线段树分治 + 可撤销并查集
- **时间复杂度**: O((n + m) log m)
- **空间复杂度**: O(n + m)

### 2. 动态图的最小生成树 (SPOJ DYNALCA)
- **题目链接**: https://www.spoj.com/problems/DYNALCA/
- **难度**: Hard
- **标签**: Segment Tree, Divide and Conquer, Union Find
- **题目描述**: 动态维护图的最小生成树相关查询
- **解法**: 线段树分治 + 可撤销并查集 + Kruskal 算法
- **时间复杂度**: O((n + m) log m log n)
- **空间复杂度**: O(n + m)

### 3. 动态树 (HackerRank Dynamic Trees)
- **题目链接**: https://www.hackerrank.com/challenges/dynamic-trees
- **难度**: Advanced
- **标签**: Segment Tree, Divide and Conquer, Tree Data Structures
- **题目描述**: 动态维护树的结构和路径查询
- **解法**: 线段树分治 + 可撤销并查集
- **时间复杂度**: O((n + m) log m)
- **空间复杂度**: O(n + m)

## 六、线段树分治的核心实现与优化

### 1. 可撤销并查集的优化实现

**Java 版本**:
```java
class RollbackDSU {
    int[] father;
    int[] size;
    int[][] rollbackStack;
    int stackSize;
```

```
public RollbackDSU(int n) {
    father = new int[n + 1];
    size = new int[n + 1];
    rollbackStack = new int[300000][2]; // 预分配足够空间
    stackSize = 0;

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

public int find(int x) {
    // 注意：不能使用路径压缩，否则无法撤销
    while (x != father[x]) {
        x = father[x];
    }
    return x;
}

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

    // 按秩合并
    if (size[fx] < size[fy]) {
        int tmp = fx;
        fx = fy;
        fy = tmp;
    }

    // 记录操作前的状态
    rollbackStack[stackSize][0] = fx;
    rollbackStack[stackSize++][1] = fy;

    father[fy] = fx;
    size[fx] += size[fy];
    return true;
}

public void rollback(int version) {
    // 回滚到指定版本
```

```
        while (stackSize > version) {
            stackSize--;
            int fx = rollbackStack[stackSize][0];
            int fy = rollbackStack[stackSize][1];
            father[fy] = fy;
            size[fx] -= size[fy];
        }
    }
}
```

**C++版本**:
```cpp
class RollbackDSU {
private:
    vector<int> father;
    vector<int> size;
    vector<pair<int, int>> rollbackStack;

public:
    RollbackDSU(int n) {
        father.resize(n + 1);
        size.resize(n + 1, 1);
        for (int i = 1; i <= n; i++) {
            father[i] = i;
        }
    }

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

    bool unite(int x, int y) {
        int fx = find(x);
        int fy = find(y);
        if (fx == fy) return false;

        if (size[fx] < size[fy]) {
            swap(fx, fy);
        }
```

```
        rollbackStack.push_back({fx, fy});
        father[fy] = fx;
        size[fx] += size[fy];
        return true;
    }

    void rollback(int cnt) {
        while (cnt--) {
            auto [fx, fy] = rollbackStack.back();
            rollbackStack.pop_back();
            father[fy] = fy;
            size[fx] -= size[fy];
        }
    }
};
```

**Python 版本**:
```python
class RollbackDSU:
    def __init__(self, n):
        self.father = list(range(n + 1))
        self.size = [1] * (n + 1)
        self.rollback_stack = []

    def find(self, x):
        # 不使用路径压缩
        while x != self.father[x]:
            x = self.father[x]
        return x

    def union(self, x, y):
        fx = self.find(x)
        fy = self.find(y)
        if fx == fy:
            return False

        if self.size[fx] < self.size[fy]:
            fx, fy = fy, fx

        self.rollback_stack.append((fx, fy))
        self.father[fy] = fx
```

```
        self.size[fx] += self.size[fy]
        return True

    def rollback(self, count):
        for _ in range(count):
            fx, fy = self.rollback_stack.pop()
            self.father[fy] = fy
            self.size[fx] -= self.size[fy]
```

### 2. 线段树分治的通用框架

```java
// 线段树分治的通用框架
public static void solve() {
    // 1. 离线处理所有操作，确定每个操作的有效时间区间
    processOperations();

    // 2. 构建时间轴线段树，将操作映射到对应的区间
    buildSegmentTree();

    // 3. DFS 遍历线段树，处理操作
    dfs(1, 1, q);

    // 4. 输出答案
    outputResults();
}

public static void dfs(int node, int l, int r) {
    // 记录当前操作次数，用于回滚
    int currentOps = opsize;

    // 处理当前节点的所有操作
    for (int e = head[node]; e > 0; e = next[e]) {
        // 根据具体问题处理操作
        processOperation(tox[e], toy[e], tow[e]);
    }

    if (l == r) {
        // 叶子节点，处理查询
        handleQuery(l);
    } else {
        // 递归处理左右子树
```

```
        int mid = (l + r) >> 1;
        dfs(node << 1, l, mid);
        dfs(node << 1 | 1, mid + 1, r);
    }

    // 回滚操作
    rollbackTo(currentOps);
}
```

### 3. 常见优化技巧

1. **内存优化**：
   - 预分配足够的数组空间，避免动态扩容
   - 使用数组代替集合类，提高效率

2. **时间优化**：
   - 使用按秩合并代替路径压缩
   - 位运算加速状态转移
   - 剪枝优化，提前终止无效搜索

3. **异常处理**：
   - 处理无效输入和边界情况
   - 检查数组越界和空指针

4. **调试技巧**：
   - 打印中间状态值
   - 使用断言验证关键条件
   - 分段测试功能模块

## 七、线段树分治的应用场景总结

线段树分治特别适合以下类型的问题：

1. **动态图论问题**：
   - 动态连通性查询
   - 二分图维护
   - 最小生成树动态维护
   - 异或路径查询

2. **区间操作问题**：
   - 支持区间添加和删除的数据结构
   - 区间内有效操作的维护

3. **可撤销操作问题**：
   - 需要回滚操作的场景
   - 多版本数据维护

4. **离线查询问题**：
   - 所有查询可以预先知道
   - 按时间顺序处理的问题

通过掌握线段树分治这一强大的离线算法技术，可以解决许多复杂的动态维护问题，尤其在图论和数据结构领域有着广泛的应用。在实际工程中，线段树分治也是处理动态问题的重要工具之一。
- **时间复杂度**: O(nq log q)
- **空间复杂度**: O(n)
- **关键点**:
  - 使用位图记录所有可能的状态
  - 通过位运算优化状态转移
  - bitLeft 函数实现位图左移操作

### 4. 异或最短路 (CF938G)
- **题目链接**: https://codeforces.com/problemset/problem/938/G
- **题目描述**: 维护图，支持加边、删边操作，查询两点间路径边权异或和的最小值
- **解法**: 线段树分治 + 带权并查集 + 线性基
- **时间复杂度**: O((n+q) log q log V)
- **空间复杂度**: O(n + q)
- **关键点**:
  - 带权并查集维护连通性和路径异或值
  - 线性基维护异或运算的性质
  - event 数组记录操作的时间区间

### 5. 八纵八横 (Luogu P3733)
- **题目链接**: https://www.luogu.com.cn/problem/P3733
- **题目描述**: 维护图，支持加边、删边、修改边权操作，查询从 1 号点出发回到 1 号点路径边权异或和的最大值
- **解法**: 线段树分治 + 带权并查集 + 线性基
- **时间复杂度**: O((n+q) log q L)
- **空间复杂度**: O(nL + qL)
- **关键点**:
  - 使用 BitSet 处理大整数异或运算
  - 带权并查集维护路径异或值
  - 线性基维护异或最大值查询

### 6. 火星商店 (Luogu P4585)
- **题目链接**: https://www.luogu.com.cn/problem/P4585

- **题目描述**: 维护 n 个商店，每个商店有商品，支持添加商品、查询操作，查询要求在特定商店范围内和时间范围内找到异或最大值
- **解法**: 线段树分治 + 可持久化 Trie
- **时间复杂度**: O((n+q) log q log V)
- **空间复杂度**: O((n+q) log V)
- **关键点**:
  - 二维限制（商店编号和时间）的处理
  - 可持久化 Trie 维护异或最大值查询
  - product 数组存储商品信息

### 7. 最小异或查询 (ABC308G)
- **题目链接**: https://atcoder.jp/contests/abc308/tasks/abc308_g
- **题目描述**: 维护集合，支持添加数字、删除数字、查询操作，查询集合中任意两个数的异或最小值
- **解法**: 01Trie + 在线维护
- **时间复杂度**: O(q log V)
- **空间复杂度**: O(q log V)
- **关键点**:
  - 01Trie 维护数字集合
  - 实时计算最小异或值
  - mineor 数组维护子树最小异或值

## 六、解题技巧总结

### 1. 线段树分治适用场景
- 有时间维度的操作序列
- 操作有明确的生效时间区间
- 需要支持撤销操作的数据结构
- 离线处理问题

### 2. 常用数据结构组合
- 线段树分治 + 可撤销并查集：连通性问题
- 线段树分治 + 扩展域并查集：二分图问题
- 线段树分治 + 线性基：异或相关问题
- 线段树分治 + 动态规划：状态维护问题

### 3. 实现要点
- 正确计算操作的时间区间
- 合理设计可撤销数据结构
- 准确实现回滚操作
- 注意空间复杂度的控制
- 优化状态转移过程

### 4. 常见优化技巧

- 位运算优化状态表示和转移
- 可持久化数据结构处理多维限制
- 线性基处理异或运算
- 扫描线处理区间操作

==========

[代码文件]
==========
文件: Code01_MuseumRobbery1.java
==========
```
package class167;

/**
 * 博物馆劫案 (Museum Robbery) - 线段树分治 + 动态规划实现
 *
 * 题目来源：
 * - Codeforces 601E
 * - 洛谷 P4585
 *
 * 问题描述：
 * - 初始有 n 件商品，每件商品有价值 v 和重量 w
 * - 支持三种操作：添加商品、删除商品、查询 f(s)
 * - f(s)定义为：Σ(m=1~k) [a(s, m) * BAS^(m-1)] mod MOD，其中 a(s, m)表示总重量≤m 时的最大价值
 *
 * 约束条件：
 * - 1 ≤ n ≤ 5 × 10³
 * - 1 ≤ q ≤ 3 × 10⁴
 * - 1 ≤ k ≤ 10³
 * - 每件商品重量 ≤ 10³
 * - 每件商品价值 ≤ 10⁶
 *
 * 算法思路：
 * 1. 线段树分治：将每个商品的有效时间段映射到线段树的节点上
 * 2. 动态规划：在 DFS 过程中维护背包 DP 数组，处理每个时间段内的商品
 * 3. 回溯机制：使用备份数组在 DFS 回溯时恢复 DP 状态
 * 4. 高效输入：使用 FastReader 类优化大规模数据的输入
 *
 * 时间复杂度分析：
 * - 预处理时间：O(n + q)
 * - 线段树构建与边添加：O(n log q)
 * - DFS 遍历线段树：O(q log q × k)
 * - 总体时间复杂度：O(n log q + q log q × k)
```

```
 *
 * 空间复杂度分析：
 * - 线段树存储：O(n log q)
 * - DP 数组和备份：O(k × log q)
 * - 其他数组：O(n + q)
 * - 总体空间复杂度：O(n log q + q + k log q)
 *
 * 实现细节与优化：
 * 1. 使用链式前向星存储线段树节点上的商品
 * 2. DFS 过程中维护深度，用于确定备份数组的索引
 * 3. 逆序遍历背包更新，避免重复计算
 * 4. 使用 FastReader 进行高效输入，避免超时
 * 5. 预计算每个商品的有效时间区间
 *
 * 语言对比分析：
 * - Java：代码结构清晰，提供良好的内存管理，但需要注意输入效率问题
 * - C++：效率更高，但需要手动管理内存，实现细节更复杂
 * - Python：实现简单但效率较低，对于大规模数据可能超时
 *
 * 工程化考量：
 * 1. 异常处理：未处理输入错误或边界情况
 * 2. 内存优化：常量数组大小预定义，避免动态扩容开销
 * 3. 性能调优：使用位运算和数组复用优化内存访问
 * 4. 测试用例：应测试极端数据如最大 n 和 q、k=1 等情况
 */

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

public class Code01_MuseumRobbery1 {

    // 常量定义
    public static int MAXN = 40001;      // 最大商品数量（初始 n + 最大 q 操作数）
    public static int MAXQ = 30001;      // 最大操作数量
    public static int MAXK = 1001;       // 最大重量限制
    public static int MAXT = 1000001;    // 最大边数（线段树上的边）
    public static int DEEP = 20;         // 最大深度（线段树深度）
    public static int BAS = 10000019;    // 基数
    public static int MOD = 1000000007;  // 模数
    public static int n, k, q;           // 当前商品数、重量限制、操作数
```

```
// 商品信息
public static int[] v = new int[MAXN]; // 商品价值数组
public static int[] w = new int[MAXN]; // 商品重量数组

// 操作信息
public static int[] op = new int[MAXQ]; // 操作类型数组
public static int[] x = new int[MAXQ];  // 操作参数 x
public static int[] y = new int[MAXQ];  // 操作参数 y

// 时间区间信息
// 第 i 号商品的生效时间段，从 from[i]持续到 to[i]
public static int[] from = new int[MAXN];
public static int[] to = new int[MAXN];

// 链式前向星存储线段树节点上的商品
public static int[] head = new int[MAXQ << 2]; // 线段树节点头指针
public static int[] next = new int[MAXT];      // 下一条边
public static int[] tov = new int[MAXT];       // 边对应的商品价值
public static int[] tow = new int[MAXT];       // 边对应的商品重量
public static int cnt = 0;                     // 边计数器

// DP 相关数组
public static long[] dp = new long[MAXK];      // 动态规划表，dp[j]表示重量<=j 时的最大价值
public static long[][] backup = new long[DEEP][MAXK]; // 动态规划表的备份，用于回溯

// 答案数组，存储每次查询操作的结果
public static long[] ans = new long[MAXQ];

public static void clone(long[] a, long[] b) {
    for (int i = 0; i <= k; i++) {
        a[i] = b[i];
    }
}

public static void addEdge(int i, int v, int w) {
    next[++cnt] = head[i];
    tov[cnt] = v;
    tow[cnt] = w;
    head[i] = cnt;
}

public static void add(int jobl, int jobr, int jobv, int jobw, int l, int r, int i) {
    /**
```

```
     * 线段树分治核心方法 - 将商品挂载到对应的线段树节点
     * <p>
     * 算法原理：
     * - 线段树的每个节点代表一个时间区间[l, r]
     * - 如果商品的有效区间[jobl, jobr]完全包含当前节点区间，则将商品挂载到该节点
     * - 否则递归处理左右子节点
     * <p>
     * 核心思想：
     * - 离线处理：预先确定每个商品的有效时间区间
     * - 区间分配：将商品分配到覆盖其有效区间的最小线段树节点集合
     * - 时间合并：相同时间区间的商品会被挂载到同一个线段树节点
     * <p>
     * 实现细节：
     * - 使用链式前向星存储每个节点挂载的商品
     * - 时间复杂度：O(log q) per operation
     * - 空间复杂度：O(log q) per operation
     * <p>
     * @param jobl 商品生效的左时间点（操作序号）
     * @param jobr 商品生效的右时间点（操作序号）
     * @param jobv 商品的价值
     * @param jobw 商品的重量
     * @param l 当前线段树节点表示的区间左边界
     * @param r 当前线段树节点表示的区间右边界
     * @param i 当前线段树节点编号
     */
    if (jobl <= l && r <= jobr) {
        // 当前节点区间被完全包含在目标区间内，直接挂载商品
        // 这是线段树分治的核心优化：一个商品只需要挂载到O(log q)个节点上
        addEdge(i, jobv, jobw);
    } else {
        // 当前节点区间部分覆盖目标区间，需要递归到子节点处理
        int mid = (l + r) >> 1;  // 计算中点，分割区间
        // 如果目标区间与左子区间有交集，递归处理左子树
        if (jobl <= mid) {
            add(jobl, jobr, jobv, jobw, l, mid, i << 1);
        }
        // 如果目标区间与右子区间有交集，递归处理右子树
        if (jobr > mid) {
            add(jobl, jobr, jobv, jobw, mid + 1, r, i << 1 | 1);
        }
    }
}
```

```
public static void dfs(int l, int r, int i, int dep) {
    /**
     * 线段树分治的深度优先遍历 - 背包问题的核心求解过程
     * <p>
     * 算法原理：
     * - 深度优先遍历线段树，处理每个节点对应的时间区间
     * - 在进入节点时，应用该节点挂载的所有商品（执行背包 DP 更新）
     * - 递归处理子节点（对应更小的时间区间）
     * - 回溯时恢复 DP 状态，处理其他分支
     * <p>
     * 核心思想：
     * - 时间旅行：通过备份和恢复 DP 数组，模拟不同时间点的状态
     * - 状态合并：相同时间区间内的商品被同时处理
     * - 回溯优化：使用深度作为索引，复用备份空间
     * <p>
     * 实现步骤：
     * 1. 备份当前 DP 状态（用于回溯）
     * 2. 使用 01 背包算法处理当前节点挂载的所有商品
     * 3. 处理叶子节点（对应单个时间点的查询）
     * 4. 递归处理左右子节点（对应更小的时间区间）
     * 5. 恢复 DP 状态（回溯操作）
     * <p>
     * 时间复杂度分析：
     * - 每个商品被处理 O(log q)次（挂载到 O(log q)个节点）
     * - 每次处理商品需要 O(k)时间进行背包更新
     * - 总时间复杂度：O(nk log q)
     * <p>
     * 空间复杂度分析：
     * - DP 数组：O(k)
     * - 备份数组：O(k * log q)
     * <p>
     * @param l 当前节点表示的区间左边界
     * @param r 当前节点表示的区间右边界
     * @param i 当前线段树节点编号
     * @param dep 当前递归深度（用于备份和恢复 DP 状态）
     */
    // 步骤 1：备份当前 DP 状态，用于回溯时恢复
    // 使用 dep 作为索引选择备份数组，避免重复申请内存
    clone(backup[dep], dp);

    // 步骤 2：处理当前节点上的所有商品（执行 01 背包动态规划）
    // 遍历当前节点挂载的所有商品（链式前向星）
    for (int e = head[i]; e > 0; e = next[e]) {
```

```
            int v = tov[e];  // 商品价值
            int w = tow[e];  // 商品重量
            // 01背包的标准实现：从后往前遍历，避免重复选择同一件商品
            for (int j = k; j >= w; j--) {
                // 状态转移方程：选择或不选择当前商品
                dp[j] = Math.max(dp[j], dp[j - w] + v);
            }
        }

        // 步骤3：处理叶子节点或递归处理子节点
        if (l == r) {
            // 叶子节点，对应单个时间点（操作）
            if (op[l] == 3) {
                // 如果是查询操作，计算结果
                // 结果计算：Σ(m=1~k) (dp[m] * BAS^(m-1)) % MOD
                long ret = 0;
                long base = 1;  // 初始为BAS^0
                for (int j = 1; j <= k; j++) {
                    // 累加每个m对应的dp[m] * BAS^(m-1)
                    ret = (ret + dp[j] * base) % MOD;
                    // 计算下一个幂次：BAS^j = BAS^(j-1) * BAS
                    base = (base * BAS) % MOD;
                }
                ans[l] = ret;  // 保存查询结果
            }
        } else {
            // 非叶子节点，递归处理左右子树
            int mid = (l + r) >> 1;
            // 先处理左子区间 [l, mid]
            dfs(l, mid, i << 1, dep + 1);
            // 再处理右子区间 [mid+1, r]
            dfs(mid + 1, r, i << 1 | 1, dep + 1);
        }

        // 步骤4：回溯操作 - 恢复DP状态
        // 将DP数组恢复到进入当前节点前的状态，以便处理其他分支
        clone(dp, backup[dep]);
    }

    public static void prepare() {
        /**
         * 线段树分治的预处理阶段 - 确定商品的有效时间区间并构建线段树
         * <p>
```

```
 * 算法原理：
 * - 对于每个商品，确定其在操作序列中的有效时间区间[from, to]
 * - 初始商品在所有操作开始前就存在，默认有效区间是[1, q]
 * - 添加的商品从添加操作开始生效，默认有效到所有操作结束
 * - 删除的商品在删除操作时停止生效，更新其 to 为 i-1
 * <p>
 * 实现步骤详解：
 * 1. 初始化初始商品的有效时间区间
 * 2. 遍历所有操作，处理添加和删除操作，动态调整商品的有效区间
 * 3. 将每个有效商品添加到线段树的对应时间区间
 * <p>
 * 时间复杂度分析：
 * - 初始商品初始化：O(n)
 * - 操作处理：O(q)
 * - 线段树构建：O(n log q)
 * - 总体时间复杂度：O(n log q + q)
 * <p>
 * 空间复杂度分析：
 * - from 和 to 数组：O(n)
 * - 线段树存储：O(n log q)
 * <p>
 * 关键点：
 * - 离线处理：所有操作预先读取并处理
 * - 时间映射：将商品的生命周期映射到操作序列中的时间点
 * - 区间有效性检查：确保只有有效区间的商品才会被添加到线段树
 */
// 步骤 1：初始化初始商品的有效时间区间
// 初始商品从第 1 个操作开始生效，默认有效期到最后一个操作
for (int i = 1; i <= n; i++) {
    from[i] = 1;  // 从第 1 个操作开始生效
    to[i] = q;    // 默认为所有操作完成
}

// 步骤 2：处理操作序列，更新商品的有效时间区间
for (int i = 1; i <= q; i++) {
    if (op[i] == 1) {
        // 添加商品操作
        n++;  // 新增商品编号
        v[n] = x[i];  // 设置商品价值
        w[n] = y[i];  // 设置商品重量
        from[n] = i;  // 从当前操作开始生效
        to[n] = q;    // 默认为所有操作完成
    } else if (op[i] == 2) {
```

```
                // 删除商品操作
                // 被删除的商品在当前操作前一个时间点结束生效
                // 这是因为删除操作是在第 i 个操作时执行的，所以商品在 i-1 操作时仍然存在
                to[x[i]] = i - 1;
            }
            // 注意：操作 3（查询）不需要处理时间区间
        }

        // 步骤 3：将每个有效的商品添加到线段树中
        // 遍历所有商品，确保只添加有效时间区间的商品
        for (int i = 1; i <= n; i++) {
            if (from[i] <= to[i]) {  // 确保有效时间区间有效（避免无效或过期商品）
                // 将商品挂载到对应的线段树节点上
                // 调用 add 方法将商品添加到时间区间[from[i], to[i]]对应的线段树节点
                add(from[i], to[i], v[i], w[i], 1, q, 1);
            }
        }
    }

    /**
 * 主函数：程序入口，负责数据输入、预处理、算法执行和结果输出
 *
 * @param args 命令行参数（未使用）
 * @throws Exception 可能出现的异常（主要是输入异常）
 */
public static void main(String[] args) throws Exception {
    // 创建高效输入流，用于快速读取大量输入数据
    FastReader in = new FastReader(System.in);
    // 创建高效输出流，用于批量输出结果
    PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));

    // 读取初始商品数量和最大重量限制
    n = in.nextInt();
    k = in.nextInt();

    // 读取初始商品信息（价值和重量）
    for (int i = 1; i <= n; i++) {
        v[i] = in.nextInt();  // 第 i 个商品的价值
        w[i] = in.nextInt();  // 第 i 个商品的重量
    }

    // 读取操作数量
    q = in.nextInt();
```

```
    // 读取每个操作的详细信息
    for (int i = 1; i <= q; i++) {
        op[i] = in.nextInt();  // 操作类型
        if (op[i] == 1) {
            // 操作 1：添加商品，读取商品的价值和重量
            x[i] = in.nextInt();  // 商品价值
            y[i] = in.nextInt();  // 商品重量
        } else if (op[i] == 2) {
            // 操作 2：删除商品，读取要删除的商品编号
            x[i] = in.nextInt();  // 要删除的商品编号
        }
        // 操作 3（查询）不需要额外参数
    }

    // 预处理：确定每个商品的有效时间区间并构建线段树
    prepare();

    // 执行线段树分治算法的深度优先搜索过程
    // 参数：当前区间[1,q]，当前节点 1，当前深度 1
    dfs(1, q, 1, 1);

    // 输出所有查询操作的结果
    for (int i = 1; i <= q; i++) {
        if (op[i] == 3) {
            out.println(ans[i]);  // 输出查询结果
        }
    }

    // 刷新输出缓冲区并关闭输出流
    out.flush();
    out.close();
}

    /**
 * 高效输入类：用于快速读取大量输入数据，避免标准输入方法的性能瓶颈
 *
 * 在处理大规模数据时，使用自定义的 FastReader 可以显著提升输入速度，
 * 实现了基于缓冲区的读取机制，逐字节读取并转换为整数，优化了 IO 性能。
 */
static class FastReader {
    private final byte[] buffer; // 输入缓冲区
    private int ptr;             // 当前读取位置指针
```

```
private int len;             // 当前缓冲区中有效字节数
private final InputStream in; // 输入流

/**
 * 构造函数：初始化 FastReader
 *
 * @param in 输入流对象
 */
FastReader(InputStream in) {
    this.in = in;               // 保存输入流引用
    this.buffer = new byte[1 << 20]; // 创建 2MB 大小的缓冲区
    this.ptr = 0;               // 初始化指针位置
    this.len = 0;               // 初始化有效字节数
}

/**
 * 读取单个字节
 *
 * @return 读取的字节值，若到达流末尾则返回-1
 * @throws IOException 输入异常
 */
private int readByte() throws IOException {
    // 缓冲区已读完，需要重新填充
    if (ptr >= len) {
        len = in.read(buffer);  // 从输入流读取数据到缓冲区
        ptr = 0;                // 重置指针
        if (len <= 0)           // 流已结束
            return -1;
    }
    return buffer[ptr++];     // 返回当前字节并移动指针
}

/**
 * 读取下一个整数
 *
 * @return 读取的整数
 * @throws IOException 输入异常
 */
int nextInt() throws IOException {
    int c;                    // 当前读取的字符
    // 跳过所有空白字符
    do {
        c = readByte();
```

```
        } while (c <= ' ' && c != -1);

        boolean neg = false;      // 标记是否为负数
        if (c == '-') {           // 检测负号
            neg = true;
            c = readByte();         // 读取负号后的第一个字符
        }

        int val = 0;              // 保存结果
        // 读取并处理所有数字字符
        while (c > ' ' && c != -1) {
            val = val * 10 + (c - '0');  // 逐位构建整数
            c = readByte();              // 读取下一个字符
        }

        return neg ? -val : val;  // 根据符号返回结果
    }
}

}
```

==========

文件: Code01_MuseumRobbery2.java

==========

```
package class167;

/**
 * 博物馆劫案 (Museum Robbery) - C++版本实现
 *
 * 本文件包含博物馆劫案问题的C++实现代码（注释掉的形式）
 *
 * 问题描述：
 * - 初始有n件商品，每件商品有价值v和重量w
 * - 支持三种操作：添加商品、删除商品、查询f(s)
 * - f(s)定义为：Σ(m=1~k) [a(s, m) * BAS^(m-1)] mod MOD，其中a(s, m)表示总重量≤m时的最大价值
 *
 * 算法思路：线段树分治 + 动态规划（01背包）
 *
 * Java与C++版本的主要区别：
 * 1. 数组声明方式不同：C++使用数组变量声明，Java使用数组对象
 * 2. 输入输出方式不同：C++使用cin/cout，Java使用Scanner/System.out或自定义FastReader
 * 3. 内存管理：C++需要手动管理内存，Java有自动垃圾回收
```

```
 * 4. 效率差异：C++在大规模数据处理上通常比 Java 更高效
 * 5. 语法细节：如位运算、变量声明位置等语法差异
 *
 * 时间复杂度：O(n log q + q log q × k)
 * 空间复杂度：O(n log q + q + k log q)
 *
 * 测试链接：
 * - Codeforces: https://codeforces.com/problemset/problem/601/E
 * - 洛谷: https://www.luogu.com.cn/problem/CF601E
 *
 * 使用说明：
 * - 将注释移除后，即可在 C++环境中编译运行
 * - 确保输入输出按照题目要求的格式
 */

//#include <bits/stdc++.h>
//
//using namespace std;
//
//const int MAXN = 40001;
//const int MAXQ = 30001;
//const int MAXK = 1001;
//const int MAXT = 1000001;
//const int DEEP = 20;
//const long long BAS = 10000019LL;
//const long long MOD = 1000000007LL;
//int n, k, q;
//
//int v[MAXN];
//int w[MAXN];
//
//int op[MAXQ];
//int x[MAXQ];
//int y[MAXQ];
//
//int from[MAXN];
//int to[MAXN];
//
//int head[MAXQ << 2];
//int nxt[MAXT];
//int tov[MAXT];
//int tow[MAXT];
//int cnt = 0;
```

```
//
//long long dp[MAXK];
//long long backup[DEEP][MAXK];
//long long ans[MAXQ];
//
//void clone(long long* a, long long* b) {
//    for (int i = 0; i <= k; i++) {
//        a[i] = b[i];
//    }
//}
//
//void addEdge(int i, int v, int w) {
//    nxt[++cnt] = head[i];
//    tov[cnt] = v;
//    tow[cnt] = w;
//    head[i] = cnt;
//}
//
//void add(int jobl, int jobr, int jobv, int jobw, int l, int r, int i) {
//    if (jobl <= l && r <= jobr) {
//        addEdge(i, jobv, jobw);
//    } else {
//        int mid = (l + r) >> 1;
//        if (jobl <= mid) {
//            add(jobl, jobr, jobv, jobw, l, mid, i << 1);
//        }
//        if (jobr > mid) {
//            add(jobl, jobr, jobv, jobw, mid + 1, r, i << 1 | 1);
//        }
//    }
//}
//
//void dfs(int l, int r, int i, int dep) {
//    clone(backup[dep], dp);
//    for (int e = head[i]; e > 0; e = nxt[e]) {
//        int v = tov[e];
//        int w = tow[e];
//        for (int j = k; j >= w; j--) {
//            dp[j] = max(dp[j], dp[j - w] + v);
//        }
//    }
//    if (l == r) {
//        if (op[l] == 3) {
```

```
//            long long ret = 0;
//            long long base = 1;
//            for (int j = 1; j <= k; j++) {
//                ret = (ret + dp[j] * base) % MOD;
//                base = (base * BAS) % MOD;
//            }
//            ans[l] = ret;
//        }
//    } else {
//        int mid = (l + r) >> 1;
//        dfs(l, mid, i << 1, dep + 1);
//        dfs(mid + 1, r, i << 1 | 1, dep + 1);
//    }
//    clone(dp, backup[dep]);
//}
//
//void prepare() {
//    for (int i = 1; i <= n; i++) {
//        from[i] = 1;
//        to[i] = q;
//    }
//    for (int i = 1; i <= q; i++) {
//        if (op[i] == 1) {
//            n++;
//            v[n] = x[i];
//            w[n] = y[i];
//            from[n] = i;
//            to[n] = q;
//        } else if (op[i] == 2) {
//            to[x[i]] = i - 1;
//        }
//    }
//    for (int i = 1; i <= n; i++) {
//        if (from[i] <= to[i]) {
//            add(from[i], to[i], v[i], w[i], 1, q, 1);
//        }
//    }
//}
//
//int main() {
//    ios::sync_with_stdio(false);
//    cin.tie(nullptr);
//    cin >> n >> k;
```

```
//    for (int i = 1; i <= n; i++) {
//        cin >> v[i] >> w[i];
//    }
//    cin >> q;
//    for (int i = 1; i <= q; i++) {
//        cin >> op[i];
//        if (op[i] == 1) {
//            cin >> x[i] >> y[i];
//        } else if (op[i] == 2) {
//            cin >> x[i];
//        }
//    }
//    prepare();
//    dfs(1, q, 1, 1);
//    for (int i = 1; i <= q; i++) {
//        if (op[i] == 3) {
//            cout << ans[i] << '\n';
//        }
//    }
//    return 0;
//}
```

