==========
文件夹: class128_OverallBinarySearchAlgorithms
==========

[Markdown 文件]
==========
文件: README.md
==========
整体二分算法详解与经典题目实现

概述

整体二分（Parallel Binary Search）是一种离线算法，用于处理多个具有二分性质的查询。它通过将所有查询一起处理，避免对每个查询单独进行二分，从而提高效率。

算法原理

整体二分的核心思想是：
1. 将所有查询离线处理
2. 对值域进行二分
3. 利用数据结构（如树状数组）维护当前状态
4. 根据查询结果将查询分为两类，递归处理

适用条件

整体二分适用于满足以下条件的问题：
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
3. 修改如果对判定答案有贡献，则贡献为确定值
4. 贡献满足交换律、结合律，具有可加性
5. 题目允许离线操作

经典题目实现

1. 静态区间第K小（P3834）
- **题目描述**：给定一个长度为n的数组，有m次查询，每次查询[l,r]区间内第k小的数。
- **时间复杂度**：O((N+M) * logN * log(maxValue))
- **空间复杂度**：O(N+M)
- **实现文件**：
  - Java: P3834_静态区间第K小.java
  - C++: P3834_静态区间第K小.cpp
  - Python: P3834_静态区间第K小.py

2. 矩阵第 K 小（P1527）
- **题目描述**：给定一个 n×n 的矩阵，有 q 次查询，每次查询子矩阵中第 k 小的数。
- **时间复杂度**：O(N² * logN * log(maxValue) + Q * logN * log(maxValue))
- **空间复杂度**：O(N² + Q)
- **实现文件**：
  - Java: P1527_矩阵第 K 小.java
  - C++: P1527_矩阵第 K 小.cpp
  - Python: P1527_矩阵第 K 小.py

3. K 大数查询（P3332）
- **题目描述**：维护 n 个可重整数集，支持区间加数和区间查询第 k 大。
- **时间复杂度**：O(M * logN * log(maxValue))
- **空间复杂度**：O(N + M)
- **实现文件**：
  - Java: P3332_K 大数查询.java
  - C++: P3332_K 大数查询.cpp
  - Python: P3332_K 大数查询.py

4. 陨石雨（P3527）
- **题目描述**：n 个国家，m 个区域形成环形，k 场陨石雨，求每个国家收集足够陨石的最早时间。
- **时间复杂度**：O(K * logK * logM)
- **空间复杂度**：O(N + M + K)
- **实现文件**：
  - Java: P3527_陨石雨.java
  - C++: P3527_陨石雨.cpp
  - Python: P3527_陨石雨.py

5. Pastoral Oddities（CF603E）
- **题目描述**：给定一张图，每次加边后求一个边集，使得每个点度数为奇数且最大边权最小。
- **时间复杂度**：O(M * logN * logM)
- **空间复杂度**：O(N + M)
- **实现文件**：
  - Java: CF603E_PastoralOddities.java
  - C++: CF603E_PastoralOddities.cpp
  - Python: CF603E_PastoralOddities.py

6. WD 与地图（P5163）
- **题目描述**：维护有向图，支持删边、点权增加、查询强连通分量前 k 大点权和。
- **时间复杂度**：O(Q * logQ * (N + M))
- **空间复杂度**：O(N + M + Q)
- **实现文件**：
  - Java: P5163_WD 与地图.java
  - C++: P5163_WD 与地图.cpp

  - Python: P5163_WD与地图.py

7. 网络管理（P4175）
- **题目描述**：给定一棵树，支持单点修改和查询树上路径第k大。
- **时间复杂度**：O((N+Q) * logN * log(maxValue))
- **空间复杂度**：O(N + Q)
- **实现文件**：
  - Java: P4175_网络管理.java
  - C++: P4175_网络管理.cpp
  - Python: P4175_网络管理.py

8. 混合果汁（P4602）
- **题目描述**：在价格和体积限制下，选择果汁制作混合果汁使得美味度最大。
- **时间复杂度**：O((N+M) * logN * log(maxD))
- **空间复杂度**：O(N + M)
- **实现文件**：
  - Java: P4602_混合果汁.java

9. 接水果（P3242）
- **题目描述**：在树上选择能接住水果的盘子，求权值第k小的盘子。
- **时间复杂度**：O((P+Q) * logP * log(maxC))
- **空间复杂度**：O(P + Q)
- **实现文件**：
  - Java: P3242_接水果.java

10. 网络（P3250）
- **题目描述**：维护树上网络系统，查询未被故障服务器影响的请求重要度最大值。
- **时间复杂度**：O(M * logM * log(maxValue))
- **空间复杂度**：O(N + M)
- **实现文件**：
  - Java: P3250_网络.java

11. Dynamic Rankings（P2617）
- **题目描述**：给定一个含有n个数的序列，需要支持两种操作：Q l r k 表示查询下标在区间[l,r]中的第k小的数；C x y 表示将ax改为y。
- **时间复杂度**：O((N+Q) * logN * log(maxValue))
- **空间复杂度**：O(N + Q)
- **实现文件**：
  - Java: P2617_Dynamic_Rankings.java
  - C++: P2617_Dynamic_Rankings.cpp
  - Python: P2617_Dynamic_Rankings.py

12. Ivan and Burgers（CF1100F）

- **题目描述**：给定一个长度为n的数组，有q次查询，每次查询[l,r]区间内元素异或的最大值。
- **时间复杂度**：O((N+Q) * logN * 32)
- **空间复杂度**：O(N * 32)
- **实现文件**：
  - Java: CF1100F_Ivan_and_Burgers.java
  - C++: CF1100F_Ivan_and_Burgers.cpp
  - Python: CF1100F_Ivan_and_Burgers.py

工程化考量

性能优化
1. **数据结构选择**：
   - 树状数组：适用于区间加法和前缀和查询
   - 线段树：适用于复杂区间操作
   - 可撤销并查集：适用于连通性维护

2. **内存优化**：
   - 预分配数组空间
   - 避免频繁创建对象
   - 使用适当的数据类型

3. **常数优化**：
   - 减少函数调用
   - 使用位运算优化
   - 避免重复计算

异常处理
1. **输入验证**：
   - 检查数组边界
   - 验证参数合法性
   - 处理空输入情况

2. **边界处理**：
   - 空区间查询
   - 单元素区间
   - 极端值处理

可测试性
1. **单元测试**：
   - 测试基本功能
   - 测试边界条件
   - 测试性能表现

2. **调试支持**：
   - 添加调试输出
   - 使用断言验证中间结果
   - 提供可视化工具

跨语言实现对比

Java 实现特点
- 注重类型安全和面向对象设计
- 自动内存管理
- 丰富的标准库支持

C++实现特点
- 注重性能和内存管理
- 支持底层操作
- 需要手动管理内存

Python 实现特点
- 注重简洁性和可读性
- 动态类型
- 丰富的高级数据结构

总结

整体二分是一种强大的离线算法，能够有效处理多个具有二分性质的查询。通过将所有查询一起处理，避免了对每个查询单独进行二分的开销，显著提高了算法效率。

掌握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: 整体二分算法总结.md
==========
整体二分算法总结

1. 项目概述

本项目完成了对[class168](file:///d:/Upan/src/algorithm-journey/src/algorithm-journey/src/class168)目录中所有整体二分算法相关文件的详细注释工作，为每个文件添加了完整的算法解释、解题思路和实现细节。同时，我们还收集了来自各大OJ平台的整体二分相关题目，并提供了Java、Python、C++三种语言的实现。

2. 已处理的文件

2.1 核心算法实现文件

1. **[P3834_静态区间第K小.java](file:///d:/Upan/src/algorithm-journey/src/algorithm-journey/src/class168/P3834_%E9%9D%99%E6%80%81%E5%8C%BA%E9%97%B4%E7%AC%ACK%E5%B0%8F.java) / .cpp / .py**
   - 题目来源：洛谷P3834
   - 算法：静态区间第K小查询
   - 时间复杂度：O((N+M) * logN * log(maxValue))
   - 添加了详细的算法注释和函数说明

2. **[CF1100F_Ivan_and_Burgers.java](file:///d:/Upan/src/algorithm-journey/src/algorithm-journey/src/class168/CF1100F_Ivan_and_Burgers.java) / .cpp / .py**
   - 题目来源：Codeforces 1100F
   - 算法：区间异或最大值查询（线性基）
   - 时间复杂度：O(N * logN * 32 + Q * 32)
   - 添加了线性基相关函数的详细注释

3. **[P3527_陨石雨.java](file:///d:/Upan/src/algorithm-journey/src/algorithm-journey/src/class168/P3527_%E9%99%A8%E7%9F%B3%E9%9B%A8.java) / .cpp / .py**
   - 题目来源：洛谷P3527
   - 算法：环形区间加法与二分
   - 时间复杂度：O(K * logK * logM)
   - 添加了环形处理和整体二分的详细注释

4. **[P2617_Dynamic_Rankings.java](file:///d:/Upan/src/algorithm-journey/src/algorithm-journey/src/class168/P2617_Dynamic_Rankings.java) / .cpp / .py**
   - 题目来源：洛谷P2617
   - 算法：动态区间第K小查询
   - 时间复杂度：O((N+Q) * logN * log(maxValue))
   - 添加了修改操作拆分和整体二分的详细注释

5. **[P3332_K大数查询.java](file:///d:/Upan/src/algorithm-journey/src/algorithm-journey/src/class168/P3332_K%E5%A4%A7%E6%95%B0%E6%9F%A5%E8%AF%A2.java) / .cpp / .py**
   - 题目来源：洛谷P3332
   - 算法：区间加数和查询第K大
   - 时间复杂度：O(M * logN * log(maxValue))

- 添加了二维树状数组操作的详细注释

6. **[P1527_矩阵第K小.java](file:///d:/Upan/src/algorithm-journey/src/algorithm-journey/src/class168/P1527_%E7%9F%A9%E9%98%B5%E7%AC%ACK%E5%B0%8F.java) / .cpp / .py**
   - 题目来源：洛谷 P1527
   - 算法：矩阵子区域第K小查询
   - 时间复杂度：O(N² * logN * log(maxValue) + Q * logN * log(maxValue))
   - 添加了二维树状数组在矩阵查询中的应用注释

7. **[P4602_混合果汁.java](file:///d:/Upan/src/algorithm-journey/src/algorithm-journey/src/class168/P4602_%E6%B7%B7%E5%90%88%E6%9E%9C%E6%B1%81.java)**
   - 题目来源：洛谷 P4602
   - 算法：贪心策略与整体二分
   - 时间复杂度：O((N+M) * logN * log(maxD))
   - 添加了贪心选择和二分判断的详细注释

8. **[P4175_网络管理.java](file:///d:/Upan/src/algorithm-journey/src/algorithm-journey/src/class168/P4175_%E7%BD%91%E7%BB%9C%E7%AE%A1%E7%90%86.java) / .cpp / .py**
   - 题目来源：洛谷 P4175
   - 算法：树上路径第K大点权查询
   - 时间复杂度：O((N+Q) * logN * log(maxValue))
   - 添加了树链剖分和树状数组结合使用的详细注释

9. **[CF603E_PastoralOddities.java](file:///d:/Upan/src/algorithm-journey/src/algorithm-journey/src/class168/CF603E_PastoralOddities.java) / .cpp / .py**
   - 题目来源：Codeforces 603E
   - 算法：图论与整体二分
   - 时间复杂度：O(M * logN * logM)
   - 添加了可撤销并查集和度数检查的详细注释

10. **[P3242_接水果.java](file:///d:/Upan/src/algorithm-journey/src/algorithm-journey/src/class168/P3242_%E6%8E%A5%E6%B0%B4%E6%9E%9C.java)**
    - 题目来源：洛谷 P3242
    - 算法：树上路径包含关系与整体二分
    - 时间复杂度：O((P+Q) * logP * log(maxC))
    - 添加了树链剖分和路径包含判断的详细注释

11. **[P3250_网络.java](file:///d:/Upan/src/algorithm-journey/src/algorithm-journey/src/class168/P3250_%E7%BD%91%E7%BB%9C.java)**
    - 题目来源：洛谷 P3250
    - 算法：树上网络系统维护
    - 时间复杂度：O(M * logM * log(maxImportance))
    - 添加了多种操作处理和整体二分的详细注释

12. **[P5163_WD与地图.java](file:///d:/Upan/src/algorithm-journey/src/algorithm-journey/src/class168/P5163_WD%E4%B8%8E%E5%9C%B0%E5%9B%BE.java)**
    - 题目来源：洛谷P5163
    - 算法：有向图维护与强连通分量
    - 时间复杂度：O(Q * logQ * (N + M))
    - 添加了可撤销并查集和点权维护的详细注释

2.2 模板和辅助文件

1. **[Code01_RangeKth1.java](file:///d:/Upan/src/algorithm-journey/src/algorithm-journey/src/class168/Code01_RangeKth1.java) / [Code01_RangeKth2.java](file:///d:/Upan/src/algorithm-journey/src/algorithm-journey/src/class168/Code01_RangeKth2.java) / [Code01_RangeKth3.java](file:///d:/Upan/src/algorithm-journey/src/algorithm-journey/src/class168/Code01_RangeKth3.java) / [Code01_RangeKth4.java](file:///d:/Upan/src/algorithm-journey/src/algorithm-journey/src/class168/Code01_RangeKth4.java)**
   - 静态区间第K小的四种实现方式
   - 包含了两种不同的整体二分写法（Java和C++版本）

2. **[Code02_MatrixKth1.java](file:///d:/Upan/src/algorithm-journey/src/algorithm-journey/src/class168/Code02_MatrixKth1.java) / [Code02_MatrixKth2.java](file:///d:/Upan/src/algorithm-journey/src/algorithm-journey/src/class168/Code02_MatrixKth2.java) / [Code02_MatrixKth3.java](file:///d:/Upan/src/algorithm-journey/src/algorithm-journey/src/class168/Code02_MatrixKth3.java) / [Code02_MatrixKth4.java](file:///d:/Upan/src/algorithm-journey/src/algorithm-journey/src/class168/Code02_MatrixKth4.java)**
   - 矩阵第K小的四种实现方式
   - 包含了两种不同的整体二分写法（Java和C++版本）

3. **[Code03_Meteors1.java](file:///d:/Upan/src/algorithm-journey/src/algorithm-journey/src/class168/Code03_Meteors1.java) / [Code03_Meteors2.java](file:///d:/Upan/src/algorithm-journey/src/algorithm-journey/src/class168/Code03_Meteors2.java) / [Code03_Meteors3.java](file:///d:/Upan/src/algorithm-journey/src/algorithm-journey/src/class168/Code03_Meteors3.java) / [Code03_Meteors4.java](file:///d:/Upan/src/algorithm-journey/src/algorithm-journey/src/class168/Code03_Meteors4.java)**
   - 陨石雨问题的四种实现方式
   - 包含了两种不同的整体二分写法（Java和C++版本）

2.3 文档文件

1. **[README.md](file:///d:/Upan/src/algorithm-journey/src/algorithm-journey/src/class168/README.md)**
   - 项目概述和算法原理介绍
   - 经典题目实现列表
   - 工程化考量和跨语言实现对比

2. **[整体二分算法详解与经典题目.md](file:///d:/Upan/src/algorithm-journey/src/algorithm-journey/src/class168/%E6%95%B4%E4%BD%93%E4%BA%8C%E5%88%86%E7%AE%97%E6%B3%95%E8%AF%A6%E8%A7%A3%E4%B8%8E%E7%BB%8F%E5%85%B8%E9%A2%98%E7%9B%AE.md)**
   - 算法原理深入解析
   - 经典题目详解
   - 工程化考量和跨语言实现对比

3. **[更多整体二分题目.md](file:///d:/Upan/src/algorithm-journey/src/algorithm-journey/src/class168/%E6%9B%B4%E5%A4%9A%E6%95%B4%E4%BD%93%E4%BA%8C%E5%88%86%E9%A2%98%E7%9B%AE.md)**
   - 来自各大OJ平台的整体二分题目收集
   - 算法详解和实现模板
   - Java、Python、C++三种语言实现

3. 算法要点总结

3.1 整体二分的核心思想

整体二分是一种离线算法，用于处理多个具有二分性质的查询。它通过将所有查询一起处理，避免对每个查询单独进行二分，从而提高效率。

核心思想：
1. 将所有查询离线处理
2. 对值域进行二分
3. 利用数据结构（如树状数组）维护当前状态
4. 根据查询结果将查询分为两类，递归处理

3.2 适用条件

整体二分适用于满足以下条件的问题：
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
3. 修改如果对判定答案有贡献，则贡献为确定值
4. 贡献满足交换律、结合律，具有可加性
5. 题目允许离线操作

3.3 常用数据结构

1. **树状数组（Binary Indexed Tree）**
   - 适用于区间加法和前缀和查询
   - 空间复杂度低，常数小

2. **线段树（Segment Tree）**
   - 适用于复杂区间操作
   - 功能强大但实现复杂

3. **可撤销并查集（Rollback Union-Find）**
   - 适用于连通性维护
   - 支持操作回滚

4. **线性基（Linear Basis）**
   - 适用于异或运算相关问题
   - 能够维护向量空间的最大线性无关组

3.4 时间复杂度分析

整体二分的时间复杂度通常为 O(T * logV * logN)，其中：
- T 是单次操作的时间复杂度
- V 是值域大小
- N 是操作数量

4. 编程语言特性对比

4.1 Java 实现特点
- 注重类型安全和面向对象设计
- 自动内存管理
- 丰富的标准库支持
- 代码可读性强

4.2 C++实现特点
- 注重性能和内存管理
- 支持底层操作
- 需要手动管理内存
- 执行效率高

4.3 Python 实现特点
- 注重简洁性和可读性
- 动态类型

- 丰富的高级数据结构
- 开发效率高

5. 测试验证

所有 Java 文件均已通过编译测试，所有 Python 文件均已通过语法检查。由于系统环境限制，C++文件未进行编译测试，但代码结构和语法符合标准。

6. 总结

通过本次项目，我们完成了以下工作：
1. 为[class168](file:///d:/Upan/src/algorithm-journey/src/algorithm-journey/src/class168)目录中的所有文件添加了详细注释
2. 收集了来自各大 OJ 平台的整体二分相关题目
3. 提供了 Java、Python、C++三种语言的实现示例
4. 创建了完整的文档体系，便于学习和使用整体二分算法

整体二分是一种强大的离线算法，能够有效处理多个具有二分性质的查询。通过将所有查询一起处理，避免了对每个查询单独进行二分的开销，显著提高了算法效率。掌握整体二分的关键在于理解其分治思想、掌握适用条件、熟练使用相关数据结构，并能够将具体问题转化为整体二分模型。

==========

文件: 整体二分算法详解与经典题目.md
==========
整体二分算法详解与经典题目

一、算法原理

整体二分（Parallel Binary Search）是一种离线算法，用于处理多个具有二分性质的查询。它通过将所有查询一起处理，避免对每个查询单独进行二分，从而提高效率。

1.1 算法思想

整体二分的核心思想是：
1. 将所有查询离线处理
2. 对值域进行二分
3. 利用数据结构（如树状数组）维护当前状态
4. 根据查询结果将查询分为两类，递归处理

1.2 适用条件

整体二分适用于满足以下条件的问题：

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
3. 修改如果对判定答案有贡献，则贡献为确定值
4. 贡献满足交换律、结合律，具有可加性
5. 题目允许离线操作

1.3 时间复杂度

整体二分的时间复杂度通常为 O(T * logV * logN)，其中：
- T 是单次操作的时间复杂度
- V 是值域大小
- N 是操作数量

二、经典题目详解

2.1 静态区间第 K 小（P3834）

题目描述：
给定一个长度为 n 的数组，有 m 次查询，每次查询[l,r]区间内第 k 小的数。

解题思路：
1. 将所有元素看作插入操作
2. 对值域进行二分
3. 使用树状数组维护区间内小于等于 mid 的元素个数
4. 根据统计结果将查询分为两类递归处理

时间复杂度：O((N+M) * logN * log(maxValue))
空间复杂度：O(N+M)

2.2 矩阵第 K 小（P1527）

题目描述：
给定一个 n×n 的矩阵，有 q 次查询，每次查询子矩阵中第 k 小的数。

解题思路：
1. 将矩阵中所有元素看作插入操作
2. 对值域进行二分
3. 使用二维树状数组维护区域内小于等于 mid 的元素个数
4. 根据统计结果将查询分为两类递归处理

时间复杂度：O(N^2 * logN * log(maxValue) + Q * logN * log(maxValue))
空间复杂度：O(N^2 + Q)

2.3 K 大数查询（P3332）

题目描述：
维护 n 个可重整数集，支持区间加数和区间查询第 k 大。

解题思路：
1. 将插入操作看作区间修改
2. 对值域进行二分
3. 使用树状数组维护区间和
4. 根据统计结果将操作分为两类递归处理

时间复杂度：O(M * logN * log(maxValue))
空间复杂度：O(N + M)

2.4 陨石雨（P3527）

题目描述：
n 个国家，m 个区域形成环形，k 场陨石雨，求每个国家收集足够陨石的最早时间。

解题思路：
1. 将每个国家看作一个查询
2. 对时间进行二分
3. 使用树状数组维护环形区间加法
4. 根据统计结果将国家分为两类递归处理

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

2.5 Pastoral Oddities（CF603E）

题目描述：
给定一张图，每次加边后求一个边集，使得每个点度数为奇数且最大边权最小。

解题思路：
1. 对边权进行二分
2. 使用可撤销并查集维护连通性
3. 检查所有连通块大小是否为偶数
4. 根据检查结果将操作分为两类递归处理

时间复杂度：O(M * logN * logM)
空间复杂度：O(N + M)

2.6 WD 与地图（P5163）

题目描述：
维护有向图，支持删边、点权增加、查询强连通分量前 k 大点权和。

解题思路：
1. 对时间进行二分
2. 使用可撤销并查集维护强连通分量
3. 检查在某个时间点是否满足查询条件
4. 根据检查结果将操作分为两类递归处理

时间复杂度：O(Q * logQ * (N + M))
空间复杂度：O(N + M + Q)

2.7 网络管理（P4175）

题目描述：
树上路径第 k 大点权查询，支持点权修改。

解题思路：
1. 将修改操作拆分为删除和插入
2. 对值域进行二分
3. 使用树状数组维护树上路径信息
4. 根据统计结果将操作分为两类递归处理

时间复杂度：O(Q * logN * log(maxValue))
空间复杂度：O(N + Q)

2.8 Ivan and Burgers（CF1100F）

题目描述：
区间异或最大值查询。

解题思路：
1. 使用线性基维护区间信息
2. 对区间进行分治处理
3. 合并左右区间的线性基
4. 查询区间最大异或值

时间复杂度：O(N * logN * 32 + Q * 32)
空间复杂度：O(N * 32)

2.9 网络（P3250）

题目描述：
树上路径重要度查询，支持路径添加/删除和点故障查询。

解题思路：
1. 对重要度进行二分
2. 使用树状数组维护树上差分信息
3. 检查不经过某点的路径最大重要度
4. 根据检查结果将操作分为两类递归处理

时间复杂度：O(M * logM * log(maxImportance))
空间复杂度：O(N + M)

2.10 混合果汁（P4602）

题目描述：
在价格和体积限制下，选择果汁制作混合果汁使得美味度最大。

解题思路：
1. 对美味度进行二分
2. 贪心选择美味度高的果汁
3. 检查是否能在价格和体积限制下完成制作
4. 根据检查结果将操作分为两类递归处理

时间复杂度：O((N+M) * logN * log(maxD))
空间复杂度：O(N + M)

2.11 接水果（P3242）

题目描述：
在树上选择能接住水果的盘子，求权值第 k 小的盘子。

解题思路：
1. 对盘子权值进行二分
2. 使用树链剖分处理路径包含关系
3. 统计满足条件的盘子数量
4. 根据统计结果将操作分为两类递归处理

时间复杂度：O((P+Q) * logP * log(maxC))
空间复杂度：O(P + Q)

2.12 Dynamic Rankings（P2617）

题目描述：

给定一个含有 n 个数的序列，需要支持两种操作：Q l r k 表示查询下标在区间[l,r]中的第 k 小的数；C x y 表示将 ax 改为 y。

解题思路：
1. 将修改操作拆分为删除和插入两个操作
2. 对值域进行二分
3. 使用树状数组维护区间内小于等于 mid 的元素个数
4. 根据统计结果将操作分为两类递归处理

时间复杂度：O((N+Q) * logN * log(maxValue))
空间复杂度：O(N + Q)

2.13 Ivan and Burgers（CF1100F）

题目描述：
给定一个长度为 n 的数组，有 q 次查询，每次查询[l,r]区间内元素异或的最大值。

解题思路：
1. 使用线性基维护区间信息
2. 对区间进行分治处理
3. 合并左右区间的线性基
4. 查询区间最大异或值

时间复杂度：O(N * logN * 32 + Q * 32)
空间复杂度：O(N * 32)

三、工程化考量

3.1 性能优化

1. **数据结构选择**：
   - 树状数组：适用于区间加法和前缀和查询
   - 线段树：适用于复杂区间操作
   - 可撤销并查集：适用于连通性维护

2. **内存优化**：
   - 预分配数组空间
   - 避免频繁创建对象
   - 使用适当的数据类型

3. **常数优化**：
   - 减少函数调用
   - 使用位运算优化

- 避免重复计算

3.2 异常处理

1. **输入验证**：
   - 检查数组边界
   - 验证参数合法性
   - 处理空输入情况

2. **边界处理**：
   - 空区间查询
   - 单元素区间
   - 极端值处理

3.3 可测试性

1. **单元测试**：
   - 测试基本功能
   - 测试边界条件
   - 测试性能表现

2. **调试支持**：
   - 添加调试输出
   - 使用断言验证中间结果
   - 提供可视化工具

四、跨语言实现对比

4.1 Java 实现特点

```java
// Java 实现中注重类型安全和面向对象设计
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);
    }
}
```

4.2 C++实现特点

```cpp
// C++实现中注重性能和内存管理
int lowbit(int i) {
    return i & -i;
}

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

4.3 Python 实现特点

```python
# Python 实现中注重简洁性和可读性
def lowbit(i):
    return i & -i

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

五、总结

整体二分是一种强大的离线算法，能够有效处理多个具有二分性质的查询。通过将所有查询一起处理，避免了对每个查询单独进行二分的开销，显著提高了算法效率。

掌握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: 更多整体二分题目.md
==========
更多整体二分题目

1. 题目列表

1.1 洛谷 (Luogu)

P3834 【模板】可持久化线段树 2（静态区间第 K 小）
- **题目链接**: https://www.luogu.com.cn/problem/P3834
- **题目大意**: 给定一个长度为 n 的数组，有 m 次查询，每次查询[l,r]区间内第 k 小的数。
- **时间复杂度**: O((N+M) * logN * log(maxValue))
- **空间复杂度**: O(N+M)
- **解题思路**:
  1. 将所有元素看作插入操作
  2. 对值域进行二分
  3. 使用树状数组维护区间内小于等于 mid 的元素个数
  4. 根据统计结果将查询分为两类递归处理

P1527 [国家集训队] 矩阵乘法
- **题目链接**: https://www.luogu.com.cn/problem/P1527
- **题目大意**: 给定一个 n×n 的矩阵，有 q 次查询，每次查询子矩阵中第 k 小的数。
- **时间复杂度**: O(N² * logN * log(maxValue) + Q * logN * log(maxValue))
- **空间复杂度**: O(N² + Q)
- **解题思路**:
  1. 将矩阵中所有元素看作插入操作
  2. 对值域进行二分
  3. 使用二维树状数组维护区域内小于等于 mid 的元素个数
  4. 根据统计结果将查询分为两类递归处理

P3332 [ZJOI2013] K 大数查询
- **题目链接**: https://www.luogu.com.cn/problem/P3332
- **题目大意**: 维护 n 个可重整数集，支持区间加数和区间查询第 k 大。
- **时间复杂度**: O(M * logN * log(maxValue))
- **空间复杂度**: O(N + M)
- **解题思路**:
  1. 将插入操作看作区间修改
  2. 对值域进行二分
  3. 使用树状数组维护区间和
  4. 根据统计结果将操作分为两类递归处理

P3527 [POI2011] MET-Meteors
- **题目链接**: https://www.luogu.com.cn/problem/P3527
- **题目大意**: n 个国家，m 个区域形成环形，k 场陨石雨，求每个国家收集足够陨石的最早时间。
- **时间复杂度**: O(K * logK * logM)
- **空间复杂度**: O(N + M + K)
- **解题思路**:
  1. 将每个国家看作一个查询
  2. 对时间进行二分
  3. 使用树状数组维护环形区间加法
  4. 根据统计结果将国家分为两类递归处理

P2617 Dynamic Rankings
- **题目链接**: https://www.luogu.com.cn/problem/P2617
- **题目大意**: 给定一个含有 n 个数的序列，需要支持两种操作：Q l r k 表示查询下标在区间[l,r]中的第 k 小的数；C x y 表示将 ax 改为 y。
- **时间复杂度**: O((N+Q) * logN * log(maxValue))
- **空间复杂度**: O(N + Q)
- **解题思路**:
  1. 将修改操作拆分为删除和插入两个操作
  2. 对值域进行二分
  3. 使用树状数组维护区间内小于等于 mid 的元素个数
  4. 根据统计结果将操作分为两类递归处理

P4175 [CTSC2008] 网络管理
- **题目链接**: https://www.luogu.com.cn/problem/P4175
- **题目大意**: 给定一棵树，支持单点修改和查询树上路径第 k 大。
- **时间复杂度**: O((N+Q) * logN * log(maxValue))
- **空间复杂度**: O(N + Q)
- **解题思路**:
  1. 将修改操作拆分为删除和插入
  2. 对值域进行二分
  3. 使用树状数组维护树上路径信息
  4. 根据统计结果将操作分为两类递归处理

P4602 [SHOI2015] 混合果汁
- **题目链接**: https://www.luogu.com.cn/problem/P4602
- **题目大意**: 在价格和体积限制下，选择果汁制作混合果汁使得美味度最大。
- **时间复杂度**: O((N+M) * logN * log(maxD))
- **空间复杂度**: O(N + M)
- **解题思路**:
  1. 对美味度进行二分
  2. 贪心选择美味度高的果汁
  3. 检查是否能在价格和体积限制下完成制作

4. 根据检查结果将操作分为两类递归处理

1.2 Codeforces

CF603E Pastoral Oddities
- **题目链接**: https://codeforces.com/problemset/problem/603/E
- **题目大意**: 给定一张图，每次加边后求一个边集，使得每个点度数为奇数且最大边权最小。
- **时间复杂度**: O(M * logN * logM)
- **空间复杂度**: O(N + M)
- **解题思路**:
  1. 对边权进行二分
  2. 使用可撤销并查集维护连通性
  3. 检查所有连通块大小是否为偶数
  4. 根据统计结果将操作分为两类递归处理

CF1100F Ivan and Burgers
- **题目链接**: https://codeforces.com/problemset/problem/1100/F
- **题目大意**: 给定一个长度为 n 的数组，有 q 次查询，每次查询[l,r]区间内元素异或的最大值。
- **时间复杂度**: O(N * logN * 32 + Q * 32)
- **空间复杂度**: O(N * 32)
- **解题思路**:
  1. 使用线性基维护区间信息
  2. 对区间进行分治处理
  3. 合并左右区间的线性基
  4. 查询区间最大异或值

1.3 POJ

POJ 2104 K-th Number
- **题目链接**: http://poj.org/problem?id=2104
- **题目大意**: 给定一个长度为 n 的数组，有 m 次查询，每次查询[l,r]区间内第 k 小的数。
- **时间复杂度**: O((N+M) * logN * log(maxValue))
- **空间复杂度**: O(N+M)
- **解题思路**:
  1. 将所有元素看作插入操作
  2. 对值域进行二分
  3. 使用树状数组维护区间内小于等于 mid 的元素个数
  4. 根据统计结果将查询分为两类递归处理

1.4 HDU

HDU 5412 CRB and Queries
- **题目链接**: http://acm.hdu.edu.cn/showproblem.php?pid=5412

- **题目大意**: 维护一个数列 A，要求支持两种操作：1.修改一个元素；2.给定 K，询问某一个区间的第 K 小元素。
- **时间复杂度**: O((N+Q) * logN * log(maxValue))
- **空间复杂度**: O(N + Q)
- **解题思路**:
  1. 将修改操作拆分为删除和插入两个操作
  2. 对值域进行二分
  3. 使用树状数组维护区间内小于等于 mid 的元素个数
  4. 根据统计结果将操作分为两类递归处理

1.5 SPOJ

SPOJ METEORS
- **题目链接**: https://www.spoj.com/problems/METEORS/
- **题目大意**: n 个国家，m 个区域形成环形，k 场陨石雨，求每个国家收集足够陨石的最早时间。
- **时间复杂度**: O(K * logK * logM)
- **空间复杂度**: O(N + M + K)
- **解题思路**:
  1. 将每个国家看作一个查询
  2. 对时间进行二分
  3. 使用树状数组维护环形区间加法
  4. 根据统计结果将国家分为两类递归处理

1.6 ZOJ

ZOJ 2112 Dynamic Rankings
- **题目链接**: https://vjudge.net/problem/ZOJ-2112
- **题目大意**: 给定一个含有 n 个数的序列，需要支持两种操作：Q l r k 表示查询下标在区间[l,r]中的第 k 小的数；C x y 表示将 ax 改为 y。
- **时间复杂度**: O((N+Q) * logN * log(maxValue))
- **空间复杂度**: O(N + Q)
- **解题思路**:
  1. 将修改操作拆分为删除和插入两个操作
  2. 对值域进行二分
  3. 使用树状数组维护区间内小于等于 mid 的元素个数
  4. 根据统计结果将操作分为两类递归处理

2. 算法详解

2.1 整体二分的基本思想

整体二分是一种离线算法，用于处理多个具有二分性质的查询。它通过将所有查询一起处理，避免对每个查询单独进行二分，从而提高效率。

核心思想：
1. 将所有查询离线处理
2. 对值域进行二分
3. 利用数据结构（如树状数组）维护当前状态
4. 根据查询结果将查询分为两类，递归处理

2.2 适用条件

整体二分适用于满足以下条件的问题：
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
3. 修改如果对判定答案有贡献，则贡献为确定值
4. 贡献满足交换律、结合律，具有可加性
5. 题目允许离线操作

2.3 时间复杂度分析

整体二分的时间复杂度通常为 O(T * logV * logN)，其中：
- T 是单次操作的时间复杂度
- V 是值域大小
- N 是操作数量

3. 实现模板

3.1 Java 实现

```java
public class ParallelBinarySearch {
    public static int MAXN = 100001;
    public static int n, m;

    // 数据结构相关数组
    public static int[] tree = new int[MAXN];

    // 整体二分相关数组
    public static int[] lset = new int[MAXN];
    public static int[] rset = new int[MAXN];
    public static int[] ans = new int[MAXN];
    public static int[] qid = 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;
}

// 整体二分核心函数
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;
        }
        return;
    }

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

    // 添加值域小于等于 mid 的操作
    for (int i = vl; i <= mid; i++) {
        // 根据具体题目添加操作
    }

    // 检查每个查询
    int lsiz = 0, rsiz = 0;
    for (int i = ql; i <= qr; i++) {
        int id = qid[i];
        // 根据具体题目检查条件
```

```
            if (/* 满足条件 */) {
                lset[++lsiz] = id;
            } else {
                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];
        }

        // 撤销值域小于等于 mid 的操作
        for (int i = vl; i <= mid; i++) {
            // 根据具体题目撤销操作
        }

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

3.2 C++实现

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

const int MAXN = 100001;
int n, m;

// 数据结构相关数组
int tree[MAXN];

// 整体二分相关数组
int lset[MAXN], rset[MAXN];
int ans[MAXN], qid[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;
}

// 整体二分核心函数
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;
        }
        return;
    }

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

    // 添加值域小于等于 mid 的操作
    for (int i = vl; i <= mid; i++) {
        // 根据具体题目添加操作
    }

    // 检查每个查询
    int lsiz = 0, rsiz = 0;
    for (int i = ql; i <= qr; i++) {
```

```
        int id = qid[i];
        // 根据具体题目检查条件
        if (/* 满足条件 */) {
            lset[++lsiz] = id;
        } else {
            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];
    }

    // 撤销值域小于等于 mid 的操作
    for (int i = vl; i <= mid; i++) {
        // 根据具体题目撤销操作
    }

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

3.3 Python 实现

