==========
文件夹: class129_ParallelBinarySearchAlgorithms
==========

[Markdown 文件]
==========
文件: additional_problems.md
==========
整体二分算法补充题目汇总

题目分类与详细信息

1. 静态区间第K小问题

1.1 POJ 2104 K-th Number
- **题目链接**: http://poj.org/problem?id=2104
- **题目描述**: 给定一个长度为N的数组，有M个查询，每个查询要求在指定区间内找到第K小的数
- **算法类型**: 整体二分 + 树状数组
- **难度等级**: 2000
- **解题思路**: 使用整体二分处理静态区间第K小问题，将所有查询一起处理，二分答案的值域，利用树状数组维护区间内小于等于mid的元素个数

1.2 HDU 2665 Kth Number
- **题目链接**: http://acm.hdu.edu.cn/showproblem.php?pid=2665
- **题目描述**: 与POJ 2104相同题目，静态区间第K小查询
- **算法类型**: 整体二分 + 树状数组
- **难度等级**: 2000

1.3 BZOJ 2738 矩阵乘法
- **题目链接**: https://www.lydsy.com/JudgeOnline/problem.php?id=2738
- **题目描述**: 给定一个N*N的矩阵，多次求某个子矩阵中的第K小
- **算法类型**: 整体二分 + 二维树状数组
- **难度等级**: 省选/NOI-

2. 带修改的区间第K小问题

2.1 HDU 5412 CRB and Queries
- **题目链接**: http://acm.hdu.edu.cn/showproblem.php?pid=5412
- **题目描述**: 带修改区间第K小查询
- **算法类型**: 整体二分 + 树状数组
- **难度等级**: 2500

2.2 洛谷 P2617 Dynamic Rankings

- **题目链接**: https://www.luogu.com.cn/problem/P2617
- **题目描述**: 带修改区间第K小值
- **算法类型**: 整体二分 + 树状数组
- **难度等级**: 省选/NOI-

2.3 ZOJ 2112 Dynamic Rankings
- **题目链接**: http://acm.zju.edu.cn/onlinejudge/showProblem.do?problemCode=2112
- **题目描述**: 求动态区间第K大
- **算法类型**: 整体二分 + 树状数组
- **难度等级**: 2500

3. 带修改的区间第K大问题

3.1 洛谷 P3332 [ZJOI2013]K大数查询
- **题目链接**: https://www.luogu.com.cn/problem/P3332
- **题目描述**: 给定一个长度为N的数组，支持区间加元素和查询区间第K大
- **算法类型**: 整体二分 + 树状数组
- **难度等级**: 提高+/省选-

4. 混合果汁问题

4.1 洛谷 P4602 [CTSC2018]混合果汁
- **题目链接**: https://www.luogu.com.cn/problem/P4602
- **题目描述**: 多种果汁按不同比例混合，满足小朋友的预算和容量要求
- **算法类型**: 整体二分 + 线段树
- **难度等级**: 省选/NOI-

5. 矩阵相关问题

5.1 洛谷 P1527 [国家集训队]矩阵乘法
- **题目链接**: https://www.luogu.com.cn/problem/P1527
- **题目描述**: 查询子矩阵中第K小的元素
- **算法类型**: 整体二分 + 二维树状数组
- **难度等级**: 省选/NOI-

6. 树上问题

6.1 洛谷 P3242 [HNOI2015]接水果
- **题目链接**: https://www.luogu.com.cn/problem/P3242
- **题目描述**: 树上路径包含关系的查询问题
- **算法类型**: 整体二分 + 扫描线 + 树状数组
- **难度等级**: 省选/NOI-

6.2 洛谷 P3250 [HNOI2016]网络
- **题目链接**: https://www.luogu.com.cn/problem/P3250
- **题目描述**: 树上路径修改和查询问题
- **算法类型**: 整体二分 + 树上差分 + 树状数组
- **难度等级**: 省选/NOI-

7. 国家收集陨石问题

7.1 洛谷 P3527 [POI2011]MET-Meteors
- **题目链接**: https://www.luogu.com.cn/problem/P3527
- **题目描述**: 区间加法和查询满足条件的时间点
- **算法类型**: 整体二分 + 树状数组
- **难度等级**: 省选/NOI-

7.2 SPOJ METEORS
- **题目链接**: https://www.spoj.com/problems/METEORS/
- **题目描述**: 国家收集陨石问题
- **算法类型**: 整体二分 + 树状数组
- **难度等级**: 2500

8. 动态图问题

8.1 洛谷 P5163 WD与地图
- **题目链接**: https://www.luogu.com.cn/problem/P5163
- **题目描述**: 动态图连通性问题
- **算法类型**: 整体二分 + 可撤销并查集
- **难度等级**: NOI/NOI+/CTSC

9. 区间最大异或和问题

9.1 Codeforces 1100F Ivan and Burgers
- **题目链接**: https://codeforces.com/problemset/problem/1100/F
- **题目描述**: 区间最大异或和查询
- **算法类型**: 整体二分 + 线性基
- **难度等级**: 2200

10. 边权最小瓶颈问题

10.1 Codeforces 603E Pastoral Oddities
- **题目链接**: https://codeforces.com/problemset/problem/603/E
- **题目描述**: 边权最小瓶颈生成子图问题
- **算法类型**: 整体二分 + 可撤销并查集
- **难度等级**: 3000

11. 并查集相关二分问题

11.1 AtCoder AGC002D Stamp Rally
- **题目链接**: https://atcoder.jp/contests/agc002/tasks/agc002_d
- **题目描述**: 并查集相关的二分答案问题
- **算法类型**: 整体二分 + 并查集
- **难度等级**: 1700

12. 矩形查询问题

12.1 CodeChef QRECT
- **题目链接**: https://www.codechef.com/problems/QRECT
- **题目描述**: 矩形查询问题
- **算法类型**: 整体二分 + 容斥原理
- **难度等级**: 2500

13. 敌人血量减少问题

13.1 CodeChef MONSTER
- **题目链接**: https://www.codechef.com/problems/MONSTER
- **题目描述**: 敌人血量减少问题
- **算法类型**: 整体二分 + 分块
- **难度等级**: 3000

14. 区间不同元素个数查询

14.1 UVa 12345 Dynamic len(set(a[L:R]))
- **题目链接**:
https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&page=show_problem&problem=3765
- **题目描述**: 区间不同元素个数查询
- **算法类型**: 整体二分 + 树状数组
- **难度等级**: 2500

15. 奶牛跑步问题

15.1 USACO 2015 March Gold - Cow Jog
- **题目链接**: http://www.usaco.org/index.php?page=viewproblem2&cpid=530
- **题目描述**: 奶牛跑步问题
- **算法类型**: 整体二分 + 贪心
- **难度等级**: 1800

16. 树上第 K 小路径问题

16.1 SPOJ COT - Count on a tree
- **题目链接**: https://www.spoj.com/problems/COT/
- **题目描述**: 树上路径第K小
- **算法类型**: 主席树 or 整体二分
- **难度等级**: 2500

16.2 BZOJ 2588 Spoj 10628. Count on a tree
- **题目链接**: https://www.lydsy.com/JudgeOnline/problem.php?id=2588
- **题目描述**: 树上第K小路径
- **算法类型**: 主席树 or 整体二分

题目分类总结

按数据结构分类

1. **树状数组类**:
   - POJ 2104 K-th Number
   - HDU 2665 Kth Number
   - HDU 5412 CRB and Queries
   - 洛谷 P2617 Dynamic Rankings
   - 洛谷 P3332 [ZJOI2013]K大数查询
   - 洛谷 P3527 [POI2011]MET-Meteors
   - UVa 12345 Dynamic len(set(a[L:R]))

2. **线段树类**:
   - 洛谷 P4602 [CTSC2018]混合果汁

3. **二维数据结构类**:
   - 洛谷 P1527 [国家集训队]矩阵乘法
   - BZOJ 2738 矩阵乘法

4. **图论数据结构类**:
   - 洛谷 P3250 [HNOI2016]网络
   - Codeforces 603E Pastoral Oddities
   - 洛谷 P5163 WD与地图
   - AtCoder AGC002D Stamp Rally

5. **特殊数据结构类**:
   - Codeforces 1100F Ivan and Burgers (线性基)
   - 洛谷 P3242 [HNOI2015]接水果 (扫描线)
   - CodeChef MONSTER (分块)

按问题类型分类

1. **区间查询类**:
   - POJ 2104 K-th Number
   - HDU 2665 Kth Number
   - HDU 5412 CRB and Queries
   - 洛谷 P2617 Dynamic Rankings
   - 洛谷 P3332 [ZJOI2013]K 大数查询
   - 洛谷 P1527 [国家集训队]矩阵乘法
   - BZOJ 2738 矩阵乘法
   - UVa 12345 Dynamic len(set(a[L:R]))

2. **树上问题类**:
   - 洛谷 P3242 [HNOI2015]接水果
   - 洛谷 P3250 [HNOI2016]网络
   - Codeforces 1100F Ivan and Burgers
   - SPOJ COT - Count on a tree
   - BZOJ 2588 Spoj 10628. Count on a tree

3. **图论问题类**:
   - Codeforces 603E Pastoral Oddities
   - 洛谷 P5163 WD 与地图
   - AtCoder AGC002D Stamp Rally

4. **优化问题类**:
   - 洛谷 P4602 [CTSC2018]混合果汁
   - USACO 2015 March Gold - Cow Jog

5. **容斥原理类**:
   - CodeChef QRECT

学习建议

入门阶段
1. 先掌握基础的整体二分思想
2. 练习简单的区间查询问题，如 POJ 2104 K-th Number
3. 理解分治过程的实现

进阶阶段
1. 学习各种数据结构在整体二分中的应用
2. 练习树上问题和图论问题
3. 掌握复杂问题的建模方法

提高阶段
1. 研究整体二分与其他算法的结合
2. 学习优化技巧和实现细节
3. 解决高难度的竞赛题目

常见陷阱及解决方案

1. 状态撤销问题
- **问题描述**: 在分治过程中需要恢复之前的状态
- **解决方案**: 使用可撤销数据结构或手动撤销操作

2. 操作顺序问题
- **问题描述**: 操作顺序影响判定结果
- **解决方案**: 严格按照时间顺序处理操作

3. 边界处理问题
- **问题描述**: 边界条件处理不当导致错误
- **解决方案**: 仔细分析边界情况，进行特殊处理

==========

文件: problems.md
==========
整体二分相关题目汇总

洛谷平台题目

1. P3332 [ZJOI2013]K 大数查询
- 题目链接: https://www.luogu.com.cn/problem/P3332
- 题目描述: 给定一个长度为 N 的数组，支持区间加元素和查询区间第 K 大
- 算法类型: 整体二分 + 树状数组
- 难度等级: 提高+/省选-

2. P2617 Dynamic Rankings
- 题目链接: https://www.luogu.com.cn/problem/P2617
- 题目描述: 带修改区间第 k 小值
- 算法类型: 整体二分 + 树状数组
- 难度等级: 省选/NOI-

3. P4602 [CTSC2018]混合果汁
- 题目链接: https://www.luogu.com.cn/problem/P4602
- 题目描述: 多种果汁按不同比例混合，满足小朋友的预算和容量要求
- 算法类型: 整体二分 + 线段树

- 难度等级: 省选/NOI-

4. P1527 [国家集训队]矩阵乘法
- 题目链接: https://www.luogu.com.cn/problem/P1527
- 题目描述: 查询子矩阵中第 k 小的元素
- 算法类型: 整体二分 + 二维树状数组
- 难度等级: 省选/NOI-

5. P3242 [HNOI2015]接水果
- 题目链接: https://www.luogu.com.cn/problem/P3242
- 题目描述: 树上路径包含关系的查询问题
- 算法类型: 整体二分 + 扫描线 + 树状数组
- 难度等级: 省选/NOI-

6. P3250 [HNOI2016]网络
- 题目链接: https://www.luogu.com.cn/problem/P3250
- 题目描述: 树上路径修改和查询问题
- 算法类型: 整体二分 + 树上差分 + 树状数组
- 难度等级: 省选/NOI-

7. P3527 [POI2011]MET-Meteors
- 题目链接: https://www.luogu.com.cn/problem/P3527
- 题目描述: 区间加法和查询满足条件的时间点
- 算法类型: 整体二分 + 树状数组
- 难度等级: 省选/NOI-

8. P5163 WD与地图
- 题目链接: https://www.luogu.com.cn/problem/P5163
- 题目描述: 动态图连通性问题
- 算法类型: 整体二分 + 可撤销并查集
- 难度等级: NOI/NOI+/CTSC

Codeforces平台题目

1. CF1100F Ivan and Burgers
- 题目链接: https://codeforces.com/problemset/problem/1100/F
- 题目描述: 区间最大异或和查询
- 算法类型: 整体二分 + 线性基
- 难度等级: 2200

2. CF603E Pastoral Oddities
- 题目链接: https://codeforces.com/problemset/problem/603/E
- 题目描述: 边权最小瓶颈生成子图问题

- 算法类型: 整体二分 + 可撤销并查集
- 难度等级: 3000

AtCoder 平台题目

1. AGC002D Stamp Rally
- 题目链接: https://atcoder.jp/contests/agc002/tasks/agc002_d
- 题目描述: 并查集相关的二分答案问题
- 算法类型: 整体二分 + 并查集
- 难度等级: 1700

牛客网平台题目

1. 牛客练习赛 A 区间第 k 小
- 题目描述: 带修改的区间第 k 小查询
- 算法类型: 整体二分 + 树状数组

POJ 平台题目

1. POJ 2104 K-th Number
- 题目链接: http://poj.org/problem?id=2104
- 题目描述: 静态区间第 k 小查询
- 算法类型: 整体二分 + 树状数组
- 难度等级: 2000

HDU 平台题目

1. HDU 2665 Kth Number
- 题目链接: http://acm.hdu.edu.cn/showproblem.php?pid=2665
- 题目描述: 静态区间第 k 小查询
- 算法类型: 整体二分 + 树状数组
- 难度等级: 2000

2. HDU 5412 CRB and Queries
- 题目链接: http://acm.hdu.edu.cn/showproblem.php?pid=5412
- 题目描述: 带修改区间第 k 小查询
- 算法类型: 整体二分 + 树状数组
- 难度等级: 2500

SPOJ 平台题目

1. SPOJ METEORS
- 题目链接: https://www.spoj.com/problems/METEORS/

- 题目描述: 国家收集陨石问题
- 算法类型: 整体二分 + 树状数组
- 难度等级: 2500

2. SPOJ COT - Count on a tree
- 题目描述: 树上路径第 k 小
- 算法类型: 主席树 or 整体二分

CodeChef 平台题目

1. CodeChef QRECT
- 题目链接: https://www.codechef.com/problems/QRECT
- 题目描述: 矩形查询问题
- 算法类型: 整体二分 + 容斥原理
- 难度等级: 2500

2. CodeChef MONSTER
- 题目链接: https://www.codechef.com/problems/MONSTER
- 题目描述: 敌人血量减少问题
- 算法类型: 整体二分 + 分块
- 难度等级: 3000

UVa 平台题目

1. UVa 12345 Dynamic len(set(a[L:R]))
- 题目链接:
https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&page=show_problem&problem=3765
- 题目描述: 区间不同元素个数查询
- 算法类型: 整体二分 + 树状数组
- 难度等级: 2500

USACO 平台题目

1. USACO 2015 March Gold - Cow Jog
- 题目链接: http://www.usaco.org/index.php?page=viewproblem2&cpid=530
- 题目描述: 奶牛跑步问题
- 算法类型: 整体二分 + 贪心
- 难度等级: 1800

其他平台题目

1. BZOJ 2588 Spoj 10628. Count on a tree
- 题目描述: 树上第 k 小路径

- 算法类型: 主席树 or 整体二分

题目分类

按数据结构分类

1. 树状数组类
- P3332 [ZJOI2013]K 大数查询
- P2617 Dynamic Rankings
- P3527 [POI2011]MET-Meteors
- POJ 2104 K-th Number
- HDU 2665 Kth Number
- HDU 5412 CRB and Queries
- UVa 12345 Dynamic len(set(a[L:R]))

2. 线段树类
- P4602 [CTSC2018]混合果汁

3. 二维数据结构类
- P1527 [国家集训队]矩阵乘法

4. 图论数据结构类
- P3250 [HNOI2016]网络
- CF603E Pastoral Oddities
- P5163 WD 与地图
- AGC002D Stamp Rally

5. 特殊数据结构类
- CF1100F Ivan and Burgers (线性基)
- P3242 [HNOI2015]接水果 (扫描线)
- CodeChef MONSTER (分块)

按问题类型分类

1. 区间查询类
- P3332 [ZJOI2013]K 大数查询
- P2617 Dynamic Rankings
- P1527 [国家集训队]矩阵乘法
- POJ 2104 K-th Number
- HDU 2665 Kth Number
- HDU 5412 CRB and Queries
- UVa 12345 Dynamic len(set(a[L:R]))

2. 树上问题类
- P3242 [HNOI2015]接水果
- P3250 [HNOI2016]网络
- CF1100F Ivan and Burgers
- SPOJ COT - Count on a tree
- BZOJ 2588 Spoj 10628. Count on a tree

3. 图论问题类
- CF603E Pastoral Oddities
- P5163 WD与地图
- AGC002D Stamp Rally

4. 优化问题类
- P4602 [CTSC2018]混合果汁
- USACO 2015 March Gold - Cow Jog

5. 容斥原理类
- CodeChef QRECT

解题技巧总结

1. 确定二分对象
- 值域（如权值、时间等）
- 位置（如区间端点等）

2. 设计判定过程
- 利用合适的数据结构维护状态
- 确保判定过程满足整体二分的要求

3. 处理操作分类
- 将修改操作按与mid的关系分类
- 将查询操作按判定结果分类

4. 优化实现细节
- 合理使用数据结构减少时间复杂度
- 注意空间复杂度的控制
- 处理边界条件和特殊情况

常见陷阱及解决方案

1. 状态撤销问题
- 问题描述：在分治过程中需要恢复之前的状态
- 解决方案：使用可撤销数据结构或手动撤销操作

2. 操作顺序问题
- 问题描述：操作顺序影响判定结果
- 解决方案：严格按照时间顺序处理操作

3. 边界处理问题
- 问题描述：边界条件处理不当导致错误
- 解决方案：仔细分析边界情况，进行特殊处理

学习建议

1. 入门阶段
- 先掌握基础的整体二分思想
- 练习简单的区间查询问题
- 理解分治过程的实现

2. 进阶阶段
- 学习各种数据结构在整体二分中的应用
- 练习树上问题和图论问题
- 掌握复杂问题的建模方法

3. 提高阶段
- 研究整体二分与其他算法的结合
- 学习优化技巧和实现细节
- 解决高难度的竞赛题目

==========

文件：README.md
==========
整体二分算法详解

什么是整体二分

整体二分（Parallel Binary Search）是一种离线算法，用于解决多个具有相同结构的二分答案问题。它的核心思想是将所有查询放在一起进行二分，而不是对每个查询单独进行二分。

适用条件

整体二分适用于满足以下性质的问题：

1. 询问的答案具有可二分性
2. 修改对判定答案的贡献互相独立，修改之间互不影响效果

3. 修改如果对判定答案有贡献，则贡献为一确定的与判定标准无关的值
4. 贡献满足交换律、结合律，具有可加性
5. 题目允许使用离线算法

算法原理

整体二分的基本思路是：
1. 将所有操作（包括修改和查询）按时间顺序排列
2. 对答案进行分治处理
3. 在每一层分治中，利用数据结构统计当前查询的答案与中点值的关系
4. 根据比较结果将操作分为两部分，分别递归处理

经典题目解析

1. 混合果汁（Code01_Juice）

题目描述
有 n 种果汁，每种果汁有美味度 d、每升价格 p、添加上限 l。制作混合果汁时每种果汁不能超过添加上限，其中美味度最低的果汁决定混合果汁的美味度。有 m 个小朋友，给每位制作混合果汁时，钱数不超过 money[i]，体积不少于 least[i]。打印每个小朋友能得到的混合果汁最大美味度，如果无法满足，打印-1。

解题思路
1. 将果汁按美味度从高到低排序
2. 使用整体二分，二分美味度范围
3. 对于每个美味度范围，维护线段树记录价格和数量信息
4. 判断是否能满足小朋友的需求

时间复杂度
- O((n+m) * log(n) * log(max_p))

空间复杂度
- O(n * log(max_p))

2. 带修改的区间第 k 小（Code02_DynamicRankings）

题目描述
给定一个长度为 n 的数组 arr，接下来是 m 条操作，每种操作是如下两种类型的一种：
- 操作 C x y：把 x 位置的值修改成 y
- 操作 Q x y v：查询 arr[x..y]范围上第 v 小的值

解题思路
1. 将所有操作（修改和查询）按时间顺序处理
2. 使用整体二分，二分值域范围

3. 利用树状数组维护区间内小于等于某个值的元素个数
4. 根据查询结果将操作分为两部分递归处理

时间复杂度
- O((n+m) * log(n) * log(max_value))

空间复杂度
- O(n)

3. 网络（Code03_Network）

题目描述
有 n 个服务器连成一棵树，某些服务器之间有请求路径。操作包括：
- 添加路径请求
- 删除路径请求
- 查询与某个服务器无关的所有请求中最大的重要度

解题思路
1. 使用树上差分技术处理路径修改
2. 结合 LCA 算法处理树上路径
3. 使用整体二分，二分重要度范围
4. 利用树状数组维护子树信息

时间复杂度
- O(m * log(m) * log(max_importance))

空间复杂度
- O(n)

4. 接水果（Code04_Fruit）

题目描述
给定一棵树，有 p 个盘子（路径）和 q 个水果（路径），如果一个盘子路径完全在一个水果路径的内部，那么该盘子可以接住该水果。对于每个水果，打印可以接住它的盘子中第 k 小的权值。

解题思路
1. 使用 DFS 序将树上问题转化为区间问题
2. 利用扫描线技术处理二维偏序问题
3. 使用整体二分，二分盘子权值范围
4. 通过树状数组维护满足条件的盘子数量

时间复杂度
- O((p+q) * log(p) * log(max_weight))

空间复杂度
- O(n)

5. 点的度都是奇数的最小瓶颈（Code05_PastoralOddities）

题目描述
有 n 个点，初始没有边，依次加入 m 条无向边，每条边有边权。每次加入后，询问是否存在一个边集，满足每个点连接的边的数量都是奇数。如果存在，希望边集的最大边权尽可能小。

解题思路
1. 利用图论知识：每个点度数为奇数的边集存在的充要条件是每个连通分量大小都是偶数
2. 使用可撤销并查集维护连通性
3. 使用整体二分，二分边权范围
4. 通过并查集判断当前状态是否满足条件

时间复杂度
- O(m * log(m) * log(n))

空间复杂度
- O(n)

6. 范围最大异或和（Code06_IvanAndBurgers）

题目描述
给定一个长度为 n 的数组 arr，有 q 条查询，每条查询格式为 l r，要求在 arr[l..r]中选若干个数，打印最大的异或和。

解题思路
1. 使用线性基处理异或问题
2. 预处理前缀线性基
3. 使用整体二分，二分查询区间
4. 通过线性基合并处理跨越中点的查询

时间复杂度
- O((n+q) * log(n) * log(max_value))

空间复杂度
- O(n * log(max_value))

算法模板

```java
```

```
public static void compute(int el, int er, int vl, int vr) {
    if (el > er) {
        return;
    }
    if (vl == vr) {
        // 找到答案，处理所有查询
        for (int i = el; i <= er; i++) {
            // 处理查询
        }
    } else {
        int mid = (vl + vr) >> 1;
        // 将操作分为两部分
        int lsiz = 0, rsiz = 0;
        for (int i = el; i <= er; i++) {
            // 根据 mid 值将操作分类
            if (/* 条件 */) {
                lset[++lsiz] = /* 操作 */;
            } else {
                rset[++rsiz] = /* 操作 */;
            }
        }
        // 重新排列操作顺序
        for (int i = 1; i <= lsiz; i++) {
            eid[el + i - 1] = lset[i];
        }
        for (int i = 1; i <= rsiz; i++) {
            eid[el + lsiz + i - 1] = rset[i];
        }
        // 递归处理两部分
        compute(el, el + lsiz - 1, vl, mid);
        compute(el + lsiz, er, mid + 1, vr);
    }
}
```

工程化考量

异常处理
1. 输入验证：检查参数范围和格式
2. 边界条件：处理空输入、极值等特殊情况
3. 内存管理：合理分配和释放内存

性能优化

1. 数据结构选择：根据具体问题选择合适的数据结构
2. 算法优化：减少不必要的计算和内存访问
3. 缓存友好：优化数据访问模式

可配置性
1. 参数化：将关键参数设计为可配置项
2. 模块化：将功能拆分为独立模块

线程安全
1. 对于多线程环境，需要考虑数据竞争问题
2. 可以通过加锁或使用线程安全的数据结构解决

与其他算法的对比

与主席树的对比
1. 整体二分：离线算法，空间复杂度较低
2. 主席树：在线算法，支持实时查询

与树套树的对比
1. 整体二分：代码实现相对简单
2. 树套树：功能更强大，但实现复杂度高

应用场景

整体二分广泛应用于以下场景：
1. 区间查询问题（如第 k 小值）
2. 动态图问题（如连通性维护）
3. 树上问题（如路径查询）
4. 异或相关问题（如最大异或和）
5. 优化问题（如最小瓶颈问题）

扩展题目

POJ 平台
1. POJ 2104 K-th Number：静态区间第 k 小查询，是整体二分的经典入门题

HDU 平台
1. HDU 2665 Kth Number：与 POJ 2104 相同题目，静态区间第 k 小查询
2. HDU 5412 CRB and Queries：带修改区间第 k 小查询，是整体二分的进阶题

SPOJ 平台
1. SPOJ METEORS：国家收集陨石问题，需要使用整体二分结合树状数组解决

CodeChef 平台
1. CodeChef QRECT：矩形查询问题，需要使用整体二分结合容斥原理解决
2. CodeChef MONSTER：敌人血量减少问题，需要使用整体二分结合分块解决

UVa 平台
1. UVa 12345 Dynamic len(set(a[L:R]))：区间不同元素个数查询，可以使用整体二分解决

USACO 平台
1. USACO 2015 March Gold - Cow Jog：奶牛跑步问题，可以使用整体二分结合贪心解决

AtCoder 平台
1. AGC002D Stamp Rally：并查集相关的二分答案问题，是整体二分的经典应用

洛谷平台
1. 洛谷 P3242 [HNOI2015]接水果
   - 题目描述：给定一棵树，有 p 个盘子（路径）和 q 个水果（路径），如果一个盘子路径完全在一个水果路径的内部，那么该盘子可以接住该水果。对于每个水果，打印可以接住它的盘子中第 k 小的权值。
   - 解题思路：使用 DFS 序将树上问题转化为平面矩形区域，结合扫描线和树状数组进行区间覆盖和查询，通过整体二分求解第 k 小问题。
   - 时间复杂度：O((p+q) * log(p) * log(max_weight))
   - 代码实现：已在 supplementary_solutions 目录下提供 Java、Python 和 C++版本

2. 洛谷 P4602 [CTSC2018]混合果汁
   - 题目描述：有 n 种果汁，每种果汁有美味度 d，单价 p，数量 l。现在有 m 个询问，每个询问给出需要的总数量 g 和最高预算 v，要求选一些果汁，使得总数量至少 g，总费用不超过 v，并且所选果汁的最低美味度尽可能大。
   - 解题思路：使用整体二分法二分可能的最低美味度，对于每个美味度，使用线段树维护价格和数量信息，查询在预算下最多能购买的果汁数量。
   - 时间复杂度：O((n+m) * log(n) * log(max_p))
   - 代码实现：已在 supplementary_solutions 目录下提供 Java、Python 和 C++版本

总结

整体二分是一种强大的离线算法，能够有效解决多个具有相同结构的二分答案问题。通过将所有查询一起处理，避免了重复计算，提高了效率。掌握整体二分对于解决复杂的算法问题具有重要意义。

整体二分的关键在于：
1. 理解其分治思想
2. 掌握适用条件
3. 熟练使用相关数据结构
4. 能够将具体问题转化为整体二分模型

通过大量练习经典题目，可以加深对整体二分算法的理解，并提高解决实际问题的能力。

==========

文件: README_FINAL.md
==========
整体二分算法详解与实践

项目概述

本项目系统地介绍了整体二分算法的原理、应用和实现，包含详细的代码注释和多种编程语言的实现版本。通过本项目的学习，你可以深入理解整体二分算法的核心思想，并掌握其在解决各类算法问题中的应用。

目录结构

```
class169/
├── 基础实现文件
│   ├── Code01_Juice1.java - 混合果汁问题
│   ├── Code02_DynamicRankings1.java - 带修改的区间第 k 小
│   ├── Code03_Network1.java - 网络问题
│   ├── Code04_Fruit1.java - 接水果问题
│   ├── Code05_PastoralOddities1.java - 点的度都是奇数的最小瓶颈
│   ├── Code06_IvanAndBurgers1.java - 范围最大异或和
│   ├── Code07_KthNumber1.java - POJ 2104 K-th Number (Java)
│   ├── Code07_KthNumber1.py - POJ 2104 K-th Number (Python)
│   ├── Code07_KthNumber1.cpp - POJ 2104 K-th Number (C++)
│   ├── Code08_CRBAndQueries1.java - HDU 5412 CRB and Queries
│   ├── Code09_Meteors1.java - SPOJ METEORS
│   └── Code10_StampRally1.java - AGC002D Stamp Rally
├── 补充题目和解答
│   ├── additional_problems.md - 补充题目汇总
│   ├── supplementary_solutions/ - 补充题目的三种语言实现
│   │   ├── POJ2104_KthNumber.java
│   │   ├── POJ2104_KthNumber.py
│   │   ├── POJ2104_KthNumber.cpp
│   │   ├── HDU2665_KthNumber.java
│   │   ├── HDU2665_KthNumber.py
│   │   └── HDU2665_KthNumber.cpp
├── 学习资料
│   ├── README.md - 原始说明文档
│   ├── problems.md - 题目汇总
│   ├── solutions.md - 解题详解
│   ├── training_plan.md - 训练计划
```

```
|   └── additional_problems.md - 补充题目
└── 其他文件
    └── ... (其他辅助文件)
```

核心算法介绍

什么是整体二分

整体二分（Parallel Binary Search）是一种离线算法，用于解决多个具有相同结构的二分答案问题。它的核心思想是将所有查询放在一起进行二分，而不是对每个查询单独进行二分。

适用条件

整体二分适用于满足以下性质的问题：
1. 询问的答案具有可二分性
2. 修改对判定答案的贡献互相独立，修改之间互不影响效果
3. 修改如果对判定答案有贡献，则贡献为一确定的与判定标准无关的值
4. 贡献满足交换律、结合律，具有可加性
5. 题目允许使用离线算法

经典题目解析

1. 混合果汁（Code01_Juice1）
- **题目描述**: 多种果汁按不同比例混合，满足小朋友的预算和容量要求
- **解题思路**: 将果汁按美味度从高到低排序，使用整体二分，二分美味度范围
- **时间复杂度**: O((n+m) * log(n) * log(max_p))

2. 带修改的区间第 k 小（Code02_DynamicRankings1）
- **题目描述**: 支持区间修改和查询第 k 小值
- **解题思路**: 将所有操作（修改和查询）按时间顺序处理，使用整体二分，二分值域范围
- **时间复杂度**: O((n+m) * log(n) * log(max_value))

3. 网络（Code03_Network1）
- **题目描述**: 树上路径修改和查询问题
- **解题思路**: 使用树上差分技术处理路径修改，结合 LCA 算法处理树上路径
- **时间复杂度**: O(m * log(m) * log(max_importance))

4. 接水果（Code04_Fruit1）
- **题目描述**: 树上路径包含关系的查询问题
- **解题思路**: 使用 DFS 序将树上问题转化为区间问题，利用扫描线技术处理二维偏序问题
- **时间复杂度**: O((p+q) * log(p) * log(max_weight))

5. 点的度都是奇数的最小瓶颈（Code05_PastoralOddities1）
- **题目描述**: 边集的最大边权尽可能小
- **解题思路**: 利用图论知识，使用可撤销并查集维护连通性
- **时间复杂度**: O(m * log(m) * log(n))

6. 范围最大异或和（Code06_IvanAndBurgers1）
- **题目描述**: 区间内选若干个数，使得它们的异或和最大
- **解题思路**: 使用线性基处理异或问题，预处理前缀线性基
- **时间复杂度**: O((n+q) * log(n) * log(max_value))

算法模板

```java
public static void compute(int el, int er, int vl, int vr) {
    if (el > er) {
        return;
    }
    if (vl == vr) {
        // 找到答案，处理所有查询
        for (int i = el; i <= er; i++) {
            // 处理查询
        }
    } else {
        int mid = (vl + vr) >> 1;
        // 将操作分为两部分
        int lsiz = 0, rsiz = 0;
        for (int i = el; i <= er; i++) {
            // 根据 mid 值将操作分类
            if (/* 条件 */) {
                lset[++lsiz] = /* 操作 */;
            } else {
                rset[++rsiz] = /* 操作 */;
            }
        }
        // 重新排列操作顺序
        for (int i = 1; i <= lsiz; i++) {
            eid[el + i - 1] = lset[i];
        }
        for (int i = 1; i <= rsiz; i++) {
            eid[el + lsiz + i - 1] = rset[i];
        }
        // 递归处理两部分
        compute(el, el + lsiz - 1, vl, mid);
```

```
        compute(el + lsiz, er, mid + 1, vr);
    }
}
```

工程化考量

异常处理
1. 输入验证：检查参数范围和格式
2. 边界条件：处理空输入、极值等特殊情况
3. 内存管理：合理分配和释放内存

性能优化
1. 数据结构选择：根据具体问题选择合适的数据结构
2. 算法优化：减少不必要的计算和内存访问
3. 缓存友好：优化数据访问模式

可配置性
1. 参数化：将关键参数设计为可配置项
2. 模块化：将功能拆分为独立模块

线程安全
1. 对于多线程环境，需要考虑数据竞争问题
2. 可以通过加锁或使用线程安全的数据结构解决

与其他算法的对比

与主席树的对比
1. 整体二分：离线算法，空间复杂度较低
2. 主席树：在线算法，支持实时查询

与树套树的对比
1. 整体二分：代码实现相对简单
2. 树套树：功能更强大，但实现复杂度高

应用场景

整体二分广泛应用于以下场景：
1. 区间查询问题（如第 k 小值）
2. 动态图问题（如连通性维护）
3. 树上问题（如路径查询）
4. 异或相关问题（如最大异或和）
5. 优化问题（如最小瓶颈问题）

补充题目

我们还提供了丰富的补充题目，涵盖各大算法平台：
- **POJ 平台**: POJ 2104 K-th Number
- **HDU 平台**: HDU 2665 Kth Number, HDU 5412 CRB and Queries
- **洛谷平台**: P3332 [ZJOI2013]K 大数查询, P2617 Dynamic Rankings 等
- **Codeforces 平台**: CF1100F Ivan and Burgers, CF603E Pastoral Oddities
- **AtCoder 平台**: AGC002D Stamp Rally
- **SPOJ 平台**: SPOJ METEORS
- **CodeChef 平台**: CodeChef QRECT, CodeChef MONSTER
- **UVa 平台**: UVa 12345 Dynamic len(set(a[L:R]))
- **USACO 平台**: USACO 2015 March Gold - Cow Jog

训练计划

我们提供了详细的训练计划，帮助你系统地掌握整体二分算法：
1. **入门阶段**: 掌握基础的整体二分思想
2. **进阶阶段**: 学习各种数据结构在整体二分中的应用
3. **提高阶段**: 研究整体二分与其他算法的结合

总结

整体二分是一种强大的离线算法，能够有效解决多个具有相同结构的二分答案问题。通过将所有查询一起处理，避免了重复计算，提高了效率。掌握整体二分对于解决复杂的算法问题具有重要意义。

整体二分的关键在于：
1. 理解其分治思想
2. 掌握适用条件
3. 熟练使用相关数据结构
4. 能够将具体问题转化为整体二分模型

通过大量练习经典题目，可以加深对整体二分算法的理解，并提高解决实际问题的能力。

==========

文件: solutions.md
==========
整体二分题目详解与实现

1. P3332 [ZJOI2013]K 大数查询 - 带修改区间第 K 大

题目描述

给定一个长度为N的数组，支持以下操作：
1. 区间加元素
2. 查询区间第K大

解题思路
使用整体二分，将所有操作（包括修改和查询）一起处理。二分答案的值域，利用树状数组维护区间内小于等于mid的元素个数。

Java实现

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

public class P3332_KthNumber {
    static final int MAXN = 50005;
    static final long INF = 1000000001L;

    static int n, m;
    static long[] ans = new long[MAXN];

    // 操作信息
    static int[] op = new int[MAXN * 2];  // 1: add, 2: query
    static int[] x = new int[MAXN * 2];
    static int[] y = new int[MAXN * 2];
    static long[] v = new long[MAXN * 2];
    static int[] qid = new int[MAXN * 2];  // 查询编号

    // 树状数组
    static long[] tree = new long[MAXN];

    // 整体二分相关
    static int[] eid = new int[MAXN * 2];
    static int[] lset = new int[MAXN * 2];
    static int[] rset = new int[MAXN * 2];

    static int cnt = 0;

    static int lowbit(int x) {
        return x & (-x);
    }

    static void add(int pos, long val) {
```

```
    while (pos <= n) {
        tree[pos] += val;
        pos += lowbit(pos);
    }
}

static long sum(int pos) {
    long ret = 0;
    while (pos > 0) {
        ret += tree[pos];
        pos -= lowbit(pos);
    }
    return ret;
}

static long query(int l, int r) {
    return sum(r) - sum(l - 1);
}

static void compute(int el, int er, long vl, long vr) {
    if (el > er) return;

    if (vl == vr) {
        for (int i = el; i <= er; i++) {
            int id = eid[i];
            if (op[id] == 2) {
                ans[qid[id]] = vl;
            }
        }
        return;
    }

    long mid = (vl + vr) >> 1;
    int lsiz = 0, rsiz = 0;

    for (int i = el; i <= er; i++) {
        int id = eid[i];
        if (op[id] == 1) {  // 修改操作
            if (v[id] <= mid) {
                add(x[id], 1);
                lset[++lsiz] = id;
            } else {
                rset[++rsiz] = id;
```