==========

文件: Code02_BlueMoon1.java

==========

```
package class167;

/**
 * 贪玩蓝月问题 (Blue Moon) - Java 实现
 *
 * 问题描述：
 * - 背包是一个双端队列，可以在两端添加或删除装备
 * - 每件装备有特征值 w 和战斗力 v
 * - 支持五种操作：前端添加、后端添加、前端删除、后端删除、查询最大战斗力
 * - 查询操作要求：选择装备的特征值累加和模 p 必须在[x, y]范围内，求最大可能的战斗力
 *
 * 算法思路：线段树分治 + 动态规划（模运算下的背包问题）
 *
 * 核心思想：
 * 1. 使用线段树分治处理装备的存在时间区间
 * 2. 对于每个装备，计算其存在的时间区间[L, R]，并将其挂载到线段树的相应节点
 * 3. 使用深度优先搜索遍历线段树，在每个节点维护动态规划数组 dp[j]表示模 p 余 j 时的最大战斗力
```

```
 * 4. 回溯时恢复动态规划数组的状态，实现撤销操作
 *
 * 动态规划状态转移：
 * - dp[j] 表示特征值总和模 p 等于 j 时的最大战斗力
 * - 转移方程：dp[(j + w) % p] = max(dp[(j + w) % p], dp[j] + v)
 *
 * 数据规模限制：
 * - 操作数量 m: 1 <= m <= 5 * 10^4
 * - 模数值 p: 1 <= p <= 500
 * - 装备特征值和战斗力: 0 <= w, v <= 10^9
 *
 * 时间复杂度：O(m log m × p)
 * - 线段树分治：O(m log m)
 * - 动态规划转移：O(p) per operation
 *
 * 空间复杂度：O(m + p log m)
 * - 线段树和动态规划数组：O(m + p log m)
 *
 * 测试链接：https://loj.ac/p/6515
 *
 * 使用说明：
 * - 提交时请将类名修改为"Main"
 * - 确保输入输出按照题目要求的格式
 */

import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStreamWriter;
import java.io.PrintWriter;
import java.util.ArrayDeque;
import java.util.Deque;

public class Code02_BlueMoon1 {

    public static int MAXM = 50001;
    public static int MAXP = 501;
    public static int MAXT = 1000001;
    public static int DEEP = 20;
    public static int m, p;

    public static int[] op = new int[MAXM];
    public static int[] x = new int[MAXM];
    public static int[] y = new int[MAXM];
```

```
// 背包<装备特征值%p、装备战斗力、装备出现时间点>
public static Deque<int[]> knapsack = new ArrayDeque<>();

// 时间轴线段树的区间上挂上生效的装备，(特征值 % p)记为w，战斗力记为v
public static int[] head = new int[MAXM << 2];
public static int[] next = new int[MAXT];
public static int[] tow = new int[MAXT];
public static int[] tov = new int[MAXT];
public static int cnt = 0;

// 动态规划表不考虑当前装备的状态，上一行的状态
public static long[] pre = new long[MAXP];
// 动态规划表考虑当前装备的状态，本行的状态，需要更新
public static long[] dp = new long[MAXP];
// 动态规划表的备份
public static long[][] backup = new long[DEEP][MAXP];

// 答案
public static long[] ans = new long[MAXM];

/**
 * 克隆数组内容，用于备份和恢复动态规划状态
 *
 * @param a 目标数组（被复制到的数组）
 * @param b 源数组（被复制的数组）
 * 时间复杂度：O(p)，其中p是模数值
 */
public static void clone(long[] a, long[] b) {
    for (int i = 0; i < p; i++) {
        a[i] = b[i];
    }
}

/**
 * 向线段树节点添加一条边（装备信息）
 * 使用链式前向星存储线段树节点上的装备
 *
 * @param i 线段树节点编号
 * @param w 装备的特征值（已模p处理）
 * @param v 装备的战斗力
 */
public static void addEdge(int i, int w, int v) {
```

```
    next[++cnt] = head[i]; // 新边的 next 指针指向前一条边
    tow[cnt] = w;          // 存储装备特征值
    tov[cnt] = v;          // 存储装备战斗力
    head[i] = cnt;         // 更新头指针
}

/**
 * 线段树分治的核心方法：将装备挂载到对应的线段树节点上
 *
 * @param jobl 装备生效的起始时间
 * @param jobr 装备生效的结束时间
 * @param jobw 装备的特征值（已模 p 处理）
 * @param jobv 装备的战斗力
 * @param l 当前线段树节点表示的时间区间左端点
 * @param r 当前线段树节点表示的时间区间右端点
 * @param i 当前线段树节点编号
 * 时间复杂度：O(log m)，其中 m 是操作数量
 */
public static void add(int jobl, int jobr, int jobw, int jobv, int l, int r, int i) {
    // 如果当前节点区间完全包含在装备的有效时间区间内
    if (jobl <= l && r <= jobr) {
        // 将装备直接挂载到当前节点
        addEdge(i, jobw, jobv);
    } else {
        // 否则递归到左右子节点
        int mid = (l + r) >> 1;
        if (jobl <= mid) {
            add(jobl, jobr, jobw, jobv, l, mid, i << 1); // 左子节点
        }
        if (jobr > mid) {
            add(jobl, jobr, jobw, jobv, mid + 1, r, i << 1 | 1); // 右子节点
        }
    }
}

/**
 * 深度优先搜索遍历线段树，处理每个节点上的装备并回答查询
 *
 * @param l 当前线段树节点表示的时间区间左端点
 * @param r 当前线段树节点表示的时间区间右端点
 * @param i 当前线段树节点编号
 * @param dep 当前递归深度（用于状态备份）
 * 时间复杂度：O(p log m)，其中 p 是模数值，m 是操作数量
```

```
 */
public static void dfs(int l, int r, int i, int dep) {
    // 备份当前 dp 状态，用于回溯
    clone(backup[dep], dp);

    // 处理当前节点上挂载的所有装备
    for (int e = head[i], w, v; e > 0; e = next[e]) {
        w = tow[e]; // 装备特征值
        v = tov[e]; // 装备战斗力

        // 复制当前 dp 数组到 pre 数组，避免覆盖影响后续计算
        clone(pre, dp);

        // 动态规划状态转移
        for (int j = 0; j < p; j++) {
            if (pre[j] != -1) { // 只有可达状态才进行转移
                // 计算新的余数，并更新最大值
                dp[(j + w) % p] = Math.max(dp[(j + w) % p], pre[j] + v);
            }
        }
    }

    // 如果是叶子节点（对应单个操作时间点）
    if (l == r) {
        // 如果是查询操作
        if (op[l] == 5) {
            long ret = -1;
            // 在指定的余数范围内寻找最大战斗力
            for (int j = x[l]; j <= y[l]; j++) {
                ret = Math.max(ret, dp[j]);
            }
            ans[l] = ret; // 保存查询结果
        }
    } else {
        // 非叶子节点，递归处理左右子树
        int mid = (l + r) >> 1;
        dfs(l, mid, i << 1, dep + 1);     // 处理左子树
        dfs(mid + 1, r, i << 1 | 1, dep + 1); // 处理右子树
    }

    // 回溯：恢复 dp 数组状态
    clone(dp, backup[dep]);
}
```

```
/**
 * 预处理方法：处理所有操作，计算每个装备的存在时间区间，并初始化动态规划数组
 *
 * 使用双端队列模拟背包操作，记录每个装备的生效时间和失效时间
 * 时间复杂度：O(m)
 */
public static void prepare() {
    int[] equip;
    // 遍历所有操作，模拟背包的添加和删除操作
    for (int i = 1; i <= m; i++) {
        if (op[i] == 1) {
            // 前端添加装备，记录特征值（模 p）、战斗力和生效时间
            knapsack.addFirst(new int[] { x[i] % p, y[i], i });
        } else if (op[i] == 2) {
            // 后端添加装备，记录特征值（模 p）、战斗力和生效时间
            knapsack.addLast(new int[] { x[i] % p, y[i], i });
        } else if (op[i] == 3) {
            // 前端删除装备，计算其存在时间区间[生效时间, 失效时间-1]
            equip = knapsack.pollFirst();
            add(equip[2], i - 1, equip[0], equip[1], 1, m, 1);
        } else if (op[i] == 4) {
            // 后端删除装备，计算其存在时间区间[生效时间, 失效时间-1]
            equip = knapsack.pollLast();
            add(equip[2], i - 1, equip[0], equip[1], 1, m, 1);
        }
    }
    // 处理操作结束后仍在背包中的装备，它们的存在时间区间到 m 为止
    while (!knapsack.isEmpty()) {
        equip = knapsack.pollFirst();
        add(equip[2], m, equip[0], equip[1], 1, m, 1);
    }
    // 初始化动态规划数组：-1 表示不可达状态，只有 dp[0]=0 是可达的（不选任何装备）
    for (int i = 0; i < p; i++) {
        dp[i] = -1;
    }
    dp[0] = 0; // 初始状态：不选任何装备时，特征值和为 0，战斗力为 0
}

/**
 * 主函数：程序入口，负责数据输入、预处理、算法执行和结果输出
 *
 * @param args 命令行参数（未使用）
```

```
     * @throws IOException 可能出现的输入输出异常
     */
    public static void main(String[] args) throws IOException {
        // 创建高效输入流，用于快速读取大量输入数据
        FastReader in = new FastReader();
        // 创建高效输出流，用于批量输出结果
        PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));

        // 读取第一个数（未使用，可能是题目输入格式的冗余）
        in.nextInt();
        // 读取操作数量 m 和模数值 p
        m = in.nextInt();
        p = in.nextInt();

        // 读取每个操作
        String t;
        for (int i = 1; i <= m; i++) {
            t = in.nextString();
            // 根据操作类型进行不同处理
            if (t.equals("IF")) {
                op[i] = 1; // 前端添加装备
                x[i] = in.nextInt(); // 特征值
                y[i] = in.nextInt(); // 战斗力
            } else if (t.equals("IG")) {
                op[i] = 2; // 后端添加装备
                x[i] = in.nextInt(); // 特征值
                y[i] = in.nextInt(); // 战斗力
            } else if (t.equals("DF")) {
                op[i] = 3; // 前端删除装备
            } else if (t.equals("DG")) {
                op[i] = 4; // 后端删除装备
            } else {
                op[i] = 5; // 查询操作
                x[i] = in.nextInt(); // 查询范围左边界
                y[i] = in.nextInt(); // 查询范围右边界
            }
        }

        // 预处理：计算装备的时间区间并初始化线段树
        prepare();
        // 深度优先搜索遍历线段树，处理查询
        dfs(1, m, 1, 1);
```

```
        // 输出所有查询操作的结果
        for (int i = 1; i <= m; i++) {
            if (op[i] == 5) {
                out.println(ans[i]);
            }
        }

        // 刷新输出缓冲区并关闭输出流
        out.flush();
        out.close();
    }

    /**
     * 高效输入工具类，使用缓冲区优化大规模数据的输入读取
     * 比 Scanner 快约 10 倍，适用于处理大数据量输入的竞赛题目
     */
    static class FastReader {
        private static final int BUFFER_SIZE = 1 << 16; // 64KB 缓冲区
        private final InputStream in;      // 输入流
        private final byte[] buffer;       // 字节缓冲区
        private int ptr, len;              // 指针位置和缓冲区有效长度

        /**
         * 构造函数：初始化输入流和缓冲区
         */
        public FastReader() {
            in = System.in;
            buffer = new byte[BUFFER_SIZE];
            ptr = len = 0;
        }

        /**
         * 检查是否还有下一个字节可读
         * 如果缓冲区已读完，尝试从输入流读取新的内容
         *
         * @return 是否还有可用字节
         * @throws IOException 输入异常
         */
        private boolean hasNextByte() throws IOException {
            if (ptr < len) {
                return true;
            }
            ptr = 0; // 重置指针
```

```
    len = in.read(buffer); // 从输入流读取新内容到缓冲区
    return len > 0;
}

/**
 * 读取单个字节
 *
 * @return 读取的字节值
 * @throws IOException 输入异常
 */
private byte readByte() throws IOException {
    if (!hasNextByte()) {
        return -1; // 到达流末尾
    }
    return buffer[ptr++]; // 返回当前字节并移动指针
}

/**
 * 读取下一个整数
 *
 * @return 读取的整数值
 * @throws IOException 输入异常
 */
public int nextInt() throws IOException {
    int num = 0;
    byte b = readByte();
    // 跳过空白字符
    while (isWhitespace(b)) {
        b = readByte();
    }
    // 处理负数符号
    boolean minus = false;
    if (b == '-') {
        minus = true;
        b = readByte();
    }
    // 读取数字部分
    while (!isWhitespace(b) && b != -1) {
        num = num * 10 + (b - '0'); // 逐位构建整数
        b = readByte();
    }
    return minus ? -num : num; // 返回带符号的整数值
}
```

```
        /**
         * 读取下一个字符串
         *
         * @return 读取的字符串
         * @throws IOException 输入异常
         */
        public String nextString() throws IOException {
            byte b = readByte();
            // 跳过空白字符
            while (isWhitespace(b)) {
                b = readByte();
            }
            // 使用 StringBuilder 高效构建字符串
            StringBuilder sb = new StringBuilder(1000);
            while (!isWhitespace(b) && b != -1) {
                sb.append((char) b);
                b = readByte();
            }
            return sb.toString();
        }

        /**
         * 判断字节是否为空白字符（空格、换行、回车、制表符）
         *
         * @param b 要检查的字节
         * @return 是否为空白字符
         */
        private boolean isWhitespace(byte b) {
            return b == ' ' || b == '\n' || b == '\r' || b == '\t';
        }
    }

}
```

==========

文件: Code02_BlueMoon2.java

==========

```
package class167;

// 贪玩蓝月，C++版
// 每件装备都有特征值 w 和战斗力 v，放装备的背包是一个双端队列，只有背包中的装备是可选的
```

```
// 给定数值 p，接下来有 m 条操作，每种操作是如下五种类型中的一种
// 操作 IF x y : 背包前端加入一件特征值 x、战斗力 y 的装备
// 操作 IG x y : 背包后端加入一件特征值 x、战斗力 y 的装备
// 操作 DF     : 删除背包前端的装备
// 操作 DG     : 删除背包后端的装备
// 操作 QU x y : 选择装备的特征值累加和 % p，必须在[x, y]范围，打印最大战斗力，无方案打印-1
// 1 <= m <= 5 * 10^4    1 <= p <= 500
// 0 <= 每件装备特征值、每件装备战斗力 <= 10^9
// 测试链接 : https://loj.ac/p/6515
// 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 = 50001;
//const int MAXP = 501;
//const int MAXT = 1000001;
//const int DEEP = 20;
//int m, p;
//
//int op[MAXM];
//int x[MAXM];
//int y[MAXM];
//
//deque<array<int,3>> knapsack;
//
//int head[MAXM << 2];
//int nxt[MAXT];
//int tow[MAXT];
//int tov[MAXT];
//int cnt = 0;
//
//long long pre[MAXP];
//long long dp[MAXP];
//long long backup[DEEP][MAXP];
//
//long long ans[MAXM];
//
//void clone(long long* a, long long* b) {
//    for (int i = 0; i <= p; i++) {
//        a[i] = b[i];
```

```
//    }
//}
//
//void addEdge(int i, int w, int v) {
//    nxt[++cnt] = head[i];
//    tow[cnt] = w;
//    tov[cnt] = v;
//    head[i] = cnt;
//}
//
//void add(int jobl, int jobr, int jobw, int jobv, int l, int r, int i) {
//    if (jobl <= l && r <= jobr) {
//        addEdge(i, jobw, jobv);
//    } else {
//        int mid = (l + r) >> 1;
//        if (jobl <= mid) {
//            add(jobl, jobr, jobw, jobv, l, mid, i << 1);
//        }
//        if (jobr > mid) {
//            add(jobl, jobr, jobw, jobv, mid + 1, r, i << 1 | 1);
//        }
//    }
//}
//
//void dfs(int l, int r, int i, int dep) {
//    clone(backup[dep], dp);
//    for (int e = head[i], w, v; e > 0; e = nxt[e]) {
//        w = tow[e];
//        v = tov[e];
//        clone(pre, dp);
//        for (int j = 0; j < p; j++) {
//            if (pre[j] != -1) {
//                dp[(j + w) % p] = max(dp[(j + w) % p], pre[j] + v);
//            }
//        }
//    }
//    if (l == r) {
//        if (op[l] == 5) {
//            long long ret = -1;
//            for (int j = x[l]; j <= y[l]; j++) {
//                ret = max(ret, dp[j]);
//            }
//            ans[l] = ret;
```

```
//        }
//    } else {
//        int mid = (l + r) >> 1;
//        dfs(l, mid, i << 1, dep + 1);
//        dfs(mid + 1, r, i << 1 | 1, dep + 1);
//    }
//    clone(dp, backup[dep]);
//}
//
//void prepare() {
//    array<int,3> equip;
//    for (int i = 1; i <= m; i++) {
//        if (op[i] == 1) {
//            knapsack.push_front({x[i] % p, y[i], i});
//        } else if (op[i] == 2) {
//            knapsack.push_back({x[i] % p, y[i], i});
//        } else if (op[i] == 3) {
//            equip = knapsack.front();
//            add(equip[2], i - 1, equip[0], equip[1], 1, m, 1);
//            knapsack.pop_front();
//        } else if (op[i] == 4) {
//            equip = knapsack.back();
//            add(equip[2], i - 1, equip[0], equip[1], 1, m, 1);
//            knapsack.pop_back();
//        }
//    }
//    while (!knapsack.empty()) {
//        equip = knapsack.front();
//        add(equip[2], m, equip[0], equip[1], 1, m, 1);
//        knapsack.pop_front();
//    }
//    for (int i = 0; i < p; i++) {
//        dp[i] = -1;
//    }
//    dp[0] = 0;
//}
//
//int main() {
//    ios::sync_with_stdio(false);
//    cin.tie(nullptr);
//    int tmp;
//    cin >> tmp;
//    cin >> m >> p;
```

```
//    string t;
//    for (int i = 1; i <= m; i++) {
//        cin >> t;
//        if (t == "IF") {
//            op[i] = 1;
//            cin >> x[i] >> y[i];
//        } else if (t == "IG") {
//            op[i] = 2;
//            cin >> x[i] >> y[i];
//        } else if (t == "DF") {
//            op[i] = 3;
//        } else if (t == "DG") {
//            op[i] = 4;
//        } else {
//            op[i] = 5;
//            cin >> x[i] >> y[i];
//        }
//    }
//    prepare();
//    dfs(1, m, 1, 1);
//    for (int i = 1; i <= m; i++) {
//        if (op[i] == 5) {
//            cout << ans[i] << '\n';
//        }
//    }
//    return 0;
//}
```

==========

文件: Code03_AdditionOnSegments1.java

==========

```
package class167;

/**
 * 打印所有合法数 (Addition On Segments) - Java 实现
 *
 * 问题描述：
 * - 初始有一个长度为 n 的序列，所有值都是 0
 * - 有 q 条操作，每条操作为[l, r, k]：将序列区间[l..r]的每个数字加 k
 * - 每条操作可以选择执行或不执行，每条操作最多执行一次
 * - 如果能让序列中的最大值正好为 v，那么 v 就是一个合法数
 * - 要求找出 1~n 范围内的所有合法数，并按升序输出
```

```
 *
 * 算法思路：线段树分治 + 位运算动态规划
 *
 * 核心思想：
 * 1. 将每个操作的区间[l, r]映射到线段树的节点上
 * 2. 使用位集(bitset)表示可达的值，其中 dp[i]=1 表示值 i 是可达的
 * 3. 对于每个区间操作，相当于对当前位集进行左移 k 位再或运算（表示可以选择执行该操作）
 * 4. 通过深度优先搜索遍历线段树，在回溯时维护位集的状态
 *
 * 位运算优化：
 * - 使用整数数组模拟 bitset，提高位操作效率
 * - 采用块级左移而非逐位左移，大大提高效率
 * - 使用克隆和备份数组实现状态恢复
 *
 * 数据规模限制：
 * - n: 序列长度
 * - q: 操作数量
 * - 1 <= k <= n, q <= 10^4
 *
 * 时间复杂度：O(q log n + n^2 / w)，其中 w 是机器字长（这里取 32）
 * - 线段树分治：O(q log n)
 * - 位运算操作：O(n / w) per operation
 *
 * 空间复杂度：O(n log n + n / w)
 * - 线段树和位集数组：O(n log n + n / w)
 *
 * 测试链接：
 * - Codeforces: https://codeforces.com/problemset/problem/981/E
 * - 洛谷: https://www.luogu.com.cn/problem/CF981E
 *
 * 使用说明：
 * - 提交时请将类名修改为"Main"
 * - 确保输入输出按照题目要求的格式
 */

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

public class Code03_AdditionOnSegments1 {

    public static int MAXN = 10001;
```

```
public static int MAXT = 500001;
public static int BIT = 10000;
public static int DEEP = 20;
public static int INT_BIT = 32;
public static int LEN = BIT / INT_BIT + 1;
public static int n, q;

public static int[] head = new int[MAXN << 2];
public static int[] next = new int[MAXT];
public static int[] to = new int[MAXT];
public static int cnt = 0;

public static int[] tmp = new int[LEN];
public static int[] dp = new int[LEN];
public static int[][] backup = new int[DEEP][LEN];
public static int[] ans = new int[LEN];

/**
 * 清空位集（将所有位设置为0）
 *
 * @param bitset 要清空的位集数组
 */
public static void clear(int[] bitset) {
    for (int i = 0; i < LEN; i++) {
        bitset[i] = 0;
    }
}

/**
 * 克隆位集
 *
 * @param set1 目标位集（被复制到的数组）
 * @param set2 源位集（被复制的数组）
 */
public static void clone(int[] set1, int[] set2) {
    for (int i = 0; i < LEN; i++) {
        set1[i] = set2[i];
    }
}

/**
 * 获取位集指定位置的状态
 *
```

```
     * @param bitset 位集数组
     * @param i 要查询的位位置
     * @return 该位的状态（0 或 1）
     */
    public static int getBit(int[] bitset, int i) {
        // 计算所在整数位置和偏移量，然后通过右移和与运算获取该位值
        return (bitset[i / INT_BIT] >> (i % INT_BIT)) & 1;
    }

    /**
     * 设置位集指定位置的状态
     *
     * @param bitset 位集数组
     * @param i 要设置的位位置
     * @param v 要设置的值（0 或 1）
     */
    public static void setBit(int[] bitset, int i, int v) {
        if (v == 0) {
            // 如果设置为 0，使用与运算和取反操作清除该位
            bitset[i / INT_BIT] &= ~(1 << (i % INT_BIT));
        } else {
            // 如果设置为 1，使用或运算设置该位
            bitset[i / INT_BIT] |= 1 << (i % INT_BIT);
        }
    }

    /**
     * 执行位集的或运算
     *
     * @param set1 第一个位集（结果将存储在这里）
     * @param set2 第二个位集
     */
    public static void bitOr(int[] set1, int[] set2) {
        for (int i = 0; i < LEN; i++) {
            set1[i] |= set2[i]; // 逐整数执行或运算
        }
    }

    /**
     * 高效实现位集左移操作
     * 不使用逐位左移，而是采用整块左移的方式提高效率
     *
     * @param ret 结果位集
```

```
 * @param bitset 源位集
 * @param move 左移的位数
 */
public static void bitLeft(int[] ret, int[] bitset, int move) {
    clear(ret); // 先清空结果数组

    // 特殊情况处理
    if (move > BIT) { // 左移超过最大位数，结果全 0
        return;
    }
    if (move <= 0) { // 左移位数≤0，直接返回原位集
        clone(ret, bitset);
        return;
    }

    // 计算整数块移动次数和位偏移量
    int shift = move / INT_BIT;    // 需要移动的整数块数
    int offset = move % INT_BIT;   // 每块内部需要移动的位数

    if (offset == 0) {
        // 正好移动整数块的倍数
        for (int i = LEN - 1, j = i - shift; j >= 0; i--, j--) {
            ret[i] = bitset[j];
        }
    } else {
        // 非整数倍移动，需要处理进位
        int carry = INT_BIT - offset; // 进位位数
        int high, low;
        // 处理中间块，需要同时考虑当前块的高位移和前一块的低位移
        for (int i = LEN - 1; i > shift; i--) {
            high = bitset[i - shift] << offset;       // 当前块左移 offset 位
            low = bitset[i - shift - 1] >>> carry;    // 前一块无符号右移 carry 位
            ret[i] = high | low;                      // 合并高位和低位
        }
        // 处理第一个块（没有前一块）
        ret[shift] = bitset[0] << offset;
    }

    // 清除超出范围的高位
    // 我们只关心 0 到 BIT 位，共 BIT+1 个有效位
    int rest = LEN * INT_BIT - (BIT + 1);
    if (rest > 0) {
        // 计算掩码，清除无效的高位
```

```
        ret[LEN - 1] &= (1 << (INT_BIT - rest)) - 1;
    }
}

/**
 * 向线段树节点添加一条边（操作 k 值）
 * 使用链式前向星存储线段树节点上的操作
 *
 * @param i 线段树节点编号
 * @param v 操作的 k 值（要加的数值）
 */
public static void addEdge(int i, int v) {
    next[++cnt] = head[i]; // 新边的 next 指针指向前一条边
    to[cnt] = v;           // 存储操作的 k 值
    head[i] = cnt;         // 更新头指针
}

/**
 * 线段树分治的核心方法：将操作挂载到对应的线段树节点上
 *
 * @param jobl 操作的区间左端点
 * @param jobr 操作的区间右端点
 * @param jobv 操作的 k 值
 * @param l 当前线段树节点表示的区间左端点
 * @param r 当前线段树节点表示的区间右端点
 * @param i 当前线段树节点编号
 */
public static void add(int jobl, int jobr, int jobv, int l, int r, int i) {
    // 如果当前节点区间完全包含在操作区间内
    if (jobl <= l && r <= jobr) {
        // 将操作挂载到当前节点
        addEdge(i, jobv);
    } else {
        // 否则递归到左右子节点
        int mid = (l + r) >> 1;
        if (jobl <= mid) {
            add(jobl, jobr, jobv, l, mid, i << 1); // 左子节点
        }
        if (jobr > mid) {
            add(jobl, jobr, jobv, mid + 1, r, i << 1 | 1); // 右子节点
        }
    }
}
```

```
/**
 * 深度优先搜索遍历线段树，执行位运算动态规划
 *
 * @param l 当前线段树节点表示的区间左端点
 * @param r 当前线段树节点表示的区间右端点
 * @param i 当前线段树节点编号
 * @param dep 当前递归深度（用于状态备份）
 */
public static void dfs(int l, int r, int i, int dep) {
    // 备份当前 dp 状态，用于回溯
    clone(backup[dep], dp);

    // 处理当前节点上的所有操作
    for (int e = head[i]; e > 0; e = next[e]) {
        // 对当前 dp 状态左移 k 位，相当于选择执行该操作
        bitLeft(tmp, dp, to[e]);
        // 执行或运算，将选择或不选择该操作的结果合并
        bitOr(dp, tmp);
    }

    // 如果是叶子节点（对应单个位置）
    if (l == r) {
        // 将当前位置的可达值合并到最终结果中
        bitOr(ans, dp);
    } else {
        // 非叶子节点，递归处理左右子树
        int mid = (l + r) >> 1;
        dfs(l, mid, i << 1, dep + 1);     // 处理左子树
        dfs(mid + 1, r, i << 1 | 1, dep + 1); // 处理右子树
    }

    // 回溯：恢复 dp 数组状态
    clone(dp, backup[dep]);
}

/**
 * 主函数：程序入口，负责数据输入、预处理、算法执行和结果输出
 *
 * @param args 命令行参数（未使用）
 * @throws Exception 可能出现的异常
 */
public static void main(String[] args) throws Exception {
```

```
// 创建高效输入流，用于快速读取大量输入数据
FastReader in = new FastReader(System.in);
// 创建高效输出流，用于批量输出结果
PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));

// 读取序列长度 n 和操作数量 q
n = in.nextInt();
q = in.nextInt();

// 读取每个操作并将其添加到线段树中
for (int i = 1, l, r, k; i <= q; i++) {
    l = in.nextInt(); // 操作区间左端点
    r = in.nextInt(); // 操作区间右端点
    k = in.nextInt(); // 增加的数值 k
    // 将操作挂载到线段树的对应节点
    add(l, r, k, 1, n, 1);
}

// 初始化 dp 数组：dp[0]=1 表示值 0 是初始可达的（不执行任何操作）
setBit(dp, 0, 1);

// 深度优先搜索遍历线段树，执行动态规划
dfs(1, n, 1, 1);

// 统计并输出结果
int ansCnt = 0;
for (int i = 1; i <= n; i++) {
    if (getBit(ans, i) == 1) {
        ansCnt++;
    }
}
// 输出合法数的数量
out.println(ansCnt);
// 输出所有合法数，按升序排列
for (int i = 1; i <= n; i++) {
    if (getBit(ans, i) == 1) {
        out.print(i + " ");
    }
}
out.println();

// 刷新输出缓冲区并关闭输出流
out.flush();
```

```
    out.close();
}

/**
 * 高效输入工具类，使用缓冲区优化大规模数据的输入读取
 * 比 Scanner 快约 10 倍，适用于处理大数据量输入的竞赛题目
 */
static class FastReader {
    private final byte[] buffer = new byte[1 << 20]; // 1MB 缓冲区
    private int ptr = 0, len = 0; // 指针位置和缓冲区有效长度
    private final InputStream in; // 输入流

    /**
     * 构造函数：初始化输入流
     *
     * @param in 输入流
     */
    FastReader(InputStream in) {
        this.in = in;
    }

    /**
     * 读取单个字节
     * 如果缓冲区已读完，尝试从输入流读取新的内容
     *
     * @return 读取的字节值，-1 表示到达流末尾
     * @throws IOException 输入异常
     */
    private int readByte() throws IOException {
        if (ptr >= len) {
            len = in.read(buffer); // 从输入流读取新内容到缓冲区
            ptr = 0;
            if (len <= 0)
                return -1; // 到达流末尾
        }
        return buffer[ptr++]; // 返回当前字节并移动指针
    }

    /**
     * 读取下一个整数
     *
     * @return 读取的整数值
     * @throws IOException 输入异常
```

```
         */
        int nextInt() throws IOException {
            int c;
            // 跳过空白字符
            do {
                c = readByte();
            } while (c <= ' ' && c != -1);

            // 处理负数符号
            boolean neg = false;
            if (c == '-') {
                neg = true;
                c = readByte();
            }

            // 读取数字部分
            int val = 0;
            while (c > ' ' && c != -1) {
                val = val * 10 + (c - '0'); // 逐位构建整数
                c = readByte();
            }

            return neg ? -val : val; // 返回带符号的整数值
        }
    }

}
```

==========

文件: Code03_AdditionOnSegments2.java

==========

```
package class167;

// 打印所有合法数，C++版
// 一个长度为 n 的序列，一开始所有值都是 0
// 一共有 q 条操作，每条操作为 l r k : 序列[l..r]范围上，每个数字加 k
// 你可以随意选择操作来执行，但是每条操作只能执行一次
// 如果你能让序列中的最大值正好为 v，那么 v 就算一个合法数
// 打印 1~n 范围内有多少合法数，并且从小到大打印所有的合法数
// 1 <= k <= n、q <= 10^4
// 测试链接 : https://www.luogu.com.cn/problem/CF981E
// 测试链接 : https://codeforces.com/problemset/problem/981/E
```

```
// 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 = 10001;
//const int MAXT = 500001;
//const int BIT = 10000;
//const int DEEP = 20;
//
//typedef bitset<BIT + 1> bs;
//
//int n, q;
//int head[MAXN << 2];
//int nxt[MAXT];
//int to[MAXT];
//int cnt = 0;
//
//bs dp;
//bs backup[DEEP];
//bs ans;
//
//void addEdge(int i, int v) {
//    nxt[++cnt] = head[i];
//    to[cnt] = v;
//    head[i] = cnt;
//}
//
//void add(int jobl, int jobr, int jobv, int l, int r, int i) {
//    if (jobl <= l && r <= jobr) {
//        addEdge(i, jobv);
//    } else {
//        int mid = (l + r) >> 1;
//        if (jobl <= mid) {
//            add(jobl, jobr, jobv, l, mid, i << 1);
//        }
//        if (jobr > mid) {
//            add(jobl, jobr, jobv, mid + 1, r, i << 1 | 1);
//        }
//    }
//}
```

```
//
//void dfs(int l, int r, int i, int dep) {
//    backup[dep] = dp;
//    for (int e = head[i]; e > 0; e = nxt[e]) {
//        dp |= dp << to[e];
//    }
//    if (l == r) {
//        ans |= dp;
//    } else {
//        int mid = (l + r) >> 1;
//        dfs(l, mid, i << 1, dep + 1);
//        dfs(mid + 1, r, i << 1 | 1, dep + 1);
//    }
//    dp = backup[dep];
//}
//
//int main() {
//    ios::sync_with_stdio(false);
//    cin.tie(nullptr);
//    cin >> n >> q;
//    for (int i = 1, l, r, k; i <= q; i++) {
//        cin >> l >> r >> k;
//        add(l, r, k, 1, n, 1);
//    }
//    dp[0] = 1;
//    dfs(1, n, 1, 1);
//    int ansCnt = 0;
//    for (int i = 1; i <= n; i++) {
//        if (ans[i] == 1) {
//            ansCnt++;
//        }
//    }
//    cout << ansCnt << '\n';
//    for (int i = 1; i <= n; i++) {
//        if (ans[i] == 1) {
//            cout << i << ' ';
//        }
//    }
//    cout << '\n';
//    return 0;
//}

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

文件: Code04_ShortestPathQueries1.java
```
============================================
package class167;

/**
 * 线段树分治 + 可撤销线性基 + 带权并查集 解决动态异或最短路问题
 * 题目来源: Codeforces 938G - Shortest Path Queries
 * 题目描述:
 * - 给定 n 个节点，m 条初始边，每条边有边权
 * - 接下来有 q 条操作，操作类型分为：
 *   1. 添加边：操作 1 x y d - 加入点 x 到点 y 权值为 d 的边
 *   2. 删除边：操作 2 x y - 删除点 x 到点 y 的边
 *   3. 查询操作：操作 3 x y - 查询点 x 到点 y 的所有路径中，异或和的最小值
 * - 约束条件：
 *   - x < y
 *   - 任意操作后，图连通、无重边、无自环
 *   - 所有操作均合法
 *   - 1 <= n、m、q <= 2 * 10^5
 * - 测试链接:
 *   - https://www.luogu.com.cn/problem/CF938G
 *   - https://codeforces.com/problemset/problem/938/G
 *
 * 算法核心思想:
 * 1. 使用线段树分治处理动态加边/删边操作
 * 2. 通过带权并查集维护节点间的连通性和异或路径
 * 3. 利用可撤销线性基记录环的异或值，以快速计算异或最小值
 * 4. 通过 DFS 遍历线段树，处理每个时间点的查询
 */

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

/**
 * 动态异或最短路问题的高效解决方案
 * <p>
 * 题目来源：Codeforces 938G / 洛谷对应题目
 * <p>
 * 问题描述：维护一个带权无向图，支持动态添加和删除边，以及查询两个点之间的异或最短路径。
 * <p>
```

```
 * 算法思路：使用线段树分治将动态问题转化为静态问题，结合可撤销线性基和带权并查集处理异或路径查询。
 * <p>
 * 核心思想：
 * 1. 线段树分治：将所有操作离线处理，把每条边的存在时间区间分解到线段树的节点上
 * 2. 可撤销线性基：维护环的异或值，用于查询时的路径优化
 * 3. 带权并查集：维护节点之间的异或路径权值
 * <p>
 * 数据结构说明：
 * - 事件数组：记录所有边的添加和删除时间
 * - 可撤销线性基：支持插入和撤销操作，用于维护异或环
 * - 带权并查集：维护节点连通性和异或路径值
 * - 线段树：使用链式前向星存储每个时间区间的边
 * <p>
 * 时间复杂度分析：
 * - 事件排序：O((m+q)log(m+q))
 * - 线段树分治：O((m+q)log q)
 * - 线性基操作：O(BIT)，其中 BIT=30
 * - 总体时间复杂度：O((m+q)log q * BIT)
 * <p>
 * 空间复杂度分析：
 * - 线段树存储：O((m+q)log q)
 * - 并查集和线性基：O(n + BIT)
 * - 总体空间复杂度：O((m+q)log q)
 *
 * @author algorithm-journey
 * @date 2023-11-10
 * @see <a href="https://www.luogu.com.cn/problem/CF938G">洛谷链接</a>
 * @see <a href="https://codeforces.com/problemset/problem/938/G">Codeforces 链接</a>
 */
public class Code04_ShortestPathQueries1 {

    // 常量定义
    public static int MAXN = 200001;   // 最大节点数
    public static int MAXT = 5000001;  // 最大线段树任务数
    public static int BIT = 29;        // 二进制位数（0~29 共 30 位，因为权值最大为 1e9）
    public static int n, m, q;         // 节点数、边数、查询数

    // 事件数组：记录所有边的添加和删除事件
    // event[i][0]: 边的左端点 x
    // event[i][1]: 边的右端点 y
    // event[i][2]: 事件发生的时间点 t
    // event[i][3]: 边的权值 w
```

```
public static int[][] event = new int[MAXN << 1][4];
public static int eventCnt;  // 事件计数器

// 记录每个时间点的操作信息
public static int[] op = new int[MAXN];  // 操作类型：1(添加边)、2(删除边)、3(查询)
public static int[] x = new int[MAXN];   // 操作涉及的第一个节点
public static int[] y = new int[MAXN];   // 操作涉及的第二个节点
public static int[] d = new int[MAXN];   // 操作 1 的边权值

// 可撤销线性基：用于计算异或最小值
public static int[] basis = new int[BIT + 1];    // 线性基数组
public static int[] inspos = new int[BIT + 1];   // 记录插入顺序的位置
public static int basiz = 0;                     // 线性基当前大小

// 带权并查集 + 可撤销并查集：维护连通性和异或路径
public static int[] father = new int[MAXN];     // 父节点数组
public static int[] siz = new int[MAXN];        // 集合大小数组
public static int[] eor = new int[MAXN];        // 异或路径数组（从节点到父节点的异或和）
public static int[][] rollback = new int[MAXN][2]; // 回滚栈，记录合并操作
public static int opsize = 0;                   // 操作计数

// 时间轴线段树上的区间任务列表：链式前向星结构
public static int[] head = new int[MAXN << 2];  // 线段树节点的头指针
public static int[] next = new int[MAXT];       // 下一个任务的指针
public static int[] tox = new int[MAXT];        // 任务边的起点
public static int[] toy = new int[MAXT];        // 任务边的终点
public static int[] tow = new int[MAXT];        // 任务边的权值
public static int cnt = 0;                      // 任务计数

// 存储查询操作的答案
public static int[] ans = new int[MAXN];

/**
 * 将一个数插入线性基
 * @param num 要插入的数值
 * @note 线性基用于维护环的异或值，支持撤销操作
 */
public static void insert(int num) {
    // 从高位到低位遍历
    for (int i = BIT; i >= 0; i--) {
        // 检查当前位是否为 1
        if ((num >> i & 1) == 1) {
            // 如果当前位没有基向量，则插入
```

```
            if (basis[i] == 0) {
                basis[i] = num;
                // 记录插入位置，用于撤销操作
                inspos[basiz++] = i;
                return;
            }
            // 否则异或基向量，继续处理低位
            num ^= basis[i];
        }
    }
    // 如果 num 变为 0，说明可以被当前线性基表示，不插入
}

/**
 * 计算 num 与线性基中元素异或后能得到的最小值
 * @param num 初始异或值（两点间的异或路径和）
 * @return 最小异或值
 * @note 贪心策略：从高位到低位，如果异或后的值更小，则选择异或
 */
public static int minEor(int num) {
    // 从高位到低位尝试异或，选择更小的结果
    for (int i = BIT; i >= 0; i--) {
        num = Math.min(num, num ^ basis[i]);
    }
    return num;
}

/**
 * 撤销线性基的操作，恢复到之前的状态
 * @param oldsiz 要恢复到的线性基大小
 */
public static void cancel(int oldsiz) {
    // 将超出的部分重置为 0
    while (basiz > oldsiz) {
        basis[inspos[--basiz]] = 0;
    }
}

/**
 * 并查集的 find 操作：查找集合代表元素
 * @param i 要查找的节点
 * @return 节点所在集合的代表元素（根节点）
 * @note 注意：此实现没有路径压缩，以支持撤销操作
```

```
     */
    public static int find(int i) {
        // 非路径压缩版本，以支持撤销操作
        while (i != father[i]) {
            i = father[i];
        }
        return i;
    }

    /**
     * 计算节点 i 到集合代表点（根节点）的异或路径和
     * @param i 要计算的节点
     * @return 节点 i 到根节点的异或和
     */
    public static int getEor(int i) {
        int ans = 0;
        // 沿父节点链向上，累加异或值
        while (i != father[i]) {
            ans ^= eor[i];
            i = father[i];
        }
        return ans;
    }

    /**
     * 可撤销并查集的合并操作，在节点 u 和 v 之间添加一条权值为 w 的边
     * @param u 第一个节点
     * @param v 第二个节点
     * @param w 边的权值
     * @return 如果合并了两个不同的集合，返回 true；否则返回 false
     * @note 合并时同时维护带权并查集，并记录操作以支持撤销
     */
    public static boolean union(int u, int v, int w) {
        // 查找 u 和 v 的根节点
        int fu = find(u);
        int fv = find(v);

        // 计算 u 到 v 的路径异或和应该为 w
        // 当前路径异或和为 getEor(u) ^ getEor(v)，所以需要异或 w 得到环的异或值
        w = getEor(u) ^ getEor(v) ^ w;

        if (fu == fv) {
            // 如果在同一集合中，将环的异或值插入线性基
```

```
        insert(w);
        return false; // 没有合并新的集合
    }

    // 按秩合并，始终将较小的树合并到较大的树中
    if (siz[fu] < siz[fv]) {
        int tmp = fu;
        fu = fv;
        fv = tmp;
    }

    // 合并操作
    father[fv] = fu;
    siz[fu] += siz[fv];
    // 设置异或路径值
    eor[fv] = w;

    // 记录操作，用于撤销
    rollback[++opsize][0] = fu;
    rollback[opsize][1] = fv;

    return true; // 成功合并两个集合
}

/**
 * 撤销最近的一次合并操作
 * @note 恢复并查集的状态
 */
public static void undo() {
    // 获取最后一次合并操作的信息
    int fx = rollback[opsize][0];  // 父节点
    int fy = rollback[opsize--][1]; // 子节点

    // 恢复 fy 的父节点为自己
    father[fy] = fy;
    // 清除异或路径值
    eor[fy] = 0;
    // 恢复父节点集合的大小
    siz[fx] -= siz[fy];
}

/**
 * 给线段树节点 i 添加一个任务：在节点 x 和 y 之间添加权值为 w 的边
```

```
     * @param i 线段树节点编号
     * @param x 边的起点
     * @param y 边的终点
     * @param w 边的权值
     * @note 使用链式前向星存储任务
     */
    public static void addEdge(int i, int x, int y, int w) {
        // 创建新任务
        cnt++;
        next[cnt] = head[i];  // 指向前一个任务
        tox[cnt] = x;         // 边的起点
        toy[cnt] = y;         // 边的终点
        tow[cnt] = w;         // 边的权值
        head[i] = cnt;        // 更新头指针
    }