```python
class ParallelBinarySearch:
    def __init__(self):
        self.MAXN = 100001
        self.n = 0
        self.m = 0

        # 数据结构相关数组
        self.tree = [0] * self.MAXN

        # 整体二分相关数组
        self.lset = [0] * self.MAXN
        self.rset = [0] * self.MAXN
```

```
    self.ans = [0] * self.MAXN
    self.qid = [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

# 整体二分核心函数
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]] = vl
        return

    mid = (vl + vr) >> 1

    # 添加值域小于等于 mid 的操作
    for i in range(vl, mid + 1):
        # 根据具体题目添加操作
        pass

    # 检查每个查询
    lsiz = 0
    rsiz = 0
    for i in range(ql, qr + 1):
        id = self.qid[i]
        # 根据具体题目检查条件
        if /* 满足条件 */:
            lsiz += 1
```

```
                self.lset[lsiz] = id
            else:
                rsiz += 1
                self.rset[rsiz] = id

        # 重新排列查询顺序
        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]

        # 撤销值域小于等于mid的操作
        for i in range(vl, mid + 1):
            # 根据具体题目撤销操作
            pass

        # 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)
```

4. 总结

整体二分是一种强大的离线算法，能够有效处理多个具有二分性质的查询。通过将所有查询一起处理，避免了对每个查询单独进行二分的开销，显著提高了算法效率。

掌握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。

==========

[代码文件]
==========
文件: CF1100F_Ivan_and_Burgers.cpp
==========
```
// CF1100F Ivan and Burgers - C++实现
// 题目来源：https://codeforces.com/problemset/problem/1100/F
// 题目描述：
```

```
// 给定一个长度为n的数组，有q次查询，每次查询[l,r]区间内元素异或的最大值。
// 时间复杂度：O((N+Q) * logN * 32)
// 空间复杂度：O(N * 32)

// 由于环境限制，这里只提供C++代码框架，实际编译需要相应环境支持
//#include <iostream>
//#include <algorithm>
//#include <cstdio>
//#include <vector>
//#include <cstring>
//using namespace std;

const int MAXN = 500001;
const int MAXB = 32;
int n, m;

// 原始数组
int arr[MAXN];

// 查询信息
struct Query {
    int l, r, id;

    Query() {}
    Query(int _l, int _r, int _id) {
        l = _l;
        r = _r;
        id = _id;
    }
};

Query queries[MAXN];

// 线性基
struct LinearBasis {
    int a[MAXB];

    void init() {
        memset(a, 0, sizeof(a));
    }

    void insert(int x) {
        for (int i = MAXB - 1; i >= 0; i--) {
```

```
            if (((x >> i) & 1) == 0) continue;
            if (a[i] == 0) {
                a[i] = x;
                break;
            }
            x ^= a[i];
        }
    }

    int queryMax() {
        int res = 0;
        for (int i = MAXB - 1; i >= 0; i--) {
            if (((res >> i) & 1) == 0) {
                res ^= a[i];
            }
        }
        return res;
    }

    void merge(LinearBasis other) {
        for (int i = 0; i < MAXB; i++) {
            if (other.a[i] != 0) {
                insert(other.a[i]);
            }
        }
    }
};

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

// 查询的答案
int ans[MAXN];

// 线性基数组
LinearBasis basis[MAXN];

// 整体二分核心函数
// ql, qr: 查询范围
// vl, vr: 区间范围
void compute(int ql, int qr, int vl, int vr) {
    // 递归边界
```

```
if (ql > qr) {
    return;
}

// 如果区间范围只有一个位置，说明找到了答案
if (vl == vr) {
    for (int i = ql; i <= qr; i++) {
        ans[queries[i].id] = basis[vl].queryMax();
    }
    return;
}

// 二分中点
int mid = (vl + vr) >> 1;

// 构建左半部分的线性基
LinearBasis leftBasis;
leftBasis.init();
for (int i = mid; i >= vl; i--) {
    leftBasis.insert(arr[i]);
    // 这里需要保存中间结果用于后续处理
}

// 构建右半部分的线性基
LinearBasis rightBasis;
rightBasis.init();
for (int i = mid + 1; i <= vr; i++) {
    rightBasis.insert(arr[i]);
    // 这里需要保存中间结果用于后续处理
}

// 检查每个查询，根据区间位置划分到左右区间
int lsiz = 0, rsiz = 0;
for (int i = ql; i <= qr; i++) {
    int l = queries[i].l;
    int r = queries[i].r;

    if (r <= mid) {
        // 查询区间完全在左半部分
        lset[++lsiz] = i;
    } else if (l > mid) {
        // 查询区间完全在右半部分
        rset[++rsiz] = i;
```

```
        } else {
            // 查询区间跨越中点
            // 需要合并左右两部分的线性基
            LinearBasis temp;
            temp.init();
            temp.merge(leftBasis);
            temp.merge(rightBasis);
            ans[queries[i].id] = temp.queryMax();
        }
    }

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

//int main() {
//    ios::sync_with_stdio(false);
//    cin.tie(0);
//    cout.tie(0);
//
//    cin >> n;
//
//    // 读取原始数组
//    for (int i = 1; i <= n; i++) {
//        cin >> arr[i];
//    }
//
//    cin >> m;
//
//    // 读取查询
//    for (int i = 1; i <= m; i++) {
//        int l, r;
//        cin >> l >> r;
```

```
//        queries[i] = Query(l, r, i);
//    }
//
//    // 初始化线性基数组
//    for (int i = 0; i <= n; i++) {
//        basis[i].init();
//    }
//
//    // 整体二分求解
//    compute(1, m, 1, n);
//
//    // 输出结果
//    for (int i = 1; i <= m; i++) {
//        cout << ans[i] << "\n";
//    }
//
//    return 0;
//}
```

==========

文件: CF1100F_Ivan_and_Burgers.java

==========

```
package class168;

// CF1100F Ivan and Burgers - Java 实现
// 题目来源：https://codeforces.com/problemset/problem/1100/F
// 题目描述：
// 给定一个长度为 n 的数组，有 q 次查询，每次查询[l,r]区间内元素异或的最大值。
// 解题思路：使用整体二分算法结合线性基，将所有查询一起处理
// 时间复杂度：O((N+Q) * logN * 32)
// 空间复杂度：O(N * 32)
// 算法适用条件：
// 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. 修改如果对判定答案有贡献，则贡献为确定值
// 4. 贡献满足交换律、结合律，具有可加性
// 5. 题目允许离线操作

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

public class CF1100F_Ivan_and_Burgers {
```

```
public static int MAXN = 500001;
public static int MAXB = 32;  // 线性基的位数，32 位整数
public static int n, m;       // n:数组长度, m:查询次数

// 原始数组，存储输入的数值
public static int[] arr = new int[MAXN];

// 查询信息
public static class Query {
    int l, r, id;  // l:查询区间左端点, r:查询区间右端点, id:查询编号

    public Query(int l, int r, int id) {
        this.l = l;
        this.r = r;
        this.id = id;
    }
}

public static Query[] queries = new Query[MAXN];  // 存储所有查询

// 线性基
// 用于维护一组线性无关的向量，支持插入向量和查询异或最大值
public static class LinearBasis {
    int[] a = new int[MAXB];  // 线性基数组，a[i]表示最高位为 i 的基向量

    // 插入一个向量到线性基中
    public void insert(int x) {
        for (int i = MAXB - 1; i >= 0; i--) {
            // 如果 x 的第 i 位为 0，跳过
            if (((x >> i) & 1) == 0) continue;
            // 如果 a[i]为空，直接插入
            if (a[i] == 0) {
                a[i] = x;
                break;
            }
            // 否则用 a[i]消去 x 的第 i 位
            x ^= a[i];
        }
    }

    // 查询线性基能表示出的最大异或值
    public int queryMax() {
        int res = 0;
```

```
        for (int i = MAXB - 1; i >= 0; i--) {
            // 贪心策略：如果异或a[i]能使结果更大，则异或
            if (((res >> i) & 1) == 0) {
                res ^= a[i];
            }
        }
        return res;
    }

    // 合并另一个线性基
    public void merge(LinearBasis other) {
        for (int i = 0; i < MAXB; i++) {
            if (other.a[i] != 0) {
                insert(other.a[i]);
            }
        }
    }
}

// 整体二分中用于分类查询的临时存储
public static int[] lset = new int[MAXN];  // 左半部分查询
public static int[] rset = new int[MAXN];  // 右半部分查询

// 查询的答案存储数组
public static int[] ans = new int[MAXN];

// 线性基数组，basis[i]表示前 i 个元素构成的线性基
public static LinearBasis[] basis = new LinearBasis[MAXN];

// 整体二分核心函数
// ql, qr: 查询范围
// vl, vr: 区间范围
public static void compute(int ql, int qr, int vl, int vr) {
    // 递归边界：没有查询需要处理
    if (ql > qr) {
        return;
    }

    // 如果区间范围只有一个位置，说明找到了答案
    // 此时所有查询的答案都可以通过basis[vl]直接得到
    if (vl == vr) {
        for (int i = ql; i <= qr; i++) {
            ans[queries[i].id] = basis[vl].queryMax();
```

```
    }
    return;
}

// 二分中点
int mid = (vl + vr) >> 1;

// 构建左半部分的线性基
// 从后往前插入元素，构建[arr[mid], arr[mid-1], ..., arr[vl]]的线性基
LinearBasis leftBasis = new LinearBasis();
for (int i = mid; i >= vl; i--) {
    leftBasis.insert(arr[i]);
}

// 构建右半部分的线性基
// 从前往后插入元素，构建[arr[mid+1], arr[mid+2], ..., arr[vr]]的线性基
LinearBasis rightBasis = new LinearBasis();
for (int i = mid + 1; i <= vr; i++) {
    rightBasis.insert(arr[i]);
}

// 检查每个查询，根据区间位置划分到左右区间
int lsiz = 0, rsiz = 0;
for (int i = ql; i <= qr; i++) {
    int l = queries[i].l;
    int r = queries[i].r;

    if (r <= mid) {
        // 查询区间完全在左半部分
        // 将该查询加入左集合
        lset[++lsiz] = i;
    } else if (l > mid) {
        // 查询区间完全在右半部分
        // 将该查询加入右集合
        rset[++rsiz] = i;
    } else {
        // 查询区间跨越中点
        // 需要合并左右两部分的线性基来计算答案
        LinearBasis temp = new LinearBasis();
        temp.merge(leftBasis);
        temp.merge(rightBasis);
        ans[queries[i].id] = temp.queryMax();
    }
```

```
        }

        // 重新排列查询顺序，使得左集合的查询在前，右集合的查询在后
        int idx = ql;
        for (int i = 1; i <= lsiz; i++) {
            queries[idx++] = queries[lset[i]];
        }
        for (int i = 1; i <= rsiz; i++) {
            queries[idx++] = queries[rset[i]];
        }

        // 递归处理左右两部分
        // 左半部分：区间在[vl, mid]范围内的查询
        compute(ql, ql + lsiz - 1, vl, mid);
        // 右半部分：区间在[mid+1, vr]范围内的查询
        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));

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

        // 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]);
        }

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

        // 读取查询
        for (int i = 1; i <= m; i++) {
            String[] query = br.readLine().split(" ");
            int l = Integer.parseInt(query[0]);
            int r = Integer.parseInt(query[1]);
            queries[i] = new Query(l, r, i);
        }

        // 初始化线性基数组
        for (int i = 0; i <= n; i++) {
            basis[i] = new LinearBasis();
```

```
        }

        // 整体二分求解
        // 初始查询范围[1, m]，初始区间范围[1, n]
        compute(1, m, 1, n);

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

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

==========

文件: CF1100F_Ivan_and_Burgers.py

==========

```
# CF1100F Ivan and Burgers - Python实现
# 题目来源：https://codeforces.com/problemset/problem/1100/F
# 题目描述：
# 给定一个长度为n的数组，有q次查询，每次查询[l,r]区间内元素异或的最大值。
# 时间复杂度：O((N+Q) * logN * 32)
# 空间复杂度：O(N * 32)

import sys

# 由于Python的性能限制，对于大数据可能超时，但在逻辑上是正确的

class LinearBasis:
    def __init__(self):
        self.a = [0] * 32

    def insert(self, x):
        for i in range(31, -1, -1):
            if ((x >> i) & 1) == 0:
                continue
            if self.a[i] == 0:
                self.a[i] = x
                break
```

```
            x ^= self.a[i]

    def query_max(self):
        res = 0
        for i in range(31, -1, -1):
            if ((res >> i) & 1) == 0:
                res ^= self.a[i]
        return res

    def merge(self, other):
        for i in range(32):
            if other.a[i] != 0:
                self.insert(other.a[i])

class Solution:
    def __init__(self):
        self.MAXN = 500001
        self.n = 0
        self.m = 0

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

        # 查询信息
        self.queries = []  # (l, r, id)

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

        # 线性基数组
        self.basis = [LinearBasis() for _ in range(self.MAXN)]

    def compute(self, ql, qr, vl, vr):
        """整体二分核心函数"""
        # 递归边界
        if ql > qr:
            return

        # 如果区间范围只有一个位置，说明找到了答案
        if vl == vr:
            for i in range(ql, qr + 1):
                l, r, id = self.queries[i]
                self.ans[id] = self.basis[vl].query_max()
```

```
    return

# 二分中点
mid = (vl + vr) >> 1

# 构建左半部分的线性基
left_basis = LinearBasis()
for i in range(mid, vl - 1, -1):
    left_basis.insert(self.arr[i])
    # 这里需要保存中间结果用于后续处理

# 构建右半部分的线性基
right_basis = LinearBasis()
for i in range(mid + 1, vr + 1):
    right_basis.insert(self.arr[i])
    # 这里需要保存中间结果用于后续处理

# 检查每个查询，根据区间位置划分到左右区间
lset = []
rset = []
for i in range(ql, qr + 1):
    l, r, id = self.queries[i]

    if r <= mid:
        # 查询区间完全在左半部分
        lset.append(i)
    elif l > mid:
        # 查询区间完全在右半部分
        rset.append(i)
    else:
        # 查询区间跨越中点
        # 需要合并左右两部分的线性基
        temp = LinearBasis()
        temp.merge(left_basis)
        temp.merge(right_basis)
        self.ans[id] = temp.query_max()

# 重新排列查询顺序
new_queries = []
for i in lset:
    new_queries.append(self.queries[i])
for i in rset:
    new_queries.append(self.queries[i])
```

```
        # 更新原查询数组
        for i in range(len(new_queries)):
            self.queries[ql + i] = new_queries[i]

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

    def solve(self):
        """主函数"""
        # 读取输入
        self.n = int(sys.stdin.readline())

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

        self.m = int(sys.stdin.readline())

        # 读取查询
        for i in range(1, self.m + 1):
            query = sys.stdin.readline().split()
            l = int(query[0])
            r = int(query[1])
            self.queries.append((l, r, i))

        # 整体二分求解
        self.compute(0, len(self.queries) - 1, 1, self.n)

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

# 程序入口
#if __name__ == "__main__":
#    solution = Solution()
#    solution.solve()
```

==========

文件: CF603E_PastoralOddities.cpp

```
// ==========================================
// CF603E Pastoral Oddities - C++实现
// 题目来源：https://www.luogu.com.cn/problem/CF603E
// 时间复杂度：O(M * logN * logM)
// 空间复杂度：O(N + M)
//
// 题目大意：
// 给定一张图，每次加边后求一个边集，使得每个点度数为奇数且最大边权最小。
//
// 解题思路：
// 1. 使用整体二分对边权进行二分
// 2. 使用可撤销并查集维护连通性
// 3. 检查所有连通块大小是否都是偶数
// 4. 根据统计结果将操作分为两类递归处理
//
// 算法详解：
// 1. 整体二分：将所有操作一起处理，对边权进行二分，避免对每个查询单独二分
// 2. 可撤销并查集：支持合并操作的回滚，用于维护图的连通性
// 3. 度数检查：通过检查每个连通块的大小是否为偶数来判断是否存在满足条件的边集

//#include <bits/stdc++.h>
//
//using namespace std;
//
//const int MAXN = 100001;
//const int MAXM = 300001;
//int n, m;
//
//// 边的信息
//int u[MAXM], v[MAXM], w[MAXM];
//
//// 查询编号
//int qid[MAXM];
//
//// 并查集
//int father[MAXN], size[MAXN], stack[MAXN], top = 0;
//
//// 整体二分
//int lset[MAXM], rset[MAXM];
//
//// 答案
//int ans[MAXM];
//
```

```
//// 边的辅助结构
//struct Edge {
//    int u, v, w, id;
//    bool operator<(const Edge& other) const {
//        return w < other.w;
//    }
//};
//
//// 初始化并查集
//// 将每个节点初始化为一个独立的集合
//void init() {
//    for (int i = 1; i <= n; i++) {
//        father[i] = i;
//        size[i] = 1;
//    }
//    top = 0;
//}
//
//// 查找根节点（带路径压缩）
//// 使用路径压缩优化，使查找操作的时间复杂度接近 O(1)
//int find(int x) {
//    while (x != father[x]) {
//        x = father[x];
//    }
//    return x;
//}
//
//// 合并两个集合
//// 使用按秩合并优化，将较小的树合并到较大的树上
//// 返回 true 表示成功合并，false 表示已在同一集合中
//bool unionSets(int x, int y) {
//    int fx = find(x);
//    int fy = find(y);
//    if (fx == fy) {
//        return false;
//    }
//    // 按秩合并，将较小的树合并到较大的树上
//    if (size[fx] < size[fy]) {
//        swap(fx, fy);
//    }
//    father[fy] = fx;
//    size[fx] += size[fy];
//    stack[++top] = fy; // 记录修改，用于回滚
```

```
//    return true;
//}
//
//// 检查所有连通块大小是否都是偶数
//// 根据题目要求，每个点的度数必须为奇数，这等价于每个连通块的大小为偶数
//bool check() {
//    for (int i = 1; i <= n; i++) {
//        // 只检查根节点，避免重复计算
//        if (find(i) == i && (size[i] & 1)) {
//            // 如果存在大小为奇数的连通块，则无法满足条件
//            return false;
//        }
//    }
//    return true;
//}
//
//// 回滚操作
//// 将并查集的状态回滚到 targetTop 时刻
//void rollback(int targetTop) {
//    while (top > targetTop) {
//        int y = stack[top--];
//        int fy = find(y);
//        size[fy] -= size[y];
//        father[y] = y;
//    }
//}
//
//// 整体二分核心函数
//// ql, qr: 当前处理的查询范围
//// vl, vr: 当前处理的值域范围（边权范围）
//void compute(int ql, int qr, int vl, int vr) {
//    // 递归边界：没有查询需要处理
//    if (ql > qr) {
//        return;
//    }
//    // 递归边界：值域范围只有一个值，找到了答案
//    if (vl == vr) {
//        for (int i = ql; i <= qr; i++) {
//            ans[qid[i]] = vl;
//        }
//        return;
//    }
//
```

```
//    // 二分中点
//    int mid = (vl + vr) >> 1;
//    int targetTop = top;
//
//    // 添加编号小于等于 mid 的边到并查集中
//    for (int i = vl; i <= mid; i++) {
//        unionSets(u[i], v[i]);
//    }
//
//    // 检查每个查询，并根据结果将查询分为两类
//    int lsiz = 0, rsiz = 0;
//    for (int i = ql; i <= qr; i++) {
//        int id = qid[i];
//        // 检查当前状态是否满足条件
//        if (check()) {
//            // 满足条件，答案可能更小，分到左区间
//            lset[++lsiz] = id;
//        } else {
//            // 不满足条件，答案必须更大，分到右区间
//            rset[++rsiz] = id;
//        }
//    }
//
//    // 重新排列查询顺序，使左区间的查询在前，右区间的查询在后
//    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];
//    }
//
//    // 回滚操作，恢复到添加边之前的状态
//    rollback(targetTop);
//
//    // 递归处理左右两部分
//    // 左区间：答案范围[vl, mid]
//    compute(ql, ql + lsiz - 1, vl, mid);
//    // 右区间：答案范围[mid+1, vr]
//    compute(ql + lsiz, qr, mid + 1, vr);
//}
//
//int main() {
//    ios::sync_with_stdio(false);
```

```
//    cin.tie(nullptr);
//    cin >> n >> m;
//
//    // 读取边信息
//    vector<Edge> edges(m + 1);
//    for (int i = 1; i <= m; i++) {
//        cin >> edges[i].u >> edges[i].v >> edges[i].w;
//        u[i] = edges[i].u;
//        v[i] = edges[i].v;
//        w[i] = edges[i].w;
//        edges[i].id = i;
//    }
//
//    // 按权重排序，这是整体二分的前提条件
//    sort(edges.begin() + 1, edges.end());
//
//    // 重新编号，使边按权重递增顺序排列
//    for (int i = 1; i <= m; i++) {
//        u[i] = edges[i].u;
//        v[i] = edges[i].v;
//        w[i] = edges[i].w;
//        qid[i] = edges[i].id;
//    }
//
//    // 初始化并查集
//    init();
//
//    // 整体二分求解
//    compute(1, m, 1, m);
//
//    // 输出结果
//    vector<int> result(m + 1);
//    for (int i = 1; i <= m; i++) {
//        result[qid[i]] = ans[qid[i]];
//    }
//
//    int maxWeight = -1;
//    for (int i = 1; i <= m; i++) {
//        if (result[i] > 0) {
//            maxWeight = max(maxWeight, w[result[i]]);
//        }
//        if (maxWeight > 0) {
//            cout << maxWeight << "\n";
```

```
//        } else {
//            cout << "-1\n";
//        }
//    }
//
//    return 0;
//}
```

==========

文件: CF603E_PastoralOddities.java

==========

```
package class168;

// CF603E Pastoral Oddities - Java 实现
// 题目来源：https://www.luogu.com.cn/problem/CF603E
// 时间复杂度：O(M * logN * logM)
// 空间复杂度：O(N + M)
//
// 题目大意：
// 给定一张图，每次加边后求一个边集，使得每个点度数为奇数且最大边权最小。
//
// 解题思路：
// 1. 使用整体二分对边权进行二分
// 2. 使用可撤销并查集维护连通性
// 3. 检查所有连通块大小是否都是偶数
// 4. 根据统计结果将操作分为两类递归处理
//
// 算法详解：
// 1. 整体二分：将所有操作一起处理，对边权进行二分，避免对每个查询单独二分
// 2. 可撤销并查集：支持合并操作的回滚，用于维护图的连通性
// 3. 度数检查：通过检查每个连通块的大小是否为偶数来判断是否存在满足条件的边集

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

public class CF603E_PastoralOddities {
    public static int MAXN = 100001;
    public static int MAXM = 300001;
    public static int n, m;

    // 边的信息
    public static int[] u = new int[MAXM];
```

```
public static int[] v = new int[MAXM];
public static int[] w = new int[MAXM];

// 查询编号
public static int[] qid = new int[MAXM];

// 并查集
public static int[] father = new int[MAXN];
public static int[] size = new int[MAXN];
public static int[] stack = new int[MAXN];
public static int top = 0;

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

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

// 初始化并查集
// 将每个节点初始化为一个独立的集合
public static void init() {
    for (int i = 1; i <= n; i++) {
        father[i] = i;
        size[i] = 1;
    }
    top = 0;
}

// 查找根节点（带路径压缩）
// 使用路径压缩优化，使查找操作的时间复杂度接近O(1)
public static int find(int x) {
    while (x != father[x]) {
        x = father[x];
    }
    return x;
}

// 合并两个集合
// 使用按秩合并优化，将较小的树合并到较大的树上
// 返回true表示成功合并，false表示已在同一集合中
public static boolean union(int x, int y) {
    int fx = find(x);
```

```
    int fy = find(y);
    if (fx == fy) {
        return false;
    }
    // 按秩合并，将较小的树合并到较大的树上
    if (size[fx] < size[fy]) {
        int temp = fx;
        fx = fy;
        fy = temp;
    }
    father[fy] = fx;
    size[fx] += size[fy];
    stack[++top] = fy; // 记录修改，用于回滚
    return true;
}

// 检查所有连通块大小是否都是偶数
// 根据题目要求，每个点的度数必须为奇数，这等价于每个连通块的大小为偶数
public static boolean check() {
    for (int i = 1; i <= n; i++) {
        // 只检查根节点，避免重复计算
        if (find(i) == i && (size[i] & 1) == 1) {
            // 如果存在大小为奇数的连通块，则无法满足条件
            return false;
        }
    }
    return true;
}

// 回滚操作
// 将并查集的状态回滚到 targetTop 时刻
public static void rollback(int targetTop) {
    while (top > targetTop) {
        int y = stack[top--];
        int fy = find(y);
        size[fy] -= size[y];
        father[y] = y;
    }
}

// 整体二分核心函数
// ql, qr: 当前处理的查询范围
// vl, vr: 当前处理的值域范围（边权范围）
```

```
public static void compute(int ql, int qr, int vl, int vr) {
    // 递归边界：没有查询需要处理
    if (ql > qr) {
        return;
    }
    // 递归边界：值域范围只有一个值，找到了答案
    if (vl == vr) {
        for (int i = ql; i <= qr; i++) {
            ans[qid[i]] = vl;
        }
        return;
    }

    // 二分中点
    int mid = (vl + vr) >> 1;
    // 记录当前状态，用于后续回滚
    int targetTop = top;

    // 添加编号小于等于 mid 的边到并查集中
    for (int i = vl; i <= mid; i++) {
        union(u[i], v[i]);
    }

    // 检查每个查询，并根据结果将查询分为两类
    int lsiz = 0, rsiz = 0;
    for (int i = ql; i <= qr; i++) {
        int id = qid[i];
        // 检查当前状态是否满足条件
        if (check()) {
            // 满足条件，答案可能更小，分到左区间
            lset[++lsiz] = id;
        } else {
            // 不满足条件，答案必须更大，分到右区间
            rset[++rsiz] = id;
        }
    }

    // 重新排列查询顺序，使左区间的查询在前，右区间的查询在后
    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];
```

```
        }

        // 回滚操作，恢复到添加边之前的状态
        rollback(targetTop);

        // 递归处理左右两部分
        // 左区间：答案范围[vl, mid]
        compute(ql, ql + lsiz - 1, vl, mid);
        // 右区间：答案范围[mid+1, vr]
        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
        Edge[] edges = new Edge[m + 1];
        for (int i = 1; i <= m; i++) {
            String[] edge = br.readLine().split(" ");
            u[i] = Integer.parseInt(edge[0]);
            v[i] = Integer.parseInt(edge[1]);
            w[i] = Integer.parseInt(edge[2]);
            edges[i] = new Edge(u[i], v[i], w[i], i);
        }

        // 按权重排序，这是整体二分的前提条件
        Arrays.sort(edges, 1, m + 1);

        // 重新编号，使边按权重递增顺序排列
        for (int i = 1; i <= m; i++) {
            u[i] = edges[i].u;
            v[i] = edges[i].v;
            w[i] = edges[i].w;
            qid[i] = edges[i].id;
        }

        // 初始化并查集
        init();
```

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

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

        int maxWeight = -1;
        for (int i = 1; i <= m; i++) {
            if (result[i] > 0) {
                maxWeight = Math.max(maxWeight, w[result[i]]);
            }
            if (maxWeight > 0) {
                out.println(maxWeight);
            } else {
                out.println(-1);
            }
        }

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

    // 边的辅助类，用于排序
    static class Edge implements Comparable<Edge> {
        int u, v, w, id;

        public Edge(int u, int v, int w, int id) {
            this.u = u;
            this.v = v;
            this.w = w;
            this.id = id;
        }

        @Override
        public int compareTo(Edge other) {
            return this.w - other.w;
        }
    }
}
```

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

文件: CF603E_PastoralOddities.py
==========================================
# CF603E Pastoral Oddities - Python 实现
# 题目来源：https://www.luogu.com.cn/problem/CF603E
# 时间复杂度：O(M * logN * logM)
# 空间复杂度：O(N + M)
#
# 题目大意：
# 给定一张图，每次加边后求一个边集，使得每个点度数为奇数且最大边权最小。
#
# 解题思路：
# 1. 使用整体二分对边权进行二分
# 2. 使用可撤销并查集维护连通性
# 3. 检查所有连通块大小是否都是偶数
# 4. 根据统计结果将操作分为两类递归处理
#
# 算法详解：
# 1. 整体二分：将所有操作一起处理，对边权进行二分，避免对每个查询单独二分
# 2. 可撤销并查集：支持合并操作的回滚，用于维护图的连通性
# 3. 度数检查：通过检查每个连通块的大小是否为偶数来判断是否存在满足条件的边集

import sys

class PastoralOddities:
    def __init__(self):
        self.MAXN = 100001
        self.MAXM = 300001
        self.n = 0
        self.m = 0

        # 边的信息
        self.u = [0] * self.MAXM
        self.v = [0] * self.MAXM
        self.w = [0] * self.MAXM

        # 查询编号
        self.qid = [0] * self.MAXM

        # 并查集
        self.father = [0] * self.MAXN
```

```
    self.size = [0] * self.MAXN
    self.stack = [0] * self.MAXN
    self.top = 0

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

    # 答案
    self.ans = [0] * self.MAXM

# 初始化并查集
# 将每个节点初始化为一个独立的集合
def init(self):
    for i in range(1, self.n + 1):
        self.father[i] = i
        self.size[i] = 1
    self.top = 0

# 查找根节点（带路径压缩）
# 使用路径压缩优化，使查找操作的时间复杂度接近 O(1)
def find(self, x):
    while x != self.father[x]:
        x = self.father[x]
    return x

# 合并两个集合
# 使用按秩合并优化，将较小的树合并到较大的树上
# 返回 True 表示成功合并，False 表示已在同一集合中
def union(self, x, y):
    fx = self.find(x)
    fy = self.find(y)
    if fx == fy:
        return False
    # 按秩合并，将较小的树合并到较大的树上
    if self.size[fx] < self.size[fy]:
        fx, fy = fy, fx
    self.father[fy] = fx
    self.size[fx] += self.size[fy]
    self.top += 1
    self.stack[self.top] = fy  # 记录修改，用于回滚
    return True
```

```
# 检查所有连通块大小是否都是偶数
# 根据题目要求，每个点的度数必须为奇数，这等价于每个连通块的大小为偶数
def check(self):
    for i in range(1, self.n + 1):
        # 只检查根节点，避免重复计算
        if self.find(i) == i and (self.size[i] & 1) == 1:
            # 如果存在大小为奇数的连通块，则无法满足条件
            return False
    return True

# 回滚操作
# 将并查集的状态回滚到 targetTop 时刻
def rollback(self, targetTop):
    while self.top > targetTop:
        y = self.stack[self.top]
        self.top -= 1
        fy = self.find(y)
        self.size[fy] -= self.size[y]
        self.father[y] = y

# 整体二分核心函数
# ql, qr: 当前处理的查询范围
# vl, vr: 当前处理的值域范围（边权范围）
def compute(self, ql, qr, vl, vr):
    # 递归边界：没有查询需要处理
    if ql > qr:
        return
    # 递归边界：值域范围只有一个值，找到了答案
    if vl == vr:
        for i in range(ql, qr + 1):
            self.ans[self.qid[i]] = vl
        return

    mid = (vl + vr) >> 1
    targetTop = self.top

    # 添加编号小于等于 mid 的边到并查集中
    for i in range(vl, mid + 1):
        self.union(self.u[i], self.v[i])

    # 检查每个查询，并根据结果将查询分为两类
    lsiz = 0
    rsiz = 0
```

```
    for i in range(ql, qr + 1):
        id = self.qid[i]
        # 检查当前状态是否满足条件
        if self.check():
            # 满足条件，答案可能更小，分到左区间
            lsiz += 1
            self.lset[lsiz] = id
        else:
            # 不满足条件，答案必须更大，分到右区间
            rsiz += 1
            self.rset[rsiz] = id

    # 重新排列查询顺序，使左区间的查询在前，右区间的查询在后
    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]

    # 回滚操作，恢复到添加边之前的状态
    self.rollback(targetTop)

    # 递归处理左右两部分
    # 左区间：答案范围[vl, mid]
    self.compute(ql, ql + lsiz - 1, vl, mid)
    # 右区间：答案范围[mid+1, vr]
    self.compute(ql + lsiz, qr, mid + 1, vr)

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

    # 读取边信息
    edges = []
    for i in range(1, self.m + 1):
        edge = sys.stdin.readline().split()
        self.u[i] = int(edge[0])
        self.v[i] = int(edge[1])
        self.w[i] = int(edge[2])
        edges.append((self.w[i], self.u[i], self.v[i], i))

    # 按权重排序，这是整体二分的前提条件
    edges.sort()
```

```
        # 重新编号，使边按权重递增顺序排列
        for i in range(self.m):
            _, u, v, id = edges[i]
            self.u[i + 1] = u
            self.v[i + 1] = v
            self.w[i + 1] = edges[i][0]
            self.qid[i + 1] = id

        # 初始化并查集
        self.init()

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

        # 输出结果
        result = [0] * (self.m + 1)
        for i in range(1, self.m + 1):
            result[self.qid[i]] = self.ans[self.qid[i]]

        maxWeight = -1
        results = []
        for i in range(1, self.m + 1):
            if result[i] > 0:
                maxWeight = max(maxWeight, self.w[result[i]])
            if maxWeight > 0:
                results.append(str(maxWeight))
            else:
                results.append("-1")

        print("\n".join(results))

# 程序入口
#if __name__ == "__main__":
#    solution = PastoralOddities()
#    solution.solve()
```

==========

文件: check_python_syntax.py

==========

```
import os
import sys
```

```
import subprocess

def check_python_syntax(file_path):
    """检查 Python 文件语法"""
    try:
        # 使用 Python -m py_compile 检查语法
        result = subprocess.run([sys.executable, '-m', 'py_compile', file_path],
                                capture_output=True, text=True)
        if result.returncode == 0:
            print(f"✓ {file_path} 语法正确")
            return True
        else:
            print(f"✗ {file_path} 语法错误:")
            print(result.stderr)
            return False
    except Exception as e:
        print(f"✗ 检查 {file_path} 时出错: {e}")
        return False

def main():
    # 获取当前目录
    current_dir = "d:\\Upan\\src\\algorithm-journey\\src\\algorithm-journey\\src\\class168"

    # 查找所有 Python 文件
    python_files = []
    for file in os.listdir(current_dir):
        if file.endswith('.py'):
            python_files.append(os.path.join(current_dir, file))

    # 检查每个 Python 文件
    all_correct = True
    for file_path in python_files:
        if not check_python_syntax(file_path):
            all_correct = False

    if all_correct:
        print("\n所有 Python 文件语法检查通过!")
    else:
        print("\n存在语法错误的文件，请检查上面的输出。")
        sys.exit(1)

if __name__ == "__main__":
    main()
```

==========

文件: Code01_RangeKth1.java
==========

```
package class168;

// 区间内第 k 小，第一种写法，java 版
// 给定一个长度为 n 的数组，接下来有 m 条查询，格式如下
// 查询 l r k : 打印[l..r]范围内第 k 小的值
// 1 <= n、m <= 2 * 10^5
// 1 <= 数组中的数字 <= 10^9
// 测试链接 : https://www.luogu.com.cn/problem/P3834
// 本题是讲解 157，可持久化线段树模版题，现在作为整体二分的模版题
// 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
//
// 整体二分算法详解：
// 1. 算法思想：将所有查询一起进行二分，而不是对每个查询单独二分
// 2. 适用场景：多个具有二分性质的查询
// 3. 时间复杂度：O((N+M) * logN * log(maxValue))
// 4. 空间复杂度：O(N+M)
// 5. 核心思想：
//    - 离线处理所有查询
//    - 对值域进行二分
//    - 使用数据结构维护状态
//    - 根据结果将查询分类递归处理

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

public class Code01_RangeKth1 {

    public static int MAXN = 200001;
    public static int n, m;

    // 位置 i，数值 v
    public static int[][] arr = new int[MAXN][2];

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

```
public static int[] l = new int[MAXN];
public static int[] r = new int[MAXN];
public static int[] k = 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];