```
            }
        } else {  // 查询操作
            long cnt = query(x[id], y[id]);
            if (v[id] <= cnt) {
                lset[++lsiz] = id;
            } else {
                v[id] -= cnt;
                rset[++rsiz] = id;
            }
        }
    }

    // 撤销修改操作的影响
    for (int i = 1; i <= lsiz; i++) {
        int id = lset[i];
        if (op[id] == 1 && v[id] <= mid) {
            add(x[id], -1);
        }
    }

    // 重新排列操作顺序
    for (int i = 1; i <= lsiz; i++) {
        eid[el + i - 1] = lset[i];
    }
    for (int i = 1; i <= rsiz; i++) {
        eid[el + lsiz + i - 1] = rset[i];
    }

    compute(el, el + lsiz - 1, vl, mid);
    compute(el + lsiz, er, mid + 1, vr);
}

public static void main(String[] args) throws IOException {
    BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
    PrintWriter out = new PrintWriter(System.out);

    String[] tokens = br.readLine().split(" ");
    n = Integer.parseInt(tokens[0]);
    m = Integer.parseInt(tokens[1]);

    int queryCnt = 0;
    for (int i = 1; i <= m; i++) {
        tokens = br.readLine().split(" ");
```

```
            op[i] = Integer.parseInt(tokens[0]);
            if (op[i] == 1) {
                x[i] = Integer.parseInt(tokens[1]);
                y[i] = Integer.parseInt(tokens[2]);
                v[i] = Long.parseLong(tokens[3]);
                eid[++cnt] = i;
            } else {
                x[i] = Integer.parseInt(tokens[1]);
                y[i] = Integer.parseInt(tokens[2]);
                v[i] = Long.parseLong(tokens[3]);
                qid[i] = ++queryCnt;
                eid[++cnt] = i;
            }
        }

        compute(1, cnt, -INF, INF);

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

C++实现

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

const int MAXN = 50005;
const long long INF = 1000000001LL;

int n, m;
long long ans[MAXN];

// 操作信息
int op[MAXN * 2];  // 1: add, 2: query
int x[MAXN * 2];
int y[MAXN * 2];
long long v[MAXN * 2];
int qid[MAXN * 2];  // 查询编号
```

```
// 树状数组
long long tree[MAXN];

// 整体二分相关
int eid[MAXN * 2];
int lset[MAXN * 2];
int rset[MAXN * 2];

int cnt = 0;

int lowbit(int x) {
    return x & (-x);
}

void add(int pos, long long val) {
    while (pos <= n) {
        tree[pos] += val;
        pos += lowbit(pos);
    }
}

long long sum(int pos) {
    long long ret = 0;
    while (pos > 0) {
        ret += tree[pos];
        pos -= lowbit(pos);
    }
    return ret;
}

long long query(int l, int r) {
    return sum(r) - sum(l - 1);
}

void compute(int el, int er, long long vl, long long vr) {
    if (el > er) return;

    if (vl == vr) {
        for (int i = el; i <= er; i++) {
            int id = eid[i];
            if (op[id] == 2) {
                ans[qid[id]] = vl;
```

```
        }
    }
    return;
}

long long mid = (vl + vr) >> 1;
int lsiz = 0, rsiz = 0;

for (int i = el; i <= er; i++) {
    int id = eid[i];
    if (op[id] == 1) {  // 修改操作
        if (v[id] <= mid) {
            add(x[id], 1);
            lset[++lsiz] = id;
        } else {
            rset[++rsiz] = id;
        }
    } else {  // 查询操作
        long long cnt = query(x[id], y[id]);
        if (v[id] <= cnt) {
            lset[++lsiz] = id;
        } else {
            v[id] -= cnt;
            rset[++rsiz] = id;
        }
    }
}

// 撤销修改操作的影响
for (int i = 1; i <= lsiz; i++) {
    int id = lset[i];
    if (op[id] == 1 && v[id] <= mid) {
        add(x[id], -1);
    }
}

// 重新排列操作顺序
for (int i = 1; i <= lsiz; i++) {
    eid[el + i - 1] = lset[i];
}
for (int i = 1; i <= rsiz; i++) {
    eid[el + lsiz + i - 1] = rset[i];
}
```

```
    compute(el, el + lsiz - 1, vl, mid);
    compute(el + lsiz, er, mid + 1, vr);
}

int main() {
    ios::sync_with_stdio(false);
    cin.tie(0);

    cin >> n >> m;

    int queryCnt = 0;
    for (int i = 1; i <= m; i++) {
        cin >> op[i];
        if (op[i] == 1) {
            cin >> x[i] >> y[i] >> v[i];
            eid[++cnt] = i;
        } else {
            cin >> x[i] >> y[i] >> v[i];
            qid[i] = ++queryCnt;
            eid[++cnt] = i;
        }
    }

    compute(1, cnt, -INF, INF);

    for (int i = 1; i <= queryCnt; i++) {
        cout << ans[i] << "\n";
    }

    return 0;
}
```

Python 实现

```python
import sys
from bisect import bisect_left

class Solution:
    def __init__(self):
        self.MAXN = 50005
```

```
        self.INF = 1000000001

        self.n = 0
        self.m = 0
        self.ans = [0] * self.MAXN

        # 操作信息
        self.op = [0] * (self.MAXN * 2)  # 1: add, 2: query
        self.x = [0] * (self.MAXN * 2)
        self.y = [0] * (self.MAXN * 2)
        self.v = [0] * (self.MAXN * 2)
        self.qid = [0] * (self.MAXN * 2)  # 查询编号

        # 树状数组
        self.tree = [0] * self.MAXN

        # 整体二分相关
        self.eid = [0] * (self.MAXN * 2)
        self.lset = [0] * (self.MAXN * 2)
        self.rset = [0] * (self.MAXN * 2)

        self.cnt = 0

    def lowbit(self, x):
        return x & (-x)

    def add(self, pos, val):
        while pos <= self.n:
            self.tree[pos] += val
            pos += self.lowbit(pos)

    def sum(self, pos):
        ret = 0
        while pos > 0:
            ret += self.tree[pos]
            pos -= self.lowbit(pos)
        return ret

    def query(self, l, r):
        return self.sum(r) - self.sum(l - 1)

    def compute(self, el, er, vl, vr):
        if el > er:
```

```
        return

    if vl == vr:
        for i in range(el, er + 1):
            id = self.eid[i]
            if self.op[id] == 2:
                self.ans[self.qid[id]] = vl
        return

    mid = (vl + vr) // 2
    lsiz = 0
    rsiz = 0

    for i in range(el, er + 1):
        id = self.eid[i]
        if self.op[id] == 1:  # 修改操作
            if self.v[id] <= mid:
                self.add(self.x[id], 1)
                lsiz += 1
                self.lset[lsiz] = id
            else:
                rsiz += 1
                self.rset[rsiz] = id
        else:  # 查询操作
            cnt = self.query(self.x[id], self.y[id])
            if self.v[id] <= cnt:
                lsiz += 1
                self.lset[lsiz] = id
            else:
                self.v[id] -= cnt
                rsiz += 1
                self.rset[rsiz] = id

    # 撤销修改操作的影响
    for i in range(1, lsiz + 1):
        id = self.lset[i]
        if self.op[id] == 1 and self.v[id] <= mid:
            self.add(self.x[id], -1)

    # 重新排列操作顺序
    for i in range(1, lsiz + 1):
        self.eid[el + i - 1] = self.lset[i]
    for i in range(1, rsiz + 1):
```

```
            self.eid[el + lsiz + i - 1] = self.rset[i]

        self.compute(el, el + lsiz - 1, vl, mid)
        self.compute(el + lsiz, er, mid + 1, vr)

    def solve(self):
        line = sys.stdin.readline().split()
        self.n = int(line[0])
        self.m = int(line[1])

        queryCnt = 0
        for i in range(1, self.m + 1):
            line = sys.stdin.readline().split()
            self.op[i] = int(line[0])
            if self.op[i] == 1:
                self.x[i] = int(line[1])
                self.y[i] = int(line[2])
                self.v[i] = int(line[3])
                self.cnt += 1
                self.eid[self.cnt] = i
            else:
                self.x[i] = int(line[1])
                self.y[i] = int(line[2])
                self.v[i] = int(line[3])
                queryCnt += 1
                self.qid[i] = queryCnt
                self.cnt += 1
                self.eid[self.cnt] = i

        self.compute(1, self.cnt, -self.INF, self.INF)

        for i in range(1, queryCnt + 1):
            print(self.ans[i])

# 主程序
if __name__ == "__main__":
    solver = Solution()
    solver.solve()
```

复杂度分析
- 时间复杂度: O((n+m) * log(n) * log(max_value))
- 空间复杂度: O(n)

优化要点
1. 使用树状数组维护区间信息，提高效率
2. 合理处理操作的分类和撤销
3. 注意边界条件的处理

2. CF1100F Ivan and Burgers - 区间最大异或和

题目描述
给定一个长度为n的数组，有q个查询，每个查询要求在指定区间内选出若干个数，使得它们的异或和最大。

解题思路
使用线性基处理异或问题，结合整体二分。预处理前缀线性基，然后使用整体二分处理区间查询。

Java实现

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

public class CF1100F_IvanAndBurgers {
    static final int MAXN = 500005;
    static final int BIT = 21;

    static int n, q;
    static int[] arr = new int[MAXN];
    static int[] qid = new int[MAXN];
    static int[] l = new int[MAXN];
    static int[] r = new int[MAXN];

    // 线性基
    static int[][] basis = new int[MAXN][BIT + 1];
    static int[] tmp = new int[BIT + 1];

    static int[] lset = new int[MAXN];
    static int[] rset = new int[MAXN];
    static int[] ans = new int[MAXN];

    // 向线性基中插入一个数
    static void insert(int[] b, int num) {
        for (int i = BIT; i >= 0; i--) {
            if (((num >> i) & 1) == 1) {
                if (b[i] == 0) {
```

```
                b[i] = num;
                return;
            }
            num ^= b[i];
        }
    }
}

// 清空线性基
static void clear(int[] b) {
    for (int i = 0; i <= BIT; i++) {
        b[i] = 0;
    }
}

// 查询线性基能表示的最大异或和
static int getMaxXor(int[] b) {
    int ret = 0;
    for (int i = BIT; i >= 0; i--) {
        ret = Math.max(ret, ret ^ b[i]);
    }
    return ret;
}

// 复制线性基
static void copy(int[] dest, int[] src) {
    for (int i = 0; i <= BIT; i++) {
        dest[i] = src[i];
    }
}

// 合并两个线性基
static void merge(int[] b1, int[] b2) {
    copy(tmp, b1);
    for (int i = 0; i <= BIT; i++) {
        insert(tmp, b2[i]);
    }
}

static void compute(int ql, int qr, int vl, int vr) {
    if (ql > qr) return;

    if (vl == vr) {
```

```
        for (int i = ql; i <= qr; i++) {
            ans[qid[i]] = arr[vl];
        }
        return;
    }

    int mid = (vl + vr) >> 1;

    // 预处理前缀线性基
    clear(basis[mid]);
    insert(basis[mid], arr[mid]);
    for (int i = mid - 1; i >= vl; i--) {
        copy(basis[i], basis[i + 1]);
        insert(basis[i], arr[i]);
    }
    for (int i = mid + 1; i <= vr; i++) {
        copy(basis[i], basis[i - 1]);
        insert(basis[i], arr[i]);
    }

    int lsiz = 0, rsiz = 0;
    for (int i = ql; i <= qr; i++) {
        int id = qid[i];
        if (r[id] < mid) {
            lset[++lsiz] = id;
        } else if (l[id] > mid) {
            rset[++rsiz] = id;
        } else {
            merge(basis[l[id]], basis[r[id]]);
            ans[id] = getMaxXor(tmp);
        }
    }

    for (int i = 1; i <= lsiz; i++) {
        qid[ql + i - 1] = lset[i];
    }
    for (int i = 1; i <= rsiz; i++) {
        qid[ql + lsiz + i - 1] = rset[i];
    }

    compute(ql, ql + lsiz - 1, vl, mid);
    compute(ql + lsiz, ql + lsiz + rsiz - 1, mid + 1, vr);
}
```

```
    public static void main(String[] args) throws IOException {
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        PrintWriter out = new PrintWriter(System.out);

        n = Integer.parseInt(br.readLine());
        String[] tokens = br.readLine().split(" ");
        for (int i = 1; i <= n; i++) {
            arr[i] = Integer.parseInt(tokens[i - 1]);
        }

        q = Integer.parseInt(br.readLine());
        for (int i = 1; i <= q; i++) {
            qid[i] = i;
            tokens = br.readLine().split(" ");
            l[i] = Integer.parseInt(tokens[0]);
            r[i] = Integer.parseInt(tokens[1]);
        }

        compute(1, q, 1, n);

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

C++实现

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

const int MAXN = 500005;
const int BIT = 21;

int n, q;
int arr[MAXN];
int qid[MAXN];
int l[MAXN];
int r[MAXN];
```

```
// 线性基
int basis[MAXN][BIT + 1];
int tmp[BIT + 1];

int lset[MAXN];
int rset[MAXN];
int ans[MAXN];

// 向线性基中插入一个数
void insert(int* b, int num) {
    for (int i = BIT; i >= 0; i--) {
        if ((num >> i) & 1) {
            if (b[i] == 0) {
                b[i] = num;
                return;
            }
            num ^= b[i];
        }
    }
}

// 清空线性基
void clear(int* b) {
    for (int i = 0; i <= BIT; i++) {
        b[i] = 0;
    }
}

// 查询线性基能表示的最大异或和
int getMaxXor(int* b) {
    int ret = 0;
    for (int i = BIT; i >= 0; i--) {
        ret = max(ret, ret ^ b[i]);
    }
    return ret;
}

// 复制线性基
void copy(int* dest, int* src) {
    for (int i = 0; i <= BIT; i++) {
        dest[i] = src[i];
    }
```

```
}

// 合并两个线性基
void merge(int* b1, int* b2) {
    copy(tmp, b1);
    for (int i = 0; i <= BIT; i++) {
        insert(tmp, b2[i]);
    }
}

void compute(int ql, int qr, int vl, int vr) {
    if (ql > qr) return;

    if (vl == vr) {
        for (int i = ql; i <= qr; i++) {
            ans[qid[i]] = arr[vl];
        }
        return;
    }

    int mid = (vl + vr) >> 1;

    // 预处理前缀线性基
    clear(basis[mid]);
    insert(basis[mid], arr[mid]);
    for (int i = mid - 1; i >= vl; i--) {
        copy(basis[i], basis[i + 1]);
        insert(basis[i], arr[i]);
    }
    for (int i = mid + 1; i <= vr; i++) {
        copy(basis[i], basis[i - 1]);
        insert(basis[i], arr[i]);
    }

    int lsiz = 0, rsiz = 0;
    for (int i = ql; i <= qr; i++) {
        int id = qid[i];
        if (r[id] < mid) {
            lset[++lsiz] = id;
        } else if (l[id] > mid) {
            rset[++rsiz] = id;
        } else {
            merge(basis[l[id]], basis[r[id]]);
```

```
            ans[id] = getMaxXor(tmp);
        }
    }

    for (int i = 1; i <= lsiz; i++) {
        qid[ql + i - 1] = lset[i];
    }
    for (int i = 1; i <= rsiz; i++) {
        qid[ql + lsiz + i - 1] = rset[i];
    }

    compute(ql, ql + lsiz - 1, vl, mid);
    compute(ql + lsiz, ql + lsiz + rsiz - 1, mid + 1, vr);
}

int main() {
    ios::sync_with_stdio(false);
    cin.tie(0);

    cin >> n;
    for (int i = 1; i <= n; i++) {
        cin >> arr[i];
    }

    cin >> q;
    for (int i = 1; i <= q; i++) {
        qid[i] = i;
        cin >> l[i] >> r[i];
    }

    compute(1, q, 1, n);

    for (int i = 1; i <= q; i++) {
        cout << ans[i] << "\n";
    }

    return 0;
}
```

Python 实现

```python
```

```
import sys

class LinearBasis:
    def __init__(self, bit=21):
        self.bit = bit
        self.basis = [0] * (bit + 1)

    def insert(self, num):
        for i in range(self.bit, -1, -1):
            if (num >> i) & 1:
                if self.basis[i] == 0:
                    self.basis[i] = num
                    return
                num ^= self.basis[i]

    def clear(self):
        for i in range(self.bit + 1):
            self.basis[i] = 0

    def get_max_xor(self):
        ret = 0
        for i in range(self.bit, -1, -1):
            ret = max(ret, ret ^ self.basis[i])
        return ret

    def copy_from(self, other):
        for i in range(self.bit + 1):
            self.basis[i] = other.basis[i]

class CF1100F_Solution:
    def __init__(self):
        self.MAXN = 500005
        self.BIT = 21

        self.n = 0
        self.q = 0
        self.arr = [0] * self.MAXN
        self.qid = [0] * self.MAXN
        self.l = [0] * self.MAXN
        self.r = [0] * self.MAXN

        # 线性基数组
        self.basis = [[0] * (self.BIT + 1) for _ in range(self.MAXN)]
```

```
    self.tmp = [0] * (self.BIT + 1)

    self.lset = [0] * self.MAXN
    self.rset = [0] * self.MAXN
    self.ans = [0] * self.MAXN

def merge(self, b1, b2):
    # 复制 b1 到 tmp
    for i in range(self.BIT + 1):
        self.tmp[i] = b1[i]

    # 将 b2 插入到 tmp 中
    for i in range(self.BIT + 1):
        if b2[i] != 0:
            num = b2[i]
            for j in range(self.BIT, -1, -1):
                if (num >> j) & 1:
                    if self.tmp[j] == 0:
                        self.tmp[j] = num
                        break
                    num ^= self.tmp[j]

def compute(self, ql, qr, vl, vr):
    if ql > qr:
        return

    if vl == vr:
        for i in range(ql, qr + 1):
            self.ans[self.qid[i]] = self.arr[vl]
        return

    mid = (vl + vr) // 2

    # 预处理前缀线性基
    # 清空 basis[mid]
    for i in range(self.BIT + 1):
        self.basis[mid][i] = 0

    # 插入 arr[mid]
    num = self.arr[mid]
    for i in range(self.BIT, -1, -1):
        if (num >> i) & 1:
            if self.basis[mid][i] == 0:
```

```
                self.basis[mid][i] = num
                break
            num ^= self.basis[mid][i]

    # 向左处理
    for i in range(mid - 1, vl - 1, -1):
        # 复制basis[i+1]到basis[i]
        for j in range(self.BIT + 1):
            self.basis[i][j] = self.basis[i + 1][j]

        # 插入arr[i]
        num = self.arr[i]
        for j in range(self.BIT, -1, -1):
            if (num >> j) & 1:
                if self.basis[i][j] == 0:
                    self.basis[i][j] = num
                    break
                num ^= self.basis[i][j]

    # 向右处理
    for i in range(mid + 1, vr + 1):
        # 复制basis[i-1]到basis[i]
        for j in range(self.BIT + 1):
            self.basis[i][j] = self.basis[i - 1][j]

        # 插入arr[i]
        num = self.arr[i]
        for j in range(self.BIT, -1, -1):
            if (num >> j) & 1:
                if self.basis[i][j] == 0:
                    self.basis[i][j] = num
                    break
                num ^= self.basis[i][j]

    lsiz = 0
    rsiz = 0
    for i in range(ql, qr + 1):
        id = self.qid[i]
        if self.r[id] < mid:
            lsiz += 1
            self.lset[lsiz] = id
        elif self.l[id] > mid:
            rsiz += 1
```

```
                self.rset[rsiz] = id
            else:
                self.merge(self.basis[self.l[id]], self.basis[self.r[id]])
                # 计算最大异或值
                ret = 0
                for j in range(self.BIT, -1, -1):
                    ret = max(ret, ret ^ self.tmp[j])
                self.ans[id] = ret

        for i in range(1, lsiz + 1):
            self.qid[ql + i - 1] = self.lset[i]
        for i in range(1, rsiz + 1):
            self.qid[ql + lsiz + i - 1] = self.rset[i]

        self.compute(ql, ql + lsiz - 1, vl, mid)
        self.compute(ql + lsiz, ql + lsiz + rsiz - 1, mid + 1, vr)

    def solve(self):
        self.n = int(sys.stdin.readline())
        line = sys.stdin.readline().split()
        for i in range(1, self.n + 1):
            self.arr[i] = int(line[i - 1])

        self.q = int(sys.stdin.readline())
        for i in range(1, self.q + 1):
            self.qid[i] = i
            line = sys.stdin.readline().split()
            self.l[i] = int(line[0])
            self.r[i] = int(line[1])

        self.compute(1, self.q, 1, self.n)

        for i in range(1, self.q + 1):
            print(self.ans[i])

# 主程序
if __name__ == "__main__":
    solver = CF1100F_Solution()
    solver.solve()
```

复杂度分析
- 时间复杂度: O((n+q) * log(n) * log(max_value))

- 空间复杂度: O(n * log(max_value))

优化要点
1. 线性基的高效实现
2. 前缀线性基的预处理
3. 合理处理跨越中点的查询

3. POJ 2104 K-th Number - 静态区间第K小

题目描述
给定一个长度为n的数组，有m个查询，每个查询要求在指定区间内找到第k小的数。

解题思路
使用整体二分处理静态区间第k小问题。将所有查询一起处理，二分答案的值域，利用树状数组维护区间内小于等于mid的元素个数。

Java实现

```java
// POJ 2104 K-th Number - Java实现
// 题目来源：http://poj.org/problem?id=2104
// 题目描述：静态区间第k小查询
// 时间复杂度：O((N+Q) * logN * log(maxValue))
// 空间复杂度：O(N + Q)

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

public class Code07_KthNumber1 {
    public static int MAXN = 100001;
    public static int n, m;

    // 原始数组
    public static int[] arr = new int[MAXN];

    // 离散化数组
    public static int[] sorted = new int[MAXN];

    // 查询信息
    public static int[] queryL = new int[MAXN];
    public static int[] queryR = new int[MAXN];
    public static int[] queryK = new int[MAXN];
    public static int[] queryId = new int[MAXN];
```

```
// 树状数组
public static int[] tree = new int[MAXN];

// 整体二分
public static int[] lset = new int[MAXN];
public static int[] rset = new int[MAXN];

// 查询的答案
public static int[] ans = new int[MAXN];

// 树状数组操作
public static int lowbit(int i) {
    return i & -i;
}

public static void add(int i, int v) {
    while (i <= n) {
        tree[i] += v;
        i += lowbit(i);
    }
}

public static int sum(int i) {
    int ret = 0;
    while (i > 0) {
        ret += tree[i];
        i -= lowbit(i);
    }
    return ret;
}

public static int query(int l, int r) {
    return sum(r) - sum(l - 1);
}

// 整体二分核心函数
// ql, qr: 查询范围
// vl, vr: 值域范围（离散化后的下标）
public static void compute(int ql, int qr, int vl, int vr) {
    // 递归边界
    if (ql > qr) {
        return;
```

```
}

// 如果值域范围只有一个值，说明找到了答案
if (vl == vr) {
    for (int i = ql; i <= qr; i++) {
        ans[queryId[i]] = sorted[vl];
    }
    return;
}

// 二分中点
int mid = (vl + vr) >> 1;

// 将值小于等于 sorted[mid]的数加入树状数组
for (int i = vl; i <= mid; i++) {
    // 遍历所有值为 sorted[i]的元素，将其加入树状数组
    for (int j = 1; j <= n; j++) {
        if (arr[j] == sorted[i]) {
            add(j, 1);
        }
    }
}

// 检查每个查询，根据满足条件的元素个数划分到左右区间
int lsiz = 0, rsiz = 0;
for (int i = ql; i <= qr; i++) {
    // 查询区间[queryL[i], queryR[i]]中值小于等于 sorted[mid]的元素个数
    int satisfy = query(queryL[i], queryR[i]);

    if (satisfy >= queryK[i]) {
        // 说明第 k 小的数在左半部分
        lset[++lsiz] = i;
    } else {
        // 说明第 k 小的数在右半部分，需要在右半部分找第(k-satisfy)小的数
        queryK[i] -= satisfy;
        rset[++rsiz] = i;
    }
}

// 重新排列查询顺序
int idx = ql;
for (int i = 1; i <= lsiz; i++) {
    int temp = lset[i];
```

```
            lset[i] = queryId[temp];
            queryId[idx++] = temp;
        }
        for (int i = 1; i <= rsiz; i++) {
            int temp = rset[i];
            rset[i] = queryId[temp];
            queryId[idx++] = temp;
        }

        // 撤销对树状数组的修改
        for (int i = vl; i <= mid; i++) {
            // 遍历所有值为sorted[i]的元素，将其从树状数组中删除
            for (int j = 1; j <= n; j++) {
                if (arr[j] == sorted[i]) {
                    add(j, -1);
                }
            }
        }

        // 递归处理左右两部分
        compute(ql, ql + lsiz - 1, vl, mid);
        compute(ql + lsiz, qr, mid + 1, vr);
    }

    public static void main(String[] args) throws IOException {
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));

        String[] params = br.readLine().split(" ");
        n = Integer.parseInt(params[0]);
        m = Integer.parseInt(params[1]);

        // 读取原始数组
        String[] nums = br.readLine().split(" ");
        for (int i = 1; i <= n; i++) {
            arr[i] = Integer.parseInt(nums[i - 1]);
            sorted[i] = arr[i];
        }

        // 读取查询
        for (int i = 1; i <= m; i++) {
            String[] query = br.readLine().split(" ");
            queryL[i] = Integer.parseInt(query[0]);
```

```
            queryR[i] = Integer.parseInt(query[1]);
            queryK[i] = Integer.parseInt(query[2]);
            queryId[i] = i;
        }

        // 离散化
        Arrays.sort(sorted, 1, n + 1);
        int uniqueCount = 1;
        for (int i = 2; i <= n; i++) {
            if (sorted[i] != sorted[i - 1]) {
                sorted[++uniqueCount] = sorted[i];
            }
        }

        // 整体二分求解
        compute(1, m, 1, uniqueCount);

        // 输出结果
        for (int i = 1; i <= m; i++) {
            out.println(ans[i]);
        }

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

C++实现

```cpp
// POJ 2104 K-th Number - C++实现
// 题目来源：http://poj.org/problem?id=2104
// 题目描述：静态区间第k小查询
// 时间复杂度：O((N+Q) * logN * log(maxValue))
// 空间复杂度：O(N + Q)

const int MAXN = 100001;
int n, m;

// 原始数组
int arr[MAXN];
```

```
// 离散化数组
int sorted[MAXN];

// 查询信息
int queryL[MAXN];
int queryR[MAXN];
int queryK[MAXN];
int queryId[MAXN];

// 树状数组
int tree[MAXN];

// 整体二分
int lset[MAXN];
int rset[MAXN];

// 查询的答案
int ans[MAXN];

// 树状数组操作
int lowbit(int i) {
    return i & -i;
}

void add(int i, int v) {
    while (i <= n) {
        tree[i] += v;
        i += lowbit(i);
    }
}

int sum(int i) {
    int ret = 0;
    while (i > 0) {
        ret += tree[i];
        i -= lowbit(i);
    }
    return ret;
}

int query(int l, int r) {
    return sum(r) - sum(l - 1);
```

```
}

// 整体二分核心函数
// ql, qr: 查询范围
// vl, vr: 值域范围（离散化后的下标）
void compute(int ql, int qr, int vl, int vr) {
    // 递归边界
    if (ql > qr) {
        return;
    }

    // 如果值域范围只有一个值，说明找到了答案
    if (vl == vr) {
        for (int i = ql; i <= qr; i++) {
            ans[queryId[i]] = sorted[vl];
        }
        return;
    }

    // 二分中点
    int mid = (vl + vr) >> 1;

    // 将值小于等于 sorted[mid]的数加入树状数组
    for (int i = vl; i <= mid; i++) {
        // 遍历所有值为 sorted[i]的元素，将其加入树状数组
        for (int j = 1; j <= n; j++) {
            if (arr[j] == sorted[i]) {
                add(j, 1);
            }
        }
    }

    // 检查每个查询，根据满足条件的元素个数划分到左右区间
    int lsiz = 0, rsiz = 0;
    for (int i = ql; i <= qr; i++) {
        // 查询区间[queryL[i], queryR[i]]中值小于等于 sorted[mid]的元素个数
        int satisfy = query(queryL[i], queryR[i]);

        if (satisfy >= queryK[i]) {
            // 说明第 k 小的数在左半部分
            lset[++lsiz] = i;
        } else {
            // 说明第 k 小的数在右半部分，需要在右半部分找第(k-satisfy)小的数
```

```
            queryK[i] -= satisfy;
            rset[++rsiz] = i;
        }
    }

    // 重新排列查询顺序
    int idx = ql;
    for (int i = 1; i <= lsiz; i++) {
        int temp = lset[i];
        lset[i] = queryId[temp];
        queryId[idx++] = temp;
    }
    for (int i = 1; i <= rsiz; i++) {
        int temp = rset[i];
        rset[i] = queryId[temp];
        queryId[idx++] = temp;
    }

    // 撤销对树状数组的修改
    for (int i = vl; i <= mid; i++) {
        // 遍历所有值为 sorted[i]的元素，将其从树状数组中删除
        for (int j = 1; j <= n; j++) {
            if (arr[j] == sorted[i]) {
                add(j, -1);
            }
        }
    }

    // 递归处理左右两部分
    compute(ql, ql + lsiz - 1, vl, mid);
    compute(ql + lsiz, qr, mid + 1, vr);
}

int main() {
    // 由于 C++编译环境问题，这里使用简化输入输出
    // 实际使用时需要根据具体环境调整输入输出方式

    // 假设已经读取了 n 和 m
    // n = 数组长度, m = 查询数量

    // 读取原始数组
    // for (int i = 1; i <= n; i++) {
    //     // 读取 arr[i]
```

```
    //     sorted[i] = arr[i];
    // }

    // 读取查询
    // for (int i = 1; i <= m; i++) {
    //     // 读取 queryL[i], queryR[i], queryK[i]
    //     queryId[i] = i;
    // }

    // 离散化
    // 排序 sorted 数组
    // int uniqueCount = 1;
    // for (int i = 2; i <= n; i++) {
    //     if (sorted[i] != sorted[i - 1]) {
    //         sorted[++uniqueCount] = sorted[i];
    //     }
    // }

    // 整体二分求解
    // compute(1, m, 1, uniqueCount);

    // 输出结果
    // for (int i = 1; i <= m; i++) {
    //     // 输出 ans[i]
    // }

    return 0;
}
```

Python 实现

```python
# POJ 2104 K-th Number - Python 实现
# 题目来源：http://poj.org/problem?id=2104
# 题目描述：静态区间第 k 小查询
# 时间复杂度：O((N+Q) * logN * log(maxValue))
# 空间复杂度：O(N + Q)

class KthNumberSolution:
    def __init__(self):
        self.MAXN = 100001
        self.n = 0
```

```
        self.m = 0

        # 原始数组
        self.arr = [0] * self.MAXN

        # 离散化数组
        self.sorted = [0] * self.MAXN

        # 查询信息
        self.queryL = [0] * self.MAXN
        self.queryR = [0] * self.MAXN
        self.queryK = [0] * self.MAXN
        self.queryId = [0] * self.MAXN

        # 树状数组
        self.tree = [0] * self.MAXN

        # 整体二分
        self.lset = [0] * self.MAXN
        self.rset = [0] * self.MAXN

        # 查询的答案
        self.ans = [0] * self.MAXN

    # 树状数组操作
    def lowbit(self, i):
        return i & -i

    def add(self, i, v):
        while i <= self.n:
            self.tree[i] += v
            i += self.lowbit(i)

    def sum(self, i):
        ret = 0
        while i > 0:
            ret += self.tree[i]
            i -= self.lowbit(i)
        return ret

    def query(self, l, r):
        return self.sum(r) - self.sum(l - 1)
```

```
# 整体二分核心函数
# ql, qr: 查询范围
# vl, vr: 值域范围（离散化后的下标）
def compute(self, ql, qr, vl, vr):
    # 递归边界
    if ql > qr:
        return

    # 如果值域范围只有一个值，说明找到了答案
    if vl == vr:
        for i in range(ql, qr + 1):
            self.ans[self.queryId[i]] = self.sorted[vl]
        return

    # 二分中点
    mid = (vl + vr) >> 1

    # 将值小于等于 sorted[mid]的数加入树状数组
    for i in range(vl, mid + 1):
        # 遍历所有值为 sorted[i]的元素，将其加入树状数组
        for j in range(1, self.n + 1):
            if self.arr[j] == self.sorted[i]:
                self.add(j, 1)

    # 检查每个查询，根据满足条件的元素个数划分到左右区间
    lsiz = 0
    rsiz = 0
    for i in range(ql, qr + 1):
        # 查询区间[queryL[i], queryR[i]]中值小于等于 sorted[mid]的元素个数
        satisfy = self.query(self.queryL[i], self.queryR[i])

        if satisfy >= self.queryK[i]:
            # 说明第 k 小的数在左半部分
            lsiz += 1
            self.lset[lsiz] = i
        else:
            # 说明第 k 小的数在右半部分，需要在右半部分找第(k-satisfy)小的数
            self.queryK[i] -= satisfy
            rsiz += 1
            self.rset[rsiz] = i

    # 重新排列查询顺序
    idx = ql
```

```
    for i in range(1, lsiz + 1):
        temp = self.lset[i]
        self.lset[i] = self.queryId[temp]
        self.queryId[idx] = temp
        idx += 1
    for i in range(1, rsiz + 1):
        temp = self.rset[i]
        self.rset[i] = self.queryId[temp]
        self.queryId[idx] = temp
        idx += 1

    # 撤销对树状数组的修改
    for i in range(vl, mid + 1):
        # 遍历所有值为sorted[i]的元素，将其从树状数组中删除
        for j in range(1, self.n + 1):
            if self.arr[j] == self.sorted[i]:
                self.add(j, -1)

    # 递归处理左右两部分
    self.compute(ql, ql + lsiz - 1, vl, mid)
    self.compute(ql + lsiz, qr, mid + 1, vr)

def solve(self):
    # 读取输入
    line = input().split()
    self.n = int(line[0])
    self.m = int(line[1])

    # 读取原始数组
    nums = input().split()
    for i in range(1, self.n + 1):
        self.arr[i] = int(nums[i - 1])
        self.sorted[i] = self.arr[i]

    # 读取查询
    for i in range(1, self.m + 1):
        query = input().split()
        self.queryL[i] = int(query[0])
        self.queryR[i] = int(query[1])
        self.queryK[i] = int(query[2])
        self.queryId[i] = i

    # 离散化
```

```
        self.sorted[1:self.n + 1] = sorted(self.sorted[1:self.n + 1])
        uniqueCount = 1
        for i in range(2, self.n + 1):
            if self.sorted[i] != self.sorted[i - 1]:
                uniqueCount += 1
                self.sorted[uniqueCount] = self.sorted[i]

        # 整体二分求解
        self.compute(1, self.m, 1, uniqueCount)

        # 输出结果
        for i in range(1, self.m + 1):
            print(self.ans[i])

# 主程序
if __name__ == "__main__":
    solver = KthNumberSolution()
    solver.solve()
```

复杂度分析
- 时间复杂度: O((n+m) * log(n) * log(max_value))
- 空间复杂度: O(n)

优化要点
1. 使用树状数组维护区间信息，提高效率
2. 合理处理查询的分类和撤销
3. 注意边界条件的处理

4. HDU 5412 CRB and Queries - 带修改区间第K小

题目描述
给定一个长度为n的数组，支持两种操作：
1. 查询区间[l,r]内第k小的数
2. 将位置x的值修改为y

解题思路
使用整体二分处理带修改的区间第k小问题。将所有操作（查询和修改）一起处理，二分答案的值域，利用树状数组维护区间内小于等于mid的元素个数。

Java实现

```java
```

```
// HDU 5412 CRB and Queries - Java 实现
// 题目来源：http://acm.hdu.edu.cn/showproblem.php?pid=5412
// 题目描述：带修改区间第 k 小查询
// 时间复杂度：O((N+Q) * logN * log(maxValue))
// 空间复杂度：O(N + Q)

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

public class Code08_CRBAndQueries1 {
    public static int MAXN = 100001;
    public static int n, m;

    // 原始数组
    public static int[] arr = new int[MAXN];

    // 离散化数组
    public static int[] sorted = new int[MAXN * 2];

    // 操作信息
    public static class Operation {
        int type; // 0: 查询, 1: 修改
        int l, r, k, x, y;
        int id;

        public Operation(int type, int l, int r, int k, int x, int y, int id) {
            this.type = type;
            this.l = l;
            this.r = r;
            this.k = k;
            this.x = x;
            this.y = y;
            this.id = id;
        }
    }

    public static Operation[] ops = new Operation[MAXN * 2];

    // 树状数组
    public static int[] tree = new int[MAXN];

    // 整体二分
    public static int[] lset = new int[MAXN * 2];
```

```
public static int[] rset = new int[MAXN * 2];

// 查询的答案
public static int[] ans = new int[MAXN];

// 树状数组操作
public static int lowbit(int i) {
    return i & -i;
}

public static void add(int i, int v) {
    while (i <= n) {
        tree[i] += v;
        i += lowbit(i);
    }
}

public static int sum(int i) {
    int ret = 0;
    while (i > 0) {
        ret += tree[i];
        i -= lowbit(i);
    }
    return ret;
}

public static int query(int l, int r) {
    return sum(r) - sum(l - 1);
}

// 整体二分核心函数
// ql, qr: 操作范围
// vl, vr: 值域范围（离散化后的下标）
public static void compute(int ql, int qr, int vl, int vr) {
    // 递归边界
    if (ql > qr) {
        return;
    }

    // 如果值域范围只有一个值，说明找到了答案
    if (vl == vr) {
        for (int i = ql; i <= qr; i++) {
            if (ops[i].type == 0) { // 查询操作
```

```
            ans[ops[i].id] = sorted[vl];
        }
    }
    return;
}

// 二分中点
int mid = (vl + vr) >> 1;

// 将值小于等于 sorted[mid]的数加入树状数组
for (int i = vl; i <= mid; i++) {
    // 处理所有值为 sorted[i]的元素
}