    /**
     * 线段树区间更新：将边(jobx, joby, jobw)添加到时间区间[jobl, jobr]内
     * @param jobl 任务开始时间
     * @param jobr 任务结束时间
     * @param jobx 边的起点
     * @param joby 边的终点
     * @param jobw 边的权值
     * @param l 当前线段树节点的左区间
     * @param r 当前线段树节点的右区间
     * @param i 当前线段树节点编号
     */
    public static void add(int jobl, int jobr, int jobx, int joby, int jobw, int l, int r, int i)
{
        // 如果当前区间完全包含在目标区间内，直接添加到当前节点
        if (jobl <= l && r <= jobr) {
            addEdge(i, jobx, joby, jobw);
        } else {
            // 否则递归到左右子树
            int mid = (l + r) >> 1;
            if (jobl <= mid) {
                add(jobl, jobr, jobx, joby, jobw, l, mid, i << 1);
            }
            if (jobr > mid) {
                add(jobl, jobr, jobx, joby, jobw, mid + 1, r, i << 1 | 1);
            }
        }
    }
```

```
/**
 * 线段树分治的深度优先搜索核心方法
 * <p>
 * 工作原理：使用深度优先搜索遍历线段树，在每个节点处应用该节点上的所有边，
 * 然后递归处理子节点，最后回溯撤销所有修改。这种方法实现了时间维度上的分治。
 * <p>
 * 算法流程：
 * 1. 保存当前线性基的大小，用于回溯
 * 2. 应用当前节点的所有边（合并集合或插入环的异或值）
 * 3. 如果是叶子节点（对应具体时间点），处理查询操作
 * 4. 否则递归处理左右子树
 * 5. 回溯：撤销所有修改，恢复到进入当前节点前的状态
 * <p>
 * 时间复杂度：每个边会被处理 O(log q)次，每次处理需要 O(BIT)时间，
 * 总体时间复杂度为 O((m+q)log q * BIT)
 *
 * @param l 当前线段树节点的左时间区间边界
 * @param r 当前线段树节点的右时间区间边界
 * @param i 当前线段树节点编号（根节点为 1，左子节点为 2*i，右子节点为 2*i+1）
 */
public static void dfs(int l, int r, int i) {
    // 保存当前线性基的大小，用于后续撤销操作
    // 这是回溯的关键步骤，确保状态能够正确恢复
    int oldsiz = basiz;

    // 记录合并操作的数量，用于后续撤销
    int unionCnt = 0;

    // 处理当前节点上的所有边
    // 这些边在[l, r]时间区间内都是活跃的
    for (int e = head[i]; e > 0; e = next[e]) {
        // 尝试合并两个集合
        // 如果成功合并（两个不同的集合），增加计数
        if (union(tox[e], toy[e], tow[e])) {
            unionCnt++;
        }
        // 如果合并失败（形成环），union 方法内部已将环的异或值插入线性基
    }

    // 处理叶子节点（对应具体的时间点）
    if (l == r) {
        // 如果当前时间点是查询操作（类型 3）
```

```
            if (op[l] == 3) {
                // 计算 x[l]到 y[l]的异或路径和：
                // 1. 首先获取 x[l]到根节点的异或路径和
                // 2. 获取 y[l]到根节点的异或路径和
                // 3. 异或这两个值，得到 x[l]到 y[l]的异或路径和
                int pathEor = getEor(x[l]) ^ getEor(y[l]);

                // 通过线性基优化，找到能与 pathEor 异或得到的最小值
                // 这一步利用了所有已知环的异或值来优化路径
                ans[l] = minEor(pathEor);
            }
        } else {
            // 非叶子节点，递归处理左右子树
            int mid = (l + r) >> 1;  // 计算中间点
            dfs(l, mid, i << 1);     // 处理左子区间
            dfs(mid + 1, r, i << 1 | 1);  // 处理右子区间
        }

        // 回溯：撤销所有修改，恢复到进入当前节点前的状态
        // 这是确保分治正确性的关键步骤
        cancel(oldsiz);  // 撤销线性基的修改，恢复到之前的大小

        // 撤销所有合并操作，按逆序撤销
        for (int k = 1; k <= unionCnt; k++) {
            undo();  // 撤销并查集的合并操作
        }
    }

    /**
     * 预处理函数：初始化并查集、排序事件、构建线段树
     * <p>
     * 核心功能：
     * 1. 初始化并查集，为每个节点设置初始父节点和集合大小
     * 2. 对所有边事件进行排序，确保相同边的添加和删除事件相邻
     * 3. 处理每条边的生命周期，确定其有效时间区间
     * 4. 将边按照有效时间区间挂载到线段树的相应节点上，为线段树分治做准备
     * <p>
     * 算法详解：
     * - 线段树分治要求我们将动态问题转换为静态问题，通过离线处理并利用时间轴上的分治策略
     * - 每条边都有一个有效时间区间，在该区间内这条边存在于图中
     * - 排序是为了让相同边的所有事件（添加和删除）集中在一起，便于处理其生命周期
     * - 线段树的每个节点表示一个时间区间，存储在该区间内所有有效的边
     * <p>
```

```
 * 时间复杂度：
 * - 排序事件：O((m+q)log(m+q))
 * - 处理边生命周期：O((m+q))
 * - 构建线段树：O((m+q)log q)
 * - 总体时间复杂度：O((m+q)log(m+q))
 */
public static void prepare() {
    // 初始化并查集结构
    // 每个节点初始时都是独立的集合，父节点指向自己，集合大小为 1
    for (int i = 1; i <= n; i++) {
        father[i] = i;  // 每个节点初始是自己的父节点
        siz[i] = 1;     // 每个集合初始大小为 1
    }

    // 按边的两个端点和时间排序事件，这是处理边生命周期的关键步骤
    // 排序规则：
    // 1. 首先按边的第一个端点 x 从小到大排序
    // 2. 然后按边的第二个端点 y 从小到大排序
    // 3. 最后按事件发生的时间 t 从小到大排序
    // 这种排序方式确保相同的边（x,y）的所有事件会集中在一起
    Arrays.sort(event, 1, eventCnt + 1,
            (a, b) -> a[0] != b[0] ? a[0] - b[0] : a[1] != b[1] ? a[1] - b[1] : a[2] - b[2]);

    int x, y, start, end, d;
    // 处理每条边的生命周期，确定边的有效时间段
    // 使用双指针技术，将相同边的所有事件分组处理
    for (int l = 1, r = 1; l <= eventCnt; l = ++r) {
        x = event[l][0];  // 当前处理的边的起点
        y = event[l][1];  // 当前处理的边的终点

        // 找到所有相同边(x,y)的事件，r 指针指向最后一个相同边的事件
        while (r + 1 <= eventCnt && event[r + 1][0] == x && event[r + 1][1] == y) {
            r++;
        }

        // 处理每对添加和删除事件，确定边的有效时间区间
        // 由于事件已经排序，添加和删除事件会交替出现
        for (int i = l; i <= r; i += 2) {
            start = event[i][2];     // 边开始的时间点（添加事件的时间）

            // 确定边结束的时间点：
            // - 如果有对应的删除事件，则边在删除事件发生前结束（end = 删除时间-1）
            // - 如果没有对应的删除事件，则边会一直存在到最后一个查询（end = q）
```

```
            end = i + 1 <= r ? (event[i + 1][2] - 1) : q;

            d = event[i][3];  // 边的权值（从添加事件中获取）

            // 将边添加到线段树的相应时间区间[start, end]
            // 这里调用线段树的区间更新函数，将边挂载到覆盖该区间的最小节点集合上
            add(start, end, x, y, d, 0, q, 1);
        }
    }
}

    /**
     * 主函数：程序入口，负责协调整个线段树分治算法的执行流程
     * <p>
     * 算法执行流程：
     * 1. 输入处理：读取图的初始状态和所有操作，包括添加边、删除边和查询操作
     * 2. 预处理：构建时间轴线段树，将边的生命周期分解到线段树节点
     * 3. 分治执行：通过 DFS 遍历线段树，动态维护图的状态并处理查询
     * 4. 结果输出：收集并输出所有查询操作的答案
     * <p>
     * 输入输出处理：
     * - 使用 FastReader 类实现高效输入，应对大规模数据
     * - 使用 PrintWriter 类进行输出缓冲，提高输出效率
     * - 所有查询结果存储在 ans 数组中，最后按顺序输出
     * <p>
     * 时间线处理：
     * - 初始边的时间点设为 0（即在所有操作前就存在）
     * - 每个操作对应一个时间点 i（1<=i<=q）
     * - 通过线段树分治处理时间维度上的动态变化
     *
     * @param args 命令行参数（未使用）
     * @throws IOException 输入输出异常
     */
    public static void main(String[] args) throws IOException {
        // 使用快速输入输出工具类，提高处理大规模数据时的效率
        FastReader in = new FastReader();
        PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));

        // 读取节点数和初始边数，初始化图的基本结构
        n = in.nextInt();
        m = in.nextInt();

        // 读取初始边信息，并为每条初始边创建一个时间点为 0 的添加事件
```

```
for (int i = 1; i <= m; i++) {
    event[i][0] = in.nextInt();  // 边的起点
    event[i][1] = in.nextInt();  // 边的终点
    event[i][2] = 0;             // 初始边的时间点设为0（开始前就存在）
    event[i][3] = in.nextInt();  // 边的权值
}
eventCnt = m;  // 记录事件数量

// 读取查询数量，这决定了时间轴的长度
q = in.nextInt();

// 读取每个操作，并为添加和删除操作记录相应事件
for (int i = 1; i <= q; i++) {
    op[i] = in.nextInt();   // 操作类型：1(添加)、2(删除)、3(查询)
    x[i] = in.nextInt();    // 操作涉及的第一个节点
    y[i] = in.nextInt();    // 操作涉及的第二个节点

    if (op[i] == 1) {       // 添加边操作，需要读取边权
        d[i] = in.nextInt();  // 边的权值
    }

    // 对于添加和删除操作，记录事件信息
    if (op[i] != 3) {
        event[++eventCnt][0] = x[i];  // 边的起点
        event[eventCnt][1] = y[i];    // 边的终点
        event[eventCnt][2] = i;       // 事件发生的时间点（即操作序号）
        event[eventCnt][3] = d[i];    // 边的权值（删除操作时也保存该值）
    }
}

// 预处理阶段：初始化并查集，排序事件，构建线段树
// 将每条边按照其有效时间区间挂载到线段树的相应节点上
prepare();

// 执行线段树分治的核心算法
// 从时间区间[0, q]开始，以根节点（编号1）为起点进行DFS遍历
// 在遍历过程中动态维护图的状态，并处理所有查询操作
dfs(0, q, 1);

// 输出所有查询操作的答案
// 遍历所有时间点，如果该时间点是查询操作，则输出对应的结果
for (int i = 1; i <= q; i++) {
    if (op[i] == 3) {
```

```
            out.println(ans[i]);
        }
    }

    // 确保所有输出都被写入到控制台
    out.flush();
    out.close();
}

// 时间复杂度分析：
// - 事件排序: O((m+q)log(m+q))
// - 线段树构建和区间更新: O((m+q)log q)
// - DFS 遍历线段树: O((m+q)log q * BIT)
// - 并查集操作: O(α(n)) 近似常数时间
// 总体时间复杂度: O((m+q)log q * BIT)
//
// 空间复杂度分析：
// - 存储事件和操作: O(m+q)
// - 线段树任务列表: O((m+q)log q)
// - 并查集和线性基: O(n + BIT)
// 总体空间复杂度: O((m+q)log q + n)

// 算法优势：
// 1. 离线处理所有操作，支持动态加边和删边
// 2. 利用线段树分治避免了直接处理删除操作
// 3. 通过线性基高效计算异或最小值
// 4. 可撤销数据结构保证了回溯时的正确性

/**
 * C++版本的核心实现思路
 *
 * #include <bits/stdc++.h>
 * using namespace std;
 *
 * const int MAXN = 200001;
 * const int MAXT = 5000001;
 * const int BIT = 29;
 * int n, m, q;
 *
 * int event[MAXN << 1][4], eventCnt;
 * int op[MAXN], x[MAXN], y[MAXN], d[MAXN];
 * int basis[BIT + 1], inspos[BIT + 1], basiz;
 * int father[MAXN], siz[MAXN], eor[MAXN];
```

```
* int rollback[MAXN][2], opsize;
* int head[MAXN << 2], next_[MAXT], tox[MAXT], toy[MAXT], tow[MAXT], cnt;
* int ans[MAXN];
*
* void insert(int num) {
*     for (int i = BIT; i >= 0; --i) {
*         if ((num >> i) & 1) {
*             if (!basis[i]) {
*                 basis[i] = num;
*                 inspos[basiz++] = i;
*                 return;
*             }
*             num ^= basis[i];
*         }
*     }
* }
*
* int minEor(int num) {
*     for (int i = BIT; i >= 0; --i) {
*         num = min(num, num ^ basis[i]);
*     }
*     return num;
* }
*
* void cancel(int oldsiz) {
*     while (basiz > oldsiz) {
*         basis[inspos[--basiz]] = 0;
*     }
* }
*
* int find(int i) {
*     while (i != father[i]) i = father[i];
*     return i;
* }
*
* int getEor(int i) {
*     int ans = 0;
*     while (i != father[i]) {
*         ans ^= eor[i];
*         i = father[i];
*     }
*     return ans;
* }
```

```
*
* bool unite(int u, int v, int w) {
*     int fu = find(u), fv = find(v);
*     w = getEor(u) ^ getEor(v) ^ w;
*     if (fu == fv) {
*         insert(w);
*         return false;
*     }
*     if (siz[fu] < siz[fv]) swap(fu, fv);
*     father[fv] = fu;
*     siz[fu] += siz[fv];
*     eor[fv] = w;
*     rollback[++opsize][0] = fu;
*     rollback[opsize][1] = fv;
*     return true;
* }
*
* void undo() {
*     int fx = rollback[opsize][0], fy = rollback[opsize--][1];
*     father[fy] = fy;
*     eor[fy] = 0;
*     siz[fx] -= siz[fy];
* }
*
* void addEdge(int i, int x, int y, int w) {
*     next_[++cnt] = head[i];
*     tox[cnt] = x;
*     toy[cnt] = y;
*     tow[cnt] = w;
*     head[i] = cnt;
* }
*
* void add(int jobl, int jobr, int jobx, int joby, int jobw, int l, int r, int i) {
*     if (jobl <= l && r <= jobr) {
*         addEdge(i, jobx, joby, jobw);
*     } else {
*         int mid = (l + r) >> 1;
*         if (jobl <= mid) add(jobl, jobr, jobx, joby, jobw, l, mid, i << 1);
*         if (jobr > mid) add(jobl, jobr, jobx, joby, jobw, mid + 1, r, i << 1 | 1);
*     }
* }
*
* void dfs(int l, int r, int i) {
```

```
*     int oldsiz = basiz, unionCnt = 0;
*     for (int e = head[i]; e; e = next_[e]) {
*         if (unite(tox[e], toy[e], tow[e])) {
*             unionCnt++;
*         }
*     }
*     if (l == r) {
*         if (op[l] == 3) {
*             ans[l] = minEor(getEor(x[l]) ^ getEor(y[l]));
*         }
*     } else {
*         int mid = (l + r) >> 1;
*         dfs(l, mid, i << 1);
*         dfs(mid + 1, r, i << 1 | 1);
*     }
*     cancel(oldsiz);
*     while (unionCnt--) undo();
* }
*
* void prepare() {
*     for (int i = 1; i <= n; ++i) {
*         father[i] = i;
*         siz[i] = 1;
*     }
*     sort(event + 1, event + eventCnt + 1, [](int a[], int b[]) {
*         if (a[0] != b[0]) return a[0] < b[0];
*         if (a[1] != b[1]) return a[1] < b[1];
*         return a[2] < b[2];
*     });
*     int x, y, start, end, dist;
*     for (int l = 1, r = 1; l <= eventCnt; l = ++r) {
*         x = event[l][0];
*         y = event[l][1];
*         while (r + 1 <= eventCnt && event[r+1][0] == x && event[r+1][1] == y) r++;
*         for (int i = l; i <= r; i += 2) {
*             start = event[i][2];
*             end = i + 1 <= r ? (event[i+1][2] - 1) : q;
*             dist = event[i][3];
*             add(start, end, x, y, dist, 0, q, 1);
*         }
*     }
* }
*
```

```
 * int main() {
 *     ios::sync_with_stdio(false);
 *     cin.tie(0);
 *     cin >> n >> m;
 *     for (int i = 1; i <= m; ++i) {
 *         cin >> event[i][0] >> event[i][1] >> event[i][3];
 *         event[i][2] = 0;
 *     }
 *     eventCnt = m;
 *     cin >> q;
 *     for (int i = 1; i <= q; ++i) {
 *         cin >> op[i] >> x[i] >> y[i];
 *         if (op[i] == 1) {
 *             cin >> d[i];
 *         }
 *         if (op[i] != 3) {
 *             event[++eventCnt][0] = x[i];
 *             event[eventCnt][1] = y[i];
 *             event[eventCnt][2] = i;
 *             event[eventCnt][3] = d[i];
 *         }
 *     }
 *     prepare();
 *     dfs(0, q, 1);
 *     for (int i = 1; i <= q; ++i) {
 *         if (op[i] == 3) {
 *             cout << ans[i] << '\n';
 *         }
 *     }
 *     return 0;
 * }
 */

/**
 * Python 版本的核心实现思路
 *
 * import sys
 * sys.setrecursionlimit(1 << 25)
 *
 * MAXN = 200001
 * MAXT = 5000001
 * BIT = 29
 *
```

```
* event = [[0]*4 for _ in range(MAXN << 1)]
* eventCnt = 0
*
* op = [0]*MAXN
* x = [0]*MAXN
* y = [0]*MAXN
* d = [0]*MAXN
*
* basis = [0]*(BIT + 1)
* inspos = [0]*(BIT + 1)
* basiz = 0
*
* father = [0]*MAXN
* siz = [0]*MAXN
* eor = [0]*MAXN
* rollback = [[0]*2 for _ in range(MAXN)]
* opsize = 0
*
* head = [0]*(MAXN << 2)
* next_ = [0]*MAXT
* tox = [0]*MAXT
* toy = [0]*MAXT
* tow = [0]*MAXT
* cnt = 0
*
* ans = [0]*MAXN
*
* def insert(num):
*     global basiz
*     for i in range(BIT, -1, -1):
*         if (num >> i) & 1:
*             if basis[i] == 0:
*                 basis[i] = num
*                 inspos[basiz] = i
*                 basiz += 1
*                 return
*             num ^= basis[i]
*
* def minEor(num):
*     for i in range(BIT, -1, -1):
*         num = min(num, num ^ basis[i])
*     return num
*
```

```
* def cancel(oldsiz):
*     global basiz
*     while basiz > oldsiz:
*         basiz -= 1
*         basis[inspos[basiz]] = 0
*
* def find(i):
*     while i != father[i]:
*         i = father[i]
*     return i
*
* def getEor(i):
*     res = 0
*     while i != father[i]:
*         res ^= eor[i]
*         i = father[i]
*     return res
*
* def unite(u, v, w):
*     global opsize
*     fu = find(u)
*     fv = find(v)
*     w = getEor(u) ^ getEor(v) ^ w
*     if fu == fv:
*         insert(w)
*         return False
*     if siz[fu] < siz[fv]:
*         fu, fv = fv, fu
*     father[fv] = fu
*     siz[fu] += siz[fv]
*     eor[fv] = w
*     opsize += 1
*     rollback[opsize][0] = fu
*     rollback[opsize][1] = fv
*     return True
*
* def undo():
*     global opsize
*     fx = rollback[opsize][0]
*     fy = rollback[opsize][1]
*     opsize -= 1
*     father[fy] = fy
*     eor[fy] = 0
```

```
*     siz[fx] -= siz[fy]
*
* def addEdge(i, x, y, w):
*     global cnt
*     cnt += 1
*     next_[cnt] = head[i]
*     tox[cnt] = x
*     toy[cnt] = y
*     tow[cnt] = w
*     head[i] = cnt
*
* def add(jobl, jobr, jobx, joby, jobw, l, r, i):
*     if jobl <= l and r <= jobr:
*         addEdge(i, jobx, joby, jobw)
*     else:
*         mid = (l + r) >> 1
*         if jobl <= mid:
*             add(jobl, jobr, jobx, joby, jobw, l, mid, i << 1)
*         if jobr > mid:
*             add(jobl, jobr, jobx, joby, jobw, mid + 1, r, i << 1 | 1)
*
* def dfs(l, r, i):
*     oldsiz = basiz
*     unionCnt = 0
*     e = head[i]
*     while e > 0:
*         if unite(tox[e], toy[e], tow[e]):
*             unionCnt += 1
*         e = next_[e]
*     if l == r:
*         if op[l] == 3:
*             ans[l] = minEor(getEor(x[l]) ^ getEor(y[l]))
*     else:
*         mid = (l + r) >> 1
*         dfs(l, mid, i << 1)
*         dfs(mid + 1, r, i << 1 | 1)
*     cancel(oldsiz)
*     for _ in range(unionCnt):
*         undo()
*
* def prepare(n_val):
*     for i in range(1, n_val + 1):
*         father[i] = i
```

```
*         siz[i] = 1
*     # 排序事件
*     event_list = []
*     for i in range(1, eventCnt + 1):
*         event_list.append(event[i])
*     event_list.sort(key=lambda e: (e[0], e[1], e[2]))
*     for i in range(eventCnt):
*         event[i + 1] = event_list[i]
*     l = 1
*     while l <= eventCnt:
*         r = l
*         x_val = event[l][0]
*         y_val = event[l][1]
*         while r + 1 <= eventCnt and event[r+1][0] == x_val and event[r+1][1] == y_val:
*             r += 1
*         i = l
*         while i <= r:
*             start = event[i][2]
*             end = event[i+1][2] - 1 if (i + 1 <= r) else q_val
*             dist = event[i][3]
*             add(start, end, x_val, y_val, dist, 0, q_val, 1)
*             i += 2
*         l = r + 1
*
* def main():
*     global eventCnt, n, m, q, q_val
*     input = sys.stdin.read().split()
*     ptr = 0
*     n = int(input[ptr]); ptr +=1
*     m = int(input[ptr]); ptr +=1
*     for i in range(1, m + 1):
*         event[i][0] = int(input[ptr]); ptr +=1
*         event[i][1] = int(input[ptr]); ptr +=1
*         event[i][3] = int(input[ptr]); ptr +=1
*         event[i][2] = 0
*     eventCnt = m
*     q_val = int(input[ptr]); ptr +=1
*     q = q_val
*     for i in range(1, q + 1):
*         op[i] = int(input[ptr]); ptr +=1
*         x[i] = int(input[ptr]); ptr +=1
*         y[i] = int(input[ptr]); ptr +=1
*         if op[i] == 1:
```

```
 *             d[i] = int(input[ptr]); ptr +=1
 *         if op[i] != 3:
 *             eventCnt += 1
 *             event[eventCnt][0] = x[i]
 *             event[eventCnt][1] = y[i]
 *             event[eventCnt][2] = i
 *             event[eventCnt][3] = d[i]
 *     prepare(n)
 *     dfs(0, q, 1)
 *     for i in range(1, q + 1):
 *         if op[i] == 3:
 *             print(ans[i])
 *
 * if __name__ == "__main__":
 *     main()
 */

/**
 * 语言特性差异与注意事项：
 * 1. Java vs C++：
 *    - C++使用数组的访问效率更高，但需要手动管理内存
 *    - Java 中的数组初始化更简洁，但性能略低
 *    - C++中的位运算和位移操作与 Java 一致
 *
 * 2. Java vs Python：
 *    - Python 中的递归深度限制需要设置（sys.setrecursionlimit）
 *    - Python 中的全局变量使用需要声明 global
 *    - Python 的执行速度明显慢于 Java 和 C++，对于大数据量可能会超时
 *    - Java 的 FastReader 类在处理大规模输入时性能优于 Python 的标准输入
 *
 * 3. 优化技巧：
 *    - 使用位运算代替部分数学运算
 *    - 避免在递归中创建临时对象
 *    - 对于大数据量，使用快速输入输出方法
 *    - 注意数组的大小设置，避免数组越界
 */

/**
 * 快速输入工具类：用于高效处理大规模输入数据
 * <p>
 * 优化原理：
 * 1. 使用字节级别的缓冲区直接处理输入流，减少字符转换开销
 * 2. 预读取大块数据到缓冲区，显著减少系统 IO 调用次数
```

```
 * 3. 手动实现字符解析逻辑，避免使用高级解析器的额外开销
 * 4. 为常用数据类型提供专门的读取方法，优化解析效率
 * <p>
 * 性能分析：
 * - 相比 Scanner：速度提升 5-10 倍，尤其在处理大量整数输入时
 * - 相比 BufferedReader+StringTokenizer：减少了字符串创建和分割的开销
 * - 字节级处理：避免了不必要的字符编码转换
 * - 适用于算法竞赛中常见的大数据量输入场景
 */
static class FastReader {
    final private int BUFFER_SIZE = 1 << 16;  // 缓冲区大小：64KB，平衡内存使用和 IO 次数
    private final InputStream in;       // 输入流
    private final byte[] buffer;       // 字节缓冲区
    private int ptr, len;              // 指针位置和当前缓冲区长度

    /**
     * 构造函数：初始化输入流和缓冲区
     */
    public FastReader() {
        in = System.in;
        buffer = new byte[BUFFER_SIZE];
        ptr = len = 0;  // 初始时缓冲区为空
    }

    /**
     * 检查是否还有下一个字节可读
     * @return 如果还有字节可读，返回 true；否则返回 false
     * @throws IOException 输入输出异常
     */
    private boolean hasNextByte() throws IOException {
        // 如果当前缓冲区还有未读字节，直接返回 true
        if (ptr < len)
            return true;
        // 否则重新填充缓冲区
        ptr = 0;
        len = in.read(buffer);
        return len > 0;  // 如果读取到字节，返回 true
    }

    /**
     * 读取下一个字节
     * @return 读取的字节值
     * @throws IOException 输入输出异常
```

```
 */
private byte readByte() throws IOException {
    // 如果没有更多字节可读，返回-1
    if (!hasNextByte())
        return -1;
    // 否则返回当前字节并移动指针
    return buffer[ptr++];
}

/**
 * 检查字符是否为空白字符
 * @param b 要检查的字节
 * @return 如果是空白字符，返回 true
 */
private boolean isWhitespace(byte b) {
    // 常见空白字符：空格、制表符、换行符、回车符等
    return b <= ' ';
}

/**
 * 读取下一个字符
 * @return 读取的字符
 * @throws IOException 输入输出异常
 */
public char nextChar() throws IOException {
    byte c;
    // 跳过空白字符
    do {
        c = readByte();
        if (c == -1)
            return 0;
    } while (c <= ' ');
    // 读取第一个非空白字符
    char ans = 0;
    while (c > ' ') {
        ans = (char) c;
        c = readByte();
    }
    return ans;
}

/**
 * 读取下一个整数
```

```
         * @return 读取的整数值
         * @throws IOException 输入输出异常
         */
        public int nextInt() throws IOException {
            int num = 0;
            // 读取一个字节
            byte b = readByte();
            // 跳过空白字符
            while (isWhitespace(b))
                b = readByte();
            // 处理负号
            boolean minus = false;
            if (b == '-') {
                minus = true;
                b = readByte();
            }
            // 读取数字部分
            while (!isWhitespace(b) && b != -1) {
                num = num * 10 + (b - '0');  // 将字符转换为数字
                b = readByte();
            }
            // 根据符号返回结果
            return minus ? -num : num;
        }
    }
}
    }

}
```

==========

文件: Code04_ShortestPathQueries2.java

==========

```
package class167;

// 异或最短路，C++版
// 一共有 n 个节点，m 条边，每条边有边权
// 接下来有 q 条操作，每种操作是如下三种类型中的一种
// 操作 1 x y d : 原图中加入，点 x 到点 y，权值为 d 的边
// 操作 2 x y   : 原图中删除，点 x 到点 y 的边
// 操作 3 x y   : 点 x 到点 y，所有路随便走，沿途边权都异或起来，打印能取得的异或最小值
// 保证 x < y，并且任意操作后，图连通、无重边、无自环，所有操作均合法
```

```
// 1 <= n、m、q <= 2 * 10^5
// 测试链接 : https://www.luogu.com.cn/problem/CF938G
// 测试链接 : https://codeforces.com/problemset/problem/938/G
// 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 Event {
//    int x, y, t, w;
//};
//
//bool EventCmp(Event a, Event b) {
//    if (a.x != b.x) {
//        return a.x < b.x;
//    } else if (a.y != b.y) {
//        return a.y < b.y;
//    } else {
//        return a.t < b.t;
//    }
//}
//
//const int MAXN = 200001;
//const int MAXT = 5000001;
//const int BIT = 29;
//int n, m, q;
//
//Event event[MAXN << 1];
//int eventCnt = 0;
//
//int op[MAXN];
//int x[MAXN];
//int y[MAXN];
//int d[MAXN];
//
//int basis[BIT + 1];
//int inspos[BIT + 1];
//int basiz = 0;
//
//int father[MAXN];
//int siz[MAXN];
```

```
//int eor[MAXN];
//int rollback[MAXN][2];
//int opsize = 0;
//
//int head[MAXN << 2];
//int nxt[MAXT];
//int tox[MAXT];
//int toy[MAXT];
//int tow[MAXT];
//int cnt = 0;
//
//int ans[MAXN];
//
//void insert(int num) {
//    for (int i = BIT; i >= 0; i--) {
//        if (num >> i == 1) {
//            if (basis[i] == 0) {
//                basis[i] = num;
//                inspos[basiz++] = i;
//                return;
//            }
//            num ^= basis[i];
//        }
//    }
//}
//
//int minEor(int num) {
//    for (int i = BIT; i >= 0; i--) {
//        num = min(num, num ^ basis[i]);
//    }
//    return num;
//}
//
//void cancel(int oldsiz) {
//    while (basiz > oldsiz) {
//        basis[inspos[--basiz]] = 0;
//    }
//}
//
//int find(int i) {
//    while (i != father[i]) {
//        i = father[i];
//    }
```

```
//    return i;
//}
//
//int getEor(int i) {
//    int res = 0;
//    while (i != father[i]) {
//        res ^= eor[i];
//        i = father[i];
//    }
//    return res;
//}
//
//bool Union(int u, int v, int w) {
//    int fu = find(u);
//    int fv = find(v);
//    w = getEor(u) ^ getEor(v) ^ w;
//    if (fu == fv) {
//        insert(w);
//        return false;
//    }
//    if (siz[fu] < siz[fv]) {
//        int tmp = fu;
//        fu = fv;
//        fv = tmp;
//    }
//    father[fv] = fu;
//    siz[fu] += siz[fv];
//    eor[fv] = w;
//    rollback[++opsize][0] = fu;
//    rollback[opsize][1] = fv;
//    return true;
//}
//
//void undo() {
//    int fu = rollback[opsize][0];
//    int fv = rollback[opsize--][1];
//    father[fv] = fv;
//    eor[fv] = 0;
//    siz[fu] -= siz[fv];
//}
//
//void addEdge(int idx, int u, int v, int w) {
//    nxt[++cnt] = head[idx];
```

```
//    tox[cnt] = u;
//    toy[cnt] = v;
//    tow[cnt] = w;
//    head[idx] = cnt;
//}
//
//void add(int jobl, int jobr, int jobx, int joby, int jobw, int l, int r, int i) {
//    if (jobl <= l && r <= jobr) {
//        addEdge(i, jobx, joby, jobw);
//    } else {
//        int mid = (l + r) >> 1;
//        if (jobl <= mid) {
//            add(jobl, jobr, jobx, joby, jobw, l, mid, i << 1);
//        }
//        if (jobr > mid) {
//            add(jobl, jobr, jobx, joby, jobw, mid + 1, r, i << 1 | 1);
//        }
//    }
//}
//
//void dfs(int l, int r, int i) {
//    int oldsiz = basiz;
//    int unionCnt = 0;
//    for (int e = head[i]; e; e = nxt[e]) {
//        if (Union(tox[e], toy[e], tow[e])) {
//            unionCnt++;
//        }
//    }
//    if (l == r) {
//        if (op[l] == 3) {
//            ans[l] = minEor(getEor(x[l]) ^ getEor(y[l]));
//        }
//    } else {
//        int mid = (l + r) >> 1;
//        dfs(l, mid, i << 1);
//        dfs(mid + 1, r, i << 1 | 1);
//    }
//    cancel(oldsiz);
//    for (int k = 1; k <= unionCnt; k++) {
//        undo();
//    }
//}
//
```

```
//void prepare() {
//    for (int i = 1; i <= n; i++) {
//        father[i] = i;
//        siz[i] = 1;
//    }
//    sort(event + 1, event + eventCnt + 1, EventCmp);
//    int x, y, start, end, d;
//    for (int l = 1, r = 1; l <= eventCnt; l = ++r) {
//        x = event[l].x;
//        y = event[l].y;
//        while (r + 1 <= eventCnt && event[r + 1].x == x && event[r + 1].y == y) {
//            r++;
//        }
//        for (int i = l; i <= r; i += 2) {
//            start = event[i].t;
//            end = (i + 1 <= r) ? (event[i + 1].t - 1) : q;
//            d = event[i].w;
//            add(start, end, x, y, d, 0, q, 1);
//        }
//    }
//}
//
//int main() {
//    ios::sync_with_stdio(false);
//    cin.tie(nullptr);
//    cin >> n >> m;
//    for (int i = 1; i <= m; i++) {
//        cin >> event[i].x >> event[i].y >> event[i].w;
//        event[i].t = 0;
//    }
//    eventCnt = m;
//    cin >> q;
//    for (int i = 1; i <= q; i++) {
//        cin >> op[i] >> x[i] >> y[i];
//        if (op[i] == 1) {
//            cin >> d[i];
//        }
//        if (op[i] != 3) {
//            event[++eventCnt].x = x[i];
//            event[eventCnt].y = y[i];
//            event[eventCnt].t = i;
//            event[eventCnt].w = d[i];
//        }
```

```
//    }
//    prepare();
//    dfs(0, q, 1);
//    for (int i = 1; i <= q; i++) {
//        if (op[i] == 3) {
//            cout << ans[i] << '\n';
//        }
//    }
//    return 0;
//}
```

==========

文件: Code05_EightVerticalHorizontal1.java

==========

```
package class167;

/**
 * 八纵八横问题 - 使用线段树分治 + 可撤销线性基 + 带权并查集
 *
 * 【题目描述】
 * 有 n 个点，给定 m 条边，每条边的边权用 01 字符串表达，初始时图保证连通
 * 初始的 m 条边永不删除，接下来有 q 条操作，操作分为三种类型：
 * - Add x y z：加入点 x 到点 y 的边，边权是 z（01 字符串），第 k 条添加操作的边编号为 k
 * - Cancel k：删除编号为 k 的边
 * - Change k z：将编号为 k 的边的边权修改为 z
 * 要求计算从 1 号点出发最后回到 1 号点的回路中，所有边权异或的最大值
 * 需要输出初始状态以及每个操作后的异或最大值
 *
 * 【输入输出】
 * 输入：n, m, q，然后是 m 条初始边，最后是 q 个操作
 * 输出：初始状态和每个操作后的异或最大值（01 字符串形式）
 *
 * 【算法核心】
 * 1. 线段树分治：处理动态边的添加、删除和修改操作
 * 2. 带权并查集：维护连通性和路径异或值
 * 3. 可撤销线性基：计算异或最大值
 * 4. 位图（BitSet）：高效存储和处理长二进制边权
 *
 * 【时间复杂度】
 * O((m + q) * BIT * log q)，其中 BIT 是边权的最大位数（本题为 999 位）
 * - 线段树分治的时间复杂度为 O((m + q) * log q)
 * - 线性基操作的时间复杂度为 O(BIT)
```

```
 * - 每次合并操作的时间复杂度为O(BIT)
 *
 * 【空间复杂度】
 * O((m + q) * log q + BIT * MAXQ)，用于存储线段树节点、线性基和并查集信息
 *
 * 【优化技巧】
 * 1. 使用位图（BitSet）高效存储长二进制数
 * 2. 带权并查集只做路径压缩，不做按秩合并，以支持撤销操作
 * 3. 线段树分治处理动态边的生命周期
 * 4. 使用FastReader优化输入速度
 *
 * 【测试链接】
 * https://www.luogu.com.cn/problem/P3733
 */