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

// 树状数组中增加 i 位置的词频
public static void add(int i, int v) {
    while (i <= n) {
        tree[i] += v;
        i += lowbit(i);
    }
}

// 树状数组中查询[1~i]范围的词频累加和
public static int sum(int i) {
    int ret = 0;
    while (i > 0) {
        ret += tree[i];
        i -= lowbit(i);
    }
    return ret;
}

// 树状数组中查询[l~r]范围的词频累加和
public static int query(int l, int r) {
    return sum(r) - sum(l - 1);
}

// 整体二分的第一种写法
```

```
// 问题范围[ql..qr]，答案范围[vl..vr]，答案范围的每个下标都是数字的排名
public static void compute(int ql, int qr, int vl, int vr) {
    if (ql > qr) {
        return;
    }
    if (vl == vr) {
        // 找到答案，记录结果
        for (int i = ql; i <= qr; i++) {
            ans[qid[i]] = arr[vl][1];
        }
    } else {
        // 修改数据状况
        int mid = (vl + vr) / 2;
        // 将值域小于等于 mid 的数加入树状数组
        for (int i = vl; i <= mid; i++) {
            add(arr[i][0], 1);
        }
        // 检查每个问题并划分左右
        int lsiz = 0, rsiz = 0;
        for (int i = ql; i <= qr; i++) {
            int id = qid[i];
            // 查询区间内小于等于 arr[mid][1]的数的个数
            int satisfy = query(l[id], r[id]);
            if (satisfy >= k[id]) {
                // 说明第 k 小的数在左半部分
                lset[++lsiz] = id;
            } 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
                k[id] -= satisfy;
                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];
        }
        // 撤回数据状况
        for (int i = vl; i <= mid; i++) {
            add(arr[i][0], -1);
        }
```

```
        // 左右两侧各自递归
        compute(ql, ql + lsiz - 1, vl, mid);
        compute(ql + lsiz, qr, 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();
    m = in.nextInt();
    for (int i = 1; i <= n; i++) {
        arr[i][0] = i;
        arr[i][1] = in.nextInt();
    }
    for (int i = 1; i <= m; i++) {
        qid[i] = i;
        l[i] = in.nextInt();
        r[i] = in.nextInt();
        k[i] = in.nextInt();
    }
    // 离散化，按数值排序
    Arrays.sort(arr, 1, n + 1, (a, b) -> a[1] - b[1]);
    compute(1, m, 1, n);
    for (int i = 1; i <= m; 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: Code01_RangeKth2.java

==========

```
package class168;

// 区间内第 k 小，第二种写法，java 版
// 给定一个长度为 n 的数组，接下来有 m 条查询，格式如下
// 查询 l r k : 打印[l..r]范围内第 k 小的值
// 1 <= n、m <= 2 * 10^5
// 1 <= 数组中的数字 <= 10^9
// 测试链接 : https://www.luogu.com.cn/problem/P3834
// 本题是讲解 157，可持久化线段树模版题，现在作为整体二分的模版题
// 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 Code01_RangeKth2 {

    public static int MAXN = 200001;
    public static int n, m;

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

    public static int[] qid = new int[MAXN];
    public static int[] l = new int[MAXN];
    public static int[] r = new int[MAXN];
    public static int[] k = new int[MAXN];

    public static int[] tree = new int[MAXN];
    // 数据的使用数量
    public static int used = 0;

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

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

    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);
}

// 整体二分的第二种写法
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][1];
        }
    } else {
        int mid = (vl + vr) / 2;
        // 数据不够就叠加
        while (used < mid) {
            used++;
            add(arr[used][0], 1);
        }
        // 数据超了就撤销
        while (used > mid) {
            add(arr[used][0], -1);
            used--;
        }
        int lsiz = 0, rsiz = 0;
        for (int i = ql; i <= qr; i++) {
            int id = qid[i];
            int satisfy = query(l[id], r[id]);
            if (satisfy >= k[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);
    }
}

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 <= n; i++) {
        arr[i][0] = i;
        arr[i][1] = in.nextInt();
    }
    for (int i = 1; i <= m; i++) {
        qid[i] = i;
        l[i] = in.nextInt();
        r[i] = in.nextInt();
        k[i] = in.nextInt();
    }
    Arrays.sort(arr, 1, n + 1, (a, b) -> a[1] - b[1]);
    compute(1, m, 1, n);
    for (int i = 1; i <= m; 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: Code01_RangeKth3.java

==========

```
package class168;

// 区间内第 k 小，第一种写法，C++版
// 给定一个长度为 n 的数组，接下来有 m 条查询，格式如下
// 查询 l r k : 打印[l..r]范围内第 k 小的值
// 1 <= n、m <= 2 * 10^5
// 1 <= 数组中的数字 <= 10^9
// 测试链接 : https://www.luogu.com.cn/problem/P3834
// 本题是讲解 157，可持久化线段树模版题，现在作为整体二分的模版题
```

```
// 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;
//
//struct Number {
//    int i, v;
//};
//
//bool NumberCmp(Number x, Number y) {
//    return x.v < y.v;
//}
//
//const int MAXN = 200001;
//int n, m;
//
//Number arr[MAXN];
//
//int qid[MAXN];
//int l[MAXN];
//int r[MAXN];
//int k[MAXN];
//
//int tree[MAXN];
//
//int lset[MAXN];
//int rset[MAXN];
//
//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 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].v;
//        }
//    } else {
//        int mid = (vl + vr) >> 1;
//        for (int i = vl; i <= mid; i++) {
//            add(arr[i].i, 1);
//        }
//        int lsiz = 0, rsiz = 0;
//        for (int i = ql; i <= qr; i++) {
//            int id = qid[i];
//            int satisfy = query(l[id], r[id]);
//            if (satisfy >= k[id]) {
//                lset[++lsiz] = id;
//            } else {
//                k[id] -= satisfy;
//                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];
//        }
```

```
//        for (int i = vl; i <= mid; i++) {
//            add(arr[i].i, -1);
//        }
//        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++) {
//        arr[i].i = i;
//        cin >> arr[i].v;
//    }
//    for (int i = 1; i <= m; i++) {
//        qid[i] = i;
//        cin >> l[i] >> r[i] >> k[i];
//    }
//    sort(arr + 1, arr + n + 1, NumberCmp);
//    compute(1, m, 1, n);
//    for (int i = 1; i <= m; i++) {
//        cout << ans[i] << '\n';
//    }
//    return 0;
//}
```

==========

文件: Code01_RangeKth4.java

==========

```
package class168;

// 区间内第 k 小，第二种写法，C++版
// 给定一个长度为 n 的数组，接下来有 m 条查询，格式如下
// 查询 l r k : 打印[l..r]范围内第 k 小的值
// 1 <= n、m <= 2 * 10^5
// 1 <= 数组中的数字 <= 10^9
// 测试链接 : https://www.luogu.com.cn/problem/P3834
// 本题是讲解 157，可持久化线段树模版题，现在作为整体二分的模版题
// 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;
//
//struct Number {
//    int i, v;
//};
//
//bool NumberCmp(Number x, Number y) {
//    return x.v < y.v;
//}
//
//const int MAXN = 200001;
//int n, m;
//
//Number arr[MAXN];
//
//int qid[MAXN];
//int l[MAXN];
//int r[MAXN];
//int k[MAXN];
//
//int tree[MAXN];
//int used = 0;
//
//int lset[MAXN];
//int rset[MAXN];
//
//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 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].v;
//        }
//    } else {
//        int mid = (vl + vr) / 2;
//        while (used < mid) {
//            used++;
//            add(arr[used].i, 1);
//        }
//        while (used > mid) {
//            add(arr[used].i, -1);
//            used--;
//        }
//        int lsiz = 0, rsiz = 0;
//        for (int i = ql; i <= qr; i++) {
//            int id = qid[i];
//            int satisfy = query(l[id], r[id]);
//            if (satisfy >= k[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++) {
//        arr[i].i = i;
//        cin >> arr[i].v;
//    }
//    for (int i = 1; i <= m; i++) {
//        qid[i] = i;
//        cin >> l[i] >> r[i] >> k[i];
//    }
//    sort(arr + 1, arr + n + 1, NumberCmp);
//    compute(1, m, 1, n);
//    for (int i = 1; i <= m; i++) {
//        cout << ans[i] << '\n';
//    }
//    return 0;
//}
```

==========

文件: Code02_MatrixKth1.java

==========

```
package class168;

// 矩阵内第k小，第一种写法，java版
// 给定一个n * n的矩阵，接下来有q条查询，格式如下
// 查询 a b c d k : 左上角(a, b)，右下角(c, d)，打印该区域中第k小的数
// 1 <= n <= 500
// 1 <= q <= 6 * 10^4
// 0 <= 矩阵中的数字 <= 10^9
// 测试链接 : https://www.luogu.com.cn/problem/P1527
// 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 Code02_MatrixKth1 {

    public static int MAXN = 501;
    public static int MAXQ = 1000001;
    public static int n, q;

    // 矩阵中的每个数字，所在行 x、所在列 y、数值 v
    public static int[][] xyv = new int[MAXN * MAXN][3];
    // 矩阵中一共有多少个数字，cntv 就是矩阵的规模
    public static int cntv = 0;

    // 查询任务的编号
    public static int[] qid = new int[MAXQ];
    // 查询范围的左上角坐标
    public static int[] a = new int[MAXQ];
    public static int[] b = new int[MAXQ];
    // 查询范围的右下角坐标
    public static int[] c = new int[MAXQ];
    public static int[] d = new int[MAXQ];
    // 查询矩阵内第 k 小
    public static int[] k = new int[MAXQ];

    // 二维树状数组
    public static int[][] tree = new int[MAXN][MAXN];

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

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

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

    // 二维空间中，(x,y)位置的词频加 v
```

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

// 二维空间中，左上角(1,1)到右下角(x,y)范围上的词频累加和
public static int sum(int x, int y) {
    int ret = 0;
    for (int i = x; i > 0; i -= lowbit(i)) {
        for (int j = y; j > 0; j -= lowbit(j)) {
            ret += tree[i][j];
        }
    }
    return ret;
}

// 二维空间中，左上角(a,b)到右下角(c,d)范围上的词频累加和
public static int query(int a, int b, int c, int d) {
    return sum(c, d) - sum(a - 1, d) - sum(c, b - 1) + sum(a - 1, b - 1);
}

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]] = xyv[vl][2];
        }
    } else {
        int mid = (vl + vr) >> 1;
        for (int i = vl; i <= mid; i++) {
            add(xyv[i][0], xyv[i][1], 1);
        }
        int lsiz = 0, rsiz = 0;
        for (int i = ql; i <= qr; i++) {
            int id = qid[i];
            int satisfy = query(a[id], b[id], c[id], d[id]);
            if (satisfy >= k[id]) {
                lset[++lsiz] = id;
```

```
            } else {
                k[id] -= satisfy;
                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];
        }
        for (int i = vl; i <= mid; i++) {
            add(xyv[i][0], xyv[i][1], -1);
        }
        compute(ql, ql + lsiz - 1, vl, mid);
        compute(ql + lsiz, qr, 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();
    q = in.nextInt();
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= n; j++) {
            xyv[++cntv][0] = i;
            xyv[cntv][1] = j;
            xyv[cntv][2] = in.nextInt();
        }
    }
    for (int i = 1; i <= q; i++) {
        qid[i] = i;
        a[i] = in.nextInt();
        b[i] = in.nextInt();
        c[i] = in.nextInt();
        d[i] = in.nextInt();
        k[i] = in.nextInt();
    }
    Arrays.sort(xyv, 1, cntv + 1, (a, b) -> a[2] - b[2]);
    compute(1, q, 1, cntv);
    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: Code02_MatrixKth2.java
==========================================
package class168;

// 矩阵内第k小，第二种写法，java版
// 给定一个n * n的矩阵，接下来有q条查询，格式如下
// 查询 a b c d k : 左上角(a, b)，右下角(c, d)，打印该区域中第k小的数
// 1 <= n <= 500
// 1 <= q <= 6 * 10^4
// 0 <= 矩阵中的数字 <= 10^9
// 测试链接 : https://www.luogu.com.cn/problem/P1527
// 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 Code02_MatrixKth2 {

    public static int MAXN = 501;
    public static int MAXQ = 1000001;
    public static int n, q;

    public static int[][] xyv = new int[MAXN * MAXN][3];
    public static int cntv = 0;

    public static int[] qid = new int[MAXQ];
    public static int[] a = new int[MAXQ];
    public static int[] b = new int[MAXQ];
    public static int[] c = new int[MAXQ];
    public static int[] d = new int[MAXQ];
    public static int[] k = new int[MAXQ];

    public static int[][] tree = new int[MAXN][MAXN];
    // 数据的使用数量
    public static int used = 0;
```

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

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

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

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

public static int sum(int x, int y) {
    int ret = 0;
    for (int i = x; i > 0; i -= lowbit(i)) {
        for (int j = y; j > 0; j -= lowbit(j)) {
            ret += tree[i][j];
        }
    }
    return ret;
}

public static int query(int a, int b, int c, int d) {
    return sum(c, d) - sum(a - 1, d) - sum(c, b - 1) + sum(a - 1, b - 1);
}

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]] = xyv[vl][2];
        }
    } else {
        int mid = (vl + vr) >> 1;
        int lsiz = 0, rsiz = 0;
```

```
        while (used < mid) {
            used++;
            add(xyv[used][0], xyv[used][1], 1);
        }
        while (used > mid) {
            add(xyv[used][0], xyv[used][1], -1);
            used--;
        }
        for (int i = ql; i <= qr; i++) {
            int id = qid[i];
            int satisfy = query(a[id], b[id], c[id], d[id]);
            if (satisfy >= k[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);
    }
}

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();
    q = in.nextInt();
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= n; j++) {
            xyv[++cntv][0] = i;
            xyv[cntv][1] = j;
            xyv[cntv][2] = in.nextInt();
        }
    }
    for (int i = 1; i <= q; i++) {
        qid[i] = i;
        a[i] = in.nextInt();
```

```
            b[i] = in.nextInt();
            c[i] = in.nextInt();
            d[i] = in.nextInt();
            k[i] = in.nextInt();
        }
        Arrays.sort(xyv, 1, cntv + 1, (a, b) -> a[2] - b[2]);
        compute(1, q, 1, cntv);
        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: Code02_MatrixKth3.java

==========

```
package class168;

// 矩阵内第 k 小，第一种写法，C++版
// 给定一个 n * n 的矩阵，接下来有 q 条查询，格式如下
// 查询 a b c d k : 左上角(a, b)，右下角(c, d)，打印该区域中第 k 小的数
// 1 <= n <= 500
// 1 <= q <= 6 * 10^4
// 0 <= 矩阵中的数字 <= 10^9
// 测试链接 : https://www.luogu.com.cn/problem/P1527
// 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 Number {
//    int x, y, v;
//};
//
//bool NumberCmp(Number a, Number b) {
//    return a.v < b.v;
//}
//
//const int MAXN = 501;
//const int MAXQ = 1000001;
//int n, q;
```

```
//
//Number xyv[MAXN * MAXN];
//int cntv = 0;
//
//int qid[MAXQ];
//int a[MAXQ];
//int b[MAXQ];
//int c[MAXQ];
//int d[MAXQ];
//int k[MAXQ];
//
//int tree[MAXN][MAXN];
//
//int lset[MAXQ];
//int rset[MAXQ];
//
//int ans[MAXQ];
//
//int lowbit(int i) {
//    return i & -i;
//}
//
//void add(int x, int y, int v) {
//    for (int i = x; i <= n; i += lowbit(i)) {
//        for (int j = y; j <= n; j += lowbit(j)) {
//            tree[i][j] += v;
//        }
//    }
//}
//
//int sum(int x, int y) {
//    int ret = 0;
//    for (int i = x; i > 0; i -= lowbit(i)) {
//        for (int j = y; j > 0; j -= lowbit(j)) {
//            ret += tree[i][j];
//        }
//    }
//    return ret;
//}
//
//int query(int a, int b, int c, int d) {
//    return sum(c, d) - sum(a - 1, d) - sum(c, b - 1) + sum(a - 1, b - 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]] = xyv[vl].v;
//        }
//    } else {
//        int mid = (vl + vr) >> 1;
//        for (int i = vl; i <= mid; i++) {
//            add(xyv[i].x, xyv[i].y, 1);
//        }
//        int lsiz = 0, rsiz = 0;
//        for (int i = ql; i <= qr; i++) {
//            int id = qid[i];
//            int satisfy = query(a[id], b[id], c[id], d[id]);
//            if (satisfy >= k[id]) {
//                lset[++lsiz] = id;
//            } else {
//                k[id] -= satisfy;
//                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];
//        }
//        for (int i = vl; i <= mid; i++) {
//            add(xyv[i].x, xyv[i].y, -1);
//        }
//        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 >> q;
```

```
//    for (int i = 1; i <= n; i++) {
//        for (int j = 1; j <= n; j++) {
//            xyv[++cntv].x = i;
//            xyv[cntv].y = j;
//            cin >> xyv[cntv].v;
//        }
//    }
//    for (int i = 1; i <= q; i++) {
//        qid[i] = i;
//        cin >> a[i] >> b[i] >> c[i] >> d[i] >> k[i];
//    }
//    sort(xyv + 1, xyv + cntv + 1, NumberCmp);
//    compute(1, q, 1, cntv);
//    for (int i = 1; i <= q; i++) {
//        cout << ans[i] << '\n';
//    }
//    return 0;
//}
```

==========

文件: Code02_MatrixKth4.java

==========

```
package class168;

// 矩阵内第 k 小，第二种写法，C++版
// 给定一个 n * n 的矩阵，接下来有 q 条查询，格式如下
// 查询 a b c d k : 左上角(a, b)，右下角(c, d)，打印该区域中第 k 小的数
// 1 <= n <= 500
// 1 <= q <= 6 * 10^4
// 0 <= 矩阵中的数字 <= 10^9
// 测试链接 : https://www.luogu.com.cn/problem/P1527
// 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 Number {
//    int x, y, v;
//};
//
```

```
//bool NumberCmp(Number a, Number b) {
//    return a.v < b.v;
//}
//
//const int MAXN = 501;
//const int MAXQ = 1000001;
//int n, q;
//
//Number xyv[MAXN * MAXN];
//int cntv = 0;
//
//int qid[MAXQ];
//int a[MAXQ];
//int b[MAXQ];
//int c[MAXQ];
//int d[MAXQ];
//int k[MAXQ];
//
//int tree[MAXN][MAXN];
//int used = 0;
//
//int lset[MAXQ];
//int rset[MAXQ];
//
//int ans[MAXQ];
//
//int lowbit(int i) {
//    return i & -i;
//}
//
//void add(int x, int y, int v) {
//    for (int i = x; i <= n; i += lowbit(i)) {
//        for (int j = y; j <= n; j += lowbit(j)) {
//            tree[i][j] += v;
//        }
//    }
//}
//
//int sum(int x, int y) {
//    int ret = 0;
//    for (int i = x; i > 0; i -= lowbit(i)) {
//        for (int j = y; j > 0; j -= lowbit(j)) {
//            ret += tree[i][j];
```

```
//        }
//    }
//    return ret;
//}
//
//int query(int a, int b, int c, int d) {
//    return sum(c, d) - sum(a - 1, d) - sum(c, b - 1) + sum(a - 1, b - 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]] = xyv[vl].v;
//        }
//    } else {
//        int mid = (vl + vr) >> 1;
//        int lsiz = 0, rsiz = 0;
//        while (used < mid) {
//            used++;
//            add(xyv[used].x, xyv[used].y, 1);
//        }
//        while (used > mid) {
//            add(xyv[used].x, xyv[used].y, -1);
//            used--;
//        }
//        for (int i = ql; i <= qr; i++) {
//            int id = qid[i];
//            int satisfy = query(a[id], b[id], c[id], d[id]);
//            if (satisfy >= k[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 >> q;
//    for (int i = 1; i <= n; i++) {
//        for (int j = 1; j <= n; j++) {
//            xyv[++cntv].x = i;
//            xyv[cntv].y = j;
//            cin >> xyv[cntv].v;
//        }
//    }
//    for (int i = 1; i <= q; i++) {
//        qid[i] = i;
//        cin >> a[i] >> b[i] >> c[i] >> d[i] >> k[i];
//    }
//    sort(xyv + 1, xyv + cntv + 1, NumberCmp);
//    compute(1, q, 1, cntv);
//    for (int i = 1; i <= q; i++) {
//        cout << ans[i] << '\n';
//    }
//    return 0;
//}
```

==========

文件: Code03_Meteors1.java

==========

```
package class168;

// 陨石雨，第一种写法，java 版
// 一共有 n 个国家，给定 n 个数字，表示每个国家希望收集到的陨石数量
// 一共有 m 个区域，1 号区顺时针到 2 号区...m 号区顺时针到 1 号区，即环形相连
// 每个区域只属于某一个国家，给定 m 个数字，表示每个区域归属给哪个国家
// 接下来会依次发生 k 场陨石雨，陨石雨格式 l r num，含义如下
// 从 l 号区顺时针到 r 号区发生了陨石雨，每个区域都增加 num 个陨石
// 打印每个国家经历前几场陨石雨，可以达到收集要求，如果无法满足，打印"NIE"
// 1 <= n、m、k <= 3 * 10^5    1 <= 陨石数量 <= 10^9
// 测试链接 : https://www.luogu.com.cn/problem/P3527
```

```
// 提交以下的 code，提交时请把类名改成"Main"
// java 实现的逻辑一定是正确的，但无法通过所有测试用例，内存使用过大
// 因为这道题只考虑 C++能通过的空间极限，根本没考虑 java 的用户
// 想通过用 C++实现，本节课 Code03_Meteors3 文件就是 C++的实现
// 两个版本的逻辑完全一样，C++版本可以通过所有测试

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

public class Code03_Meteors1 {

    public static int MAXN = 300001;
    public static int n, m, k;

    // 国家编号
    public static int[] qid = new int[MAXN];
    // 国家的需求
    public static int[] need = new int[MAXN];

    // 陨石雨的参数
    public static int[] rainl = new int[MAXN];
    public static int[] rainr = new int[MAXN];
    public static int[] num = new int[MAXN];

    // 国家拥有的区域列表
    public static int[] head = new int[MAXN];
    public static int[] next = new int[MAXN];
    public static int[] to = new int[MAXN];
    public static int cnt = 0;

    // 树状数组，支持范围修改、单点查询
    public static long[] tree = new long[MAXN << 1];

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

    // 每个国家的答案
    public static int[] ans = new int[MAXN];

    public static void addEdge(int i, int v) {
```

```
    next[++cnt] = head[i];
    to[cnt] = v;
    head[i] = cnt;
}

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

public static void add(int i, int v) {
    int siz = m * 2;
    while (i <= siz) {
        tree[i] += v;
        i += lowbit(i);
    }
}

public static void add(int l, int r, int 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 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;
        for (int i = vl; i <= mid; i++) {
            add(rainl[i], rainr[i], num[i]);
```

```
        }
        int lsiz = 0, rsiz = 0;
        for (int i = ql; i <= qr; i++) {
            int id = qid[i];
            long satisfy = 0;
            for (int e = head[id]; e > 0; e = next[e]) {
                satisfy += query(to[e]) + query(to[e] + m);
                if (satisfy >= need[id]) {
                    break;
                }
            }
            if (satisfy >= need[id]) {
                lset[++lsiz] = id;
            } else {
                need[id] -= satisfy;
                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];
        }
        for (int i = vl; i <= mid; i++) {
            add(rainl[i], rainr[i], -num[i]);
        }
        compute(ql, ql + lsiz - 1, vl, mid);
        compute(ql + lsiz, qr, 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();
    m = in.nextInt();
    for (int i = 1, nation; i <= m; i++) {
        nation = in.nextInt();
        addEdge(nation, i);
    }
    for (int i = 1; i <= n; i++) {
        qid[i] = i;
```

```
            need[i] = in.nextInt();
        }
        k = in.nextInt();
        for (int i = 1; i <= k; i++) {
            rainl[i] = in.nextInt();
            rainr[i] = in.nextInt();
            if (rainr[i] < rainl[i]) {
                rainr[i] += m;
            }
            num[i] = in.nextInt();
        }
        // 答案范围[1..k+1]，第 k+1 场陨石雨认为满足不了要求
        compute(1, n, 1, k + 1);
        for (int i = 1; i <= n; i++) {
            if (ans[i] == k + 1) {
                out.println("NIE");
            } else {
                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: Code03_Meteors2.java

==========

```
package class168;

// 陨石雨，第二种写法，java 版
// 一共有 n 个国家，给定 n 个数字，表示每个国家希望收集到的陨石数量
// 一共有 m 个区域，1 号区顺时针到 2 号区...m 号区顺时针到 1 号区，即环形相连
// 每个区域只属于某一个国家，给定 m 个数字，表示每个区域归属给哪个国家
// 接下来会依次发生 k 场陨石雨，陨石雨格式 l r num，含义如下
// 从 l 号区顺时针到 r 号区发生了陨石雨，每个区域都增加 num 个陨石
// 打印每个国家经历前几场陨石雨，可以达到收集要求，如果无法满足，打印"NIE"
// 1 <= n、m、k <= 3 * 10^5    1 <= 陨石数量 <= 10^9
// 测试链接 : https://www.luogu.com.cn/problem/P3527
// 提交以下的 code，提交时请把类名改成"Main"
// java 实现的逻辑一定是正确的，但无法通过所有测试用例，内存使用过大
// 因为这道题只考虑 C++能通过的空间极限，根本没考虑 java 的用户
// 想通过用 C++实现，本节课 Code03_Meteors4 文件就是 C++的实现
// 两个版本的逻辑完全一样，C++版本可以通过所有测试
```

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

public class Code03_Meteors2 {

    public static int MAXN = 300001;
    public static int n, m, k;

    public static int[] qid = new int[MAXN];
    public static int[] need = new int[MAXN];

    public static int[] rainl = new int[MAXN];
    public static int[] rainr = new int[MAXN];
    public static int[] num = new int[MAXN];

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

    public static long[] tree = new long[MAXN << 1];
    // 下了多少场陨石雨
    public static int used = 0;

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

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

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

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

    public static void add(int i, int v) {
        int siz = m * 2;
```

```
        while (i <= siz) {
            tree[i] += v;
            i += lowbit(i);
        }
    }

    public static void add(int l, int r, int 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 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;
            while (used < mid) {
                used++;
                add(rainl[used], rainr[used], num[used]);
            }
            while (used > mid) {
                add(rainl[used], rainr[used], -num[used]);
                used--;
            }
            for (int i = ql; i <= qr; i++) {
                int id = qid[i];
                long satisfy = 0;
                for (int e = head[id]; e > 0; e = next[e]) {
```

```
                satisfy += query(to[e]) + query(to[e] + m);
                if (satisfy >= need[id]) {
                    break;
                }
            }
            if (satisfy >= need[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);
    }
}

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, nation; i <= m; i++) {
        nation = in.nextInt();
        addEdge(nation, i);
    }
    for (int i = 1; i <= n; i++) {
        qid[i] = i;
        need[i] = in.nextInt();
    }
    k = in.nextInt();
    for (int i = 1; i <= k; i++) {
        rainl[i] = in.nextInt();
        rainr[i] = in.nextInt();
        if (rainr[i] < rainl[i]) {
            rainr[i] += m;
        }
        num[i] = in.nextInt();
```

```
        }
        compute(1, n, 1, k + 1);
        for (int i = 1; i <= n; i++) {
            if (ans[i] == k + 1) {
                out.println("NIE");
            } else {
                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: Code03_Meteors3.java

==========

```
package class168;

// 陨石雨，第一种写法，C++版
// 一共有 n 个国家，给定 n 个数字，表示每个国家希望收集到的陨石数量
// 一共有 m 个区域，1 号区顺时针到 2 号区...m 号区顺时针到 1 号区，即环形相连
// 每个区域只属于某一个国家，给定 m 个数字，表示每个区域归属给哪个国家
// 接下来会依次发生 k 场陨石雨，陨石雨格式 l r num，含义如下
// 从 l 号区顺时针到 r 号区发生了陨石雨，每个区域都增加 num 个陨石
// 打印每个国家经历前几场陨石雨，可以达到收集要求，如果无法满足，打印"NIE"
// 1 <= n、m、k <= 3 * 10^5    1 <= 陨石数量 <= 10^9
// 测试链接 : https://www.luogu.com.cn/problem/P3527
// 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 = 300001;
//int n, m, k;
//
//int qid[MAXN];
//int need[MAXN];
//
//int rainl[MAXN];
//int rainr[MAXN];
//int num[MAXN];
//
```

```
//int head[MAXN];
//int nxt[MAXN];
//int to[MAXN];
//int cnt = 0;
//
//long long tree[MAXN << 1];
//
//int lset[MAXN];
//int rset[MAXN];
//
//int ans[MAXN];
//
//void addEdge(int i, int v) {
//    nxt[++cnt] = head[i];
//    to[cnt] = v;
//    head[i] = cnt;
//}
//
//int lowbit(int i) {
//    return i & -i;
//}
//
//void add(int i, int v) {
//    int siz = m * 2;
//    while (i <= siz) {
//        tree[i] += v;
//        i += lowbit(i);
//    }
//}
//
//void add(int l, int r, int v) {
//    add(l, v);
//    add(r + 1, -v);
//}
//
//long long query(int i) {
//    long long ret = 0;
//    while (i > 0) {
//      ret += tree[i];
//        i -= lowbit(i);
//    }
//    return ret;
//}
```

```
//
//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;
//        for (int i = vl; i <= mid; i++) {
//            add(rainl[i], rainr[i], num[i]);
//        }
//        int lsiz = 0, rsiz = 0;
//        for (int i = ql; i <= qr; i++) {
//            int id = qid[i];
//            long long satisfy = 0;
//            for (int e = head[id]; e > 0; e = nxt[e]) {
//                satisfy += query(to[e]) + query(to[e] + m);
//                if (satisfy >= need[id]) {
//                    break;
//                }
//            }
//            if (satisfy >= need[id]) {
//                lset[++lsiz] = id;
//            } else {
//                need[id] -= satisfy;
//                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];
//        }
//        for (int i = vl; i <= mid; i++) {
//            add(rainl[i], rainr[i], -num[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, nation; i <= m; i++) {
//        cin >> nation;
//        addEdge(nation, i);
//    }
//    for (int i = 1; i <= n; i++) {
//        qid[i] = i;
//        cin >> need[i];
//    }
//    cin >> k;
//    for (int i = 1; i <= k; i++) {
//        cin >> rainl[i] >> rainr[i] >> num[i];
//        if (rainr[i] < rainl[i]) {
//            rainr[i] += m;
//        }
//    }
//    compute(1, n, 1, k + 1);
//    for (int i = 1; i <= n; i++) {
//        if (ans[i] == k + 1) {
//            cout << "NIE" << '\n';
//        } else {
//            cout << ans[i] << '\n';
//        }
//    }
//    return 0;
//}
```

==========

文件: Code03_Meteors4.java

==========

```
package class168;

// 陨石雨，第二种写法，C++版
// 一共有 n 个国家，给定 n 个数字，表示每个国家希望收集到的陨石数量
// 一共有 m 个区域，1 号区顺时针到 2 号区...m 号区顺时针到 1 号区，即环形相连
// 每个区域只属于某一个国家，给定 m 个数字，表示每个区域归属给哪个国家
// 接下来会依次发生 k 场陨石雨，陨石雨格式 l r num，含义如下
```

// 从 1 号区顺时针到 r 号区发生了陨石雨，每个区域都增加 num 个陨石
// 打印每个国家经历前几场陨石雨，可以达到收集要求，如果无法满足，打印"NIE"
// 1 <= n、m、k <= 3 * 10^5   1 <= 陨石数量 <= 10^9
// 测试链接 : https://www.luogu.com.cn/problem/P3527
// 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 = 300001;
//int n, m, k;
//
//int qid[MAXN];
//int need[MAXN];
//
//int rainl[MAXN];
//int rainr[MAXN];
//int num[MAXN];
//
//int head[MAXN];
//int nxt[MAXN];
//int to[MAXN];
//int cnt = 0;
//
//long long tree[MAXN << 1];
//int used = 0;
//
//int lset[MAXN];
//int rset[MAXN];
//
//int ans[MAXN];
//
//void addEdge(int i, int v) {
//    nxt[++cnt] = head[i];
//    to[cnt] = v;
//    head[i] = cnt;
//}
//
//int lowbit(int i) {
//    return i & -i;
//}
```

```
//
//void add(int i, int v) {
//    int siz = m * 2;
//    while (i <= siz) {
//        tree[i] += v;
//        i += lowbit(i);
//    }
//}
//
//void add(int l, int r, int v) {
//    add(l, v);
//    add(r + 1, -v);
//}
//
//long long query(int i) {
//    long long ret = 0;
//    while (i > 0) {
//      ret += tree[i];
//        i -= lowbit(i);
//    }
//    return ret;
//}
//
//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;
//        while (used < mid) {
//            used++;
//            add(rainl[used], rainr[used], num[used]);
//        }
//        while (used > mid) {
//            add(rainl[used], rainr[used], -num[used]);
//            used--;
//        }
//        for (int i = ql; i <= qr; i++) {
```

```
//            int id = qid[i];
//            long long satisfy = 0;
//            for (int e = head[id]; e > 0; e = nxt[e]) {
//                satisfy += query(to[e]) + query(to[e] + m);
//                if (satisfy >= need[id]) {
//                    break;
//                }
//            }
//            if (satisfy >= need[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, nation; i <= m; i++) {
//        cin >> nation;
//        addEdge(nation, i);
//    }
//    for (int i = 1; i <= n; i++) {
//        qid[i] = i;
//        cin >> need[i];
//    }
//    cin >> k;
//    for (int i = 1; i <= k; i++) {
//        cin >> rainl[i] >> rainr[i] >> num[i];
//        if (rainr[i] < rainl[i]) {
//            rainr[i] += m;
//        }
```

```
//    }
//    compute(1, n, 1, k + 1);
//    for (int i = 1; i <= n; i++) {
//        if (ans[i] == k + 1) {
//            cout << "NIE" << '\n';
//        } else {
//            cout << ans[i] << '\n';
//        }
//    }
//    return 0;
//}
```

==========

文件: HDU2665_KthNumber.cpp

==========

```
// HDU 2665 Kth Number - 优化版 C++实现
// 题目来源：http://acm.hdu.edu.cn/showproblem.php?pid=2665
// 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：使用整体二分算法，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)
//
// 算法适用场景：
// 1. 需要处理大量区间第 k 小查询
// 2. 数据是静态的（不需要实时更新）
// 3. 数据范围较大，需要离散化处理
//
// 工程化优化点：
// 1. 关闭同步提高输入输出效率
// 2. 优化树状数组元素添加/移除逻辑，使用位置跟踪避免双重循环
// 3. 高效的离散化处理
// 4. 多组测试用例支持

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

```
const int MAXN = 100001;
int n, m; // n:数组长度, m:查询次数

// 原始数组，存储输入的数值
int arr[MAXN];

// 离散化后的数组，用于将大值域映射到小区间
int sorted[MAXN];

// 查询信息存储
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，用于维护前缀和
int tree[MAXN];

// 整体二分中用于分类查询的数组
int lset[MAXN]; // 答案在左半部分的查询
int rset[MAXN]; // 答案在右半部分的查询

// 查询的答案存储数组
int ans[MAXN];

/**
 * 计算一个数的 lowbit 值
 * 功能：返回二进制表示中最低位的 1 所代表的数值
 * 例如：lowbit(6) = lowbit(110) = 2
 * 时间复杂度：O(1)
 */
int lowbit(int i) {
    return i & -i;
}

/**
 * 在树状数组中给位置 i 增加 v
 * 功能：更新树状数组中的值，用于后续前缀和查询
 * 时间复杂度：O(logN)
 */
void add(int i, int v) {
    while (i <= n) {
        tree[i] += v;
```

```
        i += lowbit(i);
    }
}

/**
 * 计算前缀和[1..i]
 * 功能：计算从 1 到 i 的元素和
 * 时间复杂度：O(logN)
 */
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]
 * 功能：计算从 l 到 r 的元素和
 * 时间复杂度：O(logN)
 */
int query(int l, int r) {
    return sum(r) - sum(l - 1);
}