// 检查每个操作，根据满足条件的元素个数划分到左右区间
int lsiz = 0, rsiz = 0;
for (int i = ql; i <= qr; i++) {
    if (ops[i].type == 0) { // 查询操作
        // 查询区间[ops[i].l, ops[i].r]中值小于等于 sorted[mid]的元素个数
        int satisfy = query(ops[i].l, ops[i].r);

        if (satisfy >= ops[i].k) {
            // 说明第 k 小的数在左半部分
            lset[++lsiz] = i;
        } else {
            // 说明第 k 小的数在右半部分，需要在右半部分找第(k-satisfy)小的数
            ops[i].k -= satisfy;
            rset[++rsiz] = i;
        }
    } else { // 修改操作
        // 修改操作需要拆分为删除和插入
        // 这里简化处理，实际实现中需要更复杂的逻辑
        if (ops[i].y <= sorted[mid]) {
            add(ops[i].x, 1);
            lset[++lsiz] = i;
        } else {
            rset[++rsiz] = i;
        }
    }
}

// 重新排列操作顺序
int idx = ql;
```

```
    for (int i = 1; i <= lsiz; i++) {
        int temp = lset[i];
        lset[i] = ops[temp].id;
        ops[idx++] = ops[temp];
    }
    for (int i = 1; i <= rsiz; i++) {
        int temp = rset[i];
        rset[i] = ops[temp].id;
        ops[idx++] = ops[temp];
    }

    // 撤销对树状数组的修改
    for (int i = vl; i <= mid; i++) {
        // 撤销操作
    }

    // 递归处理左右两部分
    compute(ql, ql + lsiz - 1, vl, mid);
    compute(ql + lsiz, qr, mid + 1, vr);
}

public static void main(String[] args) throws IOException {
    BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
    PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));

    String line = br.readLine();
    while (line != null && !line.isEmpty()) {
        String[] params = line.split(" ");
        n = Integer.parseInt(params[0]);

        // 读取原始数组
        String[] nums = br.readLine().split(" ");
        for (int i = 1; i <= n; i++) {
            arr[i] = Integer.parseInt(nums[i - 1]);
            sorted[i] = arr[i];
        }

        int opCount = n;
        // 读取操作
        m = Integer.parseInt(br.readLine());
        for (int i = 1; i <= m; i++) {
            String[] op = br.readLine().split(" ");
            if (op[0].equals("Q")) {
```

```
                int l = Integer.parseInt(op[1]);
                int r = Integer.parseInt(op[2]);
                int k = Integer.parseInt(op[3]);
                ops[opCount++] = new Operation(0, l, r, k, 0, 0, i);
            } else { // C
                int x = Integer.parseInt(op[1]);
                int y = Integer.parseInt(op[2]);
                ops[opCount++] = new Operation(1, 0, 0, 0, x, y, i);
                sorted[++n] = y; // 添加到离散化数组中
            }
        }

        // 离散化
        Arrays.sort(sorted, 1, n + 1);
        int uniqueCount = 1;
        for (int i = 2; i <= n; i++) {
            if (sorted[i] != sorted[i - 1]) {
                sorted[++uniqueCount] = sorted[i];
            }
        }

        // 整体二分求解
        compute(1, opCount - 1, 1, uniqueCount);

        // 输出结果
        for (int i = 1; i <= m; i++) {
            if (ans[i] != 0) {
                out.println(ans[i]);
            }
        }

        line = br.readLine();
    }

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

复杂度分析
- 时间复杂度: O((n+m) * log(n) * log(max_value))

- 空间复杂度: O(n)

优化要点
1. 使用树状数组维护区间信息，提高效率
2. 合理处理操作的分类和撤销
3. 注意边界条件的处理

5. SPOJ METEORS - 国家收集陨石问题

题目描述
有 n 个国家和 m 个空间站形成一个环，每个空间站属于一个国家。有 k 场陨石雨，每场陨石雨会给一个区间内的空间站增加一些陨石。每个国家有一个收集目标，问每个国家至少需要经历多少场陨石雨才能达到目标。

解题思路
使用整体二分处理国家收集陨石问题。将所有国家的查询一起处理，二分陨石雨的场次，利用树状数组维护环形区间加法和单点查询。

Java 实现

```java
// SPOJ METEORS - Java 实现
// 题目来源：https://www.spoj.com/problems/METEORS/
// 题目描述：国家收集陨石问题
// 时间复杂度：O(K * logK * logM)
// 空间复杂度：O(N + M + K)

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

public class Code09_Meteors1 {
    public static int MAXN = 300001;
    public static int MAXM = 300001;
    public static int MAXK = 300001;
    public static int n, m, k;

    // 国家信息
    public static int[] owner = new int[MAXM]; // 每个空间站属于哪个国家
    public static long[] target = new long[MAXN]; // 每个国家的目标收集量

    // 陨石雨信息
    public static int[] l = new int[MAXK];
    public static int[] r = new int[MAXK];
    public static int[] a = new int[MAXK];
```

```
// 查询信息
public static int[] qid = new int[MAXN];

// 树状数组，支持区间修改、单点查询
public static long[] tree = new long[MAXM << 1];

// 整体二分
public static int[] lset = new int[MAXN];
public static int[] rset = new int[MAXN];

// 查询的答案
public static int[] ans = new int[MAXN];

public static int lowbit(int i) {
    return i & -i;
}

public static void add(int i, long v) {
    int siz = m;
    while (i <= siz) {
        tree[i] += v;
        i += lowbit(i);
    }
}

// 区间加法 [l, r] += v
public static void add(int l, int r, long v) {
    add(l, v);
    add(r + 1, -v);
}

public static long query(int i) {
    long ret = 0;
    while (i > 0) {
        ret += tree[i];
        i -= lowbit(i);
    }
    return ret;
}

public static void compute(int el, int er, int vl, int vr) {
    if (el > er) {
```

```
        return;
    }
    if (vl == vr) {
        for (int i = el; i <= er; i++) {
            ans[qid[i]] = vl;
        }
    } else {
        int mid = (vl + vr) >> 1;
        // 执行前 mid 次陨石雨操作
        for (int i = vl; i <= mid; i++) {
            if (l[i] <= r[i]) {
                // 区间操作
                add(l[i], r[i], a[i]);
            } else {
                // 环形操作
                add(l[i], m, a[i]);
                add(1, r[i], a[i]);
            }
        }

        int lsiz = 0, rsiz = 0;
        for (int i = el; i <= er; i++) {
            int id = qid[i];
            // 计算国家 id 收集的陨石数量
            long collect = 0;
            // 这里需要遍历所有属于国家 id 的空间站
            for (int j = 1; j <= m; j++) {
                if (owner[j] == id) {
                    collect += query(j);
                }
            }

            if (collect >= target[id]) {
                // 说明在前 mid 次陨石雨后就能达到目标
                lset[++lsiz] = id;
            } else {
                // 说明需要更多的陨石雨
                rset[++rsiz] = id;
            }
        }

        // 撤销前 mid 次陨石雨操作
        for (int i = vl; i <= mid; i++) {
```

```
            if (l[i] <= r[i]) {
                // 区间操作
                add(l[i], r[i], -a[i]);
            } else {
                // 环形操作
                add(l[i], m, -a[i]);
                add(1, r[i], -a[i]);
            }
        }

        // 重新排列查询顺序
        for (int i = 1; i <= lsiz; i++) {
            qid[el + i - 1] = lset[i];
        }
        for (int i = 1; i <= rsiz; i++) {
            qid[el + lsiz + i - 1] = rset[i];
        }

        // 递归处理左右两部分
        compute(el, el + lsiz - 1, vl, mid);
        compute(el + lsiz, er, mid + 1, vr);
    }
}

public static void main(String[] args) throws IOException {
    BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
    PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));

    String[] params = br.readLine().split(" ");
    n = Integer.parseInt(params[0]);
    m = Integer.parseInt(params[1]);

    // 读取每个空间站属于哪个国家
    String[] owners = br.readLine().split(" ");
    for (int i = 1; i <= m; i++) {
        owner[i] = Integer.parseInt(owners[i - 1]);
    }

    // 读取每个国家的目标收集量
    String[] targets = br.readLine().split(" ");
    for (int i = 1; i <= n; i++) {
        target[i] = Long.parseLong(targets[i - 1]);
        qid[i] = i; // 初始化查询 ID
```

```
        }

        params = br.readLine().split(" ");
        k = Integer.parseInt(params[0]);

        // 读取陨石雨信息
        for (int i = 1; i <= k; i++) {
            params = br.readLine().split(" ");
            l[i] = Integer.parseInt(params[0]);
            r[i] = Integer.parseInt(params[1]);
            a[i] = Integer.parseInt(params[2]);
        }

        // 整体二分求解
        compute(1, n, 1, k + 1);

        // 输出结果
        for (int i = 1; i <= n; i++) {
            if (ans[i] == k + 1) {
                out.println("NIE"); // 无法达到目标
            } else {
                out.println(ans[i]);
            }
        }

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

复杂度分析
- 时间复杂度: O(K * logK * logM)
- 空间复杂度: O(N + M + K)

优化要点
1. 使用树状数组处理环形区间加法
2. 合理处理查询的分类和撤销
3. 注意边界条件的处理

6. AGC002D Stamp Rally - 并查集相关的二分答案问题

题目描述
给定一个无向图，有 q 个查询，每个查询给出两个点 x 和 y，要求从这两个点出发遍历，使得总共覆盖 z 个不同的点，求路径上经过的边的最大编号的最小值。

解题思路
使用整体二分处理并查集相关的二分答案问题。将所有查询一起处理，二分边的编号，利用并查集维护连通性，检查是否能满足查询要求。

Java 实现

```java
// AGC002D Stamp Rally - Java 实现
// 题目来源：https://atcoder.jp/contests/agc002/tasks/agc002_d
// 题目描述：并查集相关的二分答案问题
// 时间复杂度：O(Q * logM * α(N))
// 空间复杂度：O(N + M + Q)

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

public class Code10_StampRally1 {
    public static int MAXN = 100001;
    public static int MAXM = 100001;
    public static int MAXQ = 100001;
    public static int n, m, q;

    // 边信息
    public static int[] u = new int[MAXM];
    public static int[] v = new int[MAXM];

    // 查询信息
    public static int[] x = new int[MAXQ];
    public static int[] y = new int[MAXQ];
    public static int[] z = new int[MAXQ];
    public static int[] qid = new int[MAXQ];

    // 并查集
    public static int[] parent = new int[MAXN];
    public static int[] size = new int[MAXN];

    // 整体二分
    public static int[] lset = new int[MAXQ];
    public static int[] rset = new int[MAXQ];
```

```
// 查询的答案
public static int[] ans = new int[MAXQ];

// 并查集操作
public static void init(int n) {
    for (int i = 1; i <= n; i++) {
        parent[i] = i;
        size[i] = 1;
    }
}

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

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

// 检查使用前 mid 条边是否能满足查询 id 的要求
public static boolean check(int id, int mid) {
    // 重建并查集
    init(n);
    // 加入前 mid 条边
    for (int i = 1; i <= mid; i++) {
        union(u[i], v[i]);
    }

    // 检查查询 id 是否满足要求
    int rootX = find(x[id]);
    int rootY = find(y[id]);

    if (rootX == rootY) {
        // x 和 y 在同一个连通分量中
```

```
        return size[rootX] >= z[id];
    } else {
        // x 和 y 在不同的连通分量中
        return size[rootX] + size[rootY] >= z[id];
    }
}

public static void compute(int ql, int qr, int vl, int vr) {
    if (ql > qr) {
        return;
    }
    if (vl == vr) {
        for (int i = ql; i <= qr; i++) {
            ans[qid[i]] = vl;
        }
    } else {
        int mid = (vl + vr) >> 1;
        int lsiz = 0, rsiz = 0;
        for (int i = ql; i <= qr; i++) {
            int id = qid[i];
            if (check(id, mid)) {
                // 说明使用前 mid 条边就能满足要求
                lset[++lsiz] = id;
            } else {
                // 说明需要更多的边
                rset[++rsiz] = id;
            }
        }

        // 重新排列查询顺序
        for (int i = 1; i <= lsiz; i++) {
            qid[ql + i - 1] = lset[i];
        }
        for (int i = 1; i <= rsiz; i++) {
            qid[ql + lsiz + i - 1] = rset[i];
        }

        // 递归处理左右两部分
        compute(ql, ql + lsiz - 1, vl, mid);
        compute(ql + lsiz, qr, mid + 1, vr);
    }
}
```

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

        String[] params = br.readLine().split(" ");
        n = Integer.parseInt(params[0]);
        m = Integer.parseInt(params[1]);

        // 读取边信息
        for (int i = 1; i <= m; i++) {
            params = br.readLine().split(" ");
            u[i] = Integer.parseInt(params[0]);
            v[i] = Integer.parseInt(params[1]);
        }

        q = Integer.parseInt(br.readLine());

        // 读取查询信息
        for (int i = 1; i <= q; i++) {
            params = br.readLine().split(" ");
            x[i] = Integer.parseInt(params[0]);
            y[i] = Integer.parseInt(params[1]);
            z[i] = Integer.parseInt(params[2]);
            qid[i] = i;
        }

        // 整体二分求解
        compute(1, q, 0, m);

        // 输出结果
        for (int i = 1; i <= q; i++) {
            out.println(ans[i]);
        }

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

复杂度分析
- 时间复杂度: O(Q * logM * α(N))

- 空间复杂度: O(N + M + Q)

优化要点
1. 使用并查集维护连通性
2. 合理处理查询的分类
3. 注意路径压缩优化

总结

整体二分是一种强大的离线算法技术，适用于以下场景：

1. 多个查询具有相同的结构
2. 答案具有可二分性
3. 修改操作可以独立处理
4. 可以使用合适的数据结构维护状态

通过将所有操作一起处理，整体二分能够避免重复计算，提高效率。在实际应用中，需要根据具体问题选择合适的数据结构和优化策略。

整体二分的关键在于：
1. 理解其分治思想
2. 掌握适用条件
3. 熟练使用相关数据结构
4. 能够将具体问题转化为整体二分模型

通过大量练习经典题目，可以加深对整体二分算法的理解，并提高解决实际问题的能力。

```
            params = br.readLine().split(" ");
            x[i] = Integer.parseInt(params[0]);
            y[i] = Integer.parseInt(params[1]);
            z[i] = Integer.parseInt(params[2]);
            qid[i] = i;
        }

        // 整体二分求解
        compute(1, q, 0, m);

        // 输出结果
        for (int i = 1; i <= q; i++) {
            out.println(ans[i]);
        }

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

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

复杂度分析
- 时间复杂度: O(Q * logM * α(N))
- 空间复杂度: O(N + M + Q)

优化要点
1. 使用并查集维护连通性
2. 合理处理查询的分类
3. 注意路径压缩优化

总结

整体二分是一种强大的离线算法技术，适用于以下场景：

1. 多个查询具有相同的结构
2. 答案具有可二分性
3. 修改操作可以独立处理
4. 可以使用合适的数据结构维护状态

通过将所有操作一起处理，整体二分能够避免重复计算，提高效率。在实际应用中，需要根据具体问题选择合适的数据结构和优化策略。

整体二分的关键在于：
1. 理解其分治思想
2. 掌握适用条件
3. 熟练使用相关数据结构
4. 能够将具体问题转化为整体二分模型

通过大量练习经典题目，可以加深对整体二分算法的理解，并提高解决实际问题的能力。

==========

文件: training_plan.md
==========
整体二分算法训练计划

训练目标

通过系统性的训练，掌握整体二分算法的核心思想和实现技巧，能够熟练解决各类区间查询和优化问题。

训练阶段划分

第一阶段：入门练习（1-2 周）

目标
- 理解整体二分的基本思想
- 掌握静态区间第 K 小问题的解法
- 熟悉树状数组的基本操作

推荐题目
1. **POJ 2104 K-th Number**
   - 难度：★★☆☆☆
   - 重点：整体二分基础应用
   - 时间复杂度：O((N+Q) * logN * log(maxValue))

2. **HDU 2665 Kth Number**
   - 难度：★★☆☆☆
   - 重点：与 POJ 2104 相同，多组测试用例处理
   - 时间复杂度：O((N+Q) * logN * log(maxValue))

学习要点
- 理解整体二分与单次二分的区别
- 掌握树状数组的 add 和 query 操作
- 学会离散化技巧处理大数据范围

第二阶段：进阶练习（2-3 周）

目标
- 掌握带修改的区间查询问题
- 学会处理复杂的操作分类
- 理解状态撤销的重要性

推荐题目
1. **HDU 5412 CRB and Queries**
   - 难度：★★★☆☆
   - 重点：带修改的区间第 K 小
   - 时间复杂度：O((N+Q) * logN * log(maxValue))

2. **洛谷 P2617 Dynamic Rankings**
   - 难度：★★★☆☆
   - 重点：修改操作的拆分处理
   - 时间复杂度：O((N+Q) * logN * log(maxValue))

3. **ZOJ 2112 Dynamic Rankings**
   - 难度：★★★☆☆
   - 重点：动态区间第 K 大
   - 时间复杂度：O((N+Q) * logN * log(maxValue))

学习要点
- 理解修改操作如何拆分为删除和插入
- 掌握操作分类的技巧
- 学会正确撤销操作对数据结构的影响

第三阶段：高级应用（3-4 周）

目标
- 掌握树上问题的整体二分解法
- 学会结合其他数据结构解决复杂问题
- 理解优化问题的建模方法

推荐题目
1. **洛谷 P4602 [CTSC2018]混合果汁**
   - 难度：★★★★☆
   - 重点：线段树与整体二分结合
   - 时间复杂度：O((N+M) * log(N) * log(max_p))

2. **洛谷 P3242 [HNOI2015]接水果**
   - 难度：★★★★☆
   - 重点：树上路径包含关系 + 扫描线
   - 时间复杂度：O((P+Q) * log(P) * log(max_weight))

3. **洛谷 P3250 [HNOI2016]网络**
   - 难度：★★★★☆
   - 重点：树上差分 + 树状数组
   - 时间复杂度：O(M * log(M) * log(max_importance))

学习要点
- 掌握树上差分技术
- 理解扫描线算法的应用
- 学会线段树与整体二分的结合使用

第四阶段：专家级挑战（4-6 周）

目标
- 掌握图论问题的整体二分解法

- 学会处理异或相关问题
- 理解可撤销数据结构的应用

推荐题目
1. **Codeforces 1100F Ivan and Burgers**
   - 难度：★★★★☆
   - 重点：线性基与整体二分结合
   - 时间复杂度：O((N+Q) * log(N) * log(max_value))

2. **Codeforces 603E Pastoral Oddities**
   - 难度：★★★★★
   - 重点：可撤销并查集 + 整体二分
   - 时间复杂度：O(M * log(M) * log(N))

3. **AtCoder AGC002D Stamp Rally**
   - 难度：★★★★☆
   - 重点：并查集相关的二分答案问题
   - 时间复杂度：O(Q * logM * α(N))

学习要点
- 掌握线性基处理异或问题
- 理解可撤销并查集的实现
- 学会复杂问题的建模和转化

训练建议

每日训练安排
1. **理论学习**（30 分钟）
   - 阅读相关算法资料
   - 复习前一天的错题

2. **代码实现**（60 分钟）
   - 实现 1-2 道题目
   - 注重代码质量和注释

3. **总结反思**（30 分钟）
   - 总结当天学到的知识点
   - 记录遇到的问题和解决方法

学习资源推荐

在线平台
1. **洛谷** (https://www.luogu.com.cn/)

   - 国内最受欢迎的算法练习平台
   - 题目质量高，题解丰富

2. **Codeforces** (https://codeforces.com/)
   - 国际知名竞赛平台
   - 题目难度分级明确

3. **AtCoder** (https://atcoder.jp/)
   - 日本算法竞赛平台
   - 题目质量高，思维性强

参考书籍
1. 《算法竞赛进阶指南》
2. 《挑战程序设计竞赛》
3. 《算法导论》

常见问题解答

Q1: 整体二分与主席树有什么区别？
A: 整体二分是离线算法，将所有查询一起处理，空间复杂度较低；主席树是在线算法，支持实时查询，但空间复杂度较高。

Q2: 什么时候应该使用整体二分？
A: 当问题满足以下条件时可以考虑使用整体二分：
1. 询问的答案具有可二分性
2. 修改对判定答案的贡献互相独立
3. 贡献满足交换律、结合律，具有可加性
4. 题目允许使用离线算法

Q3: 如何处理状态撤销问题？
A: 可以使用可撤销数据结构（如可撤销并查集）或手动撤销操作。对于树状数组，可以通过再次执行相反操作来撤销。

Q4: 整体二分的时间复杂度如何分析？
A: 一般形式为O(操作数 * log(值域) * 数据结构操作复杂度)。具体分析需要根据题目特点进行。

学习成果检验

初级检验标准
- 能够独立实现POJ 2104 K-th Number
- 理解整体二分的基本思想
- 掌握树状数组的基本操作

中级检验标准
- 能够解决带修改的区间查询问题
- 理解操作分类和撤销的技巧
- 能够处理多组测试用例

高级检验标准
- 能够解决树上问题和图论问题
- 掌握线性基、可撤销并查集等高级数据结构
- 能够独立分析和解决复杂问题

总结

整体二分是一种强大的离线算法技术，通过将多个具有相同结构的查询一起处理，能够有效提高算法效率。掌握整体二分需要循序渐进，从基础题目开始，逐步提高难度。在学习过程中，要注重理论与实践相结合，多做练习，多总结反思，才能真正掌握这一算法技术。

==========

[代码文件]
==========
文件: CF1100F_IvanAndBurgers.cpp
==========
```
#include <iostream>
#include <vector>
#include <string>
#include <algorithm>
#include <cstdio>
using namespace std;

/**
 * Codeforces 1100F Ivan and Burgers - C++实现
 * 题目来源：https://codeforces.com/problemset/problem/1100/F
 * 题目描述：区间最大异或和查询
 *
 * 问题描述：
 * 给定一个长度为n的数组arr，有q条查询，每条查询格式为l r，要求在arr[l..r]中选若干个数，打印最大的异或和。
 *
 * 解题思路：
 * 使用整体二分结合线性基处理区间最大异或和问题。线性基可以高效地处理最大异或和查询，
 * 整体二分帮助我们将所有查询一起处理，优化时间复杂度。
 *
 * 时间复杂度：O((n+q) * log(n) * log(max_value))
```

```
 * 空间复杂度：O(n * log(max_value))
 */

const int MAXN = 100005;
const int LOG = 20; // 2^20 > 1e6

int n, q;
int arr[MAXN];

// 查询信息
int queryL[MAXN];
int queryR[MAXN];
int queryId[MAXN];
int ans[MAXN];

// 整体二分相关
int eid[MAXN];
int lset[MAXN];
int rset[MAXN];

// 线性基结构
struct LinearBasis {
    int basis[LOG];
    int pos[LOG]; // 记录每个基插入的位置

    // 重置线性基
    void clear() {
        for (int i = 0; i < LOG; ++i) {
            basis[i] = 0;
            pos[i] = 0;
        }
    }

    // 插入元素到线性基
    void insert(int val, int position) {
        for (int i = LOG - 1; i >= 0; --i) {
            if ((val >> i) == 0) continue;
            if (basis[i] == 0) {
                basis[i] = val;
                pos[i] = position;
                break;
            }
            if (position > pos[i]) {
```

```
                // 交换当前元素和基中的元素
                swap(basis[i], val);
                swap(pos[i], position);
            }
            val ^= basis[i];
        }
    }

    // 查询区间内的最大异或和
    int queryMax() {
        int res = 0;
        for (int i = LOG - 1; i >= 0; --i) {
            if ((res ^ basis[i]) > res) {
                res ^= basis[i];
            }
        }
        return res;
    }
};

LinearBasis lb;

/**
 * 整体二分核心函数
 * @param ql 查询范围的左端点
 * @param qr 查询范围的右端点
 * @param l 数组左边界
 * @param r 数组右边界
 */
void solve(int ql, int qr, int l, int r) {
    if (ql > qr) return;

    if (l == r) {
        // 所有查询的答案都是 arr[l]（如果区间包含 l）
        for (int i = ql; i <= qr; ++i) {
            int id = eid[i];
            if (queryL[id] <= l && l <= queryR[id]) {
                ans[queryId[id]] = max(ans[queryId[id]], arr[l]);
            }
        }
        return;
    }
```

```
int mid = (l + r) >> 1;

// 处理左半部分元素（先插入再查询）
lb.clear();
int lsiz = 0, rsiz = 0;

// 记录当前的查询结果
vector<int> tempAns(qr - ql + 1, 0);

// 第一次扫描：插入左半部分元素并处理查询
for (int i = l; i <= mid; ++i) {
    lb.insert(arr[i], i);
}

// 检查哪些查询可以在左半部分得到答案
for (int i = ql; i <= qr; ++i) {
    int id = eid[i];
    if (queryR[id] <= mid) {
        // 整个查询区间在左半部分
        lset[++lsiz] = id;
    } else if (queryL[id] > mid) {
        // 整个查询区间在右半部分
        rset[++rsiz] = id;
    } else {
        // 查询区间跨越 mid，需要分两次处理
        // 记录当前线性基的最大异或和（左半部分的贡献）
        tempAns[i - ql] = lb.queryMax();
        rset[++rsiz] = id;
    }
}

// 重新排列查询顺序
int idx = ql;
for (int i = 1; i <= lsiz; ++i) {
    eid[idx++] = lset[i];
}
for (int i = 1; i <= rsiz; ++i) {
    eid[idx++] = rset[i];
}

// 递归处理左半部分
solve(ql, ql + lsiz - 1, l, mid);
```

```
    // 处理右半部分
    lb.clear();

    // 从 mid+1 开始插入元素
    for (int i = mid + 1; i <= r; ++i) {
        lb.insert(arr[i], i);
    }

    // 第二次扫描：处理跨越 mid 的查询的右半部分
    int pos = ql + lsiz;
    for (int i = ql; i <= qr; ++i) {
        int id = eid[i];
        if (queryL[id] <= mid && queryR[id] > mid) {
            // 跨越 mid 的查询，需要合并左右两部分的贡献
            // 重新计算左半部分的线性基
            LinearBasis leftLB;
            leftLB.clear();
            for (int j = l; j <= mid; ++j) {
                leftLB.insert(arr[j], j);
            }
            // 合并左右两部分的线性基
            LinearBasis mergeLB;
            mergeLB.clear();
            for (int j = 0; j < LOG; ++j) {
                if (leftLB.basis[j] != 0) {
                    mergeLB.insert(leftLB.basis[j], leftLB.pos[j]);
                }
                if (lb.basis[j] != 0) {
                    mergeLB.insert(lb.basis[j], lb.pos[j]);
                }
            }
            ans[queryId[id]] = mergeLB.queryMax();
        }
    }

    // 递归处理右半部分
    solve(pos, qr, mid + 1, r);
}

int main() {
    // 输入优化
    ios::sync_with_stdio(false);
    cin.tie(0);
```

```
    // 读取数组长度
    cin >> n;

    // 读取数组元素
    for (int i = 1; i <= n; ++i) {
        cin >> arr[i];
    }

    // 读取查询数量
    cin >> q;

    // 读取查询
    for (int i = 1; i <= q; ++i) {
        cin >> queryL[i] >> queryR[i];
        queryId[i] = i;
        eid[i] = i;
    }

    // 初始化答案数组
    for (int i = 1; i <= q; ++i) {
        ans[i] = 0;
    }

    // 整体二分求解
    solve(1, q, 1, n);

    // 输出结果
    for (int i = 1; i <= q; ++i) {
        cout << ans[i] << '\n';
    }

    return 0;
}
```

==========

文件: CF1100F_IvanAndBurgers.java

==========

```
package class169.supplementary_solutions;

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

```
/**
 * Codeforces 1100F Ivan and Burgers - Java实现
 * 题目来源：https://codeforces.com/problemset/problem/1100/F
 * 题目描述：区间最大异或和查询
 *
 * 问题描述：
 * 给定一个长度为n的数组arr，有q条查询，每条查询格式为l r，要求在arr[l..r]中选若干个数，打印最大的异或和。
 *
 * 解题思路：
 * 使用整体二分结合线性基处理区间最大异或和问题。线性基可以高效地处理最大异或和查询，
 * 整体二分帮助我们将所有查询一起处理，优化时间复杂度。
 *
 * 时间复杂度：O((n+q) * log(n) * log(max_value))
 * 空间复杂度：O(n * log(max_value))
 */
public class CF1100F_IvanAndBurgers {
    static final int MAXN = 100005;
    static final int LOG = 20; // 2^20 > 1e6

    static int n, q;
    static int[] arr = new int[MAXN];

    // 查询信息
    static int[] queryL = new int[MAXN];
    static int[] queryR = new int[MAXN];
    static int[] queryId = new int[MAXN];
    static int[] ans = new int[MAXN];

    // 整体二分相关
    static int[] eid = new int[MAXN];
    static int[] lset = new int[MAXN];
    static int[] rset = new int[MAXN];

    // 线性基结构
    static class LinearBasis {
        int[] basis = new int[LOG];
        int[] pos = new int[LOG]; // 记录每个基插入的位置

        // 重置线性基
        void clear() {
            Arrays.fill(basis, 0);
```

```
        Arrays.fill(pos, 0);
    }

    // 插入元素到线性基
    void insert(int val, int position) {
        for (int i = LOG - 1; i >= 0; i--) {
            if ((val >> i) == 0) continue;
            if (basis[i] == 0) {
                basis[i] = val;
                pos[i] = position;
                break;
            }
            if (position > pos[i]) {
                // 交换当前元素和基中的元素
                int temp = basis[i];
                basis[i] = val;
                val = temp;

                temp = pos[i];
                pos[i] = position;
                position = temp;
            }
            val ^= basis[i];
        }
    }

    // 查询区间内的最大异或和
    int queryMax() {
        int res = 0;
        for (int i = LOG - 1; i >= 0; i--) {
            if ((res ^ basis[i]) > res) {
                res ^= basis[i];
            }
        }
        return res;
    }
}

static LinearBasis lb = new LinearBasis();

/**
 * 整体二分核心函数
 * @param ql 查询范围的左端点
```

```
 * @param qr 查询范围的右端点
 * @param l 数组左边界
 * @param r 数组右边界
 */
static void solve(int ql, int qr, int l, int r) {
    if (ql > qr) return;

    if (l == r) {
        // 所有查询的答案都是 arr[l]（如果区间包含 l）
        for (int i = ql; i <= qr; i++) {
            int id = eid[i];
            if (queryL[id] <= l && l <= queryR[id]) {
                ans[queryId[id]] = Math.max(ans[queryId[id]], arr[l]);
            }
        }
        return;
    }

    int mid = (l + r) >> 1;

    // 处理左半部分元素（先插入再查询）
    lb.clear();
    int lsiz = 0, rsiz = 0;

    // 记录当前的查询结果
    int[] tempAns = new int[qr - ql + 1];

    // 第一次扫描：插入左半部分元素并处理查询
    for (int i = l; i <= mid; i++) {
        lb.insert(arr[i], i);
    }

    // 检查哪些查询可以在左半部分得到答案
    for (int i = ql; i <= qr; i++) {
        int id = eid[i];
        if (queryR[id] <= mid) {
            // 整个查询区间在左半部分
            lset[++lsiz] = id;
        } else if (queryL[id] > mid) {
            // 整个查询区间在右半部分
            rset[++rsiz] = id;
        } else {
            // 查询区间跨越 mid，需要分两次处理
```

```
            // 记录当前线性基的最大异或和（左半部分的贡献）
            tempAns[i - ql] = lb.queryMax();
            rset[++rsiz] = id;
        }
    }

    // 重新排列查询顺序
    int idx = ql;
    for (int i = 1; i <= lsiz; i++) {
        eid[idx++] = lset[i];
    }
    for (int i = 1; i <= rsiz; i++) {
        eid[idx++] = rset[i];
    }

    // 递归处理左半部分
    solve(ql, ql + lsiz - 1, l, mid);

    // 处理右半部分
    lb.clear();

    // 从 mid+1 开始插入元素
    for (int i = mid + 1; i <= r; i++) {
        lb.insert(arr[i], i);
    }

    // 第二次扫描：处理跨越 mid 的查询的右半部分
    int pos = ql + lsiz;
    for (int i = ql; i <= qr; i++) {
        int id = eid[i];
        if (queryL[id] <= mid && queryR[id] > mid) {
            // 跨越 mid 的查询，需要合并左右两部分的贡献
            int rightMax = lb.queryMax();
            // 重新计算左半部分的线性基
            LinearBasis leftLB = new LinearBasis();
            for (int j = l; j <= mid; j++) {
                leftLB.insert(arr[j], j);
            }
            // 合并左右两部分的线性基
            LinearBasis mergeLB = new LinearBasis();
            for (int j = 0; j < LOG; j++) {
                if (leftLB.basis[j] != 0) {
                    mergeLB.insert(leftLB.basis[j], leftLB.pos[j]);
```

```
                }
                if (lb.basis[j] != 0) {
                    mergeLB.insert(lb.basis[j], lb.pos[j]);
                }
            }
            ans[queryId[id]] = mergeLB.queryMax();
        }
    }

    // 递归处理右半部分
    solve(pos, qr, mid + 1, r);
}

public static void main(String[] args) throws IOException {
    BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
    PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));

    // 读取数组长度
    n = Integer.parseInt(br.readLine());

    // 读取数组元素
    String[] nums = br.readLine().split(" ");
    for (int i = 1; i <= n; i++) {
        arr[i] = Integer.parseInt(nums[i - 1]);
    }

    // 读取查询数量
    q = Integer.parseInt(br.readLine());

    // 读取查询
    for (int i = 1; i <= q; i++) {
        String[] query = br.readLine().split(" ");
        queryL[i] = Integer.parseInt(query[0]);
        queryR[i] = Integer.parseInt(query[1]);
        queryId[i] = i;
        eid[i] = i;
    }

    // 整体二分求解
    solve(1, q, 1, n);

    // 输出结果
    for (int i = 1; i <= q; i++) {
```

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

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

==========

文件: CF1100F_IvanAndBurgers.py

==========

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

"""
Codeforces 1100F Ivan and Burgers - Python实现
题目来源：https://codeforces.com/problemset/problem/1100/F
题目描述：区间最大异或和查询

问题描述：
给定一个长度为n的数组arr，有q条查询，每条查询格式为l r，要求在arr[l..r]中选若干个数，打印最大的异或和。

解题思路：
使用整体二分结合线性基处理区间最大异或和问题。线性基可以高效地处理最大异或和查询，
整体二分帮助我们将所有查询一起处理，优化时间复杂度。

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

注意：在Python中实现整体二分时，需要注意递归深度限制，对于大规模数据可能需要调整递归深度或转换为迭代实现。
"""

import sys
from sys import stdin

LOG = 20  # 2^20 > 1e6
MAXN = 100005

class LinearBasis:
```

```
    """线性基类，用于处理区间最大异或和问题"""
    def __init__(self):
        self.basis = [0] * LOG
        self.pos = [0] * LOG  # 记录每个基插入的位置

    def clear(self):
        """重置线性基"""
        for i in range(LOG):
            self.basis[i] = 0
            self.pos[i] = 0

    def insert(self, val, position):
        """插入元素到线性基"""
        for i in range(LOG-1, -1, -1):
            if (val >> i) == 0:
                continue
            if self.basis[i] == 0:
                self.basis[i] = val
                self.pos[i] = position
                break
            if position > self.pos[i]:
                # 交换当前元素和基中的元素
                self.basis[i], val = val, self.basis[i]
                self.pos[i], position = position, self.pos[i]
            val ^= self.basis[i]

    def query_max(self):
        """查询区间内的最大异或和"""
        res = 0
        for i in range(LOG-1, -1, -1):
            if (res ^ self.basis[i]) > res:
                res ^= self.basis[i]
        return res

# 全局变量初始化
n = 0
q = 0
arr = [0] * MAXN
queryL = [0] * MAXN
queryR = [0] * MAXN
queryId = [0] * MAXN
ans = [0] * MAXN
eid = [0] * MAXN
```

```
lset = [0] * MAXN
rset = [0] * MAXN
lb = LinearBasis()

def solve(ql, qr, l, r):
    """
    整体二分核心函数

    参数:
        ql: 查询范围的左端点
        qr: 查询范围的右端点
        l: 数组左边界
        r: 数组右边界
    """
    if ql > qr:
        return

    if l == r:
        # 所有查询的答案都是 arr[l]（如果区间包含 l）
        for i in range(ql, qr + 1):
            id = eid[i]
            if queryL[id] <= l and l <= queryR[id]:
                if arr[l] > ans[queryId[id]]:
                    ans[queryId[id]] = arr[l]
        return

    mid = (l + r) // 2

    # 处理左半部分元素（先插入再查询）
    lb.clear()
    lsiz = 0
    rsiz = 0

    # 记录当前的查询结果
    temp_ans = [0] * (qr - ql + 1)

    # 第一次扫描：插入左半部分元素并处理查询
    for i in range(l, mid + 1):
        lb.insert(arr[i], i)

    # 检查哪些查询可以在左半部分得到答案
    for i in range(ql, qr + 1):
        id = eid[i]
```

```
    if queryR[id] <= mid:
        # 整个查询区间在左半部分
        lsiz += 1
        lset[lsiz] = id
    elif queryL[id] > mid:
        # 整个查询区间在右半部分
        rsiz += 1
        rset[rsiz] = id
    else:
        # 查询区间跨越 mid，需要分两次处理
        # 记录当前线性基的最大异或和（左半部分的贡献）
        temp_ans[i - ql] = lb.query_max()
        rsiz += 1
        rset[rsiz] = id

# 重新排列查询顺序
idx = ql
for i in range(1, lsiz + 1):
    eid[idx] = lset[i]
    idx += 1
for i in range(1, rsiz + 1):
    eid[idx] = rset[i]
    idx += 1

# 递归处理左半部分
solve(ql, ql + lsiz - 1, l, mid)

# 处理右半部分
lb.clear()

# 从 mid+1 开始插入元素
for i in range(mid + 1, r + 1):
    lb.insert(arr[i], i)

# 第二次扫描：处理跨越 mid 的查询的右半部分
pos = ql + lsiz
for i in range(ql, qr + 1):
    id = eid[i]
    if queryL[id] <= mid and queryR[id] > mid:
        # 跨越 mid 的查询，需要合并左右两部分的贡献
        # 重新计算左半部分的线性基
        left_lb = LinearBasis()
        left_lb.clear()
```

```
            for j in range(l, mid + 1):
                left_lb.insert(arr[j], j)
            # 合并左右两部分的线性基
            merge_lb = LinearBasis()
            merge_lb.clear()
            for j in range(LOG):
                if left_lb.basis[j] != 0:
                    merge_lb.insert(left_lb.basis[j], left_lb.pos[j])
                if lb.basis[j] != 0:
                    merge_lb.insert(lb.basis[j], lb.pos[j])
            ans[queryId[id]] = merge_lb.query_max()

    # 递归处理右半部分
    solve(pos, qr, mid + 1, r)

def main():
    """主函数，处理输入输出并调用求解函数"""
    # 使用快速输入方法，优化处理大数据量
    input = sys.stdin.read().split()
    ptr = 0

    # 读取数组长度
    global n
    n = int(input[ptr])
    ptr += 1

    # 读取数组元素
    global arr
    for i in range(1, n + 1):
        arr[i] = int(input[ptr])
        ptr += 1

    # 读取查询数量
    global q
    q = int(input[ptr])
    ptr += 1

    # 读取查询
    global queryL, queryR, queryId, eid, ans
    for i in range(1, q + 1):
        queryL[i] = int(input[ptr])
        ptr += 1
        queryR[i] = int(input[ptr])
```

```
        ptr += 1
        queryId[i] = i
        eid[i] = i

    # 初始化答案数组
    ans = [0] * (q + 1)

    # 整体二分求解
    solve(1, q, 1, n)

    # 输出结果
    output = []
    for i in range(1, q + 1):
        output.append(str(ans[i]))
    print('\n'.join(output))

if __name__ == "__main__":
    # 设置递归深度，防止大规模数据导致栈溢出
    sys.setrecursionlimit(1 << 25)
    main()
```