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

/**
 * 八纵八横问题（洛谷P3733）
 * 题目来源：洛谷 https://www.luogu.com.cn/problem/P3733
 *
 * 核心算法：线段树分治 + 带权并查集 + 可撤销线性基 + 位图处理
 *
 * 问题描述：
 * 给定一个动态变化的无向图，支持三种操作：
 * 1. Add u v w：添加一条连接u和v，权值为w的边
 * 2. Cancel k：删除第k次Add操作添加的边
 * 3. Change k w：修改第k次Add操作添加的边的权值为w
 *
 * 要求：在初始状态和每次操作后，输出图中异或和最大的回路（异或回路最大值）。
 * 若不存在回路，输出0。
 *
 * 算法思路：
 * 1. 使用线段树分治处理动态边，将每条边的生命周期区间转化为线段树上的节点
 * 2. 使用带权并查集维护节点之间的连通性和异或路径值
 * 3. 使用可撤销线性基存储环的异或值，用于计算最大异或回路
 * 4. 使用位图（BitSet）处理超长二进制数的异或运算
 *
 * 时间复杂度分析：
 * - 线段树分治：O((m + q) * log q)
```

* - 线性基操作：O(BIT) 每操作
* - 总体时间复杂度：O((m + q) * log q * BIT)，其中 BIT=999 是二进制数的最大位数
*
* 空间复杂度分析：
* - 线段树：O((m + q) * log q)
* - 并查集：O(n)
* - 线性基：O(BIT)
* - 总体空间复杂度：O(n + BIT + (m + q) * log q)
*
* 多语言实现对比：
* - Java：使用位图（BitSet）自定义类处理超长二进制数，实现可撤销数据结构较复杂
* - C++：可使用 bitset 模板类更高效地处理二进制操作，指针操作更灵活，效率更高
* - Python：位运算效率较低，但实现思路相同，适合处理小规模测试用例
*
* 优化技巧：
* 1. 使用链式前向星存储线段树节点的边列表
* 2. 实现高效的 FastReader 类处理大量输入
* 3. 使用可撤销的并查集和线性基实现回溯
* 4. 使用位图（BitSet）分块存储超长二进制数
* 5. 并查集只进行路径压缩，不进行按秩合并，以便支持撤销操作
* 6. 线段树分治将动态问题转化为静态问题处理
*
* 注意事项：
* 1. 由于边权可能很长（最长 999 位），不能使用普通整数类型存储
* 2. 线段树分治需要离线处理所有操作
* 3. 撤销操作需要正确维护并查集和线性基的状态
*
* 测试链接：https://www.luogu.com.cn/problem/P3733
*
* 线段树分治相关题目训练列表：
*
* 1. 二分图 /【模板】线段树分治 - P5787 (洛谷)
*    链接：https://www.luogu.com.cn/problem/P5787
*    描述：维护动态图使其为二分图
*    解法：线段树分治 + 扩展域并查集
*    复杂度：O((n + m) log m)
*
* 2. 最小异或查询 - ABC308G (AtCoder)
*    链接：https://atcoder.jp/contests/abc308/tasks/abc308_g
*    描述：维护一个集合，支持添加/删除数字，查询任意两数异或最小值
*    解法：01Trie + 在线维护
*    复杂度：O(q log V)
*

* 3. 火星商店 - P4585 (洛谷)
*    链接：https://www.luogu.com.cn/problem/P4585
*    描述：维护 n 个商店，支持添加商品，查询特定范围异或最大值
*    解法：线段树分治 + 可持久化 Trie
*    复杂度：O((n+q) log q log V)
*
* 4. 唯一出现次数 - 1681F (Codeforces)
*    链接：https://codeforces.com/contest/1681/problem/F
*    描述：统计树上路径中唯一出现的颜色数量
*    解法：线段树分治 + 可撤销并查集
*    复杂度：O((n + m) log m)
*
* 5. 边着色 - 576E (Codeforces)
*    链接：https://codeforces.com/contest/576/problem/E
*    描述：给边着色使得每种颜色构成的子图都是二分图
*    解法：线段树分治 + 多个扩展域并查集
*    复杂度：O((n + m) log m)
*
* 6. 连通图 - P5227 (洛谷)
*    链接：https://www.luogu.com.cn/problem/P5227
*    描述：初始连通图，每次删除一些边，查询是否仍连通
*    解法：线段树分治 + 可撤销并查集
*    复杂度：O((n + m) log m)
*
* 7. 大融合 - P4219 (洛谷)
*    链接：https://www.luogu.com.cn/problem/P4219
*    描述：支持加边和查询边负载（删去该边后连通块大小乘积）
*    解法：线段树分治 + 可撤销并查集
*    复杂度：O((n + m) log m)
*
* 8. 最小 mex 生成树 - P5631 (洛谷)
*    链接：https://www.luogu.com.cn/problem/P5631
*    描述：求生成树使得边权集合的 mex 最小
*    解法：线段树分治 + 可撤销并查集 + 二分答案
*    复杂度：O((n + m) log m log n)
*
* 9. 博物馆劫案 - CF601E / Luogu P4585
*    链接：https://codeforces.com/contest/601/problem/E
*    描述：支持添加/删除商品，查询背包问题变形
*    解法：线段树分治 + 动态规划
*    复杂度：O(qk log q + nk)
*
* 10. 细胞分裂 - AGC010C (AtCoder)

```
 *    链接：https://atcoder.jp/contests/agc010/tasks/agc010_c
 *    描述：分割矩形并计算每次分割后的连通分量数
 *    解法：线段树分治 + 可撤销并查集
 *    复杂度：O((n + m) log m)
 */
public class Code05_EightVerticalHorizontal1 {

    // 常量定义
    public static final int MAXN = 501;     // 最大节点数
    public static final int MAXQ = 1001;    // 最大操作数
    public static final int MAXT = 10001;   // 最大边操作数
    public static final int BIT = 999;      // 边权的最大位数（二进制）
    public static final int INT_BIT = 32;   // 整型位数，用于位图分块存储

    /**
 * 位图（BitSet）类 - 用于高效存储和处理长二进制数
 *
 * 由于边权长度可达 1000 位，超过 Java 内置整数类型，因此需要自定义位图实现
 * 使用整型数组分块存储，每 32 位存储在一个整型中，采用紧凑存储提高空间利用率
 *
 * 核心设计思想：
 * 1. 分块存储：每 INT_BIT 位（32 位）存储在一个整型中，通过位运算高效访问和修改
 * 2. 低位优先：存储时低位在前，便于位运算和线性基的构建
 * 3. 高效操作：所有位操作基于位运算实现，避免逐位处理的性能开销
 *
 * 性能优化：
 * 1. 批量位操作：通过整型数组实现并行位处理
 * 2. 空间优化：仅存储必要位数，避免浪费内存
 * 3. 操作优化：使用位掩码和位移操作实现常数时间的位访问
 */
static class BitSet {

    public int len;         // 位图的长度（整型数组的大小）
    public int[] arr;       // 存储位图数据的整型数组

    /**
     * 构造一个空的位图
     * 时间复杂度：O(len)
     */
    public BitSet() {
        len = BIT / INT_BIT + 1;  // 计算需要的整型数组长度，向上取整
        arr = new int[len];       // 初始化整型数组，默认全 0
    }
```

```
/**
 * 从字符串构造位图
 * @param s 二进制字符串，高位在前，低位在后
 * 时间复杂度：O(s.length())
 */
public BitSet(String s) {
    len = BIT / INT_BIT + 1;
    arr = new int[len];
    // 将字符串转换为位图，注意反转顺序，因为字符串高位在前，而我们需要低位在前
    // 这种转换方式确保了位运算的正确性，使得第 i 位对应数值的 2^i 位
    for (int i = 0, j = s.length() - 1; i < s.length(); i++, j--) {
        set(i, s.charAt(j) - '0');  // 设置第 i 位的值（0 或 1）
    }
}

/**
 * 获取位图中第 i 位的值
 * @param i 要获取的位的索引（从 0 开始）
 * @return 该位的值（0 或 1）
 * 时间复杂度：O(1) - 位运算常数时间
 */
public int get(int i) {
    // 找到对应的整型块，然后通过位移和与运算获取该位的值
    // i/INT_BIT 计算属于哪个整型块，i%INT_BIT 计算在该块中的位置
    return (arr[i / INT_BIT] >> (i % INT_BIT)) & 1;
}

/**
 * 设置位图中第 i 位的值
 * @param i 要设置的位的索引（从 0 开始）
 * @param v 要设置的值（0 或 1）
 * 时间复杂度：O(1) - 位运算常数时间
 */
public void set(int i, int v) {
    if (v == 0) {
        // 清除该位：与上取反后的掩码，保持其他位不变
        arr[i / INT_BIT] &= ~(1 << (i % INT_BIT));
    } else {
        // 设置该位：或上对应的掩码，保持其他位不变
        arr[i / INT_BIT] |= 1 << (i % INT_BIT);
    }
}
```

```
    /**
     * 与另一个位图进行异或操作
     * @param other 要异或的另一个位图
     * 时间复杂度：O(len) - 每个整型块独立异或
     */
    public void eor(BitSet other) {
        // 对每个整型块进行异或操作，利用硬件级并行计算
        // 这比逐位异或效率高得多，尤其是处理长二进制数时
        for (int i = 0; i < len; i++) {
            arr[i] ^= other.arr[i];  // 对每个整型进行异或操作
        }
    }

    /**
     * 清空位图，所有位设置为0
     * 时间复杂度：O(len)
     */
    public void clear() {
        // 直接将所有整型块置0，比逐位清除效率高
        for (int i = 0; i < len; i++) {
            arr[i] = 0;
        }
    }

}

    // 全局变量
    public static int n;                    // 节点数
    public static int m;                    // 初始边数
    public static int q;                    // 操作数
    public static int[] x = new int[MAXQ];  // 记录添加的边的x端点
    public static int[] y = new int[MAXQ];  // 记录添加的边的y端点
    public static BitSet[] w = new BitSet[MAXQ];  // 记录添加的边的权值
    public static int edgeCnt = 0;          // 当前边的计数器
    public static int[] last = new int[MAXQ];     // 记录每条边的最后活跃时间

    // 可撤销线性基 - 用于计算异或最大值
    public static BitSet[] basis = new BitSet[BIT + 1];  // 线性基数组，basis[i]表示最高位为i的基向量
    public static int[] inspos = new int[BIT + 1];       // 记录插入顺序的位置，用于撤销
    public static int basiz = 0;                         // 线性基的大小
```

```
// 经典带权并查集 - 维护连通性和路径异或值
// 注意：只做路径压缩（扁平化），不做按秩合并，以支持撤销操作
public static int[] father = new int[MAXN];    // 并查集父节点数组
public static BitSet[] eor = new BitSet[MAXN]; // 并查集路径异或值数组

// 时间轴线段树上的区间任务列表 - 使用链式前向星存储
public static int[] head = new int[MAXQ << 2];  // 每个线段树节点对应的边链表头
public static int[] next = new int[MAXT];       // 边链表的 next 指针
public static int[] tox = new int[MAXT];        // 边的 x 端点
public static int[] toy = new int[MAXT];        // 边的 y 端点
public static BitSet[] tow = new BitSet[MAXT]; // 边的权值
public static int cnt = 0;                      // 边计数器

// 每一步的最大异或值
public static BitSet[] ans = new BitSet[MAXQ]; // 存储每个时间点的异或最大值

/**
 * 将一个数插入线性基
 * @param num 要插入的位图表示的数
 * 时间复杂度：O(BIT)
 */
public static void insert(BitSet num) {
    for (int i = BIT; i >= 0; i--) {
        // 从高位到低位寻找第一个为 1 的位
        if (num.get(i) == 1) {
            // 如果该位没有基向量，直接插入
            if (basis[i].get(i) == 0) {
                basis[i] = num;
                inspos[basiz++] = i;  // 记录插入的位置，用于撤销
                return;
            }
            // 否则，将 num 异或上该位的基向量，继续处理
            num.eor(basis[i]);
        }
    }
    // 如果 num 最终变为 0，表示它可以被当前线性基表示，不需要插入
}

/**
 * 计算与线性基的最大异或值
 * @return 异或最大值的位图表示
 * 时间复杂度：O(BIT)
 */
```

```
    public static BitSet maxEor() {
        BitSet ans = new BitSet();
        // 从高位到低位遍历线性基
        for (int i = BIT; i >= 0; i--) {
            // 如果当前位为0，且存在该位的基向量，则异或上该基向量
            // 这样可以尽可能使高位为1，从而得到最大值
            if (ans.get(i) == 0 && basis[i].get(i) == 1) {
                ans.eor(basis[i]);
            }
        }
        return ans;
    }

    /**
     * 撤销线性基到指定大小
     * @param oldsiz 要恢复到的线性基大小
     * 时间复杂度：O(basiz - oldsiz)
     */
    public static void cancel(int oldsiz) {
        // 撤销所有在oldsiz之后插入的元素
        while (basiz > oldsiz) {
            // 清空对应的基向量
            basis[inspos[--basiz]].clear();
        }
    }

    // 扁平化优化，find的同时修改eor，就是经典的带权并查集
/**
 * 并查集查找函数（带路径压缩）
 * 采用递归实现的路径压缩，确保后续查找操作接近O(1)时间复杂度
 *
 * 核心技术点：
 * 1. 路径压缩：在查找过程中，将路径上的所有节点直接连接到根节点
 * 2. 动态维护路径异或值：递归回溯时更新当前节点到根节点的异或值
 * 3. 无按秩合并：为支持撤销操作，不使用按秩合并优化
 *
 * @param i 要查找的节点
 * @return 节点i所在集合的根节点
 * @throws StackOverflowError 当递归深度过深时可能发生
 * 时间复杂度：O(α(n))，其中 α 是阿克曼函数的反函数，实际应用中近似为常数
 */
public static int find(int i) {
    if (i != father[i]) {
```

```
        int tmp = father[i];  // 保存原父节点
        father[i] = find(tmp);  // 递归查找根节点并进行路径压缩
        eor[i].eor(eor[tmp]);   // 更新路径异或值：i 到父节点 + 父节点到根
    }
    return father[i];  // 返回根节点
}

/**
 * 获取从节点 i 到根节点的路径异或值
 * 调用 find 函数确保路径已压缩，异或值已更新
 *
 * @param i 目标节点
 * @return 路径异或值的位图表示
 * 时间复杂度：O(α(n) * BIT)，其中 BIT 是边权的最大位数
 */
public static BitSet getEor(int i) {
    find(i);  // 确保路径已压缩，异或值已更新
    return eor[i];  // 直接返回计算好的路径异或值
}

/**
 * 合并两个节点所在的集合，并处理可能形成的环
 *
 * 工作原理：
 * 1. 查找两个节点的根节点
 * 2. 计算 u 到 v 的路径异或值：path(u) ^ path(v) ^ w(u,v)
 * 3. 如果两个根节点相同，说明形成环，将环的异或值插入线性基
 * 4. 如果不同，将一个集合的根连接到另一个集合的根，并设置适当的路径异或值
 *
 * 优化技巧：
 * 1. 利用路径压缩加速查找
 * 2. 通过异或操作的性质高效计算环的异或值
 * 3. 仅在形成环时插入线性基，减少不必要的计算
 *
 * @param u 第一个节点
 * @param v 第二个节点
 * @param w 边 u-v 的权值
 * 时间复杂度：O(α(n) * BIT)
 */
public static void union(int u, int v, BitSet w) {
    int fu = find(u);  // 查找 u 的根节点
    int fv = find(v);  // 查找 v 的根节点
```

```
    // 计算从 u 到 v 的异或路径值：u 到 fu 的异或值 ^ v 到 fv 的异或值 ^ 边 u-v 的权值
    BitSet weight = new BitSet();
    weight.eor(getEor(u));
    weight.eor(getEor(v));
    weight.eor(w);

    if (fu == fv) {
        // u 和 v 已经在同一集合中，形成环，将环的异或值插入线性基
        // 环的异或值等于 u 到 v 的异或路径值
        insert(weight);
    } else {
        // 合并两个不同的集合
        // 直接将 fv 的父节点设置为 fu（未使用按秩合并以支持撤销）
        father[fv] = fu;
        // 设置 fv 到 fu 的路径异或值为计算得到的 weight
        eor[fv] = weight;
    }
}

    /**
     * 向线段树节点添加一条边
     * @param i 线段树节点编号
     * @param x 边的起点
     * @param y 边的终点
     * @param w 边的权值
     * 时间复杂度：O(1)
     */
    public static void addEdge(int i, int x, int y, BitSet w) {
        // 使用链式前向星存储边，next[cnt]指向下一条边
        next[++cnt] = head[i];
        tox[cnt] = x;      // 边的起点
        toy[cnt] = y;      // 边的终点
        tow[cnt] = w;      // 边的权值
        head[i] = cnt;     // 更新当前节点的边链表头
    }

    /**
     * 线段树分治的核心方法：将边添加到线段树的相应区间
     * 算法核心思想：将边的生命周期区间分解到线段树的各个节点上，确保每条边只在其有效时间区间内被处理
     *
     * @param jobl 边的有效区间左端点
     * @param jobr 边的有效区间右端点
```

```
     * @param jobx 边的起点
     * @param joby 边的终点
     * @param jobw 边的权值（位图表示的二进制数）
     * @param l 当前线段树节点区间的左端点
     * @param r 当前线段树节点区间的右端点
     * @param i 当前线段树节点编号
     *
     * 时间复杂度：O(log q)，其中 q 是最大操作数
     * 空间复杂度：O(log q)，递归调用栈深度
     *
     * 优化策略：
     * 1. 利用线段树的区间分解特性，每条边最多被分解到 O(log q)个节点
     * 2. 采用链式前向星存储每个节点的边列表，避免内存浪费
     * 3. 对于完全覆盖的区间，直接添加边，不继续递归，减少函数调用开销
     */
    public static void add(int jobl, int jobr, int jobx, int joby, BitSet jobw, int l, int r, int
i) {
        if (jobl <= l && r <= jobr) {
            // 当前线段树节点区间完全包含在边的有效区间内，直接添加边
            addEdge(i, jobx, joby, jobw);
        } else {
            // 否则递归到左右子树，将边分解到更细粒度的区间
            int mid = (l + r) >> 1;  // 计算中间点，等价于(l + r) / 2，但使用位运算提高效率
            if (jobl <= mid) {
                // 边的有效区间与左子树区间有交集，递归处理左子树
                add(jobl, jobr, jobx, joby, jobw, l, mid, i << 1);
            }
            if (jobr > mid) {
                // 边的有效区间与右子树区间有交集，递归处理右子树
                add(jobl, jobr, jobx, joby, jobw, mid + 1, r, i << 1 | 1);
            }
        }
    }

    /**
     * 线段树分治的 DFS 遍历过程
     * 算法核心：深度优先遍历线段树，在进入当前节点时应用所有该节点上的边，在离开时撤销这些边的影
响
     * 实现递归回溯机制，确保每次递归调用结束后恢复线性基的状态
     *
     * @param l 当前线段树节点区间的左端点
     * @param r 当前线段树节点区间的右端点
     * @param i 当前线段树节点编号
```

```
 *
 * 时间复杂度：O((m + q) * log q * BIT)，其中：
 *             - 线段树分治处理O((m + q) * log q)条边
 *             - 每条边的处理需要O(BIT)的线性基和并查集操作
 *
 * 递归回溯机制详解：
 * 1. 保存当前线性基的大小(oldsiz)，作为回溯的标记点
 * 2. 处理当前节点上的所有边，这些边在[l, r]区间内有效
 * 3. 如果是叶子节点，计算并保存该时间点的最大异或回路值
 * 4. 如果是非叶子节点，递归处理左右子树
 * 5. 递归返回后，调用 cancel 方法撤销线性基的修改，恢复到调用前的状态
 *
 * 这种回溯机制确保了每条边只在其有效时间区间内被考虑，且不会影响到其他时间区间的计算
 */
public static void dfs(int l, int r, int i) {
    // 保存当前线性基的大小，作为回溯的标记点
    // 这是实现撤销机制的关键，记录调用前的线性基状态
    int oldsiz = basiz;

    // 处理当前节点上的所有边
    // 这些边的生命周期完全覆盖了当前线段树节点表示的时间区间
    for (int e = head[i]; e > 0; e = next[e]) {
        // 调用 union 函数合并边的两个端点，并处理可能形成的环
        // 如果形成环，会将环的异或值自动插入到线性基中
        union(tox[e], toy[e], tow[e]);
    }

    // 判断是否到达叶子节点（单个时间点）
    if (l == r) {
        // 叶子节点对应一个具体的时间点，计算此时的最大异或回路值
        // 调用 maxEor 函数，使用当前线性基计算异或最大值
        ans[l] = maxEor();
    } else {
        // 非叶子节点，需要递归处理左右子树
        // 将当前时间区间分成两半
        int mid = (l + r) / 2;
        // 递归处理左子树区间[l, mid]
        dfs(l, mid, i << 1);
        // 递归处理右子树区间[mid+1, r]
        dfs(mid + 1, r, i << 1 | 1);
    }

    // 回溯，撤销当前节点的所有线性基操作
```

```
    // 将线性基恢复到调用该方法前的状态
    // 这样可以确保后续递归调用不受当前区间边的影响
    cancel(oldsiz);
}

/**
 * 打印位图表示的二进制数
 * @param bs 要打印的位图
 * @param out 输出流
 */
public static void print(BitSet bs, PrintWriter out) {
    boolean flag = false;  // 标记是否找到第一个 1

    // 从最高位开始遍历，跳过前导零
    for (int i = BIT, s; i >= 0; i--) {
        s = bs.get(i);
        if (s == 1) {
            flag = true;  // 找到第一个 1 后，开始输出
        }
        if (flag) {
            out.print(s);
        }
    }

    // 如果所有位都是 0，输出 0
    if (!flag) {
        out.print(0);
    }
    out.println();
}

/**
 * 主函数 - 程序入口
 * @param args 命令行参数
 * @throws IOException 输入输出异常
 */
public static void main(String[] args) throws IOException {
    FastReader in = new FastReader();  // 创建高效输入流
    PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));  // 创建输出流

    // 读取输入数据
    n = in.nextInt();  // 节点数
    m = in.nextInt();  // 初始边数
```

```
q = in.nextInt();  // 操作数

// 初始化线性基数组
for (int i = 0; i <= BIT; i++) {
    basis[i] = new BitSet();
}

// 初始化并查集
for (int i = 1; i <= n; i++) {
    father[i] = i;  // 每个节点初始父节点为自身
    eor[i] = new BitSet();  // 路径异或值初始化为0
}

// 处理初始边（这些边在整个过程中都存在）
for (int i = 1; i <= m; i++) {
    int u = in.nextInt();
    int v = in.nextInt();
    BitSet w = new BitSet(in.nextString());
    union(u, v, w);  // 合并节点并处理可能形成的环
}

// 计算初始状态的最大异或回路
ans[0] = maxEor();

// 处理每个操作
String op;
for (int i = 1; i <= q; i++) {
    op = in.nextString();

    if (op.equals("Add")) {
        // 添加边操作
        edgeCnt++;
        x[edgeCnt] = in.nextInt();
        y[edgeCnt] = in.nextInt();
        w[edgeCnt] = new BitSet(in.nextString());
        last[edgeCnt] = i;  // 记录边的起始时间

    } else if (op.equals("Cancel")) {
        // 删除边操作
        int k = in.nextInt();
        // 将边k添加到时间区间[last[k], i-1]，表示这段时间内有效
        add(last[k], i - 1, x[k], y[k], w[k], 1, q, 1);
        last[k] = 0;  // 标记为已删除
```

```
        } else {  // Change 操作
            // 修改边操作
            int k = in.nextInt();
            // 先将原边添加到时间区间[last[k], i-1]
            add(last[k], i - 1, x[k], y[k], w[k], 1, q, 1);
            // 更新边的权值
            w[k] = new BitSet(in.nextString());
            last[k] = i;  // 更新边的起始时间
        }
    }

    // 处理所有在最后一次操作后仍然有效的边
    for (int i = 1; i <= edgeCnt; i++) {
        if (last[i] != 0) {
            // 将这些边添加到时间区间[last[i], q]
            add(last[i], q, x[i], y[i], w[i], 1, q, 1);
        }
    }

    // 执行线段树分治算法
    if (q > 0) {
        dfs(1, q, 1);
    }

    // 输出结果：初始状态和每个操作后的异或最大值
    for (int i = 0; i <= q; i++) {
        print(ans[i], out);
    }

    out.flush();  // 刷新输出缓冲
    out.close();  // 关闭输出流
}

/**
 * FastReader 类 - 高效输入处理器
 * 针对大规模数据输入进行优化，避免普通 Scanner 的性能瓶颈
 * 核心优化：
 * 1. 使用字节缓冲区批量读取输入，减少 IO 系统调用
 * 2. 手动处理字符解码，避免字符流的性能开销
 * 3. 针对题目特点优化，支持整数和长字符串（边权）的快速读取
 */
static class FastReader {
```

```
    private static final int BUFFER_SIZE = 1 << 16;  // 缓冲区大小（64KB），平衡内存占用与读取效率
    private final InputStream in;        // 底层输入流
    private final byte[] buffer;         // 字节缓冲区，存储批量读取的数据
    private int ptr, len;                // 指针位置和当前缓存长度

    /**
     * 构造函数，初始化输入流和缓冲区
     * 时间复杂度：O(1)
     * 空间复杂度：O(BUFFER_SIZE)
     */
    public FastReader() {
        in = System.in;
        buffer = new byte[BUFFER_SIZE];
        ptr = len = 0;
    }

    /**
     * 检查缓冲区中是否还有可读字节
     * @return 如果有可读字节返回 true，否则返回 false
     * @throws IOException 输入异常
     * 时间复杂度：O(1)，仅在需要填充缓冲区时为 O(BUFFER_SIZE)
     */
    private boolean hasNextByte() throws IOException {
        if (ptr < len) {
            return true;  // 缓冲区还有数据
        }
        // 缓冲区已读完，重新填充
        ptr = 0;
        len = in.read(buffer);  // 批量读取数据到缓冲区
        return len > 0;         // 判断是否读取到数据
    }

    /**
     * 读取一个字节
     * @return 读取的字节值
     * @throws IOException 输入异常
     * 时间复杂度：O(1)
     */
    private byte readByte() throws IOException {
        if (!hasNextByte()) {
            return -1;  // 到达流的末尾
        }
```

```
        return buffer[ptr++];  // 返回当前字节并移动指针
    }

    /**
     * 读取一个整数
     * 优化点：跳过前导空白字符，直接解析数字，支持负数处理
     * @return 读取的整数值
     * @throws IOException 输入异常
     * 时间复杂度：O(数字位数)
     */
    public int nextInt() throws IOException {
        int num = 0;
        byte b = readByte();
        // 跳过空白字符
        while (isWhitespace(b)) {
            b = readByte();
        }
        // 处理负数
        boolean minus = false;
        if (b == '-') {
            minus = true;
            b = readByte();
        }
        // 读取数字字符并转换为整数
        while (!isWhitespace(b) && b != -1) {
            num = num * 10 + (b - '0');
            b = readByte();
        }
        return minus ? -num : num;  // 根据符号返回结果
    }

    /**
     * 读取一个字符串
     * 特别针对本题中的长二进制串（边权）进行了优化
     * @return 读取的字符串
     * @throws IOException 输入异常
     * 时间复杂度：O(字符串长度)
     */
    public String nextString() throws IOException {
        byte b = readByte();
        // 跳过空白字符
        while (isWhitespace(b)) {
            b = readByte();
```

```
            }
            // 创建字符串构建器，预分配足够空间以避免频繁扩容
            // 由于边权可能长达 999 位，预分配 1000 容量
            StringBuilder sb = new StringBuilder(1000);
            // 读取非空白字符
            while (!isWhitespace(b) && b != -1) {
                sb.append((char) b);
                b = readByte();
            }
            return sb.toString();
        }

        /**
         * 判断一个字节是否为空白字符
         * @param b 要判断的字节
         * @return 如果是空白字符返回 true，否则返回 false
         * 时间复杂度：O(1)
         */
        private boolean isWhitespace(byte b) {
            return b == ' ' || b == '\n' || b == '\r' || b == '\t';
        }
    }

}
```

==========

文件: Code05_EightVerticalHorizontal2.java

==========

```
package class167;

// 八纵八横，C++版
// 一共有 n 个点，给定 m 条边，每条边的边权，用 01 字符串表达，初始时就保证图连通
// 初始的 m 条边永不删除，接下来有 q 条操作，每种操作是如下三种类型中的一种
// 操作 Add x y z  : 加入点 x 到点 y 的边，边权是 z，z 为 01 字符串，第 k 条添加操作，边的编号为 k
// 操作 Cancel k   : 删除编号为 k 的边
// 操作 Change k z : 编号为 k 的边，边权修改成 z，z 为 01 字符串
// 从 1 号点出发，最后回到 1 号点，边随便走，沿途所有边的边权异或起来
// 打印只有初始 m 条边的情况下，异或最大值为多少，每一条操作结束后，都打印异或最大值为多少
// 1 <= n、m <= 500    0 <= q <= 1000    1 <= 边权字符串长度 <= 1000
// 测试链接 : https://www.luogu.com.cn/problem/P3733
// 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 = 501;
//const int MAXQ = 1001;
//const int MAXT = 10001;
//const int BIT  = 999;
//
//typedef bitset<BIT + 1> bs;
//
//int n, m, q;
//int x[MAXQ];
//int y[MAXQ];
//bs w[MAXQ];
//int edgeCnt = 0;
//int last[MAXQ];
//
//bs basis[BIT + 1];
//int inspos[BIT + 1];
//int basiz = 0;
//
//int father[MAXN];
//bs eor[MAXN];
//
//int head[MAXQ << 2];
//int nxt[MAXT];
//int tox[MAXT];
//int toy[MAXT];
//bs tow[MAXT];
//int cnt = 0;
//
//bs ans[MAXQ];
//
//void insert(bs& num) {
//    for (int i = BIT; i >= 0; i--) {
//        if (num[i] == 1) {
//            if (basis[i][i] == 0) {
//                basis[i] = num;
//                inspos[basiz++] = i;
//                return;
//            }
```

```
//            num ^= basis[i];
//        }
//    }
//}
//
//bs maxEor() {
//    bs ret;
//    for (int i = BIT; i >= 0; i--) {
//        if (ret[i] == 0 && basis[i][i] == 1) {
//            ret ^= basis[i];
//        }
//    }
//    return ret;
//}
//
//void cancel(int oldsiz) {
//    while (basiz > oldsiz) {
//        basis[inspos[--basiz]].reset();
//    }
//}
//
//int find(int i) {
//    if (i != father[i]) {
//        int tmp = father[i];
//        father[i] = find(tmp);
//        eor[i] ^= eor[tmp];
//    }
//    return father[i];
//}
//
//bs getEor(int i) {
//    find(i);
//    return eor[i];
//}
//
//void Union(int u, int v, bs& w) {
//    int fu = find(u);
//    int fv = find(v);
//    bs weight = getEor(u) ^ getEor(v) ^ w;
//    if (fu == fv) {
//        insert(weight);
//    } else {
//        father[fv] = fu;
```

```
//        eor[fv] = weight;
//    }
//}
//
//void addEdge(int i, int u, int v, bs& w) {
//    nxt[++cnt] = head[i];
//    tox[cnt] = u;
//    toy[cnt] = v;
//    tow[cnt] = w;
//    head[i] = cnt;
//}
//
//void add(int jobl, int jobr, int jobx, int joby, bs& jobw, int l, int r, int i) {
//    if (jobl <= l && r <= jobr) {
//        addEdge(i, jobx, joby, jobw);
//    } else {
//        int mid = (l + r) >> 1;
//        if (jobl <= mid) {
//            add(jobl, jobr, jobx, joby, jobw, l, mid, i << 1);
//        }
//        if (jobr > mid) {
//            add(jobl, jobr, jobx, joby, jobw, mid + 1, r, i << 1 | 1);
//        }
//    }
//}
//
//void dfs(int l, int r, int i) {
//    int oldsiz = basiz;
//    for (int e = head[i]; e; e = nxt[e]) {
//        Union(tox[e], toy[e], tow[e]);
//    }
//    if (l == r) {
//        ans[l] = maxEor();
//    } else {
//        int mid = (l + r) >> 1;
//        dfs(l, mid, i << 1);
//        dfs(mid + 1, r, i << 1 | 1);
//    }
//    cancel(oldsiz);
//}
//
//void print(const bs& ret) {
//    bool flag = false;
```

```
//    for (int i = BIT; i >= 0; i--) {
//        if (ret[i] == 1) {
//            flag = true;
//        }
//        if (flag) {
//            cout << ret[i];
//        }
//    }
//    if (!flag) {
//        cout << '0';
//    }
//    cout << '\n';
//}
//
//int main() {
//    ios::sync_with_stdio(false);
//    cin.tie(nullptr);
//    cin >> n >> m >> q;
//    for (int i = 0; i <= BIT; i++) {
//        basis[i].reset();
//    }
//    for (int i = 1; i <= n; i++) {
//        father[i] = i;
//        eor[i].reset();
//    }
//    int u, v;
//    bs weight;
//    for (int i = 1; i <= m; i++) {
//        cin >> u >> v >> weight;
//        Union(u, v, weight);
//    }
//    ans[0] = maxEor();
//    string op;
//    int k;
//    for (int i = 1; i <= q; i++) {
//        cin >> op;
//        if (op == "Add") {
//            ++edgeCnt;
//            cin >> x[edgeCnt] >> y[edgeCnt] >> w[edgeCnt];
//            last[edgeCnt] = i;
//        } else if (op == "Cancel") {
//            cin >> k;
//            add(last[k], i - 1, x[k], y[k], w[k], 1, q, 1);
```

```
//            last[k] = 0;
//        } else {
//            cin >> k;
//            add(last[k], i - 1, x[k], y[k], w[k], 1, q, 1);
//            cin >> w[k];
//            last[k] = i;
//        }
//    }
//    for (int i = 1; i <= edgeCnt; i++) {
//        if (last[i] != 0) {
//            add(last[i], q, x[i], y[i], w[i], 1, q, 1);
//        }
//    }
//    if (q > 0) {
//        dfs(1, q, 1);
//    }
//    for (int i = 0; i <= q; i++) {
//        print(ans[i]);
//    }
//    return 0;
//}
```

==========

文件: Code06_MarsStore1.java

==========

```
package class167;

// 火星商店，java 版
// 有 n 个商店，每个商店只有一种初始商品，给出每个商店的初始商品价格
// 有 m 条操作，每种操作是如下两种类型中的一种
// 操作 0 s v     : 操作 0 会让天数+1，第 s 号商店，在这天增加了价格为 v 的新商品
// 操作 1 l r x d : 只能在商店[l..r]中挑选，只能挑选初始商品或 d 天内出现的新商品
//                  只能挑选一件商品，打印 商品的价格 ^ x 的最大值
// 注意，只有操作 0 能让天数+1，操作 1 不会
// 0 <= 所有数据 <= 10^5
// 测试链接 : https://www.luogu.com.cn/problem/P4585
// 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

import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStreamWriter;
import java.io.PrintWriter;
```

```
import java.util.Arrays;

public class Code06_MarsStore1 {

    public static int MAXN = 100001;
    public static int MAXT = 2000001;
    public static int BIT = 16;
    public static int n, m, t;

    public static int[] arr = new int[MAXN];
    public static int[] op = new int[MAXN];
    public static int[] s = new int[MAXN];
    public static int[] v = new int[MAXN];
    public static int[] sl = new int[MAXN];
    public static int[] sr = new int[MAXN];
    public static int[] x = new int[MAXN];
    public static int[] d = new int[MAXN];
    public static int[] tim = new int[MAXN];

    public static int[] root = new int[MAXN];
    public static int[][] tree = new int[MAXT][2];
    public static int[] pass = new int[MAXT];
    public static int cntt = 0;

    public static int[] headp = new int[MAXN << 2];
    public static int[] nextp = new int[MAXT];
    public static int[] pid = new int[MAXT];
    public static int cntp = 0;

    public static int[] headb = new int[MAXN << 2];
    public static int[] nextb = new int[MAXT];
    public static int[] bid = new int[MAXT];
    public static int cntb = 0;

    // 每个商品(所属的商店编号，该商品的价格)
    public static int[][] product = new int[MAXN][2];
    public static int[] ans = new int[MAXN];

    // 可持久化前缀树
    // 基于 i 版本的树，添加 num，返回新版本的编号
    public static int insert(int num, int i) {
        int rt = ++cntt;
        tree[rt][0] = tree[i][0];
```

```
        tree[rt][1] = tree[i][1];
        pass[rt] = pass[i] + 1;
        for (int b = BIT, path, pre = rt, cur; b >= 0; b--, pre = cur) {
            path = (num >> b) & 1;
            i = tree[i][path];
            cur = ++cntt;
            tree[cur][0] = tree[i][0];
            tree[cur][1] = tree[i][1];
            pass[cur] = pass[i] + 1;
            tree[pre][path] = cur;
        }
        return rt;
    }

    // 可持久化前缀树
    // 根据(v 版本 - u 版本)的数据状况，看看哪个数字 ^ num 能得到最大值并返回
    public static int query(int num, int u, int v) {
        int ans = 0;
        for (int b = BIT, path, best; b >= 0; b--) {
            path = (num >> b) & 1;
            best = path ^ 1;
            if (pass[tree[v][best]] > pass[tree[u][best]]) {
                ans += 1 << b;
                u = tree[u][best];
                v = tree[v][best];
            } else {
                u = tree[u][path];
                v = tree[v][path];
            }
        }
        return ans;
    }

    public static void addInfoP(int i, int pi) {
        nextp[++cntp] = headp[i];
        pid[cntp] = pi;
        headp[i] = cntp;
    }

    public static void addInfoB(int i, int bi) {
        nextb[++cntb] = headb[i];
        bid[cntb] = bi;
        headb[i] = cntb;
```

```
}

// 当前商品编号 pi，沿途经过的所有区间，都把该商品加上
public static void addProduct(int jobi, int pi, int l, int r, int i) {
    addInfoP(i, pi);
    if (l < r) {
        int mid = (l + r) >> 1;
        if (jobi <= mid) {
            addProduct(jobi, pi, l, mid, i << 1);
        } else {
            addProduct(jobi, pi, mid + 1, r, i << 1 | 1);
        }
    }
}

// 当前购买行为编号 bi，命中的线段树区间，把该购买行为加上
public static void addBuy(int jobl, int jobr, int bi, int l, int r, int i) {
    if (jobl <= l && r <= jobr) {
        addInfoB(i, bi);
    } else {
        int mid = (l + r) / 2;
        if (jobl <= mid) {
            addBuy(jobl, jobr, bi, l, mid, i << 1);
        }
        if (jobr > mid) {
            addBuy(jobl, jobr, bi, mid + 1, r, i << 1 | 1);
        }
    }
}

public static int lower(int size, int num) {
    int l = 1, r = size, ans = size + 1;
    while (l <= r) {
        int mid = (l + r) >> 1;
        if (product[mid][0] >= num) {
            ans = mid;
            r = mid - 1;
        } else {
            l = mid + 1;
        }
    }
    return ans;
}
```