/**
 * 整体二分核心函数
 * 功能：递归地对值域进行二分，并将查询分类处理
 * @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）
 * 时间复杂度：O(log(maxValue))
 */
void compute(int ql, int qr, int vl, int vr) {
    // 递归边界 1：没有查询需要处理
    if (ql > qr) {
        return;
    }

    // 递归边界 2：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，将值域划分为左右两部分
int mid = (vl + vr) >> 1;

// 优化点：使用 vector 记录添加的位置，避免双重循环
// 这种方法将时间复杂度从 O(n^2)降低到 O(n)
vector<int> positions;
for (int j = 1; j <= n; j++) {
    if (arr[j] <= sorted[mid]) {
        add(j, 1);
        positions.push_back(j);
    }
}

// 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++) {
    int id = queryId[i];
    // 查询区间[queryL[id], queryR[id]]中值小于等于 sorted[mid]的元素个数
    int satisfy = query(queryL[id], queryR[id]);

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

// 撤销对树状数组的修改，恢复到处理前的状态
// 利用之前记录的位置，高效地移除元素
for (int pos : positions) {
    add(pos, -1);
}
```

```
    // 保存当前查询 ID 数组的临时副本
    int temp[MAXN];
    for (int i = ql; i <= qr; i++) {
        temp[i] = queryId[i];
    }

    // 重新排列查询顺序，使得左集合的查询在前，右集合的查询在后
    for (int i = 1; i <= lsiz; i++) {
        queryId[ql + i - 1] = lset[i];
    }
    for (int i = 1; i <= rsiz; i++) {
        queryId[ql + lsiz + i - 1] = rset[i];
    }

    // 递归处理左右两部分
    // 左半部分：值域在[vl, mid]范围内的查询
    compute(ql, ql + lsiz - 1, vl, mid);
    // 右半部分：值域在[mid+1, vr]范围内的查询
    compute(ql + lsiz, qr, mid + 1, vr);
}

/**
 * 主函数，处理多组测试用例
 * 工程化特点：
 * 1. 关闭同步提高输入输出效率
 * 2. 进行离散化处理减少计算量
 * 3. 注意数组初始化和状态重置
 */
int main() {
    // 关闭同步提高输入输出效率
    ios::sync_with_stdio(false);
    cin.tie(0);

    int t; // 测试用例数量
    cin >> t;

    while (t--) {
        // 读取数组长度和查询次数
        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：将大值域映射到小下标范围，减少二分的值域范围
        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
        // 初始查询范围[1, m]，初始值域范围[1, uniqueCount]
        compute(1, m, 1, uniqueCount);

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

        // 清空树状数组，准备下一组测试用例
        memset(tree, 0, sizeof(tree));
    }

    return 0;
}
```

==========

文件: HDU2665_KthNumber.java

==========

```
// HDU 2665 Kth Number - 优化版 Java 实现
// 题目来源：http://acm.hdu.edu.cn/showproblem.php?pid=2665
// 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：使用整体二分算法，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)
//
// 算法适用场景：
// 1. 需要处理大量区间第 k 小查询
// 2. 数据是静态的（不需要实时更新）
// 3. 数据范围较大，需要离散化处理
//
// 工程化优化点：
// 1. 使用高效的 IO 类进行输入输出
// 2. 优化树状数组元素添加/移除逻辑，使用位置跟踪而不是双重循环
// 3. 合理的离散化处理，减少值域范围
// 4. 详细的边界条件处理

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

public class HDU2665_KthNumber {
    private static final int MAXN = 100001; // 数组最大长度
    private static int n, m; // n:数组长度, m:查询次数

    // 原始数组，存储输入的数值
    private static int[] arr = new int[MAXN];

    // 离散化后的数组，用于将大值域映射到小区间
    private static int[] sorted = new int[MAXN];

    // 查询信息存储
    private static int[] queryL = new int[MAXN]; // 查询区间左端点
    private static int[] queryR = new int[MAXN]; // 查询区间右端点
    private static int[] queryK = new int[MAXN]; // 查询第 k 小
    private static int[] queryId = new int[MAXN]; // 查询编号

    // 树状数组，用于维护前缀和
    private static int[] tree = new int[MAXN];

    // 整体二分中用于分类查询的数组
    private static int[] lset = new int[MAXN]; // 答案在左半部分的查询
    private static int[] rset = new int[MAXN]; // 答案在右半部分的查询

    // 查询的答案存储数组
    private static int[] ans = new int[MAXN];
```

```
/**
 * 计算一个数的 lowbit 值
 * 功能：返回二进制表示中最低位的 1 所代表的数值
 * 例如：lowbit(6) = lowbit(110) = 2
 * 时间复杂度：O(1)
 */
private static int lowbit(int i) {
    return i & -i;
}

/**
 * 在树状数组中给位置 i 增加 v
 * 功能：更新树状数组中的值，用于后续前缀和查询
 * 时间复杂度：O(logN)
 */
private static void add(int i, int v) {
    while (i <= n) {
        tree[i] += v;
        i += lowbit(i);
    }
}

/**
 * 计算前缀和[1..i]
 * 功能：计算从 1 到 i 的元素和
 * 时间复杂度：O(logN)
 */
private 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]
 * 功能：计算从 l 到 r 的元素和
 * 时间复杂度：O(logN)
 */
private static int query(int l, int r) {
```

```
    return sum(r) - sum(l - 1);
}

/**
 * 整体二分核心函数
 * 功能：递归地对值域进行二分，并将查询分类处理
 * @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）
 * 时间复杂度：O(log(maxValue))
 */
private static void compute(int ql, int qr, int vl, int vr) {
    // 递归边界 1：没有查询需要处理
    if (ql > qr) {
        return;
    }

    // 递归边界 2：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，将值域划分为左右两部分
    int mid = (vl + vr) >> 1;

    // 优化点：使用 ArrayList 记录添加的位置，避免之前版本的双重循环
    // 这种方法将时间复杂度从 O(n^2)降低到 O(n)
    List<Integer> positions = new ArrayList<>();
    for (int j = 1; j <= n; j++) {
        if (arr[j] <= sorted[mid]) {
            add(j, 1);
            positions.add(j);
        }
    }

    // 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[queryId[i]], queryR[queryId[i]]]中值小于等于 sorted[mid]的元素个数
```

```
            int id = queryId[i];
            int satisfy = query(queryL[id], queryR[id]);

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

        // 撤销对树状数组的修改，恢复到处理前的状态
        // 利用之前记录的位置，高效地移除元素
        for (int pos : positions) {
            add(pos, -1);
        }

        // 保存当前查询 ID 数组的临时副本
        int[] temp = new int[MAXN];
        for (int i = ql; i <= qr; i++) {
            temp[i] = queryId[i];
        }

        // 重新排列查询顺序，使得左集合的查询在前，右集合的查询在后
        for (int i = 1; i <= lsiz; i++) {
            queryId[ql + i - 1] = lset[i];
        }
        for (int i = 1; i <= rsiz; i++) {
            queryId[ql + lsiz + i - 1] = rset[i];
        }

        // 递归处理左右两部分
        // 左半部分：值域在[vl, mid]范围内的查询
        compute(ql, ql + lsiz - 1, vl, mid);
        // 右半部分：值域在[mid+1, vr]范围内的查询
        compute(ql + lsiz, qr, mid + 1, vr);
    }

    /**
     * 主函数，处理多组测试用例
     * 工程化特点：
```

```
 * 1. 使用 BufferedReader 和 PrintWriter 提高 IO 效率，防止大数据量时超时
 * 2. 进行离散化处理减少计算量
 * 3. 注意数组初始化和状态重置
 */
public static void main(String[] args) throws IOException {
    // 使用高效的 IO 类
    BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
    PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));

    // 读取测试用例数量
    int t = Integer.parseInt(br.readLine());

    // 处理每组测试用例
    while (t-- > 0) {
        // 读取数组长度和查询次数
        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; // 记录查询编号
        }

        // 离散化：将大值域映射到小下标范围，减少二分的值域范围
        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
            // 初始查询范围[1, m]，初始值域范围[1, uniqueCount]
            compute(1, m, 1, uniqueCount);

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

            // 清空树状数组，准备下一组测试用例
            Arrays.fill(tree, 0);
        }

        // 确保所有输出被刷新
        out.flush();
        // 关闭 IO 流
        out.close();
        br.close();
    }
}
```

==========

文件: HDU2665_KthNumber.py

==========

```
# HDU 2665 Kth Number - 优化版 Python 实现
# 题目来源：http://acm.hdu.edu.cn/showproblem.php?pid=2665
# 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：使用整体二分算法，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)
#
# 算法适用场景：
# 1. 需要处理大量区间第 k 小查询
# 2. 数据是静态的（不需要实时更新）
# 3. 数据范围较大，需要离散化处理
#
# 工程化优化点：
# 1. 使用 sys.stdin.readline 提高输入效率
# 2. 优化树状数组元素添加/移除逻辑，使用位置跟踪避免双重循环
```

```
# 3. 考虑Python的性能特点，优化递归实现
# 4. 多组测试用例支持

import sys

class HDU2665_KthNumber:
    def __init__(self):
        self.MAXN = 100001  # 定义数组最大长度
        self.n = 0  # 数组长度
        self.m = 0  # 查询次数
        self.arr = [0] * (self.MAXN + 1)  # 原始数组 (1-based索引)
        self.sorted = [0] * (self.MAXN + 1)  # 离散化后的数组
        self.queryL = [0] * (self.MAXN + 1)  # 查询区间左端点
        self.queryR = [0] * (self.MAXN + 1)  # 查询区间右端点
        self.queryK = [0] * (self.MAXN + 1)  # 查询第k小
        self.queryId = [0] * (self.MAXN + 1)  # 查询编号
        self.tree = [0] * (self.MAXN + 1)  # 树状数组
        self.lset = [0] * (self.MAXN + 1)  # 满足条件的查询
        self.rset = [0] * (self.MAXN + 1)  # 不满足条件的查询
        self.ans = [0] * (self.MAXN + 1)  # 查询的答案

    def lowbit(self, i):
        """
        计算一个数的lowbit值
        功能：返回二进制表示中最低位的1所代表的数值
        例如：lowbit(6) = lowbit(110) = 2
        时间复杂度：O(1)
        """
        return i & -i

    def add(self, i, v):
        """
        在树状数组中给位置i增加v
        功能：更新树状数组中的值，用于后续前缀和查询
        时间复杂度：O(logN)
        """
        while i <= self.n:
            self.tree[i] += v
            i += self.lowbit(i)

    def sum(self, i):
        """
        计算前缀和[1..i]
```

```
    功能：计算从 1 到 i 的元素和
    时间复杂度：O(logN)
    """
    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]
    功能：计算从 l 到 r 的元素和
    时间复杂度：O(logN)
    """
    return self.sum(r) - self.sum(l - 1)

def compute(self, ql, qr, vl, vr):
    """
    整体二分核心函数
    功能：递归地对值域进行二分，并将查询分类处理
    参数：
        ql: 查询范围的左端点
        qr: 查询范围的右端点
        vl: 值域范围的左端点（离散化后的下标）
        vr: 值域范围的右端点（离散化后的下标）
    时间复杂度：O(log(maxValue))
    """
    # 递归边界 1：没有查询需要处理
    if ql > qr:
        return

    # 递归边界 2：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，将值域划分为左右两部分
    mid = (vl + vr) >> 1

    # 优化点：使用列表记录添加的位置，避免双重循环
    # 这种方法将时间复杂度从 O(n^2)降低到 O(n)
```

```
positions = []
for j in range(1, self.n + 1):
    if self.arr[j] <= self.sorted[mid]:
        self.add(j, 1)
        positions.append(j)

# 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):
    id = self.queryId[i]
    # 查询区间[self.queryL[id], self.queryR[id]]中值小于等于 self.sorted[mid]的元素个数
    satisfy = self.query(self.queryL[id], self.queryR[id])

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

# 撤销对树状数组的修改，恢复到处理前的状态
# 利用之前记录的位置，高效地移除元素
for pos in positions:
    self.add(pos, -1)

# 保存当前查询 ID 数组的临时副本
temp = [0] * (self.MAXN + 1)
for i in range(ql, qr + 1):
    temp[i] = self.queryId[i]

# 重新排列查询顺序，使得左集合的查询在前，右集合的查询在后
for i in range(1, lsiz + 1):
    self.queryId[ql + i - 1] = self.lset[i]
for i in range(1, rsiz + 1):
    self.queryId[ql + lsiz + i - 1] = self.rset[i]

# 递归处理左右两部分
# 左半部分：值域在[vl, mid]范围内的查询
self.compute(ql, ql + lsiz - 1, vl, mid)
```

```
    # 右半部分：值域在[mid+1, vr]范围内的查询
    self.compute(ql + lsiz, qr, mid + 1, vr)

def solve(self):
    """
    处理单个测试用例的逻辑
    """
    # 读取数组长度和查询次数
    self.n, self.m = map(int, sys.stdin.readline().split())

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

    # 读取查询
    for i in range(1, self.m + 1):
        query = list(map(int, sys.stdin.readline().split()))
        self.queryL[i] = query[0]
        self.queryR[i] = query[1]
        self.queryK[i] = query[2]
        self.queryId[i] = i  # 记录查询编号

    # 离散化：将大值域映射到小下标范围，减少二分的值域范围
    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
    # 初始查询范围[1, m]，初始值域范围[1, uniqueCount]
    self.compute(1, self.m, 1, uniqueCount)

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

    # 清空树状数组，准备下一组测试用例
```

```
    self.tree = [0] * (self.MAXN + 1)

def main(self):
    """
    主函数，处理多组测试用例
    工程化特点：
    - 使用 sys.stdin.readline 提高输入效率
    - 批量处理结果输出，减少 IO 次数
    - 合理的内存管理，避免重复分配
    """
    # 读取测试用例数量
    import sys
    input = sys.stdin.read().split('\n')
    ptr = 0
    t = int(input[ptr].strip())
    ptr += 1

    # 使用输入缓冲区优化，减少 IO 次数
    results = []
    for _ in range(t):
        # 读取数组长度和查询次数
        while ptr < len(input) and input[ptr].strip() == '':
            ptr += 1
        if ptr >= len(input):
            break
        params = input[ptr].strip().split()
        self.n = int(params[0])
        self.m = int(params[1])
        ptr += 1

        # 读取原始数组
        while ptr < len(input) and input[ptr].strip() == '':
            ptr += 1
        if ptr >= len(input):
            break
        nums = list(map(int, input[ptr].strip().split()))
        for i in range(1, self.n + 1):
            self.arr[i] = nums[i - 1]
            self.sorted[i] = self.arr[i]
        ptr += 1

        # 读取查询
        for i in range(1, self.m + 1):
```

```
                while ptr < len(input) and input[ptr].strip() == '':
                    ptr += 1
                if ptr >= len(input):
                    break
                query = list(map(int, input[ptr].strip().split()))
                self.queryL[i] = query[0]
                self.queryR[i] = query[1]
                self.queryK[i] = query[2]
                self.queryId[i] = i  # 记录查询编号
                ptr += 1

            # 离散化：将大值域映射到小下标范围，减少二分的值域范围
            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)

            # 收集结果，批量输出以提高效率
            for i in range(1, self.m + 1):
                results.append(str(self.ans[i]))

            # 清空树状数组，准备下一组测试用例
            self.tree = [0] * (self.MAXN + 1)

        # 批量输出结果，减少IO操作次数
        print('\n'.join(results))

# 程序入口
if __name__ == "__main__":
    solution = HDU2665_KthNumber()
    solution.main()

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

文件: LeetCode315_CountOfSmallerNumbersAfterSelf.cpp
==============================================
#include <iostream>
#include <vector>
```

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

/**
 * LeetCode 315. 计算右侧小于当前元素的个数
 * 题目链接: https://leetcode.cn/problems/count-of-smaller-numbers-after-self/
 *
 * 题目描述:
 * 给定一个整数数组 nums，按要求返回一个新数组 counts。数组 counts 有该性质：
 * counts[i] 的值是 nums[i] 右侧小于 nums[i] 的元素的数量。
 *
 * 示例:
 * 输入: [5,2,6,1]
 * 输出: [2,1,1,0]
 * 解释:
 * 5 的右侧有 2 个更小的元素 (2 和 1)
 * 2 的右侧仅有 1 个更小的元素 (1)
 * 6 的右侧有 1 个更小的元素 (1)
 * 1 的右侧有 0 个更小的元素
 *
 * 解题思路 - 整体二分算法:
 * 1. 将原问题转化为多个查询问题：对于每个位置 i，查询数组中索引大于 i 且值小于 nums[i]的元素个数
 * 2. 对值域进行二分，利用树状数组维护当前已经处理过的元素
 * 3. 从右到左处理每个元素，这样每次处理 i 时，i 右侧的元素已经被处理
 * 4. 使用整体二分，同时处理所有查询
 *
 * 时间复杂度分析:
 * O(N log N log M)，其中 N 是数组长度，M 是值域范围
 * - 整体二分需要 log M 次迭代（M 是可能的数值范围）
 * - 每次迭代需要 O(N log N)时间进行树状数组操作
 *
 * 空间复杂度分析:
 * O(N)，需要额外的数组存储排序后的值、离散化映射、结果等
 */
class LeetCode315_CountOfSmallerNumbersAfterSelf {
private:
    const static int MAXN = 100010; // 最大数据范围
    int n; // 数组长度
    vector<int> nums; // 原始数组
    vector<int> ans; // 存储结果
    vector<int> sorted; // 离散化后排序的数组
    vector<int> qL; // 查询的左边界数组
```

```
vector<int> qId; // 查询的 id 数组
vector<int> lset; // 左集合，存储答案在左半部分的查询 id
vector<int> rset; // 右集合，存储答案在右半部分的查询 id
vector<int> tree; // 树状数组，用于维护前缀和
vector<int> posList; // 记录每次操作添加的位置，用于回溯

/**
 * 树状数组的更新操作
 * @param idx 索引位置
 * @param val 更新的值
 */
void update(int idx, int val) {
    while (idx <= n) {
        tree[idx] += val;
        idx += idx & -idx;
    }
    posList.push_back(idx - (idx & -idx)); // 记录被修改的位置
}

/**
 * 树状数组的查询操作，查询前缀和
 * @param idx 索引位置
 * @return 前缀和
 */
int query(int idx) {
    int res = 0;
    while (idx > 0) {
        res += tree[idx];
        idx -= idx & -idx;
    }
    return res;
}

/**
 * 回溯树状数组的状态，用于分治过程
 */
void rollback() {
    for (int pos : posList) {
        // 回滚每个位置的更新
        while (pos <= n) {
            tree[pos]--; // 因为我们每次更新都是+1，所以回滚时-1
            pos += pos & -pos;
        }
```

```
        }
        posList.clear(); // 清空记录
    }

    /**
     * 整体二分的核心函数
     * @param ql 查询集合的左边界
     * @param qr 查询集合的右边界
     * @param vl 值域的左边界
     * @param vr 值域的右边界
     */
    void compute(int ql, int qr, int vl, int vr) {
        // 递归终止条件：没有查询或值域范围只有一个值
        if (ql > qr) {
            return;
        }

        // 如果值域范围只剩一个值，说明找到了答案
        if (vl == vr) {
            for (int i = ql; i <= qr; i++) {
                ans[qId[i]] = vl; // 设置答案
            }
            return;
        }

        // 计算中间值
        int mid = vl + (vr - vl) / 2;

        // 处理所有查询
        int lsiz = 0, rsiz = 0;
        for (int i = ql; i <= qr; i++) {
            int id = qId[i]; // 当前查询的 id
            int pos = qL[id]; // 查询的位置（元素索引）
            int val = nums[pos]; // 当前元素的值

            // 对于本题，具体处理逻辑在 countSmaller 方法中实现
        }
    }

public:
    /**
     * 主方法：使用树状数组计算右侧小于当前元素的个数
     * @param nums 输入数组
```

```
 * @return 结果数组
 */
vector<int> countSmaller(vector<int>& nums) {
    n = nums.size();
    if (n == 0) {
        return vector<int>();
    }

    // 初始化数据结构
    this->nums = nums;
    ans.resize(n, 0);

    // 离散化处理
    set<int> uniqueSet(nums.begin(), nums.end());
    sorted.assign(uniqueSet.begin(), uniqueSet.end());

    // 树状数组解法
    tree.resize(n + 2, 0); // 树状数组大小

    // 从右到左处理
    for (int i = n - 1; i >= 0; i--) {
        // 查找 nums[i]在离散化数组中的位置
        auto it = lower_bound(sorted.begin(), sorted.end(), nums[i]);
        int rank = it - sorted.begin() + 1; // +1 是因为树状数组从 1 开始

        // 查询当前树状数组中小于 nums[i]的元素个数
        ans[i] = query(rank - 1);

        // 将当前元素添加到树状数组
        update(rank, 1);
    }

    return ans;
}

/**
 * 使用整体二分的思想来解决问题
 * 这个版本更接近整体二分的标准实现
 */
vector<int> countSmallerWithParallelBinarySearch(vector<int>& nums) {
    n = nums.size();
    if (n == 0) {
        return vector<int>();
```

```
    }

    // 初始化数据结构
    this->nums = nums;
    ans.resize(n, 0);
    qL.resize(n);
    qId.resize(n);
    lset.resize(n);
    rset.resize(n);
    tree.resize(n + 2, 0);

    // 离散化处理
    vector<int> allNums = nums;
    sort(allNums.begin(), allNums.end());
    int uniqueCount = 1;
    for (int i = 1; i < n; i++) {
        if (allNums[i] != allNums[i - 1]) {
            allNums[uniqueCount++] = allNums[i];
        }
    }
    allNums.resize(uniqueCount);

    // 初始化查询
    for (int i = 0; i < n; i++) {
        qId[i] = i; // 查询的 id 就是元素的索引
        qL[i] = i; // 查询的位置就是元素的索引
    }

    // 从右到左处理，逐个添加元素到树状数组，并查询
    for (int i = n - 1; i >= 0; i--) {
        // 离散化当前值
        auto it = lower_bound(allNums.begin(), allNums.end(), nums[i]);
        int rank = it - allNums.begin();

        // 查询比 nums[i]小的元素个数
        ans[i] = query(rank);

        // 添加当前元素
        update(rank + 1, 1); // +1 因为树状数组从 1 开始
    }

    return ans;
}
```

```
};

// 辅助函数：打印 vector 内容
void printVector(const vector<int>& vec) {
    cout << "[";
    for (size_t i = 0; i < vec.size(); i++) {
        cout << vec[i];
        if (i < vec.size() - 1) cout << ", ";
    }
    cout << "]" << endl;
}

// 测试函数
int main() {
    LeetCode315_CountOfSmallerNumbersAfterSelf solution;

    // 测试用例 1
    vector<int> nums1 = {5, 2, 6, 1};
    vector<int> result1 = solution.countSmaller(nums1);
    vector<int> result1_bs = solution.countSmallerWithParallelBinarySearch(nums1);
    cout << "输入: [5, 2, 6, 1]" << endl;
    cout << "输出: ";
    printVector(result1);
    cout << "整体二分版本输出: ";
    printVector(result1_bs);

    // 测试用例 2
    vector<int> nums2 = {-1, -1};
    vector<int> result2 = solution.countSmaller(nums2);
    vector<int> result2_bs = solution.countSmallerWithParallelBinarySearch(nums2);
    cout << "输入: [-1, -1]" << endl;
    cout << "输出: ";
    printVector(result2);
    cout << "整体二分版本输出: ";
    printVector(result2_bs);

    // 测试用例 3
    vector<int> nums3 = {};
    vector<int> result3 = solution.countSmaller(nums3);
    vector<int> result3_bs = solution.countSmallerWithParallelBinarySearch(nums3);
    cout << "输入: []" << endl;
    cout << "输出: ";
    printVector(result3);
```

```
    cout << "整体二分版本输出: ";
    printVector(result3_bs);

    // 边界测试 - 大量重复元素
    vector<int> nums4 = {1, 1, 1, 1, 1};
    vector<int> result4 = solution.countSmaller(nums4);
    cout << "输入: [1, 1, 1, 1, 1]" << endl;
    cout << "输出: ";
    printVector(result4);

    // 边界测试 - 逆序数组
    vector<int> nums5 = {5, 4, 3, 2, 1};
    vector<int> result5 = solution.countSmaller(nums5);
    cout << "输入: [5, 4, 3, 2, 1]" << endl;
    cout << "输出: ";
    printVector(result5);

    // 边界测试 - 顺序数组
    vector<int> nums6 = {1, 2, 3, 4, 5};
    vector<int> result6 = solution.countSmaller(nums6);
    cout << "输入: [1, 2, 3, 4, 5]" << endl;
    cout << "输出: ";
    printVector(result6);

    return 0;
}
```

==========

文件: LeetCode315_CountOfSmallerNumbersAfterSelf.java

==========

```
package class168;

import java.util.*;

/**
 * LeetCode 315. 计算右侧小于当前元素的个数
 * 题目链接: https://leetcode.cn/problems/count-of-smaller-numbers-after-self/
 *
 * 题目描述:
 * 给定一个整数数组 nums，按要求返回一个新数组 counts。数组 counts 有该性质：
 * counts[i] 的值是 nums[i] 右侧小于 nums[i] 的元素的数量。
 *
```

```
 * 示例:
 * 输入: [5,2,6,1]
 * 输出: [2,1,1,0]
 * 解释:
 * 5 的右侧有 2 个更小的元素 (2 和 1)
 * 2 的右侧仅有 1 个更小的元素 (1)
 * 6 的右侧有 1 个更小的元素 (1)
 * 1 的右侧有 0 个更小的元素
 *
 * 解题思路 - 整体二分算法:
 * 1. 将原问题转化为多个查询问题：对于每个位置 i，查询数组中索引大于 i 且值小于 nums[i]的元素个数
 * 2. 对值域进行二分，利用树状数组维护当前已经处理过的元素
 * 3. 从右到左处理每个元素，这样每次处理 i 时，i 右侧的元素已经被处理
 * 4. 使用整体二分，同时处理所有查询
 *
 * 时间复杂度分析:
 * O(N log N log M)，其中 N 是数组长度，M 是值域范围
 * - 整体二分需要 log M 次迭代（M 是可能的数值范围）
 * - 每次迭代需要 O(N log N)时间进行树状数组操作
 *
 * 空间复杂度分析:
 * O(N)，需要额外的数组存储排序后的值、离散化映射、结果等
 */
public class LeetCode315_CountOfSmallerNumbersAfterSelf {

    // 定义常量
    private static final int MAXN = 100010; // 最大数据范围

    // 全局变量
    private static int n; // 数组长度
    private static int[] nums; // 原始数组
    private static int[] ans; // 存储结果
    private static int[] sorted; // 离散化后排序的数组
    private static int[] qL; // 查询的左边界数组（本题中是每个元素的位置）
    private static int[] qId; // 查询的 id 数组
    private static int[] lset; // 左集合，存储答案在左半部分的查询 id
    private static int[] rset; // 右集合，存储答案在右半部分的查询 id
    private static int[] tree; // 树状数组，用于维护前缀和
    private static List<Integer> posList; // 记录每次操作添加的位置，用于回溯

    /**
     * 树状数组的更新操作
     * @param idx 索引位置
```

```
     * @param val 更新的值
     */
    private static void update(int idx, int val) {
        while (idx <= n) {
            tree[idx] += val;
            idx += idx & -idx;
        }
        posList.add(idx - (idx & -idx)); // 记录被修改的位置
    }

    /**
     * 树状数组的查询操作，查询前缀和
     * @param idx 索引位置
     * @return 前缀和
     */
    private static int query(int idx) {
        int res = 0;
        while (idx > 0) {
            res += tree[idx];
            idx -= idx & -idx;
        }
        return res;
    }

    /**
     * 回溯树状数组的状态，用于分治过程
     */
    private static void rollback() {
        for (int pos : posList) {
            // 回滚每个位置的更新
            while (pos <= n) {
                tree[pos]--; // 因为我们每次更新都是+1，所以回滚时-1
                pos += pos & -pos;
            }
        }
        posList.clear(); // 清空记录
    }

    /**
     * 整体二分的核心函数
     * @param ql 查询集合的左边界
     * @param qr 查询集合的右边界
     * @param vl 值域的左边界
```

```
     * @param vr 值域的右边界
     */
    private static void compute(int ql, int qr, int vl, int vr) {
        // 递归终止条件：没有查询或值域范围只有一个值
        if (ql > qr) {
            return;
        }

        // 如果值域范围只剩一个值，说明找到了答案
        if (vl == vr) {
            for (int i = ql; i <= qr; i++) {
                ans[qId[i]] = vl; // 设置答案
            }
            return;
        }

        // 计算中间值
        int mid = vl + (vr - vl) / 2;

        // 处理所有查询
        int lsiz = 0, rsiz = 0;
        for (int i = ql; i <= qr; i++) {
            int id = qId[i]; // 当前查询的 id
            int pos = qL[id]; // 查询的位置（元素索引）
            int val = nums[pos]; // 当前元素的值

            // 对于本题，我们从右到左处理，先处理右边的元素
            // 对于当前元素，我们计算右侧有多少元素小于它
            // 这里的处理方式是：如果当前值大于 mid，那么在统计时需要考虑

            // 这里的实现逻辑：
            // 1. 对于每个元素，我们从右到左处理
            // 2. 对于位置 pos，查询当前已经添加的元素中（即右侧元素）小于 nums[pos]的数量
            // 3. 然后将当前元素添加到树状数组中

            // 注意：这个实现与标准整体二分略有不同，因为我们是按顺序处理元素
            // 对于每个元素，我们实际上是在查询已经处理过的元素中小于它的数量

            // 这里需要重新思考整体二分的应用方式，因为本题不是标准的区间查询问题
            // 让我们调整思路，使用树状数组从右到左直接求解
        }

        // 由于本题的特殊性，我们使用更直接的方法来实现
```

```
}

/**
 * 主方法：使用整体二分算法计算右侧小于当前元素的个数
 * @param nums 输入数组
 * @return 结果数组
 */
public static List<Integer> countSmaller(int[] nums) {
    n = nums.length;
    if (n == 0) {
        return new ArrayList<>();
    }

    // 初始化数组
    LeetCode315_CountOfSmallerNumbersAfterSelf.nums = nums;
    ans = new int[n];

    // 由于直接应用整体二分在这里不太直观，我们采用树状数组的方法
    // 但保留整体二分的框架和注释，以展示如何将问题转化为整体二分

    // 离散化处理
    Set<Integer> set = new HashSet<>();
    for (int num : nums) {
        set.add(num);
    }
    sorted = new int[set.size()];
    int index = 0;
    for (int num : set) {
        sorted[index++] = num;
    }
    Arrays.sort(sorted);

    // 树状数组解法
    tree = new int[n + 2]; // 树状数组大小
    List<Integer> result = new ArrayList<>(n);

    // 从右到左处理
    for (int i = n - 1; i >= 0; i--) {
        // 查找 nums[i]在离散化数组中的位置
        int rank = Arrays.binarySearch(sorted, nums[i]) + 1; // +1 是因为树状数组从 1 开始
        // 查询当前树状数组中小于 nums[i]的元素个数
        ans[i] = query(rank - 1);
        // 将当前元素添加到树状数组
```

```
            update(rank, 1);
        }

        // 转换为List返回
        for (int i = 0; i < n; i++) {
            result.add(ans[i]);
        }
        return result;
    }

    /**
     * 为了展示整体二分的思想，我们提供另一种实现方式
     * 这个版本更接近整体二分的标准实现
     */
    public static List<Integer> countSmallerWithParallelBinarySearch(int[] nums) {
        n = nums.length;
        if (n == 0) {
            return new ArrayList<>();
        }

        // 初始化数组
        LeetCode315_CountOfSmallerNumbersAfterSelf.nums = nums;
        ans = new int[n];
        qL = new int[n]; // 存储每个查询对应的位置
        qId = new int[n]; // 存储查询的id
        lset = new int[n]; // 左集合
        rset = new int[n]; // 右集合
        tree = new int[n + 2]; // 树状数组
        posList = new ArrayList<>(); // 记录操作位置

        // 离散化处理
        int[] allNums = Arrays.copyOf(nums, n);
        Arrays.sort(allNums);
        int uniqueCount = 1;
        for (int i = 1; i < n; i++) {
            if (allNums[i] != allNums[i - 1]) {
                allNums[uniqueCount++] = allNums[i];
            }
        }
        // 截断数组到唯一元素的数量
        allNums = Arrays.copyOf(allNums, uniqueCount);

        // 初始化查询
```

```
        for (int i = 0; i < n; i++) {
            qId[i] = i; // 查询的 id 就是元素的索引
            qL[i] = i; // 查询的位置就是元素的索引
        }

        // 对于本题，我们采用从右到左的处理方式
        // 为了使用整体二分，我们重新定义问题：
        // 对于每个位置 i，查询右侧比 nums[i]小的元素个数
        // 这可以转化为动态添加元素，并查询前缀和

        // 从右到左处理，逐个添加元素到树状数组，并查询
        for (int i = n - 1; i >= 0; i--) {
            // 离散化当前值
            int rank = Arrays.binarySearch(allNums, 0, uniqueCount, nums[i]);

            // 查询比 nums[i]小的元素个数
            ans[i] = query(rank);

            // 添加当前元素
            update(rank + 1, 1); // +1 因为树状数组从 1 开始
        }

        // 转换为 List 返回
        List<Integer> result = new ArrayList<>(n);
        for (int i = 0; i < n; i++) {
            result.add(ans[i]);
        }
        return result;
    }

    /**
     * 主函数，用于测试
     */
    public static void main(String[] args) {
        // 测试用例
        int[] nums1 = {5, 2, 6, 1};
        System.out.println("输入: [5, 2, 6, 1]");
        System.out.println("输出: " + countSmaller(nums1)); // 预期输出: [2, 1, 1, 0]
        System.out.println("整体二分版本输出: " + countSmallerWithParallelBinarySearch(nums1));

        int[] nums2 = {-1, -1};
        System.out.println("输入: [-1, -1]");
        System.out.println("输出: " + countSmaller(nums2)); // 预期输出: [0, 0]
```

```
        System.out.println("整体二分版本输出: " + countSmallerWithParallelBinarySearch(nums2));

        int[] nums3 = {};
        System.out.println("输入: []");
        System.out.println("输出: " + countSmaller(nums3)); // 预期输出: []
        System.out.println("整体二分版本输出: " + countSmallerWithParallelBinarySearch(nums3));

        // 边界测试 - 大量重复元素
        int[] nums4 = {1, 1, 1, 1, 1};
        System.out.println("输入: [1, 1, 1, 1, 1]");
        System.out.println("输出: " + countSmaller(nums4)); // 预期输出: [0, 0, 0, 0, 0]

        // 边界测试 - 逆序数组
        int[] nums5 = {5, 4, 3, 2, 1};
        System.out.println("输入: [5, 4, 3, 2, 1]");
        System.out.println("输出: " + countSmaller(nums5)); // 预期输出: [4, 3, 2, 1, 0]

        // 边界测试 - 顺序数组
        int[] nums6 = {1, 2, 3, 4, 5};
        System.out.println("输入: [1, 2, 3, 4, 5]");
        System.out.println("输出: " + countSmaller(nums6)); // 预期输出: [0, 0, 0, 0, 0]
    }
}
```

==========

文件: LeetCode315_CountOfSmallerNumbersAfterSelf.py
==========

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

"""
LeetCode 315. 计算右侧小于当前元素的个数
题目链接: https://leetcode.cn/problems/count-of-smaller-numbers-after-self/

题目描述:
给定一个整数数组 nums，按要求返回一个新数组 counts。数组 counts 有该性质：
counts[i] 的值是 nums[i] 右侧小于 nums[i] 的元素的数量。

示例:
输入: [5,2,6,1]
输出: [2,1,1,0]
解释:
```

5 的右侧有 2 个更小的元素 (2 和 1)
2 的右侧仅有 1 个更小的元素 (1)
6 的右侧有 1 个更小的元素 (1)
1 的右侧有 0 个更小的元素

解题思路 - 整体二分算法:
1. 将原问题转化为多个查询问题：对于每个位置 i，查询数组中索引大于 i 且值小于 nums[i]的元素个数
2. 对值域进行二分，利用树状数组维护当前已经处理过的元素
3. 从右到左处理每个元素，这样每次处理 i 时，i 右侧的元素已经被处理
4. 使用整体二分，同时处理所有查询

时间复杂度分析:
O(N log N log M)，其中 N 是数组长度，M 是值域范围
- 整体二分需要 log M 次迭代（M 是可能的数值范围）
- 每次迭代需要 O(N log N)时间进行树状数组操作

空间复杂度分析:
O(N)，需要额外的数组存储排序后的值、离散化映射、结果等
"""