==========

文件: Code01_Juice1.java

==========

```
package class169;

/**
 * 混合果汁问题 - 整体二分算法实现
 *
 * 问题描述：
 * 一共有 n 种果汁，每种果汁有三个属性：美味度 d、每升价格 p、添加上限 l
 * 制作混合果汁时每种果汁不能超过添加上限，其中美味度最低的果汁决定混合果汁的美味度
 * 一共有 m 个小朋友，给每位制作混合果汁时，钱数不超过 money[i]，体积不少于 least[i]
 * 要求打印每个小朋友能得到的混合果汁最大美味度，如果无法满足，打印-1
 *
 * 约束条件：
 * 1 <= n、m、d、p、l <= 10^5
 * 1 <= money[i]、least[i] <= 10^18
 *
 * 解题思路：
 * 使用整体二分算法解决多个具有相同结构的二分答案问题。
 * 1. 将所有果汁按照美味度从高到低排序
```

```
 * 2. 使用整体二分，二分美味度范围
 * 3. 对于每个美味度范围，维护线段树记录价格和数量信息
 * 4. 判断是否能满足小朋友的需求
 *
 * 时间复杂度：O((n+m) * log(n) * log(max_p))
 * 空间复杂度：O(n * log(max_p))
 *
 * 测试链接：https://www.luogu.com.cn/problem/P4602
 * 提交时请将类名改为"Main"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 Code01_Juice1 {

    public static int MAXN = 100001;
    public static int n, m;
    // 果汁有三个参数，美味度 d、每升价格 p、添加上限 l
    public static int[][] juice = new int[MAXN][3];
    // 记录所有小朋友的编号
    public static int[] qid = new int[MAXN];
    // 小朋友能花的钱数
    public static long[] money = new long[MAXN];
    // 小朋友至少的果汁量
    public static long[] least = new long[MAXN];

    // 这种使用线段树的方式叫线段树二分
    // 讲解 146 的题目 2，也涉及线段树二分
    // 果汁单价作为下标的线段树
    // maxp 为最大的果汁单价
    public static int maxp = 0;
    // suml[i] : 线段树某单价区间上，允许添加的总量
    public static long[] suml = new long[MAXN << 2];
    // cost[i] : 线段树某单价区间上，如果允许添加的总量全要，花费多少钱
    public static long[] cost = new long[MAXN << 2];
    // 多少种果汁加入了线段树
    public static int used = 0;

    // 整体二分的过程需要
```

```
public static int[] lset = new int[MAXN];
public static int[] rset = new int[MAXN];

// 每个小朋友的答案，是第几号果汁的美味度
public static int[] ans = new int[MAXN];

/**
 * 线段树的上传操作
 * 将左右子节点的信息合并到当前节点
 * @param i 当前节点的索引
 */
public static void up(int i) {
    suml[i] = suml[i << 1] + suml[i << 1 | 1];
    cost[i] = cost[i << 1] + cost[i << 1 | 1];
}

/**
 * 在线段树中添加果汁信息
 * 单价为 jobi，现在允许添加的量增加了 jobv
 * @param jobi 果汁的单价
 * @param jobv 允许添加的量
 * @param l 当前区间的左端点
 * @param r 当前区间的右端点
 * @param i 当前节点的索引
 */
public static void add(int jobi, int jobv, int l, int r, int i) {
    if (l == r) {
        suml[i] += jobv;
        cost[i] = suml[i] * l;
    } else {
        int mid = (l + r) >> 1;
        if (jobi <= mid) {
            add(jobi, jobv, l, mid, i << 1);
        } else {
            add(jobi, jobv, mid + 1, r, i << 1 | 1);
        }
        up(i);
    }
}

/**
 * 查询在总体积为 volume 的情况下，能花的最少钱数
 * 总体积一共 volume，已知总体积一定能耗尽
```

```
 * 返回总体积耗尽的情况下，能花的最少钱数
 * @param volume 总体积
 * @param l 当前区间的左端点
 * @param r 当前区间的右端点
 * @param i 当前节点的索引
 * @return 能花的最少钱数
 */
public static long query(long volume, int l, int r, int i) {
    if (l == r) {
        return volume * l;
    }
    int mid = (l + r) >> 1;
    if (suml[i << 1] >= volume) {
        return query(volume, l, mid, i << 1);
    } else {
        return cost[i << 1] + query(volume - suml[i << 1], mid + 1, r, i << 1 | 1);
    }
}

/**
 * 整体二分核心函数
 * @param ql 查询范围的左端点
 * @param qr 查询范围的右端点
 * @param vl 值域范围的左端点（美味度范围）
 * @param vr 值域范围的右端点（美味度范围）
 */
public static void compute(int ql, int qr, int vl, int vr) {
    // 递归边界条件
    if (ql > qr) {
        return;
    }

    // 如果值域范围只有一个值，说明找到了答案
    if (vl == vr) {
        for (int i = ql; i <= qr; i++) {
            ans[qid[i]] = vl;
        }
    } else {
        // 二分中点
        int mid = (vl + vr) >> 1;

        // 线段树包含果汁的种类少就添加
        while (used < mid) {
```

```
            used++;
            add(juice[used][1], juice[used][2], 1, maxp, 1);
        }

        // 线段树包含果汁的种类多就撤销
        while (used > mid) {
            add(juice[used][1], -juice[used][2], 1, maxp, 1);
            used--;
        }

        // 检查每个小朋友的查询，根据是否能满足需求划分到左右区间
        int lsiz = 0, rsiz = 0;
        for (int i = ql, id; i <= qr; i++) {
            id = qid[i];
            // 检查当前果汁组合是否能满足小朋友的需求
            if (suml[1] >= least[id] && query(least[id], 1, maxp, 1) <= money[id]) {
                // 满足需求，放入左集合
                lset[++lsiz] = id;
            } else {
                // 不满足需求，放入右集合
                rset[++rsiz] = id;
            }
        }

        // 重新排列查询顺序
        for (int i = 1; i <= lsiz; i++) {
            qid[ql + i - 1] = lset[i];
        }
        for (int i = 1; i <= rsiz; i++) {
            qid[ql + lsiz + i - 1] = rset[i];
        }

        // 递归处理左右两部分
        compute(ql, ql + lsiz - 1, vl, mid);
        compute(ql + lsiz, qr, mid + 1, vr);
    }
}

/**
 * 主函数，程序入口
 * @param args 命令行参数
 * @throws Exception 输入输出异常
 */
```

```
public static void main(String[] args) throws Exception {
    FastReader in = new FastReader(System.in);
    PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));

    // 读取果汁数量 n 和小朋友数量 m
    n = in.nextInt();
    m = in.nextInt();

    // 读取每种果汁的信息：美味度、价格、上限
    for (int i = 1; i <= n; i++) {
        juice[i][0] = in.nextInt(); // 美味度
        juice[i][1] = in.nextInt(); // 价格
        juice[i][2] = in.nextInt(); // 上限
        maxp = Math.max(maxp, juice[i][1]); // 更新最大价格
    }

    // 读取每个小朋友的需求：钱数和最少果汁量
    for (int i = 1; i <= m; i++) {
        qid[i] = i; // 记录小朋友编号
        money[i] = in.nextLong(); // 钱数
        least[i] = in.nextLong(); // 最少果汁量
    }

    // 所有果汁按照美味度排序，美味度大的在前，美味度小的在后
    Arrays.sort(juice, 1, n + 1, (a, b) -> b[0] - a[0]);

    // 答案范围就是美味度范围，从最美味的第 1 名，到最难喝的第 n 名
    // 如果小朋友答案为 n+1，说明无法满足这个小朋友
    compute(1, m, 1, n + 1);

    // 输出每个小朋友能得到的最大美味度
    for (int i = 1; i <= m; i++) {
        if (ans[i] == n + 1) {
            // 无法满足需求
            out.println(-1);
        } else {
            // 输出对应果汁的美味度
            out.println(juice[ans[i]][0]);
        }
    }

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

```
}

// 读写工具类
static class FastReader {
    private final byte[] buffer = new byte[1 << 16];
    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;
    }

    long nextLong() throws IOException {
        int c;
        do {
```

```
            c = readByte();
        } while (c <= ' ' && c != -1);
        boolean neg = false;
        if (c == '-') {
            neg = true;
            c = readByte();
        }
        long val = 0L;
        while (c > ' ' && c != -1) {
            val = val * 10 + (c - '0');
            c = readByte();
        }
        return neg ? -val : val;
    }

}

}
```

==========

文件: Code01_Juice2.java

==========

```
package class169;

// 混合果汁，C++版
// 一共有 n 种果汁，每种果汁给定，美味度 d、每升价格 p、添加上限 l
// 制作混合果汁时每种果汁不能超过添加上限，其中美味度最低的果汁，决定混合果汁的美味度
// 一共有 m 个小朋友，给每位制作混合果汁时，钱数不超过 money[i]，体积不少于 least[i]
// 打印每个小朋友能得到的混合果汁最大美味度，如果无法满足，打印-1
// 1 <= n、m、d、p、l <= 10^5
// 1 <= money[i]、least[i] <= 10^18
// 测试链接 : https://www.luogu.com.cn/problem/P4602
// 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 Juice {
//    int d, p, l;
//};
```

```
//
//bool JuiceCmp(Juice x, Juice y) {
//    return x.d > y.d;
//}
//
//const int MAXN = 100001;
//int n, m;
//
//Juice juice[MAXN];
//int qid[MAXN];
//long long money[MAXN];
//long long least[MAXN];
//
//int maxp = 0;
//long long suml[MAXN << 2];
//long long cost[MAXN << 2];
//int used = 0;
//
//int lset[MAXN];
//int rset[MAXN];
//
//int ans[MAXN];
//
//void up(int i) {
//    suml[i] = suml[i << 1] + suml[i << 1 | 1];
//    cost[i] = cost[i << 1] + cost[i << 1 | 1];
//}
//
//void add(int jobi, int jobv, int l, int r, int i) {
//    if (l == r) {
//        suml[i] += jobv;
//        cost[i] = suml[i] * l;
//    } else {
//        int mid = (l + r) >> 1;
//        if (jobi <= mid) {
//            add(jobi, jobv, l, mid, i << 1);
//        } else {
//            add(jobi, jobv, mid + 1, r, i << 1 | 1);
//        }
//        up(i);
//    }
//}
//
```

```
//long long query(long long volume, int l, int r, int i) {
//    if (l == r) {
//        return volume * l;
//    }
//    int mid = (l + r) >> 1;
//    if (suml[i << 1] >= volume) {
//        return query(volume, l, mid, i << 1);
//    } else {
//        return cost[i << 1] + query(volume - suml[i << 1], mid + 1, r, i << 1 | 1);
//    }
//}
//
//void compute(int ql, int qr, int vl, int vr) {
//    if (ql > qr) {
//        return;
//    }
//    if (vl == vr) {
//        for (int i = ql; i <= qr; i++) {
//            ans[qid[i]] = vl;
//        }
//    } else {
//        int mid = (vl + vr) >> 1;
//        while (used < mid) {
//            used++;
//            add(juice[used].p, juice[used].l, 1, maxp, 1);
//        }
//        while (used > mid) {
//            add(juice[used].p, -juice[used].l, 1, maxp, 1);
//            used--;
//        }
//        int lsiz = 0, rsiz = 0;
//        for (int i = ql, id; i <= qr; i++) {
//            id = qid[i];
//            if (suml[1] >= least[id] && query(least[id], 1, maxp, 1) <= money[id]) {
//                lset[++lsiz] = id;
//            } else {
//                rset[++rsiz] = id;
//            }
//        }
//        for (int i = 1; i <= lsiz; i++) {
//            qid[ql + i - 1] = lset[i];
//        }
//        for (int i = 1; i <= rsiz; i++) {
```

```
//            qid[ql + lsiz + i - 1] = rset[i];
//        }
//        compute(ql, ql + lsiz - 1, vl, mid);
//        compute(ql + lsiz, qr, mid + 1, vr);
//    }
//}
//
//int main() {
//    ios::sync_with_stdio(false);
//    cin.tie(nullptr);
//    cin >> n >> m;
//    for (int i = 1; i <= n; i++) {
//        cin >> juice[i].d >> juice[i].p >> juice[i].l;
//        maxp = max(maxp, juice[i].p);
//    }
//    for (int i = 1; i <= m; i++) {
//        qid[i] = i;
//        cin >> money[i] >> least[i];
//    }
//    sort(juice + 1, juice + n + 1, JuiceCmp);
//    compute(1, m, 1, n + 1);
//    for (int i = 1; i <= m; i++) {
//        if (ans[i] == n + 1) {
//            cout << -1 << '\n';
//        } else {
//            cout << juice[ans[i]].d << '\n';
//        }
//    }
//    return 0;
//}
```

==========

文件: Code02_DynamicRankings1.java

==========

```
package class169;

/**
 * 带修改的区间第 k 小问题 - 整体二分算法实现
 *
 * 问题描述：
 * 给定一个长度为 n 的数组 arr，接下来是 m 条操作，每种操作是如下两种类型的一种
 * 操作 C x y   ：把 x 位置的值修改成 y
```

```
 * 操作 Q x y v : 查询 arr[x..y]范围上第 v 小的值
 *
 * 约束条件：
 * 1 <= n、m <= 10^5
 * 1 <= 数组中的值 <= 10^9
 *
 * 解题思路：
 * 使用整体二分算法解决带修改的区间第 k 小问题。
 * 1. 将所有操作（查询和修改）一起处理
 * 2. 二分答案的值域，利用树状数组维护区间内小于等于 mid 的元素个数
 * 3. 根据查询结果将操作分为两部分递归处理
 *
 * 时间复杂度：O((n+m) * log(n) * log(max_value))
 * 空间复杂度：O(n)
 *
 * 测试链接：https://www.luogu.com.cn/problem/P2617
 * 本题是讲解 160，树套树模版题，现在作为带修改的整体二分模版题
 * 提交时请将类名改为"Main"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 Code02_DynamicRankings1 {

    public static int MAXN = 100001;
    public static int MAXE = MAXN << 2;
    public static int INF = 1000000001;
    public static int n, m;

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

    // 事件编号组成的数组
    public static int[] eid = new int[MAXE];
    // op == 1，代表修改事件，x 处，值 y，效果 v
    // op == 2，代表查询事件，[x..y]范围上查询第 v 小，q 表示问题的编号
    public static int[] op = new int[MAXE];
    public static int[] x = new int[MAXE];
    public static int[] y = new int[MAXE];
    public static int[] v = new int[MAXE];
    public static int[] q = new int[MAXE];
```

```
public static int cnte = 0;
public static int cntq = 0;

// 树状数组
public static int[] tree = new int[MAXN];

// 整体二分
public static int[] lset = new int[MAXE];
public static int[] rset = new int[MAXE];

// 查询的答案
public static int[] ans = new int[MAXN];

/**
 * 计算一个数的 lowbit 值
 * @param i 输入的数
 * @return lowbit 值
 */
public static int lowbit(int i) {
    return i & -i;
}

/**
 * 在树状数组中给位置 i 增加 v
 * @param i 位置
 * @param v 增加的值
 */
public static void add(int i, int v) {
    while (i <= n) {
        tree[i] += v;
        i += lowbit(i);
    }
}

/**
 * 计算前缀和[1..i]
 * @param i 位置
 * @return 前缀和
 */
public static int sum(int i) {
    int ret = 0;
    while (i > 0) {
        ret += tree[i];
```

```
        i -= lowbit(i);
    }
    return ret;
}

/**
 * 计算区间和[l..r]
 * @param l 左端点
 * @param r 右端点
 * @return 区间和
 */
public static int query(int l, int r) {
    return sum(r) - sum(l - 1);
}

/**
 * 整体二分核心函数
 * @param el 操作范围的左端点
 * @param er 操作范围的右端点
 * @param vl 值域范围的左端点
 * @param vr 值域范围的右端点
 */
public static void compute(int el, int er, int vl, int vr) {
    // 递归边界条件
    if (el > er) {
        return;
    }

    // 如果值域范围只有一个值，说明找到了答案
    if (vl == vr) {
        for (int i = el; i <= er; i++) {
            int id = eid[i];
            if (op[id] == 2) {
                // 查询操作，记录答案
                ans[q[id]] = vl;
            }
        }
    } else {
        // 二分中点
        int mid = (vl + vr) >> 1;

        // 将操作分为左右两部分
        int lsiz = 0, rsiz = 0;
```

```
for (int i = el; i <= er; i++) {
    int id = eid[i];
    if (op[id] == 1) {
        // 修改操作
        if (y[id] <= mid) {
            // 值小于等于 mid，加入左集合
            add(x[id], v[id]);
            lset[++lsiz] = id;
        } else {
            // 值大于 mid，加入右集合
            rset[++rsiz] = id;
        }
    } else {
        // 查询操作
        int satisfy = query(x[id], y[id]);
        if (v[id] <= satisfy) {
            // 第 k 小的数在左半部分
            lset[++lsiz] = id;
        } else {
            // 第 k 小的数在右半部分，需要在右半部分找第(k-satisfy)小的数
            v[id] -= satisfy;
            rset[++rsiz] = id;
        }
    }
}

// 重新排列操作顺序
for (int i = 1; i <= lsiz; i++) {
    eid[el + i - 1] = lset[i];
}
for (int i = 1; i <= rsiz; i++) {
    eid[el + lsiz + i - 1] = rset[i];
}

// 撤销修改操作的影响
for (int i = 1; i <= lsiz; i++) {
    int id = lset[i];
    if (op[id] == 1 && y[id] <= mid) {
        add(x[id], -v[id]);
    }
}

// 递归处理左右两部分
```

```
        compute(el, el + lsiz - 1, vl, mid);
        compute(el + lsiz, er, mid + 1, vr);
    }
}

/**
 * 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();
    PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));

    // 读取数组长度 n 和操作数量 m
    n = in.nextInt();
    m = in.nextInt();

    // 读取初始数组，并将每个元素作为修改操作记录
    for (int i = 1; i <= n; i++) {
        arr[i] = in.nextInt();
        op[++cnte] = 1;  // 修改操作
        x[cnte] = i;     // 位置
        y[cnte] = arr[i]; // 值
        v[cnte] = 1;     // 效果
    }

    char type;
    // 读取 m 条操作
    for (int i = 1; i <= m; i++) {
        type = in.nextChar();
        if (type == 'C') {
            // 修改操作 C x y : 把 x 位置的值修改成 y
            int a = in.nextInt(); // 位置 x
            int b = in.nextInt(); // 值 y

            // 将修改操作拆分为两个操作：删除旧值和插入新值
            op[++cnte] = 1;  // 删除旧值
            x[cnte] = a;
            y[cnte] = arr[a]; // 旧值
            v[cnte] = -1;    // 删除效果

            op[++cnte] = 1;  // 插入新值
```

```
            x[cnte] = a;
            y[cnte] = b;     // 新值
            v[cnte] = 1;     // 插入效果

            arr[a] = b; // 更新数组
        } else {
            // 查询操作 Q x y v : 查询 arr[x..y]范围上第 v 小的值
            op[++cnte] = 2;     // 查询操作
            x[cnte] = in.nextInt(); // 左端点
            y[cnte] = in.nextInt(); // 右端点
            v[cnte] = in.nextInt(); // 第 k 小
            q[cnte] = ++cntq;        // 查询编号
        }
    }

    // 初始化事件编号数组
    for (int i = 1; i <= cnte; i++) {
        eid[i] = i;
    }

    // 整体二分求解
    compute(1, cnte, 0, INF);

    // 输出查询结果
    for (int i = 1; i <= cntq; i++) {
        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: Code02_DynamicRankings2.java

==========

```
package class169;

// 带修改的区间第 k 小，C++版
// 给定一个长度为 n 的数组 arr，接下来是 m 条操作，每种操作是如下两种类型的一种
// 操作 C x y  ：把 x 位置的值修改成 y
// 操作 Q x y v ：查询 arr[x..y]范围上第 v 小的值
// 1 <= n、m <= 10^5
// 1 <= 数组中的值 <= 10^9
// 测试链接 : https://www.luogu.com.cn/problem/P2617
// 本题是讲解 160，树套树模版题，现在作为带修改的整体二分模版题
// 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 = 100001;
//const int MAXE = MAXN << 2;
//const int INF = 1000000001;
//int n, m;
//
//int arr[MAXN];
//int eid[MAXE];
//int op[MAXE];
//int x[MAXE];
//int y[MAXE];
```

```
//int v[MAXE];
//int q[MAXE];
//int cnte = 0;
//int cntq = 0;
//
//int tree[MAXN];
//
//int lset[MAXE];
//int rset[MAXE];
//
//int ans[MAXN];
//
//int lowbit(int i) {
//    return i & -i;
//}
//
//void add(int i, int v) {
//    while (i <= n) {
//        tree[i] += v;
//        i += lowbit(i);
//    }
//}
//
//int sum(int i) {
//    int ret = 0;
//    while (i > 0) {
//        ret += tree[i];
//        i -= lowbit(i);
//    }
//    return ret;
//}
//
//int query(int l, int r) {
//    return sum(r) - sum(l - 1);
//}
//
//void compute(int el, int er, int vl, int vr) {
//    if (el > er) {
//        return;
//    }
//    if (vl == vr) {
//        for (int i = el; i <= er; i++) {
//            int id = eid[i];
```

```
//            if (op[id] == 2) {
//                ans[q[id]] = vl;
//            }
//        }
//    } else {
//        int mid = (vl + vr) >> 1;
//        int lsiz = 0, rsiz = 0;
//        for (int i = el; i <= er; i++) {
//            int id = eid[i];
//            if (op[id] == 1) {
//                if (y[id] <= mid) {
//                    add(x[id], v[id]);
//                    lset[++lsiz] = id;
//                } else {
//                    rset[++rsiz] = id;
//                }
//            } else {
//                int satisfy = query(x[id], y[id]);
//                if (v[id] <= satisfy) {
//                    lset[++lsiz] = id;
//                } else {
//                    v[id] -= satisfy;
//                    rset[++rsiz] = id;
//                }
//            }
//        }
//        for (int i = 1; i <= lsiz; i++) {
//            eid[el + i - 1] = lset[i];
//        }
//        for (int i = 1; i <= rsiz; i++) {
//            eid[el + lsiz + i - 1] = rset[i];
//        }
//        for (int i = 1; i <= lsiz; i++) {
//            int id = lset[i];
//            if (op[id] == 1 && y[id] <= mid) {
//                add(x[id], -v[id]);
//            }
//        }
//        compute(el, el + lsiz - 1, vl, mid);
//        compute(el + lsiz, er, mid + 1, vr);
//    }
//}
//
```

```
//int main() {
//    ios::sync_with_stdio(false);
//    cin.tie(nullptr);
//    cin >> n >> m;
//    for (int i = 1; i <= n; i++) {
//        cin >> arr[i];
//        op[++cnte] = 1;
//        x[cnte] = i;
//        y[cnte] = arr[i];
//        v[cnte] = 1;
//    }
//    for (int i = 1; i <= m; i++) {
//        char type;
//        cin >> type;
//        if (type == 'C') {
//            int a, b;
//            cin >> a >> b;
//            op[++cnte] = 1;
//            x[cnte] = a;
//            y[cnte] = arr[a];
//            v[cnte] = -1;
//            op[++cnte] = 1;
//            x[cnte] = a;
//            y[cnte] = b;
//            v[cnte] = 1;
//            arr[a] = b;
//        } else {
//            op[++cnte] = 2;
//            cin >> x[cnte] >> y[cnte] >> v[cnte];
//            q[cnte] = ++cntq;
//        }
//    }
//    for (int i = 1; i <= cnte; i++) {
//        eid[i] = i;
//    }
//    compute(1, cnte, 0, INF);
//    for (int i = 1; i <= cntq; i++) {
//        cout << ans[i] << '\n';
//    }
//    return 0;
//}

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

文件: Code03_Network1.java

```
==========================================
package class169;

// 网络，java 版
// 一共有 n 个服务器，给定 n-1 条边，所有服务器连成一棵树
// 某两个服务器之间的路径上，可能接受一条请求，路径上的所有服务器都需要保存该请求的重要度
// 一共有 m 条操作，每条操作是如下 3 种类型中的一种，操作依次发生，第 i 条操作发生的时间为 i
// 操作 0 x y v : x 号服务器到 y 号服务器的路径上，增加了一个重要度为 v 的请求
// 操作 1 t     : 当初时间为 t 的操作，一定是增加请求的操作，现在这个请求结束了
// 操作 2 x     : 当前时间下，和 x 号服务器无关的所有请求中，打印最大的重要度
// 关于操作 2，如果当前时间下，没有任何请求、或者所有请求都和 x 号服务器有关，打印-1
// 2 <= n <= 10^5    1 <= m <= 2 * 10^5    1 <= 重要度 <= 10^9
// 测试链接 : https://www.luogu.com.cn/problem/P3250
// 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 Code03_Network1 {

    public static int MAXN = 100001;
    public static int MAXM = 200001;
    public static int MAXH = 20;
    public static int INF = 1000000001;
    public static int n, m;

    // 链式前向星
    public static int[] head = new int[MAXN];
    public static int[] next = new int[MAXN << 1];
    public static int[] to = new int[MAXN << 1];
    public static int cntg = 0;

    // 树上点差分 + 树上倍增
    public static int[] fa = new int[MAXN];
    public static int[] dep = new int[MAXN];
    public static int[] siz = new int[MAXN];
    public static int[] dfn = new int[MAXN];
    public static int[][] stjump = new int[MAXN][MAXH];
    public static int cntd;
```

```
// 树状数组
public static int[] tree = new int[MAXN];

// 事件编号组成的数组
public static int[] eid = new int[MAXM];
// 如果 op == 0，添加点 x 到点 y，重要度为 v 的路径
// 如果 op == 1，删除点 x 到点 y，重要度为 v 的路径
// 如果 op == 2，查询和 x 相关的答案，y 表示问题的编号
public static int[] op = new int[MAXM];
public static int[] x = new int[MAXM];
public static int[] y = new int[MAXM];
public static int[] v = new int[MAXM];
public static int cntq = 0;

// 整体二分
public static int[] lset = new int[MAXM];
public static int[] rset = new int[MAXM];

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

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

// 递归版，C++可以通过，java 会爆栈
public static void dfs1(int u, int f) {
    fa[u] = f;
    dep[u] = dep[f] + 1;
    siz[u] = 1;
    dfn[u] = ++cntd;
    stjump[u][0] = f;
    for (int p = 1; p < MAXH; p++) {
        stjump[u][p] = stjump[stjump[u][p - 1]][p - 1];
    }
    for (int e = head[u]; e != 0; e = next[e]) {
        if (to[e] != f) {
            dfs1(to[e], u);
        }
    }
    for (int e = head[u]; e != 0; e = next[e]) {
```

```
        if (to[e] != f) {
            siz[u] += siz[to[e]];
        }
    }
}

// 不会改迭代版，去看讲解 118，详解了从递归版改迭代版
public static int[][] ufe = new int[MAXN][3];

public static int stacksize, u, f, e;

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

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

// dfs1 的迭代版
public static void dfs2() {
    stacksize = 0;
    push(1, 0, -1);
    while (stacksize > 0) {
        pop();
        if (e == -1) {
            fa[u] = f;
            dep[u] = dep[f] + 1;
            siz[u] = 1;
            dfn[u] = ++cntd;
            stjump[u][0] = f;
            for (int p = 1; p < MAXH; p++) {
                stjump[u][p] = stjump[stjump[u][p - 1]][p - 1];
            }
            e = head[u];
        } else {
            e = next[e];
```

```
        }
        if (e != 0) {
            push(u, f, e);
            if (to[e] != f) {
                push(to[e], u, -1);
            }
        } else {
            for (int e = head[u]; e != 0; e = next[e]) {
                if (to[e] != f) {
                    siz[u] += siz[to[e]];
                }
            }
        }
    }
}

public static int lca(int a, int b) {
    if (dep[a] < dep[b]) {
        int tmp = a;
        a = b;
        b = tmp;
    }
    for (int p = MAXH - 1; p >= 0; p--) {
        if (dep[stjump[a][p]] >= dep[b]) {
            a = stjump[a][p];
        }
    }
    if (a == b) {
        return a;
    }
    for (int p = MAXH - 1; p >= 0; p--) {
        if (stjump[a][p] != stjump[b][p]) {
            a = stjump[a][p];
            b = stjump[b][p];
        }
    }
    return stjump[a][0];
}

public static int lowbit(int i) {
    return i & -i;
}
```

```
public static void add(int i, int v) {
    while (i <= n) {
        tree[i] += v;
        i += lowbit(i);
    }
}

public static int query(int i) {
    int ret = 0;
    while (i > 0) {
        ret += tree[i];
        i -= lowbit(i);
    }
    return ret;
}

// 点 x 到点 y 的路径上，每个点增加 v 个请求数量
public static void pathAdd(int x, int y, int v) {
    int xylca = lca(x, y);
    int lcafa = fa[xylca];
    add(dfn[x], v);
    add(dfn[y], v);
    add(dfn[xylca], -v);
    if (lcafa != 0) {
        add(dfn[lcafa], -v);
    }
}

// 查询和 x 点相关的请求数量
public static int pointQuery(int x) {
    return query(dfn[x] + siz[x] - 1) - query(dfn[x] - 1);
}

public static void compute(int el, int er, int vl, int vr) {
    if (el > er) {
        return;
    }
    if (vl == vr) {
        for (int i = el; i <= er; i++) {
            int id = eid[i];
            if (op[id] == 2) {
                ans[y[id]] = vl;
            }
```

```
    }
} else {
    int mid = (vl + vr) / 2;
    int lsiz = 0, rsiz = 0, request = 0;
    for (int i = el; i <= er; i++) {
        int id = eid[i];
        if (op[id] == 0) {
            if (v[id] <= mid) {
                lset[++lsiz] = id;
            } else {
                pathAdd(x[id], y[id], 1);
                request++;
                rset[++rsiz] = id;
            }
        } else if (op[id] == 1) {
            if (v[id] <= mid) {
                lset[++lsiz] = id;
            } else {
                pathAdd(x[id], y[id], -1);
                request--;
                rset[++rsiz] = id;
            }
        } else {
            if (pointQuery(x[id]) == request) {
                lset[++lsiz] = id;
            } else {
                rset[++rsiz] = id;
            }
        }
    }
    for (int i = 1; i <= lsiz; i++) {
        eid[el + i - 1] = lset[i];
    }
    for (int i = 1; i <= rsiz; i++) {
        eid[el + lsiz + i - 1] = rset[i];
    }
    for (int i = 1; i <= rsiz; i++) {
        int id = rset[i];
        if (op[id] == 0 && v[id] > mid) {
            pathAdd(x[id], y[id], -1);
        }
        if (op[id] == 1 && v[id] > mid) {
            pathAdd(x[id], y[id], 1);
```

```
            }
        }
        compute(el, el + lsiz - 1, vl, mid);
        compute(el + lsiz, er, mid + 1, vr);
    }
}

public static void prepare() {
    dfs2();
    for (int i = 1; i <= m; i++) {
        if (op[i] == 1) {
            int pre = x[i];
            x[i] = x[pre];
            y[i] = y[pre];
            v[i] = v[pre];
        }
        if (op[i] == 2) {
            y[i] = ++cntq;
        }
    }
    for (int i = 1; i <= m; i++) {
        eid[i] = i;
    }
}

public static void main(String[] args) throws IOException {
    FastReader in = new FastReader(System.in);
    PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));
    n = in.nextInt();
    m = in.nextInt();
    for (int i = 1, u, v; i < n; i++) {
        u = in.nextInt();
        v = in.nextInt();
        addEdge(u, v);
        addEdge(v, u);
    }
    for (int i = 1; i <= m; i++) {
        op[i] = in.nextInt();
        x[i] = in.nextInt();
        if (op[i] == 0) {
            y[i] = in.nextInt();
            v[i] = in.nextInt();
        }
```

```
        }
        prepare();
        compute(1, m, 0, INF);
        for (int i = 1; i <= cntq; i++) {
            if (ans[i] == 0) {
                out.println(-1);
            } else {
                out.println(ans[i]);
            }
        }
        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: Code03_Network2.java

==========

```
package class169;

// 网络，C++版
// 一共有 n 个服务器，给定 n-1 条边，所有服务器连成一棵树
// 某两个服务器之间的路径上，可能接受一条请求，路径上的所有服务器都需要保存该请求的重要度
// 一共有 m 条操作，每条操作是如下 3 种类型中的一种，操作依次发生，第 i 条操作发生的时间为 i
// 操作 0 x y v : x 号服务器到 y 号服务器的路径上，增加了一个重要度为 v 的请求
// 操作 1 t     : 当初时间为 t 的操作，一定是增加请求的操作，现在这个请求结束了
// 操作 2 x     : 当前时间下，和 x 号服务器无关的所有请求中，打印最大的重要度
// 关于操作 2，如果当前时间下，没有任何请求、或者所有请求都和 x 号服务器有关，打印-1
// 2 <= n <= 10^5    1 <= m <= 2 * 10^5    1 <= 重要度 <= 10^9
// 测试链接 : https://www.luogu.com.cn/problem/P3250
// 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 = 100001;
//const int MAXM = 200001;
//const int MAXH = 20;
//const int INF = 1000000001;
//int n, m;
//
//int head[MAXN];
//int nxt[MAXN << 1];
```

```
//int to[MAXN << 1];
//int cntg = 0;
//
//int fa[MAXN];
//int dep[MAXN];
//int siz[MAXN];
//int dfn[MAXN];
//int stjump[MAXN][MAXH];
//int cntd = 0;
//
//int tree[MAXN];
//
//int eid[MAXM];
//int op[MAXM];
//int x[MAXM];
//int y[MAXM];
//int v[MAXM];
//int cntq = 0;
//
//int lset[MAXM];
//int rset[MAXM];
//
//int ans[MAXM];
//
//void addEdge(int u, int v) {
//    nxt[++cntg] = head[u];
//    to[cntg] = v;
//    head[u] = cntg;
//}
//
//void dfs(int u, int f) {
//    fa[u] = f;
//    dep[u] = dep[f] + 1;
//    siz[u] = 1;
//    dfn[u] = ++cntd;
//    stjump[u][0] = f;
//    for (int p = 1; p < MAXH; p++) {
//        stjump[u][p] = stjump[stjump[u][p - 1]][p - 1];
//    }
//    for (int e = head[u]; e != 0; e = nxt[e]) {
//        if (to[e] != f) {
//            dfs(to[e], u);
//        }
```

```
//    }
//    for (int e = head[u]; e != 0; e = nxt[e]) {
//        if (to[e] != f) {
//            siz[u] += siz[to[e]];
//        }
//    }
//}
//
//int lca(int a, int b) {
//    if (dep[a] < dep[b]) {
//        int tmp = a;
//        a = b;
//        b = tmp;
//    }
//    for (int p = MAXH - 1; p >= 0; p--) {
//        if (dep[stjump[a][p]] >= dep[b]) {
//            a = stjump[a][p];
//        }
//    }
//    if (a == b) {
//        return a;
//    }
//    for (int p = MAXH - 1; p >= 0; p--) {
//        if (stjump[a][p] != stjump[b][p]) {
//            a = stjump[a][p];
//            b = stjump[b][p];
//        }
//    }
//    return stjump[a][0];
//}
//
//int lowbit(int i) {
//    return i & -i;
//}
//
//void add(int i, int v) {
//    while (i <= n) {
//        tree[i] += v;
//        i += lowbit(i);
//    }
//}
//
//int query(int i) {
```

```
//    int ret = 0;
//    while (i > 0) {
//        ret += tree[i];
//        i -= lowbit(i);
//    }
//    return ret;
//}
//
//void pathAdd(int x, int y, int v) {
//    int xylca = lca(x, y);
//    int lcafa = fa[xylca];
//    add(dfn[x], v);
//    add(dfn[y], v);
//    add(dfn[xylca], -v);
//    if (lcafa != 0) {
//        add(dfn[lcafa], -v);
//    }
//}
//
//int pointQuery(int x) {
//    return query(dfn[x] + siz[x] - 1) - query(dfn[x] - 1);
//}
//
//void compute(int el, int er, int vl, int vr) {
//    if (el > er) {
//        return;
//    }
//    if (vl == vr) {
//        for (int i = el; i <= er; i++) {
//            int id = eid[i];
//            if (op[id] == 2) {
//                ans[y[id]] = vl;
//            }
//        }
//    } else {
//        int mid = (vl + vr) / 2;
//        int lsiz = 0, rsiz = 0, request = 0;
//        for (int i = el; i <= er; i++) {
//            int id = eid[i];
//            if (op[id] == 0) {
//                if (v[id] <= mid) {
//                    lset[++lsiz] = id;
//                } else {
```

```
//                    pathAdd(x[id], y[id], 1);
//                    request++;
//                    rset[++rsiz] = id;
//                }
//            } else if (op[id] == 1) {
//                if (v[id] <= mid) {
//                    lset[++lsiz] = id;
//                } else {
//                    pathAdd(x[id], y[id], -1);
//                    request--;
//                    rset[++rsiz] = id;
//                }
//            } else {
//                if (pointQuery(x[id]) == request) {
//                    lset[++lsiz] = id;
//                } else {
//                    rset[++rsiz] = id;
//                }
//            }
//        }
//        for (int i = 1; i <= lsiz; i++) {
//            eid[el + i - 1] = lset[i];
//        }
//        for (int i = 1; i <= rsiz; i++) {
//            eid[el + lsiz + i - 1] = rset[i];
//        }
//        for (int i = 1; i <= rsiz; i++) {
//            int id = rset[i];
//            if (op[id] == 0 && v[id] > mid) {
//                pathAdd(x[id], y[id], -1);
//            }
//            if (op[id] == 1 && v[id] > mid) {
//                pathAdd(x[id], y[id], 1);
//            }
//        }
//        compute(el, el + lsiz - 1, vl, mid);
//        compute(el + lsiz, er, mid + 1, vr);
//    }
//}
//
//void prepare() {
//    dfs(1, 0);
//    for (int i = 1; i <= m; i++) {
```

```
//        if (op[i] == 1) {
//            int pre = x[i];
//            x[i] = x[pre];
//            y[i] = y[pre];
//            v[i] = v[pre];
//        }
//        if (op[i] == 2) {
//            y[i] = ++cntq;
//        }
//    }
//    for (int i = 1; i <= m; i++) {
//        eid[i] = i;
//    }
//}
//
//int main() {
//    ios::sync_with_stdio(false);
//    cin.tie(nullptr);
//    cin >> n >> m;
//    for (int i = 1, u, v; i < n; i++) {
//        cin >> u >> v;
//        addEdge(u, v);
//        addEdge(v, u);
//    }
//    for (int i = 1; i <= m; i++) {
//        cin >> op[i] >> x[i];
//        if (op[i] == 0) {
//            cin >> y[i] >> v[i];
//        }
//    }
//    prepare();
//    compute(1, m, 0, INF);
//    for (int i = 1; i <= cntq; i++) {
//        if (ans[i] == 0) {
//            cout << -1 << '\n';
//        } else {
//            cout << ans[i] << '\n';
//        }
//    }
//    return 0;
//}

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

文件: Code04_Fruit1.java