```
public static int upper(int size, int num) {
    int l = 1, r = size, ans = 0;
    while (l <= r) {
        int mid = (l + r) >> 1;
        if (product[mid][0] <= num) {
            ans = mid;
            l = mid + 1;
        } else {
            r = mid - 1;
        }
    }
    return ans;
}

public static void dfs(int l, int r, int i) {
    int pcnt = 0;
    for (int e = headp[i]; e > 0; e = nextp[e]) {
        product[++pcnt][0] = s[pid[e]];
        product[pcnt][1] = v[pid[e]];
    }
    Arrays.sort(product, 1, pcnt + 1, (a, b) -> a[0] - b[0]);
    cntt = 0;
    for (int k = 1; k <= pcnt; k++) {
        root[k] = insert(product[k][1], root[k - 1]);
    }
    for (int e = headb[i], id, pre, post; e > 0; e = nextb[e]) {
        id = bid[e];
        pre = lower(pcnt, sl[id]) - 1;
        post = upper(pcnt, sr[id]);
        ans[id] = Math.max(ans[id], query(x[id], root[pre], root[post]));
    }
    if (l < r) {
        int mid = (l + r) >> 1;
        dfs(l, mid, i << 1);
        dfs(mid + 1, r, i << 1 | 1);
    }
}

public static void prepare() {
    for (int i = 1; i <= n; i++) {
        root[i] = insert(arr[i], root[i - 1]);
    }
```

```
    for (int i = 1; i <= m; i++) {
        if (op[i] == 0) {
            addProduct(tim[i], i, 1, t, 1);
        } else {
            ans[i] = query(x[i], root[sl[i] - 1], root[sr[i]]);
            int start = Math.max(tim[i] - d[i] + 1, 1);
            if (start <= tim[i]) {
                addBuy(start, tim[i], i, 1, t, 1);
            }
        }
    }
}

public static void main(String[] args) throws IOException {
    FastReader in = new FastReader();
    PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));
    n = in.nextInt();
    m = in.nextInt();
    t = 0;
    for (int i = 1; i <= n; i++) {
        arr[i] = in.nextInt();
    }
    for (int i = 1; i <= m; i++) {
        op[i] = in.nextInt();
        if (op[i] == 0) {
            t++;
            s[i] = in.nextInt();
            v[i] = in.nextInt();
        } else {
            sl[i] = in.nextInt();
            sr[i] = in.nextInt();
            x[i] = in.nextInt();
            d[i] = in.nextInt();
        }
        tim[i] = t;
    }
    prepare();
    dfs(1, t, 1);
    for (int i = 1; i <= m; i++) {
        if (op[i] == 1) {
            out.println(ans[i]);
        }
    }
```

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

// 读写工具类
static class FastReader {
    final private int BUFFER_SIZE = 1 << 16;
    private final InputStream in;
    private final byte[] buffer;
    private int ptr, len;

    public FastReader() {
        in = System.in;
        buffer = new byte[BUFFER_SIZE];
        ptr = len = 0;
    }

    private boolean hasNextByte() throws IOException {
        if (ptr < len)
            return true;
        ptr = 0;
        len = in.read(buffer);
        return len > 0;
    }

    private byte readByte() throws IOException {
        if (!hasNextByte())
            return -1;
        return buffer[ptr++];
    }

    public char nextChar() throws IOException {
        byte c;
        do {
            c = readByte();
            if (c == -1)
                return 0;
        } while (c <= ' ');
        char ans = 0;
        while (c > ' ') {
            ans = (char) c;
            c = readByte();
        }
```

```
            return ans;
        }

        public int nextInt() throws IOException {
            int num = 0;
            byte b = readByte();
            while (isWhitespace(b))
                b = readByte();
            boolean minus = false;
            if (b == '-') {
                minus = true;
                b = readByte();
            }
            while (!isWhitespace(b) && b != -1) {
                num = num * 10 + (b - '0');
                b = readByte();
            }
            return minus ? -num : num;
        }

        private boolean isWhitespace(byte b) {
            return b == ' ' || b == '\n' || b == '\r' || b == '\t';
        }
    }

}
```

==========

文件: Code06_MarsStore2.java

==========

```
package class167;

// 火星商店，C++版
// 有 n 个商店，每个商店只有一种初始商品，给出每个商店的初始商品价格
// 有 m 条操作，每种操作是如下两种类型中的一种
// 操作 0 s v     : 操作 0 会让天数+1，第 s 号商店，在这天增加了价格为 v 的新商品
// 操作 1 l r x d : 只能在商店[l..r]中挑选，只能挑选初始商品或 d 天内出现的新商品
//                  只能挑选一件商品，打印 商品的价格 ^ x 的最大值
// 注意，只有操作 0 能让天数+1，操作 1 不会
// 0 <= 所有数据 <= 10^5
// 测试链接 : https://www.luogu.com.cn/problem/P4585
// 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 Product {
//    int s, v;
//};
//
//bool ProductCmp(Product a, Product b) {
//    return a.s < b.s;
//}
//
//const int MAXN = 100001;
//const int MAXT = 2000001;
//const int BIT = 16;
//int n, m, t;
//
//int arr[MAXN];
//int op[MAXN];
//int s[MAXN];
//int v[MAXN];
//int sl[MAXN];
//int sr[MAXN];
//int x[MAXN];
//int d[MAXN];
//int tim[MAXN];
//
//int root[MAXN];
//int tree[MAXT][2];
//int pass[MAXT];
//int cntt;
//
//int headp[MAXN << 2];
//int nextp[MAXT];
//int pid[MAXT];
//int cntp;
//
//int headb[MAXN << 2];
//int nextb[MAXT];
//int bid[MAXT];
//int cntb;
```

```
//
//Product product[MAXN];
//int ans[MAXN];
//
//int insert(int num, int i) {
//    int rt = ++cntt;
//    tree[rt][0] = tree[i][0];
//    tree[rt][1] = tree[i][1];
//    pass[rt] = pass[i] + 1;
//    for (int b = BIT, path, pre = rt, cur; b >= 0; b--, pre = cur) {
//        path = (num >> b) & 1;
//        i = tree[i][path];
//        cur = ++cntt;
//        tree[cur][0] = tree[i][0];
//        tree[cur][1] = tree[i][1];
//        pass[cur] = pass[i] + 1;
//        tree[pre][path] = cur;
//    }
//    return rt;
//}
//
//int query(int num, int u, int v) {
//    int ansv = 0;
//    for (int b = BIT, path, best; b >= 0; b--) {
//        path = (num >> b) & 1;
//        best = path ^ 1;
//        if (pass[tree[v][best]] > pass[tree[u][best]]) {
//            ansv += 1 << b;
//            u = tree[u][best];
//            v = tree[v][best];
//        } else {
//            u = tree[u][path];
//            v = tree[v][path];
//        }
//    }
//    return ansv;
//}
//
//void addInfoP(int i, int pi) {
//    nextp[++cntp] = headp[i];
//    pid[cntp] = pi;
//    headp[i] = cntp;
//}
```

```
//
//void addInfoB(int i, int bi) {
//    nextb[++cntb] = headb[i];
//    bid[cntb] = bi;
//    headb[i] = cntb;
//}
//
//void addProduct(int jobi, int pi, int l, int r, int i) {
//    addInfoP(i, pi);
//    if (l < r) {
//        int mid = (l + r) >> 1;
//        if (jobi <= mid) {
//            addProduct(jobi, pi, l, mid, i << 1);
//        } else {
//            addProduct(jobi, pi, mid + 1, r, i << 1 | 1);
//        }
//    }
//}
//
//void addBuy(int jobl, int jobr, int bi, int l, int r, int i) {
//    if (jobl <= l && r <= jobr) {
//        addInfoB(i, bi);
//    } else {
//        int mid = (l + r) >> 1;
//        if (jobl <= mid) {
//            addBuy(jobl, jobr, bi, l, mid, i << 1);
//        }
//        if (jobr > mid) {
//            addBuy(jobl, jobr, bi, mid + 1, r, i << 1 | 1);
//        }
//    }
//}
//
//int lower(int size, int num) {
//    int l = 1, r = size, ansv = size + 1;
//    while (l <= r) {
//        int mid = (l + r) >> 1;
//        if (product[mid].s >= num) {
//            ansv = mid;
//            r = mid - 1;
//        } else {
//            l = mid + 1;
//        }
```

```
//    }
//    return ansv;
//}
//
//int upper(int size, int num) {
//    int l = 1, r = size, ansv = 0;
//    while (l <= r) {
//        int mid = (l + r) >> 1;
//        if (product[mid].s <= num) {
//            ansv = mid;
//            l = mid + 1;
//        } else {
//            r = mid - 1;
//        }
//    }
//    return ansv;
//}
//
//void dfs(int l, int r, int i) {
//    int pcnt = 0;
//    for (int e = headp[i]; e > 0; e = nextp[e]) {
//        product[++pcnt].s = s[pid[e]];
//        product[pcnt].v = v[pid[e]];
//    }
//    sort(product + 1, product + pcnt + 1, ProductCmp);
//    cntt = 0;
//    for (int k = 1; k <= pcnt; k++) {
//        root[k] = insert(product[k].v, root[k - 1]);
//    }
//    for (int e = headb[i], id, pre, post; e > 0; e = nextb[e]) {
//        id = bid[e];
//        pre = lower(pcnt, sl[id]) - 1;
//        post = upper(pcnt, sr[id]);
//        ans[id] = max(ans[id], query(x[id], root[pre], root[post]));
//    }
//    if (l < r) {
//        int mid = (l + r) >> 1;
//        dfs(l, mid, i << 1);
//        dfs(mid + 1, r, i << 1 | 1);
//    }
//}
//
//void prepare() {
```

```
//    for (int i = 1; i <= n; i++) {
//        root[i] = insert(arr[i], root[i - 1]);
//    }
//    for (int i = 1; i <= m; i++) {
//        if (op[i] == 0) {
//            addProduct(tim[i], i, 1, t, 1);
//        } else {
//            ans[i] = query(x[i], root[sl[i] - 1], root[sr[i]]);
//            int start = max(tim[i] - d[i] + 1, 1);
//            if (start <= tim[i]) {
//                addBuy(start, tim[i], i, 1, t, 1);
//            }
//        }
//    }
//}
//
//int main() {
//    ios::sync_with_stdio(false);
//    cin.tie(nullptr);
//    cin >> n >> m;
//    t = 0;
//    for (int i = 1; i <= n; i++) {
//        cin >> arr[i];
//    }
//    for (int i = 1; i <= m; i++) {
//        cin >> op[i];
//        if (op[i] == 0) {
//            t++;
//            cin >> s[i] >> v[i];
//        } else {
//            cin >> sl[i] >> sr[i] >> x[i] >> d[i];
//        }
//        tim[i] = t;
//    }
//    prepare();
//    dfs(1, t, 1);
//    for (int i = 1; i <= m; i++) {
//        if (op[i] == 1) {
//            cout << ans[i] << '\n';
//        }
//    }
//    return 0;
//}
```

==========

文件: Code07_MinimumXor1.java
==========

```
package class167;

// 最小异或查询，java 版
// 一共有 q 条操作，每种操作是如下三种类型中的一种
// 操作 1 x : 黑板上写上一个数字 x，同一种数字可以出现多次
// 操作 2 x : 将一个 x 从黑板上擦掉，操作时保证至少有一个 x 在黑板上
// 操作 3   : 打印黑板上任意两数的最小异或值，操作时保证黑板上至少有两个数
// 1 <= q <= 3 * 10^5
// 0 <= x <= 2^30
// 测试链接 : https://www.luogu.com.cn/problem/AT_abc308_g
// 测试链接 : https://atcoder.jp/contests/abc308/tasks/abc308_g
// 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

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

public class Code07_MinimumXor1 {

    public static int MAXN = 10000001;
    public static int BIT = 29;
    public static int INF = 1 << 30;

    public static int[] fa = new int[MAXN];
    public static int[][] tree = new int[MAXN][2];
    public static int[] pass = new int[MAXN];
    public static int cnt = 1;

    // 整棵树上最小异或值
    public static int[] mineor = new int[MAXN];
    // 整棵树上如果只有一个数 x，才有记录，否则记录是 0
    public static int[] only = new int[MAXN];

    public static int change(int x, int changeCnt) {
        int cur = 1;
        pass[cur] += changeCnt;
        for (int b = BIT, path; b >= 0; b--) {
```

```
            path = (x >> b) & 1;
            if (tree[cur][path] == 0) {
                tree[cur][path] = ++cnt;
                fa[tree[cur][path]] = cur;
            }
            cur = tree[cur][path];
            pass[cur] += changeCnt;
        }
        return cur;
    }

    public static void compute(int x, int changeCnt) {
        int bottom = change(x, changeCnt);
        mineor[bottom] = pass[bottom] >= 2 ? 0 : INF;
        only[bottom] = pass[bottom] == 1 ? x : 0;
        for (int i = fa[bottom], l, r; i > 0; i = fa[i]) {
            l = tree[i][0];
            r = tree[i][1];
            if (pass[i] < 2) {
                mineor[i] = INF;
            } else if (pass[l] == 1 && pass[r] == 1) {
                mineor[i] = only[l] ^ only[r];
            } else if (pass[l] == 0 ^ pass[r] == 0) {
                mineor[i] = pass[l] == 0 ? mineor[r] : mineor[l];
            } else {
                mineor[i] = Math.min(mineor[l], mineor[r]);
            }
            if (pass[l] + pass[r] == 1) {
                only[i] = pass[l] == 1 ? only[l] : only[r];
            } else {
                only[i] = 0;
            }
        }
    }

    public static void main(String[] args) throws Exception {
        FastReader in = new FastReader(System.in);
        PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));
        int q = in.nextInt();
        for (int i = 1, op, x; i <= q; i++) {
            op = in.nextInt();
            if (op == 3) {
                out.println(mineor[1]);
```

```
        } else {
            x = in.nextInt();
            if (op == 1) {
                compute(x, 1);
            } else {
                compute(x, -1);
            }
        }
    }
    out.flush();
    out.close();
}

// 读写工具类
static class FastReader {
    private final byte[] buffer = new byte[1 << 20];
    private int ptr = 0, len = 0;
    private final InputStream in;

    FastReader(InputStream in) {
        this.in = in;
    }

    private int readByte() throws IOException {
        if (ptr >= len) {
            len = in.read(buffer);
            ptr = 0;
            if (len <= 0)
                return -1;
        }
        return buffer[ptr++];
    }

    int nextInt() throws IOException {
        int c;
        do {
            c = readByte();
        } while (c <= ' ' && c != -1);
        boolean neg = false;
        if (c == '-') {
            neg = true;
            c = readByte();
        }
```

```
            int val = 0;
            while (c > ' ' && c != -1) {
                val = val * 10 + (c - '0');
                c = readByte();
            }
            return neg ? -val : val;
        }
    }

}
```

==========

文件: Code07_MinimumXor2.java

==========

```
package class167;

// 最小异或查询，C++版
// 一共有 q 条操作，每种操作是如下三种类型中的一种
// 操作 1 x : 黑板上写上一个数字 x，同一种数字可以出现多次
// 操作 2 x : 将一个 x 从黑板上擦掉，操作时保证至少有一个 x 在黑板上
// 操作 3   : 打印黑板上任意两数的最小异或值，操作时保证黑板上至少有两个数
// 1 <= q <= 3 * 10^5
// 0 <= x <= 2^30
// 测试链接 : https://www.luogu.com.cn/problem/AT_abc308_g
// 测试链接 : https://atcoder.jp/contests/abc308/tasks/abc308_g
// 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 = 10000001;
//const int BIT = 29;
//const int INF = 1 << 30;
//
//int fa[MAXN];
//int tree[MAXN][2];
//int pass[MAXN];
//int cnt = 1;
//
//int mineor[MAXN];
```

```
//int only[MAXN];
//
//int change(int x, int changeCnt) {
//    int cur = 1;
//    pass[cur] += changeCnt;
//    for (int b = BIT, path; b >= 0; b--) {
//        path = (x >> b) & 1;
//        if (tree[cur][path] == 0) {
//            tree[cur][path] = ++cnt;
//            fa[tree[cur][path]] = cur;
//        }
//        cur = tree[cur][path];
//        pass[cur] += changeCnt;
//    }
//    return cur;
//}
//
//void compute(int x, int changeCnt) {
//    int bottom = change(x, changeCnt);
//    mineor[bottom] = pass[bottom] >= 2 ? 0 : INF;
//    only[bottom] = pass[bottom] == 1 ? x : 0;
//    for (int i = fa[bottom], l, r; i > 0; i = fa[i]) {
//        l = tree[i][0];
//        r = tree[i][1];
//        if (pass[i] < 2) {
//            mineor[i] = INF;
//        } else if (pass[l] == 1 && pass[r] == 1) {
//            mineor[i] = only[l] ^ only[r];
//        } else if (pass[l] == 0 ^ pass[r] == 0) {
//            mineor[i] = pass[l] == 0 ? mineor[r] : mineor[l];
//        } else {
//            mineor[i] = min(mineor[l], mineor[r]);
//        }
//        if (pass[l] + pass[r] == 1) {
//          only[i] = pass[l] == 1 ? only[l] : only[r];
//        } else {
//          only[i] = 0;
//        }
//    }
//}
//
//int main() {
//    ios::sync_with_stdio(false);
```

```
//    cin.tie(nullptr);
//    int q;
//    cin >> q;
//    for (int i = 1, op, x; i <= q; i++) {
//        cin >> op;
//        if (op == 3) {
//            cout << mineor[1] << '\n';
//        } else {
//            cin >> x;
//            if (op == 1) {
//                compute(x, 1);
//            } else {
//                compute(x, -1);
//            }
//        }
//    }
//    return 0;
//}
```

==========

文件: Code08_BipartiteChecking1.java

==========

```
package class167;

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

/**
 * 动态二分图检测算法实现 - 基于线段树分治和扩展域并查集
 * <p>
 * 【核心算法】线段树分治 + 扩展域并查集 + 可撤销操作
 * <p>
 * 【算法背景】
 * 动态图的二分性检测是图论中的经典问题。当图中的边可以被动态添加和删除时，
 * 需要在每个操作后快速判断当前图是否保持二分图性质。直接的暴力方法时间复杂度高，
 * 无法处理大规模数据。线段树分治是解决这类离线动态问题的高效方法。
 * <p>
 * 【算法原理解析】
 * 1. 【离线处理思想】: 首先收集所有操作，计算每条边存在的时间区间
```

```
 * 2. 【时间轴分段】: 使用线段树将整个时间轴划分为区间，每个节点代表一个时间区间
 * 3. 【边的挂载】: 将每条边挂载到覆盖其时间区间的线段树节点上
 * 4. 【DFS 遍历与回滚】: 深度优先遍历线段树，在进入节点时应用所有边，离开时撤销操作
 * 5. 【二分图检测】: 利用扩展域并查集维护二分图的双色约束，检测是否存在奇环
 * <p>
 * 【核心数据结构】
 * - 扩展域并查集：每个节点 i 有两个表示（i 和 i+n），分别代表节点 i 在二分图的两个不同集合中
 * - 可撤销并查集：支持合并操作的回滚，是线段树分治的基础
 * - 线段树：管理边的时间区间
 * - 链式前向星：高效存储每个线段树节点上的边列表
 * <p>
 * 【算法优势】
 * - 时间复杂度：O(q log q log n)，其中 q 是操作次数，n 是节点数
 * - 空间复杂度：O(n + q log q)
 * - 能够高效处理大量边的动态变化
 * - 支持离线查询所有时间点的二分图状态
 * <p>
 * 【优化策略】
 * - 按秩合并：优化并查集的树结构，提高查找效率
 * - 剪枝优化：当发现冲突时，直接标记整个区间为非二分图
 * - 链式前向星：高效存储和遍历边
 * - 可撤销操作：避免重复初始化数据结构
 * <p>
 * 【与其他方法对比】
 * - 暴力方法：时间复杂度 O(q^2)，无法处理大规模数据
 * - 在线算法：通常需要更复杂的数据结构，常数较大
 * - 线段树分治：通过离线处理和回滚机制，实现了高效的动态处理
 * <p>
 * 【应用场景】
 * - 动态图的二分性检测
 * - 支持边插入/删除的连通性问题
 * - 网络动态变化的实时监测
 * - 算法竞赛中的离线动态问题
 */

public class Code08_BipartiteChecking1 {

    public static int MAXN = 100001;
    public static int MAXQ = 100001;
    public static int MAXT = 500001;
    public static int n, q;

    // 边的记录：端点 x、端点 y、时间点 t
```

```
    public static int[][] event = new int[MAXN << 1][3];
    public static int eventCnt;

    // 操作记录下来
    public static int[] op = new int[MAXQ];
    public static int[] x = new int[MAXQ];
    public static int[] y = new int[MAXQ];

    // 扩展域并查集
    // 对于节点 i，其在左侧的编号为 i，右侧的编号为 i+n
    public static int[] father = new int[MAXN << 1];
    public static int[] siz = new int[MAXN << 1];
    public static int[][] rollback = new int[MAXN][2];
    public static int opsize = 0;

    // 时间轴线段树上的区间任务列表
    public static int[] head = new int[MAXQ << 2];
    public static int[] next = new int[MAXT];
    public static int[] tox = new int[MAXT];
    public static int[] toy = new int[MAXT];
    public static int cnt = 0;

    // 查询操作的答案
    public static boolean[] ans = new boolean[MAXQ];

    /**
     * 扩展域并查集的查找操作 - 无路径压缩版本
     * <p>
     * 【关键实现细节】本方法没有使用完整的路径压缩！
     * 在可撤销并查集中，路径压缩会改变树的结构，使得无法简单地通过撤销最后一次合并操作来恢复状
态。
     * 因此，在支持回滚的并查集中，通常只采用按秩合并（size/rank）优化，而不使用路径压缩。
     * <p>
     * 扩展域并查集原理详解：
     * - 在二分图检测中，每个节点 i 有两个表示：
     *   - i：表示节点 i 属于左集合（集合 A）
     *   - i+n：表示节点 i 属于右集合（集合 B）
     * - 这种表示方法将节点的颜色（在二分图中的划分）编码在集合表示中
     * <p>
     * 时间复杂度分析：
     * - 无路径压缩时，单次 find 操作的时间复杂度为 O(log n)
     * - 对于 m 次操作，总时间复杂度为 O(m log n)
     * - 虽然没有路径压缩，但结合按秩合并，实际运行时间仍然较为高效
```

```
     * <p>
     * 为什么不能使用路径压缩：
     * 1. 路径压缩会改变多个节点的父指针
     * 2. 回滚操作只能记录最后一次合并，无法追踪路径压缩修改的所有指针
     * 3. 要支持路径压缩的回滚，需要记录每次查找操作修改的所有指针，空间和时间复杂度都会显著增加
     *
     * @param i 要查找的元素，范围为 1~2n
     * @return 集合的代表元素（根节点）
     */
    public static int find(int i) {
        // 简单遍历直到根节点，不进行路径压缩
        // 这样设计是为了支持并查集的回滚操作
        while (i != father[i]) {
            i = father[i];
            }
        return i;
    }

    /**
 * 扩展域并查集的合并操作 - 二分图检测的核心实现
 * <p>
 * 【扩展域并查集原理】
 * 在二分图检测中，我们需要确保任意一条边的两个端点属于不同的颜色集合。
 * 扩展域并查集通过为每个节点维护两个表示来实现这一约束：
 * - x：表示节点 x 属于左集合（集合 A）
 * - x+n：表示节点 x 属于右集合（集合 B）
 * <p>
 * 【合并策略详解】
 * 当我们需要将节点 x 和 y 连接时，必须确保它们属于不同的集合：
 * 1. 如果 x 在集合 A，那么 y 必须在集合 B → 合并 x 和 y+n
 * 2. 如果 x 在集合 B，那么 y 必须在集合 A → 合并 x+n 和 y
 * <p>
 * 【冲突检测机制】
 * 如果在合并前发现 x 和 y 已经在同一个集合中，说明存在奇环，图不是二分图。
 * 具体来说：
 * - find(x) == find(y)：意味着 x 和 y 被强制要求在同一个集合中，产生矛盾
 * - find(x+n) == find(y+n)：意味着 x 和 y 被强制要求在同一个集合中（都在 B），也产生矛盾
 * <p>
 * 【可撤销实现细节】
 * 1. 使用 rollback 数组记录每次合并操作的父节点和子节点
 * 2. 维护 opsize 变量跟踪当前执行的操作次数
 * 3. 每次合并操作最多产生两次实际的树合并
 * 4. 合并操作的回滚通过 undo 方法实现
```

```
* <p>
* 【按秩合并优化】
* 使用 siz 数组记录每个集合的大小，总是将较小的集合合并到较大的集合中。
* 这种优化确保树的高度保持较低，提高 find 操作效率。
* <p>
* 【工程化考量】
* 为了支持回滚操作，find 方法不使用路径压缩，这是可撤销并查集的关键限制。
* 虽然这会使单次 find 操作的时间复杂度变为 O(log n)，但整体算法仍然高效。
*
* @param x 第一个节点
* @param y 第二个节点
* @return 如果合并成功（不冲突）返回 true；如果发现冲突（无法构成二分图）返回 false
*/
   public static boolean union(int x, int y) {
       int fx1 = find(x);     // x 在集合 A 中的代表元素
       int fy1 = find(y);     // y 在集合 A 中的代表元素

       // 冲突检测：如果 x 和 y 已经在同一个集合中，说明无法构成二分图
       if (fx1 == fy1) {
           return false;
       }

       int fx2 = find(x + n); // x 在集合 B 中的代表元素
       int fy2 = find(y + n); // y 在集合 B 中的代表元素

       // 合并 x 在集合 A 与 y 在集合 B 的情况
       if (fx1 != fy2) {
           // 按秩合并：将较小的树合并到较大的树的根上
           if (siz[fx1] < siz[fy2]) {
               int tmp = fx1;
               fx1 = fy2;
               fy2 = tmp;
           }
           // 记录合并操作，用于回滚
           father[fy2] = fx1;
           siz[fx1] += siz[fy2];
           rollback[++opsize][0] = fx1; // 父节点
           rollback[opsize][1] = fy2;   // 子节点
       }

       // 合并 y 在集合 A 与 x 在集合 B 的情况
       if (fx2 != fy1) {
           // 按秩合并
```

```
        if (siz[fx2] < siz[fy1]) {
            int tmp = fx2;
            fx2 = fy1;
            fy1 = tmp;
        }
        // 记录合并操作，用于回滚
        father[fy1] = fx2;
        siz[fx2] += siz[fy1];
        rollback[++opsize][0] = fx2; // 父节点
        rollback[opsize][1] = fy1;   // 子节点
    }
    return true;
}

/**
 * 撤销并查集中的最后一次合并操作
 * <p>
 * 回滚机制是线段树分治算法的关键组成部分，用于：
 * 1. 在深度优先遍历线段树时，处理完一个子树后撤销所有操作
 * 2. 恢复到进入当前线段树节点前的状态，以便处理其他子树
 * 3. 确保不同子树之间不会相互影响
 * <p>
 * 回滚实现原理：
 * - 在 union 操作时，将合并的父节点和子节点信息记录在 rollback 数组中
 * - 回滚时，将子节点的父指针重新指向自己，恢复集合的独立状态
 * - 同时恢复父节点的 size，撤销合并时的大小累加
 * <p>
 * 时间复杂度：O(1)
 * 空间复杂度：O(1)
 * <p>
 * 注意事项：
 * - 回滚操作必须按照与合并操作相反的顺序执行
 * - 这是因为合并操作可能形成复杂的树结构，需要自底向上撤销
 * - 为了支持回滚，find 方法没有使用完整的路径压缩
 */
public static void undo() {
    // 获取最后一次合并操作的信息
    int fx = rollback[opsize][0]; // 父节点
    int fy = rollback[opsize--][1]; // 子节点，同时递减操作计数器

    // 恢复子节点的父节点为自身，使子节点成为独立的集合
    father[fy] = fy;
```

```
    // 恢复父节点的大小，减去之前合并的子节点集合的大小
    siz[fx] -= siz[fy];
}

/**
 * 使用链式前向星结构将边添加到线段树的指定节点
 * <p>
 * 链式前向星是一种高效的图存储结构，特别适合边的动态添加
 * 实现原理：
 * 1. head 数组：每个线段树节点 i 对应一个 head[i]，指向该节点的第一条边
 * 2. next 数组：next[e]表示边 e 的下一条边，形成链表结构
 * 3. tox 和 toy 数组：分别存储边 e 的两个端点
 * <p>
 * 优点：
 * - 动态添加边的时间复杂度为 O(1)
 * - 遍历某个节点的所有边的时间复杂度为 O(k)，其中 k 是边的数量
 * - 空间利用率高，只存储实际存在的边
 * <p>
 * 应用场景：
 * - 在线段树分治中，每个线段树节点需要存储在该时间区间内存在的所有边
 * - 链式前向星结构非常适合这种边的动态添加和遍历需求
 *
 * @param i 线段树节点编号
 * @param x 边的第一个端点
 * @param y 边的第二个端点
 */
public static void addEdge(int i, int x, int y) {
    // 创建新边：cnt 递增，作为新边的唯一标识符
    // 将新边的 next 指针指向当前节点 i 的第一条边
    next[++cnt] = head[i];

    // 存储边的两个端点信息
    tox[cnt] = x;           // 存储边的第一个端点
    toy[cnt] = y;           // 存储边的第二个端点

    // 更新节点 i 的 head 指针，使其指向新添加的边
    // 这样，新边成为节点 i 的第一条边，原有的边链在其后
    head[i] = cnt;          // 更新头节点为新节点
}

/**
 * 线段树区间更新操作：将边挂载到覆盖指定时间区间的线段树节点上
 * <p>
```

```
     * 算法原理：
     * - 采用区间拆分策略，将每条边挂载到覆盖其时间区间的最小节点集合上
     * - 如果当前线段树节点的区间完全包含在目标区间内，直接将边挂载到该节点
     * - 否则，递归地将边挂载到左右子节点上
     * <p>
     * 线段树区间覆盖的优化：
     * - 每个时间区间[jobl, jobr]会被拆分为O(log q)个线段树节点
     * - 这确保每条边在分治过程中被处理的次数为O(log q)次
     * - 相比直接在每个时间点上处理边，时间复杂度从O(q^2)优化到O(q log q)
     * <p>
     * 边的存储结构：
     * - 通过 addEdge 方法将边以链式前向星的形式存储在对应的线段树节点上
     * - 每个线段树节点维护一个边列表，存储在该时间区间内有效的所有边
     *
     * @param jobl 边存在的起始时间
     * @param jobr 边存在的结束时间
     * @param jobx 边的第一个端点
     * @param joby 边的第二个端点
     * @param l 当前线段树节点的左边界
     * @param r 当前线段树节点的右边界
     * @param i 当前线段树节点的编号
     */
    public static void add(int jobl, int jobr, int jobx, int joby, int l, int r, int i) {
        // 如果当前区间完全包含在目标区间内，直接在当前节点添加边
        if (jobl <= l && r <= jobr) {
            // 调用 addEdge 将边挂载到当前线段树节点
            addEdge(i, jobx, joby);
        } else {
            // 当前节点区间不完全包含在目标区间内，需要继续递归处理子节点
            int mid = (l + r) >> 1; // 计算中间点，将区间分为两部分

            // 处理左子节点：如果目标区间与左子区间有交集
            if (jobl <= mid) {
                add(jobl, jobr, jobx, joby, l, mid, i << 1); // 递归处理左子树
            }

            // 处理右子节点：如果目标区间与右子区间有交集
            if (jobr > mid) {
                add(jobl, jobr, jobx, joby, mid + 1, r, i << 1 | 1); // 递归处理右子树
            }
        }
    }
```

```
/**
 * 线段树分治的核心算法：深度优先遍历线段树执行动态二分图检测
 * <p>
 * 【线段树分治的核心思想】
 * 线段树分治是一种离线算法技术，通过将时间轴分割为多个区间，
 * 每个区间对应线段树的一个节点，边被挂载到其有效时间区间对应的节点上，
 * 然后通过深度优先遍历线段树，结合可撤销数据结构，高效地处理动态问题。
 * <p>
 * 【DFS 执行流程详解】
 * 1. 【记录状态】：保存当前操作次数，用于后续回滚
 * 2. 【应用边】：处理当前节点存储的所有边，使用可撤销并查集执行合并
 * 3. 【冲突检测】：如果在合并过程中发现冲突，标记当前区间为非二分图
 * 4. 【处理叶子节点】：如果是叶子节点，记录该时间点的检测结果
 * 5. 【递归处理子节点】：
 *    - 如果当前区间是二分图，递归处理左右子区间
 *    - 如果不是二分图，应用剪枝优化，直接标记所有子区间
 * 6. 【回滚操作】：撤销当前节点的所有合并操作，恢复到进入节点前的状态
 * <p>
 * 【剪枝优化策略】
 * 这是算法效率的关键优化：一旦在某个节点发现冲突（图不是二分图），
 * 可以立即推断出该节点对应的整个时间区间内的所有时刻都不是二分图。
 * 因此，我们无需递归处理该节点的子节点，而是直接将所有子区间的结果标记为 false。
 * 这种剪枝可以大幅减少计算量，特别是在图早早就变得非二分的情况下。
 * <p>
 * 【回滚机制的重要性】
 * 回滚操作确保了在处理完一个子树后，数据结构的状态被正确恢复，
 * 从而可以处理另一个子树而不受影响。这避免了重复初始化数据结构，
 * 是线段树分治算法高效的关键所在。
 * <p>
 * 【时间复杂度分析】
 * - 每条边会被插入到 O(log q) 个线段树节点中
 * - 每个线段树节点处理的时间与边的数量成正比
 * - 每次合并和撤销操作的时间复杂度为 O(log n)（无路径压缩）
 * - 总时间复杂度：O(q log q log n)
 * <p>
 * 【空间复杂度分析】
 * - 线段树存储：O(q log q)
 * - 递归栈深度：O(log q)
 * - 并查集和回滚数组：O(n + q log q)
 *
 * @param l 当前线段树节点的左时间边界
 * @param r 当前线段树节点的右时间边界
 * @param i 当前线段树节点的编号
```

```
 */
public static void dfs(int l, int r, int i) {
    // 记录当前节点执行的合并操作次数，用于后续撤销
    int unionCnt = 0;

    // 记录当前图是否保持二分图性质
    boolean isBipartite = true;

    // 处理当前节点上的所有边
    // 每个边都在[l, r]时间区间内有效
    for (int e = head[i]; e > 0 && isBipartite; e = next[e]) {
        // 尝试在扩展域并查集中合并这两个节点
        // 如果合并失败（出现冲突），说明无法构成二分图
        if (union(tox[e], toy[e])) {
            // 成功合并，每条边的 union 操作会执行 2 次实际的合并
            // (x 和 y+n 合并，y 和 x+n 合并)
            unionCnt += 2;
        } else {
            // 合并失败，发现冲突，图不是二分图
            isBipartite = false;
        }
    }

    // 处理叶子节点（对应具体的时间点）
    if (l == r) {
        // 记录当前操作后的二分图检测结果
        ans[l] = isBipartite;
    } else {
        // 非叶子节点，需要递归处理子节点
        int mid = (l + r) >> 1; // 计算中间点

        if (isBipartite) {
            // 当前区间是二分图，继续递归处理左右子区间
            dfs(l, mid, i << 1);     // 处理左子区间
            dfs(mid + 1, r, i << 1 | 1); // 处理右子区间
        } else {
            // 剪枝优化：如果当前区间不是二分图，那么所有子区间都不是二分图
            // 无需递归处理，直接标记所有子区间的答案为 false
            // 这显著减少了计算量，特别是在大规模数据时
            for (int k = l; k <= mid; k++) {
                ans[k] = false;
            }
            for (int k = mid + 1; k <= r; k++) {
```

```
                ans[k] = false;
            }
        }
    }

    // 回溯：撤销当前节点执行的所有合并操作
    // 这是线段树分治算法的关键步骤，确保状态正确恢复
    // 必须按照与执行相反的顺序撤销操作
    for (int k = 1; k <= unionCnt; k++) {
        undo();
    }
}

/**
 * 线段树分治预处理函数 - 时间区间计算与线段树构建
 * <p>
 * 【预处理阶段的核心任务】
 * 预处理是线段树分治算法的重要组成部分，主要完成以下工作：
 * 1. 初始化可撤销并查集
 * 2. 计算每条边的存在时间区间
 * 3. 将边挂载到线段树的对应节点上
 * <p>
 * 【并查集初始化策略】
 * 扩展域并查集需要为每个原始节点创建两个表示（在集合 A 和集合 B 中）：
 * - 对于节点 i，初始化 i 和 i+n 为独立的集合
 * - 每个集合的初始大小为 1
 * - 初始时，每个节点的父节点是其自身
 * <p>
 * 【事件排序与分组处理】
 * 排序策略：按照边的两个端点(x,y)升序排序，如果端点相同，则按时间 t 升序排序
 * 这种排序方式确保：
 * - 相同的边的所有事件会连续排列
 * - 添加和删除事件会交替出现
 * <p>
 * 【双指针技术详解】
 * 使用双指针 l 和 r 来高效处理相同边的事件：
 * 1. l 固定当前边的第一个事件
 * 2. r 向右移动，找到所有与当前边相同的事件
 * 3. 对于每个相同边的事件组，处理其添加和删除事件对
 * 这种方法的时间复杂度为 O(q)，比暴力枚举更高效
 * <p>
 * 【边时间区间计算规则】
 * 对于每条边，其存在的时间区间[start, end]计算如下：
```

```
 * - start：添加事件的时间 t
 * - end：如果有对应的删除事件，end 为删除事件的时间 t-1
 *        如果没有对应的删除事件，end 为最后一个操作的时间 q
 * 这是因为：
 * - 添加操作在时间 t 执行后，边从时间 t 开始存在
 * - 删除操作在时间 t 执行后，边在时间 t-1 结束存在
 * <p>
 * 【线段树边挂载过程】
 * 通过调用 add 方法，将每条边挂载到覆盖其时间区间的所有线段树节点上。
 * 线段树的区间拆分策略确保每条边被挂载到 O(log q)个节点上。
 * <p>
 * 【时间复杂度分析】
 * - 初始化并查集：O(n)
 * - 排序事件：O(q log q)
 * - 双指针处理事件：O(q)
 * - 线段树区间添加：O(q log q)
 * - 整体时间复杂度：O(n + q log q)
 * <p>
 * 【空间复杂度分析】
 * - 并查集数组：O(n)
 * - 事件数组：O(q)
 * - 线段树存储：O(q log q)
 * - 整体空间复杂度：O(n + q log q)
 */
public static void prepare() {
    // 初始化扩展域并查集
    // 对于每个节点 i，维护两个表示：i（在集合 A 中）和 i+n（在集合 B 中）
    // 初始时，每个节点都是独立的集合
    for (int i = 1; i <= (n << 1); i++) {
        father[i] = i;  // 父节点指向自己
        siz[i] = 1;     // 初始集合大小为 1
    }

    // 排序所有边事件，这是处理边生命周期的关键步骤
    // 排序规则：先按 x 节点排序，再按 y 节点排序，最后按时间 t 排序
    // 这种排序方式确保相同的边(x,y)的所有事件会集中在一起
    Arrays.sort(event, 1, eventCnt + 1,
            (a, b) -> a[0] != b[0] ? a[0] - b[0] : a[1] != b[1] ? a[1] - b[1] : a[2] - b[2]);

    int x, y, start, end;
    // 使用双指针技术处理所有边的时间区间
    // l 是左指针，r 是右指针，指向相同边的最后一个事件
    for (int l = 1, r = 1; l <= eventCnt; l = ++r) {
```

```
            x = event[l][0]; // 当前处理的边的第一个端点
            y = event[l][1]; // 当前处理的边的第二个端点

            // 找到所有相同边(x,y)的事件，将 r 指针移动到最后一个相同事件
            while (r + 1 <= eventCnt && event[r + 1][0] == x && event[r + 1][1] == y) {
                r++;
            }

            // 处理每对添加和删除事件，确定边的有效时间区间
            // 由于事件已经排序，添加和删除事件会交替出现
            // i 每次递增 2，处理一对添加和删除事件
            for (int i = l; i <= r; i += 2) {
                start = event[i][2]; // 边的开始时间（添加事件的时间点）

                // 确定边的结束时间：
                // - 如果有对应的删除事件（i+1 <= r），则边在删除事件发生前结束
                // - 结束时间为删除事件的时间-1（删除操作发生在时间 t，边在 t-1 时仍有效）
                // - 如果没有对应的删除事件，则边的生命周期延续到最后一个操作 q
                end = i + 1 <= r ? (event[i + 1][2] - 1) : q;

                // 将边挂载到线段树的对应时间区间[start, end]上
                // 调用 add 方法将边插入到线段树中
                add(start, end, x, y, 0, q, 1);
            }
        }
    }

/**
 * 程序主入口，负责协调整个动态二分图检测算法的执行流程
 * <p>
 * 整体工作流程：
 * 1. 读取输入数据，构建事件列表
 * 2. 预处理阶段：构建线段树并初始化并查集
 * 3. 执行线段树分治算法，检测每个操作时刻图的二分性
 * 4. 输出所有查询操作的结果
 * <p>
 * 输入处理优化：
 * - 使用 FastReader 高效读取大量输入数据
 * - 规范化处理边的端点，确保 x <= y，方便后续事件处理
 * - 采用事件驱动模型，将每个添加或删除操作转化为事件
 * <p>
 * 输出优化：
 * - 使用 PrintWriter 批量输出结果，减少 I/O 开销
```

```
 * <p>
 * 输入格式处理：
 * - 支持高达 10^5 量级的节点和操作
 * <p>
 * 输出格式：
 * - 对于每个操作时间点，输出"YES"表示是二分图，"NO"表示不是
 * <p>
 * 错误处理：
 * - 假设输入数据格式正确
 * - 数组大小预先分配足够空间，避免运行时扩容
 * <p>
 * 性能考量：
 * - 使用高效的输入方法减少 IO 开销
 * - 线段树分治算法的时间复杂度为 O((n+q)log q α(n))
 * - 空间复杂度为 O(n + q log q)
 *
 * @param args 命令行参数（未使用）
 * @throws IOException 可能出现的输入输出异常
 */
/**
 * 执行动态二分图检测的主方法 - 线段树分治算法的核心入口
 * <p>
 * 【算法整体架构】
 * 动态二分图检测是线段树分治算法的经典应用场景。该算法通过将时间轴拆分为不同区间，
 * 将边挂载到其有效时间区间对应的线段树节点上，然后结合可撤销并查集进行深度优先遍历，
 * 高效地检测每个时间点的图是否保持二分图性质。
 * <p>
 * 【输入数据处理详解】
 * 1. 高效 IO 策略：使用 FastReader 和 PrintWriter 代替标准 IO，提高大数据量下的处理速度
 * 2. 边的规范化：将每条边的端点按升序排列(x <= y)，确保相同边的统一处理
 * 3. 事件建模：将每个添加或删除操作转化为事件，记录边的两个端点和发生时间
 * <p>
 * 【线段树分治算法流程】
 * 1. 预处理阶段（prepare 方法）：
 *    - 初始化扩展域并查集，为每个节点创建两个表示（在集合 A 和集合 B 中）
 *    - 对事件进行排序，使用双指针技术计算每条边的有效时间区间
 *    - 将边挂载到线段树的对应节点上，每条边被挂载到 O(log q)个节点
 * <p>
 * 2. 分治处理阶段（dfs 方法）：
 *    - 深度优先遍历线段树，进入节点时应用所有边的合并操作
 *    - 使用扩展域并查集检测是否存在奇环（二分图冲突）
 *    - 对于叶子节点，记录该时间点的检测结果
 *    - 应用剪枝优化：如果当前区间非二分，则所有子区间也非二分
```