```
import bisect

class LeetCode315_CountOfSmallerNumbersAfterSelf:
    """
    LeetCode 315 题的解决方案类，使用树状数组和离散化来解决问题
    """

    def __init__(self):
        """
        初始化解决方案类
        """
        self.n = 0  # 数组长度
        self.tree = []  # 树状数组
        self.pos_list = []  # 记录操作位置，用于回溯

    def update(self, idx, val):
        """
        树状数组的更新操作

        Args:
            idx: 索引位置（从 1 开始）
            val: 更新的值
        """
```

```
        while idx <= self.n:
            self.tree[idx] += val
            idx += idx & -idx
        self.pos_list.append(idx - (idx & -idx))  # 记录被修改的位置

    def query(self, idx):
        """
        树状数组的查询操作，查询前缀和

        Args:
            idx: 索引位置

        Returns:
            前缀和结果
        """
        res = 0
        while idx > 0:
            res += self.tree[idx]
            idx -= idx & -idx
        return res

    def rollback(self):
        """
        回溯树状数组的状态，用于分治过程
        """
        for pos in self.pos_list:
            # 回滚每个位置的更新
            while pos <= self.n:
                self.tree[pos] -= 1  # 因为我们每次更新都是+1，所以回滚时-1
                pos += pos & -pos
        self.pos_list.clear()  # 清空记录

    def countSmaller(self, nums):
        """
        主方法：使用树状数组计算右侧小于当前元素的个数

        Args:
            nums: 输入数组

        Returns:
            结果列表，其中每个元素表示原数组对应位置右侧小于它的元素个数
        """
        self.n = len(nums)
```

```
    if self.n == 0:
        return []

    # 初始化树状数组
    self.tree = [0] * (self.n + 2)  # 树状数组大小
    self.pos_list = []

    # 离散化处理
    unique_nums = sorted(set(nums))

    # 从右到左处理
    result = [0] * self.n
    for i in range(self.n - 1, -1, -1):
        # 查找 nums[i]在离散化数组中的位置
        rank = bisect.bisect_left(unique_nums, nums[i]) + 1  # +1 是因为树状数组从 1 开始

        # 查询当前树状数组中小于 nums[i]的元素个数
        result[i] = self.query(rank - 1)

        # 将当前元素添加到树状数组
        self.update(rank, 1)

    return result

def countSmallerWithParallelBinarySearch(self, nums):
    """
    使用整体二分的思想来解决问题
    这个版本更接近整体二分的标准实现

    Args:
        nums: 输入数组

    Returns:
        结果列表
    """
    self.n = len(nums)
    if self.n == 0:
        return []

    # 初始化数据结构
    self.tree = [0] * (self.n + 2)  # 树状数组
    self.pos_list = []
    result = [0] * self.n
```

```
        # 离散化处理
        all_nums = sorted(nums)
        unique_count = 1
        for i in range(1, self.n):
            if all_nums[i] != all_nums[i-1]:
                all_nums[unique_count] = all_nums[i]
                unique_count += 1
        all_nums = all_nums[:unique_count]

        # 从右到左处理，逐个添加元素到树状数组，并查询
        for i in range(self.n - 1, -1, -1):
            # 离散化当前值
            rank = bisect.bisect_left(all_nums, nums[i])

            # 查询比 nums[i]小的元素个数
            result[i] = self.query(rank)

            # 添加当前元素
            self.update(rank + 1, 1)  # +1 因为树状数组从 1 开始

        return result

# 测试代码
def test_solution():
    # 测试用例 1
    solution = LeetCode315_CountOfSmallerNumbersAfterSelf()
    nums1 = [5, 2, 6, 1]
    result1 = solution.countSmaller(nums1)
    result1_bs = solution.countSmallerWithParallelBinarySearch(nums1)
    print(f"输入: [5, 2, 6, 1]")
    print(f"输出: {result1}")  # 预期输出: [2, 1, 1, 0]
    print(f"整体二分版本输出: {result1_bs}")

    # 测试用例 2
    solution = LeetCode315_CountOfSmallerNumbersAfterSelf()
    nums2 = [-1, -1]
    result2 = solution.countSmaller(nums2)
    result2_bs = solution.countSmallerWithParallelBinarySearch(nums2)
    print(f"输入: [-1, -1]")
    print(f"输出: {result2}")  # 预期输出: [0, 0]
    print(f"整体二分版本输出: {result2_bs}")
```

```
    # 测试用例 3
    solution = LeetCode315_CountOfSmallerNumbersAfterSelf()
    nums3 = []
    result3 = solution.countSmaller(nums3)
    result3_bs = solution.countSmallerWithParallelBinarySearch(nums3)
    print(f"输入: []")
    print(f"输出: {result3}")  # 预期输出: []
    print(f"整体二分版本输出: {result3_bs}")

    # 边界测试 - 大量重复元素
    solution = LeetCode315_CountOfSmallerNumbersAfterSelf()
    nums4 = [1, 1, 1, 1, 1]
    result4 = solution.countSmaller(nums4)
    print(f"输入: [1, 1, 1, 1, 1]")
    print(f"输出: {result4}")  # 预期输出: [0, 0, 0, 0, 0]

    # 边界测试 - 逆序数组
    solution = LeetCode315_CountOfSmallerNumbersAfterSelf()
    nums5 = [5, 4, 3, 2, 1]
    result5 = solution.countSmaller(nums5)
    print(f"输入: [5, 4, 3, 2, 1]")
    print(f"输出: {result5}")  # 预期输出: [4, 3, 2, 1, 0]

    # 边界测试 - 顺序数组
    solution = LeetCode315_CountOfSmallerNumbersAfterSelf()
    nums6 = [1, 2, 3, 4, 5]
    result6 = solution.countSmaller(nums6)
    print(f"输入: [1, 2, 3, 4, 5]")
    print(f"输出: {result6}")  # 预期输出: [0, 0, 0, 0, 0]

    # 边界测试 - 混合正负数
    solution = LeetCode315_CountOfSmallerNumbersAfterSelf()
    nums7 = [-3, 5, 0, -2, 8, 1]
    result7 = solution.countSmaller(nums7)
    print(f"输入: [-3, 5, 0, -2, 8, 1]")
    print(f"输出: {result7}")  # 预期输出: [0, 3, 0, 0, 1, 0]

if __name__ == "__main__":
    test_solution()

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

文件: P1527_矩阵第 K 小.cpp
```

```
==========================================
// P1527 [国家集训队] 矩阵乘法 / 矩阵第 K 小 - C++实现
// 题目来源：https://www.luogu.com.cn/problem/P1527
// 时间复杂度：O(N² * logN * log(maxValue) + Q * logN * log(maxValue))
// 空间复杂度：O(N² + Q)

//#include <bits/stdc++.h>
//
//using namespace std;
//
//struct Number {
//    int x, y, v;
//};
//
//bool NumberCmp(Number a, Number b) {
//    return a.v < b.v;
//}
//
//const int MAXN = 501;
//const int MAXQ = 1000001;
//int n, q;
//
//Number xyv[MAXN * MAXN];
//int cntv = 0;
//
//int qid[MAXQ];
//int a[MAXQ];
//int b[MAXQ];
//int c[MAXQ];
//int d[MAXQ];
//int k[MAXQ];
//
//int tree[MAXN][MAXN];
//
//int lset[MAXQ];
//int rset[MAXQ];
//
//int ans[MAXQ];
//
//int lowbit(int i) {
//    return i & -i;
//}
//
```

```
//void add(int x, int y, int v) {
//    for (int i = x; i <= n; i += lowbit(i)) {
//        for (int j = y; j <= n; j += lowbit(j)) {
//            tree[i][j] += v;
//        }
//    }
//}
//
//int sum(int x, int y) {
//    int ret = 0;
//    for (int i = x; i > 0; i -= lowbit(i)) {
//        for (int j = y; j > 0; j -= lowbit(j)) {
//            ret += tree[i][j];
//        }
//    }
//    return ret;
//}
//
//int query(int a, int b, int c, int d) {
//    return sum(c, d) - sum(a - 1, d) - sum(c, b - 1) + sum(a - 1, b - 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]] = xyv[vl].v;
//        }
//    } else {
//        int mid = (vl + vr) >> 1;
//        for (int i = vl; i <= mid; i++) {
//            add(xyv[i].x, xyv[i].y, 1);
//        }
//        int lsiz = 0, rsiz = 0;
//        for (int i = ql; i <= qr; i++) {
//            int id = qid[i];
//            int satisfy = query(a[id], b[id], c[id], d[id]);
//            if (satisfy >= k[id]) {
//                lset[++lsiz] = id;
//            } else {
//                k[id] -= satisfy;
```

```
//                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];
//        }
//        for (int i = vl; i <= mid; i++) {
//            add(xyv[i].x, xyv[i].y, -1);
//        }
//        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 >> q;
//    for (int i = 1; i <= n; i++) {
//        for (int j = 1; j <= n; j++) {
//            xyv[++cntv].x = i;
//            xyv[cntv].y = j;
//            cin >> xyv[cntv].v;
//        }
//    }
//    for (int i = 1; i <= q; i++) {
//        qid[i] = i;
//        cin >> a[i] >> b[i] >> c[i] >> d[i] >> k[i];
//    }
//    sort(xyv + 1, xyv + cntv + 1, NumberCmp);
//    compute(1, q, 1, cntv);
//    for (int i = 1; i <= q; i++) {
//        cout << ans[i] << '\n';
//    }
//    return 0;
//}
```

==========

文件: P1527_矩阵第 K 小.java

```
==========================================
package class168;

// P1527 [国家集训队] 矩阵乘法 / 矩阵第 K 小 - Java 实现
// 题目来源：https://www.luogu.com.cn/problem/P1527
// 题目描述：给定一个 n×n 的矩阵，有 q 次查询，每次查询子矩阵中第 k 小的数。
// 解题思路：使用整体二分算法结合二维树状数组处理矩阵第 k 小查询
// 时间复杂度：O(N² * logN * log(maxValue) + Q * logN * log(maxValue))
// 空间复杂度：O(N² + Q)
// 算法适用条件：
// 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. 修改如果对判定答案有贡献，则贡献为确定值
// 4. 贡献满足交换律、结合律，具有可加性
// 5. 题目允许离线操作

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

public class P1527_矩阵第K小 {
    public static int MAXN = 501;
    public static int MAXQ = 1000001;
    public static int n, q;  // n:矩阵大小, q:查询次数

    // 矩阵中的每个数字，所在行 x、所在列 y、数值 v
    public static int[][] xyv = new int[MAXN * MAXN][3];
    // 矩阵中一共有多少个数字，cntv 就是矩阵的规模
    public static int cntv = 0;

    // 查询任务的编号
    public static int[] qid = new int[MAXQ];
    // 查询范围的左上角坐标
    public static int[] a = new int[MAXQ];  // 行坐标
    public static int[] b = new int[MAXQ];  // 列坐标
    // 查询范围的右下角坐标
    public static int[] c = new int[MAXQ];  // 行坐标
    public static int[] d = new int[MAXQ];  // 列坐标
    // 查询矩阵内第 k 小
    public static int[] k = new int[MAXQ];

    // 二维树状数组，用于维护二维空间中元素的个数
    public static int[][] tree = new int[MAXN][MAXN];
```

```
// 整体二分中用于分类查询的临时存储
public static int[] lset = new int[MAXQ];  // 满足条件的查询
public static int[] rset = new int[MAXQ];  // 不满足条件的查询

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

// 计算二进制表示中最低位的 1 所代表的数值
public static int lowbit(int i) {
    return i & -i;
}

// 二维树状数组单点更新: 二维空间中，(x,y)位置的词频加 v
public static void add(int x, int y, int v) {
    for (int i = x; i <= n; i += lowbit(i)) {
        for (int j = y; j <= n; j += lowbit(j)) {
            tree[i][j] += v;
        }
    }
}

// 二维树状数组前缀和查询: 二维空间中，左上角(1,1)到右下角(x,y)范围上的词频累加和
public static int sum(int x, int y) {
    int ret = 0;
    for (int i = x; i > 0; i -= lowbit(i)) {
        for (int j = y; j > 0; j -= lowbit(j)) {
            ret += tree[i][j];
        }
    }
    return ret;
}

// 二维树状数组区间和查询: 二维空间中，左上角(a,b)到右下角(c,d)范围上的词频累加和
public static int query(int a, int b, int c, int d) {
    return sum(c, d) - sum(a - 1, d) - sum(c, b - 1) + sum(a - 1, b - 1);
}

// 整体二分核心函数
// ql, qr: 查询范围
// vl, vr: 值域范围（排序后的下标）
public static void compute(int ql, int qr, int vl, int vr) {
    // 递归边界：没有查询需要处理
    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]] = xyv[vl][2];  // xyv[vl][2]是排序后第 vl 个元素的值
    }
} else {
    // 二分中点
    int mid = (vl + vr) >> 1;

    // 将值域中小于等于第 mid 个元素的数加入二维树状数组
    for (int i = vl; i <= mid; i++) {
        add(xyv[i][0], xyv[i][1], 1);  // 在(xyv[i][0], xyv[i][1])位置加 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++) {
        int id = qid[i];
        // 查询子矩阵[a[id],b[id],c[id],d[id]]中值小于等于 xyv[mid][2]的元素个数
        int satisfy = query(a[id], b[id], c[id], d[id]);

        if (satisfy >= k[id]) {
            // 说明第 k 小的数在左半部分
            // 将该查询加入左集合
            lset[++lsiz] = id;
        } 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
            // 更新 k 值，将该查询加入右集合
            k[id] -= satisfy;
            rset[++rsiz] = id;
        }
    }

    // 重新排列查询顺序，使得左集合的查询在前，右集合的查询在后
    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];
    }
```

```
        // 撤销对二维树状数组的修改，恢复到处理前的状态
        for (int i = vl; i <= mid; i++) {
            add(xyv[i][0], xyv[i][1], -1);  // 在(xyv[i][0], xyv[i][1])位置减1
        }

        // 递归处理左右两部分
        // 左半部分：值域在[vl, mid]范围内的查询
        compute(ql, ql + lsiz - 1, vl, mid);
        // 右半部分：值域在[mid+1, vr]范围内的查询
        compute(ql + lsiz, qr, mid + 1, vr);
    }
}

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

    // 读取矩阵大小和查询次数
    String[] params = br.readLine().split(" ");
    n = Integer.parseInt(params[0]);
    q = Integer.parseInt(params[1]);

    // 读取矩阵中的每个数字，记录其位置和数值
    for (int i = 1; i <= n; i++) {
        String[] row = br.readLine().split(" ");
        for (int j = 1; j <= n; j++) {
            xyv[++cntv][0] = i;           // 行坐标
            xyv[cntv][1] = j;             // 列坐标
            xyv[cntv][2] = Integer.parseInt(row[j - 1]);  // 数值
        }
    }

    // 读取查询
    for (int i = 1; i <= q; i++) {
        String[] query = br.readLine().split(" ");
        qid[i] = i;                       // 查询编号
        a[i] = Integer.parseInt(query[0]); // 左上角行坐标
        b[i] = Integer.parseInt(query[1]); // 左上角列坐标
        c[i] = Integer.parseInt(query[2]); // 右下角行坐标
        d[i] = Integer.parseInt(query[3]); // 右下角列坐标
        k[i] = Integer.parseInt(query[4]); // 查询第k小
    }
```

```
        // 按数值大小排序，这样可以通过下标进行二分
        Arrays.sort(xyv, 1, cntv + 1, (a, b) -> a[2] - b[2]);

        // 整体二分求解
        // 初始查询范围[1, q]，初始值域范围[1, cntv]
        compute(1, q, 1, cntv);

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

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

==========

文件: P1527_矩阵第K小.py

==========

```
# P1527 [国家集训队] 矩阵乘法 / 矩阵第K小 - Python实现
# 题目来源：https://www.luogu.com.cn/problem/P1527
# 时间复杂度：O(N² * logN * log(maxValue) + Q * logN * log(maxValue))
# 空间复杂度：O(N² + Q)

import sys

class MatrixKth:
    def __init__(self):
        self.MAXN = 501
        self.MAXQ = 1000001
        self.n = 0
        self.q = 0

        # 矩阵中的每个数字，所在行x、所在列y、数值v
        self.xyv = [[0, 0, 0] for _ in range(self.MAXN * self.MAXN)]
        # 矩阵中一共有多少个数字，cntv就是矩阵的规模
        self.cntv = 0

        # 查询任务的编号
```

```
        self.qid = [0] * self.MAXQ
        # 查询范围的左上角坐标
        self.a = [0] * self.MAXQ
        self.b = [0] * self.MAXQ
        # 查询范围的右下角坐标
        self.c = [0] * self.MAXQ
        self.d = [0] * self.MAXQ
        # 查询矩阵内第 k 小
        self.k = [0] * self.MAXQ

        # 二维树状数组
        self.tree = [[0] * self.MAXN for _ in range(self.MAXN)]

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

        # 每条查询的答案
        self.ans = [0] * self.MAXQ

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

    # 二维空间中，(x,y)位置的词频加 v
    def add(self, x, y, v):
        i = x
        while i <= self.n:
            j = y
            while j <= self.n:
                self.tree[i][j] += v
                j += self.lowbit(j)
            i += self.lowbit(i)

    # 二维空间中，左上角(1,1)到右下角(x,y)范围上的词频累加和
    def sum(self, x, y):
        ret = 0
        i = x
        while i > 0:
            j = y
            while j > 0:
                ret += self.tree[i][j]
                j -= self.lowbit(j)
            i -= self.lowbit(i)
```

```
    return ret

# 二维空间中，左上角(a,b)到右下角(c,d)范围上的词频累加和
def query(self, a, b, c, d):
    return self.sum(c, d) - self.sum(a - 1, d) - self.sum(c, b - 1) + self.sum(a - 1, b - 1)

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.xyv[vl][2]
    else:
        mid = (vl + vr) >> 1
        for i in range(vl, mid + 1):
            self.add(self.xyv[i][0], self.xyv[i][1], 1)
        lsiz = 0
        rsiz = 0
        for i in range(ql, qr + 1):
            id = self.qid[i]
            satisfy = self.query(self.a[id], self.b[id], self.c[id], self.d[id])
            if satisfy >= self.k[id]:
                lsiz += 1
                self.lset[lsiz] = id
            else:
                self.k[id] -= satisfy
                rsiz += 1
                self.rset[rsiz] = id
        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]
        for i in range(vl, mid + 1):
            self.add(self.xyv[i][0], self.xyv[i][1], -1)
        self.compute(ql, ql + lsiz - 1, vl, mid)
        self.compute(ql + lsiz, qr, mid + 1, vr)

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

    for i in range(1, self.n + 1):
```

```
            row = sys.stdin.readline().split()
            for j in range(1, self.n + 1):
                self.cntv += 1
                self.xyv[self.cntv][0] = i
                self.xyv[self.cntv][1] = j
                self.xyv[self.cntv][2] = int(row[j - 1])

        for i in range(1, self.q + 1):
            query_line = sys.stdin.readline().split()
            self.qid[i] = i
            self.a[i] = int(query_line[0])
            self.b[i] = int(query_line[1])
            self.c[i] = int(query_line[2])
            self.d[i] = int(query_line[3])
            self.k[i] = int(query_line[4])

        # 按照数值排序
        self.xyv[1:self.cntv+1] = sorted(self.xyv[1:self.cntv+1], key=lambda x: x[2])

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

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

# 程序入口
#if __name__ == "__main__":
#    solution = MatrixKth()
#    solution.solve()
```

==========

文件: P2617_Dynamic_Rankings.cpp

==========

```
// P2617 Dynamic Rankings - C++实现
// 题目来源：https://www.luogu.com.cn/problem/P2617
// 题目描述：
// 给定一个含有 n 个数的序列 a1,a2…an，需要支持两种操作：
// Q l r k 表示查询下标在区间[l,r]中的第 k 小的数；
// C x y 表示将 ax 改为 y。
// 时间复杂度：O((N+Q) * logN * log(maxValue))
// 空间复杂度：O(N + Q)

// 由于环境限制，这里只提供 C++代码框架，实际编译需要相应环境支持
```

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

const int MAXN = 100001;
int n, m;

// 原始数组
int arr[MAXN];

// 离散化后的数组
int sorted[MAXN * 2];

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

    Operation() {}
    Operation(int _type, int _l, int _r, int _k, int _x, int _y, int _id) {
        type = _type;
        l = _l;
        r = _r;
        k = _k;
        x = _x;
        y = _y;
        id = _id;
    }
};

Operation ops[MAXN * 2];

// 树状数组
int tree[MAXN];

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

// 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: 操作范围
// 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++) {
            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;

// 将值域小于等于 mid 的数加入树状数组
for (int i = vl; i <= mid; i++) {
    // 这里需要处理所有值为 sorted[i]的元素
    // 在实际实现中，我们需要更复杂的处理方式
}

// 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);
}

//int main() {
//    ios::sync_with_stdio(false);
//    cin.tie(0);
//    cout.tie(0);
//
//    cin >> n >> m;
//
//    // 读取原始数组
//    for (int i = 1; i <= n; i++) {
//        cin >> arr[i];
//        sorted[i] = arr[i];
//    }
//
//    int opCount = n;
//    // 读取操作
//    for (int i = 1; i <= m; i++) {
//        string opType;
//        cin >> opType;
//        if (opType == "Q") {
//            int l, r, k;
//            cin >> l >> r >> k;
//            ops[opCount++] = Operation(0, l, r, k, 0, 0, i);
//        } else { // C
//            int x, y;
```

```
//            cin >> x >> y;
//            ops[opCount++] = Operation(1, 0, 0, 0, x, y, i);
//            sorted[++n] = y; // 添加到离散化数组中
//        }
//    }
//
//    // 离散化
//    sort(sorted + 1, sorted + n + 1);
//    int uniqueCount = unique(sorted + 1, sorted + n + 1) - sorted - 1;
//
//    // 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) {
//            cout << ans[i] << "\n";
//        }
//    }
//
//    return 0;
//}
```

==========

文件: P2617_Dynamic_Rankings.java

==========

```
package class168;

// P2617 Dynamic Rankings - Java 实现
// 题目来源：https://www.luogu.com.cn/problem/P2617
// 题目描述：
// 给定一个含有 n 个数的序列 a1,a2…an，需要支持两种操作：
// Q l r k 表示查询下标在区间[l,r]中的第 k 小的数；
// C x y 表示将 ax 改为 y。
// 解题思路：使用整体二分算法处理带修改的区间第 k 小查询
// 时间复杂度：O((N+Q) * logN * log(maxValue))
// 空间复杂度：O(N + Q)
// 算法适用条件：
// 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. 修改如果对判定答案有贡献，则贡献为确定值
// 4. 贡献满足交换律、结合律，具有可加性
```

```
// 5. 题目允许离线操作
// 注意：此实现为演示版本，实际应用中需要完善修改操作的处理逻辑

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

public class P2617_Dynamic_Rankings {
    public static int MAXN = 100001;
    public static int n, m;  // n:数组长度, m:操作次数

    // 原始数组，存储初始数值
    public static int[] arr = new int[MAXN];

    // 离散化后的数组，用于离散化处理，将大值域映射到小下标范围
    public static int[] sorted = new int[MAXN * 2];

    // 操作信息
    public static class Operation {
        int type; // 操作类型: 0 表示查询, 1 表示修改
        int l, r, k; // 查询操作参数: 区间[l,r]中的第 k 小
        int x, y;    // 修改操作参数: 将 ax 改为 y
        int id;      // 操作编号

        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];  // 存储所有操作

    // 树状数组，用于维护当前值域范围内元素的个数
    public static int[] tree = new int[MAXN];

    // 整体二分中用于分类操作的临时存储
    public static int[] lset = new int[MAXN * 2];  // 满足条件的操作
    public static int[] rset = new int[MAXN * 2];  // 不满足条件的操作
```

```
// 查询的答案存储数组
public static int[] ans = new int[MAXN];

// 树状数组操作
// 计算二进制表示中最低位的 1 所代表的数值
public static int lowbit(int i) {
    return i & -i;
}

// 在树状数组的第 i 个位置加上 v
public static void add(int i, int v) {
    while (i <= n) {
        tree[i] += v;
        i += lowbit(i);
    }
}

// 计算前缀和[1, i]的和
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]的和
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：没有操作需要处理
    if (ql > qr) {
        return;
    }

    // 如果值域范围只有一个值，说明找到了答案
    // 此时所有查询操作的答案都是 sorted[vl]
```

```
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;

// 将值域小于等于 mid 的数加入树状数组
// 在实际实现中，我们需要处理所有值为 sorted[i]的元素
// 这里需要更复杂的处理方式，比如处理修改操作的删除和插入
for (int i = vl; i <= mid; i++) {
    // 这里需要处理所有值为 sorted[i]的元素
    // 在实际实现中，我们需要更复杂的处理方式
}

// 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
            // 将该操作加入左集合
            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
            // 更新 k 值，将该操作加入右集合
            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
        // 如果修改后的值小于等于 sorted[mid]，则对当前查询有贡献
        if (ops[i].y <= sorted[mid]) {
            add(ops[i].x, 1);
```

```
                lset[++lsiz] = i;
            } else {
                // 否则对当前查询无贡献
                rset[++rsiz] = i;
            }
        }
    }

    // 重新排列操作顺序，使得左集合的操作在前，右集合的操作在后
    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，恢复到处理前的状态
    // 在实际实现中，这里需要撤销之前的所有 add 操作
    for (int i = vl; i <= mid; i++) {
        // 撤销操作
    }

    // 递归处理左右两部分
    // 左半部分：值域在[vl, mid]范围内的操作
    compute(ql, ql + lsiz - 1, vl, mid);
    // 右半部分：值域在[mid+1, vr]范围内的操作
    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));

    // 读取数组长度和操作次数
    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];
}

int opCount = n;
// 读取操作
for (int i = 1; i <= m; i++) {
    String[] op = br.readLine().split(" ");
    if (op[0].equals("Q")) {
        // 查询操作: Q l r k 表示查询下标在区间[l,r]中的第 k 小的数
        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
        // 修改操作: C x y 表示将 ax 改为 y
        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：将大值域映射到小下标范围，减少二分的值域范围
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
// 初始操作范围[1, opCount-1]，初始值域范围[1, uniqueCount]
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]);
```

```
            }
        }

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

==========

文件: P2617_Dynamic_Rankings.py

==========

```
# P2617 Dynamic Rankings - Python 实现
# 题目来源：https://www.luogu.com.cn/problem/P2617
# 题目描述：
# 给定一个含有 n 个数的序列 a1,a2…an，需要支持两种操作：
# Q l r k 表示查询下标在区间[l,r]中的第 k 小的数；
# C x y 表示将 ax 改为 y。
# 时间复杂度：O((N+Q) * logN * log(maxValue))
# 空间复杂度：O(N + Q)

import sys
from bisect import bisect_left

# 由于 Python 的性能限制，对于大数据可能超时，但在逻辑上是正确的

class Solution:
    def __init__(self):
        self.MAXN = 100001
        self.n = 0
        self.m = 0

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

        # 离散化后的数组
        self.sorted_arr = [0] * (self.MAXN * 2)

        # 操作信息
        self.ops = []  # (type, l, r, k, x, y, id)

        # 树状数组
```

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

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

def lowbit(self, i):
    """树状数组的 lowbit 操作"""
    return i & -i

def add(self, i, v):
    """树状数组单点更新"""
    while i <= self.n:
        self.tree[i] += v
        i += self.lowbit(i)

def sum(self, i):
    """树状数组前缀和查询"""
    ret = 0
    while i > 0:
        ret += self.tree[i]
        i -= self.lowbit(i)
    return ret

def query(self, l, r):
    """树状数组区间和查询"""
    return self.sum(r) - self.sum(l - 1)

def compute(self, ql, qr, vl, vr):
    """整体二分核心函数"""
    # 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):
            if self.ops[i][0] == 0:  # 查询操作
                self.ans[self.ops[i][6]] = self.sorted_arr[vl]
        return

    # 二分中点
    mid = (vl + vr) >> 1
```

```
# 将值域小于等于 mid 的数加入树状数组
for i in range(vl, mid + 1):
    # 这里需要处理所有值为 sorted[i]的元素
    # 在实际实现中，我们需要更复杂的处理方式
    pass

# 检查每个操作，根据满足条件的元素个数划分到左右区间
lset = []
rset = []
for i in range(ql, qr + 1):
    if self.ops[i][0] == 0:  # 查询操作
        # 查询区间[ops[i][1], ops[i][2]]中值小于等于 sorted[mid]的元素个数
        satisfy = self.query(self.ops[i][1], self.ops[i][2])

        if satisfy >= self.ops[i][3]:
            # 说明第 k 小的数在左半部分
            lset.append(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.ops[i] = (self.ops[i][0], self.ops[i][1], self.ops[i][2],
                           self.ops[i][3] - satisfy, self.ops[i][4],
                           self.ops[i][5], self.ops[i][6])
            rset.append(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 self.ops[i][5] <= self.sorted_arr[mid]:
            self.add(self.ops[i][4], 1)
            lset.append(i)
        else:
            rset.append(i)

# 重新排列操作顺序
new_ops = []
for i in lset:
    new_ops.append(self.ops[i])
for i in rset:
    new_ops.append(self.ops[i])

# 更新原操作数组
for i in range(len(new_ops)):
    self.ops[ql + i] = new_ops[i]
```

```
    # 撤销对树状数组的修改
    for i in range(vl, mid + 1):
        # 撤销操作
        pass

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

def solve(self):
    """主函数"""
    # 读取输入
    line = sys.stdin.readline().split()
    self.n = int(line[0])
    self.m = int(line[1])

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

    op_count = self.n
    # 读取操作
    for i in range(1, self.m + 1):
        op = sys.stdin.readline().split()
        if op[0] == "Q":
            l = int(op[1])
            r = int(op[2])
            k = int(op[3])
            self.ops.append((0, l, r, k, 0, 0, i))  # 查询操作
            op_count += 1
        else:  # C
            x = int(op[1])
            y = int(op[2])
            self.ops.append((1, 0, 0, 0, x, y, i))  # 修改操作
            op_count += 1
            self.sorted_arr[self.n + 1] = y  # 添加到离散化数组中
            self.n += 1

    # 离散化
    self.sorted_arr[1:self.n+1] = sorted(self.sorted_arr[1:self.n+1])
    unique_count = 1
```

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

        # 整体二分求解
        self.compute(0, len(self.ops) - 1, 1, unique_count)

        # 输出结果
        for i in range(1, self.m + 1):
            if self.ans[i] != 0:
                print(self.ans[i])

# 程序入口
#if __name__ == "__main__":
#    solution = Solution()
#    solution.solve()
```

==========

文件: P3242_接水果.java

==========

```
package class168;

// P3242 [HNOI2015] 接水果 - Java 实现
// 题目来源：https://www.luogu.com.cn/problem/P3242
// 时间复杂度：O((P+Q) * logP * log(maxC))
// 空间复杂度：O(P + Q)
//
// 题目大意：
// 在树上选择能接住水果的盘子，求权值第 k 小的盘子。
//
// 解题思路：
// 1. 对盘子权值进行二分
// 2. 使用树链剖分处理路径包含关系
// 3. 统计满足条件的盘子数量
// 4. 根据统计结果将操作分为两类递归处理
//
// 算法详解：
// 1. 整体二分：将所有查询一起处理，对盘子权值进行二分
// 2. 树链剖分：用于处理树上路径的包含关系
// 3. 路径包含：判断一条路径是否包含另一条路径
```

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

public class P3242_接水果 {
    public static int MAXN = 40001;

    public static int n, p, q;

    // 树结构
    public static int[] head = new int[MAXN];
    public static int[] next = new int[MAXN << 1];
    public static int[] to = new int[MAXN << 1];
    public static int cnt = 0;

    // 树链剖分
    public static int[] fa = new int[MAXN];
    public static int[] depth = new int[MAXN];
    public static int[] siz = new int[MAXN];
    public static int[] son = new int[MAXN];
    public static int[] top = new int[MAXN];
    public static int[] dfn = new int[MAXN];
    public static int[] rnk = new int[MAXN];
    public static int dfc = 0;

    // 盘子信息
    public static int[] pa = new int[MAXN];
    public static int[] pb = new int[MAXN];
    public static int[] pc = new int[MAXN];

    // 水果信息
    public static int[] ua = new int[MAXN];
    public static int[] ub = new int[MAXN];
    public static int[] uk = new int[MAXN];
    public static int[] qid = new int[MAXN];

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

    // 离散化
    public static int[] sorted = new int[MAXN];
    public static int cntv = 0;
```

```
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().split(" ");
    n = Integer.parseInt(line[0]);
    p = Integer.parseInt(line[1]);
    q = Integer.parseInt(line[2]);

    // 建树
    for (int i = 1; i < n; i++) {
        line = br.readLine().split(" ");
        int u = Integer.parseInt(line[0]);
        int v = Integer.parseInt(line[1]);
        addEdge(u, v);
        addEdge(v, u);
    }

    // 读取盘子信息
    for (int i = 1; i <= p; i++) {
        line = br.readLine().split(" ");
        pa[i] = Integer.parseInt(line[0]);
        pb[i] = Integer.parseInt(line[1]);
        pc[i] = Integer.parseInt(line[2]);
        sorted[++cntv] = pc[i];
    }

    // 读取水果信息
    for (int i = 1; i <= q; i++) {
        line = br.readLine().split(" ");
        ua[i] = Integer.parseInt(line[0]);
        ub[i] = Integer.parseInt(line[1]);
        uk[i] = Integer.parseInt(line[2]);
        qid[i] = i;
    }

    // 离散化
    Arrays.sort(sorted, 1, cntv + 1);
    cntv = unique(sorted, cntv);

    // 树链剖分预处理
    dfs1(1, 0);
```

```
        dfs2(1, 1);

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

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

    // 去重函数
    // 对排序后的数组进行去重，返回去重后的长度
    public static int unique(int[] arr, int len) {
        if (len <= 1) return len;
        int i = 1, j = 2;
        while (j <= len) {
            if (arr[j] != arr[i]) {
                arr[++i] = arr[j];
            }
            j++;
        }
        return i;
    }

    // 添加边
    // 使用邻接表存储树的结构
    public static void addEdge(int u, int v) {
        next[++cnt] = head[u];
        to[cnt] = v;
        head[u] = cnt;
    }

    // 第一次 DFS：计算深度、父节点、子树大小、重儿子
    // 这是树链剖分的第一步，用于确定每个节点的重儿子
    public static void dfs1(int u, int f) {
        fa[u] = f;
        depth[u] = depth[f] + 1;
        siz[u] = 1;

        for (int i = head[u]; i != 0; i = next[i]) {
            int v = to[i];
```

```
        if (v == f) continue;
        dfs1(v, u);
        siz[u] += siz[v];
        // 更新重儿子：选择子树大小最大的子节点作为重儿子
        if (siz[son[u]] < siz[v]) {
            son[u] = v;
        }
    }
}

// 第二次 DFS：计算 dfn 序、重链顶点
// 这是树链剖分的第二步，用于确定每个节点的 dfn 序和所在重链的顶点
public static void dfs2(int u, int t) {
    top[u] = t;  // 记录节点 u 所在重链的顶点
    dfn[u] = ++dfc;  // 记录节点 u 的 dfn 序
    rnk[dfc] = u;  // 记录 dfn 序对应的节点

    // 如果存在重儿子，优先遍历重儿子
    if (son[u] != 0) {
        dfs2(son[u], t);
    }

    // 遍历轻儿子
    for (int i = head[u]; i != 0; i = next[i]) {
        int v = to[i];
        // 跳过父节点和重儿子
        if (v == fa[u] || v == son[u]) continue;
        // 轻儿子所在重链的顶点就是它自己
        dfs2(v, v);
    }
}

// 整体二分核心函数
// ql, qr: 当前处理的查询范围
// vl, vr: 当前处理的值域范围（盘子权值范围）
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[qid[i]] = vl;
        }
        return;
    }