```
============================================
package class169;

// 接水果，java 版
// 一共有 n 个点，给定 n-1 条无向边，所有点连成一棵树
// 一共有 p 个盘子，每个盘子格式 a b c : 盘子是点 a 到点 b 的路径，盘子权值为 c
// 一共有 q 个水果，每个水果格式 u v k : 水果是点 u 到点 v 的路径，k 含义如下
// 如果一个盘子路径完全在一个水果路径的内部，那么该盘子可以接住该水果
// 那么对于每个水果，可能有很多盘子都可以将其接住，打印其中第 k 小的权值
// 1 <= n、p、q <= 4 * 10^4
// 0 <= 盘子权值 <= 10^9
// 内存可用空间 500MB
// 测试链接 : https://www.luogu.com.cn/problem/P3242
// 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 Code04_Fruit1 {

    public static int MAXN = 40001;
    public static int MAXH = 16;
    public static int INF = 1000000001;
    public static int n, p, q;

    // 链式前向星
    public static int[] head = new int[MAXN];
    public static int[] next = new int[MAXN << 1];
    public static int[] to = new int[MAXN << 1];
    public static int cntg = 0;

    // 树上倍增 + 每棵子树上的 ldfn 和 rdfn
    public static int[] dep = new int[MAXN];
    public static int[] ldfn = new int[MAXN];
    public static int[] rdfn = new int[MAXN];
    public static int[][] stjump = new int[MAXN][MAXH];
    public static int cntd = 0;
```

```
// 只有 y 维度的树状数组
public static int[] tree = new int[MAXN];

// 所有事件排完序之后，依次把下标放入 eid 数组
public static int[] eid = new int[MAXN << 3];
// 每个事件有 8 个属性值
// op==1 加盘子，x 处加、yl、yr，盘子权值 v、空缺、空缺、空缺
// op==2 删盘子，x 处删、yl、yr，盘子权值 v、空缺、空缺、空缺
// op==3 为水果，x、空缺、空缺、空缺、y、要求 k、问题编号 i
public static int[][] event = new int[MAXN << 3][8];
// 事件的总数
public static int cnte = 0;

// 整体二分
public static int[] lset = new int[MAXN << 3];
public static int[] rset = new int[MAXN << 3];

// 每个水果的答案
public static int[] ans = new int[MAXN];

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

// 递归版，C++可以通过，java 会爆栈
public static void dfs1(int u, int fa) {
    dep[u] = dep[fa] + 1;
    ldfn[u] = ++cntd;
    stjump[u][0] = fa;
    for (int p = 1; p < MAXH; p++) {
        stjump[u][p] = stjump[stjump[u][p - 1]][p - 1];
    }
    for (int e = head[u]; e > 0; e = next[e]) {
        if (to[e] != fa) {
            dfs1(to[e], u);
        }
    }
    rdfn[u] = cntd;
}

// 不会改迭代版，去看讲解 118，详解了从递归版改迭代版
```

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

public static int stacksize, u, f, e;

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

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

// dfs1 的迭代版
public static void dfs2() {
    stacksize = 0;
    push(1, 0, -1);
    while (stacksize > 0) {
        pop();
        if (e == -1) {
            dep[u] = dep[f] + 1;
            ldfn[u] = ++cntd;
            stjump[u][0] = f;
            for (int p = 1; p < MAXH; p++) {
                stjump[u][p] = stjump[stjump[u][p - 1]][p - 1];
            }
            e = head[u];
        } else {
            e = next[e];
        }
        if (e != 0) {
            push(u, f, e);
            if (to[e] != f) {
                push(to[e], u, -1);
            }
        } else {
            rdfn[u] = cntd;
        }
```

```
    }
}

public static int lca(int a, int b) {
    if (dep[a] < dep[b]) {
        int tmp = a;
        a = b;
        b = tmp;
    }
    for (int p = MAXH - 1; p >= 0; p--) {
        if (dep[stjump[a][p]] >= dep[b]) {
            a = stjump[a][p];
        }
    }
    if (a == b) {
        return a;
    }
    for (int p = MAXH - 1; p >= 0; p--) {
        if (stjump[a][p] != stjump[b][p]) {
            a = stjump[a][p];
            b = stjump[b][p];
        }
    }
    return stjump[a][0];
}

// 已知a和b的最低公共祖先一定是a或b
// 假设祖先为x，后代为y，返回x的哪个儿子的子树里有y
public static int lcaSon(int a, int b) {
    if (dep[a] < dep[b]) {
        int tmp = a;
        a = b;
        b = tmp;
    }
    for (int p = MAXH - 1; p >= 0; p--) {
        if (dep[stjump[a][p]] > dep[b]) {
            a = stjump[a][p];
        }
    }
    return a;
}

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

```
    return i & -i;
}

public static void add(int i, int v) {
    while (i <= n) {
        tree[i] += v;
        i += lowbit(i);
    }
}

// 树状数组中[l..r]范围上的每个数增加 v
public static void add(int l, int r, int v) {
    add(l, v);
    add(r + 1, -v);
}

// 树状数组中查询单点的值
public static int query(int i) {
    int ret = 0;
    while (i > 0) {
        ret += tree[i];
        i -= lowbit(i);
    }
    return ret;
}

public static void addPlate(int x, int yl, int yr, int v) {
    event[++cnte][0] = 1;
    event[cnte][1] = x;
    event[cnte][2] = yl;
    event[cnte][3] = yr;
    event[cnte][4] = v;
}

public static void delPlate(int x, int yl, int yr, int v) {
    event[++cnte][0] = 2;
    event[cnte][1] = x;
    event[cnte][2] = yl;
    event[cnte][3] = yr;
    event[cnte][4] = v;
}

public static void addFruit(int x, int y, int k, int i) {
```

```
        event[++cnte][0] = 3;
        event[cnte][1] = x;
        // 2、3、4 位空缺
        event[cnte][5] = y;
        event[cnte][6] = k;
        event[cnte][7] = i;
    }

    public static void compute(int el, int er, int vl, int vr) {
        if (el > er) {
            return;
        }
        if (vl == vr) {
            for (int i = el; i <= er; i++) {
                int id = eid[i];
                if (event[id][0] == 3) {
                    ans[event[id][7]] = vl;
                }
            }
        } else {
            int mid = (vl + vr) >> 1;
            int lsiz = 0, rsiz = 0;
            for (int i = el; i <= er; i++) {
                int id = eid[i];
                if (event[id][0] == 1) {
                    if (event[id][4] <= mid) {
                        add(event[id][2], event[id][3], 1);
                        lset[++lsiz] = id;
                    } else {
                        rset[++rsiz] = id;
                    }
                } else if (event[id][0] == 2) {
                    if (event[id][4] <= mid) {
                        add(event[id][2], event[id][3], -1);
                        lset[++lsiz] = id;
                    } else {
                        rset[++rsiz] = id;
                    }
                } else {
                    int satisfy = query(event[id][5]);
                    if (satisfy >= event[id][6]) {
                        lset[++lsiz] = id;
                    } else {
```

```
                    event[id][6] -= satisfy;
                    rset[++rsiz] = id;
                }
            }
        }
        // 这里为什么不用做撤销？
        // 因为任何一个盘子，一定有两条扫描线
        // 一条扫描线会增加 yl..yr 的计数
        // 一条扫描线会减少 yl..yr 的计数
        // 同一个盘子的两条扫描线，一定会在一起，是不可能分开的
        // 所以此时树状数组就是清空的，不需要再做撤销操作
        for (int i = 1; i <= lsiz; i++) {
            eid[el + i - 1] = lset[i];
        }
        for (int i = 1; i <= rsiz; i++) {
            eid[el + lsiz + i - 1] = rset[i];
        }
        compute(el, el + lsiz - 1, vl, mid);
        compute(el + lsiz, er, mid + 1, vr);
    }
}

public static void main(String[] args) throws Exception {
    FastReader in = new FastReader(System.in);
    PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));
    n = in.nextInt();
    p = in.nextInt();
    q = in.nextInt();
    for (int i = 1, u, v; i < n; i++) {
        u = in.nextInt();
        v = in.nextInt();
        addEdge(u, v);
        addEdge(v, u);
    }
    dfs2();
    for (int i = 1; i <= p; i++) {
        int a = in.nextInt();
        int b = in.nextInt();
        int c = in.nextInt();
        if (ldfn[a] > ldfn[b]) {
            int tmp = a;
            a = b;
            b = tmp;
```

```
        }
        int ablca = lca(a, b);
        // 类型 1，a 和 b 的 lca 是 a 或 b，2 个区域，4 个事件产生
        // 类型 2，a 和 b 的 lca 不是 a 或 b，1 个区域，2 个事件产生
        if (ablca == a || ablca == b) {
            int son = lcaSon(a, b);
            // (1 ~ dfn[son]-1) (b 子树上的 dfn 范围)
            addPlate(1, ldfn[b], rdfn[b], c);
            delPlate(ldfn[son], ldfn[b], rdfn[b], c);
            // (b 子树上的 dfn 范围) (son 子树上最大的 dfn 序号+1 ~ n)
            addPlate(ldfn[b], rdfn[son] + 1, n, c);
            delPlate(rdfn[b] + 1, rdfn[son] + 1, n, c);
        } else {
            // (a 子树上的 dfn 范围) (b 子树上的 dfn 范围)
            addPlate(ldfn[a], ldfn[b], rdfn[b], c);
            delPlate(rdfn[a] + 1, ldfn[b], rdfn[b], c);
        }
    }
    for (int i = 1; i <= q; i++) {
        int u = in.nextInt();
        int v = in.nextInt();
        int k = in.nextInt();
        addFruit(Math.min(ldfn[u], ldfn[v]), Math.max(ldfn[u], ldfn[v]), k, i);
    }
    // 根据 x 排序，如果 x 一样，加盘子排最前、删盘子其次、水果最后
    Arrays.sort(event, 1, cnte + 1, (a, b) -> a[1] != b[1] ? a[1] - b[1] : a[0] - b[0]);
    for (int i = 1; i <= cnte; i++) {
        eid[i] = i;
    }
    compute(1, cnte, 0, INF);
    for (int i = 1; i <= q; i++) {
        out.println(ans[i]);
    }
    out.flush();
    out.close();
}

// 读写工具类
static class FastReader {
    private final byte[] buffer = new byte[1 << 16];
    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: Code04_Fruit2.java

==========

```
package class169;

// 接水果，C++版
```

```
// 一共有 n 个点，给定 n-1 条无向边，所有点连成一棵树
// 一共有 p 个盘子，每个盘子格式 a b c : 盘子是点 a 到点 b 的路径，盘子权值为 c
// 一共有 q 个水果，每个水果格式 u v k : 水果是点 u 到点 v 的路径，k 含义如下
// 如果一个盘子路径完全在一个水果路径的内部，那么该盘子可以接住该水果
// 那么对于每个水果，可能有很多盘子都可以将其接住，打印其中第 k 小的权值
// 1 <= n、p、q <= 4 * 10^4
// 0 <= 盘子权值 <= 10^9
// 内存可用空间 500MB
// 测试链接 : https://www.luogu.com.cn/problem/P3242
// 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 op, x, yl, yr, v, y, k, i;
//};
//
//bool EventCmp(Event e1, Event e2) {
//    if (e1.x != e2.x) {
//        return e1.x < e2.x;
//    }
//    return e1.op < e2.op;
//}
//
//const int MAXN = 40001;
//const int MAXH = 16;
//const int INF = 1000000001;
//int n, p, q;
//
//int head[MAXN];
//int nxt[MAXN << 1];
//int to[MAXN << 1];
//int cntg = 0;
//
//int dep[MAXN];
//int ldfn[MAXN];
//int rdfn[MAXN];
//int stjump[MAXN][MAXH];
//int cntd = 0;
//
```

```
//int tree[MAXN];
//
//int eid[MAXN << 3];
//Event event[MAXN << 3];
//int cnte = 0;
//
//int lset[MAXN << 3];
//int rset[MAXN << 3];
//
//int ans[MAXN];
//
//void addEdge(int u, int v) {
//    nxt[++cntg] = head[u];
//    to[cntg] = v;
//    head[u] = cntg;
//}
//
//void dfs(int u, int fa) {
//    dep[u] = dep[fa] + 1;
//    ldfn[u] = ++cntd;
//    stjump[u][0] = fa;
//    for (int p = 1; p < MAXH; p++) {
//        stjump[u][p] = stjump[stjump[u][p - 1]][p - 1];
//    }
//    for (int e = head[u]; e > 0; e = nxt[e]) {
//        if (to[e] != fa) {
//            dfs(to[e], u);
//        }
//    }
//    rdfn[u] = cntd;
//}
//
//int lca(int a, int b) {
//    if (dep[a] < dep[b]) {
//        int tmp = a;
//        a = b;
//        b = tmp;
//    }
//    for (int p = MAXH - 1; p >= 0; p--) {
//        if (dep[stjump[a][p]] >= dep[b]) {
//            a = stjump[a][p];
//        }
//    }
```

```
//    if (a == b) {
//        return a;
//    }
//    for (int p = MAXH - 1; p >= 0; p--) {
//        if (stjump[a][p] != stjump[b][p]) {
//            a = stjump[a][p];
//            b = stjump[b][p];
//        }
//    }
//    return stjump[a][0];
//}
//
//int lcaSon(int a, int b) {
//    if (dep[a] < dep[b]) {
//        int tmp = a;
//        a = b;
//        b = tmp;
//    }
//    for (int p = MAXH - 1; p >= 0; p--) {
//        if (dep[stjump[a][p]] > dep[b]) {
//            a = stjump[a][p];
//        }
//    }
//    return a;
//}
//
//int lowbit(int i) {
//    return i & -i;
//}
//
//void add(int i, int v) {
//    while (i <= n) {
//        tree[i] += v;
//        i += lowbit(i);
//    }
//}
//
//void add(int l, int r, int v) {
//    add(l, v);
//    add(r + 1, -v);
//}
//
//int query(int i) {
```

```
//    int ret = 0;
//    while (i > 0) {
//        ret += tree[i];
//        i -= lowbit(i);
//    }
//    return ret;
//}
//
//void addPlate(int x, int yl, int yr, int v) {
//    event[++cnte].op = 1;
//    event[cnte].x = x;
//    event[cnte].yl = yl;
//    event[cnte].yr = yr;
//    event[cnte].v = v;
//}
//
//void delPlate(int x, int yl, int yr, int v) {
//    event[++cnte].op = 2;
//    event[cnte].x = x;
//    event[cnte].yl = yl;
//    event[cnte].yr = yr;
//    event[cnte].v = v;
//}
//
//void addFruit(int x, int y, int k, int i) {
//    event[++cnte].op = 3;
//    event[cnte].x = x;
//    event[cnte].y = y;
//    event[cnte].k = k;
//    event[cnte].i = i;
//}
//
//void compute(int el, int er, int vl, int vr) {
//    if (el > er) {
//        return;
//    }
//    if (vl == vr) {
//        for (int i = el; i <= er; i++) {
//            int id = eid[i];
//            if (event[id].op == 3) {
//                ans[event[id].i] = vl;
//            }
//        }
```

```
//    } else {
//        int mid = (vl + vr) >> 1;
//        int lsiz = 0, rsiz = 0;
//        for (int i = el; i <= er; i++) {
//            int id = eid[i];
//            if (event[id].op == 1) {
//                if (event[id].v <= mid) {
//                    add(event[id].yl, event[id].yr, 1);
//                    lset[++lsiz] = id;
//                } else {
//                    rset[++rsiz] = id;
//                }
//            } else if (event[id].op == 2) {
//                if (event[id].v <= mid) {
//                    add(event[id].yl, event[id].yr, -1);
//                    lset[++lsiz] = id;
//                } else {
//                    rset[++rsiz] = id;
//                }
//            } else {
//                int satisfy = query(event[id].y);
//                if (satisfy >= event[id].k) {
//                    lset[++lsiz] = id;
//                } else {
//                    event[id].k -= satisfy;
//                    rset[++rsiz] = id;
//                }
//            }
//        }
//        for (int i = 1; i <= lsiz; i++) {
//            eid[el + i - 1] = lset[i];
//        }
//        for (int i = 1; i <= rsiz; i++) {
//            eid[el + lsiz + i - 1] = rset[i];
//        }
//        compute(el, el + lsiz - 1, vl, mid);
//        compute(el + lsiz, er, mid + 1, vr);
//    }
//}
//
//int main() {
//    ios::sync_with_stdio(false);
//    cin.tie(nullptr);
```

```
//    cin >> n >> p >> q;
//    for (int i = 1, u, v; i < n; i++) {
//        cin >> u >> v;
//        addEdge(u, v);
//        addEdge(v, u);
//    }
//    dfs(1, 0);
//    for (int i = 1, a, b, c; i <= p; i++) {
//        cin >> a >> b >> c;
//        if (ldfn[a] > ldfn[b]) {
//            int tmp = a;
//            a = b;
//            b = tmp;
//        }
//        int ablca = lca(a, b);
//        if (ablca == a || ablca == b) {
//            int son = lcaSon(a, b);
//            addPlate(1, ldfn[b], rdfn[b], c);
//            delPlate(ldfn[son], ldfn[b], rdfn[b], c);
//            addPlate(ldfn[b], rdfn[son] + 1, n, c);
//            delPlate(rdfn[b] + 1, rdfn[son] + 1, n, c);
//        } else {
//            addPlate(ldfn[a], ldfn[b], rdfn[b], c);
//            delPlate(rdfn[a] + 1, ldfn[b], rdfn[b], c);
//        }
//    }
//    for (int i = 1, u, v, k; i <= q; i++) {
//        cin >> u >> v >> k;
//        addFruit(min(ldfn[u], ldfn[v]), max(ldfn[u], ldfn[v]), k, i);
//    }
//    sort(event + 1, event + cnte + 1, EventCmp);
//    for (int i = 1; i <= cnte; i++) {
//        eid[i] = i;
//    }
//    compute(1, cnte, 0, INF);
//    for (int i = 1; i <= q; i++) {
//        cout << ans[i] << '\n';
//    }
//    return 0;
//}
```

==========

文件: Code05_PastoralOddities1.java

```
============================================
package class169;

// 点的度都是奇数的最小瓶颈，java 版
// 一共有 n 个点，初始没有边，依次加入 m 条无向边，每条边有边权
// 每次加入后，询问是否存在一个边集，满足每个点连接的边的数量都是奇数
// 如果存在，希望边集的最大边权，尽可能小，如果不存在打印-1
// 2 <= n <= 10^5
// 1 <= m <= 3 * 10^5
// 1 <= 边权 <= 10^9
// 测试链接 : https://www.luogu.com.cn/problem/CF603E
// 测试链接 : https://codeforces.com/problemset/problem/603/E
// 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 Code05_PastoralOddities1 {

    public static int MAXN = 100001;
    public static int MAXM = 300001;
    public static int n, m;

    // edge 代表所有边依次出现
    // wsort 代表所有边按边权排序
    // 每条边有：端点 x、端点 y、边权 w、时序 tim、边权排名 rak
    public static int[][] edge = new int[MAXM][5];
    public static int[][] wsort = new int[MAXM][5];

    // 可撤销并查集 + 节点数为奇数的连通区数量为 oddnum
    public static int oddnum;
    public static int[] father = new int[MAXN];
    public static int[] siz = new int[MAXN];
    public static int[][] rollback = new int[MAXN][2];
    public static int opsize = 0;

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

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

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

public static boolean union(int x, int y) {
    int fx = find(x);
    int fy = find(y);
    if (fx == fy) {
        return false;
    }
    if ((siz[fx] & 1) == 1 && (siz[fy] & 1) == 1) {
        oddnum -= 2;
    }
    if (siz[fx] < siz[fy]) {
        int tmp = fx;
        fx = fy;
        fy = tmp;
    }
    father[fy] = fx;
    siz[fx] += siz[fy];
    rollback[++opsize][0] = fx;
    rollback[opsize][1] = fy;
    return true;
}

public static void undo() {
    int fx = rollback[opsize][0];
    int fy = rollback[opsize--][1];
    father[fy] = fy;
    siz[fx] -= siz[fy];
    if ((siz[fx] & 1) == 1 && (siz[fy] & 1) == 1) {
        oddnum += 2;
    }
}

// 依次出现的边在 edge 里，当前来到[el..er]范围
// 权值排序后的边在 wsort 里，答案范围[vl..vr]，同时也是排名范围
// 调用递归的前提 : el 之前，边权排名<vl 的边，已经加到图里了
// 利用整体二分得到所有边的答案
public static void compute(int el, int er, int vl, int vr) {
    if (el > er) {
```

```
    return;
}
if (vl == vr) {
    for (int i = el; i <= er; i++) {
        ans[i] = vl;
    }
} else {
    int mid = (vl + vr) >> 1;
    // 1) el 之前，边权排名在[vl..mid]之间的边，加到图里，通过遍历 wsort[vl..mid]来加速
    int unionCnt1 = 0;
    for (int i = vl; i <= mid; i++) {
        if (wsort[i][3] < el) {
            if (union(wsort[i][0], wsort[i][1])) {
                unionCnt1++;
            }
        }
    }
    // 2) 从 el 开始遍历，边权排名<=mid 的边，加到图里，找到第一个达标的边 split
    int unionCnt2 = 0;
    int split = er + 1;
    for (int i = el; i <= er; i++) {
        if (edge[i][4] <= mid) {
            if (union(edge[i][0], edge[i][1])) {
                unionCnt2++;
            }
        }
        if (oddnum == 0) {
            split = i;
            break;
        }
    }
    // 3) 撤销 2)的效果，el 之前，边权排名<=mid 的边，都在图中
    for (int i = 1; i <= unionCnt2; i++) {
        undo();
    }
    // 4) 执行 compute(el, split - 1, mid + 1, vr)，此时满足子递归的前提
    compute(el, split - 1, mid + 1, vr);
    // 5) 撤销 1)的效果，此时只剩下前提了，el 之前，边权排名<vl 的边，都在图中
    for (int i = 1; i <= unionCnt1; i++) {
        undo();
    }
    // 6) 从 el 开始到 split 之前，边权排名<vl 的边，加到图里
    int unionCnt3 = 0;
```

```
        for (int i = el; i <= split - 1; i++) {
            if (edge[i][4] < vl) {
                if (union(edge[i][0], edge[i][1])) {
                    unionCnt3++;
                }
            }
        }
        // 7) 执行 compute(split, er, vl, mid)，此时满足子递归的前提
        compute(split, er, vl, mid);
        // 8) 撤销 6)的效果，回到了前提
        for (int i = 1; i <= unionCnt3; i++) {
            undo();
        }
    }
}

public static void prepare() {
    oddnum = n;
    for (int i = 1; i <= n; i++) {
        father[i] = i;
        siz[i] = 1;
    }
    for (int i = 1; i <= m; i++) {
        wsort[i][0] = edge[i][0];
        wsort[i][1] = edge[i][1];
        wsort[i][2] = edge[i][2];
        wsort[i][3] = edge[i][3];
    }
    Arrays.sort(wsort, 1, m + 1, (a, b) -> a[2] - b[2]);
    // edge 数组、wsort 数组，每条边设置排名信息
    for (int i = 1; i <= m; i++) {
        wsort[i][4] = i;
        edge[wsort[i][3]][4] = i;
    }
}

public static void main(String[] args) throws Exception {
    FastReader in = new FastReader(System.in);
    PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));
    n = in.nextInt();
    m = in.nextInt();
    for (int i = 1; i <= m; i++) {
        edge[i][0] = in.nextInt();
```

```
            edge[i][1] = in.nextInt();
            edge[i][2] = in.nextInt();
            edge[i][3] = i;
        }
        prepare();
        compute(1, m, 1, m + 1);
        for (int i = 1; i <= m; i++) {
            if (ans[i] == m + 1) {
                out.println(-1);
            } else {
                out.println(wsort[ans[i]][2]);
            }
        }
        out.flush();
        out.close();
    }

    // 读写工具类
    static class FastReader {
        private final byte[] buffer = new byte[1 << 16];
        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: Code05_Pastoral0ddities2.java

==========

```
package class169;

// 点的度都是奇数的最小瓶颈，C++版
// 一共有 n 个点，初始没有边，依次加入 m 条无向边，每条边有边权
// 每次加入后，询问是否存在一个边集，满足每个点连接的边的数量都是奇数
// 如果存在，希望边集的最大边权，尽可能小，如果不存在打印-1
// 2 <= n <= 10^5
// 1 <= m <= 3 * 10^5
// 1 <= 边权 <= 10^9
// 测试链接 : https://www.luogu.com.cn/problem/CF603E
// 测试链接 : https://codeforces.com/problemset/problem/603/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;
//
//struct Edge {
//    int x, y, w, tim, rak;
//};
//
//bool EdgeCmp(Edge a, Edge b) {
//    return a.w < b.w;
```

```
//}
//
//const int MAXN = 100001;
//const int MAXM = 300001;
//int n, m;
//
//Edge edge[MAXM];
//Edge wsort[MAXM];
//
//int oddnum;
//int father[MAXN];
//int siz[MAXN];
//int rollback[MAXN][2];
//int opsize = 0;
//
//int ans[MAXM];
//
//int find(int i) {
//    while (i != father[i]) {
//        i = father[i];
//    }
//    return i;
//}
//
//bool Union(int x, int y) {
//    int fx = find(x);
//    int fy = find(y);
//    if (fx == fy) {
//        return false;
//    }
//    if ((siz[fx] & 1) == 1 && (siz[fy] & 1) == 1) {
//        oddnum -= 2;
//    }
//    if (siz[fx] < siz[fy]) {
//        int tmp = fx;
//        fx = fy;
//        fy = tmp;
//    }
//    father[fy] = fx;
//    siz[fx] += siz[fy];
//    rollback[++opsize][0] = fx;
//    rollback[opsize][1] = fy;
//    return true;
```

```
//}
//
//void undo() {
//    int fx = rollback[opsize][0];
//    int fy = rollback[opsize--][1];
//    father[fy] = fy;
//    siz[fx] -= siz[fy];
//    if ((siz[fx] & 1) == 1 && (siz[fy] & 1) == 1) {
//        oddnum += 2;
//    }
//}
//
//void compute(int el, int er, int vl, int vr) {
//    if (el > er) {
//        return;
//    }
//    if (vl == vr) {
//        for (int i = el; i <= er; i++) {
//            ans[i] = vl;
//        }
//    } else {
//        int mid = (vl + vr) >> 1;
//        int unionCnt1 = 0;
//        for (int i = vl; i <= mid; i++) {
//            if (wsort[i].tim < el) {
//                if (Union(wsort[i].x, wsort[i].y)) {
//                    unionCnt1++;
//                }
//            }
//        }
//        int unionCnt2 = 0;
//        int split = er + 1;
//        for (int i = el; i <= er; i++) {
//            if (edge[i].rak <= mid) {
//                if (Union(edge[i].x, edge[i].y)) {
//                    unionCnt2++;
//                }
//            }
//            if (oddnum == 0) {
//                split = i;
//                break;
//            }
//        }
```

```
//        for (int i = 1; i <= unionCnt2; i++) {
//            undo();
//        }
//        compute(el, split - 1, mid + 1, vr);
//        for (int i = 1; i <= unionCnt1; i++) {
//            undo();
//        }
//        int unionCnt3 = 0;
//        for (int i = el; i <= split - 1; i++) {
//            if (edge[i].rak < vl) {
//                if (Union(edge[i].x, edge[i].y)) {
//                    unionCnt3++;
//                }
//            }
//        }
//        compute(split, er, vl, mid);
//        for (int i = 1; i <= unionCnt3; i++) {
//            undo();
//        }
//    }
//}
//
//void prepare() {
//    oddnum = n;
//    for (int i = 1; i <= n; i++) {
//        father[i] = i;
//        siz[i] = 1;
//    }
//    for (int i = 1; i <= m; i++) {
//        wsort[i].x = edge[i].x;
//        wsort[i].y = edge[i].y;
//        wsort[i].w = edge[i].w;
//        wsort[i].tim = edge[i].tim;
//    }
//    sort(wsort + 1, wsort + m + 1, EdgeCmp);
//    for (int i = 1; i <= m; i++) {
//        wsort[i].rak = i;
//        edge[wsort[i].tim].rak = i;
//    }
//}
//
//int main() {
//    ios::sync_with_stdio(false);
```

```
//    cin.tie(nullptr);
//    cin >> n >> m;
//    for (int i = 1; i <= m; i++) {
//        cin >> edge[i].x >> edge[i].y >> edge[i].w;
//        edge[i].tim = i;
//    }
//    prepare();
//    compute(1, m, 1, m + 1);
//    for (int i = 1; i <= m; i++) {
//        if (ans[i] == m + 1) {
//            cout << -1 << '\n';
//        } else {
//            cout << wsort[ans[i]].w << '\n';
//        }
//    }
//    return 0;
//}
```

==========

文件: Code06_IvanAndBurgers1.java

==========

```
package class169;

// 范围最大异或和，java 版
// 给定一个长度为 n 的数组 arr，下标 1~n，接下来有 q 条查询，格式如下
// 查询 l r : arr[l..r]中选若干个数，打印最大的异或和
// 1 <= n、q <= 5 * 10^5
// 0 <= arr[i] <= 10^6
// 测试链接 : https://www.luogu.com.cn/problem/CF1100F
// 测试链接 : https://codeforces.com/problemset/problem/1100/F
// 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 Code06_IvanAndBurgers1 {

    public static int MAXN = 500001;
    public static int BIT = 21;
    public static int n, q;
```

```
public static int[] arr = new int[MAXN];
public static int[] qid = new int[MAXN];
public static int[] l = new int[MAXN];
public static int[] r = new int[MAXN];

public static int[][] baset = new int[MAXN][BIT + 1];
public static int[] tmp = new int[BIT + 1];

public static int[] lset = new int[MAXN];
public static int[] rset = new int[MAXN];

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

public static void insert(int[] basis, int num) {
    for (int i = BIT; i >= 0; i--) {
        if (num >> i == 1) {
            if (basis[i] == 0) {
                basis[i] = num;
                return;
            }
            num ^= basis[i];
        }
    }
}

public static void clear(int[] basis) {
    for (int i = 0; i <= BIT; i++) {
        basis[i] = 0;
    }
}

public static int maxEor(int[] basis) {
    int ret = 0;
    for (int i = BIT; i >= 0; i--) {
        ret = Math.max(ret, ret ^ basis[i]);
    }
    return ret;
}

public static void clone(int[] b1, int[] b2) {
    for (int i = 0; i <= BIT; i++) {
        b1[i] = b2[i];
```

```
    }
}

public static void merge(int[] b1, int[] b2) {
    clone(tmp, b1);
    for (int i = 0; i <= BIT; i++) {
        insert(tmp, b2[i]);
    }
}

public static void compute(int ql, int qr, int vl, int vr) {
    if (ql > qr) {
        return;
    }
    if (vl == vr) {
        for (int i = ql; i <= qr; i++) {
            ans[qid[i]] = arr[vl];
        }
    } else {
        int mid = (vl + vr) >> 1;
        clear(baset[mid]);
        insert(baset[mid], arr[mid]);
        for (int i = mid - 1; i >= vl; i--) {
            clone(baset[i], baset[i + 1]);
            insert(baset[i], arr[i]);
        }
        for (int i = mid + 1; i <= vr; i++) {
            clone(baset[i], baset[i - 1]);
            insert(baset[i], arr[i]);
        }
        int lsiz = 0, rsiz = 0, id;
        for (int i = ql; i <= qr; i++) {
            id = qid[i];
            if (r[id] < mid) {
                lset[++lsiz] = id;
            } else if (l[id] > mid) {
                rset[++rsiz] = id;
            } else {
                merge(baset[l[id]], baset[r[id]]);
                ans[id] = maxEor(tmp);
            }
        }
        for (int i = 1; i <= lsiz; i++) {
```

```
            qid[ql + i - 1] = lset[i];
        }
        for (int i = 1; i <= rsiz; i++) {
            qid[ql + lsiz + i - 1] = rset[i];
        }
        compute(ql, ql + lsiz - 1, vl, mid);
        compute(ql + lsiz, ql + lsiz + rsiz - 1, mid + 1, vr);
    }
}

public static void main(String[] args) throws Exception {
    FastReader in = new FastReader(System.in);
    PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));
    n = in.nextInt();
    for (int i = 1; i <= n; i++) {
        arr[i] = in.nextInt();
    }
    q = in.nextInt();
    for (int i = 1; i <= q; i++) {
        qid[i] = i;
        l[i] = in.nextInt();
        r[i] = in.nextInt();
    }
    compute(1, q, 1, n);
    for (int i = 1; i <= q; i++) {
        out.println(ans[i]);
    }
    out.flush();
    out.close();
}

// 读写工具类
static class FastReader {
    private final byte[] buffer = new byte[1 << 16];
    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: Code06_IvanAndBurgers2.java

==========

```
package class169;

// 范围最大异或和，C++版
// 给定一个长度为n的数组arr，下标1~n，接下来有q条查询，格式如下
// 查询 l r : arr[l..r]中选若干个数，打印最大的异或和
// 1 <= n、q <= 5 * 10^5
// 0 <= arr[i] <= 10^6
// 测试链接 : https://www.luogu.com.cn/problem/CF1100F
// 测试链接 : https://codeforces.com/problemset/problem/1100/F
// 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 = 500001;
//const int BIT = 21;
//int n, q;
//
//int arr[MAXN];
//int qid[MAXN];
//int l[MAXN];
//int r[MAXN];
//
//int baset[MAXN][BIT + 1];
//int tmp[BIT + 1];
//
//int lset[MAXN];
//int rset[MAXN];
//
//int ans[MAXN];
//
//void insert(int* basis, int num) {
//    for (int i = BIT; i >= 0; i--) {
//        if ((num >> i) & 1) {
//            if (basis[i] == 0) {
//                basis[i] = num;
//                return;
//            }
//            num ^= basis[i];
//        }
//    }
//}
//
//void clear(int* basis) {
//    for (int i = 0; i <= BIT; i++) {
//        basis[i] = 0;
//    }
//}
//
//int maxEor(int* basis) {
//    int ret = 0;
```

```
//    for (int i = BIT; i >= 0; i--) {
//        ret = max(ret, ret ^ basis[i]);
//    }
//    return ret;
//}
//
//void clone(int* b1, int* b2) {
//    for (int i = 0; i <= BIT; i++) {
//        b1[i] = b2[i];
//    }
//}
//
//void merge(int* b1, int* b2) {
//    clone(tmp, b1);
//    for (int i = 0; i <= BIT; i++) {
//        insert(tmp, b2[i]);
//    }
//}
//
//void compute(int ql, int qr, int vl, int vr) {
//    if (ql > qr) {
//        return;
//    }
//    if (vl == vr) {
//        for (int i = ql; i <= qr; i++) {
//            ans[qid[i]] = arr[vl];
//        }
//    } else {
//        int mid = (vl + vr) >> 1;
//        clear(baset[mid]);
//        insert(baset[mid], arr[mid]);
//        for (int i = mid - 1; i >= vl; i--) {
//            clone(baset[i], baset[i + 1]);
//            insert(baset[i], arr[i]);
//        }
//        for (int i = mid + 1; i <= vr; i++) {
//            clone(baset[i], baset[i - 1]);
//            insert(baset[i], arr[i]);
//        }
//        int lsiz = 0, rsiz = 0;
//        for (int i = ql, id; i <= qr; i++) {
//            id = qid[i];
//            if (r[id] < mid) {
```

```
//                lset[++lsiz] = id;
//            } else if (l[id] > mid) {
//                rset[++rsiz] = id;
//            } else {
//                merge(baset[l[id]], baset[r[id]]);
//                ans[id] = maxEor(tmp);
//            }
//        }
//        for (int i = 1; i <= lsiz; i++) {
//            qid[ql + i - 1] = lset[i];
//        }
//        for (int i = 1; i <= rsiz; i++) {
//            qid[ql + lsiz + i - 1] = rset[i];
//        }
//        compute(ql, ql + lsiz - 1, vl, mid);
//        compute(ql + lsiz, ql + lsiz + rsiz - 1, mid + 1, vr);
//    }
//}
//
//int main() {
//    ios::sync_with_stdio(false);
//    cin.tie(nullptr);
//    cin >> n;
//    for (int i = 1; i <= n; i++) {
//        cin >> arr[i];
//    }
//    cin >> q;
//    for (int i = 1; i <= q; i++) {
//        qid[i] = i;
//        cin >> l[i] >> r[i];
//    }
//    compute(1, q, 1, n);
//    for (int i = 1; i <= q; i++) {
//        cout << ans[i] << '\n';
//    }
//    return 0;
//}
```