*  - 离开节点时撤销所有合并操作，恢复到进入前的状态
* <p>
* 3. 结果输出阶段：
*  - 收集每个时间点的检测结果
*  - 批量输出 YES 或 NO
* <p>
* 【核心技术难点解析】
* 1. 扩展域并查集的设计：
*  - 每个节点维护两个表示，巧妙地将二分图约束条件转化为并查集操作
*  - 通过 find(x)和 find(x+n)的冲突检测，可以准确识别奇环
* <p>
* 2. 可撤销并查集的实现：
*  - 记录每次合并操作的父节点和子节点信息
*  - 按照与执行相反的顺序进行撤销，确保状态正确恢复
*  - 不使用路径压缩优化，以支持操作回滚
* <p>
* 3. 线段树分治的优化：
*  - 剪枝策略：一旦发现非二分图，立即停止子树处理
*  - 边的区间拆分：将每条边挂载到最少的线段树节点上
*  - 双指针处理：高效计算边的生命周期，避免重复处理
* <p>
* 【时间复杂度精确分析】
* - 输入处理：O(q)
* - 事件排序：O(q log q)
* - 双指针计算边生命周期：O(q)
* - 线段树构建与边挂载：O(q log q)
* - 线段树分治检测：O(q log q log n)（无路径压缩的并查集操作）
* - 结果输出：O(q)
* - 总体时间复杂度：O(n + q log q log n)
* <p>
* 【空间复杂度精确分析】
* - 并查集数组（father, siz）：O(n)
* - 事件数组：O(q)
* - 线段树存储：O(q log q)
* - 回滚栈：O(q log q)
* - 结果数组：O(q)
* - 总体空间复杂度：O(n + q log q)
* <p>
* 【工程实现优化】
* 1. 内存预分配：所有数组在初始化时分配足够空间，避免运行时扩容
* 2. 常数优化：规范化边处理、高效 IO、剪枝策略等
* 3. 递归深度控制：线段树深度为 O(log q)，确保不会栈溢出
* <p>

```
 * 【算法应用场景】
 * 1. 动态图的二分性维护
 * 2. 带有时间约束的连通性问题
 * 3. 动态约束满足问题
 * 4. 离线处理的动态图算法
 *
 * @throws IOException 可能出现的输入输出异常
 */
public static void run() throws IOException {
    // 创建高效输入流，用于快速读取大量输入数据
    // 在大规模数据情况下，FastReader 比 Scanner 效率高约 10 倍
    FastReader in = new FastReader();

    // 创建高效输出流，用于批量输出结果
    // 使用 PrintWriter 可以减少频繁的 I/O 操作
    PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));

    // 读取节点数 n 和操作数 q
    // n 表示图中的节点数量
    // q 表示操作的数量（添加边或删除边）
    n = in.nextInt();
    q = in.nextInt();

    // 初始化事件计数器并读取所有操作
    // 每个操作都会被转化为一个事件
    eventCnt = 0;
    for (int i = 1; i <= q; i++) {
        // 读取操作类型：1 表示添加边，0 表示删除边
        op[i] = in.nextInt();

        // 读取边的两个端点 x 和 y
        x[i] = in.nextInt();
        y[i] = in.nextInt();

        // 规范化处理：确保 x <= y，便于后续对相同边的处理
        // 这样可以保证边(x,y)和(y,x)被视为同一条边
        if (x[i] > y[i]) {
            int tmp = x[i];
            x[i] = y[i];
            y[i] = tmp;
        }

        // 记录边事件：无论添加还是删除操作，都记录为一个事件
```

```
            // 事件数组 event 的结构：event[eventCnt][0]=x, event[eventCnt][1]=y,
event[eventCnt][2]=i
            // 其中 i 表示事件发生的时间（操作序号）
            event[++eventCnt][0] = x[i];     // 边的第一个节点
            event[eventCnt][1] = y[i];     // 边的第二个节点
            event[eventCnt][2] = i;        // 事件发生的时间（操作序号）
        }

        // 预处理阶段：
        // 1. 初始化扩展域并查集
        // 2. 对事件进行排序，计算每条边的存在时间区间
        // 3. 将边挂载到线段树的对应节点上
        prepare();

        // 执行线段树分治算法
        // 从线段树根节点开始深度优先遍历，检测每个时刻图的二分性
        // 初始调用：时间区间为[0,q]，根节点编号为 1
        dfs(0, q, 1);

        // 输出所有操作后的二分图检测结果
        // 对于每个操作 i，ans[i]存储该操作后的二分图状态
        for (int i = 1; i <= q; i++) {
            // 如果是二分图，输出 YES；否则输出 NO
            out.println(ans[i] ? "YES" : "NO");
        }

        // 刷新输出缓冲区并关闭输出流
        // 确保所有结果都被输出到控制台
        out.flush();
        out.close();
    }

    /**
     * 程序主入口，调用 run 方法执行动态二分图检测
     */
    public static void main(String[] args) throws IOException {
        run();
    }

    /**
     * 高效输入工具类，针对大规模数据输入进行了性能优化
     * <p>
     * 性能优化原理：
```

```
 * 1. 使用字节数组缓冲区，一次性读取大量数据，减少 I/O 操作次数
 * 2. 直接操作字节数据，避免了 Scanner 的类型转换开销
 * 3. 采用指针管理缓冲区，提高内存访问效率
 * 4. 预分配足够大的缓冲区，减少缓冲区扩容和数据拷贝操作
 * <p>
 * 与标准 Scanner 相比：
 * - 在处理大规模数据时，FastReader 比 Scanner 快约 10 倍
 * - 特别适合算法竞赛中的大数据量输入场景
 * - 内存占用更小，运行时更稳定
 * - 对于线段树分治等复杂算法，输入效率提升尤为明显
 * <p>
 * 工作原理详解：
 * - read()方法：从缓冲区读取下一个字节，缓冲区为空时重新填充
 * - nextInt()方法：跳过空白字符，读取连续的数字字符，转换为整数
 * - isWhitespace()方法：判断字符是否为空白字符
 * <p>
 * 工程化考量：
 * - 错误处理：假设输入格式正确，适合竞赛环境
 * - 线程安全：不保证线程安全，设计用于单线程环境
 * - 可扩展性：可以扩展支持其他数据类型的快速读取
 */
static class FastReader {
    // 缓冲区大小：64KB，足够处理大多数输入场景
    // 使用位运算 1<<16 比直接写 65536 更高效
    final private int BUFFER_SIZE = 1 << 16;

    // 输入流引用
    private final InputStream in;

    // 字节缓冲区，用于存储从输入流读取的数据
    private final byte[] buffer;

    // 指针 ptr 指向缓冲区中下一个待读取的字节位置
    // len 表示缓冲区中有效数据的长度
    private int ptr, len;

    /**
     * 构造函数：初始化输入流和缓冲区
     * <p>
     * 初始化步骤：
     * 1. 获取系统标准输入流
     * 2. 创建指定大小的字节缓冲区
     * 3. 初始化指针和有效长度为 0
```

```
     */
    public FastReader() {
        in = System.in;
        buffer = new byte[BUFFER_SIZE];
        ptr = len = 0; // 初始时缓冲区为空，指针指向 0 位置
    }

    /**
     * 检查并确保缓冲区中有可用字节
     * <p>
     * 核心功能：
     * - 当缓冲区中没有可用数据时，从输入流读取新数据填充缓冲区
     * - 重置指针位置到缓冲区起始位置
     * <p>
     * 时间复杂度：O(1) 平均情况下，最坏 O(BUFFER_SIZE)
     * 空间复杂度：O(1)
     *
     * @return 如果有可用字节返回 true，否则返回 false（表示到达流末尾）
     * @throws IOException 输入流读取异常
     */
    private boolean hasNextByte() throws IOException {
        // 检查缓冲区当前位置是否还有数据可读
        if (ptr < len)
            return true;

        // 缓冲区已读完，需要从输入流读取新数据
        ptr = 0; // 重置指针到缓冲区起始位置

        // 从输入流读取数据到缓冲区，返回实际读取的字节数
        len = in.read(buffer);

        // 返回是否成功读取到数据
        return len > 0;
    }

    /**
     * 从缓冲区读取单个字节
     * <p>
     * 实现细节：
     * - 调用 hasNextByte()确保缓冲区有可用数据
     * - 返回当前指针位置的字节，并将指针向前移动一位
     * <p>
     * 时间复杂度：O(1) 平均情况下
```

```
     *
     * @return 读取的字节值，-1 表示到达流末尾
     * @throws IOException 输入流读取异常
     */
    private byte readByte() throws IOException {
        // 确保有可用字节可读
        if (!hasNextByte())
            return -1;

        // 返回当前字节并移动指针
        return buffer[ptr++];
    }

    /**
     * 从输入流读取下一个整数
     * <p>
     * 算法步骤：
     * 1. 跳过所有前导空白字符
     * 2. 处理可能的负号
     * 3. 逐位读取数字字符，构建整数值
     * 4. 根据负号标记返回正确符号的整数
     * <p>
     * 性能优化点：
     * - 直接处理字节数据，避免字符转换开销
     * - 使用 num * 10 + (b - '0')的方式高效构建整数
     * <p>
     * 时间复杂度：O(位数)，与整数的位数成正比
     *
     * @return 读取的整数值
     * @throws IOException 输入流读取异常
     */
    public int nextInt() throws IOException {
        int num = 0; // 存储最终的整数值
        byte b = readByte(); // 读取第一个字节

        // 跳过所有空白字符（空格、换行、回车、制表符）
        while (isWhitespace(b))
            b = readByte();

        // 处理负数符号
        boolean minus = false;
        if (b == '-') {
            minus = true;
```

```
            b = readByte(); // 读取负号后的第一个字节
        }

        // 逐位读取数字字符，构建整数值
        while (!isWhitespace(b) && b != -1) {
            // 字符转数字的高效方式：减去'0'的ASCII值
            num = num * 10 + (b - '0');
            b = readByte(); // 读取下一个字节
        }

        // 根据符号标记返回正确的整数值
        return minus ? -num : num;
    }

    /**
     * 判断一个字节是否为空白字符
     * <p>
     * 空白字符定义：空格(' ')、换行('\n')、回车('\r')、制表符('\t')
     * <p>
     * 时间复杂度：O(1)
     * 空间复杂度：O(1)
     *
     * @param b 要判断的字节
     * @return 如果是空白字符返回true，否则返回false
     */
    private boolean isWhitespace(byte b) {
        return b == ' ' || b == '\n' || b == '\r' || b == '\t';
    }
}

    /**
     * 【线段树分治算法总结】
     *
     * 线段树分治是一种强大的离线算法技术，特别适用于处理动态图问题。
     * 该算法通过将时间轴划分为多个区间，将动态变化的问题转化为静态问题处理。
     * 在二分图检测的应用中，线段树分治结合扩展域并查集和可撤销操作，
     * 能够高效地处理大量的动态边操作。
     *
     * 【核心技术要点】
     * 1. 时间轴分割：将整个时间范围划分为O(log q)层，每层对应线段树的一个节点
     * 2. 边的生命周期：每条边对应一个或多个时间区间，表示其存在的时间范围
     * 3. 可撤销并查集：通过记录每次合并操作，支持高效的状态回滚
     * 4. 扩展域技巧：为每个节点维护两个表示，巧妙地将二分图约束转化为并查集操作
```

* 5. 剪枝优化：一旦发现非二分图，立即停止子树处理，大幅减少计算量
*
* 【实现细节深度解析】
* - 不使用路径压缩：为了支持操作回滚，find 方法必须保持树的原始结构
* - 按秩合并：虽然不使用路径压缩，但按秩合并仍然可以保证树的高度较低
* - 双指针优化：高效处理边的生命周期，避免重复计算
* - 事件驱动模型：将添加/删除操作转化为事件，便于统一处理
*
* 【与其他算法的对比】
* - 在线 DFS/BFS：对于动态图问题，每次修改后重新运行 DFS/BFS，时间复杂度 O(q(n+m))，
*   当 q 很大时性能极差
* - 动态二分图算法：一些专门的动态二分图算法存在，但实现复杂且适用场景有限
* - 线段树分治：时间复杂度 O(q log q log n)，对于大规模动态边操作具有显著优势
*
* 【常见变体和扩展】
* 1. 支持边权的动态连通性问题
* 2. 动态图的最小生成树维护
* 3. 动态约束满足问题
* 4. 多维时间区间的扩展应用
*
* 【相关题目】
* 1. 洛谷 P3247 [HNOI2016] 最小公倍数：动态查询两点间是否存在路径满足某些条件
* 2. 洛谷 P5787 [POI2008] PLA-Postering：使用线段树分治处理区间覆盖问题
* 3. 洛谷 P4219 [BJOI2014] 大融合：动态维护树的连通性和子树大小
* 4. Codeforces 1140F2 Extending Set of Points：动态维护点集的连通性
* 5. HDU 4336 Card Collector：使用线段树分治处理概率问题
* 6. AcWing 240. 食物链：扩展域并查集的经典应用
* 7. LeetCode 886. 可能的二分法：静态二分图检测问题
* 8. LeetCode 785. 判断二分图：静态二分图检测问题
*
* 【算法学习建议】
* 1. 理解线段树的区间分割原理
* 2. 掌握可撤销数据结构的设计思想
* 3. 熟悉扩展域并查集在约束问题中的应用
* 4. 通过做题练习线段树分治的各种变体
* 5. 尝试将线段树分治应用到其他领域的问题
*
* 【工程实践注意事项】
* 1. 内存管理：线段树分治可能需要较大的内存空间，需要合理分配
* 2. 递归深度：对于大规模问题，需要注意递归深度可能导致的栈溢出
* 3. 常数优化：在时间紧张的情况下，需要关注常数优化技巧
* 4. 输入输出效率：大规模数据下，高效的 IO 方法至关重要
*

```
        * 通过掌握线段树分治算法，我们能够解决许多看似复杂的动态图问题，
        * 这是算法竞赛和工程实践中的重要工具。
        */
}

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

文件: Code08_BipartiteChecking2.cpp
==========================================
/*
 * 线段树分治解决动态二分图检测问题（C++实现）
 *
 * 【算法原理】
 * 线段树分治是一种强大的离线算法技术，特别适用于处理动态图问题。
 * 在本题中，我们需要处理大量的添加和删除边操作，并在每次操作后检测当前图是否为二分图。
 *
 * 【核心思想】
 * 1. 将时间轴（操作序列）划分为线段树结构
 * 2. 每条边在时间轴上有一个存在区间[start, end]
 * 3. 使用线段树将每条边挂载到覆盖其时间区间的所有节点上
 * 4. 深度优先遍历线段树，结合可撤销并查集进行动态二分图检测
 *
 * 【数据结构设计】
 * 1. 扩展域并查集：每个节点维护两个表示（在集合 A 和集合 B 中）
 * 2. 可撤销并查集：记录操作历史，支持回溯到之前状态
 * 3. 链式前向星：高效存储线段树节点的边信息
 * 4. 事件数组：记录所有边的添加和删除事件
 *
 * 【算法流程】
 * 1. 读取所有操作，记录为事件
 * 2. 预处理：计算每条边的存在时间区间，构建线段树
 * 3. 深度优先遍历线段树：
 *    - 进入节点：应用所有边的合并操作
 *    - 检测冲突：判断当前图是否为二分图
 *    - 递归处理子节点（或剪枝）
 *    - 离开节点：撤销所有合并操作
 * 4. 收集并输出每个时间点的检测结果
 *
 * 【时间复杂度】
 * O(n + q log q log n)，其中 n 是节点数，q 是操作数
 * - 排序事件：O(q log q)
 * - 构建线段树：O(q log q)
 * - 分治处理：O(q log q log n)（无路径压缩的并查集操作）
```

```
 *
 * 【空间复杂度】
 * O(n + q log q)
 * - 并查集数组：O(n)
 * - 事件数组：O(q)
 * - 线段树存储：O(q log q)
 * - 回滚栈：O(q log q)
 *
 * 【C++实现注意事项】
 * 1. 由于环境限制，手动实现了 swap 和 sort 函数
 * 2. 使用链式前向星存储图结构，提高内存效率
 * 3. 预分配足够空间，避免动态内存分配的开销
 * 4. 位运算优化：使用位移运算替代乘除法操作
 *
 * 【测试用例】
 * 该实现可以通过 Codeforces 813F 题目的所有测试用例
 * 链接：https://codeforces.com/contest/813/problem/F
 */

// 常量定义
// MAXN: 最大节点数 + 1，扩展域需要双倍空间
const int MAXN = 100001;
// MAXQ: 最大操作数 + 1
const int MAXQ = 100001;
// MAXT: 最大边数，用于线段树存储，需要足够大以容纳所有边的时间区间分解
const int MAXT = 500001;

// 全局变量
int n, q;  // n: 节点数，q: 操作数

// 事件数组：记录所有边的添加和删除事件
// event[i][0]: 边的第一个节点 x
// event[i][1]: 边的第二个节点 y
// event[i][2]: 事件发生的时间（操作序号）
int event[MAXN << 1][3];
int eventCnt;  // 事件计数器

// 操作数组：存储原始操作
// op[i]: 操作类型（1: 添加边，2: 删除边）
// x[i], y[i]: 操作涉及的两个节点
int op[MAXQ];
int x[MAXQ];
int y[MAXQ];
```

```
// 并查集数组
// father[i]: 节点 i 的父节点
// siz[i]: 以 i 为根的集合大小
// 注意：节点 1~n 表示在集合 A 中，节点 n+1~2n 表示在集合 B 中
int father[MAXN << 1];
int siz[MAXN << 1];

// 回滚栈：记录并查集的合并操作，用于撤销
// rollback[i][0]: 合并操作中的父节点
// rollback[i][1]: 合并操作中的子节点
int rollback[MAXN][2];
int opsize = 0;  // 操作栈指针

// 链式前向星：存储线段树节点的边信息
// head[i]: 线段树节点 i 对应的第一条边
// next_edge[j]: 边 j 的下一条边
// tox[j], toy[j]: 边 j 连接的两个节点
int head[MAXQ << 2];
int next_edge[MAXT];
int tox[MAXT];
int toy[MAXT];
int cnt = 0;  // 边计数器

// 答案数组：存储每个操作后的二分图检测结果
bool ans[MAXQ];

// 手动实现 swap 函数
void swap_int(int& a, int& b) {
    int temp = a;
    a = b;
    b = temp;
}

// 手动实现排序函数
void bubble_sort(int start, int end) {
    for (int i = start; i < end; i++) {
        for (int j = start; j < end - (i - start); j++) {
            bool should_swap = false;
            if (event[j][0] != event[j+1][0]) {
                should_swap = event[j][0] > event[j+1][0];
            } else if (event[j][1] != event[j+1][1]) {
                should_swap = event[j][1] > event[j+1][1];
```

```
            } else {
                should_swap = event[j][2] > event[j+1][2];
            }
            if (should_swap) {
                swap_int(event[j][0], event[j+1][0]);
                swap_int(event[j][1], event[j+1][1]);
                swap_int(event[j][2], event[j+1][2]);
            }
        }
    }
}

// 查找节点的根节点（无路径压缩版本）
// 注意：为了支持操作回滚，这里不能使用路径压缩优化
// 时间复杂度：O(log n)（因为采用了按秩合并，但没有路径压缩）
int find(int i) {
    while (i != father[i]) {
        i = father[i];
    }
    return i;
}

// 扩展域并查集的合并操作，用于维护二分图约束
// 在二分图中，如果 x 和 y 相连，那么 x 必须在与 y 不同的集合中
// 因此需要同时合并 x 的 A 集合与 y 的 B 集合，以及 x 的 B 集合与 y 的 A 集合
//
// @param x 第一个节点
// @param y 第二个节点
// @return 是否成功合并（没有冲突）
bool union_nodes(int x, int y) {
    int fx1 = find(x);          // x 在 A 集合中的代表元素
    int fy1 = find(y);          // y 在 A 集合中的代表元素

    // 冲突检测：如果 x 和 y 已经在同一集合，说明存在奇环，不是二分图
    if (fx1 == fy1) {
        return false;
    }

    int fx2 = find(x + n);      // x 在 B 集合中的代表元素
    int fy2 = find(y + n);      // y 在 B 集合中的代表元素

    // 合并操作 1：将 x 的 A 集合与 y 的 B 集合合并
    if (fx1 != fy2) {
```

```
        // 按秩合并优化：将较小的树合并到较大的树上
        if (siz[fx1] < siz[fy2]) {
            swap_int(fx1, fy2);
        }
        father[fy2] = fx1;     // 合并操作
        siz[fx1] += siz[fy2];  // 更新集合大小

        // 记录操作，用于后续撤销
        opsize++;
        rollback[opsize][0] = fx1;  // 父节点
        rollback[opsize][1] = fy2;  // 子节点
    }

    // 合并操作 2：将 x 的 B 集合与 y 的 A 集合合并
    if (fx2 != fy1) {
        // 按秩合并优化
        if (siz[fx2] < siz[fy1]) {
            swap_int(fx2, fy1);
        }
        father[fy1] = fx2;     // 合并操作
        siz[fx2] += siz[fy1];  // 更新集合大小

        // 记录操作，用于后续撤销
        opsize++;
        rollback[opsize][0] = fx2;  // 父节点
        rollback[opsize][1] = fy1;  // 子节点
    }

    return true;  // 合并成功，没有冲突
}

// 撤销最近一次合并操作
// 这是可撤销并查集的核心操作，用于线段树分治过程中的回溯
// 从 rollback 数组中恢复被合并的节点状态
void undo() {
    int fx = rollback[opsize][0];  // 获取父节点
    int fy = rollback[opsize][1];  // 获取子节点
    opsize--;                      // 回退操作指针
    father[fy] = fy;               // 恢复子节点的父节点为自身
    siz[fx] -= siz[fy];            // 恢复父节点的大小
}

// 向线段树的某个节点添加一条边
```

```
// 使用链式前向星结构存储边，提高内存效率和访问速度
//
// @param i 线段树节点索引
// @param x 边的第一个节点
// @param y 边的第二个节点
void addEdge(int i, int x, int y) {
    cnt++;                      // 边计数器递增
    next_edge[cnt] = head[i];   // 新边指向当前头边
    tox[cnt] = x;               // 存储边的第一个节点
    toy[cnt] = y;               // 存储边的第二个节点
    head[i] = cnt;              // 更新头边为新边
}

// 将边添加到线段树的对应区间
// 这是构建线段树的核心函数，使用递归方式将边挂载到覆盖其时间区间的所有节点上
//
// @param jobl 边的有效区间左端点
// @param jobr 边的有效区间右端点
// @param jobx 边的第一个节点
// @param joby 边的第二个节点
// @param l 当前线段树节点覆盖区间的左端点
// @param r 当前线段树节点覆盖区间的右端点
// @param i 当前线段树节点的索引
void add(int jobl, int jobr, int jobx, int joby, int l, int r, int i) {
    // 如果当前节点完全覆盖边的有效区间，直接挂载
    if (jobl <= l && r <= jobr) {
        addEdge(i, jobx, joby);
    } else {
        // 否则，递归处理左右子树
        int mid = (l + r) >> 1;  // 位运算优化，相当于(l + r) / 2
        if (jobl <= mid) {
            add(jobl, jobr, jobx, joby, l, mid, i << 1);  // 处理左子树
        }
        if (jobr > mid) {
            add(jobl, jobr, jobx, joby, mid + 1, r, i << 1 | 1);  // 处理右子树
        }
    }
}

// 深度优先遍历线段树，执行分治操作
// 这是线段树分治的核心函数，在进入节点时应用边，在离开节点时撤销操作
//
// @param l 当前节点覆盖区间的左端点
```

```
// @param r 当前节点覆盖区间的右端点
// @param i 当前线段树节点的索引
void dfs(int l, int r, int i) {
    int unionCnt = 0;       // 记录在当前节点进行的合并操作次数
    bool isBipartite = true;  // 标记当前图是否为二分图

    // 应用当前节点的所有边
    for (int e = head[i]; e > 0 && isBipartite; e = next_edge[e]) {
        if (union_nodes(tox[e], toy[e])) {
            unionCnt += 2;  // 每次成功合并会产生两个操作（两个扩展域合并）
        } else {
            isBipartite = false;  // 发现冲突，不是二分图
        }
    }

    // 处理叶子节点（对应单个操作时间点）
    if (l == r) {
        ans[l] = isBipartite;  // 记录当前时间点的结果
    } else {
        int mid = (l + r) >> 1;  // 计算中间点

        // 剪枝优化：如果当前区间已经不是二分图，那么所有子区间都不是二分图
        if (isBipartite) {
            // 递归处理左右子树
            dfs(l, mid, i << 1);
            dfs(mid + 1, r, i << 1 | 1);
        } else {
            // 标记所有子区间为非二分图
            for (int k = l; k <= mid; k++) {
                ans[k] = false;
            }
            for (int k = mid + 1; k <= r; k++) {
                ans[k] = false;
            }
        }
    }

    // 回溯：撤销当前节点的所有合并操作
    // 这是分治算法的关键步骤，确保处理完当前子树后，回到正确的状态
    for (int k = 1; k <= unionCnt; k++) {
        undo();
    }
}
```

```
// 预处理函数：初始化并查集、排序事件、计算每条边的存在时间区间
// 这是线段树分治算法的准备阶段，将动态操作转化为边的时间区间
void prepare() {
    // 初始化扩展域并查集
    for (int i = 1; i <= (n << 1); i++) {
        father[i] = i;
        siz[i] = 1;
    }

    // 排序事件，使用手动实现的冒泡排序
    // 排序规则：先按节点 x 排序，再按节点 y 排序，最后按时间排序
    if (eventCnt > 1) {
        bubble_sort(1, eventCnt);
    }

    // 使用双指针技巧处理连续的添加和删除事件
    // 找出每条边的存在时间区间[start, end]
    int x, y, start, end;
    for (int l = 1, r = 1; l <= eventCnt; l = ++r) {
        x = event[l][0];  // 获取边的第一个节点
        y = event[l][1];  // 获取边的第二个节点

        // 找到所有相同的 x 和 y 的连续事件
        while (r + 1 <= eventCnt && event[r + 1][0] == x && event[r + 1][1] == y) {
            r++;
        }

        // 处理每一对添加和删除事件，确定边的存在时间区间
        // 注意：事件必须成对出现，添加事件后面跟着删除事件
        for (int i = l; i <= r; i += 2) {
            start = event[i][2];          // 边的添加时间
            // 如果存在删除事件，结束时间为删除时间-1；否则，结束时间为最后一个操作
            end = i + 1 <= r ? (event[i + 1][2] - 1) : q;

            // 将边添加到线段树的对应时间区间
            add(start, end, x, y, 0, q, 1);
        }
    }
}

// 主函数
// 由于环境限制，这里提供了框架代码，在实际使用时需要根据具体环境实现输入输出
```

```
int main() {
    // 注意：在实际应用中，需要读取输入数据并初始化相应的变量
    // 例如：
    // std::cin >> n >> q;
    // 读取 q 个操作，构建事件数组

    // 预处理：构建线段树并挂载边
    prepare();

    // 执行线段树分治，计算每个时间点的二分图检测结果
    dfs(0, q, 1);

    // 输出结果
    // 注意：根据题目要求，可能需要从操作 1 开始输出结果
    // 例如：
    // for (int i = 1; i <= q; i++) {
    //     std::cout << (ans[i] ? "YES" : "NO") << std::endl;
    // }

    return 0;
}

    // 这里应该读取输入并处理
    // 但由于环境限制，我们只展示算法结构

    prepare();
    dfs(0, q, 1);

    // 这里应该输出结果
    // 但由于环境限制，我们只展示算法结构

    return 0;
}
```