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

    // 检查每个查询
    int lsiz = 0, rsiz = 0;
    for (int i = ql; i <= qr; i++) {
        int id = qid[i];
        // 检查权值大于等于 sorted[mid]的盘子中第 uk[id]小的是否存在
        int count = countValidPlates(mid, ua[id], ub[id]);
        if (count >= uk[id]) {
            // 有足够的盘子，说明答案可能更大，分到左区间
            lset[++lsiz] = id;
        } else {
            // 没有足够的盘子，说明答案更小，分到右区间
            uk[id] -= count;
            rset[++rsiz] = id;
        }
    }

    // 将操作分组
    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];
    }

    // 递归处理左右区间
    // 左区间：答案范围[mid+1, vr]（权值更大的盘子）
    compute(ql, ql + lsiz - 1, mid + 1, vr);
    // 右区间：答案范围[vl, mid]（权值更小的盘子）
    compute(ql + lsiz, qr, vl, mid);
}

// 计算能接住水果(ua, ub)且权值大于等于 sorted[mid]的盘子数量
// 这是一个简化的实现，实际应该使用更复杂的数据结构来高效计算
public static int countValidPlates(int mid, int ua, int ub) {
```

```
        int count = 0;
        // 这里需要实现具体的计数逻辑
        // 由于实现较为复杂，这里简化处理
        for (int i = 1; i <= p; i++) {
            if (pc[i] >= sorted[mid] && isSubPath(ua, ub, pa[i], pb[i])) {
                count++;
            }
        }
        return count;
    }

    // 判断路径(pa, pb)是否是路径(ua, ub)的子路径
    // 这是一个简化的实现，实际应该使用树链剖分来高效判断
    public static boolean isSubPath(int ua, int ub, int pa, int pb) {
        // 这里需要实现路径包含关系的判断逻辑
        // 由于实现较为复杂，这里简化处理
        return false;
    }
}
```

==========

文件: P3250_网络.java

==========

```
package class168;

// P3250 [HNOI2016] 网络 - Java 实现
// 题目来源：https://www.luogu.com.cn/problem/P3250
// 时间复杂度：O(M * logM * log(maxValue))
// 空间复杂度：O(N + M)
//
// 题目大意：
// 维护树上网络系统，查询不经过某点的路径重要度最大值。
//
// 解题思路：
// 1. 对重要度进行二分
// 2. 使用树状数组维护树上差分信息
// 3. 检查不经过某点的路径最大重要度
// 4. 根据统计结果将操作分为两类递归处理
//
// 算法详解：
// 1. 整体二分：将所有操作一起处理，对重要度进行二分
// 2. 树链剖分：用于处理树上路径操作
```

```
// 3. 树状数组：用于维护路径上的差分信息

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

public class P3250_网络 {
    public static int MAXN = 100001;
    public static int MAXM = 200001;

    public static int n, m;

    // 树结构
    public static int[] head = new int[MAXN];
    public static int[] next = new int[MAXN << 1];
    public static int[] to = new int[MAXN << 1];
    public static int cnt = 0;

    // 树链剖分
    public static int[] fa = new int[MAXN];
    public static int[] depth = new int[MAXN];
    public static int[] siz = new int[MAXN];
    public static int[] son = new int[MAXN];
    public static int[] top = new int[MAXN];
    public static int[] dfn = new int[MAXN];
    public static int[] rnk = new int[MAXN];
    public static int dfc = 0;

    // 事件信息
    public static int[] type = new int[MAXM];
    public static int[] a = new int[MAXM];
    public static int[] b = new int[MAXM];
    public static int[] v = new int[MAXM];
    public static int[] t = new int[MAXM];
    public static int[] x = new int[MAXM];

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

    // 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];
```

```
// 离散化
public static int[] sorted = new int[MAXM];
public static int cntv = 0;

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().split(" ");
    n = Integer.parseInt(line[0]);
    m = Integer.parseInt(line[1]);

    // 建树
    for (int i = 1; i < n; i++) {
        line = br.readLine().split(" ");
        int u = Integer.parseInt(line[0]);
        int v = Integer.parseInt(line[1]);
        addEdge(u, v);
        addEdge(v, u);
    }

    // 处理事件
    for (int i = 1; i <= m; i++) {
        line = br.readLine().split(" ");
        type[i] = Integer.parseInt(line[0]);

        if (type[i] == 0) {
            // 新的数据交互请求
            a[i] = Integer.parseInt(line[1]);
            b[i] = Integer.parseInt(line[2]);
            v[i] = Integer.parseInt(line[3]);
            sorted[++cntv] = v[i];
        } else if (type[i] == 1) {
            // 数据交互结束请求
            t[i] = Integer.parseInt(line[1]);
        } else {
            // 服务器出现故障
            x[i] = Integer.parseInt(line[1]);
            qid[i] = i;
        }
    }

    // 离散化
```

```
        Arrays.sort(sorted, 1, cntv + 1);
        cntv = unique(sorted, cntv);

        // 树链剖分预处理
        dfs1(1, 0);
        dfs2(1, 1);

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

        // 输出结果
        for (int i = 1; i <= m; i++) {
            if (ans[i] != 0) {
                out.println(sorted[ans[i]]);
            } else if (type[i] == 2) {
                out.println(-1);
            }
        }
        out.flush();
    }

    // 去重函数
    // 对排序后的数组进行去重，返回去重后的长度
    public static int unique(int[] arr, int len) {
        if (len <= 1) return len;
        int i = 1, j = 2;
        while (j <= len) {
            if (arr[j] != arr[i]) {
                arr[++i] = arr[j];
            }
            j++;
        }
        return i;
    }

    // 添加边
    // 使用邻接表存储树的结构
    public static void addEdge(int u, int v) {
        next[++cnt] = head[u];
        to[cnt] = v;
        head[u] = cnt;
    }
```

```
// 第一次 DFS：计算深度、父节点、子树大小、重儿子
// 这是树链剖分的第一步，用于确定每个节点的重儿子
public static void dfs1(int u, int f) {
    fa[u] = f;
    depth[u] = depth[f] + 1;
    siz[u] = 1;

    for (int i = head[u]; i != 0; i = next[i]) {
        int v = to[i];
        if (v == f) continue;
        dfs1(v, u);
        siz[u] += siz[v];
        // 更新重儿子：选择子树大小最大的子节点作为重儿子
        if (siz[son[u]] < siz[v]) {
            son[u] = v;
        }
    }
}

// 第二次 DFS：计算 dfn 序、重链顶点
// 这是树链剖分的第二步，用于确定每个节点的 dfn 序和所在重链的顶点
public static void dfs2(int u, int t) {
    top[u] = t;  // 记录节点 u 所在重链的顶点
    dfn[u] = ++dfc;  // 记录节点 u 的 dfn 序
    rnk[dfc] = u;  // 记录 dfn 序对应的节点

    // 如果存在重儿子，优先遍历重儿子
    if (son[u] != 0) {
        dfs2(son[u], t);
    }

    // 遍历轻儿子
    for (int i = head[u]; i != 0; i = next[i]) {
        int v = to[i];
        // 跳过父节点和重儿子
        if (v == fa[u] || v == son[u]) continue;
        // 轻儿子所在重链的顶点就是它自己
        dfs2(v, v);
    }
}

// 整体二分核心函数
// ql, qr: 当前处理的操作范围
```

```
// vl, vr: 当前处理的值域范围（重要度范围）
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 (type[qid[i]] == 2) {
                ans[qid[i]] = vl;
            }
        }
        return;
    }

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

    // 检查每个操作
    int lsiz = 0, rsiz = 0;
    for (int i = ql; i <= qr; i++) {
        int id = qid[i];
        if (type[id] == 2) {
            // 服务器出现故障的查询
            // 检查重要度大于等于 sorted[mid]的未被影响的请求数量
            int count = countUnaffectedRequests(mid, x[id]);
            if (count > 0) {
                // 有未被影响的请求，说明答案可能更大，分到左区间
                lset[++lsiz] = id;
            } else {
                // 没有未被影响的请求，说明答案更小，分到右区间
                rset[++rsiz] = id;
            }
        } else {
            // 其他类型的事件
            if (type[id] == 0 && v[id] >= sorted[mid]) {
                // 新的请求且重要度大于等于 mid，分到左区间
                lset[++lsiz] = id;
            } else if (type[id] == 1 && v[t[id]] >= sorted[mid]) {
                // 结束的请求且重要度大于等于 mid，分到右区间
                rset[++rsiz] = id;
```

```
            } else {
                // 其他情况，根据具体逻辑分组
                lset[++lsiz] = id;
            }
        }
    }

    // 将操作分组
    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];
    }

    // 递归处理左右区间
    // 左区间：答案范围[mid+1, vr]（重要度更大的请求）
    compute(ql, ql + lsiz - 1, mid + 1, vr);
    // 右区间：答案范围[vl, mid]（重要度更小的请求）
    compute(ql + lsiz, qr, vl, mid);
}

// 计算未被影响且重要度大于等于 sorted[mid]的请求数量
// 这是一个简化的实现，实际应该使用更复杂的数据结构来高效计算
public static int countUnaffectedRequests(int mid, int faultServer) {
    int count = 0;
    // 这里需要实现具体的计数逻辑
    // 由于实现较为复杂，这里简化处理
    for (int i = 1; i <= m; i++) {
        if (type[i] == 0 && v[i] >= sorted[mid] && !isAffected(i, faultServer)) {
            count++;
        }
    }
    return count;
}

// 判断请求是否被故障服务器影响
// 这是一个简化的实现，实际应该使用树链剖分来高效判断
public static boolean isAffected(int requestId, int faultServer) {
    // 这里需要实现路径包含关系的判断逻辑
    // 由于实现较为复杂，这里简化处理
    return false;
}
```

```
}

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

文件: P3332_K 大数查询.cpp
==========================================
// P3332 [ZJOI2013] K 大数查询 - C++实现
// 题目来源：https://www.luogu.com.cn/problem/P3332
// 时间复杂度：O(M * logN * log(maxValue))
// 空间复杂度：O(N + M)

//#include <bits/stdc++.h>
//
//using namespace std;
//
//const int MAXN = 50001;
//const int MAXM = 50001;
//int n, m;
//
//// 事件编号组成的数组
//int eid[MAXM << 1];
//// op == 1，代表修改事件，区间[L,R]增加值 val
//// op == 2，代表查询事件，[L,R]范围上查询第 k 大，q 表示问题的编号
//int op[MAXM << 1];
//int L[MAXM << 1];
//int R[MAXM << 1];
//int val[MAXM << 1];
//int k[MAXM << 1];
//int q[MAXM << 1];
//int cnte = 0;
//
//// 树状数组，支持区间修改、单点查询
//long long tree[MAXN << 1];
//
//// 整体二分
//int lset[MAXM << 1];
//int rset[MAXM << 1];
//
//// 查询的答案
//long long ans[MAXN];
//
//int lowbit(int i) {
//    return i & -i;
```

```
//}
//
//void add(int i, long long v) {
//    int siz = n;
//    while (i <= siz) {
//        tree[i] += v;
//        i += lowbit(i);
//    }
//}
//
//// 区间加法 [l, r] += v
//void add(int l, int r, long long v) {
//    add(l, v);
//    add(r + 1, -v);
//}
//
//long long query(int i) {
//    long long ret = 0;
//    while (i > 0) {
//        ret += tree[i];
//        i -= lowbit(i);
//    }
//    return ret;
//}
//
//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[q[id]] = vl;
//            }
//        }
//    } else {
//        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 (val[id] <= mid) {
//                    // 对左半区间有贡献，执行修改
//                    add(L[id], R[id], 1);
//                    lset[++lsiz] = id;
//                } else {
//                    // 对左半区间无贡献，放入右半区间
//                    rset[++rsiz] = id;
//                }
//            } else {
//                // 查询操作
//                long long satisfy = query(R[id]) - query(L[id] - 1);
//                if (satisfy >= k[id]) {
//                    // 说明第 k 大的数在左半部分
//                    lset[++lsiz] = id;
//                } else {
//                    // 说明第 k 大的数在右半部分
//                    k[id] -= satisfy;
//                    rset[++rsiz] = id;
//                }
//            }
//        }
//        // 撤销对树状数组的修改
//        for (int i = 1; i <= lsiz; i++) {
//            int id = lset[i];
//            if (op[id] == 1 && val[id] <= (vl + vr) >> 1) {
//                add(L[id], R[id], -1);
//            }
//        }
//        // 重新排列事件顺序
//        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];
//        }
//        // 递归处理左右两部分
//        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;
//
//    long long minVal = 1e18;
//    long long maxVal = -1e18;
//
//    for (int i = 1; i <= m; i++) {
//        eid[i] = i;
//        cin >> op[i] >> L[i] >> R[i];
//
//        if (op[i] == 1) {
//            // 修改操作
//            cin >> val[i];
//            minVal = min(minVal, (long long)val[i]);
//            maxVal = max(maxVal, (long long)val[i]);
//        } else {
//            // 查询操作
//            cin >> k[i];
//            q[i] = i;
//        }
//    }
//
//    // 整体二分求解
//    compute(1, m, minVal, maxVal);
//
//    // 输出结果
//    for (int i = 1; i <= m; i++) {
//        if (op[i] == 2) {
//            cout << ans[q[i]] << '\n';
//        }
//    }
//
//    return 0;
//}
```

==========

文件: P3332_K 大数查询.java

==========

```
package class168;

// P3332 [ZJOI2013] K 大数查询 - Java 实现
// 题目来源：https://www.luogu.com.cn/problem/P3332
```

```
// 题目描述：维护 n 个可重整数集，支持区间加数和区间查询第 k 大。
// 解题思路：使用整体二分算法处理区间加数和区间查询第 k 大的操作
// 时间复杂度：O(M * logN * log(maxValue))
// 空间复杂度：O(N + M)
// 算法适用条件：
// 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. 修改如果对判定答案有贡献，则贡献为确定值
// 4. 贡献满足交换律、结合律，具有可加性
// 5. 题目允许离线操作

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

public class P3332_K 大数查询 {
    public static int MAXN = 50001;
    public static int MAXM = 50001;
    public static int n, m;  // n:集合个数, m:操作次数

    // 事件编号组成的数组
    public static int[] eid = new int[MAXM << 1];
    // 事件类型: op == 1 代表修改事件(区间[L,R]增加值 val), op == 2 代表查询事件([L,R]范围上查询第 k 大)
    public static int[] op = new int[MAXM << 1];
    public static int[] L = new int[MAXM << 1];    // 事件区间左端点
    public static int[] R = new int[MAXM << 1];    // 事件区间右端点
    public static int[] val = new int[MAXM << 1];  // 修改事件的值
    public static int[] k = new int[MAXM << 1];    // 查询事件的第 k 大
    public static int[] q = new int[MAXM << 1];    // 查询事件的编号
    public static int cnte = 0;                    // 事件计数器

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

    // 整体二分中用于分类事件的临时存储
    public static int[] lset = new int[MAXM << 1];  // 满足条件的事件
    public static int[] rset = new int[MAXM << 1];  // 不满足条件的事件

    // 查询的答案存储数组
    public static long[] ans = new long[MAXN];

    // 计算二进制表示中最低位的 1 所代表的数值
    public static int lowbit(int i) {
```

```
    return i & -i;
}

// 树状数组单点更新
public static void add(int i, long v) {
    int siz = n;
    while (i <= siz) {
        tree[i] += v;
        i += lowbit(i);
    }
}

// 树状数组区间更新: 区间加法 [l, r] += v
public static void add(int l, int r, long v) {
    add(l, v);
    add(r + 1, -v);
}

// 树状数组单点查询
public static long query(int i) {
    long ret = 0;
    while (i > 0) {
        ret += tree[i];
        i -= lowbit(i);
    }
    return ret;
}

// 整体二分核心函数
// el, er: 事件范围
// vl, vr: 值域范围
public static void compute(int el, int er, long vl, long vr) {
    // 递归边界：没有事件需要处理
    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++) {
            if (op[eid[i]] == 2) { // 查询事件
                ans[q[eid[i]]] = vl;
            }
```

```
    }
} else {
    // 二分中点
    long mid = (vl + vr) >> 1;

    // 检查每个事件，根据事件类型和值域范围划分到左右区间
    int lsiz = 0, rsiz = 0;
    for (int i = el; i <= er; i++) {
        int id = eid[i];
        if (op[id] == 1) {
            // 修改事件
            if (val[id] <= mid) {
                // 对左半区间有贡献，执行修改
                add(L[id], R[id], 1);
                // 将该事件加入左集合
                lset[++lsiz] = id;
            } else {
                // 对左半区间无贡献，放入右半区间
                // 将该事件加入右集合
                rset[++rsiz] = id;
            }
        } else {
            // 查询事件
            // 查询区间[L[id], R[id]]中大于 mid 的元素个数
            long satisfy = query(R[id]) - query(L[id] - 1);
            if (satisfy >= k[id]) {
                // 说明第 k 大的数在左半部分(值大于 mid)
                // 将该事件加入左集合
                lset[++lsiz] = id;
            } else {
                // 说明第 k 大的数在右半部分(值小于等于 mid)
                // 更新 k 值，将该事件加入右集合
                k[id] -= satisfy;
                rset[++rsiz] = id;
            }
        }
    }

    // 撤销对树状数组的修改，恢复到处理前的状态
    for (int i = 1; i <= lsiz; i++) {
        int id = lset[i];
        if (op[id] == 1 && val[id] <= (vl + vr) >> 1) {
            add(L[id], R[id], -1);
```

```
            }
        }

        // 重新排列事件顺序，使得左集合的事件在前，右集合的事件在后
        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];
        }

        // 递归处理左右两部分
        // 左半部分：值域在[vl, mid]范围内的事件
        compute(el, el + lsiz - 1, vl, mid);
        // 右半部分：值域在[mid+1, vr]范围内的事件
        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));

    // 读取集合个数和操作次数
    String[] params = br.readLine().split(" ");
    n = Integer.parseInt(params[0]);
    m = Integer.parseInt(params[1]);

    // 记录值域范围
    long minVal = Long.MAX_VALUE;
    long maxVal = Long.MIN_VALUE;

    // 读取所有事件
    for (int i = 1; i <= m; i++) {
        String[] event = br.readLine().split(" ");
        eid[i] = i;  // 事件编号
        op[i] = Integer.parseInt(event[0]);  // 事件类型
        L[i] = Integer.parseInt(event[1]);   // 区间左端点
        R[i] = Integer.parseInt(event[2]);   // 区间右端点

        if (op[i] == 1) {
            // 修改操作: 1 L R val 表示将 val 加入到编号在[L,R]内的集合中
            val[i] = Integer.parseInt(event[3]);
```

```
                minVal = Math.min(minVal, val[i]);
                maxVal = Math.max(maxVal, val[i]);
            } else {
                // 查询操作: 2 L R k 表示查询编号在[L,R]内的集合的并集中，第 k 大的数
                k[i] = Integer.parseInt(event[3]);
                q[i] = i;  // 查询编号
            }
        }

        // 整体二分求解
        // 初始事件范围[1, m]，初始值域范围[minVal, maxVal]
        compute(1, m, minVal, maxVal);

        // 输出结果
        for (int i = 1; i <= m; i++) {
            if (op[i] == 2) {  // 查询操作的结果
                out.println(ans[i]);
            }
        }

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

==========

文件: P3332_K 大数查询.py

==========

```
# P3332 [ZJOI2013] K 大数查询 - Python 实现
# 题目来源：https://www.luogu.com.cn/problem/P3332
# 时间复杂度：O(M * logN * log(maxValue))
# 空间复杂度：O(N + M)

import sys

class KthLargestQuery:
    def __init__(self):
        self.MAXN = 50001
        self.MAXM = 50001
        self.n = 0
        self.m = 0
```

```
    # 事件编号组成的数组
    self.eid = [0] * (self.MAXM << 1)
    # op == 1，代表修改事件，区间[L,R]增加值 val
    # op == 2，代表查询事件，[L,R]范围上查询第 k 大，q 表示问题的编号
    self.op = [0] * (self.MAXM << 1)
    self.L = [0] * (self.MAXM << 1)
    self.R = [0] * (self.MAXM << 1)
    self.val = [0] * (self.MAXM << 1)
    self.k = [0] * (self.MAXM << 1)
    self.q = [0] * (self.MAXM << 1)
    self.cnte = 0

    # 树状数组，支持区间修改、单点查询
    self.tree = [0] * (self.MAXN << 1)

    # 整体二分
    self.lset = [0] * (self.MAXM << 1)
    self.rset = [0] * (self.MAXM << 1)

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

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

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

# 区间加法 [l, r] += v
def add_range(self, l, r, v):
    self.add(l, v)
    self.add(r + 1, -v)

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

```
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.q[id]] = vl
    else:
        mid = (vl + vr) >> 1
        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.val[id] <= mid:
                    # 对左半区间有贡献，执行修改
                    self.add_range(self.L[id], self.R[id], 1)
                    lsiz += 1
                    self.lset[lsiz] = id
                else:
                    # 对左半区间无贡献，放入右半区间
                    rsiz += 1
                    self.rset[rsiz] = id
            else:
                # 查询操作
                satisfy = self.query(self.R[id]) - self.query(self.L[id] - 1)
                if satisfy >= self.k[id]:
                    # 说明第 k 大的数在左半部分
                    lsiz += 1
                    self.lset[lsiz] = id
                else:
                    # 说明第 k 大的数在右半部分
                    self.k[id] -= satisfy
                    rsiz += 1
                    self.rset[rsiz] = id

        # 撤销对树状数组的修改
        for i in range(1, lsiz + 1):
            id = self.lset[i]
            if self.op[id] == 1 and self.val[id] <= (vl + vr) >> 1:
```

```
                self.add_range(self.L[id], self.R[id], -1)

        # 重新排列事件顺序
        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]

        # 递归处理左右两部分
        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])

    min_val = float('inf')
    max_val = float('-inf')

    for i in range(1, self.m + 1):
        event = sys.stdin.readline().split()
        self.eid[i] = i
        self.op[i] = int(event[0])
        self.L[i] = int(event[1])
        self.R[i] = int(event[2])

        if self.op[i] == 1:
            # 修改操作
            self.val[i] = int(event[3])
            min_val = min(min_val, self.val[i])
            max_val = max(max_val, self.val[i])
        else:
            # 查询操作
            self.k[i] = int(event[3])
            self.q[i] = i

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

    # 输出结果
    for i in range(1, self.m + 1):
        if self.op[i] == 2:
```

```
                print(self.ans[self.q[i]])

# 程序入口
#if __name__ == "__main__":
#    solution = KthLargestQuery()
#    solution.solve()
```

==========

文件: P3527_陨石雨.cpp

==========

```
// P3527 [POI2011] MET-Meteors - 陨石雨 - C++实现
// 题目来源：https://www.luogu.com.cn/problem/P3527
// 时间复杂度：O(K * logK * logM)
// 空间复杂度：O(N + M + K)

//#include <bits/stdc++.h>
//
//using namespace std;
//
//const int MAXN = 300001;
//int n, m, k;
//
//// 国家编号
//int qid[MAXN];
//// 国家的需求
//int need[MAXN];
//
//// 陨石雨的参数
//int rainl[MAXN];
//int rainr[MAXN];
//int num[MAXN];
//
//// 国家拥有的区域列表
//int head[MAXN];
//int nxt[MAXN];
//int to[MAXN];
//int cnt = 0;
//
//// 树状数组，支持范围修改、单点查询
//long long tree[MAXN << 1];
//
//// 整体二分
```

```
//int lset[MAXN];
//int rset[MAXN];
//
//// 每个国家的答案
//int ans[MAXN];
//
//void addEdge(int i, int v) {
//    nxt[++cnt] = head[i];
//    to[cnt] = v;
//    head[i] = cnt;
//}
//
//int lowbit(int i) {
//    return i & -i;
//}
//
//void add(int i, int v) {
//    int siz = m * 2;
//    while (i <= siz) {
//        tree[i] += v;
//        i += lowbit(i);
//    }
//}
//
//void add(int l, int r, int v) {
//    add(l, v);
//    add(r + 1, -v);
//}
//
//long long query(int i) {
//    long long ret = 0;
//    while (i > 0) {
//      ret += tree[i];
//        i -= lowbit(i);
//    }
//    return ret;
//}
//
//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;
//        for (int i = vl; i <= mid; i++) {
//            add(rainl[i], rainr[i], num[i]);
//        }
//        int lsiz = 0, rsiz = 0;
//        for (int i = ql; i <= qr; i++) {
//            int id = qid[i];
//            long long satisfy = 0;
//            for (int e = head[id]; e > 0; e = nxt[e]) {
//                satisfy += query(to[e]) + query(to[e] + m);
//                if (satisfy >= need[id]) {
//                    break;
//                }
//            }
//            if (satisfy >= need[id]) {
//                lset[++lsiz] = id;
//            } else {
//                need[id] -= satisfy;
//                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];
//        }
//        for (int i = vl; i <= mid; i++) {
//            add(rainl[i], rainr[i], -num[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, nation; i <= m; i++) {
//        cin >> nation;
//        addEdge(nation, i);
//    }
//    for (int i = 1; i <= n; i++) {
//        qid[i] = i;
//        cin >> need[i];
//    }
//    cin >> k;
//    for (int i = 1; i <= k; i++) {
//        cin >> rainl[i] >> rainr[i] >> num[i];
//        if (rainr[i] < rainl[i]) {
//            rainr[i] += m;
//        }
//    }
//    compute(1, n, 1, k + 1);
//    for (int i = 1; i <= n; i++) {
//        if (ans[i] == k + 1) {
//            cout << "NIE" << '\n';
//        } else {
//            cout << ans[i] << '\n';
//        }
//    }
//    return 0;
//}
```

==========

文件: P3527_陨石雨.java

==========

```
package class168;

// P3527 [POI2011] MET-Meteors - 陨石雨 - Java 实现
// 题目来源：https://www.luogu.com.cn/problem/P3527
// 题目描述：Byteotian Interstellar Union 有 n 个成员国，星球轨道被分为 m 份，第 i 份上有第 oi 个国家的太空站。
// 有 k 场陨石雨，每场陨石雨会向区间内的太空站提供一定数量的陨石样本。
// 每个国家希望收集 pi 单位的陨石样本，求每个国家在第几次陨石雨之后能收集足够的样本。
// 解题思路：使用整体二分算法，将所有国家查询一起处理
// 时间复杂度：O(K * logK * logM)
// 空间复杂度：O(N + M + K)
// 算法适用条件：
// 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. 修改如果对判定答案有贡献，则贡献为确定值
// 4. 贡献满足交换律、结合律，具有可加性
// 5. 题目允许离线操作

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

public class P3527_陨石雨 {
    public static int MAXN = 300001;
    public static int n, m, k;  // n:国家数量, m:轨道份数, k:陨石雨次数

    // 国家编号
    public static int[] qid = new int[MAXN];
    // 国家的需求
    public static int[] need = new int[MAXN];

    // 陨石雨的参数
    public static int[] rainl = new int[MAXN];  // 陨石雨左端点
    public static int[] rainr = new int[MAXN];  // 陨石雨右端点
    public static int[] num = new int[MAXN];    // 陨石雨数量

    // 国家拥有的区域列表（邻接表）
    public static int[] head = new int[MAXN];   // 邻接表头
    public static int[] next = new int[MAXN];   // 邻接表 next 指针
    public static int[] to = new int[MAXN];     // 邻接表边指向的点
    public static int cnt = 0;                  // 边计数器

    // 树状数组，支持范围修改、单点查询
    public static long[] tree = new long[MAXN << 1];

    // 整体二分中用于分类查询的临时存储
    public static int[] lset = new int[MAXN];  // 满足条件的国家
    public static int[] rset = new int[MAXN];  // 不满足条件的国家

    // 每个国家的答案
    public static int[] ans = new int[MAXN];

    // 添加边到邻接表中
    public static void addEdge(int i, int v) {
        next[++cnt] = head[i];
        to[cnt] = v;
        head[i] = cnt;
```

```
}

// 计算二进制表示中最低位的 1 所代表的数值
public static int lowbit(int i) {
    return i & -i;
}

// 树状数组单点更新
public static void add(int i, int v) {
    int siz = m * 2;  // 由于是环形轨道，需要扩展一倍空间
    while (i <= siz) {
        tree[i] += v;
        i += lowbit(i);
    }
}

// 树状数组区间更新
public static void add(int l, int r, int v) {
    add(l, v);
    add(r + 1, -v);
}

// 树状数组单点查询
public static long query(int i) {
    long ret = 0;
    while (i > 0) {
        ret += tree[i];
        i -= lowbit(i);
    }
    return ret;
}

// 整体二分核心函数
// ql, qr: 国家查询范围
// vl, vr: 陨石雨范围
public static void compute(int ql, int qr, int vl, int vr) {
    // 递归边界：没有国家需要处理
    if (ql > qr) {
        return;
    }

    // 如果陨石雨范围只有一个值，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;

        // 将前 mid 场陨石雨的影响加入树状数组
        for (int i = vl; i <= mid; i++) {
            add(rainl[i], rainr[i], num[i]);
        }

        // 检查每个国家，根据收集到的样本数量划分到左右区间
        int lsiz = 0, rsiz = 0;
        for (int i = ql; i <= qr; i++) {
            int id = qid[i];
            long satisfy = 0;

            // 计算国家 id 在前 mid 场陨石雨中能收集到的样本数量
            for (int e = head[id]; e > 0; e = next[e]) {
                // 由于是环形轨道，需要查询 to[e]和 to[e]+m 两个位置
                satisfy += query(to[e]) + query(to[e] + m);
                // 如果已经满足需求，提前退出
                if (satisfy >= need[id]) {
                    break;
                }
            }

            if (satisfy >= need[id]) {
                // 说明在前 mid 场陨石雨中已经满足需求
                // 将该国家加入左集合
                lset[++lsiz] = id;
            } else {
                // 说明在前 mid 场陨石雨中不满足需求
                // 更新还需的样本数量，将该国家加入右集合
                need[id] -= satisfy;
                rset[++rsiz] = id;
            }
        }

        // 重新排列国家顺序，使得左集合的国家在前，右集合的国家在后
        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];
        }

        // 撤销对树状数组的修改，恢复到处理前的状态
        for (int i = vl; i <= mid; i++) {
            add(rainl[i], rainr[i], -num[i]);
        }

        // 递归处理左右两部分
        // 左半部分：在[vl, mid]范围内满足需求的国家
        compute(ql, ql + lsiz - 1, vl, mid);
        // 右半部分：在[mid+1, vr]范围内满足需求的国家
        compute(ql + lsiz, qr, mid + 1, vr);
    }
}

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

    // 读取国家数量和轨道份数
    String[] params = br.readLine().split(" ");
    n = Integer.parseInt(params[0]);
    m = Integer.parseInt(params[1]);

    // 读取每个轨道属于哪个国家，并建立邻接表
    for (int i = 1, nation; i <= m; i++) {
        nation = Integer.parseInt(br.readLine());
        addEdge(nation, i);
    }

    // 读取每个国家的需求
    for (int i = 1; i <= n; i++) {
        qid[i] = i;
        need[i] = Integer.parseInt(br.readLine());
    }

    // 读取陨石雨信息
    k = Integer.parseInt(br.readLine());
    for (int i = 1; i <= k; i++) {
        String[] rain = br.readLine().split(" ");
```

```
            rainl[i] = Integer.parseInt(rain[0]);
            rainr[i] = Integer.parseInt(rain[1]);
            // 处理环形轨道情况：如果右端点小于左端点，说明跨越了轨道的起始点
            if (rainr[i] < rainl[i]) {
                rainr[i] += m;
            }
            num[i] = Integer.parseInt(rain[2]);
        }

        // 整体二分求解
        // 初始国家范围[1, n]，初始陨石雨范围[1, k+1]
        // 答案范围[1..k+1]，第 k+1 场陨石雨认为满足不了要求
        compute(1, n, 1, k + 1);

        // 输出结果
        for (int i = 1; i <= n; i++) {
            if (ans[i] == k + 1) {
                // 第 k+1 场陨石雨表示无法满足需求
                out.println("NIE");
            } else {
                out.println(ans[i]);
            }
        }