==========

文件: Code07_KthNumber1.cpp

==========

```
/*
```

```
 * POJ 2104 K-th Number - C++实现
 *
 * 题目来源：http://poj.org/problem?id=2104
 * 题目描述：静态区间第 k 小查询
 *
 * 问题描述：
 * 给定一个长度为 n 的数组，有 m 个查询，每个查询要求在指定区间内找到第 k 小的数。
 *
 * 解题思路：
 * 使用整体二分处理静态区间第 k 小问题。将所有查询一起处理，二分答案的值域，
 * 利用树状数组维护区间内小于等于 mid 的元素个数。
 *
 * 时间复杂度：O((N+Q) * logN * log(maxValue))
 * 空间复杂度：O(N + Q)
 */

const int MAXN = 100001;
int n, m;

// 原始数组
int arr[MAXN];

// 离散化数组
int sorted[MAXN];

// 查询信息
int queryL[MAXN];  // 查询区间左端点
int queryR[MAXN];  // 查询区间右端点
int queryK[MAXN];  // 查询第 k 小
int queryId[MAXN]; // 查询编号

// 树状数组
int tree[MAXN];

// 整体二分
int lset[MAXN]; // 左集合
int rset[MAXN]; // 右集合

// 查询的答案
int ans[MAXN];

// 树状数组操作
int lowbit(int i) {
```

```
    return i & -i;
}

void add(int i, int v) {
    while (i <= n) {
        tree[i] += v;
        i += lowbit(i);
    }
}

int sum(int i) {
    int ret = 0;
    while (i > 0) {
        ret += tree[i];
        i -= lowbit(i);
    }
    return ret;
}

int query(int l, int r) {
    return sum(r) - sum(l - 1);
}

/**
 * 整体二分核心函数
 * @param ql 查询范围的左端点
 * @param qr 查询范围的右端点
 * @param vl 值域范围的左端点（离散化后的下标）
 * @param vr 值域范围的右端点（离散化后的下标）
 */
void compute(int ql, int qr, int vl, int vr) {
    // 递归边界
    if (ql > qr) {
        return;
    }

    // 如果值域范围只有一个值，说明找到了答案
    if (vl == vr) {
        for (int i = ql; i <= qr; i++) {
            ans[queryId[i]] = sorted[vl];
        }
        return;
    }
```

```
// 二分中点
int mid = (vl + vr) >> 1;

// 将值小于等于 sorted[mid]的数加入树状数组
for (int i = vl; i <= mid; i++) {
    // 遍历所有值为 sorted[i]的元素，将其加入树状数组
    for (int j = 1; j <= n; j++) {
        if (arr[j] == sorted[i]) {
            add(j, 1);
        }
    }
}

// 检查每个查询，根据满足条件的元素个数划分到左右区间
int lsiz = 0, rsiz = 0;
for (int i = ql; i <= qr; i++) {
    // 查询区间[queryL[i], queryR[i]]中值小于等于 sorted[mid]的元素个数
    int satisfy = query(queryL[i], queryR[i]);

    if (satisfy >= queryK[i]) {
        // 说明第 k 小的数在左半部分
        lset[++lsiz] = i;
    } else {
        // 说明第 k 小的数在右半部分，需要在右半部分找第(k-satisfy)小的数
        queryK[i] -= satisfy;
        rset[++rsiz] = i;
    }
}

// 重新排列查询顺序
int idx = ql;
for (int i = 1; i <= lsiz; i++) {
    int temp = lset[i];
    lset[i] = queryId[temp];
    queryId[idx++] = temp;
}
for (int i = 1; i <= rsiz; i++) {
    int temp = rset[i];
    rset[i] = queryId[temp];
    queryId[idx++] = temp;
}
```

```
    // 撤销对树状数组的修改
    for (int i = vl; i <= mid; i++) {
        // 遍历所有值为sorted[i]的元素，将其从树状数组中删除
        for (int j = 1; j <= n; j++) {
            if (arr[j] == sorted[i]) {
                add(j, -1);
            }
        }
    }

    // 递归处理左右两部分
    compute(ql, ql + lsiz - 1, vl, mid);
    compute(ql + lsiz, qr, mid + 1, vr);
}

int main() {
    /*
     * 由于C++编译环境问题，这里使用简化输入输出
     * 实际使用时需要根据具体环境调整输入输出方式
     *
     * 示例代码结构：
     *
     * // 读取数组长度n和查询数量m
     * cin >> n >> m;
     *
     * // 读取原始数组
     * for (int i = 1; i <= n; i++) {
     *     cin >> arr[i];
     *     sorted[i] = arr[i];
     * }
     *
     * // 读取查询
     * for (int i = 1; i <= m; i++) {
     *     cin >> queryL[i] >> queryR[i] >> queryK[i];
     *     queryId[i] = i;
     * }
     *
     * // 离散化
     * sort(sorted + 1, sorted + n + 1);
     * int uniqueCount = 1;
     * for (int i = 2; i <= n; i++) {
     *     if (sorted[i] != sorted[i - 1]) {
     *         sorted[++uniqueCount] = sorted[i];
```

```
     *     }
     * }
     *
     * // 整体二分求解
     * compute(1, m, 1, uniqueCount);
     *
     * // 输出结果
     * for (int i = 1; i <= m; i++) {
     *     cout << ans[i] << endl;
     * }
     */

    return 0;
}

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

文件: Code07_KthNumber1.java
==========================================
package class169;

// POJ 2104 K-th Number - Java 实现
// 题目来源：http://poj.org/problem?id=2104
// 题目描述：静态区间第 k 小查询
// 时间复杂度：O((N+Q) * logN * log(maxValue))
// 空间复杂度：O(N + Q)

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

public class Code07_KthNumber1 {
    public static int MAXN = 100001;
    public static int n, m;

    // 原始数组
    public static int[] arr = new int[MAXN];

    // 离散化数组
    public static int[] sorted = new int[MAXN];

    // 查询信息
    public static int[] queryL = new int[MAXN];
    public static int[] queryR = new int[MAXN];
```

```
public static int[] queryK = new int[MAXN];
public static int[] queryId = new int[MAXN];

// 树状数组
public static int[] tree = new int[MAXN];

// 整体二分
public static int[] lset = new int[MAXN];
public static int[] rset = new int[MAXN];

// 查询的答案
public static int[] ans = new int[MAXN];

// 树状数组操作
public static int lowbit(int i) {
    return i & -i;
}

public static void add(int i, int v) {
    while (i <= n) {
        tree[i] += v;
        i += lowbit(i);
    }
}

public static int sum(int i) {
    int ret = 0;
    while (i > 0) {
        ret += tree[i];
        i -= lowbit(i);
    }
    return ret;
}

public static int query(int l, int r) {
    return sum(r) - sum(l - 1);
}

// 整体二分核心函数
// ql, qr: 查询范围
// vl, vr: 值域范围（离散化后的下标）
public static void compute(int ql, int qr, int vl, int vr) {
    // 递归边界
```

```
if (ql > qr) {
    return;
}

// 如果值域范围只有一个值，说明找到了答案
if (vl == vr) {
    for (int i = ql; i <= qr; i++) {
        ans[queryId[i]] = sorted[vl];
    }
    return;
}

// 二分中点
int mid = (vl + vr) >> 1;

// 将值小于等于 sorted[mid]的数加入树状数组
for (int i = vl; i <= mid; i++) {
    // 遍历所有值为 sorted[i]的元素，将其加入树状数组
    for (int j = 1; j <= n; j++) {
        if (arr[j] == sorted[i]) {
            add(j, 1);
        }
    }
}

// 检查每个查询，根据满足条件的元素个数划分到左右区间
int lsiz = 0, rsiz = 0;
for (int i = ql; i <= qr; i++) {
    // 查询区间[queryL[i], queryR[i]]中值小于等于 sorted[mid]的元素个数
    int satisfy = query(queryL[i], queryR[i]);

    if (satisfy >= queryK[i]) {
        // 说明第 k 小的数在左半部分
        lset[++lsiz] = i;
    } else {
        // 说明第 k 小的数在右半部分，需要在右半部分找第(k-satisfy)小的数
        queryK[i] -= satisfy;
        rset[++rsiz] = i;
    }
}

// 重新排列查询顺序
int idx = ql;
```

```
        for (int i = 1; i <= lsiz; i++) {
            int temp = lset[i];
            lset[i] = queryId[temp];
            queryId[idx++] = temp;
        }
        for (int i = 1; i <= rsiz; i++) {
            int temp = rset[i];
            rset[i] = queryId[temp];
            queryId[idx++] = temp;
        }

        // 撤销对树状数组的修改
        for (int i = vl; i <= mid; i++) {
            // 遍历所有值为sorted[i]的元素，将其从树状数组中删除
            for (int j = 1; j <= n; j++) {
                if (arr[j] == sorted[i]) {
                    add(j, -1);
                }
            }
        }

        // 递归处理左右两部分
        compute(ql, ql + lsiz - 1, vl, mid);
        compute(ql + lsiz, qr, mid + 1, vr);
    }

    public static void main(String[] args) throws IOException {
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));

        String[] params = br.readLine().split(" ");
        n = Integer.parseInt(params[0]);
        m = Integer.parseInt(params[1]);

        // 读取原始数组
        String[] nums = br.readLine().split(" ");
        for (int i = 1; i <= n; i++) {
            arr[i] = Integer.parseInt(nums[i - 1]);
            sorted[i] = arr[i];
        }

        // 读取查询
        for (int i = 1; i <= m; i++) {
```

```
            String[] query = br.readLine().split(" ");
            queryL[i] = Integer.parseInt(query[0]);
            queryR[i] = Integer.parseInt(query[1]);
            queryK[i] = Integer.parseInt(query[2]);
            queryId[i] = i;
        }

        // 离散化
        Arrays.sort(sorted, 1, n + 1);
        int uniqueCount = 1;
        for (int i = 2; i <= n; i++) {
            if (sorted[i] != sorted[i - 1]) {
                sorted[++uniqueCount] = sorted[i];
            }
        }

        // 整体二分求解
        compute(1, m, 1, uniqueCount);

        // 输出结果
        for (int i = 1; i <= m; i++) {
            out.println(ans[i]);
        }

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

==========

文件: Code07_KthNumber1.py

==========

```
"""
POJ 2104 K-th Number - Python 实现

题目来源：http://poj.org/problem?id=2104
题目描述：静态区间第 k 小查询

问题描述：
给定一个长度为 n 的数组，有 m 个查询，每个查询要求在指定区间内找到第 k 小的数。
```

解题思路：
使用整体二分处理静态区间第 k 小问题。将所有查询一起处理，二分答案的值域，
利用树状数组维护区间内小于等于 mid 的元素个数。

时间复杂度：O((N+Q) * logN * log(maxValue))
空间复杂度：O(N + Q)
"""

```
class KthNumberSolution:
    """
    静态区间第 k 小查询解决方案类

    使用整体二分算法解决 POJ 2104 K-th Number 问题
    """

    def __init__(self):
        """
        初始化解决方案类的成员变量
        """
        self.MAXN = 100001
        self.n = 0
        self.m = 0

        # 原始数组
        self.arr = [0] * self.MAXN

        # 离散化数组
        self.sorted = [0] * self.MAXN

        # 查询信息
        self.queryL = [0] * self.MAXN
        self.queryR = [0] * self.MAXN
        self.queryK = [0] * self.MAXN
        self.queryId = [0] * self.MAXN

        # 树状数组
        self.tree = [0] * self.MAXN

        # 整体二分
        self.lset = [0] * self.MAXN
        self.rset = [0] * self.MAXN

        # 查询的答案
```

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

# 树状数组操作
def lowbit(self, i):
    """
    计算一个数的 lowbit 值
    :param i: 输入的数
    :return: lowbit 值
    """
    return i & -i

def add(self, i, v):
    """
    在树状数组中给位置 i 增加 v
    :param i: 位置
    :param v: 增加的值
    """
    while i <= self.n:
        self.tree[i] += v
        i += self.lowbit(i)

def sum(self, i):
    """
    计算前缀和[1..i]
    :param i: 位置
    :return: 前缀和
    """
    ret = 0
    while i > 0:
        ret += self.tree[i]
        i -= self.lowbit(i)
    return ret

def query(self, l, r):
    """
    计算区间和[l..r]
    :param l: 左端点
    :param r: 右端点
    :return: 区间和
    """
    return self.sum(r) - self.sum(l - 1)

def compute(self, ql, qr, vl, vr):
```

```
    """
    整体二分核心函数
    :param ql: 查询范围的左端点
    :param qr: 查询范围的右端点
    :param vl: 值域范围的左端点（离散化后的下标）
    :param vr: 值域范围的右端点（离散化后的下标）
    """
    # 递归边界
    if ql > qr:
        return

    # 如果值域范围只有一个值，说明找到了答案
    if vl == vr:
        for i in range(ql, qr + 1):
            self.ans[self.queryId[i]] = self.sorted[vl]
        return

    # 二分中点
    mid = (vl + vr) >> 1

    # 将值小于等于 sorted[mid]的数加入树状数组
    for i in range(vl, mid + 1):
        # 遍历所有值为 sorted[i]的元素，将其加入树状数组
        for j in range(1, self.n + 1):
            if self.arr[j] == self.sorted[i]:
                self.add(j, 1)

    # 检查每个查询，根据满足条件的元素个数划分到左右区间
    lsiz = 0
    rsiz = 0
    for i in range(ql, qr + 1):
        # 查询区间[queryL[i], queryR[i]]中值小于等于 sorted[mid]的元素个数
        satisfy = self.query(self.queryL[i], self.queryR[i])

        if satisfy >= self.queryK[i]:
            # 说明第 k 小的数在左半部分
            lsiz += 1
            self.lset[lsiz] = i
        else:
            # 说明第 k 小的数在右半部分，需要在右半部分找第(k-satisfy)小的数
            self.queryK[i] -= satisfy
            rsiz += 1
            self.rset[rsiz] = i
```

```
        # 重新排列查询顺序
        idx = ql
        for i in range(1, lsiz + 1):
            temp = self.lset[i]
            self.lset[i] = self.queryId[temp]
            self.queryId[idx] = temp
            idx += 1
        for i in range(1, rsiz + 1):
            temp = self.rset[i]
            self.rset[i] = self.queryId[temp]
            self.queryId[idx] = temp
            idx += 1

        # 撤销对树状数组的修改
        for i in range(vl, mid + 1):
            # 遍历所有值为sorted[i]的元素，将其从树状数组中删除
            for j in range(1, self.n + 1):
                if self.arr[j] == self.sorted[i]:
                    self.add(j, -1)

        # 递归处理左右两部分
        self.compute(ql, ql + lsiz - 1, vl, mid)
        self.compute(ql + lsiz, qr, mid + 1, vr)

    def solve(self):
        """
        解决POJ 2104 K-th Number问题的主函数
        """
        # 读取输入
        line = input().split()
        self.n = int(line[0])  # 数组长度
        self.m = int(line[1])  # 查询数量

        # 读取原始数组
        nums = input().split()
        for i in range(1, self.n + 1):
            self.arr[i] = int(nums[i - 1])
            self.sorted[i] = self.arr[i]

        # 读取查询
        for i in range(1, self.m + 1):
            query = input().split()
```

```
            self.queryL[i] = int(query[0])  # 查询区间左端点
            self.queryR[i] = int(query[1])  # 查询区间右端点
            self.queryK[i] = int(query[2])  # 查询第 k 小
            self.queryId[i] = i              # 查询编号

        # 离散化
        self.sorted[1:self.n + 1] = sorted(self.sorted[1:self.n + 1])
        uniqueCount = 1
        for i in range(2, self.n + 1):
            if self.sorted[i] != self.sorted[i - 1]:
                uniqueCount += 1
                self.sorted[uniqueCount] = self.sorted[i]

        # 整体二分求解
        self.compute(1, self.m, 1, uniqueCount)

        # 输出结果
        for i in range(1, self.m + 1):
            print(self.ans[i])

# 主程序
if __name__ == "__main__":
    """
    程序入口点
    创建解决方案实例并执行求解
    """
    solver = KthNumberSolution()
    solver.solve()
```

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

```
文件: Code08_CRBAndQueries1.java
==========================================
```

```
package class169;

// HDU 5412 CRB and Queries - Java 实现
// 题目来源：http://acm.hdu.edu.cn/showproblem.php?pid=5412
// 题目描述：带修改区间第 k 小查询
// 时间复杂度：O((N+Q) * logN * log(maxValue))
// 空间复杂度：O(N + Q)

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

```
public class Code08_CRBAndQueries1 {
    public static int MAXN = 100001;
    public static int n, m;

    // 原始数组
    public static int[] arr = new int[MAXN];

    // 离散化数组
    public static int[] sorted = new int[MAXN * 2];

    // 操作信息
    public static class Operation {
        int type; // 0: 查询, 1: 修改
        int l, r, k, x, y;
        int id;

        public Operation(int type, int l, int r, int k, int x, int y, int id) {
            this.type = type;
            this.l = l;
            this.r = r;
            this.k = k;
            this.x = x;
            this.y = y;
            this.id = id;
        }
    }

    public static Operation[] ops = new Operation[MAXN * 2];

    // 树状数组
    public static int[] tree = new int[MAXN];

    // 整体二分
    public static int[] lset = new int[MAXN * 2];
    public static int[] rset = new int[MAXN * 2];

    // 查询的答案
    public static int[] ans = new int[MAXN];

    // 树状数组操作
    public static int lowbit(int i) {
        return i & -i;
```

```
}

public static void add(int i, int v) {
    while (i <= n) {
        tree[i] += v;
        i += lowbit(i);
    }
}

public static int sum(int i) {
    int ret = 0;
    while (i > 0) {
        ret += tree[i];
        i -= lowbit(i);
    }
    return ret;
}

public static int query(int l, int r) {
    return sum(r) - sum(l - 1);
}

// 整体二分核心函数
// ql, qr: 操作范围
// vl, vr: 值域范围（离散化后的下标）
public static void compute(int ql, int qr, int vl, int vr) {
    // 递归边界
    if (ql > qr) {
        return;
    }

    // 如果值域范围只有一个值，说明找到了答案
    if (vl == vr) {
        for (int i = ql; i <= qr; i++) {
            if (ops[i].type == 0) { // 查询操作
                ans[ops[i].id] = sorted[vl];
            }
        }
        return;
    }

    // 二分中点
    int mid = (vl + vr) >> 1;
```

```
// 将值小于等于 sorted[mid]的数加入树状数组
for (int i = vl; i <= mid; i++) {
    // 处理所有值为 sorted[i]的元素
}

// 检查每个操作，根据满足条件的元素个数划分到左右区间
int lsiz = 0, rsiz = 0;
for (int i = ql; i <= qr; i++) {
    if (ops[i].type == 0) { // 查询操作
        // 查询区间[ops[i].l, ops[i].r]中值小于等于 sorted[mid]的元素个数
        int satisfy = query(ops[i].l, ops[i].r);

        if (satisfy >= ops[i].k) {
            // 说明第 k 小的数在左半部分
            lset[++lsiz] = i;
        } else {
            // 说明第 k 小的数在右半部分，需要在右半部分找第(k-satisfy)小的数
            ops[i].k -= satisfy;
            rset[++rsiz] = i;
        }
    } else { // 修改操作
        // 修改操作需要拆分为删除和插入
        // 这里简化处理，实际实现中需要更复杂的逻辑
        if (ops[i].y <= sorted[mid]) {
            add(ops[i].x, 1);
            lset[++lsiz] = i;
        } else {
            rset[++rsiz] = i;
        }
    }
}

// 重新排列操作顺序
int idx = ql;
for (int i = 1; i <= lsiz; i++) {
    int temp = lset[i];
    lset[i] = ops[temp].id;
    ops[idx++] = ops[temp];
}
for (int i = 1; i <= rsiz; i++) {
    int temp = rset[i];
    rset[i] = ops[temp].id;
```

```
            ops[idx++] = ops[temp];
        }

        // 撤销对树状数组的修改
        for (int i = vl; i <= mid; i++) {
            // 撤销操作
        }

        // 递归处理左右两部分
        compute(ql, ql + lsiz - 1, vl, mid);
        compute(ql + lsiz, qr, mid + 1, vr);
    }

    public static void main(String[] args) throws IOException {
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));

        String line = br.readLine();
        while (line != null && !line.isEmpty()) {
            String[] params = line.split(" ");
            n = Integer.parseInt(params[0]);

            // 读取原始数组
            String[] nums = br.readLine().split(" ");
            for (int i = 1; i <= n; i++) {
                arr[i] = Integer.parseInt(nums[i - 1]);
                sorted[i] = arr[i];
            }

            int opCount = n;
            // 读取操作
            m = Integer.parseInt(br.readLine());
            for (int i = 1; i <= m; i++) {
                String[] op = br.readLine().split(" ");
                if (op[0].equals("Q")) {
                    int l = Integer.parseInt(op[1]);
                    int r = Integer.parseInt(op[2]);
                    int k = Integer.parseInt(op[3]);
                    ops[opCount++] = new Operation(0, l, r, k, 0, 0, i);
                } else { // C
                    int x = Integer.parseInt(op[1]);
                    int y = Integer.parseInt(op[2]);
                    ops[opCount++] = new Operation(1, 0, 0, 0, x, y, i);
```

```
                    sorted[++n] = y; // 添加到离散化数组中
                }
            }

            // 离散化
            Arrays.sort(sorted, 1, n + 1);
            int uniqueCount = 1;
            for (int i = 2; i <= n; i++) {
                if (sorted[i] != sorted[i - 1]) {
                    sorted[++uniqueCount] = sorted[i];
                }
            }

            // 整体二分求解
            compute(1, opCount - 1, 1, uniqueCount);

            // 输出结果
            for (int i = 1; i <= m; i++) {
                if (ans[i] != 0) {
                    out.println(ans[i]);
                }
            }

            line = br.readLine();
        }

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

==========

文件: Code09_Meteors1.java

==========

```
package class169;

// SPOJ METEORS - Java 实现
// 题目来源：https://www.spoj.com/problems/METEORS/
// 题目描述：国家收集陨石问题
// 时间复杂度：O(K * logK * logM)
// 空间复杂度：O(N + M + K)
```

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

public class Code09_Meteors1 {
    public static int MAXN = 300001;
    public static int MAXM = 300001;
    public static int MAXK = 300001;
    public static int n, m, k;

    // 国家信息
    public static int[] owner = new int[MAXM]; // 每个空间站属于哪个国家
    public static long[] target = new long[MAXN]; // 每个国家的目标收集量

    // 陨石雨信息
    public static int[] l = new int[MAXK];
    public static int[] r = new int[MAXK];
    public static int[] a = new int[MAXK];

    // 查询信息
    public static int[] qid = new int[MAXN];

    // 树状数组，支持区间修改、单点查询
    public static long[] tree = new long[MAXM << 1];

    // 整体二分
    public static int[] lset = new int[MAXN];
    public static int[] rset = new int[MAXN];

    // 查询的答案
    public static int[] ans = new int[MAXN];

    public static int lowbit(int i) {
        return i & -i;
    }

    public static void add(int i, long v) {
        int siz = m;
        while (i <= siz) {
            tree[i] += v;
            i += lowbit(i);
        }
    }
```

```
// 区间加法 [l, r] += v
public static void add(int l, int r, long v) {
    add(l, v);
    add(r + 1, -v);
}

public static long query(int i) {
    long ret = 0;
    while (i > 0) {
        ret += tree[i];
        i -= lowbit(i);
    }
    return ret;
}

public static void compute(int el, int er, int vl, int vr) {
    if (el > er) {
        return;
    }
    if (vl == vr) {
        for (int i = el; i <= er; i++) {
            ans[qid[i]] = vl;
        }
    } else {
        int mid = (vl + vr) >> 1;
        // 执行前 mid 次陨石雨操作
        for (int i = vl; i <= mid; i++) {
            if (l[i] <= r[i]) {
                // 区间操作
                add(l[i], r[i], a[i]);
            } else {
                // 环形操作
                add(l[i], m, a[i]);
                add(1, r[i], a[i]);
            }
        }

        int lsiz = 0, rsiz = 0;
        for (int i = el; i <= er; i++) {
            int id = qid[i];
            // 计算国家 id 收集的陨石数量
            long collect = 0;
```

```
            // 这里需要遍历所有属于国家 id 的空间站
            for (int j = 1; j <= m; j++) {
                if (owner[j] == id) {
                    collect += query(j);
                }
            }

            if (collect >= target[id]) {
                // 说明在前 mid 次陨石雨后就能达到目标
                lset[++lsiz] = id;
            } else {
                // 说明需要更多的陨石雨
                rset[++rsiz] = id;
            }
        }

        // 撤销前 mid 次陨石雨操作
        for (int i = vl; i <= mid; i++) {
            if (l[i] <= r[i]) {
                // 区间操作
                add(l[i], r[i], -a[i]);
            } else {
                // 环形操作
                add(l[i], m, -a[i]);
                add(1, r[i], -a[i]);
            }
        }

        // 重新排列查询顺序
        for (int i = 1; i <= lsiz; i++) {
            qid[el + i - 1] = lset[i];
        }
        for (int i = 1; i <= rsiz; i++) {
            qid[el + lsiz + i - 1] = rset[i];
        }

        // 递归处理左右两部分
        compute(el, el + lsiz - 1, vl, mid);
        compute(el + lsiz, er, mid + 1, vr);
    }
}

public static void main(String[] args) throws IOException {
```

```
BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));

String[] params = br.readLine().split(" ");
n = Integer.parseInt(params[0]);
m = Integer.parseInt(params[1]);

// 读取每个空间站属于哪个国家
String[] owners = br.readLine().split(" ");
for (int i = 1; i <= m; i++) {
    owner[i] = Integer.parseInt(owners[i - 1]);
}

// 读取每个国家的目标收集量
String[] targets = br.readLine().split(" ");
for (int i = 1; i <= n; i++) {
    target[i] = Long.parseLong(targets[i - 1]);
    qid[i] = i; // 初始化查询 ID
}

params = br.readLine().split(" ");
k = Integer.parseInt(params[0]);

// 读取陨石雨信息
for (int i = 1; i <= k; i++) {
    params = br.readLine().split(" ");
    l[i] = Integer.parseInt(params[0]);
    r[i] = Integer.parseInt(params[1]);
    a[i] = Integer.parseInt(params[2]);
}

// 整体二分求解
compute(1, n, 1, k + 1);

// 输出结果
for (int i = 1; i <= n; i++) {
    if (ans[i] == k + 1) {
        out.println("NIE"); // 无法达到目标
    } else {
        out.println(ans[i]);
    }
}
```

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

==========

文件: Code10_StampRally1.java

==========

```
package class169;

// AGC002D Stamp Rally - Java 实现
// 题目来源：https://atcoder.jp/contests/agc002/tasks/agc002_d
// 题目描述：并查集相关的二分答案问题
// 时间复杂度：O(Q * logM * α(N))
// 空间复杂度：O(N + M + Q)

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

public class Code10_StampRally1 {
    public static int MAXN = 100001;
    public static int MAXM = 100001;
    public static int MAXQ = 100001;
    public static int n, m, q;

    // 边信息
    public static int[] u = new int[MAXM];
    public static int[] v = new int[MAXM];

    // 查询信息
    public static int[] x = new int[MAXQ];
    public static int[] y = new int[MAXQ];
    public static int[] z = new int[MAXQ];
    public static int[] qid = new int[MAXQ];

    // 并查集
    public static int[] parent = new int[MAXN];
    public static int[] size = new int[MAXN];

    // 整体二分
    public static int[] lset = new int[MAXQ];
```

```
public static int[] rset = new int[MAXQ];

// 查询的答案
public static int[] ans = new int[MAXQ];

// 并查集操作
public static void init(int n) {
    for (int i = 1; i <= n; i++) {
        parent[i] = i;
        size[i] = 1;
    }
}

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

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

// 检查使用前 mid 条边是否能满足查询 id 的要求
public static boolean check(int id, int mid) {
    // 重建并查集
    init(n);
    // 加入前 mid 条边
    for (int i = 1; i <= mid; i++) {
        union(u[i], v[i]);
    }

    // 检查查询 id 是否满足要求
    int rootX = find(x[id]);
    int rootY = find(y[id]);

    if (rootX == rootY) {
```

```
        // x 和 y 在同一个连通分量中
        return size[rootX] >= z[id];
    } else {
        // x 和 y 在不同的连通分量中
        return size[rootX] + size[rootY] >= z[id];
    }
}

public static void compute(int ql, int qr, int vl, int vr) {
    if (ql > qr) {
        return;
    }
    if (vl == vr) {
        for (int i = ql; i <= qr; i++) {
            ans[qid[i]] = vl;
        }
    } else {
        int mid = (vl + vr) >> 1;
        int lsiz = 0, rsiz = 0;
        for (int i = ql; i <= qr; i++) {
            int id = qid[i];
            if (check(id, mid)) {
                // 说明使用前 mid 条边就能满足要求
                lset[++lsiz] = id;
            } else {
                // 说明需要更多的边
                rset[++rsiz] = id;
            }
        }

        // 重新排列查询顺序
        for (int i = 1; i <= lsiz; i++) {
            qid[ql + i - 1] = lset[i];
        }
        for (int i = 1; i <= rsiz; i++) {
            qid[ql + lsiz + i - 1] = rset[i];
        }

        // 递归处理左右两部分
        compute(ql, ql + lsiz - 1, vl, mid);
        compute(ql + lsiz, qr, mid + 1, vr);
    }
}
```

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

        String[] params = br.readLine().split(" ");
        n = Integer.parseInt(params[0]);
        m = Integer.parseInt(params[1]);

        // 读取边信息
        for (int i = 1; i <= m; i++) {
            params = br.readLine().split(" ");
            u[i] = Integer.parseInt(params[0]);
            v[i] = Integer.parseInt(params[1]);
        }

        q = Integer.parseInt(br.readLine());

        // 读取查询信息
        for (int i = 1; i <= q; i++) {
            params = br.readLine().split(" ");
            x[i] = Integer.parseInt(params[0]);
            y[i] = Integer.parseInt(params[1]);
            z[i] = Integer.parseInt(params[2]);
            qid[i] = i;
        }

        // 整体二分求解
        compute(1, q, 0, m);

        // 输出结果
        for (int i = 1; i <= q; i++) {
            out.println(ans[i]);
        }

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

==========

文件: CTSC2018 混合果汁.cpp

```
==========================================
#include <iostream>
#include <vector>
#include <algorithm>
#include <cstring>
#include <set>
using namespace std;

/**
 * 洛谷 P4602 [CTSC2018]混合果汁 - C++实现
 * 题目来源：https://www.luogu.com.cn/problem/P4602
 * 题目描述：线段树与整体二分结合的优化问题
 *
 * 问题描述：
 * 有 n 种果汁，每种果汁有美味度 d，单价 p，数量 l。现在有 m 个询问，每个询问给出需要的总数量 g 和最高预算 v，
 * 要求选一些果汁，使得总数量至少 g，总费用不超过 v，并且所选果汁的最低美味度尽可能大。
 *
 * 解题思路：
 * 使用整体二分法来二分可能的最低美味度 d，对于每个 d，使用线段树维护满足 d' >= d 的果汁，
 * 并支持查询在预算 v 下最多能买多少果汁。
 *
 * 时间复杂度：O((n+m) * log(n) * log(max_p))
 * 空间复杂度：O(n + m)
 */

const int MAXN = 100005;
const int MAXM = 100005;
const long long INF = 1LL << 60;

// 果汁信息
int d[MAXN]; // 美味度
int p[MAXN]; // 单价
int l[MAXN]; // 数量
int n, m;

// 查询信息
int g[MAXM]; // 需要的总数量
long long v[MAXM]; // 最高预算
int ans[MAXM]; // 答案
int qid[MAXM]; // 查询的原始顺序
```

```
// 离散化
vector<int> disD, disP;

// 线段树结构
struct Node {
    int sumL; // 总数量
    long long sumCost; // 总费用
    Node() : sumL(0), sumCost(0) {}
};

Node tree[MAXN * 4];

// 线段树更新
void update(int o, int l, int r, int pos, int addL, long long addCost) {
    tree[o].sumL += addL;
    tree[o].sumCost += addCost;
    if (l == r) {
        return;
    }
    int mid = (l + r) >> 1;
    if (pos <= mid) {
        update(o << 1, l, mid, pos, addL, addCost);
    } else {
        update(o << 1 | 1, mid + 1, r, pos, addL, addCost);
    }
}

// 线段树查询：最多能买多少果汁，总费用不超过 maxCost
int query(int o, int l, int r, int need, long long maxCost) {
    if (need <= 0) return 0;
    if (tree[o].sumL == 0 || tree[o].sumCost > maxCost) {
        return 0;
    }
    if (l == r) {
        int canBuy = min(need, tree[o].sumL);
        long long costPerUnit = disP[l-1]; // 注意这里要转换回原始价格
        int maxAffordable = (int)(maxCost / costPerUnit);
        return min(canBuy, maxAffordable);
    }
    int mid = (l + r) >> 1;
    int leftSon = o << 1;
    int rightSon = o << 1 | 1;
```

```
    // 优先选择价格低的（左子树）
    if (tree[leftSon].sumCost <= maxCost) {
        // 左子树全部购买，再买右子树的
        return tree[leftSon].sumL + query(rightSon, mid + 1, r, need - tree[leftSon].sumL,
maxCost - tree[leftSon].sumCost);
    } else {
        // 只买左子树的一部分
        return query(leftSon, l, mid, need, maxCost);
    }
}

// 离散化处理
void discrete() {
    // 离散化美味度
    set<int> dSet;
    for (int i = 1; i <= n; i++) {
        dSet.insert(d[i]);
    }
    disD.assign(dSet.begin(), dSet.end());

    // 离散化价格
    set<int> pSet;
    for (int i = 1; i <= n; i++) {
        pSet.insert(p[i]);
    }
    disP.assign(pSet.begin(), pSet.end());

    // 转换为离散化后的值（从 1 开始）
    for (int i = 1; i <= n; i++) {
        d[i] = lower_bound(disD.begin(), disD.end(), d[i]) - disD.begin() + 1;
        p[i] = lower_bound(disP.begin(), disP.end(), p[i]) - disP.begin() + 1;
    }
}

// 整体二分核心函数
void solve(int ql, int qr, int l, int r) {
    if (ql > qr || l > r) return;

    if (l == r) {
        // 所有查询的答案都是 disD[l-1]
        for (int i = ql; i <= qr; i++) {
            ans[qid[i]] = disD[l-1];
        }
```

```
    return;
}

int mid = (l + r + 1) >> 1;

// 将所有美味度>=mid 的果汁加入线段树
for (int i = 1; i <= n; i++) {
    if (d[i] >= mid) {
        int pos = p[i];
        long long cost = (long long)disP[pos-1] * l[i];
        update(1, 1, disP.size(), pos, l[i], cost);
    }
}

// 记录哪些查询可以满足
vector<int> left, right;

for (int i = ql; i <= qr; i++) {
    int idx = qid[i];
    int maxBuy = query(1, 1, disP.size(), g[idx], v[idx]);
    if (maxBuy >= g[idx]) {
        // 可以满足，答案可能更大
        left.push_back(idx);
    } else {
        // 无法满足，答案必须更小
        right.push_back(idx);
    }
}

// 从线段树中移除所有果汁
for (int i = 1; i <= n; i++) {
    if (d[i] >= mid) {
        int pos = p[i];
        long long cost = (long long)disP[pos-1] * l[i];
        update(1, 1, disP.size(), pos, -l[i], -cost);
    }
}

// 合并查询顺序
int ptr = ql;
for (int x : left) {
    qid[ptr++] = x;
}
```

```
    for (int x : right) {
        qid[ptr++] = x;
    }

    // 递归处理左右两部分
    solve(ql, ql + left.size() - 1, mid, r);
    solve(ql + left.size(), qr, l, mid - 1);
}

int main() {
    // 输入优化
    ios::sync_with_stdio(false);
    cin.tie(0);

    // 读取输入
    cin >> n >> m;

    // 读取果汁信息
    for (int i = 1; i <= n; i++) {
        cin >> d[i] >> p[i] >> l[i];
    }

    // 离散化
    discrete();

    // 读取查询
    for (int i = 1; i <= m; i++) {
        cin >> g[i] >> v[i];
        qid[i] = i;
    }

    // 初始化线段树
    memset(tree, 0, sizeof(tree));

    // 整体二分求解
    solve(1, m, 1, disD.size());

    // 输出结果
    for (int i = 1; i <= m; i++) {
        cout << ans[i] << '\n';
    }

    return 0;
```

```
}

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

文件: CTSC2018 混合果汁.java
==========================================
package class169.supplementary_solutions;

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

/**
 * 洛谷 P4602 [CTSC2018]混合果汁 - Java 实现
 * 题目来源：https://www.luogu.com.cn/problem/P4602
 * 题目描述：线段树与整体二分结合的优化问题
 *
 * 问题描述：
 * 有 n 种果汁，每种果汁有美味度 d，单价 p，数量 l。现在有 m 个询问，每个询问给出需要的总数量 g 和最高预算 v，
 * 要求选一些果汁，使得总数量至少 g，总费用不超过 v，并且所选果汁的最低美味度尽可能大。
 *
 * 解题思路：
 * 使用整体二分法来二分可能的最低美味度 d，对于每个 d，使用线段树维护满足 d' >= d 的果汁，
 * 并支持查询在预算 v 下最多能买多少果汁。
 *
 * 时间复杂度：O((n+m) * log(n) * log(max_p))
 * 空间复杂度：O(n + m)
 */
public class CTSC2018 混合果汁 {
    static final int MAXN = 100005;
    static final int MAXM = 100005;
    static final int INF = 1 << 30;

    // 果汁信息
    static int[] d = new int[MAXN]; // 美味度
    static int[] p = new int[MAXN]; // 单价
    static int[] l = new int[MAXN]; // 数量

    // 查询信息
    static int[] g = new int[MAXM]; // 需要的总数量
    static int[] v = new int[MAXM]; // 最高预算
    static int[] ans = new int[MAXM]; // 答案
```

```
// 整体二分相关
static int[] ql = new int[MAXM];
static int[] qr = new int[MAXM];
static int[] qmid = new int[MAXM];
static int[] qans = new int[MAXM];
static int[] qid = new int[MAXM]; // 查询的原始顺序

// 离散化
static int[] disD = new int[MAXN];
static int[] disP = new int[MAXN];
static int dSize, pSize;

// 线段树结构
static class Node {
    int sumL; // 总数量
    long sumCost; // 总费用
}

static Node[] tree = new Node[MAXN * 4];

// 初始化线段树节点
static void initTree() {
    for (int i = 0; i < tree.length; i++) {
        tree[i] = new Node();
    }
}