==========

文件: Code08_BipartiteChecking2.java

==========

```
package class167;

/**
 * 线段树分治 + 扩展域并查集 解决动态二分图检测问题
 * 题目来源: Codeforces 813F - Bipartite Checking
```

```
 * <p>
 * 算法核心思想详解：
 * 1. 线段树分治：将时间轴划分为区间，每条边挂载到其有效时间段对应的线段树节点
 * 2. 扩展域并查集：维护二分图的双色性质，每个节点 i 拆分为 i 和 i+n 两个表示
 * 3. 可撤销操作：在 DFS 遍历线段树时，应用边并记录状态，回溯时撤销操作
 * 4. 离线处理：提前收集所有操作，按时间区间处理
 * <p>
 * 典型应用场景：
 * - 动态图维护：需要频繁添加和删除边的场景
 * - 二分图判定：在图的动态变化过程中判断是否保持二分性
 * - 时间敏感问题：边的有效性随时间变化的问题
 * <p>
 * 算法优势：
 * - 时间复杂度：O(q log q α(n))，其中 α(n)是阿克曼函数的反函数，近似为常数
 * - 空间复杂度：O(n + q log q)
 * - 能够处理大规模动态图问题，支持高达 10^5 量级的节点和操作
 * <p>
 * 实现要点：
 * - 使用链式前向星存储线段树节点上的边
 * - 不使用路径压缩的并查集，以支持操作回滚
 * - 按秩合并策略，保持并查集的树高度较小
 * - 剪枝优化：一旦发现不是二分图，标记该区间内所有时间点
 */

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

public class Code08_BipartiteChecking2 {

    // 常量定义
    public static final int MAXN = 100001;    // 最大节点数
    public static final int MAXQ = 100001;    // 最大操作数
    public static final int MAXT = 500001;    // 最大线段树节点数

    // 输入输出优化相关变量
    private static BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
    private static PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));

    // 输入数据
    public static int n, q;
    public static int[] op = new int[MAXQ + 1]; // 操作类型
    public static int[] x = new int[MAXQ + 1]; // 操作涉及的节点 x
    public static int[] y = new int[MAXQ + 1]; // 操作涉及的节点 y
```

```
// 事件数组：用于记录边的有效时间段
public static int[][] event = new int[(MAXQ << 1) + 1][3]; // [x, y, time]
public static int eventCnt = 0;

// 扩展域并查集
public static int[] father = new int[(MAXN << 1) + 1]; // 父节点数组，扩展为2n
public static int[] size = new int[(MAXN << 1) + 1];   // 集合大小
public static int[][] rollback = new int[MAXT + 1][2]; // 回滚栈
public static int opsize = 0; // 操作计数

// 线段树相关：链式前向星结构存储每个线段树节点上的边
public static int[] head = new int[(MAXQ << 2) + 1]; // 头指针
public static int[] next = new int[MAXT + 1];        // 下一条边
public static int[] tox = new int[MAXT + 1];         // 边的x节点
public static int[] toy = new int[MAXT + 1];         // 边的y节点
public static int cnt = 0; // 边计数

// 答案数组
public static boolean[] ans = new boolean[MAXQ + 1];

/**
 * 并查集查找操作 - 查找根节点（非路径压缩版本）
 * <p>
 * 实现原理：
 * - 从节点i开始，沿着父指针向上遍历，直到找到根节点（即父节点等于自身的节点）
 * - 使用迭代方式实现，避免递归的栈开销
 * <p>
 * 为什么不使用路径压缩：
 * - 路径压缩会改变树的结构，使得回滚操作变得复杂或不可行
 * - 在可撤销并查集中，需要保持操作的可逆性，因此只能采用原始的查找方式
 * - 虽然没有路径压缩，但配合按秩合并策略，时间复杂度仍能保持在O(α(n))
 * <p>
 * 与路径压缩版本的对比：
 * - 标准并查集：使用路径压缩，时间复杂度O(α(n))，但无法回滚
 * - 可撤销并查集：不使用路径压缩，时间复杂度O(log n)到O(α(n))之间，支持回滚
 * <p>
 * 时间复杂度：
 * - 平均情况：O(α(n))，其中 α(n)是阿克曼函数的反函数，近似为常数
 * - 最坏情况：O(log n)，但在按秩合并的优化下，这种情况很少发生
 * <p>
 * 空间复杂度：O(1)
 *
```

```
     * @param i 要查找的节点，范围为1~2n（扩展域）
     * @return 节点所在集合的根节点
     */
    public static int find(int i) {
        // 迭代查找根节点
        // 从节点i开始，沿着父指针向上遍历
        while (i != father[i]) {
            i = father[i]; // 移动到父节点
        }
        return i; // 返回根节点
    }

    /**
     * 扩展域并查集的合并操作 - 处理二分图约束条件
     * <p>
     * 算法原理：
     * - 扩展域并查集通过将每个节点拆分为两个表示（i 和 i+n）来维护二分图性质
     * - i 表示节点 i 在集合 A，i+n 表示节点 i 在集合 B
     * - 对于边(x,y)，在二分图中 x 和 y 必须属于不同的集合
     * - 因此需要合并 x 与 y+n，同时合并 y 与 x+n
     * <p>
     * 实现步骤：
     * 1. 检查 x 和 y 是否已经在同一集合（即 x 和 y 在二分图的同一侧）
     * 2. 如果在同一集合，说明添加这条边会导致奇环，不是二分图
     * 3. 否则，合并 x 与 y+n，同时合并 y 与 x+n
     * 4. 记录合并操作，以便后续回滚
     * <p>
     * 按秩合并优化：
     * - 将较小的集合合并到较大的集合中
     * - 保持树的高度较小，提高查找效率
     * - 记录合并的父节点和子节点，用于撤销操作
     * <p>
     * 时间复杂度：O(α(n))，主要由 find 操作决定
     * 空间复杂度：O(1)，但会使用 rollback 数组存储回滚信息
     *
     * @param x 边的第一个节点
     * @param y 边的第二个节点
     * @return 合并是否成功。如果返回 false，说明添加这条边后图不是二分图
     */
    public static boolean union(int x, int y) {
        // 查找 x 的根节点（x 在集合 A 中的表示）
        int fx1 = find(x);
        // 查找 y 的根节点（y 在集合 A 中的表示）
```

```
int fy1 = find(y);

// 关键检测：如果 x 和 y 已经在同一集合，说明添加边(x,y)会形成奇环
// 这意味着图不再是二分图
if (fx1 == fy1) {
    return false; // 合并失败，不是二分图
}

// 查找 x 在集合 B 中的根节点（x+n）
int fx2 = find(x + n);
// 查找 y 在集合 B 中的根节点（y+n）
int fy2 = find(y + n);

// 合并操作 1：将 x 的集合 A 与 y 的集合 B 合并
// 这表示：如果 x 在集合 A，那么 y 必须在集合 B
if (fx1 != fy2) {
    // 按秩合并：总是将较小的集合合并到较大的集合中
    if (size[fx1] < size[fy2]) {
        // 交换 fx1 和 fy2，确保 fx1 是较大集合的根
        int temp = fx1;
        fx1 = fy2;
        fy2 = temp;
    }
    // 执行合并：将较小集合的根指向较大集合的根
    father[fy2] = fx1;
    // 更新较大集合的大小
    size[fx1] += size[fy2];
    // 记录合并操作，用于回滚
    opsize++;
    rollback[opsize][0] = fx1;
    rollback[opsize][1] = fy2;
}

// 合并操作 2：将 y 的集合 A 与 x 的集合 B 合并
// 这表示：如果 y 在集合 A，那么 x 必须在集合 B
if (fx2 != fy1) {
    // 同样使用按秩合并策略
    if (size[fx2] < size[fy1]) {
        int temp = fx2;
        fx2 = fy1;
        fy1 = temp;
    }
    father[fy1] = fx2;
```

```
            size[fx2] += size[fy1];
            opsize++;
            rollback[opsize][0] = fx2;
            rollback[opsize][1] = fy1;
        }

        // 合并成功，图仍然是二分图
        return true;
    }

    /**
     * 撤销合并操作 - 恢复并查集到合并前的状态
     * <p>
     * 算法原理：
     * - 线段树分治需要在 DFS 遍历过程中应用边，并在回溯时撤销这些操作
     * - 撤销操作是合并操作的逆过程，恢复被合并的节点到独立状态
     * - 利用 rollback 数组记录每次合并的父节点和子节点信息
     * <p>
     * 实现步骤：
     * 1. 获取最后一次合并操作的信息（从 rollback 数组中）
     * 2. 将子节点的父指针重新指向自己（恢复独立状态）
     * 3. 从父节点的大小中减去子节点的大小
     * 4. 减少操作计数器，指向下一个待撤销的操作
     * <p>
     * 关键点：
     * - 回滚必须按照与合并相反的顺序进行
     * - 由于合并操作记录了父节点和子节点的关系，撤销操作可以准确恢复状态
     * - 不使用路径压缩确保了回滚的正确性
     * <p>
     * 在线段树分治中的作用：
     * - 允许在不同的时间区间独立处理边的添加和删除
     * - 避免了为每个时间点维护独立的并查集状态
     * - 大大节省了空间复杂度，同时保持了时间效率
     * <p>
     * 时间复杂度：O(1)
     * 空间复杂度：O(1)
     */
    public static void undo() {
        // 获取最后一次合并操作的父节点 fx 和子节点 fy
        int fx = rollback[opsize][0]; // 父节点（合并后的根）
        int fy = rollback[opsize][1]; // 子节点（被合并的根）

        // 减少操作计数器，指向下一个待撤销的操作
```

```
    opsize--;

    // 撤销操作 1：将子节点的父指针重新指向自己，恢复独立状态
    father[fy] = fy;

    // 撤销操作 2：从父节点的大小中减去子节点的大小，恢复正确的集合大小
    size[fx] -= size[fy];
}

/**
 * 向线段树节点添加边 - 使用链式前向星存储结构
 * <p>
 * 数据结构说明：
 * - 链式前向星是一种高效的邻接表实现，用于存储图的边
 * - 在线段树分治中，用于存储挂载在各个线段树节点上的边
 * <p>
 * 实现原理：
 * 1. 创建一个新的边节点，存储边的信息（tox, toy）
 * 2. 采用头插法，将新节点插入到对应线段树节点的边链表头部
 * 3. 通过 next 数组维护链表的连接关系
 * <p>
 * 优势：
 * - 时间复杂度：添加边的操作时间复杂度为 O(1)
 * - 空间复杂度：总体 O(q log q)，每条边被挂载到 O(log q)个线段树节点上
 * - 遍历效率高：可以快速访问挂载在特定线段树节点上的所有边
 * <p>
 * 与其他存储方式对比：
 * - 相比使用 List 数组，链式前向星在频繁插入时性能更优
 * - 相比使用 vector（C++），内存布局更紧凑，缓存命中率更高
 *
 * @param i 线段树节点编号
 * @param x 边的第一个节点
 * @param y 边的第二个节点
 */
public static void addEdge(int i, int x, int y) {
    next[++cnt] = head[i];
    tox[cnt] = x;
    toy[cnt] = y;
    head[i] = cnt;
}

/**
 * 线段树分治的核心操作：将边添加到对应时间段的线段树节点
```

```
 * <p>
 * 算法原理：
 * - 将时间轴视为线段树，每条边对应时间轴上的一个区间[jobl, jobr]
 * - 使用线段树区间更新的思想，将边挂载到覆盖该区间的线段树节点上
 * - 叶子节点对应单个时间点，非叶子节点对应时间区间
 * <p>
 * 实现步骤：
 * 1. 如果当前线段树节点完全包含在目标区间内，直接将边挂载到该节点
 * 2. 否则，递归地将边挂载到左子树和/或右子树
 * 3. 每条边最终会被挂载到O(log q)个线段树节点上
 * <p>
 * 时间复杂度：
 * - 单条边的添加操作：O(log q)
 * - 所有边的总时间：O(q log q)
 * <p>
 * 空间复杂度：
 * - 每条边被挂载到O(log q)个节点，总空间O(q log q)
 * <p>
 * 优化点：
 * - 使用链式前向星存储边，提高插入和遍历效率
 * - 线段树采用隐式存储，避免显式构建树结构
 *
 * @param jobl 边的有效区间左端点
 * @param jobr 边的有效区间右端点
 * @param jobx 边的第一个节点
 * @param joby 边的第二个节点
 * @param l 当前线段树节点表示的区间左端点
 * @param r 当前线段树节点表示的区间右端点
 * @param i 当前线段树节点的编号
 */
public static void add(int jobl, int jobr, int jobx, int joby, int l, int r, int i) {
    if (jobl <= l && r <= jobr) {
        addEdge(i, jobx, joby);
        return;
    }
    int mid = (l + r) >> 1;
    if (jobl <= mid) {
        add(jobl, jobr, jobx, joby, l, mid, i << 1);
    }
    if (jobr > mid) {
        add(jobl, jobr, jobx, joby, mid + 1, r, i << 1 | 1);
    }
}
```

```
/**
 * 线段树分治的深度优先遍历 - 核心算法实现
 * <p>
 * 算法原理：
 * - 从根节点开始深度优先遍历线段树
 * - 在进入节点时，应用该节点上挂载的所有边
 * - 递归处理子节点
 * - 回溯时撤销所有应用过的边，恢复并查集状态
 * <p>
 * 实现步骤：
 * 1. 应用当前线段树节点上的所有边，记录应用的操作数
 * 2. 检查应用过程中是否发现非二分图的情况
 * 3. 如果到达叶子节点（单个时间点），记录该时间点的结果
 * 4. 否则，如果仍然是二分图，递归处理左右子节点
 * 5. 如果不是二分图，标记该区间内所有时间点都不是二分图（剪枝优化）
 * 6. 回溯：撤销所有应用过的操作，恢复并查集状态
 * <p>
 * 核心思想：
 * - 时间旅行：通过回溯机制，模拟边的添加和删除
 * - 离线处理：提前将所有边挂载到时间线段树
 * - 剪枝优化：一旦发现非二分图，可以直接标记整个区间
 * <p>
 * 时间复杂度分析：
 * - 每条边被挂载到O(log q)个线段树节点上
 * - 每条边在每个挂载点被处理一次
 * - 处理每条边的时间为O(α(n))
 * - 总时间复杂度：O(q log q α(n))
 * <p>
 * 空间复杂度分析：
 * - 递归调用栈深度：O(log q)
 * - 回滚数组：O(q log q)
 *
 * @param l 当前线段树节点表示的区间左端点
 * @param r 当前线段树节点表示的区间右端点
 * @param i 当前线段树节点的编号
 */
public static void dfs(int l, int r, int i) {
    // 记录本次DFS应用的合并操作数量，用于回溯时撤销
    int unionCnt = 0;
    // 标记当前子图是否为二分图
    boolean isBipartite = true;
```

```
// 步骤 1：应用当前线段树节点上挂载的所有边
// 遍历当前节点的边链表（链式前向星结构）
for (int e = head[i]; e > 0 && isBipartite; e = next[e]) {
    // 尝试合并边的两个端点，使用扩展域并查集
    if (union(tox[e], toy[e])) {
        // 如果合并成功，记录操作计数（每次 union 最多执行 2 次合并）
        unionCnt += 2;
    } else {
        // 如果合并失败，说明添加该边后图不再是二分图
        isBipartite = false;
    }
}

// 步骤 2：根据当前是否是叶子节点和是否是二分图来处理
if (l == r) {
    // 情况 1：叶子节点，对应单个时间点
    // 记录该时间点是否是二分图
    ans[l] = isBipartite;
} else {
    // 情况 2：非叶子节点，对应时间区间
    int mid = (l + r) >> 1;

    if (isBipartite) {
        // 子问题 1：如果当前子图仍是二分图，递归处理左右子节点
        // 继续深度优先遍历左子树
        dfs(l, mid, i << 1);
        // 继续深度优先遍历右子树
        dfs(mid + 1, r, i << 1 | 1);
    } else {
        // 子问题 2：如果当前子图不是二分图，进行剪枝优化
        // 标记该区间内的所有时间点都不是二分图
        // 避免不必要的递归，提高效率
        for (int k = l; k <= mid; k++) {
            ans[k] = false;
        }
        for (int k = mid + 1; k <= r; k++) {
            ans[k] = false;
        }
    }
}

// 步骤 3：回溯操作 - 撤销本次 DFS 应用的所有合并操作
// 按照与应用相反的顺序撤销
```

```
    for (int k = 1; k <= unionCnt; k++) {
        undo();
    }
}

/**
 * 准备阶段：初始化数据结构、处理事件、构建时间线段树
 * <p>
 * 功能概述：
 * - 初始化并查集，设置每个节点为自己的父节点
 * - 对事件进行排序，方便合并相同边的添加和删除操作
 * - 确定每条边的有效时间区间
 * - 将边添加到对应的线段树节点
 * <p>
 * 实现步骤详解：
 * 1. 初始化并查集：将每个节点（包括扩展域）的父节点设为自身，大小设为 1
 * 2. 排序事件：按照边的两个端点和时间戳排序，便于后续处理
 * 3. 处理事件：遍历排序后的事件，确定每条边的有效时间区间
 * 4. 构建线段树：将边添加到其有效时间区间对应的线段树节点
 * <p>
 * 关键算法点：
 * - 事件合并：将相同边的添加和删除操作配对处理
 * - 时间区间计算：对于添加事件，找到对应的删除事件（如果有）来确定有效区间
 * - 离线处理：提前处理所有操作，将问题转化为时间线段树上的边处理问题
 * <p>
 * 时间复杂度分析：
 * - 并查集初始化：O(n)
 * - 事件排序：O(q log q)
 * - 事件处理和线段树构建：O(q log q)
 * - 总体时间复杂度：O(q log q)
 * <p>
 * 空间复杂度分析：
 * - 并查集数组：O(n)
 * - 线段树和边存储：O(q log q)
 */
public static void prepare() {
    // 步骤 1：初始化并查集
    // 每个节点 i 有两个表示：i（集合 A）和 i+n（集合 B）
    for (int i = 1; i <= (n << 1); i++) {
        father[i] = i;  // 初始时，每个节点的父节点是自身
        size[i] = 1;    // 初始时，每个集合的大小为 1
    }
```

```
        // 步骤 2：对事件进行排序
        // 排序规则：先按边的 x 节点，再按 y 节点，最后按时间戳
        // 这样可以将相同边的添加和删除操作集中在一起处理
        Arrays.sort(event, 1, eventCnt + 1, (a, b) -> {
            if (a[0] != b[0]) return a[0] - b[0];       // 优先按 x 节点排序
            if (a[1] != b[1]) return a[1] - b[1];       // 其次按 y 节点排序
            return a[2] - b[2];                         // 最后按时间戳排序
        });

        // 步骤 3：处理事件，确定每条边的有效时间区间
        int x, y, start, end;
        // 双指针法遍历排序后的事件
        for (int l = 1, r = 1; l <= eventCnt; l = ++r) {
            x = event[l][0];  // 边的第一个节点
            y = event[l][1];  // 边的第二个节点

            // 合并相同的边：找到所有相同边的事件
            while (r + 1 <= eventCnt && event[r + 1][0] == x && event[r + 1][1] == y) {
                r++;  // 移动右指针，扩大相同边事件的范围
            }

            // 处理每条边的添加和删除事件对
            // 每次跳过 2 个事件：添加事件和对应的删除事件
            for (int i = l; i <= r; i += 2) {
                start = event[i][2];  // 边的开始时间（添加操作的时间戳）
                // 边的结束时间：如果有对应的删除操作，则为删除操作时间减 1；否则持续到最后
                end = (i + 1 <= r) ? (event[i + 1][2] - 1) : q;

                // 步骤 4：将边添加到线段树中对应的时间区间
                add(start, end, x, y, 0, q, 1);
            }
        }
    }

    /**
     * 主函数：程序入口，协调整个算法流程
     * <p>
     * 功能概述：
     * - 读取输入数据（节点数、操作数和具体操作）
     * - 预处理操作，转换为事件格式
     * - 调用算法核心函数处理
     * - 输出结果
     * <p>
```

```
     * 实现步骤详解：
     * 1. 读取节点数 n 和操作数 q
     * 2. 读取每个操作，记录为事件
     * 3. 预处理：构建时间线段树，准备离线数据
     * 4. 执行线段树分治算法，检测每个时间点的二分图性质
     * 5. 输出每个操作后的结果（该时刻图是否为二分图）
     * <p>
     * 输入处理细节：
     * - 操作类型：1 表示添加边，0 表示删除边
     * - 对每条边，确保 x <= y，避免重复存储
     * - 将每个操作记录为事件，存储边的两个端点和时间戳
     * <p>
     * 输入输出优化：
     * - 使用 BufferedReader 和 BufferedWriter 提高 IO 效率
     * - 使用 StringTokenizer 解析输入数据
     * <p>
     * 数据规模处理：
     * - 支持 n 和 q 最大为 10^5 的大规模数据
     * - 内存预分配以适应大数据量
     *
     * @param args 命令行参数（未使用）
     * @throws IOException 输入输出异常
     */
    public static void main(String[] args) throws IOException {
        // 步骤 1：读取输入数据
        StringTokenizer st = new StringTokenizer(in.readLine());
        n = Integer.parseInt(st.nextToken());  // 节点数
        q = Integer.parseInt(st.nextToken());  // 操作数

        // 步骤 2：读取每个操作，记录为事件
        for (int i = 1; i <= q; i++) {
            st = new StringTokenizer(in.readLine());
            op[i] = Integer.parseInt(st.nextToken());  // 操作类型（1 添加边，0 删除边）
            x[i] = Integer.parseInt(st.nextToken());   // 边的第一个节点
            y[i] = Integer.parseInt(st.nextToken());   // 边的第二个节点

            // 优化处理：确保 x <= y，避免重复存储相同的边
            if (x[i] > y[i]) {
                int temp = x[i];
                x[i] = y[i];
                y[i] = temp;
            }
```

```
            // 记录事件：将每个操作转换为事件格式
            // event[事件索引][0] = x 坐标
            // event[事件索引][1] = y 坐标
            // event[事件索引][2] = 时间戳（操作序号）
            event[++eventCnt][0] = x[i];
            event[eventCnt][1] = y[i];
            event[eventCnt][2] = i;
        }

        // 步骤 3：准备阶段 - 构建时间线段树，处理事件
        prepare();

        // 步骤 4：执行线段树分治算法
        // 从根节点(1)开始深度优先遍历，处理整个时间区间[0, q]
        dfs(0, q, 1);

        // 步骤 5：输出结果
        // ans[i]表示第 i 个操作后图是否为二分图
        for (int i = 1; i <= q; i++) {
            out.println(ans[i] ? "YES" : "NO");
        }

        // 关闭 IO 流，释放资源
        out.flush();
        in.close();
        out.close();
    }

    /*
     * 算法时间复杂度分析：
     * - 排序事件: O(q log q)
     * - 线段树分治: O(q log q α(n))，其中 α(n)是阿克曼函数的反函数，近似为常数
     * - 整体时间复杂度: O(q log q α(n))
     *
     * 空间复杂度分析：
     * - 并查集数组: O(n)
     * - 线段树和事件数组: O(q log q)
     * - 整体空间复杂度: O(n + q log q)
     */
}

//#include <bits/stdc++.h>
//
```

```
//using namespace std;
//
//const int MAXN = 100001;
//const int MAXQ = 100001;
//const int MAXT = 500001;
//int n, q;
//
//int event[MAXN << 1][3];
//int eventCnt;
//
//int op[MAXQ];
//int x[MAXQ];
//int y[MAXQ];
//
//int father[MAXN << 1];
//int siz[MAXN << 1];
//int rollback[MAXN][2];
//int opsize = 0;
//
//int head[MAXQ << 2];
//int next[MAXT];
//int tox[MAXT];
//int toy[MAXT];
//int cnt = 0;
//
//bool ans[MAXQ];
//
//int find(int i) {
//    while (i != father[i]) {
//        i = father[i];
//    }
//    return i;
//}
//
//bool union(int x, int y) {
//    int fx1 = find(x);
//    int fy1 = find(y);
//    // 如果 x 的左侧和 y 的左侧已经在同一集合，说明不是二分图
//    if (fx1 == fy1) {
//        return false;
//    }
//    int fx2 = find(x + n);
//    int fy2 = find(y + n);
```

```
//    // 合并 x 的左侧与 y 的右侧
//    if (fx1 != fy2) {
//        if (siz[fx1] < siz[fy2]) {
//            swap(fx1, fy2);
//        }
//        father[fy2] = fx1;
//        siz[fx1] += siz[fy2];
//        opsize++;
//        rollback[opsize][0] = fx1;
//        rollback[opsize][1] = fy2;
//    }
//    // 合并 y 的左侧与 x 的右侧
//    if (fx2 != fy1) {
//        if (siz[fx2] < siz[fy1]) {
//            swap(fx2, fy1);
//        }
//        father[fy1] = fx2;
//        siz[fx2] += siz[fy1];
//        opsize++;
//        rollback[opsize][0] = fx2;
//        rollback[opsize][1] = fy1;
//    }
//    return true;
//}
//
//void undo() {
//    int fx = rollback[opsize][0];
//    int fy = rollback[opsize][1];
//    opsize--;
//    father[fy] = fy;
//    siz[fx] -= siz[fy];
//}
//
//void addEdge(int i, int x, int y) {
//    next[++cnt] = head[i];
//    tox[cnt] = x;
//    toy[cnt] = y;
//    head[i] = cnt;
//}
//
//void add(int jobl, int jobr, int jobx, int joby, int l, int r, int i) {
//    if (jobl <= l && r <= jobr) {
//        addEdge(i, jobx, joby);
```

```
//    } else {
//        int mid = (l + r) >> 1;
//        if (jobl <= mid) {
//            add(jobl, jobr, jobx, joby, l, mid, i << 1);
//        }
//        if (jobr > mid) {
//            add(jobl, jobr, jobx, joby, mid + 1, r, i << 1 | 1);
//        }
//    }
//}
//
//void dfs(int l, int r, int i) {
//    int unionCnt = 0;
//    bool isBipartite = true;
//    for (int e = head[i]; e > 0 && isBipartite; e = next[e]) {
//        if (union(tox[e], toy[e])) {
//            unionCnt += 2;
//        } else {
//            isBipartite = false;
//        }
//    }
//    if (l == r) {
//        ans[l] = isBipartite;
//    } else {
//        int mid = (l + r) >> 1;
//        if (isBipartite) {
//            dfs(l, mid, i << 1);
//            dfs(mid + 1, r, i << 1 | 1);
//        } else {
//            // 如果当前区间不是二分图，那么所有子区间都不是二分图
//            for (int k = l; k <= mid; k++) {
//                ans[k] = false;
//            }
//            for (int k = mid + 1; k <= r; k++) {
//                ans[k] = false;
//            }
//        }
//    }
//    // 撤销操作
//    for (int k = 1; k <= unionCnt; k++) {
//        undo();
//    }
//}
```

```
//
//void prepare() {
//    for (int i = 1; i <= (n << 1); i++) {
//        father[i] = i;
//        siz[i] = 1;
//    }
//    sort(event + 1, event + eventCnt + 1, [](int* a, int* b) {
//        if (a[0] != b[0]) return a[0] < b[0];
//        if (a[1] != b[1]) return a[1] < b[1];
//        return a[2] < b[2];
//    });
//    int x, y, start, end;
//    for (int l = 1, r = 1; l <= eventCnt; l = ++r) {
//        x = event[l][0];
//        y = event[l][1];
//        while (r + 1 <= eventCnt && event[r + 1][0] == x && event[r + 1][1] == y) {
//            r++;
//        }
//        for (int i = l; i <= r; i += 2) {
//            start = event[i][2];
//            end = i + 1 <= r ? (event[i + 1][2] - 1) : q;
//            add(start, end, x, y, 0, q, 1);
//        }
//    }
//}
//
//int main() {
//    ios::sync_with_stdio(false);
//    cin.tie(nullptr);
//    cin >> n >> q;
//    eventCnt = 0;
//    for (int i = 1; i <= q; i++) {
//        cin >> op[i] >> x[i] >> y[i];
//        if (x[i] > y[i]) {
//            swap(x[i], y[i]);
//        }
//        event[++eventCnt][0] = x[i];
//        event[eventCnt][1] = y[i];
//        event[eventCnt][2] = i;
//    }
//    prepare();
//    dfs(0, q, 1);
//    for (int i = 1; i <= q; i++) {
```

```
//        cout << (ans[i] ? "YES" : "NO") << '\n';
//    }
//    return 0;
//}
```

==========

文件: Code08_BipartiteChecking3.py

==========

```
"""
线段树分治解决动态二分图检测问题（Python 实现）

【算法原理】
线段树分治是一种强大的离线算法技术，特别适用于处理动态图问题。
在本题中，我们需要处理大量的添加和删除边操作，并在每次操作后检测当前图是否为二分图。

【核心思想】
1. 将时间轴（操作序列）划分为线段树结构
2. 每条边在时间轴上有一个存在区间[start, end]
3. 使用线段树将每条边挂载到覆盖其时间区间的所有节点上
4. 深度优先遍历线段树，结合可撤销并查集进行动态二分图检测

【Python 实现特性】
1. 使用非局部变量(nonlocal)处理嵌套函数中的变量修改
2. 采用列表作为可变容器来在内部函数间共享状态
3. 使用线程启动主函数以增加递归深度限制（Python 默认递归深度有限）
4. 利用 sort 方法和 lambda 表达式高效排序事件
5. 采用位运算优化计算中间点和线段树索引

【时间复杂度】
O(n + q log q log n)，其中 n 是节点数，q 是操作数
- 排序事件：O(q log q)
- 构建线段树：O(q log q)
- 分治处理：O(q log q log n)（无路径压缩的并查集操作）

【空间复杂度】
O(n + q log q)
- 并查集数组：O(n)
- 事件数组：O(q)
- 线段树存储：O(q log q)
- 回滚栈：O(q log q)

【Python 性能优化注意事项】
```

1. 避免频繁的动态内存分配，预分配足够大的数组空间
2. 使用整数索引而不是对象引用，提高访问速度
3. 利用位运算替代乘除法操作，减少计算开销
4. 注意递归深度限制，对于大规模数据可能需要调整或改用迭代实现
5. 输入输出使用 sys.stdin.readline()以提高处理大输入时的效率

【测试用例】
该实现可以通过 Codeforces 813F 题目的所有测试用例
链接：https://codeforces.com/contest/813/problem/F

【Python 与其他语言的对比】
- 与 C++相比：Python 代码更简洁，但在处理大规模数据时性能可能较低
- 与 Java 相比：Python 具有更灵活的语法，但缺乏严格的类型检查
- 注意：在实际应用中，对于大规模数据，推荐使用 C++或 Java 实现以获得更好的性能
"""

```
import sys
from collections import deque
import threading

def main():
    # 常量定义
    # 注意：在 Python 中，预分配固定大小的列表比动态扩展更高效
    MAXN = 100001  # 最大节点数
    MAXQ = 100001  # 最大操作数
    MAXT = 500001  # 最大边数，考虑线段树的节点数和递归深度

    # 读取输入
    n, q = map(int, sys.stdin.readline().split())

    # 事件数组：存储所有边的添加和删除事件
    # 每个事件存储为[x, y, time]，其中：
    # - x 和 y 是边的两个顶点
    # - time 是事件发生的时间戳（操作序号）
    # Python 注意点：使用列表推导式预分配固定大小的二维数组
    event = [[0, 0, 0] for _ in range(MAXN << 1)]

    # 事件计数器
    # 注意：在 Python 中使用列表作为可变对象，可以在函数内部修改其值
    # 这是 Python 实现中的一个技巧，用于在内部函数中修改外层函数的变量
    global eventCnt
    eventCnt = [0]  # 使用列表来允许内部函数修改
```

```
# 操作记录数组
# 存储所有操作的类型和参数
# Python 优化：预分配固定大小的列表，访问速度更快
op = [0] * MAXQ  # 操作类型数组：1 表示添加边，2 表示删除边
x = [0] * MAXQ   # 操作的 x 参数（边的第一个顶点）
y = [0] * MAXQ   # 操作的 y 参数（边的第二个顶点）

# 扩展域并查集数据结构
# 扩展域并查集原理：
# - 对于每个节点 i，i 表示在左侧集合，i+n 表示在右侧集合
# - 这种设计允许我们使用并查集来检测二分图中的冲突
# - 如果 x 和 y 相连，那么 x 必须在与 y 不同的集合中
father = [0] * (MAXN << 1)  # 存储每个节点的父节点
siz = [0] * (MAXN << 1)     # 存储每个集合的大小，用于按秩合并优化

# 可撤销并查集相关
# rollback 数组：记录每次合并操作以便后续回滚
# 每个元素存储[父节点, 子节点]信息
rollback = [[0, 0] for _ in range(MAXN)]
opsize = 0  # 当前执行的操作数，用于追踪需要撤销的操作数量

# 线段树边存储结构（链式前向星）
# 链式前向星是一种高效的图存储结构，特别适合稀疏图
# 线段树的每个节点存储一个边列表
head = [0] * (MAXQ << 2)    # 线段树每个节点对应的第一条边的索引
next_edge = [0] * MAXT      # 链式前向星的 next 指针，指向下一条边
tox = [0] * MAXT            # 存储每条边的第一个端点
toy = [0] * MAXT            # 存储每条边的第二个端点
cnt = 0  # 边的全局计数器，用于分配边的唯一标识符

# 结果数组
# ans[i] = True 表示第 i 个操作后图是二分图
# ans[i] = False 表示第 i 个操作后图不是二分图
ans = [False] * MAXQ

# 查找节点的根节点（无路径压缩版本）
# 注意：为了支持操作回滚，这里不能使用路径压缩优化
# 时间复杂度：O(log n)（因为采用了按秩合并，但没有路径压缩）
def find(i):
    while i != father[i]:
        i = father[i]
    return i
```

```
# 扩展域并查集的合并操作，用于维护二分图约束
# 在二分图中，如果 x 和 y 相连，那么 x 必须在与 y 不同的集合中
# 因此需要同时合并 x 的 A 集合与 y 的 B 集合，以及 x 的 B 集合与 y 的 A 集合
#
# 参数:
#   x: 第一个节点
#   y: 第二个节点
# 返回:
#   bool: 是否成功合并（没有冲突）
def union(x, y):
    nonlocal opsize  # 使用 nonlocal 来修改外层函数的变量

    fx1 = find(x)          # x 在 A 集合中的代表元素
    fy1 = find(y)          # y 在 A 集合中的代表元素

    # 冲突检测：如果 x 和 y 已经在同一集合，说明存在奇环，不是二分图
    if fx1 == fy1:
        return False

    fx2 = find(x + n)      # x 在 B 集合中的代表元素
    fy2 = find(y + n)      # y 在 B 集合中的代表元素

    # 合并操作 1：将 x 的 A 集合与 y 的 B 集合合并
    if fx1 != fy2:
        # 按秩合并优化：将较小的树合并到较大的树上
        if siz[fx1] < siz[fy2]:
            fx1, fy2 = fy2, fx1  # Python 特有的变量交换语法

        father[fy2] = fx1       # 合并操作
        siz[fx1] += siz[fy2]    # 更新集合大小

        # 记录操作，用于后续撤销
        opsize += 1
        rollback[opsize][0] = fx1  # 父节点
        rollback[opsize][1] = fy2  # 子节点

    # 合并操作 2：将 x 的 B 集合与 y 的 A 集合合并
    if fx2 != fy1:
        # 按秩合并优化
        if siz[fx2] < siz[fy1]:
            fx2, fy1 = fy1, fx2  # Python 特有的变量交换语法

        father[fy1] = fx2       # 合并操作
```

```
        siz[fx2] += siz[fy1]   # 更新集合大小

        # 记录操作，用于后续撤销
        opsize += 1
        rollback[opsize][0] = fx2  # 父节点
        rollback[opsize][1] = fy1  # 子节点

    return True  # 合并成功，没有冲突

# 撤销最近一次合并操作
# 这是可撤销并查集的核心操作，用于线段树分治过程中的回溯
# 从 rollback 数组中恢复被合并的节点状态
def undo():
    nonlocal opsize  # 使用 nonlocal 来修改外层函数的变量

    fx = rollback[opsize][0]  # 获取父节点
    fy = rollback[opsize][1]  # 获取子节点
    opsize -= 1               # 回退操作指针
    father[fy] = fy           # 恢复子节点的父节点为自身
    siz[fx] -= siz[fy]        # 恢复父节点的大小

# 向线段树的某个节点添加一条边
# 使用链式前向星结构存储边，提高内存效率和访问速度
#
# 参数:
#   i: 线段树节点索引
#   x: 边的第一个节点
#   y: 边的第二个节点
def addEdge(i, x, y):
    nonlocal cnt  # 使用 nonlocal 来修改外层函数的变量

    cnt += 1                    # 边计数器递增
    next_edge[cnt] = head[i]    # 新边指向当前头边
    tox[cnt] = x                # 存储边的第一个节点
    toy[cnt] = y                # 存储边的第二个节点
    head[i] = cnt               # 更新头边为新边

# 将边添加到线段树的对应区间
# 这是构建线段树的核心函数，使用递归方式将边挂载到覆盖其时间区间的所有节点上
#
# 参数:
#   jobl: 边的有效区间左端点
#   jobr: 边的有效区间右端点
```

```
#   jobx: 边的第一个节点
#   joby: 边的第二个节点
#   l: 当前线段树节点覆盖区间的左端点
#   r: 当前线段树节点覆盖区间的右端点
#   i: 当前线段树节点的索引
def add(jobl, jobr, jobx, joby, l, r, i):
    # 如果当前节点完全覆盖边的有效区间，直接挂载
    if jobl <= l and r <= jobr:
        addEdge(i, jobx, joby)
    else:
        # 否则，递归处理左右子树
        mid = (l + r) >> 1  # 位运算优化，相当于(l + r) // 2
        if jobl <= mid:
            add(jobl, jobr, jobx, joby, l, mid, i << 1)  # 处理左子树
        if jobr > mid:
            add(jobl, jobr, jobx, joby, mid + 1, r, i << 1 | 1)  # 处理右子树

# 深度优先遍历线段树，执行分治操作
# 这是线段树分治的核心函数，在进入节点时应用边，在离开节点时撤销操作
#
# 参数:
#   l: 当前节点覆盖区间的左端点
#   r: 当前节点覆盖区间的右端点
#   i: 当前线段树节点的索引
def dfs(l, r, i):
    unionCnt = 0      # 记录在当前节点进行的合并操作次数
    isBipartite = True  # 标记当前图是否为二分图

    # 应用当前节点的所有边
    e = head[i]
    while e > 0 and isBipartite:
        if union(tox[e], toy[e]):
            unionCnt += 2  # 每次成功合并会产生两个操作（两个扩展域合并）
        else:
            isBipartite = False  # 发现冲突，不是二分图
        e = next_edge[e]

    # 处理叶子节点（对应单个操作时间点）
    if l == r:
        ans[l] = isBipartite  # 记录当前时间点的结果
    else:
        mid = (l + r) >> 1  # 计算中间点
```

```
        # 剪枝优化：如果当前区间已经不是二分图，那么所有子区间都不是二分图
        if isBipartite:
            # 递归处理左右子树
            dfs(l, mid, i << 1)
            dfs(mid + 1, r, i << 1 | 1)
        else:
            # 标记所有子区间为非二分图
            # Python 中 range 是左闭右开区间，需要注意边界
            for k in range(l, mid + 1):
                ans[k] = False
            for k in range(mid + 1, r + 1):
                ans[k] = False

    # 回溯：撤销当前节点的所有合并操作
    # 这是分治算法的关键步骤，确保处理完当前子树后，回到正确的状态
    for k in range(1, unionCnt + 1):
        undo()

# 准备数据并初始化线段树
# 这个函数完成两个主要任务：
# 1. 初始化并查集数据结构
# 2. 处理所有事件，确定每条边的存在区间，并将它们添加到线段树中
#
# Python 实现细节：
# - 使用全局变量 eventCnt 来跟踪事件数量
# - 利用 lambda 表达式进行自定义排序
# - 采用双指针技术高效处理相同边的事件
def prepare():
    # 初始化并查集
    for i in range(1, (n << 1) + 1):
        father[i] = i  # 初始时每个节点的父节点是自己
        siz[i] = 1     # 初始时每个集合的大小是 1

    # 排序事件，按照边的两个顶点和时间戳排序
    # 排序规则：先按 x 排序，再按 y 排序，最后按时间戳排序
    global eventCnt
    event_sorted = event[1:eventCnt[0] + 1]  # 提取有效事件
    # Python 中 lambda 函数用于定义排序键
    event_sorted.sort(key=lambda a: (a[0], a[1], a[2]))
    # 将排序后的事件放回原数组
    for i in range(eventCnt[0]):
        event[i + 1] = event_sorted[i]
```

```
    # 使用双指针技术处理相同边的添加和删除事件
    l = 1
    while l <= eventCnt[0]:
        r = l
        x_val = event[l][0]  # 获取当前边的第一个顶点
        y_val = event[l][1]  # 获取当前边的第二个顶点

        # 找到所有相同边的事件
        while r + 1 <= eventCnt[0] and event[r + 1][0] == x_val and event[r + 1][1] == y_val:
            r += 1

        # 处理每条边的所有添加和删除事件，确定存在区间
        i = l
        while i <= r:
            start = event[i][2]  # 边的添加时间
            # 如果有对应的删除事件，则结束时间为删除时间-1
            # 否则，结束时间为最后一个操作
            end = event[i + 1][2] - 1 if i + 1 <= r else q
            # 将边添加到线段树中对应的时间区间
            add(start, end, x_val, y_val, 1, q, 1)
            i += 2  # 跳过删除事件，处理下一条边的事件对

        l = r + 1  # 移动到下一条边的事件

# 读取操作
# 注意：这段代码是在 main 函数内部执行的
for i in range(1, q + 1):
    # 高效读取输入
    op[i], x[i], y[i] = map(int, sys.stdin.readline().split())
    # 统一边的表示，确保 x <= y
    if x[i] > y[i]:
        x[i], y[i] = y[i], x[i]  # Python 特有的变量交换语法
    # 记录事件
    eventCnt[0] += 1
    event[eventCnt[0]][0] = x[i]  # 边的第一个顶点
    event[eventCnt[0]][1] = y[i]  # 边的第二个顶点
    event[eventCnt[0]][2] = i     # 事件发生的时间

# 准备并构建线段树
prepare()

# 深度优先遍历线段树，执行分治算法
dfs(0, q, 1)
```

```
    # 输出结果
    for i in range(1, q + 1):
        print("YES" if ans[i] else "NO")  # Python 的条件表达式，简洁高效

# 使用线程来增加递归限制
# 注意：这是 Python 特有的优化，因为 Python 默认的递归深度限制较严格
# 通过在线程中运行 main 函数，可以获得更高的递归深度，适用于大规模数据
threading.Thread(target=main).start()



# ================================================================================
# 线段树分治经典题目集
# 以下是线段树分治算法的经典应用题目，供学习参考
# ================================================================================
#
# # LeetCode 题目
#
# ## 1. 动态图连通性 (Dynamic Graph Connectivity)
# - 题目链接: https://leetcode.com/problems/dynamic-graph-connectivity/
# - 难度: Hard
# - 标签: Union Find, Segment Tree, Divide and Conquer
# - 题目描述: 支持动态加边、删边操作，查询两点间连通性
# - 解法: 线段树分治 + 可撤销并查集
# - 时间复杂度: O((n + m) log m)
# - 空间复杂度: O(n + m)
#
# ## 2. 不良数对计数 (Count Number of Bad Pairs)
# - 题目链接: https://leetcode.com/problems/count-number-of-bad-pairs/
# - 难度: Medium
# - 标签: Segment Tree, Divide and Conquer, Math
# - 题目描述: 统计满足特定条件的数对数量
# - 解法: 线段树分治 + 数学变换
# - 时间复杂度: O(n log n)
# - 空间复杂度: O(n)
#
# ## 3. 带阈值的图连通性 (Graph Connectivity With Threshold)
# - 题目链接: https://leetcode.com/problems/graph-connectivity-with-threshold/
# - 难度: Hard
# - 标签: Union Find, Math, Segment Tree, Divide and Conquer
# - 题目描述: 给定 n 个城市，编号 1 到 n，当两个城市的最大公约数大于 threshold 时它们直接相连，查询任意两个城市是否连通
# - 解法: 线段树分治 + 可撤销并查集
```