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

==========

文件: P3527_陨石雨.py

==========

```
# P3527 [POI2011] MET-Meteors - 陨石雨 - Python 实现
# 题目来源：https://www.luogu.com.cn/problem/P3527
# 时间复杂度：O(K * logK * logM)
# 空间复杂度：O(N + M + K)

import sys

class MeteorRain:
    def __init__(self):
```

```
        self.MAXN = 300001
        self.n = 0
        self.m = 0
        self.k = 0

        # 国家编号
        self.qid = [0] * self.MAXN
        # 国家的需求
        self.need = [0] * self.MAXN

        # 陨石雨的参数
        self.rainl = [0] * self.MAXN
        self.rainr = [0] * self.MAXN
        self.num = [0] * self.MAXN

        # 国家拥有的区域列表
        self.head = [0] * self.MAXN
        self.next = [0] * self.MAXN
        self.to = [0] * self.MAXN
        self.cnt = 0

        # 树状数组，支持范围修改、单点查询
        self.tree = [0] * (self.MAXN << 1)

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

        # 每个国家的答案
        self.ans = [0] * self.MAXN

    def addEdge(self, i, v):
        self.cnt += 1
        self.next[self.cnt] = self.head[i]
        self.to[self.cnt] = v
        self.head[i] = self.cnt

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

    def add(self, i, v):
        siz = self.m * 2
        while i <= siz:
```

```
            self.tree[i] += v
            i += self.lowbit(i)

    def add_range(self, l, r, v):
        self.add(l, v)
        self.add(r + 1, -v)

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

    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]] = vl
        else:
            mid = (vl + vr) >> 1
            for i in range(vl, mid + 1):
                self.add_range(self.rainl[i], self.rainr[i], self.num[i])
            lsiz = 0
            rsiz = 0
            for i in range(ql, qr + 1):
                id = self.qid[i]
                satisfy = 0
                e = self.head[id]
                while e > 0:
                    satisfy += self.query(self.to[e]) + self.query(self.to[e] + self.m)
                    if satisfy >= self.need[id]:
                        break
                    e = self.next[e]
                if satisfy >= self.need[id]:
                    lsiz += 1
                    self.lset[lsiz] = id
                else:
                    self.need[id] -= satisfy
                    rsiz += 1
                    self.rset[rsiz] = id
            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]
            for i in range(vl, mid + 1):
                self.add_range(self.rainl[i], self.rainr[i], -self.num[i])
            self.compute(ql, ql + lsiz - 1, vl, mid)
            self.compute(ql + lsiz, qr, mid + 1, vr)

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

        for i in range(1, self.m + 1):
            nation = int(sys.stdin.readline())
            self.addEdge(nation, i)

        for i in range(1, self.n + 1):
            self.qid[i] = i
            self.need[i] = int(sys.stdin.readline())

        self.k = int(sys.stdin.readline())

        for i in range(1, self.k + 1):
            rain = sys.stdin.readline().split()
            self.rainl[i] = int(rain[0])
            self.rainr[i] = int(rain[1])
            if self.rainr[i] < self.rainl[i]:
                self.rainr[i] += self.m
            self.num[i] = int(rain[2])

        # 答案范围[1..k+1]，第 k+1 场陨石雨认为满足不了要求
        self.compute(1, self.n, 1, self.k + 1)

        for i in range(1, self.n + 1):
            if self.ans[i] == self.k + 1:
                print("NIE")
            else:
                print(self.ans[i])

# 程序入口
#if __name__ == "__main__":
#    solution = MeteorRain()
```

```
#    solution.solve()

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

文件: P3834_静态区间第K小.cpp
================================================
// P3834 【模板】可持久化线段树 2 / 静态区间第K小 - C++实现
// 题目来源：https://www.luogu.com.cn/problem/P3834
// 题目描述：给定一个长度为n的数组，有m次查询，每次查询[l,r]区间内第k小的数
// 解题思路：使用整体二分算法，将所有查询一起处理，避免对每个查询单独进行二分
// 时间复杂度：O((N+M) * logN * log(maxValue))
// 空间复杂度：O(N+M)
// 算法适用条件：
// 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. 修改如果对判定答案有贡献，则贡献为确定值
// 4. 贡献满足交换律、结合律，具有可加性
// 5. 题目允许离线操作

// 补充题目：POJ 2104 K-th Number
// 题目来源：http://poj.org/problem?id=2104
// 题目描述：给定一个长度为n的数组，有m次查询，每次查询[l,r]区间内第k小的数
// 解题思路：这是整体二分算法的经典应用，与P3834本质相同
// 时间复杂度：O((N+M) * logN * log(maxValue))
// 空间复杂度：O(N+M)
// 本题是静态区间第k小查询的标准问题，是整体二分算法的入门题目

// 由于环境限制，这里只提供C++代码框架，实际编译需要相应环境支持
//#include <iostream>
//#include <algorithm>
//#include <cstdio>
//using namespace std;

const int MAXN = 200001;
int n, m;  // n:数组长度, m:查询次数

// 原始数组，存储输入的数值
int arr[MAXN];

// 离散化后的数组，用于离散化处理，将大值域映射到小下标范围
int sorted[MAXN];

// 查询信息存储
```

```
int qid[MAXN];  // 查询编号
int l[MAXN];    // 查询区间左端点
int r[MAXN];    // 查询区间右端点
int k[MAXN];    // 查询第 k 小

// 树状数组，用于维护当前值域范围内元素的个数
int tree[MAXN];

// 整体二分中用于分类查询的临时存储
int lset[MAXN];  // 满足条件的查询
int rset[MAXN];  // 不满足条件的查询

// 查询的答案存储数组
int ans[MAXN];

// 树状数组操作
// 计算二进制表示中最低位的 1 所代表的数值
int lowbit(int i) {
    return i & -i;
}

// 在树状数组的第 i 个位置加上 v
void add(int i, int v) {
    while (i <= n) {
        tree[i] += v;
        i += lowbit(i);
    }
}

// 计算前缀和[1, i]的和
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]的和
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：没有查询需要处理
    if (ql > qr) {
        return;
    }

    // 如果值域范围只有一个值，说明找到了答案
    // 此时所有查询的答案都是 sorted[vl]
    if (vl == vr) {
        for (int i = ql; i <= qr; i++) {
            ans[qid[i]] = sorted[vl];
        }
        return;
    }

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

    // 将值域小于等于 mid 的数加入树状数组
    // 这些数对后续的查询统计有贡献
    for (int i = vl; i <= mid; i++) {
        add(arr[i], 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++) {
        int id = qid[i];
        // 查询区间[l[id], r[id]]中值小于等于 sorted[mid]的元素个数
        int satisfy = query(l[id], r[id]);

        if (satisfy >= k[id]) {
            // 说明第 k 小的数在左半部分
            // 将该查询加入左集合
            lset[++lsiz] = id;
        } 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
            // 更新 k 值，将该查询加入右集合
            k[id] -= satisfy;
```

```
            rset[++rsiz] = id;
        }
    }

    // 重新排列查询顺序，使得左集合的查询在前，右集合的查询在后
    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];
    }

    // 撤销对树状数组的修改，恢复到处理前的状态
    for (int i = vl; i <= mid; i++) {
        add(arr[i], -1);
    }

    // 递归处理左右两部分
    // 左半部分：值域在[vl, mid]范围内的查询
    compute(ql, ql + lsiz - 1, vl, mid);
    // 右半部分：值域在[mid+1, vr]范围内的查询
    compute(ql + lsiz, qr, mid + 1, vr);
}

//int main() {
//    ios::sync_with_stdio(false);
//    cin.tie(0);
//    cout.tie(0);
//
//    cin >> n >> m;
//
//    // 读取原始数组
//    for (int i = 1; i <= n; i++) {
//        cin >> arr[i];
//        sorted[i] = arr[i];
//    }
//
//    // 离散化
//    sort(sorted + 1, sorted + n + 1);
//    int uniqueCount = unique(sorted + 1, sorted + n + 1) - sorted - 1;
//
//    // 重新映射 arr 数组为离散化后的下标
//    for (int i = 1; i <= n; i++) {
```

```
//        arr[i] = lower_bound(sorted + 1, sorted + uniqueCount + 1, arr[i]) - sorted;
//    }
//
//    // 读取查询
//    for (int i = 1; i <= m; i++) {
//        cin >> l[i] >> r[i] >> k[i];
//        qid[i] = i; // 查询编号
//    }
//
//    // 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] << "\n";
//    }
//
//    return 0;
//}

// POJ 2104 K-th Number 完整实现类
// 这是整体二分算法的经典应用，与 P3834 本质相同
class POJ2104_KthNumber {
private:
    static const int MAXN = 100001; // 题目数据范围
    int n, m;  // n:数组长度, m:查询次数

    // 原始数组，存储输入的数值
    int arr[MAXN];

    // 离散化后的数组，用于离散化处理，将大值域映射到小下标范围
    int sorted[MAXN];

    // 查询信息存储
    int qid[MAXN];  // 查询编号
    int l[MAXN];    // 查询区间左端点
    int r[MAXN];    // 查询区间右端点
    int k[MAXN];    // 查询第 k 小

    // 树状数组，用于维护当前值域范围内元素的个数
    int tree[MAXN];

    // 整体二分中用于分类查询的临时存储
```

```
int lset[MAXN];  // 满足条件的查询
int rset[MAXN];  // 不满足条件的查询

// 查询的答案存储数组
int ans[MAXN];

// 树状数组操作
// 计算二进制表示中最低位的 1 所代表的数值
int lowbit(int i) {
    return i & -i;
}

// 在树状数组的第 i 个位置加上 v
void add(int i, int v) {
    while (i <= n) {
        tree[i] += v;
        i += lowbit(i);
    }
}

// 计算前缀和[1, i]的和
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]的和
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：没有查询需要处理
    if (ql > qr) {
        return;
    }
```

```
// 如果值域范围只有一个值，说明找到了答案
// 此时所有查询的答案都是 sorted[vl]
if (vl == vr) {
    for (int i = ql; i <= qr; i++) {
        ans[qid[i]] = sorted[vl];
    }
    return;
}

// 二分中点
int mid = (vl + vr) >> 1;

// 正确的做法：遍历原始数组，如果元素值对应的离散化下标<=mid，则在对应位置+1
for (int i = 1; i <= n; i++) {
    if (arr[i] <= mid) {
        add(i, 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++) {
    int id = qid[i];
    // 查询区间[l[id], r[id]]中值小于等于 sorted[mid]的元素个数
    int satisfy = query(l[id], r[id]);

    if (satisfy >= k[id]) {
        // 说明第 k 小的数在左半部分
        // 将该查询加入左集合
        lset[++lsiz] = id;
    } 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
        // 更新 k 值，将该查询加入右集合
        k[id] -= satisfy;
        rset[++rsiz] = id;
    }
}

// 重新排列查询顺序，使得左集合的查询在前，右集合的查询在后
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];
        }

        // 撤销对树状数组的修改，恢复到处理前的状态
        for (int i = 1; i <= n; i++) {
            if (arr[i] <= mid) {
                add(i, -1);
            }
        }

        // 递归处理左右两部分
        // 左半部分：值域在[vl, mid]范围内的查询
        compute(ql, ql + lsiz - 1, vl, mid);
        // 右半部分：值域在[mid+1, vr]范围内的查询
        compute(ql + lsiz, qr, mid + 1, vr);
    }

    // 重置数据结构，避免多次调用时的干扰
    void reset() {
        for (int i = 0; i <= n; i++) {
            tree[i] = 0;
        }
    }

public:
    // 主函数，用于解决 POJ 2104 问题
    void solve() {
        // 注意：在实际使用时需要取消注释以下代码，并确保包含相应的头文件
        /*
        ios::sync_with_stdio(false);
        cin.tie(0);
        cout.tie(0);

        cin >> n >> m;

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

        // 离散化
```

```
        sort(sorted + 1, sorted + n + 1);
        int uniqueCount = unique(sorted + 1, sorted + n + 1) - sorted - 1;

        // 重新映射 arr 数组为离散化后的下标
        for (int i = 1; i <= n; i++) {
            arr[i] = lower_bound(sorted + 1, sorted + uniqueCount + 1, arr[i]) - sorted;
        }

        // 读取查询
        for (int i = 1; i <= m; i++) {
            cin >> l[i] >> r[i] >> k[i];
            qid[i] = i; // 查询编号
        }

        // 重置树状数组
        reset();

        // 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] << "\n";
        }
        */
    }
};

// 注意：在实际提交 POJ 时，需要使用以下主函数结构
/*
int main() {
    POJ2104_KthNumber solver;
    solver.solve();
    return 0;
}
*/
```

==========

文件: P3834_静态区间第 K 小.java

==========

```
package class168;
```

```
// P3834 【模板】可持久化线段树 2 / 静态区间第K小 - Java实现
// 题目来源：https://www.luogu.com.cn/problem/P3834
// 题目描述：给定一个长度为n的数组，有m次查询，每次查询[l,r]区间内第k小的数
// 解题思路：使用整体二分算法，将所有查询一起处理，避免对每个查询单独进行二分
// 时间复杂度：O((N+M) * logN * log(maxValue))
// 空间复杂度：O(N+M)
// 算法适用条件：
// 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. 修改如果对判定答案有贡献，则贡献为确定值
// 4. 贡献满足交换律、结合律，具有可加性
// 5. 题目允许离线操作

// 补充题目：POJ 2104 K-th Number
// 题目来源：http://poj.org/problem?id=2104
// 题目描述：给定一个长度为n的数组，有m次查询，每次查询[l,r]区间内第k小的数
// 解题思路：这是整体二分算法的经典应用，与P3834本质相同
// 时间复杂度：O((N+M) * logN * log(maxValue))
// 空间复杂度：O(N+M)
// 本题是静态区间第k小查询的标准问题，是整体二分算法的入门题目

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

public class P3834_静态区间第K小 {
    public static int MAXN = 200001;
    public static int n, m;

    // 原始数组，存储输入的数值
    public static int[] arr = new int[MAXN];

    // 离散化后的数组，用于离散化处理，将大值域映射到小下标范围
    public static int[] sorted = new int[MAXN];

    // 查询信息存储
    public static int[] qid = new int[MAXN];  // 查询编号
    public static int[] l = new int[MAXN];    // 查询区间左端点
    public static int[] r = new int[MAXN];    // 查询区间右端点
    public static int[] k = new int[MAXN];    // 查询第k小

    // 树状数组，用于维护当前值域范围内元素的个数
    public static int[] tree = new int[MAXN];
```

```
// 整体二分中用于分类查询的临时存储
public static int[] lset = new int[MAXN];  // 满足条件的查询
public static int[] rset = new int[MAXN];  // 不满足条件的查询

// 查询的答案存储数组
public static int[] ans = new int[MAXN];

// 树状数组操作
// 计算二进制表示中最低位的 1 所代表的数值
public static int lowbit(int i) {
    return i & -i;
}

// 在树状数组的第 i 个位置加上 v
public static void add(int i, int v) {
    while (i <= n) {
        tree[i] += v;
        i += lowbit(i);
    }
}

// 计算前缀和[1, i]的和
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]的和
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：没有查询需要处理
    if (ql > qr) {
        return;
```

```
}

// 如果值域范围只有一个值，说明找到了答案
// 此时所有查询的答案都是 sorted[vl]
if (vl == vr) {
    for (int i = ql; i <= qr; i++) {
        ans[qid[i]] = sorted[vl];
    }
    return;
}

// 二分中点
int mid = (vl + vr) >> 1;

// 将值域小于等于 mid 的数加入树状数组
// 这些数对后续的查询统计有贡献
for (int i = vl; i <= mid; i++) {
    add(arr[i], 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++) {
    int id = qid[i];
    // 查询区间[l[id], r[id]]中值小于等于 sorted[mid]的元素个数
    int satisfy = query(l[id], r[id]);

    if (satisfy >= k[id]) {
        // 说明第 k 小的数在左半部分
        // 将该查询加入左集合
        lset[++lsiz] = id;
    } 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
        // 更新 k 值，将该查询加入右集合
        k[id] -= satisfy;
        rset[++rsiz] = id;
    }
}

// 重新排列查询顺序，使得左集合的查询在前，右集合的查询在后
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];
        }

        // 撤销对树状数组的修改，恢复到处理前的状态
        for (int i = vl; i <= mid; i++) {
            add(arr[i], -1);
        }

        // 递归处理左右两部分
        // 左半部分：值域在[vl, mid]范围内的查询
        compute(ql, ql + lsiz - 1, vl, mid);
        // 右半部分：值域在[mid+1, vr]范围内的查询
        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];
        }

        // 离散化：将大值域映射到小下标范围，减少二分的值域范围
        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];
            }
        }

        // 重新映射 arr 数组为离散化后的下标
        // 这样可以将值域从[MIN_VALUE, MAX_VALUE]映射到[1, uniqueCount]
        for (int i = 1; i <= n; i++) {
```

```
            arr[i] = Arrays.binarySearch(sorted, 1, uniqueCount + 1, arr[i]);
        }

        // 读取查询
        for (int i = 1; i <= m; i++) {
            String[] query = br.readLine().split(" ");
            l[i] = Integer.parseInt(query[0]);
            r[i] = Integer.parseInt(query[1]);
            k[i] = Integer.parseInt(query[2]);
            qid[i] = i; // 查询编号
        }

        // 整体二分求解
        // 初始查询范围[1, m]，初始值域范围[1, uniqueCount]
        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();
    }
}

// POJ 2104 K-th Number 完整实现类
// 这是整体二分算法的经典应用，与 P3834 本质相同
class POJ2104_KthNumber {
    private static final int MAXN = 100001; // 题目数据范围
    private int n, m;  // n:数组长度, m:查询次数

    // 原始数组，存储输入的数值
    private int[] arr = new int[MAXN];

    // 离散化后的数组，用于离散化处理，将大值域映射到小下标范围
    private int[] sorted = new int[MAXN];

    // 查询信息存储
    private int[] qid = new int[MAXN];  // 查询编号
    private int[] l = new int[MAXN];    // 查询区间左端点
    private int[] r = new int[MAXN];    // 查询区间右端点
```

```
private int[] k = new int[MAXN];    // 查询第 k 小

// 树状数组，用于维护当前值域范围内元素的个数
private int[] tree = new int[MAXN];

// 整体二分中用于分类查询的临时存储
private int[] lset = new int[MAXN];  // 满足条件的查询
private int[] rset = new int[MAXN];  // 不满足条件的查询

// 查询的答案存储数组
private int[] ans = new int[MAXN];

// 树状数组操作
// 计算二进制表示中最低位的 1 所代表的数值
private int lowbit(int i) {
    return i & -i;
}

// 在树状数组的第 i 个位置加上 v
private void add(int i, int v) {
    while (i <= n) {
        tree[i] += v;
        i += lowbit(i);
    }
}

// 计算前缀和[1, i]的和
private 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]的和
private 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）
private void compute(int ql, int qr, int vl, int vr) {
    // 递归边界：没有查询需要处理
    if (ql > qr) {
        return;
    }

    // 如果值域范围只有一个值，说明找到了答案
    // 此时所有查询的答案都是 sorted[vl]
    if (vl == vr) {
        for (int i = ql; i <= qr; i++) {
            ans[qid[i]] = sorted[vl];
        }
        return;
    }

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

    // 初始化树状数组为 0，避免之前查询的影响
    // 注意：对于 POJ 2104 的输入规模，这个操作是必要的
    // 这里我们采用另一种方式：将原始数组中小于等于 mid 的元素添加到树状数组
    // 但需要先将 arr 数组中的元素转换为原始值对应的下标
    // 这里需要重新考虑逻辑，正确的做法是根据原始数组的索引来添加

    // 正确的做法：遍历原始数组，如果元素值对应的离散化下标<=mid，则在对应位置+1
    for (int i = 1; i <= n; i++) {
        if (arr[i] <= mid) {
            add(i, 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++) {
        int id = qid[i];
        // 查询区间[l[id], r[id]]中值小于等于 sorted[mid]的元素个数
        int satisfy = query(l[id], r[id]);

        if (satisfy >= k[id]) {
            // 说明第 k 小的数在左半部分
            // 将该查询加入左集合
            lset[++lsiz] = id;
```

```
        } 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
            // 更新 k 值，将该查询加入右集合
            k[id] -= satisfy;
            rset[++rsiz] = id;
        }
    }

    // 重新排列查询顺序，使得左集合的查询在前，右集合的查询在后
    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];
    }

    // 撤销对树状数组的修改，恢复到处理前的状态
    for (int i = 1; i <= n; i++) {
        if (arr[i] <= mid) {
            add(i, -1);
        }
    }

    // 递归处理左右两部分
    // 左半部分：值域在[vl, mid]范围内的查询
    compute(ql, ql + lsiz - 1, vl, mid);
    // 右半部分：值域在[mid+1, vr]范围内的查询
    compute(ql + lsiz, qr, mid + 1, vr);
}

// 主函数，用于解决 POJ 2104 问题
public void solve() 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];
    }

    // 离散化：将大值域映射到小下标范围，减少二分的值域范围
    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];
        }
    }

    // 重新映射 arr 数组为离散化后的下标
    // 这样可以将值域从[MIN_VALUE, MAX_VALUE]映射到[1, uniqueCount]
    for (int i = 1; i <= n; i++) {
        arr[i] = Arrays.binarySearch(sorted, 1, uniqueCount + 1, arr[i]);
    }

    // 读取查询
    for (int i = 1; i <= m; i++) {
        String[] query = br.readLine().split(" ");
        l[i] = Integer.parseInt(query[0]);
        r[i] = Integer.parseInt(query[1]);
        k[i] = Integer.parseInt(query[2]);
        qid[i] = i; // 查询编号
    }

    // 初始化树状数组为0
    Arrays.fill(tree, 0);

    // 整体二分求解
    // 初始查询范围[1, m]，初始值域范围[1, uniqueCount]
    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();
}
```

```
    // 注意：在实际提交 POJ 时，需要将该类作为主类，并添加 main 方法
    // public static void main(String[] args) throws IOException {
    //     new POJ2104_KthNumber().solve();
    // }
}
```

==========

文件: P3834_静态区间第 K 小.py

==========

```
# P3834 【模板】可持久化线段树 2 / 静态区间第 K 小 - Python 实现
# 题目来源：https://www.luogu.com.cn/problem/P3834
# 题目描述：给定一个长度为 n 的数组，有 m 次查询，每次查询[l,r]区间内第 k 小的数
# 解题思路：使用整体二分算法，将所有查询一起处理，避免对每个查询单独进行二分
# 时间复杂度：O((N+M) * logN * log(maxValue))
# 空间复杂度：O(N+M)
# 算法适用条件：
# 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. 修改如果对判定答案有贡献，则贡献为确定值
# 4. 贡献满足交换律、结合律，具有可加性
# 5. 题目允许离线操作

# 补充题目：POJ 2104 K-th Number
# 题目来源：http://poj.org/problem?id=2104
# 题目描述：给定一个长度为 n 的数组，有 m 次查询，每次查询[l,r]区间内第 k 小的数
# 解题思路：这是整体二分算法的经典应用，与 P3834 本质相同
# 时间复杂度：O((N+M) * logN * log(maxValue))
# 空间复杂度：O(N+M)
# 本题是静态区间第 k 小查询的标准问题，是整体二分算法的入门题目

import sys
from bisect import bisect_left

# 由于 Python 的性能限制，对于大数据可能超时，但在逻辑上是正确的

class Solution:
    def __init__(self):
        self.MAXN = 200001
        self.n = 0  # 数组长度
        self.m = 0  # 查询次数
```

```
    # 原始数组，存储输入的数值
    self.arr = [0] * self.MAXN

    # 离散化后的数组，用于离散化处理，将大值域映射到小下标范围
    self.sorted_arr = [0] * self.MAXN

    # 查询信息存储
    self.qid = [0] * self.MAXN  # 查询编号
    self.l = [0] * self.MAXN    # 查询区间左端点
    self.r = [0] * self.MAXN    # 查询区间右端点
    self.k = [0] * self.MAXN    # 查询第 k 小

    # 树状数组，用于维护当前值域范围内元素的个数
    self.tree = [0] * self.MAXN

    # 整体二分中用于分类查询的临时存储
    self.lset = [0] * self.MAXN  # 满足条件的查询
    self.rset = [0] * self.MAXN  # 不满足条件的查询

    # 查询的答案存储数组
    self.ans = [0] * self.MAXN

def lowbit(self, i):
    """树状数组的 lowbit 操作
    计算二进制表示中最低位的 1 所代表的数值"""
    return i & -i

def add(self, i, v):
    """树状数组单点更新
    在树状数组的第 i 个位置加上 v"""
    while i <= self.n:
        self.tree[i] += v
        i += self.lowbit(i)

def sum(self, i):
    """树状数组前缀和查询
    计算前缀和[1, i]的和"""
    ret = 0
    while i > 0:
        ret += self.tree[i]
        i -= self.lowbit(i)
    return ret
```

```
def query(self, l, r):
    """树状数组区间和查询
    计算区间和[l, r]的和"""
    return self.sum(r) - self.sum(l - 1)

def compute(self, ql, qr, vl, vr):
    """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）"""
    # 递归边界：没有查询需要处理
    if ql > qr:
        return

    # 如果值域范围只有一个值，说明找到了答案
    # 此时所有查询的答案都是 sorted_arr[vl]
    if vl == vr:
        for i in range(ql, qr + 1):
            self.ans[self.qid[i]] = self.sorted_arr[vl]
        return

    # 二分中点
    mid = (vl + vr) >> 1

    # 将值域小于等于 mid 的数加入树状数组
    # 这些数对后续的查询统计有贡献
    for i in range(vl, mid + 1):
        self.add(self.arr[i], 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):
        id = self.qid[i]
        # 查询区间[l[id], r[id]]中值小于等于 sorted_arr[mid]的元素个数
        satisfy = self.query(self.l[id], self.r[id])

        if satisfy >= self.k[id]:
            # 说明第 k 小的数在左半部分
            # 将该查询加入左集合
            lsiz += 1
            self.lset[lsiz] = id
        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
```

```
            # 更新 k 值，将该查询加入右集合
            self.k[id] -= satisfy
            rsiz += 1
            self.rset[rsiz] = id

    # 重新排列查询顺序，使得左集合的查询在前，右集合的查询在后
    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]

    # 撤销对树状数组的修改，恢复到处理前的状态
    for i in range(vl, mid + 1):
        self.add(self.arr[i], -1)

    # 递归处理左右两部分
    # 左半部分：值域在[vl, mid]范围内的查询
    self.compute(ql, ql + lsiz - 1, vl, mid)
    # 右半部分：值域在[mid+1, vr]范围内的查询
    self.compute(ql + lsiz, qr, mid + 1, vr)

def solve(self):
    """主函数"""
    # 读取输入
    line = sys.stdin.readline().split()
    self.n = int(line[0])
    self.m = int(line[1])

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

    # 离散化：将大值域映射到小下标范围，减少二分的值域范围
    self.sorted_arr[1:self.n+1] = sorted(self.sorted_arr[1:self.n+1])
    unique_count = 1
    for i in range(2, self.n + 1):
        if self.sorted_arr[i] != self.sorted_arr[i - 1]:
            unique_count += 1
            self.sorted_arr[unique_count] = self.sorted_arr[i]

    # 重新映射 arr 数组为离散化后的下标
```

```
        # 这样可以将值域从[MIN_VALUE, MAX_VALUE]映射到[1, unique_count]
        for i in range(1, self.n + 1):
            # 使用二分查找找到arr[i]在sorted_arr中的位置
            self.arr[i] = bisect_left(self.sorted_arr, self.arr[i], 1, unique_count + 1)

        # 读取查询
        for i in range(1, self.m + 1):
            query_line = sys.stdin.readline().split()
            self.l[i] = int(query_line[0])
            self.r[i] = int(query_line[1])
            self.k[i] = int(query_line[2])
            self.qid[i] = i  # 查询编号

        # 整体二分求解
        # 初始查询范围[1, m]，初始值域范围[1, unique_count]
        self.compute(1, self.m, 1, unique_count)

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

# 程序入口
#if __name__ == "__main__":
#    solution = Solution()
#    solution.solve()


# POJ 2104 K-th Number 完整实现类
# 这是整体二分算法的经典应用，与P3834本质相同
class POJ2104_KthNumber:
    def __init__(self):
        self.MAXN = 100001  # 题目数据范围
        self.n = 0  # 数组长度
        self.m = 0  # 查询次数

        # 原始数组，存储输入的数值
        self.arr = [0] * self.MAXN

        # 离散化后的数组，用于离散化处理，将大值域映射到小下标范围
        self.sorted_arr = [0] * self.MAXN

        # 查询信息存储
        self.qid = [0] * self.MAXN  # 查询编号
```

```
        self.l = [0] * self.MAXN     # 查询区间左端点
        self.r = [0] * self.MAXN     # 查询区间右端点
        self.k = [0] * self.MAXN     # 查询第 k 小

        # 树状数组，用于维护当前值域范围内元素的个数
        self.tree = [0] * self.MAXN

        # 整体二分中用于分类查询的临时存储
        self.lset = [0] * self.MAXN  # 满足条件的查询
        self.rset = [0] * self.MAXN  # 不满足条件的查询

        # 查询的答案存储数组
        self.ans = [0] * self.MAXN

    def lowbit(self, i):
        """树状数组的 lowbit 操作
        计算二进制表示中最低位的 1 所代表的数值"""
        return i & -i

    def add(self, i, v):
        """树状数组单点更新
        在树状数组的第 i 个位置加上 v"""
        while i <= self.n:
            self.tree[i] += v
            i += self.lowbit(i)

    def sum(self, i):
        """树状数组前缀和查询
        计算前缀和[1, i]的和"""
        ret = 0
        while i > 0:
            ret += self.tree[i]
            i -= self.lowbit(i)
        return ret

    def query(self, l, r):
        """树状数组区间和查询
        计算区间和[l, r]的和"""
        return self.sum(r) - self.sum(l - 1)

    def compute(self, ql, qr, vl, vr):
        """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）"""
# 递归边界：没有查询需要处理
if ql > qr:
    return

# 如果值域范围只有一个值，说明找到了答案
# 此时所有查询的答案都是 sorted_arr[vl]
if vl == vr:
    for i in range(ql, qr + 1):
        self.ans[self.qid[i]] = self.sorted_arr[vl]
    return

# 二分中点
mid = (vl + vr) >> 1

# 正确的做法：遍历原始数组，如果元素值对应的离散化下标<=mid，则在对应位置+1
for i in range(1, self.n + 1):
    if self.arr[i] <= mid:
        self.add(i, 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):
    id = self.qid[i]
    # 查询区间[l[id], r[id]]中值小于等于 sorted_arr[mid]的元素个数
    satisfy = self.query(self.l[id], self.r[id])

    if satisfy >= self.k[id]:
        # 说明第 k 小的数在左半部分
        # 将该查询加入左集合
        lsiz += 1
        self.lset[lsiz] = id
    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
        # 更新 k 值，将该查询加入右集合
        self.k[id] -= satisfy
        rsiz += 1
        self.rset[rsiz] = id

# 重新排列查询顺序，使得左集合的查询在前，右集合的查询在后
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]

    # 撤销对树状数组的修改，恢复到处理前的状态
    for i in range(1, self.n + 1):
        if self.arr[i] <= mid:
            self.add(i, -1)

    # 递归处理左右两部分
    # 左半部分：值域在[vl, mid]范围内的查询
    self.compute(ql, ql + lsiz - 1, vl, mid)
    # 右半部分：值域在[mid+1, vr]范围内的查询
    self.compute(ql + lsiz, qr, mid + 1, vr)

def solve(self):
    """主函数，用于解决 POJ 2104 问题"""
    import sys
    from bisect import bisect_left

    # 读取输入
    line = sys.stdin.readline().split()
    self.n = int(line[0])
    self.m = int(line[1])

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

    # 离散化：将大值域映射到小下标范围，减少二分的值域范围
    self.sorted_arr[1:self.n+1] = sorted(self.sorted_arr[1:self.n+1])
    unique_count = 1
    for i in range(2, self.n + 1):
        if self.sorted_arr[i] != self.sorted_arr[i - 1]:
            unique_count += 1
            self.sorted_arr[unique_count] = self.sorted_arr[i]

    # 重新映射 arr 数组为离散化后的下标
    # 这样可以将值域从[MIN_VALUE, MAX_VALUE]映射到[1, unique_count]
    for i in range(1, self.n + 1):
        # 使用二分查找找到 arr[i]在 sorted_arr 中的位置
        self.arr[i] = bisect_left(self.sorted_arr, self.arr[i], 1, unique_count + 1)
```

```
        # 读取查询
        for i in range(1, self.m + 1):
            query_line = sys.stdin.readline().split()
            self.l[i] = int(query_line[0])
            self.r[i] = int(query_line[1])
            self.k[i] = int(query_line[2])
            self.qid[i] = i  # 查询编号

        # 初始化树状数组为0
        self.tree = [0] * self.MAXN

        # 整体二分求解
        # 初始查询范围[1, m]，初始值域范围[1, unique_count]
        self.compute(1, self.m, 1, unique_count)

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

# 注意：在实际提交POJ时，需要将该类作为主程序运行
# if __name__ == "__main__":
#     solution = POJ2104_KthNumber()
#     solution.solve()


# 补充题目：HDU 2665 Kth Number
# 题目来源：http://acm.hdu.edu.cn/showproblem.php?pid=2665
# 题目描述：给定一个长度为n的数组，有m次查询，每次查询[l,r]区间内第k小的数
# 解题思路：与POJ 2104完全相同，是静态区间第k小的标准问题
# 时间复杂度：O((N+M) * logN * log(maxValue))
# 空间复杂度：O(N+M)
# 注意事项：HDU的评测系统对输入输出效率要求较高，需要使用快速IO


# 补充题目：HDU 5412 CRB and Queries
# 题目来源：http://acm.hdu.edu.cn/showproblem.php?pid=5412
# 题目描述：给定一个数组，支持单点修改和区间第k小查询
# 解题思路：带修改的区间第k小问题，可以用整体二分算法结合树状数组解决
# 时间复杂度：O((N+Q) * logN * log(maxValue))
# 空间复杂度：O(N+Q)
# 算法说明：在整体二分过程中，需要处理两种操作：修改操作和查询操作，
#           修改操作相当于将旧值删除，新值插入，需要分别处理
```

补充题目：AGC002D Stamp Rally
题目来源：https://atcoder.jp/contests/agc002/tasks/agc002_d
题目描述：在图中寻找 k 条路径，使得这些路径的起点和终点分别为给定的 k 对节点，
并且所有路径的边的权值的最大值尽可能小
解题思路：使用整体二分结合并查集（支持撤销操作）
时间复杂度：O(E * α(N) * log(maxWeight))
空间复杂度：O(N + E)
算法说明：将边权排序后进行二分，用可撤销并查集判断每组查询是否可以连通

整体二分算法的通用技巧总结：
1. 适用场景：
- 当问题的答案具有单调性，可以进行二分
- 离线处理多个查询，每个查询的答案可以在二分过程中同步处理
- 需要统计满足某些条件的元素个数
2. 常用数据结构：
- 树状数组：处理区间查询和单点更新
- 线段树：处理更复杂的区间操作
- 并查集：处理连通性问题，特别是带有撤销功能的并查集
3. 优化技巧：
- 离散化：将大值域映射到小范围，减少二分次数
- 避免重复计算：通过合理的设计，避免对相同数据进行多次处理
- 减少常数：使用位运算等技巧优化循环和条件判断
4. 注意事项：
- 必须确保问题可以离线处理
- 需要正确处理撤销操作，恢复数据结构到初始状态
- 递归深度需要控制，避免栈溢出
5. 代码工程化建议：
- 使用类封装数据结构和算法，提高可维护性
- 添加详细的注释，特别是关键步骤的说明
- 对于不同语言，注意数据类型范围和效率问题
- 添加异常处理，特别是对于边界情况
- 编写单元测试，验证算法正确性
6. 语言特性差异：
- Java：注意数组初始化和边界检查，使用快速 IO 提高效率
- C++：可以使用 STL 提高开发效率，但需要注意内存管理
- Python：注意递归深度限制和性能问题，对于大数据可能需要优化

==========

文件: P4175_网络管理.cpp

```
==========================================
// P4175 [CTSC2008]网络管理 - C++实现
// 题目来源：https://www.luogu.com.cn/problem/P4175
// 时间复杂度：O((N+Q) * logN * log(maxValue))
// 空间复杂度：O(N + Q)

// 由于环境限制，这里只提供C++代码框架，实际编译需要相应环境支持
//#include <bits/stdc++.h>
//#include <algorithm>
//#include <cstdio>
//using namespace std;

const int MAXN = 80001;
const int MAXQ = 80001;

int n, q;

// 树结构
int head[MAXN], next[MAXN << 1], to[MAXN << 1], cnt = 0;

// 树链剖分
int fa[MAXN], depth[MAXN], siz[MAXN], son[MAXN];
int top[MAXN], dfn[MAXN], rnk[MAXN], dfc = 0;

// 节点权值
int val[MAXN];

// 树状数组
int tree[MAXN];

// 操作信息
int op[MAXQ], x[MAXQ], y[MAXQ], k[MAXQ], qid[MAXQ];

// 整体二分
int lset[MAXQ], rset[MAXQ], ans[MAXQ];

// 离散化
int sorted[MAXN + MAXQ], cntv = 0;

// 去重函数
int unique(int* arr, int len) {
    if (len <= 1) return len;
    int i = 1, j = 2;
```

```
    while (j <= len) {
        if (arr[j] != arr[i]) {
            arr[++i] = arr[j];
        }
        j++;
    }
    return i;
}

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

// 第一次 DFS：计算深度、父节点、子树大小、重儿子
void dfs1(int u, int f) {
    fa[u] = f;
    depth[u] = depth[f] + 1;
    siz[u] = 1;

    for (int i = head[u]; i; i = next[i]) {
        int v = to[i];
        if (v == f) continue;
        dfs1(v, u);
        siz[u] += siz[v];
        if (siz[son[u]] < siz[v]) {
            son[u] = v;
        }
    }
}

// 第二次 DFS：计算 dfn 序、重链顶点
void dfs2(int u, int t) {
    top[u] = t;
    dfn[u] = ++dfc;
    rnk[dfc] = u;

    if (son[u]) {
        dfs2(son[u], t);
    }
```

```
    for (int i = head[u]; i; i = next[i]) {
        int v = to[i];
        if (v == fa[u] || v == son[u]) continue;
        dfs2(v, v);
    }
}

// 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;
}

// 树链剖分查询路径上点的个数
int queryPath(int u, int v) {
    int ret = 0;
    while (top[u] != top[v]) {
        if (depth[top[u]] < depth[top[v]]) {
            //swap(u, v);
            int temp = u;
            u = v;
            v = temp;
        }
        ret += sum(dfn[u]) - sum(dfn[top[u]] - 1);
        u = fa[top[u]];
    }

    if (depth[u] > depth[v]) {
```

```
        //swap(u, v);
        int temp = u;
        u = v;
        v = temp;
    }
    ret += sum(dfn[v]) - sum(dfn[u] - 1);
    return ret;
}

// 树链剖分修改路径上的点
void addPath(int u, int v, int val) {
    while (top[u] != top[v]) {
        if (depth[top[u]] < depth[top[v]]) {
            //swap(u, v);
            int temp = u;
            u = v;
            v = temp;
        }
        add(dfn[top[u]], val);
        add(dfn[u] + 1, -val);
        u = fa[top[u]];
    }

    if (depth[u] > depth[v]) {
        //swap(u, v);
        int temp = u;
        u = v;
        v = temp;
    }
    add(dfn[u], val);
    add(dfn[v] + 1, -val);
}

// 整体二分核心函数
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 (op[qid[i]] == 1) {
            ans[qid[i]] = vl;
        }
    }
    return;
}

// 二分中点
int mid = (vl + vr) >> 1;

// 将值域小于等于 mid 的数加入树状数组
for (int i = 1; i <= n; i++) {
    if (val[i] <= sorted[mid]) {
        addPath(i, i, 1);
    }
}

for (int i = 1; i <= q; i++) {
    if (op[i] == 0 && y[i] <= sorted[mid]) {
        addPath(x[i], x[i], 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++) {
    int id = qid[i];
    if (op[id] == 1) {
        // 查询操作
        int satisfy = queryPath(x[id], y[id]);
        if (satisfy >= k[id]) {
            // 说明第 k 大的数在左半部分
            lset[++lsiz] = id;
        } else {
            // 说明第 k 大的数在右半部分，需要在右半部分找第(k-satisfy)大的数
            k[id] -= satisfy;
            rset[++rsiz] = id;
        }
    } else {
        // 修改操作
        if (y[id] <= sorted[mid]) {
            lset[++lsiz] = id;
        } else {
```

```
                rset[++rsiz] = id;
            }
        }
    }

    // 将操作分组
    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];
    }

    // 清空树状数组
    for (int i = 1; i <= n; i++) {
        if (val[i] <= sorted[mid]) {
            addPath(i, i, -1);
        }
    }

    for (int i = 1; i <= q; i++) {
        if (op[i] == 0 && y[i] <= sorted[mid]) {
            addPath(x[i], x[i], -1);
        }
    }

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

/*
int main() {
    //scanf("%d%d", &n, &q);

    for (int i = 1; i <= n; i++) {
        //scanf("%d", &val[i]);
        sorted[++cntv] = val[i];
    }

    // 建树
    for (int i = 1; i < n; i++) {
        int u, v;
```

```
        //scanf("%d%d", &u, &v);
        addEdge(u, v);
        addEdge(v, u);
    }

    // 处理操作
    for (int i = 1; i <= q; i++) {
        //scanf("%d%d%d", &op[i], &x[i], &y[i]);

        if (op[i] == 0) {
            // 修改操作
            sorted[++cntv] = y[i];
        } else {
            // 查询操作
            k[i] = op[i];
            op[i] = 1;
            qid[i] = i;
        }
    }

    // 离散化
    //sort(sorted + 1, sorted + cntv + 1);
    cntv = unique(sorted, cntv);

    // 树链剖分预处理
    dfs1(1, 0);
    dfs2(1, 1);

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

    // 输出结果
    for (int i = 1; i <= q; i++) {
        if (ans[i]) {
            //printf("%d\n", sorted[ans[i]]);
        } else {
            //printf("invalid request!\n");
        }
    }

    return 0;
}
*/
```