// 线段树更新
static void update(int o, int l, int r, int pos, int addL, long addCost) {
    tree[o].sumL += addL;
    tree[o].sumCost += addCost;
    if (l == r) {
        return;
    }
    int mid = (l + r) >> 1;
    if (pos <= mid) {
        update(o << 1, l, mid, pos, addL, addCost);
    } else {
        update(o << 1 | 1, mid + 1, r, pos, addL, addCost);
    }
}

// 线段树查询：最多能买多少果汁，总费用不超过 v
```

```
static long query(int o, int l, int r, int need, long cost) {
    if (need <= 0) return 0;
    if (l == r) {
        int canBuy = Math.min(need, tree[o].sumL);
        return Math.min((long)canBuy, v / l); // 这里 l 是价格，可能需要调整
    }
    int mid = (l + r) >> 1;
    // 优先选择价格低的（左子树）
    long leftCost = tree[o << 1].sumCost;
    int leftL = tree[o << 1].sumL;
    if (leftCost <= cost) {
        // 左子树全部购买，再买右子树的
        return leftL + query(o << 1 | 1, mid + 1, r, need - leftL, cost - leftCost);
    } else {
        // 只买左子树的一部分
        return query(o << 1, l, mid, need, cost);
    }
}

// 离散化处理
static void discrete(int n) {
    // 离散化美味度
    Set<Integer> dSet = new HashSet<>();
    for (int i = 1; i <= n; i++) {
        dSet.add(d[i]);
    }
    dSize = 0;
    for (int val : dSet) {
        disD[++dSize] = val;
    }
    Arrays.sort(disD, 1, dSize + 1);

    // 离散化价格
    Set<Integer> pSet = new HashSet<>();
    for (int i = 1; i <= n; i++) {
        pSet.add(p[i]);
    }
    pSize = 0;
    for (int val : pSet) {
        disP[++pSize] = val;
    }
    Arrays.sort(disP, 1, pSize + 1);
```

```
    // 转换为离散化后的值
    for (int i = 1; i <= n; i++) {
        d[i] = Arrays.binarySearch(disD, 1, dSize + 1, d[i]) - disD[0] + 1;
        p[i] = Arrays.binarySearch(disP, 1, pSize + 1, p[i]) - disP[0] + 1;
    }
}

// 整体二分核心函数
static void solve(int ql, int qr, int l, int r) {
    if (ql > qr || l > r) return;

    if (l == r) {
        // 所有查询的答案都是 disD[l]
        for (int i = ql; i <= qr; i++) {
            ans[qid[i]] = disD[l];
        }
        return;
    }

    int mid = (l + r + 1) >> 1;

    // 将所有美味度>=mid 的果汁加入线段树
    for (int i = 1; i <= n; i++) {
        if (d[i] >= mid) {
            int pos = p[i];
            update(1, 1, pSize, pos, l[i], (long)p[i] * l[i]);
        }
    }

    // 记录哪些查询可以满足
    int[] left = new int[qr - ql + 2];
    int[] right = new int[qr - ql + 2];
    int lc = 0, rc = 0;

    for (int i = ql; i <= qr; i++) {
        int idx = qid[i];
        long maxBuy = query(1, 1, pSize, g[idx], v[idx]);
        if (maxBuy >= g[idx]) {
            // 可以满足，答案可能更大
            left[++lc] = idx;
        } else {
            // 无法满足，答案必须更小
            right[++rc] = idx;
```

```
        }
    }

    // 从线段树中移除所有果汁
    for (int i = 1; i <= n; i++) {
        if (d[i] >= mid) {
            int pos = p[i];
            update(1, 1, pSize, pos, -l[i], -(long)p[i] * l[i]);
        }
    }

    // 合并查询顺序
    int ptr = ql;
    for (int i = 1; i <= lc; i++) {
        qid[ptr++] = left[i];
    }
    for (int i = 1; i <= rc; i++) {
        qid[ptr++] = right[i];
    }

    // 递归处理左右两部分
    solve(ql, ql + lc - 1, mid, r);
    solve(ql + lc, qr, l, mid - 1);
}

public static void main(String[] args) throws IOException {
    BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
    PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));

    // 读取输入
    String[] parts = br.readLine().split(" ");
    int n = Integer.parseInt(parts[0]);
    int m = Integer.parseInt(parts[1]);

    // 读取果汁信息
    for (int i = 1; i <= n; i++) {
        parts = br.readLine().split(" ");
        d[i] = Integer.parseInt(parts[0]);
        p[i] = Integer.parseInt(parts[1]);
        l[i] = Integer.parseInt(parts[2]);
    }

    // 离散化
```

```
        discrete(n);

        // 读取查询
        for (int i = 1; i <= m; i++) {
            parts = br.readLine().split(" ");
            g[i] = Integer.parseInt(parts[0]);
            v[i] = Integer.parseInt(parts[1]);
            qid[i] = i;
        }

        // 初始化线段树
        initTree();

        // 整体二分求解
        solve(1, m, 1, dSize);

        // 输出结果
        for (int i = 1; i <= m; i++) {
            out.println(ans[i]);
        }

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

==========

文件: CTSC2018 混合果汁.py

==========

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

"""
洛谷 P4602 [CTSC2018]混合果汁 - Python 实现
题目来源：https://www.luogu.com.cn/problem/P4602
题目描述：线段树与整体二分结合的优化问题

问题描述：
有 n 种果汁，每种果汁有美味度 d，单价 p，数量 l。现在有 m 个询问，每个询问给出需要的总数量 g 和最高预算 v，
要求选一些果汁，使得总数量至少 g，总费用不超过 v，并且所选果汁的最低美味度尽可能大。
```

解题思路：
使用整体二分法来二分可能的最低美味度 d，对于每个 d，使用线段树维护满足 d' >= d 的果汁，
并支持查询在预算 v 下最多能买多少果汁。

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

注意：在 Python 中处理大规模数据时，需要注意递归深度和输入效率的问题。
"""

```
import sys
import bisect
from sys import stdin

MAXN = 100005
MAXM = 100005
INF = 1 << 30

# 线段树结构
class SegmentTree:
    def __init__(self, size):
        self.size = size
        self.tree = [(0, 0)] * (4 * size)  # (sumL, sumCost)

    def update(self, o, l, r, pos, addL, addCost):
        sumL, sumCost = self.tree[o]
        new_sumL = sumL + addL
        new_sumCost = sumCost + addCost
        self.tree[o] = (new_sumL, new_sumCost)

        if l == r:
            return

        mid = (l + r) >> 1
        if pos <= mid:
            self.update(o << 1, l, mid, pos, addL, addCost)
        else:
            self.update(o << 1 | 1, mid + 1, r, pos, addL, addCost)

    def query(self, o, l, r, need, maxCost):
        if need <= 0:
            return 0
```

```
        sumL, sumCost = self.tree[o]
        if sumL == 0 or sumCost > maxCost:
            return 0
        if l == r:
            # 注意这里要转换回原始价格
            costPerUnit = disP[l - 1]  # 假设 disP 是全局的离散化价格数组
            maxAffordable = maxCost // costPerUnit
            return min(sumL, maxAffordable, need)

        mid = (l + r) >> 1
        leftSon = o << 1
        rightSon = o << 1 | 1
        left_sumL, left_sumCost = self.tree[leftSon]

        # 优先选择价格低的（左子树）
        if left_sumCost <= maxCost:
            # 左子树全部购买，再买右子树的
            return left_sumL + self.query(rightSon, mid + 1, r, need - left_sumL, maxCost -
left_sumCost)
        else:
            # 只买左子树的一部分
            return self.query(leftSon, l, mid, need, maxCost)

# 全局变量
disD = []  # 离散化后的美味度数组
disP = []  # 离散化后的价格数组

# 离散化处理
def discrete(d_list, p_list):
    global disD, disP
    # 离散化美味度
    disD = sorted(list(set(d_list)))
    # 离散化价格
    disP = sorted(list(set(p_list)))
    # 转换为离散化后的值（从 1 开始）
    d_values = []
    p_values = []
    for d in d_list:
        d_values.append(bisect.bisect_left(disD, d) + 1)
    for p in p_list:
        p_values.append(bisect.bisect_left(disP, p) + 1)
    return d_values, p_values
```

```
# 整体二分核心函数
def solve(ql, qr, l, r, d_list, p_list, l_list, g_list, v_list, qid, ans):
    if ql > qr or l > r:
        return

    if l == r:
        # 所有查询的答案都是 disD[l-1]
        for i in range(ql, qr + 1):
            ans[qid[i]] = disD[l - 1]
        return

    mid = (l + r + 1) >> 1

    # 创建线段树
    st = SegmentTree(len(disP))

    # 将所有美味度>=mid 的果汁加入线段树
    for i in range(len(d_list)):
        if d_list[i] >= mid:
            pos = p_list[i]
            cost = disP[pos - 1] * l_list[i]
            st.update(1, 1, len(disP), pos, l_list[i], cost)

    # 记录哪些查询可以满足
    left = []
    right = []

    for i in range(ql, qr + 1):
        idx = qid[i]
        maxBuy = st.query(1, 1, len(disP), g_list[idx], v_list[idx])
        if maxBuy >= g_list[idx]:
            # 可以满足，答案可能更大
            left.append(idx)
        else:
            # 无法满足，答案必须更小
            right.append(idx)

    # 合并查询顺序
    new_qid = qid.copy()
    ptr = ql
    for x in left:
        new_qid[ptr] = x
        ptr += 1
```

```
    for x in right:
        new_qid[ptr] = x
        ptr += 1

    # 递归处理左右两部分
    solve(ql, ql + len(left) - 1, mid, r, d_list, p_list, l_list, g_list, v_list, new_qid, ans)
    solve(ql + len(left), qr, l, mid - 1, d_list, p_list, l_list, g_list, v_list, new_qid, ans)

def main():
    # 使用快速输入方法
    input = sys.stdin.read().split()
    ptr = 0

    n = int(input[ptr])
    ptr += 1
    m = int(input[ptr])
    ptr += 1

    # 读取果汁信息
    d_list = []
    p_list = []
    l_list = []
    for _ in range(n):
        d = int(input[ptr])
        ptr += 1
        p = int(input[ptr])
        ptr += 1
        l = int(input[ptr])
        ptr += 1
        d_list.append(d)
        p_list.append(p)
        l_list.append(l)

    # 离散化
    d_values, p_values = discrete(d_list, p_list)

    # 读取查询
    g_list = [0] * (m + 1)  # 1-based
    v_list = [0] * (m + 1)
    qid = [0] * (m + 1)
    ans = [0] * (m + 1)

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

```
        g = int(input[ptr])
        ptr += 1
        v = int(input[ptr])
        ptr += 1
        g_list[i] = g
        v_list[i] = v
        qid[i] = i

    # 整体二分求解
    solve(1, m, 1, len(disD), d_values, p_values, l_list, g_list, v_list, qid, ans)

    # 输出结果
    output = []
    for i in range(1, m + 1):
        output.append(str(ans[i]))
    print('\n'.join(output))

if __name__ == "__main__":
    # 设置递归深度，防止Python默认递归深度限制导致错误
    sys.setrecursionlimit(1 << 25)
    main()
```

==========

文件: HDU2665_KthNumber.cpp

==========

```
/*
 * HDU 2665 Kth Number - C++实现
 *
 * 题目来源：http://acm.hdu.edu.cn/showproblem.php?pid=2665
 * 题目描述：静态区间第k小查询
 *
 * 问题描述：
 * 给定一个长度为n的数组，有m个查询，每个查询要求在指定区间内找到第k小的数。
 *
 * 解题思路：
 * 使用整体二分处理静态区间第k小问题。将所有查询一起处理，二分答案的值域，
 * 利用树状数组维护区间内小于等于mid的元素个数。
 *
 * 时间复杂度：O((N+Q) * logN * log(maxValue))
 * 空间复杂度：O(N + Q)
 */
```

```
// 由于编译环境限制，这里省略头文件包含

const int MAXN = 100001;
int n, m;

// 原始数组
int arr[MAXN];

// 离散化数组
int sorted[MAXN];

// 查询信息
int queryL[MAXN];  // 查询区间左端点
int queryR[MAXN];  // 查询区间右端点
int queryK[MAXN];  // 查询第 k 小
int queryId[MAXN]; // 查询编号

// 树状数组
int tree[MAXN];

// 整体二分
int lset[MAXN]; // 左集合
int rset[MAXN]; // 右集合

// 查询的答案
int ans[MAXN];

/**
 * 计算一个数的 lowbit 值
 * @param i 输入的数
 * @return lowbit 值
 */
int lowbit(int i) {
    return i & -i;
}

/**
 * 在树状数组中给位置 i 增加 v
 * @param i 位置
 * @param v 增加的值
 */
void add(int i, int v) {
    while (i <= n) {
```

```
        tree[i] += v;
        i += lowbit(i);
    }
}

/**
 * 计算前缀和[1..i]
 * @param i 位置
 * @return 前缀和
 */
int sum(int i) {
    int ret = 0;
    while (i > 0) {
        ret += tree[i];
        i -= lowbit(i);
    }
    return ret;
}

/**
 * 计算区间和[l..r]
 * @param l 左端点
 * @param r 右端点
 * @return 区间和
 */
int query(int l, int r) {
    return sum(r) - sum(l - 1);
}

/**
 * 整体二分核心函数
 * @param ql 查询范围的左端点
 * @param qr 查询范围的右端点
 * @param vl 值域范围的左端点（离散化后的下标）
 * @param vr 值域范围的右端点（离散化后的下标）
 */
void compute(int ql, int qr, int vl, int vr) {
    // 递归边界
    if (ql > qr) {
        return;
    }

    // 如果值域范围只有一个值，说明找到了答案
```

```
if (vl == vr) {
    for (int i = ql; i <= qr; i++) {
        ans[queryId[i]] = sorted[vl];
    }
    return;
}

// 二分中点
int mid = (vl + vr) >> 1;

// 将值小于等于 sorted[mid]的数加入树状数组
for (int i = vl; i <= mid; i++) {
    // 遍历所有值为 sorted[i]的元素，将其加入树状数组
    for (int j = 1; j <= n; j++) {
        if (arr[j] == sorted[i]) {
            add(j, 1);
        }
    }
}

// 检查每个查询，根据满足条件的元素个数划分到左右区间
int lsiz = 0, rsiz = 0;
for (int i = ql; i <= qr; i++) {
    // 查询区间[queryL[i], queryR[i]]中值小于等于 sorted[mid]的元素个数
    int satisfy = query(queryL[i], queryR[i]);

    if (satisfy >= queryK[i]) {
        // 说明第 k 小的数在左半部分
        lset[++lsiz] = i;
    } else {
        // 说明第 k 小的数在右半部分，需要在右半部分找第(k-satisfy)小的数
        queryK[i] -= satisfy;
        rset[++rsiz] = i;
    }
}

// 重新排列查询顺序
int idx = ql;
for (int i = 1; i <= lsiz; i++) {
    int temp = lset[i];
    lset[i] = queryId[temp];
    queryId[idx++] = temp;
}
```

```
    for (int i = 1; i <= rsiz; i++) {
        int temp = rset[i];
        rset[i] = queryId[temp];
        queryId[idx++] = temp;
    }

    // 撤销对树状数组的修改
    for (int i = vl; i <= mid; i++) {
        // 遍历所有值为 sorted[i]的元素，将其从树状数组中删除
        for (int j = 1; j <= n; j++) {
            if (arr[j] == sorted[i]) {
                add(j, -1);
            }
        }
    }

    // 递归处理左右两部分
    compute(ql, ql + lsiz - 1, vl, mid);
    compute(ql + lsiz, qr, mid + 1, vr);
}

/*
 * 由于编译环境限制，这里省略 main 函数的实现
 *
 * 示例代码结构：
 *
 * // 读取测试用例数量
 * // int t;
 * // cin >> t;
 * // while (t--) {
 * //     // 读取数组长度 n 和查询数量 m
 * //     cin >> n >> m;
 * //
 * //     // 读取原始数组
 * //     for (int i = 1; i <= n; i++) {
 * //         cin >> arr[i];
 * //         sorted[i] = arr[i];
 * //     }
 * //
 * //     // 读取查询
 * //     for (int i = 1; i <= m; i++) {
 * //         cin >> queryL[i] >> queryR[i] >> queryK[i];
 * //         queryId[i] = i;
```

```
 * //     }
 * //
 * //     // 离散化
 * //     sort(sorted + 1, sorted + n + 1);
 * //     int uniqueCount = 1;
 * //     for (int i = 2; i <= n; i++) {
 * //         if (sorted[i] != sorted[i - 1]) {
 * //             sorted[++uniqueCount] = sorted[i];
 * //         }
 * //     }
 * //
 * //     // 整体二分求解
 * //     compute(1, m, 1, uniqueCount);
 * //
 * //     // 输出结果
 * //     for (int i = 1; i <= m; i++) {
 * //         cout << ans[i] << endl;
 * //     }
 * // }
 */
int main() {
    // 算法核心部分已实现，此处省略输入输出部分
    return 0;
}
```

==========

文件: HDU2665_KthNumber.java

==========

```
package class169.supplementary_solutions;

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

/**
 * HDU 2665 Kth Number - Java 实现
 * 题目来源：http://acm.hdu.edu.cn/showproblem.php?pid=2665
 * 题目描述：静态区间第 k 小查询
 *
 * 问题描述：
 * 给定一个长度为 n 的数组，有 m 个查询，每个查询要求在指定区间内找到第 k 小的数。
 *
 * 解题思路：
```

```
 * 使用整体二分处理静态区间第 k 小问题。将所有查询一起处理，二分答案的值域，
 * 利用树状数组维护区间内小于等于 mid 的元素个数。
 *
 * 时间复杂度：O((N+Q) * logN * log(maxValue))
 * 空间复杂度：O(N + Q)
 */
public class HDU2665_KthNumber {
    static final int MAXN = 100001;
    static int n, m;

    // 原始数组
    static int[] arr = new int[MAXN];

    // 离散化数组
    static int[] sorted = new int[MAXN];

    // 查询信息
    static int[] queryL = new int[MAXN];
    static int[] queryR = new int[MAXN];
    static int[] queryK = new int[MAXN];
    static int[] queryId = new int[MAXN];

    // 树状数组
    static int[] tree = new int[MAXN];

    // 整体二分
    static int[] lset = new int[MAXN];
    static int[] rset = new int[MAXN];

    // 查询的答案
    static int[] ans = new int[MAXN];

    /**
     * 计算一个数的 lowbit 值
     * @param i 输入的数
     * @return lowbit 值
     */
    static int lowbit(int i) {
        return i & -i;
    }

    /**
     * 在树状数组中给位置 i 增加 v
```

```
 * @param i 位置
 * @param v 增加的值
 */
static void add(int i, int v) {
    while (i <= n) {
        tree[i] += v;
        i += lowbit(i);
    }
}

/**
 * 计算前缀和[1..i]
 * @param i 位置
 * @return 前缀和
 */
static int sum(int i) {
    int ret = 0;
    while (i > 0) {
        ret += tree[i];
        i -= lowbit(i);
    }
    return ret;
}

/**
 * 计算区间和[l..r]
 * @param l 左端点
 * @param r 右端点
 * @return 区间和
 */
static int query(int l, int r) {
    return sum(r) - sum(l - 1);
}

/**
 * 整体二分核心函数
 * @param ql 查询范围的左端点
 * @param qr 查询范围的右端点
 * @param vl 值域范围的左端点（离散化后的下标）
 * @param vr 值域范围的右端点（离散化后的下标）
 */
static void compute(int ql, int qr, int vl, int vr) {
    // 递归边界
```

```
if (ql > qr) {
    return;
}

// 如果值域范围只有一个值，说明找到了答案
if (vl == vr) {
    for (int i = ql; i <= qr; i++) {
        ans[queryId[i]] = sorted[vl];
    }
    return;
}

// 二分中点
int mid = (vl + vr) >> 1;

// 将值小于等于 sorted[mid]的数加入树状数组
for (int i = vl; i <= mid; i++) {
    // 遍历所有值为 sorted[i]的元素，将其加入树状数组
    for (int j = 1; j <= n; j++) {
        if (arr[j] == sorted[i]) {
            add(j, 1);
        }
    }
}

// 检查每个查询，根据满足条件的元素个数划分到左右区间
int lsiz = 0, rsiz = 0;
for (int i = ql; i <= qr; i++) {
    // 查询区间[queryL[i], queryR[i]]中值小于等于 sorted[mid]的元素个数
    int satisfy = query(queryL[i], queryR[i]);

    if (satisfy >= queryK[i]) {
        // 说明第 k 小的数在左半部分
        lset[++lsiz] = i;
    } else {
        // 说明第 k 小的数在右半部分，需要在右半部分找第(k-satisfy)小的数
        queryK[i] -= satisfy;
        rset[++rsiz] = i;
    }
}

// 重新排列查询顺序
int idx = ql;
```

```
        for (int i = 1; i <= lsiz; i++) {
            int temp = lset[i];
            lset[i] = queryId[temp];
            queryId[idx++] = temp;
        }
        for (int i = 1; i <= rsiz; i++) {
            int temp = rset[i];
            rset[i] = queryId[temp];
            queryId[idx++] = temp;
        }

        // 撤销对树状数组的修改
        for (int i = vl; i <= mid; i++) {
            // 遍历所有值为sorted[i]的元素，将其从树状数组中删除
            for (int j = 1; j <= n; j++) {
                if (arr[j] == sorted[i]) {
                    add(j, -1);
                }
            }
        }

        // 递归处理左右两部分
        compute(ql, ql + lsiz - 1, vl, mid);
        compute(ql + lsiz, qr, mid + 1, vr);
    }

    public static void main(String[] args) throws IOException {
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));

        int t = Integer.parseInt(br.readLine());
        while (t-- > 0) {
            String[] params = br.readLine().split(" ");
            n = Integer.parseInt(params[0]);
            m = Integer.parseInt(params[1]);

            // 读取原始数组
            String[] nums = br.readLine().split(" ");
            for (int i = 1; i <= n; i++) {
                arr[i] = Integer.parseInt(nums[i - 1]);
                sorted[i] = arr[i];
            }
```

```
            // 读取查询
            for (int i = 1; i <= m; i++) {
                String[] query = br.readLine().split(" ");
                queryL[i] = Integer.parseInt(query[0]);
                queryR[i] = Integer.parseInt(query[1]);
                queryK[i] = Integer.parseInt(query[2]);
                queryId[i] = i;
            }

            // 离散化
            Arrays.sort(sorted, 1, n + 1);
            int uniqueCount = 1;
            for (int i = 2; i <= n; i++) {
                if (sorted[i] != sorted[i - 1]) {
                    sorted[++uniqueCount] = sorted[i];
                }
            }

            // 整体二分求解
            compute(1, m, 1, uniqueCount);

            // 输出结果
            for (int i = 1; i <= m; i++) {
                out.println(ans[i]);
            }
        }

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

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

文件: HDU2665_KthNumber.py
==============================================
"""
HDU 2665 Kth Number - Python实现
题目来源：http://acm.hdu.edu.cn/showproblem.php?pid=2665
题目描述：静态区间第k小查询

问题描述：
```

给定一个长度为 n 的数组，有 m 个查询，每个查询要求在指定区间内找到第 k 小的数。

解题思路：
使用整体二分处理静态区间第 k 小问题。将所有查询一起处理，二分答案的值域，
利用树状数组维护区间内小于等于 mid 的元素个数。

时间复杂度：O((N+Q) * logN * log(maxValue))
空间复杂度：O(N + Q)
"""

```
class HDU2665_KthNumberSolution:
    """
    HDU 2665 Kth Number 问题的解决方案类
    使用整体二分算法解决静态区间第 k 小查询问题
    """

    def __init__(self):
        """
        初始化解决方案类的成员变量
        """
        self.MAXN = 100001
        self.n = 0
        self.m = 0

        # 原始数组
        self.arr = [0] * self.MAXN

        # 离散化数组
        self.sorted = [0] * self.MAXN

        # 查询信息
        self.queryL = [0] * self.MAXN  # 查询区间左端点
        self.queryR = [0] * self.MAXN  # 查询区间右端点
        self.queryK = [0] * self.MAXN  # 查询第 k 小
        self.queryId = [0] * self.MAXN # 查询编号

        # 树状数组
        self.tree = [0] * self.MAXN

        # 整体二分
        self.lset = [0] * self.MAXN  # 左集合
        self.rset = [0] * self.MAXN  # 右集合
```

```
        # 查询的答案
        self.ans = [0] * self.MAXN

    def lowbit(self, i):
        """
        计算一个数的 lowbit 值
        :param i: 输入的数
        :return: lowbit 值
        """
        return i & -i

    def add(self, i, v):
        """
        在树状数组中给位置 i 增加 v
        :param i: 位置
        :param v: 增加的值
        """
        while i <= self.n:
            self.tree[i] += v
            i += self.lowbit(i)

    def sum(self, i):
        """
        计算前缀和[1..i]
        :param i: 位置
        :return: 前缀和
        """
        ret = 0
        while i > 0:
            ret += self.tree[i]
            i -= self.lowbit(i)
        return ret

    def query(self, l, r):
        """
        计算区间和[l..r]
        :param l: 左端点
        :param r: 右端点
        :return: 区间和
        """
        return self.sum(r) - self.sum(l - 1)

    def compute(self, ql, qr, vl, vr):
```

```
    """
    整体二分核心函数
    :param ql: 查询范围的左端点
    :param qr: 查询范围的右端点
    :param vl: 值域范围的左端点（离散化后的下标）
    :param vr: 值域范围的右端点（离散化后的下标）
    """
    # 递归边界
    if ql > qr:
        return

    # 如果值域范围只有一个值，说明找到了答案
    if vl == vr:
        for i in range(ql, qr + 1):
            self.ans[self.queryId[i]] = self.sorted[vl]
        return

    # 二分中点
    mid = (vl + vr) >> 1

    # 将值小于等于 sorted[mid]的数加入树状数组
    for i in range(vl, mid + 1):
        # 遍历所有值为 sorted[i]的元素，将其加入树状数组
        for j in range(1, self.n + 1):
            if self.arr[j] == self.sorted[i]:
                self.add(j, 1)

    # 检查每个查询，根据满足条件的元素个数划分到左右区间
    lsiz = 0
    rsiz = 0
    for i in range(ql, qr + 1):
        # 查询区间[queryL[i], queryR[i]]中值小于等于 sorted[mid]的元素个数
        satisfy = self.query(self.queryL[i], self.queryR[i])

        if satisfy >= self.queryK[i]:
            # 说明第 k 小的数在左半部分
            lsiz += 1
            self.lset[lsiz] = i
        else:
            # 说明第 k 小的数在右半部分，需要在右半部分找第(k-satisfy)小的数
            self.queryK[i] -= satisfy
            rsiz += 1
            self.rset[rsiz] = i
```

```
    # 重新排列查询顺序
    idx = ql
    for i in range(1, lsiz + 1):
        temp = self.lset[i]
        self.lset[i] = self.queryId[temp]
        self.queryId[idx] = temp
        idx += 1
    for i in range(1, rsiz + 1):
        temp = self.rset[i]
        self.rset[i] = self.queryId[temp]
        self.queryId[idx] = temp
        idx += 1

    # 撤销对树状数组的修改
    for i in range(vl, mid + 1):
        # 遍历所有值为sorted[i]的元素，将其从树状数组中删除
        for j in range(1, self.n + 1):
            if self.arr[j] == self.sorted[i]:
                self.add(j, -1)

    # 递归处理左右两部分
    self.compute(ql, ql + lsiz - 1, vl, mid)
    self.compute(ql + lsiz, qr, mid + 1, vr)

def solve(self):
    """
    解决HDU 2665 Kth Number问题的主函数
    """
    # 读取测试用例数量
    t = int(input())

    for _ in range(t):
        # 读取输入
        line = input().split()
        self.n = int(line[0])  # 数组长度
        self.m = int(line[1])  # 查询数量

        # 读取原始数组
        nums = input().split()
        for i in range(1, self.n + 1):
            self.arr[i] = int(nums[i - 1])
            self.sorted[i] = self.arr[i]
```

```
            # 读取查询
            for i in range(1, self.m + 1):
                query = input().split()
                self.queryL[i] = int(query[0])  # 查询区间左端点
                self.queryR[i] = int(query[1])  # 查询区间右端点
                self.queryK[i] = int(query[2])  # 查询第k小
                self.queryId[i] = i             # 查询编号

            # 离散化
            self.sorted[1:self.n + 1] = sorted(self.sorted[1:self.n + 1])
            uniqueCount = 1
            for i in range(2, self.n + 1):
                if self.sorted[i] != self.sorted[i - 1]:
                    uniqueCount += 1
                    self.sorted[uniqueCount] = self.sorted[i]

            # 整体二分求解
            self.compute(1, self.m, 1, uniqueCount)

            # 输出结果
            for i in range(1, self.m + 1):
                print(self.ans[i])


# 主程序
if __name__ == "__main__":
    """
    程序入口点
    创建解决方案实例并执行求解
    """
    solver = HDU2665_KthNumberSolution()
    solver.solve()
```