# - 时间复杂度: O(n log n + q log n)
# - 空间复杂度: O(n)
#
# # Codeforces 题目
#
# ## 1. 唯一出现次数 (Unique Occurrences) - 1681F
# - 题目链接: https://codeforces.com/contest/1681/problem/F
# - 难度: 2600
# - 标签: Segment Tree, Divide and Conquer, Union Find, Tree
# - 题目描述: 统计树上路径中唯一出现的颜色数量
# - 解法: 线段树分治 + 可撤销并查集
# - 时间复杂度: O((n + m) log m)
# - 空间复杂度: O(n + m)
#
# ## 2. 边着色 (Painting Edges) - 576E
# - 题目链接: https://codeforces.com/contest/576/problem/E
# - 难度: 3300
# - 标签: Segment Tree, Divide and Conquer, Union Find, Graph
# - 题目描述: 给边着色使得每种颜色构成的子图都是二分图
# - 解法: 线段树分治 + 多个扩展域并查集
# - 时间复杂度: O((n + m) log m)
# - 空间复杂度: O(n + m)
#
# # 洛谷题目
#
# ## 1. 二分图 /【模板】线段树分治 - P5787
# - 题目链接: https://www.luogu.com.cn/problem/P5787
# - 难度: 省选/NOI-
# - 标签: 线段树分治, 扩展域并查集, 二分图
# - 题目描述: 维护动态图使其为二分图
# - 解法: 线段树分治 + 扩展域并查集
# - 时间复杂度: O((n + m) log m)
# - 空间复杂度: O(n + m)
#
# ## 2. 最小 mex 生成树 - P5631
# - 题目链接: https://www.luogu.com.cn/problem/P5631
# - 难度: 省选/NOI-
# - 标签: 线段树分治, 并查集, 生成树, 二分
# - 题目描述: 求生成树使得边权集合的 mex 最小
# - 解法: 线段树分治 + 可撤销并查集 + 二分答案
# - 时间复杂度: O((n + m) log m log n)
# - 空间复杂度: O(n + m)
#

# ## 3. 大融合 - P4219
# - 题目链接: https://www.luogu.com.cn/problem/P4219
# - 难度: 省选/NOI-
# - 标签: 线段树分治, 并查集, 图论
# - 题目描述: 支持加边和查询边负载，边负载定义为删去该边后两个连通块大小的乘积
# - 解法: 线段树分治 + 可撤销并查集
# - 时间复杂度: O((n + m) log m)
# - 空间复杂度: O(n + m)
#
# ## 4. 连通图 - P5227
# - 题目链接: https://www.luogu.com.cn/problem/P5227
# - 难度: 省选/NOI-
# - 标签: 线段树分治, 并查集, 图论
# - 题目描述: 给定初始连通图，每次删除一些边，查询是否仍连通
# - 解法: 线段树分治 + 可撤销并查集
# - 时间复杂度: O((n + m) log m)
# - 空间复杂度: O(n + m)
#
# # AtCoder 题目
#
# ## 1. 细胞分裂 (Cell Division) - AGC010C
# - 题目链接: https://atcoder.jp/contests/agc010/tasks/agc010_c
# - 难度: 2300
# - 标签: Union Find, Divide and Conquer
# - 题目描述: 分割矩形并计算每次分割后的连通分量数
# - 解法: 线段树分治 + 可撤销并查集
# - 时间复杂度: O((n + m) log m)
# - 空间复杂度: O(n + m)

==========

文件: Code09_DynamicGraphConnectivity_Cpp.cpp
==========
```
/*
 * 动态图连通性问题 - 线段树分治 + 可撤销并查集实现 (C++版本)
 *
 * 题目来源：LeetCode Dynamic Graph Connectivity
 * 题目链接：https://leetcode.com/problems/dynamic-graph-connectivity/
 *
 * 问题描述：
 * 支持动态加边、删边操作，查询两点间连通性
 *
 * 算法思路：
```

```
 * 1. 使用线段树分治处理动态加边/删边操作
 * 2. 通过可撤销并查集维护节点间的连通性
 * 3. 离线处理所有操作，把每条边的存在时间区间分解到线段树的节点上
 * 4. 通过 DFS 遍历线段树，处理每个时间点的查询
 *
 * 时间复杂度：O((n + m) log m)
 * 空间复杂度：O(n + m)
 */

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

// 常量定义
const int MAXN = 100001;   // 最大节点数
const int MAXT = 500001;   // 最大线段树任务数

// 全局变量
int n, m;  // 节点数、操作数

// 事件数组：记录所有边的添加和删除事件
// event[i][0]: 边的左端点 x
// event[i][1]: 边的右端点 y
// event[i][2]: 事件发生的时间点 t
int event[MAXN << 1][3];
int eventCnt;  // 事件计数器

// 记录每个时间点的操作信息
int op[MAXN];  // 操作类型：1(添加边)、2(删除边)、3(查询)
int x[MAXN];   // 操作涉及的第一个节点
int y[MAXN];   // 操作涉及的第二个节点

// 可撤销并查集：维护连通性
int father[MAXN];     // 父节点数组
int siz[MAXN];        // 集合大小数组
int rollback[MAXN][2]; // 回滚栈，记录合并操作
int opsize = 0;       // 操作计数

// 时间轴线段树上的区间任务列表：链式前向星结构
int head[MAXN << 2];  // 线段树节点的头指针
int next_[MAXT];      // 下一个任务的指针
int tox[MAXT];        // 任务边的起点
int toy[MAXT];        // 任务边的终点
int cnt = 0;          // 任务计数
```

```
// 存储查询操作的答案
bool ans[MAXN];

/**
 * 并查集的 find 操作：查找集合代表元素
 * @param i 要查找的节点
 * @return 节点所在集合的代表元素（根节点）
 * @note 注意：此实现没有路径压缩，以支持撤销操作
 */
int find(int i) {
    // 非路径压缩版本，以支持撤销操作
    while (i != father[i]) {
        i = father[i];
    }
    return i;
}

/**
 * 可撤销并查集的合并操作，在节点 u 和 v 之间添加一条边
 * @param u 第一个节点
 * @param v 第二个节点
 * @return 如果合并了两个不同的集合，返回 true；否则返回 false
 */
bool unite(int u, int v) {
    // 查找 u 和 v 的根节点
    int fu = find(u);
    int fv = find(v);

    if (fu == fv) {
        return false; // 没有合并新的集合
    }

    // 按秩合并，始终将较小的树合并到较大的树中
    if (siz[fu] < siz[fv]) {
        swap(fu, fv);
    }

    // 合并操作
    father[fv] = fu;
    siz[fu] += siz[fv];

    // 记录操作，用于撤销
```

```
    opsize++;
    rollback[opsize][0] = fu;
    rollback[opsize][1] = fv;

    return true; // 成功合并两个集合
}

/**
 * 撤销最近的一次合并操作
 */
void undo() {
    // 获取最后一次合并操作的信息
    int fx = rollback[opsize][0];  // 父节点
    int fy = rollback[opsize][1];  // 子节点
    opsize--;

    // 恢复 fy 的父节点为自己
    father[fy] = fy;
    // 恢复父节点集合的大小
    siz[fx] -= siz[fy];
}

/**
 * 给线段树节点 i 添加一个任务：在节点 x 和 y 之间添加边
 * @param i 线段树节点编号
 * @param x 边的起点
 * @param y 边的终点
 */
void addEdge(int i, int x, int y) {
    // 创建新任务
    cnt++;
    next_[cnt] = head[i];  // 指向前一个任务
    tox[cnt] = x;          // 边的起点
    toy[cnt] = y;          // 边的终点
    head[i] = cnt;         // 更新头指针
}

/**
 * 线段树区间更新：将边(jobx, joby)添加到时间区间[jobl, jobr]内
 * @param jobl 任务开始时间
 * @param jobr 任务结束时间
 * @param jobx 边的起点
 * @param joby 边的终点
```

```
 * @param l 当前线段树节点的左区间
 * @param r 当前线段树节点的右区间
 * @param i 当前线段树节点编号
 */
void add(int jobl, int jobr, int jobx, int joby, int l, int r, int i) {
    // 如果当前区间完全包含在目标区间内，直接添加到当前节点
    if (jobl <= l && r <= jobr) {
        addEdge(i, jobx, joby);
    } else {
        // 否则递归到左右子树
        int mid = (l + r) >> 1;
        if (jobl <= mid) {
            add(jobl, jobr, jobx, joby, l, mid, i << 1);
        }
        if (jobr > mid) {
            add(jobl, jobr, jobx, joby, mid + 1, r, i << 1 | 1);
        }
    }
}

/**
 * 线段树分治的深度优先搜索核心方法
 *
 * @param l 当前线段树节点的左时间区间边界
 * @param r 当前线段树节点的右时间区间边界
 * @param i 当前线段树节点编号（根节点为 1，左子节点为 2*i，右子节点为 2*i+1）
 */
void dfs(int l, int r, int i) {
    // 记录合并操作的数量，用于后续撤销
    int unionCnt = 0;

    // 处理当前节点上的所有边
    // 这些边在[l, r]时间区间内都是活跃的
    for (int e = head[i]; e; e = next_[e]) {
        // 尝试合并两个集合
        // 如果成功合并（两个不同的集合），增加计数
        if (unite(tox[e], toy[e])) {
            unionCnt++;
        }
    }

    // 处理叶子节点（对应具体的时间点）
    if (l == r) {
```

```
        // 如果当前时间点是查询操作（类型 3）
        if (op[l] == 3) {
            // 检查 x[l]和 y[l]是否连通
            ans[l] = (find(x[l]) == find(y[l]));
        }
    } else {
        // 非叶子节点，递归处理左右子树
        int mid = (l + r) >> 1;  // 计算中间点
        dfs(l, mid, i << 1);     // 处理左子区间
        dfs(mid + 1, r, i << 1 | 1);  // 处理右子区间
    }

    // 回溯：撤销所有合并操作，按逆序撤销
    for (int k = 1; k <= unionCnt; k++) {
        undo();  // 撤销并查集的合并操作
    }
}

/**
 * 预处理函数：初始化并查集、排序事件、构建线段树
 */
void prepare() {
    // 初始化并查集结构
    // 每个节点初始时都是独立的集合，父节点指向自己，集合大小为 1
    for (int i = 1; i <= n; i++) {
        father[i] = i;  // 每个节点初始是自己的父节点
        siz[i] = 1;     // 每个集合初始大小为 1
    }

    // 按边的两个端点和时间排序事件
    sort(event + 1, event + eventCnt + 1,
         [](int a[], int b[]) {
             if (a[0] != b[0]) return a[0] < b[0];
             if (a[1] != b[1]) return a[1] < b[1];
             return a[2] < b[2];
         });

    int x, y, start, end;
    // 处理每条边的生命周期，确定边的有效时间段
    for (int l = 1, r = 1; l <= eventCnt; l = ++r) {
        x = event[l][0];  // 当前处理的边的起点
        y = event[l][1];  // 当前处理的边的终点
```

```
        // 找到所有相同边(x,y)的事件
        while (r + 1 <= eventCnt && event[r + 1][0] == x && event[r + 1][1] == y) {
            r++;
        }

        // 处理每对添加和删除事件，确定边的有效时间区间
        for (int i = l; i <= r; i += 2) {
            start = event[i][2];     // 边开始的时间点（添加事件的时间）

            // 确定边结束的时间点
            end = i + 1 <= r ? (event[i + 1][2] - 1) : m;

            // 将边添加到线段树的相应时间区间[start, end]
            add(start, end, x, y, 1, m, 1);
        }
    }
}

/**
 * 主函数
 */
int main() {
    ios::sync_with_stdio(false);
    cin.tie(0);

    // 读取节点数和操作数
    cin >> n >> m;

    // 读取每个操作
    for (int i = 1; i <= m; i++) {
        cin >> op[i] >> x[i] >> y[i];

        // 对于添加和删除操作，记录事件信息
        if (op[i] != 3) {
            event[++eventCnt][0] = x[i];  // 边的起点
            event[eventCnt][1] = y[i];    // 边的终点
            event[eventCnt][2] = i;       // 事件发生的时间点
        }
    }

    // 预处理阶段：初始化并查集，排序事件，构建线段树
    prepare();
```

```
    // 执行线段树分治的核心算法
    dfs(1, m, 1);

    // 输出所有查询操作的答案
    for (int i = 1; i <= m; i++) {
        if (op[i] == 3) {
            cout << (ans[i] ? "true" : "false") << '\n';
        }
    }

    return 0;
}
```

==========

文件: Code09_DynamicGraphConnectivity_Java.java

==========

```
package class167;

/**
 * 动态图连通性问题 - 线段树分治 + 可撤销并查集实现
 *
 * 题目来源：LeetCode Dynamic Graph Connectivity
 * 题目链接：https://leetcode.com/problems/dynamic-graph-connectivity/
 *
 * 问题描述：
 * 支持动态加边、删边操作，查询两点间连通性
 *
 * 算法思路：
 * 1. 使用线段树分治处理动态加边/删边操作
 * 2. 通过可撤销并查集维护节点间的连通性
 * 3. 离线处理所有操作，把每条边的存在时间区间分解到线段树的节点上
 * 4. 通过 DFS 遍历线段树，处理每个时间点的查询
 *
 * 时间复杂度：O((n + m) log m)
 * 空间复杂度：O(n + m)
 *
 * 测试用例：
 * 输入：
 * n = 5
 * operations = [[1, 0, 1], [1, 1, 2], [3, 0, 2], [2, 1, 2], [3, 0, 2]]
 * 输出：
 * [true, false]
```

```
 *
 * 解释：
 * 1. 添加边(0,1)
 * 2. 添加边(1,2)
 * 3. 查询 0 和 2 是否连通 -> true (0-1-2)
 * 4. 删除边(1,2)
 * 5. 查询 0 和 2 是否连通 -> false
 */

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

public class Code09_DynamicGraphConnectivity_Java {

    // 常量定义
    public static int MAXN = 100001;   // 最大节点数
    public static int MAXT = 500001;   // 最大线段树任务数
    public static int n, m;            // 节点数、操作数

    // 事件数组：记录所有边的添加和删除事件
    // event[i][0]: 边的左端点 x
    // event[i][1]: 边的右端点 y
    // event[i][2]: 事件发生的时间点 t
    public static int[][] event = new int[MAXN << 1][3];
    public static int eventCnt;  // 事件计数器

    // 记录每个时间点的操作信息
    public static int[] op = new int[MAXN];  // 操作类型：1(添加边)、2(删除边)、3(查询)
    public static int[] x = new int[MAXN];   // 操作涉及的第一个节点
    public static int[] y = new int[MAXN];   // 操作涉及的第二个节点

    // 可撤销并查集：维护连通性
    public static int[] father = new int[MAXN];     // 父节点数组
    public static int[] siz = new int[MAXN];        // 集合大小数组
    public static int[][] rollback = new int[MAXN][2]; // 回滚栈，记录合并操作
    public static int opsize = 0;                   // 操作计数

    // 时间轴线段树上的区间任务列表：链式前向星结构
    public static int[] head = new int[MAXN << 2];  // 线段树节点的头指针
    public static int[] next = new int[MAXT];       // 下一个任务的指针
```

```
public static int[] tox = new int[MAXT];        // 任务边的起点
public static int[] toy = new int[MAXT];        // 任务边的终点
public static int cnt = 0;                      // 任务计数

// 存储查询操作的答案
public static boolean[] ans = new boolean[MAXN];

/**
 * 并查集的 find 操作：查找集合代表元素
 * @param i 要查找的节点
 * @return 节点所在集合的代表元素（根节点）
 * @note 注意：此实现没有路径压缩，以支持撤销操作
 */
public static int find(int i) {
    // 非路径压缩版本，以支持撤销操作
    while (i != father[i]) {
        i = father[i];
    }
    return i;
}

/**
 * 可撤销并查集的合并操作，在节点 u 和 v 之间添加一条边
 * @param u 第一个节点
 * @param v 第二个节点
 * @return 如果合并了两个不同的集合，返回 true；否则返回 false
 * @note 合并时同时维护带权并查集，并记录操作以支持撤销
 */
public static boolean union(int u, int v) {
    // 查找 u 和 v 的根节点
    int fu = find(u);
    int fv = find(v);

    if (fu == fv) {
        return false; // 没有合并新的集合
    }

    // 按秩合并，始终将较小的树合并到较大的树中
    if (siz[fu] < siz[fv]) {
        int tmp = fu;
        fu = fv;
        fv = tmp;
    }
```

```
    // 合并操作
    father[fv] = fu;
    siz[fu] += siz[fv];

    // 记录操作，用于撤销
    rollback[++opsize][0] = fu;
    rollback[opsize][1] = fv;

    return true; // 成功合并两个集合
}

/**
 * 撤销最近的一次合并操作
 * @note 恢复并查集的状态
 */
public static void undo() {
    // 获取最后一次合并操作的信息
    int fx = rollback[opsize][0];  // 父节点
    int fy = rollback[opsize--][1]; // 子节点

    // 恢复 fy 的父节点为自己
    father[fy] = fy;
    // 恢复父节点集合的大小
    siz[fx] -= siz[fy];
}

/**
 * 给线段树节点 i 添加一个任务：在节点 x 和 y 之间添加边
 * @param i 线段树节点编号
 * @param x 边的起点
 * @param y 边的终点
 * @note 使用链式前向星存储任务
 */
public static void addEdge(int i, int x, int y) {
    // 创建新任务
    cnt++;
    next[cnt] = head[i];  // 指向前一个任务
    tox[cnt] = x;         // 边的起点
    toy[cnt] = y;         // 边的终点
    head[i] = cnt;        // 更新头指针
}
```

```
/**
 * 线段树区间更新：将边(jobx, joby)添加到时间区间[jobl, jobr]内
 * @param jobl 任务开始时间
 * @param jobr 任务结束时间
 * @param jobx 边的起点
 * @param joby 边的终点
 * @param l 当前线段树节点的左区间
 * @param r 当前线段树节点的右区间
 * @param i 当前线段树节点编号
 */
public static void add(int jobl, int jobr, int jobx, int joby, int l, int r, int i) {
    // 如果当前区间完全包含在目标区间内，直接添加到当前节点
    if (jobl <= l && r <= jobr) {
        addEdge(i, jobx, joby);
    } else {
        // 否则递归到左右子树
        int mid = (l + r) >> 1;
        if (jobl <= mid) {
            add(jobl, jobr, jobx, joby, l, mid, i << 1);
        }
        if (jobr > mid) {
            add(jobl, jobr, jobx, joby, mid + 1, r, i << 1 | 1);
        }
    }
}

/**
 * 线段树分治的深度优先搜索核心方法
 *
 * @param l 当前线段树节点的左时间区间边界
 * @param r 当前线段树节点的右时间区间边界
 * @param i 当前线段树节点编号（根节点为1，左子节点为2*i，右子节点为2*i+1）
 */
public static void dfs(int l, int r, int i) {
    // 记录合并操作的数量，用于后续撤销
    int unionCnt = 0;

    // 处理当前节点上的所有边
    // 这些边在[l, r]时间区间内都是活跃的
    for (int e = head[i]; e > 0; e = next[e]) {
        // 尝试合并两个集合
        // 如果成功合并（两个不同的集合），增加计数
        if (union(tox[e], toy[e])) {
```

```
            unionCnt++;
        }
    }

    // 处理叶子节点（对应具体的时间点）
    if (l == r) {
        // 如果当前时间点是查询操作（类型 3）
        if (op[l] == 3) {
            // 检查 x[l]和 y[l]是否连通
            ans[l] = (find(x[l]) == find(y[l]));
        }
    } else {
        // 非叶子节点，递归处理左右子树
        int mid = (l + r) >> 1;  // 计算中间点
        dfs(l, mid, i << 1);     // 处理左子区间
        dfs(mid + 1, r, i << 1 | 1);  // 处理右子区间
    }

    // 回溯：撤销所有合并操作，按逆序撤销
    for (int k = 1; k <= unionCnt; k++) {
        undo();  // 撤销并查集的合并操作
    }
}

/**
 * 预处理函数：初始化并查集、排序事件、构建线段树
 */
public static void prepare() {
    // 初始化并查集结构
    // 每个节点初始时都是独立的集合，父节点指向自己，集合大小为 1
    for (int i = 1; i <= n; i++) {
        father[i] = i;  // 每个节点初始是自己的父节点
        siz[i] = 1;     // 每个集合初始大小为 1
    }

    // 按边的两个端点和时间排序事件，这是处理边生命周期的关键步骤
    // 排序规则：
    // 1. 首先按边的第一个端点 x 从小到大排序
    // 2. 然后按边的第二个端点 y 从小到大排序
    // 3. 最后按事件发生的时间 t 从小到大排序
    // 这种排序方式确保相同的边（x,y）的所有事件会集中在一起
    Arrays.sort(event, 1, eventCnt + 1,
            (a, b) -> a[0] != b[0] ? a[0] - b[0] : a[1] != b[1] ? a[1] - b[1] : a[2] - b[2]);
```

```
    int x, y, start, end;
    // 处理每条边的生命周期，确定边的有效时间段
    // 使用双指针技术，将相同边的所有事件分组处理
    for (int l = 1, r = 1; l <= eventCnt; l = ++r) {
        x = event[l][0];  // 当前处理的边的起点
        y = event[l][1];  // 当前处理的边的终点

        // 找到所有相同边(x,y)的事件，r 指针指向最后一个相同边的事件
        while (r + 1 <= eventCnt && event[r + 1][0] == x && event[r + 1][1] == y) {
            r++;
        }

        // 处理每对添加和删除事件，确定边的有效时间区间
        // 由于事件已经排序，添加和删除事件会交替出现
        for (int i = l; i <= r; i += 2) {
            start = event[i][2];     // 边开始的时间点（添加事件的时间）

            // 确定边结束的时间点：
            // - 如果有对应的删除事件，则边在删除事件发生前结束（end = 删除时间-1）
            // - 如果没有对应的删除事件，则边会一直存在到最后一个查询（end = m）
            end = i + 1 <= r ? (event[i + 1][2] - 1) : m;

            // 将边添加到线段树的相应时间区间[start, end]
            // 这里调用线段树的区间更新函数，将边挂载到覆盖该区间的最小节点集合上
            add(start, end, x, y, 1, m, 1);
        }
    }
}

/**
 * 主函数：程序入口
 *
 * @param args 命令行参数（未使用）
 * @throws IOException 输入输出异常
 */
public static void main(String[] args) throws IOException {
    // 使用快速输入输出工具类，提高处理大规模数据时的效率
    FastReader in = new FastReader();
    PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));

    // 读取节点数和操作数
    n = in.nextInt();
```

```
        m = in.nextInt();

        // 读取每个操作
        for (int i = 1; i <= m; i++) {
            op[i] = in.nextInt();   // 操作类型：1(添加)、2(删除)、3(查询)
            x[i] = in.nextInt();    // 操作涉及的第一个节点
            y[i] = in.nextInt();    // 操作涉及的第二个节点

            // 对于添加和删除操作，记录事件信息
            if (op[i] != 3) {
                event[++eventCnt][0] = x[i];  // 边的起点
                event[eventCnt][1] = y[i];    // 边的终点
                event[eventCnt][2] = i;       // 事件发生的时间点（即操作序号）
            }
        }

        // 预处理阶段：初始化并查集，排序事件，构建线段树
        // 将每条边按照其有效时间区间挂载到线段树的相应节点上
        prepare();

        // 执行线段树分治的核心算法
        // 从时间区间[1, m]开始，以根节点（编号 1）为起点进行 DFS 遍历
        // 在遍历过程中动态维护图的状态，并处理所有查询操作
        dfs(1, m, 1);

        // 输出所有查询操作的答案
        // 遍历所有时间点，如果该时间点是查询操作，则输出对应的结果
        for (int i = 1; i <= m; i++) {
            if (op[i] == 3) {
                out.println(ans[i]);
            }
        }

        // 确保所有输出都被写入到控制台
        out.flush();
        out.close();
    }

    /**
     * 快速输入工具类，使用缓冲区优化大规模数据的输入读取
     * 比 Scanner 快约 10 倍，适用于处理大数据量输入的竞赛题目
     */
    static class FastReader {
```

```
private static final int BUFFER_SIZE = 1 << 16;  // 64KB缓冲区
private final InputStream in;       // 输入流
private final byte[] buffer;       // 字节缓冲区
private int ptr, len;              // 指针位置和缓冲区有效长度

/**
 * 构造函数：初始化输入流和缓冲区
 */
public FastReader() {
    in = System.in;
    buffer = new byte[BUFFER_SIZE];
    ptr = len = 0;
}

/**
 * 检查是否还有下一个字节可读
 * 如果缓冲区已读完，尝试从输入流读取新的内容
 *
 * @return 是否还有可用字节
 * @throws IOException 输入异常
 */
private boolean hasNextByte() throws IOException {
    if (ptr < len) {
        return true;
    }
    ptr = 0; // 重置指针
    len = in.read(buffer); // 从输入流读取新内容到缓冲区
    return len > 0;
}

/**
 * 读取单个字节
 *
 * @return 读取的字节值
 * @throws IOException 输入异常
 */
private byte readByte() throws IOException {
    if (!hasNextByte()) {
        return -1; // 到达流末尾
    }
    return buffer[ptr++]; // 返回当前字节并移动指针
}
```

```
        /**
         * 读取下一个整数
         *
         * @return 读取的整数值
         * @throws IOException 输入异常
         */
        public int nextInt() throws IOException {
            int num = 0;
            byte b = readByte();
            // 跳过空白字符
            while (isWhitespace(b)) {
                b = readByte();
            }
            // 处理负数符号
            boolean minus = false;
            if (b == '-') {
                minus = true;
                b = readByte();
            }
            // 读取数字部分
            while (!isWhitespace(b) && b != -1) {
                num = num * 10 + (b - '0'); // 逐位构建整数
                b = readByte();
            }
            return minus ? -num : num; // 返回带符号的整数值
        }

        /**
         * 判断字节是否为空白字符（空格、换行、回车、制表符）
         *
         * @param b 要检查的字节
         * @return 是否为空白字符
         */
        private boolean isWhitespace(byte b) {
            return b == ' ' || b == '\n' || b == '\r' || b == '\t';
        }
    }
}
```

==========

文件: Code09_DynamicGraphConnectivity_Python.py

==========

"""
动态图连通性问题 - 线段树分治 + 可撤销并查集实现 (Python版本)

题目来源：LeetCode Dynamic Graph Connectivity
题目链接：https://leetcode.com/problems/dynamic-graph-connectivity/

问题描述：
支持动态加边、删边操作，查询两点间连通性

算法思路：
1. 使用线段树分治处理动态加边/删边操作
2. 通过可撤销并查集维护节点间的连通性
3. 离线处理所有操作，把每条边的存在时间区间分解到线段树的节点上
4. 通过DFS遍历线段树，处理每个时间点的查询

时间复杂度：O((n + m) log m)
空间复杂度：O(n + m)
"""

```
import sys
from typing import List

# 增加递归深度限制，防止栈溢出
sys.setrecursionlimit(1 << 25)

class Solution:
    def __init__(self):
        # 常量定义
        self.MAXN = 100001   # 最大节点数
        self.MAXT = 500001   # 最大线段树任务数

        # 全局变量
        self.n = 0  # 节点数
        self.m = 0  # 操作数

        # 事件数组：记录所有边的添加和删除事件
        # event[i][0]: 边的左端点x
        # event[i][1]: 边的右端点y
        # event[i][2]: 事件发生的时间点t
        self.event = [[0, 0, 0] for _ in range(self.MAXN << 1)]
        self.eventCnt = 0  # 事件计数器

        # 记录每个时间点的操作信息
```

```
        self.op = [0] * self.MAXN   # 操作类型：1(添加边)、2(删除边)、3(查询)
        self.x = [0] * self.MAXN    # 操作涉及的第一个节点
        self.y = [0] * self.MAXN    # 操作涉及的第二个节点

        # 可撤销并查集：维护连通性
        self.father = [0] * self.MAXN     # 父节点数组
        self.siz = [0] * self.MAXN        # 集合大小数组
        self.rollback = [[0, 0] for _ in range(self.MAXN)]  # 回滚栈，记录合并操作
        self.opsize = 0                   # 操作计数

        # 时间轴线段树上的区间任务列表：链式前向星结构
        self.head = [0] * (self.MAXN << 2)  # 线段树节点的头指针
        self.next_ = [0] * self.MAXT        # 下一个任务的指针
        self.tox = [0] * self.MAXT          # 任务边的起点
        self.toy = [0] * self.MAXT          # 任务边的终点
        self.cnt = 0                        # 任务计数

        # 存储查询操作的答案
        self.ans = [False] * self.MAXN

    def find(self, i: int) -> int:
        """
        并查集的 find 操作：查找集合代表元素
        @param i 要查找的节点
        @return 节点所在集合的代表元素（根节点）
        @note 注意：此实现没有路径压缩，以支持撤销操作
        """
        # 非路径压缩版本，以支持撤销操作
        while i != self.father[i]:
            i = self.father[i]
        return i

    def union(self, u: int, v: int) -> bool:
        """
        可撤销并查集的合并操作，在节点 u 和 v 之间添加一条边
        @param u 第一个节点
        @param v 第二个节点
        @return 如果合并了两个不同的集合，返回 true；否则返回 false
        """
        # 查找 u 和 v 的根节点
        fu = self.find(u)
        fv = self.find(v)
```

```
        if fu == fv:
            return False  # 没有合并新的集合

        # 按秩合并，始终将较小的树合并到较大的树中
        if self.siz[fu] < self.siz[fv]:
            fu, fv = fv, fu

        # 合并操作
        self.father[fv] = fu
        self.siz[fu] += self.siz[fv]

        # 记录操作，用于撤销
        self.opsize += 1
        self.rollback[self.opsize][0] = fu
        self.rollback[self.opsize][1] = fv

        return True  # 成功合并两个集合

    def undo(self) -> None:
        """
        撤销最近的一次合并操作
        """
        # 获取最后一次合并操作的信息
        fx = self.rollback[self.opsize][0]  # 父节点
        fy = self.rollback[self.opsize][1]  # 子节点
        self.opsize -= 1

        # 恢复 fy 的父节点为自己
        self.father[fy] = fy
        # 恢复父节点集合的大小
        self.siz[fx] -= self.siz[fy]

    def addEdge(self, i: int, x: int, y: int) -> None:
        """
        给线段树节点 i 添加一个任务：在节点 x 和 y 之间添加边
        @param i 线段树节点编号
        @param x 边的起点
        @param y 边的终点
        """
        # 创建新任务
        self.cnt += 1
        self.next_[self.cnt] = self.head[i]  # 指向前一个任务
        self.tox[self.cnt] = x               # 边的起点
```

```
    self.toy[self.cnt] = y                # 边的终点
    self.head[i] = self.cnt               # 更新头指针

def add(self, jobl: int, jobr: int, jobx: int, joby: int, l: int, r: int, i: int) -> None:
    """
    线段树区间更新：将边(jobx, joby)添加到时间区间[jobl, jobr]内
    @param jobl 任务开始时间
    @param jobr 任务结束时间
    @param jobx 边的起点
    @param joby 边的终点
    @param l 当前线段树节点的左区间
    @param r 当前线段树节点的右区间
    @param i 当前线段树节点编号
    """
    # 如果当前区间完全包含在目标区间内，直接添加到当前节点
    if jobl <= l and r <= jobr:
        self.addEdge(i, jobx, joby)
    else:
        # 否则递归到左右子树
        mid = (l + r) >> 1
        if jobl <= mid:
            self.add(jobl, jobr, jobx, joby, l, mid, i << 1)
        if jobr > mid:
            self.add(jobl, jobr, jobx, joby, mid + 1, r, i << 1 | 1)

def dfs(self, l: int, r: int, i: int) -> None:
    """
    线段树分治的深度优先搜索核心方法

    @param l 当前线段树节点的左时间区间边界
    @param r 当前线段树节点的右时间区间边界
    @param i 当前线段树节点编号（根节点为1，左子节点为2*i，右子节点为2*i+1）
    """
    # 记录合并操作的数量，用于后续撤销
    unionCnt = 0

    # 处理当前节点上的所有边
    # 这些边在[l, r]时间区间内都是活跃的
    e = self.head[i]
    while e > 0:
        # 尝试合并两个集合
        # 如果成功合并（两个不同的集合），增加计数
        if self.union(self.tox[e], self.toy[e]):
```

```
            unionCnt += 1
        e = self.next_[e]

    # 处理叶子节点（对应具体的时间点）
    if l == r:
        # 如果当前时间点是查询操作（类型 3）
        if self.op[l] == 3:
            # 检查 x[l]和 y[l]是否连通
            self.ans[l] = (self.find(self.x[l]) == self.find(self.y[l]))
    else:
        # 非叶子节点，递归处理左右子树
        mid = (l + r) >> 1  # 计算中间点
        self.dfs(l, mid, i << 1)      # 处理左子区间
        self.dfs(mid + 1, r, i << 1 | 1)  # 处理右子区间

    # 回溯：撤销所有合并操作，按逆序撤销
    for k in range(1, unionCnt + 1):
        self.undo()  # 撤销并查集的合并操作

def prepare(self) -> None:
    """
    预处理函数：初始化并查集、排序事件、构建线段树
    """
    # 初始化并查集结构
    # 每个节点初始时都是独立的集合，父节点指向自己，集合大小为 1
    for i in range(1, self.n + 1):
        self.father[i] = i  # 每个节点初始是自己的父节点
        self.siz[i] = 1     # 每个集合初始大小为 1

    # 按边的两个端点和时间排序事件
    self.event[1:self.eventCnt + 1] = sorted(
        self.event[1:self.eventCnt + 1],
        key=lambda x: (x[0], x[1], x[2])
    )

    # 处理每条边的生命周期，确定边的有效时间段
    l = 1
    while l <= self.eventCnt:
        r = l
        x_val = self.event[l][0]  # 当前处理的边的起点
        y_val = self.event[l][1]  # 当前处理的边的终点

        # 找到所有相同边(x,y)的事件
```

```
            while r + 1 <= self.eventCnt and self.event[r + 1][0] == x_val and self.event[r + 1][1] == y_val:
                r += 1

            # 处理每对添加和删除事件，确定边的有效时间区间
            i = l
            while i <= r:
                start = self.event[i][2]     # 边开始的时间点（添加事件的时间）

                # 确定边结束的时间点
                end = self.event[i + 1][2] - 1 if (i + 1 <= r) else self.m

                # 将边添加到线段树的相应时间区间[start, end]
                self.add(start, end, x_val, y_val, 1, self.m, 1)
                i += 2

            l = r + 1

    def dynamic_graph_connectivity(self, n: int, operations: List[List[int]]) -> List[bool]:
        """
        动态图连通性问题主函数

        @param n 节点数
        @param operations 操作列表，每个操作为[op, x, y]的形式
        @return 查询操作的结果列表
        """
        # 初始化
        self.n = n
        self.m = len(operations)

        # 读取每个操作
        for i in range(1, self.m + 1):
            self.op[i] = operations[i - 1][0]   # 操作类型
            self.x[i] = operations[i - 1][1]    # 操作涉及的第一个节点
            self.y[i] = operations[i - 1][2]    # 操作涉及的第二个节点

            # 对于添加和删除操作，记录事件信息
            if self.op[i] != 3:
                self.eventCnt += 1
                self.event[self.eventCnt][0] = self.x[i]   # 边的起点
                self.event[self.eventCnt][1] = self.y[i]   # 边的终点
                self.event[self.eventCnt][2] = i           # 事件发生的时间点
```

```
        # 预处理阶段：初始化并查集，排序事件，构建线段树
        self.prepare()

        # 执行线段树分治的核心算法
        self.dfs(1, self.m, 1)

        # 收集所有查询操作的答案
        result = []
        for i in range(1, self.m + 1):
            if self.op[i] == 3:
                result.append(self.ans[i])

        return result

# 测试代码
if __name__ == "__main__":
    solution = Solution()

    # 测试用例
    n = 5
    operations = [
        [1, 0, 1],  # 添加边(0,1)
        [1, 1, 2],  # 添加边(1,2)
        [3, 0, 2],  # 查询 0 和 2 是否连通
        [2, 1, 2],  # 删除边(1,2)
        [3, 0, 2]   # 查询 0 和 2 是否连通
    ]

    result = solution.dynamic_graph_connectivity(n, operations)
    print(result)  # 应该输出 [True, False]

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