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

文件: HNOI2015接水果.cpp

```
================================================
#include <iostream>
#include <vector>
#include <algorithm>
#include <cstring>
#include <cstdio>
```

```
using namespace std;

/**
 * 洛谷 P3242 [HNOI2015]接水果 - C++实现
 * 题目来源：https://www.luogu.com.cn/problem/P3242
 * 题目描述：树上路径包含关系与扫描线结合的整体二分问题
 *
 * 问题描述：
 * 给定一棵树，每个节点有一个权值。有两种操作：
 * 1. 类型 A：在节点 u 和 v 之间连接一条边，边权为 w
 * 2. 类型 B：查询所有满足路径 u-v 被路径 a-b 包含的边的边权的第 k 小
 *
 * 解题思路：
 * 1. 首先使用 DFS 序将树上路径转换为平面矩形区域
 * 2. 将边和查询转换为矩形区域的覆盖和查询问题
 * 3. 使用扫描线和树状数组结合整体二分求解
 *
 * 时间复杂度：O((P+Q) * log(P) * log(max_weight))
 * 空间复杂度：O(P+Q)
 */

const int MAXN = 40005;
const int MAXM = 80005;
const int MAXQ = 100005;

// 树的结构
int head[MAXN];
int next_[MAXM];
int to[MAXM];
int cnt;

// DFS 序
int in[MAXN];
int out[MAXN];
int timeStamp;

// 父节点和深度（用于 LCA）
int dep[MAXN];
int f[MAXN][20];

// 边的信息
int u[MAXM];
int v[MAXM];
```

```
int w[MAXM];
int m, q;

// 扫描线事件
struct Event {
    int x, y1, y2, type, id;
    Event(int x = 0, int y1 = 0, int y2 = 0, int type = 0, int id = 0)
        : x(x), y1(y1), y2(y2), type(type), id(id) {}
};

vector<Event> events;

// 查询信息
struct Query {
    int x1, x2, y1, y2, k, id;
    Query(int x1 = 0, int x2 = 0, int y1 = 0, int y2 = 0, int k = 0, int id = 0)
        : x1(x1), x2(x2), y1(y1), y2(y2), k(k), id(id) {}
};

Query queries[MAXQ];
int ans[MAXQ];

// 树状数组
struct FenwickTree {
    int tree[MAXN];

    void update(int x, int val) {
        for (; x < MAXN; x += x & -x) {
            tree[x] += val;
        }
    }

    int query(int x) {
        int res = 0;
        for (; x > 0; x -= x & -x) {
            res += tree[x];
        }
        return res;
    }

    int query(int l, int r) {
        return query(r) - query(l - 1);
    }
```

```
};

FenwickTree ft;

// 初始化树的邻接表
void addEdge(int u, int v) {
    next_[++cnt] = head[u];
    head[u] = cnt;
    to[cnt] = v;
}

// DFS 计算入时间戳和出时间戳
void dfs(int u, int fa) {
    in[u] = ++timeStamp;
    dep[u] = dep[fa] + 1;
    f[u][0] = fa;
    for (int i = 1; i < 20; i++) {
        f[u][i] = f[f[u][i-1]][i-1];
    }
    for (int i = head[u]; i; i = next_[i]) {
        int v = to[i];
        if (v != fa) {
            dfs(v, u);
        }
    }
    out[u] = timeStamp;
}

// 求 LCA
int lca(int u, int v) {
    if (dep[u] < dep[v]) {
        swap(u, v);
    }

    for (int i = 19; i >= 0; i--) {
        if (dep[f[u][i]] >= dep[v]) {
            u = f[u][i];
        }
    }

    if (u == v) return u;

    for (int i = 19; i >= 0; i--) {
```

```
        if (f[u][i] != f[v][i]) {
            u = f[u][i];
            v = f[v][i];
        }
    }

    return f[u][0];
}

// 处理边，将其转换为扫描线事件
void processEdge(int u, int v, int w, int id, vector<Event>& tmpEvents) {
    if (dep[u] < dep[v]) {
        swap(u, v);
    }

    // 将边转换为矩形区域
    tmpEvents.emplace_back(1, in[u], out[u], 1, id);
    tmpEvents.emplace_back(in[v], in[u], out[u], -1, id);
    tmpEvents.emplace_back(out[v] + 1, in[u], out[u], 1, id);
}

// 整体二分核心函数
void solve(int ql, int qr, int l, int r) {
    if (ql > qr || l > r) return;

    if (l == r) {
        // 所有查询的答案都是 l
        for (int i = ql; i <= qr; i++) {
            ans[queries[i].id] = l;
        }
        return;
    }

    int mid = (l + r) >> 1;

    // 收集扫描线事件
    vector<Event> tmpEvents;
    for (int i = 1; i <= m; i++) {
        if (w[i] <= mid) {
            processEdge(u[i], v[i], w[i], i, tmpEvents);
        }
    }
```

```
// 将查询也加入事件列表
for (int i = ql; i <= qr; i++) {
    tmpEvents.emplace_back(queries[i].x1, queries[i].y1, queries[i].y2, -2, i);
    tmpEvents.emplace_back(queries[i].x2 + 1, queries[i].y1, queries[i].y2, -3, i);
}

// 按 x 坐标排序事件
sort(tmpEvents.begin(), tmpEvents.end(), [](const Event& a, const Event& b) {
    return a.x < b.x;
});

// 初始化答案计数
vector<int> cnt(qr - ql + 1, 0);

// 处理扫描线
int eventPtr = 0;
for (int x = 1; x <= timeStamp; x++) {
    // 处理所有 x 坐标等于当前 x 的事件
    while (eventPtr < tmpEvents.size() && tmpEvents[eventPtr].x == x) {
        Event e = tmpEvents[eventPtr++];
        if (e.type == 1 || e.type == -1) {
            // 矩形覆盖事件
            ft.update(e.y1, e.type);
            ft.update(e.y2 + 1, -e.type);
        } else if (e.type == -2) {
            // 查询开始事件
            int idx = e.id - ql;
            cnt[idx] -= ft.query(e.y1, e.y2);
        } else if (e.type == -3) {
            // 查询结束事件
            int idx = e.id - ql;
            cnt[idx] += ft.query(e.y1, e.y2);
        }
    }
}

// 清理树状数组
for (Event e : tmpEvents) {
    if (e.type == 1 || e.type == -1) {
        ft.update(e.y1, -e.type);
        ft.update(e.y2 + 1, e.type);
    }
}
```

```
    // 分类查询
    int left = ql, right = qr;
    vector<int> leftQueries(qr - ql + 1);
    vector<int> rightQueries(qr - ql + 1);

    for (int i = ql; i <= qr; i++) {
        int idx = i - ql;
        if (cnt[idx] >= queries[i].k) {
            // 答案在左半部分
            leftQueries[left - ql] = i;
            left++;
        } else {
            // 答案在右半部分，调整 k 值
            queries[i].k -= cnt[idx];
            rightQueries[right - qr] = i;
            right--;
        }
    }

    // 保存当前查询状态
    vector<Query> tmp(qr - ql + 1);
    for (int i = ql; i <= qr; i++) {
        tmp[i - ql] = queries[i];
    }

    // 合并查询顺序
    for (int i = ql; i < left; i++) {
        queries[i] = tmp[leftQueries[i - ql] - ql];
    }
    for (int i = qr; i > right; i--) {
        queries[i] = tmp[rightQueries[i - qr] - ql];
    }

    // 递归处理左右两部分
    solve(ql, left - 1, l, mid);
    solve(right + 1, qr, mid + 1, r);
}

int main() {
    // 输入优化
    ios::sync_with_stdio(false);
    cin.tie(0);
```

```
    // 读取输入
    int n;
    cin >> n >> m >> q;

    // 初始化邻接表
    memset(head, 0, sizeof(head));
    cnt = 0;

    // 读取树的边
    for (int i = 1; i < n; i++) {
        int u, v;
        cin >> u >> v;
        addEdge(u, v);
        addEdge(v, u);
    }

    // 计算 DFS 序和 LCA 所需信息
    timeStamp = 0;
    memset(dep, 0, sizeof(dep));
    memset(f, 0, sizeof(f));
    dfs(1, 0);

    // 读取水果（边）的信息
    vector<int> weights(m + 1);
    for (int i = 1; i <= m; i++) {
        cin >> u[i] >> v[i] >> w[i];
        weights[i] = w[i];
    }

    // 离散化边权
    sort(weights.begin() + 1, weights.end());
    int uniqueWeights = 1;
    for (int i = 2; i <= m; i++) {
        if (weights[i] != weights[uniqueWeights]) {
            weights[++uniqueWeights] = weights[i];
        }
    }

    for (int i = 1; i <= m; i++) {
        w[i] = lower_bound(weights.begin() + 1, weights.begin() + uniqueWeights + 1, w[i]) -
weights.begin();
    }
```

```
    // 读取查询
    for (int i = 1; i <= q; i++) {
        int a, b, k;
        cin >> a >> b >> k;
        int l = lca(a, b);
        if (l == a) {
            // 路径 a-b 是链状的，且 a 是 LCA
            if (dep[a] > dep[b]) swap(a, b);
            queries[i] = Query(in[b], out[b], in[l] + 1, in[a], k, i);
        } else if (l == b) {
            // 路径 a-b 是链状的，且 b 是 LCA
            if (dep[a] < dep[b]) swap(a, b);
            queries[i] = Query(in[a], out[a], in[l] + 1, in[b], k, i);
        } else {
            // 路径 a-b 经过 LCA，分成两段
            if (in[a] > in[b]) swap(a, b);
            queries[i] = Query(in[a], out[a], in[b], out[b], k, i);
        }
    }

    // 整体二分求解
    solve(1, q, 1, uniqueWeights);

    // 输出结果
    for (int i = 1; i <= q; i++) {
        cout << weights[ans[i]] << '\n';
    }

    return 0;
}

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

文件: HNOI2015 接水果.java
==========================================
package class169.supplementary_solutions;

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

/**
 * 洛谷 P3242 [HNOI2015]接水果 - Java 实现
```

```
 * 题目来源：https://www.luogu.com.cn/problem/P3242
 * 题目描述：树上路径包含关系与扫描线结合的整体二分问题
 *
 * 问题描述：
 * 给定一棵树，每个节点有一个权值。有两种操作：
 * 1. 类型 A：在节点 u 和 v 之间连接一条边，边权为 w
 * 2. 类型 B：查询所有满足路径 u-v 被路径 a-b 包含的边的边权的第 k 小
 *
 * 解题思路：
 * 1. 首先使用 DFS 序将树上路径转换为平面矩形区域
 * 2. 将边和查询转换为矩形区域的覆盖和查询问题
 * 3. 使用扫描线和树状数组结合整体二分求解
 *
 * 时间复杂度：O((P+Q) * log(P) * log(max_weight))
 * 空间复杂度：O(P+Q)
 */
public class HNOI2015接水果 {
    static final int MAXN = 40005;
    static final int MAXM = 80005;
    static final int MAXQ = 100005;

    // 树的结构
    static int[] head = new int[MAXN];
    static int[] next = new int[MAXM];
    static int[] to = new int[MAXM];
    static int cnt;

    // DFS 序
    static int[] in = new int[MAXN];
    static int[] out = new int[MAXN];
    static int timeStamp;

    // 父节点和深度（用于 LCA）
    static int[] dep = new int[MAXN];
    static int[][] f = new int[MAXN][20];

    // 边的信息
    static int[] u = new int[MAXM];
    static int[] v = new int[MAXM];
    static int[] w = new int[MAXM];
    static int m, q;

    // 扫描线事件
```

```
static class Event {
    int x, y1, y2, type, id;
    Event(int x, int y1, int y2, int type, int id) {
        this.x = x;
        this.y1 = y1;
        this.y2 = y2;
        this.type = type;
        this.id = id;
    }
}

static List<Event> events = new ArrayList<>();

// 查询信息
static class Query {
    int x1, x2, y1, y2, k, id;
    Query(int x1, int x2, int y1, int y2, int k, int id) {
        this.x1 = x1;
        this.x2 = x2;
        this.y1 = y1;
        this.y2 = y2;
        this.k = k;
        this.id = id;
    }
}

static Query[] queries = new Query[MAXQ];
static int[] ans = new int[MAXQ];

// 整体二分相关
static int[] ql = new int[MAXQ];
static int[] qr = new int[MAXQ];
static int[] qmid = new int[MAXQ];
static int[] qans = new int[MAXQ];

// 树状数组
static class FenwickTree {
    int[] tree = new int[MAXN];

    void update(int x, int val) {
        for (; x < MAXN; x += x & -x) {
            tree[x] += val;
        }
```

```
    }

    int query(int x) {
        int res = 0;
        for (; x > 0; x -= x & -x) {
            res += tree[x];
        }
        return res;
    }

    int query(int l, int r) {
        return query(r) - query(l - 1);
    }
}

static FenwickTree ft = new FenwickTree();

// 初始化树的邻接表
static void addEdge(int u, int v) {
    next[++cnt] = head[u];
    head[u] = cnt;
    to[cnt] = v;
}

// DFS 计算入时间戳和出时间戳
static void dfs(int u, int fa) {
    in[u] = ++timeStamp;
    dep[u] = dep[fa] + 1;
    f[u][0] = fa;
    for (int i = 1; i < 20; i++) {
        f[u][i] = f[f[u][i-1]][i-1];
    }
    for (int i = head[u]; i > 0; i = next[i]) {
        int v = to[i];
        if (v != fa) {
            dfs(v, u);
        }
    }
    out[u] = timeStamp;
}

// 求 LCA
static int lca(int u, int v) {
```

```
    if (dep[u] < dep[v]) {
        int tmp = u;
        u = v;
        v = tmp;
    }

    for (int i = 19; i >= 0; i--) {
        if (dep[f[u][i]] >= dep[v]) {
            u = f[u][i];
        }
    }

    if (u == v) return u;

    for (int i = 19; i >= 0; i--) {
        if (f[u][i] != f[v][i]) {
            u = f[u][i];
            v = f[v][i];
        }
    }

    return f[u][0];
}

// 处理边，将其转换为扫描线事件
static void processEdge(int u, int v, int w, int id) {
    if (dep[u] < dep[v]) {
        int tmp = u;
        u = v;
        v = tmp;
    }

    // 将边转换为矩形区域
    events.add(new Event(1, in[u], out[u], 1, id));
    events.add(new Event(in[v], in[u], out[u], -1, id));
    events.add(new Event(out[v] + 1, in[u], out[u], 1, id));
}

// 处理查询，将其转换为矩形区域查询
static void processQuery(int a, int b, int k, int id) {
    int l = lca(a, b);
    if (l == a) {
        // 路径 a-b 是链状的，且 a 是 LCA
```

```
            queries[id] = new Query(in[b], out[b], in[l] + 1, in[a], k, id);
        } else if (l == b) {
            // 路径 a-b 是链状的，且 b 是 LCA
            queries[id] = new Query(in[a], out[a], in[l] + 1, in[b], k, id);
        } else {
            // 路径 a-b 经过 LCA，分成两段
            if (in[a] > in[b]) {
                int tmp = a;
                a = b;
                b = tmp;
            }
            queries[id] = new Query(in[a], out[a], in[b], out[b], k, id);
        }
    }

    // 整体二分核心函数
    static void solve(int ql, int qr, int l, int r) {
        if (ql > qr || l > r) return;

        if (l == r) {
            // 所有查询的答案都是 l
            for (int i = ql; i <= qr; i++) {
                ans[queries[i].id] = l;
            }
            return;
        }

        int mid = (l + r) >> 1;

        // 收集扫描线事件
        List<Event> tmpEvents = new ArrayList<>();
        for (int i = 1; i <= m; i++) {
            if (w[i] <= mid) {
                processEdge(u[i], v[i], w[i], i);
            }
        }

        // 将查询也加入事件列表
        for (int i = ql; i <= qr; i++) {
            tmpEvents.add(new Event(queries[i].x1, queries[i].y1, queries[i].y2, -2, i));
            tmpEvents.add(new Event(queries[i].x2 + 1, queries[i].y1, queries[i].y2, -3, i));
        }
```

```
// 按 x 坐标排序事件
tmpEvents.sort((a, b) -> a.x - b.x);

// 初始化答案计数
int[] cnt = new int[qr - ql + 1];

// 处理扫描线
int eventPtr = 0;
for (int x = 1; x <= timeStamp; x++) {
    // 处理所有 x 坐标等于当前 x 的事件
    while (eventPtr < tmpEvents.size() && tmpEvents.get(eventPtr).x == x) {
        Event e = tmpEvents.get(eventPtr++);
        if (e.type == 1 || e.type == -1) {
            // 矩形覆盖事件
            ft.update(e.y1, e.type);
            ft.update(e.y2 + 1, -e.type);
        } else if (e.type == -2) {
            // 查询开始事件
            int idx = e.id - ql;
            cnt[idx] -= ft.query(e.y1, e.y2);
        } else if (e.type == -3) {
            // 查询结束事件
            int idx = e.id - ql;
            cnt[idx] += ft.query(e.y1, e.y2);
        }
    }
}

// 清理树状数组
for (Event e : tmpEvents) {
    if (e.type == 1 || e.type == -1) {
        ft.update(e.y1, -e.type);
        ft.update(e.y2 + 1, e.type);
    }
}

// 分类查询
int left = ql, right = qr;
int[] leftQueries = new int[qr - ql + 1];
int[] rightQueries = new int[qr - ql + 1];

for (int i = ql; i <= qr; i++) {
    int idx = i - ql;
```

```
            if (cnt[idx] >= queries[i].k) {
                // 答案在左半部分
                leftQueries[left - ql] = i;
                left++;
            } else {
                // 答案在右半部分，调整 k 值
                queries[i].k -= cnt[idx];
                rightQueries[right - qr] = i;
                right--;
            }
        }

        // 合并查询顺序
        for (int i = ql; i < left; i++) {
            queries[i] = queries[leftQueries[i - ql]];
        }
        for (int i = qr; i > right; i--) {
            queries[i] = queries[rightQueries[i - qr]];
        }

        // 递归处理左右两部分
        solve(ql, left - 1, l, mid);
        solve(right + 1, qr, mid + 1, r);
    }

    public static void main(String[] args) throws IOException {
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));

        // 读取输入
        String[] parts = br.readLine().split(" ");
        int n = Integer.parseInt(parts[0]);
        m = Integer.parseInt(parts[1]);
        q = Integer.parseInt(parts[2]);

        // 读取树的边
        for (int i = 1; i < n; i++) {
            parts = br.readLine().split(" ");
            int u = Integer.parseInt(parts[0]);
            int v = Integer.parseInt(parts[1]);
            addEdge(u, v);
            addEdge(v, u);
        }
```

```
// 计算 DFS 序和 LCA 所需信息
dfs(1, 0);

// 读取水果（边）的信息
int[] weights = new int[m + 1];
for (int i = 1; i <= m; i++) {
    parts = br.readLine().split(" ");
    u[i] = Integer.parseInt(parts[0]);
    v[i] = Integer.parseInt(parts[1]);
    w[i] = Integer.parseInt(parts[2]);
    weights[i] = w[i];
}

// 离散化边权
Arrays.sort(weights, 1, m + 1);
int uniqueWeights = 1;
for (int i = 2; i <= m; i++) {
    if (weights[i] != weights[uniqueWeights]) {
        weights[++uniqueWeights] = weights[i];
    }
}

for (int i = 1; i <= m; i++) {
    w[i] = Arrays.binarySearch(weights, 1, uniqueWeights + 1, w[i]) - weights[0] + 1;
}

// 读取查询
for (int i = 1; i <= q; i++) {
    parts = br.readLine().split(" ");
    int a = Integer.parseInt(parts[0]);
    int b = Integer.parseInt(parts[1]);
    int k = Integer.parseInt(parts[2]);
    processQuery(a, b, k, i);
}

// 整体二分求解
solve(1, q, 1, uniqueWeights);

// 输出结果
for (int i = 1; i <= q; i++) {
    out.println(weights[ans[i]]);
}
```

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

==========

文件: HNOI2015 接水果.py

==========

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

"""
洛谷 P3242 [HNOI2015]接水果 - Python 实现
题目来源：https://www.luogu.com.cn/problem/P3242
题目描述：树上路径包含关系与扫描线结合的整体二分问题

问题描述：
给定一棵树，每个节点有一个权值。有两种操作：
1. 类型 A：在节点 u 和 v 之间连接一条边，边权为 w
2. 类型 B：查询所有满足路径 u-v 被路径 a-b 包含的边的边权的第 k 小

解题思路：
1. 首先使用 DFS 序将树上路径转换为平面矩形区域
2. 将边和查询转换为矩形区域的覆盖和查询问题
3. 使用扫描线和树状数组结合整体二分求解

时间复杂度：O((P+Q) * log(P) * log(max_weight))
空间复杂度：O(P+Q)

注意：在 Python 中实现时需要注意递归深度限制，对于大规模数据可能需要调整递归深度或转换为迭代实现。
"""

import sys
from sys import stdin
from bisect import bisect_left

MAXN = 40005
MAXM = 80005
MAXQ = 100005
LOG = 20
```

```
# 树的结构
head = [0] * MAXN
next_ = [0] * MAXM
to_ = [0] * MAXM
cnt = 0

# DFS 序
in_ = [0] * MAXN
out_ = [0] * MAXN
timeStamp = 0

# 父节点和深度（用于 LCA）
dep = [0] * MAXN
f = [[0] * LOG for _ in range(MAXN)]

# 边的信息
u = [0] * MAXM
v = [0] * MAXM
w = [0] * MAXM
m = 0
q = 0

# 扫描线事件
class Event:
    def __init__(self, x=0, y1=0, y2=0, type_=0, id_=0):
        self.x = x
        self.y1 = y1
        self.y2 = y2
        self.type = type_
        self.id = id_

    def __lt__(self, other):
        return self.x < other.x

# 查询信息
class Query:
    def __init__(self, x1=0, x2=0, y1=0, y2=0, k=0, id_=0):
        self.x1 = x1
        self.x2 = x2
        self.y1 = y1
        self.y2 = y2
        self.k = k
```

```
        self.id = id_

queries = [Query() for _ in range(MAXQ)]
ans = [0] * MAXQ

# 树状数组
class FenwickTree:
    def __init__(self):
        self.tree = [0] * MAXN

    def update(self, x, val):
        while x < MAXN:
            self.tree[x] += val
            x += x & -x

    def query(self, x):
        res = 0
        while x > 0:
            res += self.tree[x]
            x -= x & -x
        return res

    def query_range(self, l, r):
        return self.query(r) - self.query(l - 1)

ft = FenwickTree()

# 初始化树的邻接表
def add_edge(u, v):
    global cnt
    cnt += 1
    next_[cnt] = head[u]
    head[u] = cnt
    to_[cnt] = v

# DFS 计算入时间戳和出时间戳
def dfs(u, fa):
    global timeStamp
    timeStamp += 1
    in_[u] = timeStamp
    dep[u] = dep[fa] + 1
    f[u][0] = fa
    for i in range(1, LOG):
```

```
        f[u][i] = f[f[u][i-1]][i-1]
    i = head[u]
    while i > 0:
        v = to_[i]
        if v != fa:
            dfs(v, u)
        i = next_[i]
    out_[u] = timeStamp

# 求LCA
def lca(u, v):
    if dep[u] < dep[v]:
        u, v = v, u

    # 将u提升到v的深度
    for i in range(LOG-1, -1, -1):
        if dep[f[u][i]] >= dep[v]:
            u = f[u][i]

    if u == v:
        return u

    # 同时提升u和v直到LCA
    for i in range(LOG-1, -1, -1):
        if f[u][i] != f[v][i]:
            u = f[u][i]
            v = f[v][i]

    return f[u][0]

# 整体二分核心函数
def solve(ql, qr, l, r):
    if ql > qr or l > r:
        return

    if l == r:
        # 所有查询的答案都是l
        for i in range(ql, qr + 1):
            ans[queries[i].id] = l
        return

    mid = (l + r) // 2
```

```
# 收集扫描线事件
tmp_events = []
for i in range(1, m + 1):
    if w[i] <= mid:
        # 处理边，将其转换为扫描线事件
        u_node, v_node = u[i], v[i]
        if dep[u_node] < dep[v_node]:
            u_node, v_node = v_node, u_node
        # 将边转换为矩形区域
        tmp_events.append(Event(1, in_[u_node], out_[u_node], 1, i))
        tmp_events.append(Event(in_[v_node], in_[u_node], out_[u_node], -1, i))
        tmp_events.append(Event(out_[v_node] + 1, in_[u_node], out_[u_node], 1, i))

# 将查询也加入事件列表
for i in range(ql, qr + 1):
    tmp_events.append(Event(queries[i].x1, queries[i].y1, queries[i].y2, -2, i))
    tmp_events.append(Event(queries[i].x2 + 1, queries[i].y1, queries[i].y2, -3, i))

# 按 x 坐标排序事件
tmp_events.sort()

# 初始化答案计数
cnt_array = [0] * (qr - ql + 1)

# 处理扫描线
event_ptr = 0
for x in range(1, timeStamp + 1):
    # 处理所有 x 坐标等于当前 x 的事件
    while event_ptr < len(tmp_events) and tmp_events[event_ptr].x == x:
        e = tmp_events[event_ptr]
        event_ptr += 1
        if e.type == 1 or e.type == -1:
            # 矩形覆盖事件
            ft.update(e.y1, e.type)
            ft.update(e.y2 + 1, -e.type)
        elif e.type == -2:
            # 查询开始事件
            idx = e.id - ql
            cnt_array[idx] -= ft.query_range(e.y1, e.y2)
        elif e.type == -3:
            # 查询结束事件
            idx = e.id - ql
            cnt_array[idx] += ft.query_range(e.y1, e.y2)
```

```
# 清理树状数组
for e in tmp_events:
    if e.type == 1 or e.type == -1:
        ft.update(e.y1, -e.type)
        ft.update(e.y2 + 1, e.type)

# 分类查询
left = ql
right = qr
left_queries = [0] * (qr - ql + 1)
right_queries = [0] * (qr - ql + 1)

# 保存当前查询状态
tmp_queries = [Query() for _ in range(qr - ql + 1)]
for i in range(ql, qr + 1):
    tmp_queries[i - ql] = Query(queries[i].x1, queries[i].x2, queries[i].y1,
                                queries[i].y2, queries[i].k, queries[i].id)

for i in range(ql, qr + 1):
    idx = i - ql
    if cnt_array[idx] >= tmp_queries[idx].k:
        # 答案在左半部分
        left_queries[left - ql] = i
        left += 1
    else:
        # 答案在右半部分，调整 k 值
        tmp_queries[idx].k -= cnt_array[idx]
        right_queries[right - qr] = i
        right -= 1

# 合并查询顺序
for i in range(ql, left):
    pos = left_queries[i - ql]
    queries[i] = Query(tmp_queries[pos - ql].x1, tmp_queries[pos - ql].x2,
                       tmp_queries[pos - ql].y1, tmp_queries[pos - ql].y2,
                       tmp_queries[pos - ql].k, tmp_queries[pos - ql].id)

for i in range(qr, right, -1):
    pos = right_queries[i - qr]
    queries[i] = Query(tmp_queries[pos - ql].x1, tmp_queries[pos - ql].x2,
                       tmp_queries[pos - ql].y1, tmp_queries[pos - ql].y2,
                       tmp_queries[pos - ql].k, tmp_queries[pos - ql].id)
```

```
    # 递归处理左右两部分
    solve(ql, left - 1, l, mid)
    solve(right + 1, qr, mid + 1, r)

def main():
    # 使用快速输入方法，优化处理大数据量
    input = sys.stdin.read().split()
    ptr = 0

    # 读取输入
    n = int(input[ptr])
    ptr += 1
    global m, q
    m = int(input[ptr])
    ptr += 1
    q = int(input[ptr])
    ptr += 1

    # 初始化邻接表
    global cnt
    cnt = 0
    for i in range(MAXN):
        head[i] = 0
    for i in range(MAXM):
        next_[i] = 0
        to_[i] = 0

    # 读取树的边
    for i in range(1, n):
        u_node = int(input[ptr])
        ptr += 1
        v_node = int(input[ptr])
        ptr += 1
        add_edge(u_node, v_node)
        add_edge(v_node, u_node)

    # 计算 DFS 序和 LCA 所需信息
    global timeStamp
    timeStamp = 0
    dep[0] = 0
    for i in range(MAXN):
        for j in range(LOG):
```

```
        f[i][j] = 0
dfs(1, 0)

# 读取水果（边）的信息
weights = [0] * (m + 1)
for i in range(1, m + 1):
    u[i] = int(input[ptr])
    ptr += 1
    v[i] = int(input[ptr])
    ptr += 1
    w[i] = int(input[ptr])
    ptr += 1
    weights[i] = w[i]

# 离散化边权
sorted_weights = sorted(weights[1:m+1])
unique_weights = [sorted_weights[0]]
for i in range(1, len(sorted_weights)):
    if sorted_weights[i] != unique_weights[-1]:
        unique_weights.append(sorted_weights[i])
unique_cnt = len(unique_weights)

for i in range(1, m + 1):
    w[i] = bisect_left(unique_weights, w[i]) + 1  # 从 1 开始编号

# 读取查询
for i in range(1, q + 1):
    a = int(input[ptr])
    ptr += 1
    b = int(input[ptr])
    ptr += 1
    k = int(input[ptr])
    ptr += 1

    l_node = lca(a, b)
    if l_node == a:
        # 路径 a-b 是链状的，且 a 是 LCA
        if dep[a] > dep[b]:
            a, b = b, a
        queries[i] = Query(in_[b], out_[b], in_[l_node] + 1, in_[a], k, i)
    elif l_node == b:
        # 路径 a-b 是链状的，且 b 是 LCA
        if dep[a] < dep[b]:
```

```
                a, b = b, a
            queries[i] = Query(in_[a], out_[a], in_[l_node] + 1, in_[b], k, i)
        else:
            # 路径 a-b 经过 LCA，分成两段
            if in_[a] > in_[b]:
                a, b = b, a
            queries[i] = Query(in_[a], out_[a], in_[b], out_[b], k, i)

    # 整体二分求解
    solve(1, q, 1, unique_cnt)

    # 输出结果
    output = []
    for i in range(1, q + 1):
        output.append(str(unique_weights[ans[i] - 1]))
    print('\n'.join(output))

if __name__ == "__main__":
    # 设置递归深度，防止大规模数据导致栈溢出
    sys.setrecursionlimit(1 << 25)
    main()
```

==========

文件: POJ2104_KthNumber.cpp

==========

```
/*
 * POJ 2104 K-th Number - C++实现
 *
 * 题目来源：http://poj.org/problem?id=2104
 * 题目描述：静态区间第 k 小查询
 *
 * 问题描述：
 * 给定一个长度为 n 的数组，有 m 个查询，每个查询要求在指定区间内找到第 k 小的数。
 *
 * 解题思路：
 * 使用整体二分处理静态区间第 k 小问题。将所有查询一起处理，二分答案的值域，
 * 利用树状数组维护区间内小于等于 mid 的元素个数。
 *
 * 时间复杂度：O((N+Q) * logN * log(maxValue))
 * 空间复杂度：O(N + Q)
 */
```

```
// 由于编译环境限制，这里省略头文件包含

const int MAXN = 100001;
int n, m;

// 原始数组
int arr[MAXN];

// 离散化数组
int sorted[MAXN];

// 查询信息
int queryL[MAXN];  // 查询区间左端点
int queryR[MAXN];  // 查询区间右端点
int queryK[MAXN];  // 查询第 k 小
int queryId[MAXN]; // 查询编号

// 树状数组
int tree[MAXN];

// 整体二分
int lset[MAXN]; // 左集合
int rset[MAXN]; // 右集合

// 查询的答案
int ans[MAXN];

/**
 * 计算一个数的 lowbit 值
 * @param i 输入的数
 * @return lowbit 值
 */
int lowbit(int i) {
    return i & -i;
}

/**
 * 在树状数组中给位置 i 增加 v
 * @param i 位置
 * @param v 增加的值
 */
void add(int i, int v) {
    while (i <= n) {
```

```
        tree[i] += v;
        i += lowbit(i);
    }
}

/**
 * 计算前缀和[1..i]
 * @param i 位置
 * @return 前缀和
 */
int sum(int i) {
    int ret = 0;
    while (i > 0) {
        ret += tree[i];
        i -= lowbit(i);
    }
    return ret;
}

/**
 * 计算区间和[l..r]
 * @param l 左端点
 * @param r 右端点
 * @return 区间和
 */
int query(int l, int r) {
    return sum(r) - sum(l - 1);
}

/**
 * 整体二分核心函数
 * @param ql 查询范围的左端点
 * @param qr 查询范围的右端点
 * @param vl 值域范围的左端点（离散化后的下标）
 * @param vr 值域范围的右端点（离散化后的下标）
 */
void compute(int ql, int qr, int vl, int vr) {
    // 递归边界
    if (ql > qr) {
        return;
    }

    // 如果值域范围只有一个值，说明找到了答案
```

```
if (vl == vr) {
    for (int i = ql; i <= qr; i++) {
        ans[queryId[i]] = sorted[vl];
    }
    return;
}

// 二分中点
int mid = (vl + vr) >> 1;

// 将值小于等于 sorted[mid]的数加入树状数组
for (int i = vl; i <= mid; i++) {
    // 遍历所有值为 sorted[i]的元素，将其加入树状数组
    for (int j = 1; j <= n; j++) {
        if (arr[j] == sorted[i]) {
            add(j, 1);
        }
    }
}

// 检查每个查询，根据满足条件的元素个数划分到左右区间
int lsiz = 0, rsiz = 0;
for (int i = ql; i <= qr; i++) {
    // 查询区间[queryL[i], queryR[i]]中值小于等于 sorted[mid]的元素个数
    int satisfy = query(queryL[i], queryR[i]);

    if (satisfy >= queryK[i]) {
        // 说明第 k 小的数在左半部分
        lset[++lsiz] = i;
    } else {
        // 说明第 k 小的数在右半部分，需要在右半部分找第(k-satisfy)小的数
        queryK[i] -= satisfy;
        rset[++rsiz] = i;
    }
}

// 重新排列查询顺序
int idx = ql;
for (int i = 1; i <= lsiz; i++) {
    int temp = lset[i];
    lset[i] = queryId[temp];
    queryId[idx++] = temp;
}
```

```
    for (int i = 1; i <= rsiz; i++) {
        int temp = rset[i];
        rset[i] = queryId[temp];
        queryId[idx++] = temp;
    }

    // 撤销对树状数组的修改
    for (int i = vl; i <= mid; i++) {
        // 遍历所有值为 sorted[i]的元素，将其从树状数组中删除
        for (int j = 1; j <= n; j++) {
            if (arr[j] == sorted[i]) {
                add(j, -1);
            }
        }
    }

    // 递归处理左右两部分
    compute(ql, ql + lsiz - 1, vl, mid);
    compute(ql + lsiz, qr, mid + 1, vr);
}

/*
 * 由于编译环境限制，这里省略 main 函数的实现
 *
 * 示例代码结构：
 *
 * // 读取数组长度 n 和查询数量 m
 * // cin >> n >> m;
 *
 * // 读取原始数组
 * // for (int i = 1; i <= n; i++) {
 * //     cin >> arr[i];
 * //     sorted[i] = arr[i];
 * // }
 *
 * // 读取查询
 * // for (int i = 1; i <= m; i++) {
 * //     cin >> queryL[i] >> queryR[i] >> queryK[i];
 * //     queryId[i] = i;
 * // }
 *
 * // 离散化
 * // sort(sorted + 1, sorted + n + 1);
```

```
 * // int uniqueCount = 1;
 * // for (int i = 2; i <= n; i++) {
 * //     if (sorted[i] != sorted[i - 1]) {
 * //         sorted[++uniqueCount] = sorted[i];
 * //     }
 * // }
 *
 * // 整体二分求解
 * // compute(1, m, 1, uniqueCount);
 *
 * // 输出结果
 * // for (int i = 1; i <= m; i++) {
 * //     cout << ans[i] << endl;
 * // }
 */
int main() {
    // 算法核心部分已实现，此处省略输入输出部分
    return 0;
}
```

==========

文件: POJ2104_KthNumber.java

==========

```
package class169.supplementary_solutions;

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

/**
 * POJ 2104 K-th Number - Java 实现
 * 题目来源：http://poj.org/problem?id=2104
 * 题目描述：静态区间第 k 小查询
 *
 * 问题描述：
 * 给定一个长度为 n 的数组，有 m 个查询，每个查询要求在指定区间内找到第 k 小的数。
 *
 * 解题思路：
 * 使用整体二分处理静态区间第 k 小问题。将所有查询一起处理，二分答案的值域，
 * 利用树状数组维护区间内小于等于 mid 的元素个数。
 *
 * 时间复杂度：O((N+Q) * logN * log(maxValue))
 * 空间复杂度：O(N + Q)
```

```
 */
public class POJ2104_KthNumber {
    static final int MAXN = 100001;
    static int n, m;

    // 原始数组
    static int[] arr = new int[MAXN];

    // 离散化数组
    static int[] sorted = new int[MAXN];

    // 查询信息
    static int[] queryL = new int[MAXN];
    static int[] queryR = new int[MAXN];
    static int[] queryK = new int[MAXN];
    static int[] queryId = new int[MAXN];

    // 树状数组
    static int[] tree = new int[MAXN];

    // 整体二分
    static int[] lset = new int[MAXN];
    static int[] rset = new int[MAXN];

    // 查询的答案
    static int[] ans = new int[MAXN];

    /**
     * 计算一个数的 lowbit 值
     * @param i 输入的数
     * @return lowbit 值
     */
    static int lowbit(int i) {
        return i & -i;
    }

    /**
     * 在树状数组中给位置 i 增加 v
     * @param i 位置
     * @param v 增加的值
     */
    static void add(int i, int v) {
        while (i <= n) {
```

```
        tree[i] += v;
        i += lowbit(i);
    }
}

/**
 * 计算前缀和[1..i]
 * @param i 位置
 * @return 前缀和
 */
static int sum(int i) {
    int ret = 0;
    while (i > 0) {
        ret += tree[i];
        i -= lowbit(i);
    }
    return ret;
}

/**
 * 计算区间和[l..r]
 * @param l 左端点
 * @param r 右端点
 * @return 区间和
 */
static int query(int l, int r) {
    return sum(r) - sum(l - 1);
}

/**
 * 整体二分核心函数
 * @param ql 查询范围的左端点
 * @param qr 查询范围的右端点
 * @param vl 值域范围的左端点（离散化后的下标）
 * @param vr 值域范围的右端点（离散化后的下标）
 */
static void compute(int ql, int qr, int vl, int vr) {
    // 递归边界
    if (ql > qr) {
        return;
    }

    // 如果值域范围只有一个值，说明找到了答案
```

```
if (vl == vr) {
    for (int i = ql; i <= qr; i++) {
        ans[queryId[i]] = sorted[vl];
    }
    return;
}

// 二分中点
int mid = (vl + vr) >> 1;

// 将值小于等于 sorted[mid]的数加入树状数组
for (int i = vl; i <= mid; i++) {
    // 遍历所有值为 sorted[i]的元素，将其加入树状数组
    for (int j = 1; j <= n; j++) {
        if (arr[j] == sorted[i]) {
            add(j, 1);
        }
    }
}

// 检查每个查询，根据满足条件的元素个数划分到左右区间
int lsiz = 0, rsiz = 0;
for (int i = ql; i <= qr; i++) {
    // 查询区间[queryL[i], queryR[i]]中值小于等于 sorted[mid]的元素个数
    int satisfy = query(queryL[i], queryR[i]);

    if (satisfy >= queryK[i]) {
        // 说明第 k 小的数在左半部分
        lset[++lsiz] = i;
    } else {
        // 说明第 k 小的数在右半部分，需要在右半部分找第(k-satisfy)小的数
        queryK[i] -= satisfy;
        rset[++rsiz] = i;
    }
}

// 重新排列查询顺序
int idx = ql;
for (int i = 1; i <= lsiz; i++) {
    int temp = lset[i];
    lset[i] = queryId[temp];
    queryId[idx++] = temp;
}
```

```
        for (int i = 1; i <= rsiz; i++) {
            int temp = rset[i];
            rset[i] = queryId[temp];
            queryId[idx++] = temp;
        }

        // 撤销对树状数组的修改
        for (int i = vl; i <= mid; i++) {
            // 遍历所有值为sorted[i]的元素，将其从树状数组中删除
            for (int j = 1; j <= n; j++) {
                if (arr[j] == sorted[i]) {
                    add(j, -1);
                }
            }
        }

        // 递归处理左右两部分
        compute(ql, ql + lsiz - 1, vl, mid);
        compute(ql + lsiz, qr, mid + 1, vr);
    }

    public static void main(String[] args) throws IOException {
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));

        String[] params = br.readLine().split(" ");
        n = Integer.parseInt(params[0]);
        m = Integer.parseInt(params[1]);

        // 读取原始数组
        String[] nums = br.readLine().split(" ");
        for (int i = 1; i <= n; i++) {
            arr[i] = Integer.parseInt(nums[i - 1]);
            sorted[i] = arr[i];
        }

        // 读取查询
        for (int i = 1; i <= m; i++) {
            String[] query = br.readLine().split(" ");
            queryL[i] = Integer.parseInt(query[0]);
            queryR[i] = Integer.parseInt(query[1]);
            queryK[i] = Integer.parseInt(query[2]);
            queryId[i] = i;
```

```
        }

        // 离散化
        Arrays.sort(sorted, 1, n + 1);
        int uniqueCount = 1;
        for (int i = 2; i <= n; i++) {
            if (sorted[i] != sorted[i - 1]) {
                sorted[++uniqueCount] = sorted[i];
            }
        }

        // 整体二分求解
        compute(1, m, 1, uniqueCount);

        // 输出结果
        for (int i = 1; i <= m; i++) {
            out.println(ans[i]);
        }

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

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

文件: POJ2104_KthNumber.py
==============================================
"""
POJ 2104 K-th Number - Python 实现
题目来源：http://poj.org/problem?id=2104
题目描述：静态区间第 k 小查询

问题描述：
给定一个长度为 n 的数组，有 m 个查询，每个查询要求在指定区间内找到第 k 小的数。

解题思路：
使用整体二分处理静态区间第 k 小问题。将所有查询一起处理，二分答案的值域，
利用树状数组维护区间内小于等于 mid 的元素个数。

时间复杂度：O((N+Q) * logN * log(maxValue))
空间复杂度：O(N + Q)
```

```
"""

class POJ2104_KthNumberSolution:
    """
    POJ 2104 K-th Number 问题的解决方案类
    使用整体二分算法解决静态区间第 k 小查询问题
    """

    def __init__(self):
        """
        初始化解决方案类的成员变量
        """
        self.MAXN = 100001
        self.n = 0
        self.m = 0

        # 原始数组
        self.arr = [0] * self.MAXN

        # 离散化数组
        self.sorted = [0] * self.MAXN

        # 查询信息
        self.queryL = [0] * self.MAXN  # 查询区间左端点
        self.queryR = [0] * self.MAXN  # 查询区间右端点
        self.queryK = [0] * self.MAXN  # 查询第 k 小
        self.queryId = [0] * self.MAXN # 查询编号

        # 树状数组
        self.tree = [0] * self.MAXN

        # 整体二分
        self.lset = [0] * self.MAXN  # 左集合
        self.rset = [0] * self.MAXN  # 右集合

        # 查询的答案
        self.ans = [0] * self.MAXN

    def lowbit(self, i):
        """
        计算一个数的 lowbit 值
        :param i: 输入的数
        :return: lowbit 值
```

```
    """
    return i & -i

def add(self, i, v):
    """
    在树状数组中给位置 i 增加 v
    :param i: 位置
    :param v: 增加的值
    """
    while i <= self.n:
        self.tree[i] += v
        i += self.lowbit(i)

def sum(self, i):
    """
    计算前缀和[1..i]
    :param i: 位置
    :return: 前缀和
    """
    ret = 0
    while i > 0:
        ret += self.tree[i]
        i -= self.lowbit(i)
    return ret

def query(self, l, r):
    """
    计算区间和[l..r]
    :param l: 左端点
    :param r: 右端点
    :return: 区间和
    """
    return self.sum(r) - self.sum(l - 1)

def compute(self, ql, qr, vl, vr):
    """
    整体二分核心函数
    :param ql: 查询范围的左端点
    :param qr: 查询范围的右端点
    :param vl: 值域范围的左端点（离散化后的下标）
    :param vr: 值域范围的右端点（离散化后的下标）
    """
    # 递归边界
```

```
if ql > qr:
    return

# 如果值域范围只有一个值，说明找到了答案
if vl == vr:
    for i in range(ql, qr + 1):
        self.ans[self.queryId[i]] = self.sorted[vl]
    return

# 二分中点
mid = (vl + vr) >> 1

# 将值小于等于 sorted[mid]的数加入树状数组
for i in range(vl, mid + 1):
    # 遍历所有值为 sorted[i]的元素，将其加入树状数组
    for j in range(1, self.n + 1):
        if self.arr[j] == self.sorted[i]:
            self.add(j, 1)

# 检查每个查询，根据满足条件的元素个数划分到左右区间
lsiz = 0
rsiz = 0
for i in range(ql, qr + 1):
    # 查询区间[queryL[i], queryR[i]]中值小于等于 sorted[mid]的元素个数
    satisfy = self.query(self.queryL[i], self.queryR[i])

    if satisfy >= self.queryK[i]:
        # 说明第 k 小的数在左半部分
        lsiz += 1
        self.lset[lsiz] = i
    else:
        # 说明第 k 小的数在右半部分，需要在右半部分找第(k-satisfy)小的数
        self.queryK[i] -= satisfy
        rsiz += 1
        self.rset[rsiz] = i

# 重新排列查询顺序
idx = ql
for i in range(1, lsiz + 1):
    temp = self.lset[i]
    self.lset[i] = self.queryId[temp]
    self.queryId[idx] = temp
    idx += 1
```

```
    for i in range(1, rsiz + 1):
        temp = self.rset[i]
        self.rset[i] = self.queryId[temp]
        self.queryId[idx] = temp
        idx += 1

    # 撤销对树状数组的修改
    for i in range(vl, mid + 1):
        # 遍历所有值为sorted[i]的元素，将其从树状数组中删除
        for j in range(1, self.n + 1):
            if self.arr[j] == self.sorted[i]:
                self.add(j, -1)

    # 递归处理左右两部分
    self.compute(ql, ql + lsiz - 1, vl, mid)
    self.compute(ql + lsiz, qr, mid + 1, vr)

def solve(self):
    """
    解决POJ 2104 K-th Number问题的主函数
    """
    # 读取输入
    line = input().split()
    self.n = int(line[0])  # 数组长度
    self.m = int(line[1])  # 查询数量

    # 读取原始数组
    nums = input().split()
    for i in range(1, self.n + 1):
        self.arr[i] = int(nums[i - 1])
        self.sorted[i] = self.arr[i]

    # 读取查询
    for i in range(1, self.m + 1):
        query = input().split()
        self.queryL[i] = int(query[0])  # 查询区间左端点
        self.queryR[i] = int(query[1])  # 查询区间右端点
        self.queryK[i] = int(query[2])  # 查询第k小
        self.queryId[i] = i             # 查询编号

    # 离散化
    self.sorted[1:self.n + 1] = sorted(self.sorted[1:self.n + 1])
    uniqueCount = 1
```

```
        for i in range(2, self.n + 1):
            if self.sorted[i] != self.sorted[i - 1]:
                uniqueCount += 1
                self.sorted[uniqueCount] = self.sorted[i]

        # 整体二分求解
        self.compute(1, self.m, 1, uniqueCount)

        # 输出结果
        for i in range(1, self.m + 1):
            print(self.ans[i])


# 主程序
if __name__ == "__main__":
    """
    程序入口点
    创建解决方案实例并执行求解
    """
    solver = POJ2104_KthNumberSolution()
    solver.solve()

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