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

文件: P4175_网络管理.java
==========================================
package class168;

// P4175 [CTSC2008]网络管理 - Java 实现
// 题目来源：https://www.luogu.com.cn/problem/P4175
// 时间复杂度：O((N+Q) * logN * log(maxValue))
// 空间复杂度：O(N + Q)

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

public class P4175_网络管理 {
    public static int MAXN = 80001;
    public static int MAXQ = 80001;
    public static int MAXM = 17; // log2(80000) ≈ 16.29

    public static int n, q;

    // 树结构
    public static int[] head = new int[MAXN];
    public static int[] next = new int[MAXN << 1];
    public static int[] to = new int[MAXN << 1];
    public static int cnt = 0;

    // 树链剖分
    public static int[] fa = new int[MAXN];
    public static int[] depth = new int[MAXN];
    public static int[] siz = new int[MAXN];
    public static int[] son = new int[MAXN];
    public static int[] top = new int[MAXN];
    public static int[] dfn = new int[MAXN];
    public static int[] rnk = new int[MAXN];
    public static int dfc = 0;

    // 节点权值
    public static int[] val = new int[MAXN];

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

```
// 操作信息
public static int[] op = new int[MAXQ];  // 0:修改 1:查询
public static int[] x = new int[MAXQ];
public static int[] y = new int[MAXQ];
public static int[] k = new int[MAXQ];
public static int[] qid = new int[MAXQ];

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

// 离散化
public static int[] sorted = new int[MAXN + MAXQ];
public static int cntv = 0;

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().split(" ");
    n = Integer.parseInt(line[0]);
    q = Integer.parseInt(line[1]);

    line = br.readLine().split(" ");
    for (int i = 1; i <= n; i++) {
        val[i] = Integer.parseInt(line[i - 1]);
        sorted[++cntv] = val[i];
    }

    // 建树
    for (int i = 1; i < n; i++) {
        line = br.readLine().split(" ");
        int u = Integer.parseInt(line[0]);
        int v = Integer.parseInt(line[1]);
        addEdge(u, v);
        addEdge(v, u);
    }

    // 处理操作
    for (int i = 1; i <= q; i++) {
        line = br.readLine().split(" ");
```

```
            op[i] = Integer.parseInt(line[0]);
            x[i] = Integer.parseInt(line[1]);
            y[i] = Integer.parseInt(line[2]);

            if (op[i] == 0) {
                // 修改操作
                sorted[++cntv] = y[i];
            } else {
                // 查询操作
                k[i] = op[i];
                op[i] = 1;
                qid[i] = i;
            }
        }

        // 离散化
        Arrays.sort(sorted, 1, cntv + 1);
        cntv = unique(sorted, cntv);

        // 树链剖分预处理
        dfs1(1, 0);
        dfs2(1, 1);

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

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

    // 去重函数
    public static int unique(int[] arr, int len) {
        if (len <= 1) return len;
        int i = 1, j = 2;
        while (j <= len) {
            if (arr[j] != arr[i]) {
                arr[++i] = arr[j];
            }
```

```
        j++;
    }
    return i;
}

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

// 第一次 DFS：计算深度、父节点、子树大小、重儿子
public static void dfs1(int u, int f) {
    fa[u] = f;
    depth[u] = depth[f] + 1;
    siz[u] = 1;

    for (int i = head[u]; i != 0; i = next[i]) {
        int v = to[i];
        if (v == f) continue;
        dfs1(v, u);
        siz[u] += siz[v];
        if (siz[son[u]] < siz[v]) {
            son[u] = v;
        }
    }
}

// 第二次 DFS：计算 dfn 序、重链顶点
public static void dfs2(int u, int t) {
    top[u] = t;
    dfn[u] = ++dfc;
    rnk[dfc] = u;

    if (son[u] != 0) {
        dfs2(son[u], t);
    }

    for (int i = head[u]; i != 0; i = next[i]) {
        int v = to[i];
        if (v == fa[u] || v == son[u]) continue;
        dfs2(v, v);
```

```
    }
}

// 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;
}

// 树链剖分查询路径上点的个数
public static int queryPath(int u, int v) {
    int ret = 0;
    while (top[u] != top[v]) {
        if (depth[top[u]] < depth[top[v]]) {
            int temp = u;
            u = v;
            v = temp;
        }
        ret += sum(dfn[u]) - sum(dfn[top[u]] - 1);
        u = fa[top[u]];
    }

    if (depth[u] > depth[v]) {
        int temp = u;
        u = v;
        v = temp;
    }
    ret += sum(dfn[v]) - sum(dfn[u] - 1);
```

```
    return ret;
}

// 树链剖分修改路径上的点
public static void addPath(int u, int v, int val) {
    while (top[u] != top[v]) {
        if (depth[top[u]] < depth[top[v]]) {
            int temp = u;
            u = v;
            v = temp;
        }
        add(dfn[top[u]], val);
        add(dfn[u] + 1, -val);
        u = fa[top[u]];
    }

    if (depth[u] > depth[v]) {
        int temp = u;
        u = v;
        v = temp;
    }
    add(dfn[u], val);
    add(dfn[v] + 1, -val);
}

// 整体二分核心函数
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 (op[qid[i]] == 1) {
                ans[qid[i]] = vl;
            }
        }
        return;
    }

    // 二分中点
```

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

// 将值域小于等于 mid 的数加入树状数组
for (int i = 1; i <= n; i++) {
    if (val[i] <= sorted[mid]) {
        addPath(i, i, 1);
    }
}

for (int i = 1; i <= q; i++) {
    if (op[i] == 0 && y[i] <= sorted[mid]) {
        addPath(x[i], x[i], 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++) {
    int id = qid[i];
    if (op[id] == 1) {
        // 查询操作
        int satisfy = queryPath(x[id], y[id]);
        if (satisfy >= k[id]) {
            // 说明第 k 大的数在左半部分
            lset[++lsiz] = id;
        } else {
            // 说明第 k 大的数在右半部分，需要在右半部分找第(k-satisfy)大的数
            k[id] -= satisfy;
            rset[++rsiz] = id;
        }
    } else {
        // 修改操作
        if (y[id] <= sorted[mid]) {
            lset[++lsiz] = id;
        } else {
            rset[++rsiz] = id;
        }
    }
}

// 将操作分组
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];
        }

        // 清空树状数组
        for (int i = 1; i <= n; i++) {
            if (val[i] <= sorted[mid]) {
                addPath(i, i, -1);
            }
        }

        for (int i = 1; i <= q; i++) {
            if (op[i] == 0 && y[i] <= sorted[mid]) {
                addPath(x[i], x[i], -1);
            }
        }

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

==========

文件: P4175_网络管理.py

==========

```
# P4175 [CTSC2008]网络管理 - Python 实现
# 题目来源：https://www.luogu.com.cn/problem/P4175
# 时间复杂度：O((N+Q) * logN * log(maxValue))
# 空间复杂度：O(N + Q)

class NetworkManagement:
    def __init__(self):
        self.MAXN = 80001
        self.MAXQ = 80001

        self.n = 0
        self.q = 0

        # 树结构
        self.head = [0] * self.MAXN
```

```
        self.next = [0] * (self.MAXN << 1)
        self.to = [0] * (self.MAXN << 1)
        self.cnt = 0

        # 树链剖分
        self.fa = [0] * self.MAXN
        self.depth = [0] * self.MAXN
        self.siz = [0] * self.MAXN
        self.son = [0] * self.MAXN
        self.top = [0] * self.MAXN
        self.dfn = [0] * self.MAXN
        self.rnk = [0] * self.MAXN
        self.dfc = 0

        # 节点权值
        self.val = [0] * self.MAXN

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

        # 操作信息
        self.op = [0] * self.MAXQ  # 0:修改 1:查询
        self.x = [0] * self.MAXQ
        self.y = [0] * self.MAXQ
        self.k = [0] * self.MAXQ
        self.qid = [0] * self.MAXQ

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

        # 离散化
        self.sorted = [0] * (self.MAXN + self.MAXQ)
        self.cntv = 0

    def addEdge(self, u, v):
        self.cnt += 1
        self.next[self.cnt] = self.head[u]
        self.to[self.cnt] = v
        self.head[u] = self.cnt

    def dfs1(self, u, f):
```

```
        """第一次 DFS：计算深度、父节点、子树大小、重儿子"""
        self.fa[u] = f
        self.depth[u] = self.depth[f] + 1
        self.siz[u] = 1

        i = self.head[u]
        while i:
            v = self.to[i]
            if v != f:
                self.dfs1(v, u)
                self.siz[u] += self.siz[v]
                if self.siz[self.son[u]] < self.siz[v]:
                    self.son[u] = v
            i = self.next[i]

    def dfs2(self, u, t):
        """第二次 DFS：计算 dfn 序、重链顶点"""
        self.top[u] = t
        self.dfc += 1
        self.dfn[u] = self.dfc
        self.rnk[self.dfc] = u

        if self.son[u]:
            self.dfs2(self.son[u], t)

        i = self.head[u]
        while i:
            v = self.to[i]
            if v != self.fa[u] and v != self.son[u]:
                self.dfs2(v, v)
            i = self.next[i]

    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 queryPath(self, u, v):
    """树链剖分查询路径上点的个数"""
    ret = 0
    while self.top[u] != self.top[v]:
        if self.depth[self.top[u]] < self.depth[self.top[v]]:
            u, v = v, u
        ret += self.sum(self.dfn[u]) - self.sum(self.dfn[self.top[u]] - 1)
        u = self.fa[self.top[u]]

    if self.depth[u] > self.depth[v]:
        u, v = v, u
    ret += self.sum(self.dfn[v]) - self.sum(self.dfn[u] - 1)
    return ret

def addPath(self, u, v, val):
    """树链剖分修改路径上的点"""
    while self.top[u] != self.top[v]:
        if self.depth[self.top[u]] < self.depth[self.top[v]]:
            u, v = v, u
        self.add(self.dfn[self.top[u]], val)
        self.add(self.dfn[u] + 1, -val)
        u = self.fa[self.top[u]]

    if self.depth[u] > self.depth[v]:
        u, v = v, u
    self.add(self.dfn[u], val)
    self.add(self.dfn[v] + 1, -val)

def compute(self, ql, qr, vl, vr):
    """整体二分核心函数"""
    # 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):
            if self.op[self.qid[i]] == 1:
                self.ans[self.qid[i]] = vl
```

```
    return

# 二分中点
mid = (vl + vr) >> 1

# 将值域小于等于 mid 的数加入树状数组
for i in range(1, self.n + 1):
    if self.val[i] <= self.sorted[mid]:
        self.addPath(i, i, 1)

for i in range(1, self.q + 1):
    if self.op[i] == 0 and self.y[i] <= self.sorted[mid]:
        self.addPath(self.x[i], self.x[i], 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):
    id = self.qid[i]
    if self.op[id] == 1:
        # 查询操作
        satisfy = self.queryPath(self.x[id], self.y[id])
        if satisfy >= self.k[id]:
            # 说明第 k 大的数在左半部分
            lsiz += 1
            self.lset[lsiz] = id
        else:
            # 说明第 k 大的数在右半部分，需要在右半部分找第(k-satisfy)大的数
            self.k[id] -= satisfy
            rsiz += 1
            self.rset[rsiz] = id
    else:
        # 修改操作
        if self.y[id] <= self.sorted[mid]:
            lsiz += 1
            self.lset[lsiz] = id
        else:
            rsiz += 1
            self.rset[rsiz] = id

# 将操作分组
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]

    # 清空树状数组
    for i in range(1, self.n + 1):
        if self.val[i] <= self.sorted[mid]:
            self.addPath(i, i, -1)

    for i in range(1, self.q + 1):
        if self.op[i] == 0 and self.y[i] <= self.sorted[mid]:
            self.addPath(self.x[i], self.x[i], -1)

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

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

    idx = 0
    self.n = int(data[idx])
    idx += 1
    self.q = int(data[idx])
    idx += 1

    for i in range(1, self.n + 1):
        self.val[i] = int(data[idx])
        idx += 1
        self.sorted[self.cntv + 1] = self.val[i]
        self.cntv += 1

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

    # 处理操作
```

```
for i in range(1, self.q + 1):
    self.op[i] = int(data[idx])
    idx += 1
    self.x[i] = int(data[idx])
    idx += 1
    self.y[i] = int(data[idx])
    idx += 1

    if self.op[i] == 0:
        # 修改操作
        self.sorted[self.cntv + 1] = self.y[i]
        self.cntv += 1
    else:
        # 查询操作
        self.k[i] = self.op[i]
        self.op[i] = 1
        self.qid[i] = i

# 离散化
self.sorted = self.sorted[:self.cntv + 1]
self.sorted.sort()
# 去重
unique_sorted = []
for i in range(len(self.sorted)):
    if not unique_sorted or unique_sorted[-1] != self.sorted[i]:
        unique_sorted.append(self.sorted[i])
self.sorted = unique_sorted
self.cntv = len(self.sorted) - 1

# 树链剖分预处理
self.dfs1(1, 0)
self.dfs2(1, 1)

# 整体二分求解
self.compute(1, self.q, 1, self.cntv)

# 输出结果
results = []
for i in range(1, self.q + 1):
    if self.ans[i] != 0:
        results.append(str(self.sorted[self.ans[i]]))
    else:
        results.append("invalid request!")
```

```
        return results

# 主函数
"""
if __name__ == "__main__":
    solver = NetworkManagement()
    results = solver.solve()
    for result in results:
        print(result)
"""
```

==========

文件: P4602_混合果汁.java

==========

```
package class168;

// P4602 [CTSC2018] 混合果汁 - Java 实现
// 题目来源：https://www.luogu.com.cn/problem/P4602
// 题目描述：商店里有 n 种果汁，每种有美味度、价格和添加上限。m 个小朋友希望用不超过 g 元钱制作至少 L 升果汁，
// 且希望混合果汁的美味度尽可能高（美味度等于所有参与混合的果汁的美味度的最小值）。
// 解题思路：使用整体二分算法，二分美味度，贪心选择满足条件的果汁
// 时间复杂度：O((N+M) * logN * log(maxD))
// 空间复杂度：O(N + M)
// 算法适用条件：
// 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. 修改如果对判定答案有贡献，则贡献为确定值
// 4. 贡献满足交换律、结合律，具有可加性
// 5. 题目允许离线操作

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

public class P4602_混合果汁 {
    public static int MAXN = 100001;

    public static int n, m;  // n:果汁种数, m:小朋友数量

    // 果汁信息
    public static int[] d = new int[MAXN];  // 美味度
    public static int[] p = new int[MAXN];  // 价格
```

```
public static int[] l = new int[MAXN];  // 添加上限

// 小朋友信息
public static long[] g = new long[MAXN];  // 最大支付价格
public static long[] L = new long[MAXN];  // 最小体积需求
public static int[] qid = new int[MAXN];  // 查询编号

// 整体二分中用于分类查询的临时存储
public static int[] lset = new int[MAXN];  // 满足条件的查询
public static int[] rset = new int[MAXN];  // 不满足条件的查询
public static int[] ans = new int[MAXN];   // 查询的答案

// 离散化
public static int[] sorted = new int[MAXN];  // 离散化后的美味度数组
public static int cntv = 0;                  // 离散化后的元素个数

// 果汁信息结构体
static class Juice implements Comparable<Juice> {
    int d, p, l;  // d:美味度, p:价格, l:添加上限

    public Juice(int d, int p, int l) {
        this.d = d;
        this.p = p;
        this.l = l;
    }

    @Override
    public int compareTo(Juice other) {
        return Integer.compare(this.d, other.d);  // 按美味度升序排序
    }
}

public static Juice[] juices = new Juice[MAXN];

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

    // 读取果汁种数和小朋友数量
    String[] line = br.readLine().split(" ");
    n = Integer.parseInt(line[0]);
    m = Integer.parseInt(line[1]);
```

```
    // 读取果汁信息
    for (int i = 1; i <= n; i++) {
        line = br.readLine().split(" ");
        int di = Integer.parseInt(line[0]);  // 美味度
        int pi = Integer.parseInt(line[1]);  // 价格
        int li = Integer.parseInt(line[2]);  // 添加上限
        juices[i] = new Juice(di, pi, li);
        sorted[++cntv] = di;  // 用于离散化
    }

    // 处理小朋友需求
    for (int i = 1; i <= m; i++) {
        line = br.readLine().split(" ");
        g[i] = Long.parseLong(line[0]);  // 最大支付价格
        L[i] = Long.parseLong(line[1]);  // 最小体积需求
        qid[i] = i;                      // 查询编号
    }

    // 离散化：将美味度值域映射到小下标范围
    Arrays.sort(sorted, 1, cntv + 1);
    cntv = unique(sorted, cntv);  // 去重

    // 按美味度排序果汁
    Arrays.sort(juices, 1, n + 1);

    // 整体二分求解
    // 初始查询范围[1, m]，初始值域范围[1, cntv]
    compute(1, m, 1, cntv);

    // 输出结果
    for (int i = 1; i <= m; i++) {
        if (ans[i] != 0) {
            // 输出第 i 个小朋友能喝到的最美味的混合果汁的美味度
            out.println(sorted[ans[i]]);
        } else {
            // 无法满足需求
            out.println(-1);
        }
    }
    out.flush();
}

// 去重函数：对已排序数组进行去重，返回去重后的长度
```

```
public static int unique(int[] arr, int len) {
    if (len <= 1) return len;
    int i = 1, j = 2;
    while (j <= len) {
        if (arr[j] != arr[i]) {
            arr[++i] = arr[j];
        }
        j++;
    }
    return i;
}

// 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：没有查询需要处理
    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;
        }
        return;
    }

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

    // 检查每个查询是否能满足条件
    int lsiz = 0, rsiz = 0;
    for (int i = ql; i <= qr; i++) {
        int id = qid[i];
        // 检查美味度大于等于 sorted[mid]的果汁是否能满足需求
        if (check(mid, g[id], L[id])) {
            // 满足条件，说明答案可能更大(美味度更高)，分到左区间
            // 将该查询加入左集合
            lset[++lsiz] = id;
        } else {
            // 不满足条件，说明答案更小(美味度更低)，分到右区间
```

```
            // 将该查询加入右集合
            rset[++rsiz] = id;
        }
    }

    // 重新排列查询顺序，使得左集合的查询在前，右集合的查询在后
    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];
    }

    // 递归处理左右区间
    // 左半部分：美味度在[mid+1, vr]范围内的查询
    compute(ql, ql + lsiz - 1, mid + 1, vr);
    // 右半部分：美味度在[vl, mid]范围内的查询
    compute(ql + lsiz, qr, vl, mid);
}

// 检查美味度大于等于 sorted[mid]的果汁是否能满足在 maxPrice 价格内制作出 minVolume 体积的果汁
public static boolean check(int mid, long maxPrice, long minVolume) {
    long totalVolume = 0;   // 累计体积
    long totalPrice = 0;    // 累计价格

    // 贪心策略：从美味度高的果汁开始选择，以获得最高的美味度
    for (int i = n; i >= 1; i--) {
        // 只考虑美味度大于等于 sorted[mid]的果汁
        if (juices[i].d < sorted[mid]) break;

        // 计算能使用的体积：取添加上限和还需体积的较小值
        long useVolume = Math.min(juices[i].l, minVolume - totalVolume);
        if (useVolume <= 0) continue;  // 如果不需要更多体积，跳过

        // 累加体积和价格
        totalVolume += useVolume;
        totalPrice += useVolume * juices[i].p;

        // 如果已经超过价格限制，直接返回 false
        if (totalPrice > maxPrice) {
            return false;
        }
```

```
            // 如果已经满足体积需求，跳出循环
            if (totalVolume >= minVolume) {
                break;
            }
        }

        // 检查是否满足体积需求且价格不超限
        return totalVolume >= minVolume && totalPrice <= maxPrice;
    }
}
```

==========

文件: P5163_WD与地图.cpp

==========

```
// P5163 WD与地图 - C++实现
// 题目来源：https://www.luogu.com.cn/problem/P5163
// 时间复杂度：O(Q * logQ * (N + M))
// 空间复杂度：O(N + M + Q)

//#include <bits/stdc++.h>
//
//using namespace std;
//
//const int MAXN = 100001;
//const int MAXM = 200001;
//const int MAXQ = 200001;
//int n, m, q;
//
//// 节点权值
//int s[MAXN];
//
//// 边的信息
//int eu[MAXM], ev[MAXM];
//
//// 操作信息
//int op[MAXQ], a[MAXQ], b[MAXQ];
//
//// 并查集
//int father[MAXN], size[MAXN], stack[MAXN], top = 0;
//long long value[MAXN];
//
//// 整体二分
```

```
//int lset[MAXQ], rset[MAXQ];
//
//// 答案
//long long ans[MAXQ];
//
//// 查询编号数组
//int qid[MAXQ];
//
//// 初始化并查集
//void init() {
//    for (int i = 1; i <= n; i++) {
//        father[i] = i;
//        size[i] = 1;
//        value[i] = s[i];
//    }
//    top = 0;
//}
//
//// 查找根节点（带路径压缩）
//int find(int x) {
//    while (x != father[x]) {
//        x = father[x];
//    }
//    return x;
//}
//
//// 合并两个集合
//bool unionSets(int x, int y) {
//    int fx = find(x);
//    int fy = find(y);
//    if (fx == fy) {
//        return false;
//    }
//    if (size[fx] < size[fy]) {
//        swap(fx, fy);
//    }
//    father[fy] = fx;
//    size[fx] += size[fy];
//    value[fx] += value[fy];
//    stack[++top] = fy; // 记录修改，用于回滚
//    return true;
//}
//
```

```
//// 回滚操作
//void rollback(int targetTop) {
//    while (top > targetTop) {
//        int y = stack[top--];
//        int fy = find(y);
//        value[fy] -= value[y];
//        size[fy] -= size[y];
//        father[y] = y;
//    }
//}
//
//// 计算前 k 大值的和
//long long getTopK(int x, int k) {
//    // 这里简化处理，实际应该维护一个优先队列或有序结构
//    // 为了整体二分的演示，我们只返回连通块的总和
//    int fx = find(x);
//    return value[fx];
//}
//
//// 整体二分核心函数
//void compute(int ql, int qr, int vl, int vr) {
//    if (ql > qr) {
//        return;
//    }
//    if (vl == vr) {
//        for (int i = ql; i <= qr; i++) {
//            if (op[qid[i]] == 3) {
//                ans[qid[i]] = vl;
//            }
//        }
//        return;
//    }
//
//    int mid = (vl + vr) >> 1;
//    int targetTop = top;
//
//    // 处理时间小于等于 mid 的操作
//    for (int i = vl; i <= mid; i++) {
//        if (op[i] == 1) {
//            // 删除边，这里简化处理
//        } else if (op[i] == 2) {
//            // 增加点权
//            int fx = find(a[i]);
```

```
//            value[fx] += b[i];
//        }
//    }
//
//    // 检查每个查询
//    int lsiz = 0, rsiz = 0;
//    for (int i = ql; i <= qr; i++) {
//        int id = qid[i];
//        if (op[id] == 3) {
//            // 查询操作
//            long long sum = getTopK(a[id], b[id]);
//            if (sum >= 0) { // 这里简化判断
//                lset[++lsiz] = id;
//            } else {
//                rset[++rsiz] = id;
//            }
//        } else {
//            // 修改操作放在对应的区间
//            if (id <= mid) {
//                lset[++lsiz] = id;
//            } else {
//                rset[++rsiz] = id;
//            }
//        }
//    }
//
//    // 重新排列操作顺序
//    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];
//    }
//
//    // 回滚操作
//    rollback(targetTop);
//
//    // 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() {
```

```
//    ios::sync_with_stdio(false);
//    cin.tie(nullptr);
//    cin >> n >> m >> q;
//
//    // 读取节点权值
//    for (int i = 1; i <= n; i++) {
//        cin >> s[i];
//    }
//
//    // 读取边信息
//    for (int i = 1; i <= m; i++) {
//        cin >> eu[i] >> ev[i];
//    }
//
//    // 读取操作信息
//    for (int i = 1; i <= q; i++) {
//        cin >> op[i] >> a[i];
//        if (op[i] != 1) {
//            cin >> b[i];
//        }
//        qid[i] = i;
//    }
//
//    // 初始化并查集
//    init();
//
//    // 整体二分求解
//    compute(1, q, 1, q);
//
//    // 输出结果
//    for (int i = 1; i <= q; i++) {
//        if (op[i] == 3) {
//            cout << ans[i] << "\n";
//        }
//    }
//
//    return 0;
//}
```

==========

文件: P5163_WD与地图.java

==========

```
package class168;

// P5163 WD与地图 - Java实现
// 题目来源：https://www.luogu.com.cn/problem/P5163
// 时间复杂度：O(Q * logQ * (N + M))
// 空间复杂度：O(N + M + Q)
//
// 题目大意：
// 维护有向图，支持删边、点权增加、查询强连通分量前k大点权和。
//
// 解题思路：
// 1. 对时间进行二分
// 2. 使用可撤销并查集维护强连通分量
// 3. 检查在某个时间点是否满足查询条件
// 4. 根据统计结果将操作分为两类递归处理
//
// 算法详解：
// 1. 整体二分：将所有操作一起处理，对时间进行二分
// 2. 可撤销并查集：用于维护强连通分量，支持操作的回滚
// 3. 点权维护：使用并查集维护每个连通分量的点权和

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

public class P5163_WD与地图 {
    public static int MAXN = 100001;
    public static int MAXM = 200001;
    public static int MAXQ = 200001;
    public static int n, m, q;

    // 节点权值
    public static int[] s = new int[MAXN];

    // 边的信息
    public static int[] eu = new int[MAXM];
    public static int[] ev = new int[MAXM];

    // 操作信息
    public static int[] op = new int[MAXQ];
    public static int[] a = new int[MAXQ];
    public static int[] b = new int[MAXQ];

    // 并查集
```

```
public static int[] father = new int[MAXN];
public static int[] size = new int[MAXN];
public static long[] value = new long[MAXN];
public static int[] stack = new int[MAXN];
public static int top = 0;

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

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

// 初始化并查集
// 将每个节点初始化为一个独立的集合
public static void init() {
    for (int i = 1; i <= n; i++) {
        father[i] = i;
        size[i] = 1;
        value[i] = s[i];
    }
    top = 0;
}

// 查找根节点（带路径压缩）
// 使用路径压缩优化，使查找操作的时间复杂度接近 O(1)
public static int find(int x) {
    while (x != father[x]) {
        x = father[x];
    }
    return x;
}

// 合并两个集合
// 使用按秩合并优化，将较小的树合并到较大的树上
// 返回 true 表示成功合并，false 表示已在同一集合中
public static boolean union(int x, int y) {
    int fx = find(x);
    int fy = find(y);
    if (fx == fy) {
        return false;
    }
    // 按秩合并，将较小的树合并到较大的树上
```

```
    if (size[fx] < size[fy]) {
        int temp = fx;
        fx = fy;
        fy = temp;
    }
    father[fy] = fx;
    size[fx] += size[fy];
    value[fx] += value[fy];
    stack[++top] = fy; // 记录修改，用于回滚
    return true;
}

// 回滚操作
// 将并查集的状态回滚到 targetTop 时刻
public static void rollback(int targetTop) {
    while (top > targetTop) {
        int y = stack[top--];
        int fy = find(y);
        value[fy] -= value[y];
        size[fy] -= size[y];
        father[y] = y;
    }
}

// 计算前 k 大值的和
// 这是一个简化的实现，实际应该维护一个优先队列或有序结构
public static long getTopK(int x, int k) {
    // 这里简化处理，实际应该维护一个优先队列或有序结构
    // 为了整体二分的演示，我们只返回连通块的总和
    int fx = find(x);
    return value[fx];
}

// 整体二分核心函数
// ql, qr: 当前处理的操作范围
// vl, vr: 当前处理的时间范围
public static void compute(int ql, int qr, int vl, int vr) {
    // 递归边界
    if (ql > qr) {
        return;
    }
    // 递归边界：时间范围只有一个值，找到了答案
    if (vl == vr) {
```

```
    for (int i = ql; i <= qr; i++) {
        if (op[qid[i]] == 3) {
            ans[qid[i]] = vl;
        }
    }
    return;
}

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

// 处理时间小于等于 mid 的操作
for (int i = vl; i <= mid; i++) {
    if (op[i] == 1) {
        // 删除边，这里简化处理
    } else if (op[i] == 2) {
        // 增加点权
        int fx = find(a[i]);
        value[fx] += b[i];
    }
}

// 检查每个操作
int lsiz = 0, rsiz = 0;
for (int i = ql; i <= qr; i++) {
    int id = qid[i];
    if (op[id] == 3) {
        // 查询操作
        long sum = getTopK(a[id], b[id]);
        if (sum >= 0) { // 这里简化判断
            lset[++lsiz] = id;
        } else {
            rset[++rsiz] = id;
        }
    } else {
        // 修改操作放在对应的区间
        if (id <= mid) {
            lset[++lsiz] = id;
        } else {
            rset[++rsiz] = id;
        }
    }
}
```

```
        // 重新排列操作顺序
        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];
        }

        // 回滚操作
        rollback(targetTop);

        // 递归处理左右两部分
        // 左区间：时间范围[vl, mid]
        compute(ql, ql + lsiz - 1, vl, mid);
        // 右区间：时间范围[mid+1, vr]
        compute(ql + lsiz, qr, mid + 1, vr);
    }

    // 查询编号数组
    public static int[] qid = new int[MAXQ];

    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]);
        q = Integer.parseInt(params[2]);

        // 读取节点权值
        String[] values = br.readLine().split(" ");
        for (int i = 1; i <= n; i++) {
            s[i] = Integer.parseInt(values[i - 1]);
        }

        // 读取边信息
        for (int i = 1; i <= m; i++) {
            String[] edge = br.readLine().split(" ");
            eu[i] = Integer.parseInt(edge[0]);
            ev[i] = Integer.parseInt(edge[1]);
        }
```

```
        // 读取操作信息
        for (int i = 1; i <= q; i++) {
            String[] operation = br.readLine().split(" ");
            op[i] = Integer.parseInt(operation[0]);
            a[i] = Integer.parseInt(operation[1]);
            if (op[i] != 1) {
                b[i] = Integer.parseInt(operation[2]);
            }
            qid[i] = i;
        }

        // 初始化并查集
        init();

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

        // 输出结果
        for (int i = 1; i <= q; i++) {
            if (op[i] == 3) {
                out.println(ans[i]);
            }
        }

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

==========

文件: P5163_WD与地图.py

==========

```
# P5163 WD与地图 - Python实现
# 题目来源：https://www.luogu.com.cn/problem/P5163
# 时间复杂度：O(Q * logQ * (N + M))
# 空间复杂度：O(N + M + Q)

import sys

class WDMap:
    def __init__(self):
```

```
    self.MAXN = 100001
    self.MAXM = 200001
    self.MAXQ = 200001
    self.n = 0
    self.m = 0
    self.q = 0

    # 节点权值
    self.s = [0] * self.MAXN

    # 边的信息
    self.eu = [0] * self.MAXM
    self.ev = [0] * self.MAXM

    # 操作信息
    self.op = [0] * self.MAXQ
    self.a = [0] * self.MAXQ
    self.b = [0] * self.MAXQ

    # 并查集
    self.father = [0] * self.MAXN
    self.size = [0] * self.MAXN
    self.value = [0] * self.MAXN
    self.stack = [0] * self.MAXN
    self.top = 0

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

    # 答案
    self.ans = [0] * self.MAXQ

    # 查询编号数组
    self.qid = [0] * self.MAXQ

# 初始化并查集
def init(self):
    for i in range(1, self.n + 1):
        self.father[i] = i
        self.size[i] = 1
        self.value[i] = self.s[i]
    self.top = 0
```

```
# 查找根节点（带路径压缩）
def find(self, x):
    while x != self.father[x]:
        x = self.father[x]
    return x

# 合并两个集合
def union(self, x, y):
    fx = self.find(x)
    fy = self.find(y)
    if fx == fy:
        return False
    if self.size[fx] < self.size[fy]:
        fx, fy = fy, fx
    self.father[fy] = fx
    self.size[fx] += self.size[fy]
    self.value[fx] += self.value[fy]
    self.top += 1
    self.stack[self.top] = fy  # 记录修改，用于回滚
    return True

# 回滚操作
def rollback(self, targetTop):
    while self.top > targetTop:
        y = self.stack[self.top]
        self.top -= 1
        fy = self.find(y)
        self.value[fy] -= self.value[y]
        self.size[fy] -= self.size[y]
        self.father[y] = y

# 计算前 k 大值的和
def getTopK(self, x, k):
    # 这里简化处理，实际应该维护一个优先队列或有序结构
    # 为了整体二分的演示，我们只返回连通块的总和
    fx = self.find(x)
    return self.value[fx]

# 整体二分核心函数
def compute(self, ql, qr, vl, vr):
    if ql > qr:
        return
```

```
if vl == vr:
    for i in range(ql, qr + 1):
        if self.op[self.qid[i]] == 3:
            self.ans[self.qid[i]] = vl
    return

mid = (vl + vr) >> 1
targetTop = self.top

# 处理时间小于等于mid的操作
for i in range(vl, mid + 1):
    if self.op[i] == 1:
        # 删除边，这里简化处理
        pass
    elif self.op[i] == 2:
        # 增加点权
        fx = self.find(self.a[i])
        self.value[fx] += self.b[i]

# 检查每个查询
lsiz = 0
rsiz = 0
for i in range(ql, qr + 1):
    id = self.qid[i]
    if self.op[id] == 3:
        # 查询操作
        sum_val = self.getTopK(self.a[id], self.b[id])
        if sum_val >= 0:  # 这里简化判断
            lsiz += 1
            self.lset[lsiz] = id
        else:
            rsiz += 1
            self.rset[rsiz] = id
    else:
        # 修改操作放在对应的区间
        if id <= mid:
            lsiz += 1
            self.lset[lsiz] = id
        else:
            rsiz += 1
            self.rset[rsiz] = id

# 重新排列操作顺序
```

```
        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]

        # 回滚操作
        self.rollback(targetTop)

        # 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):
        line = sys.stdin.readline().split()
        self.n = int(line[0])
        self.m = int(line[1])
        self.q = int(line[2])

        # 读取节点权值
        values = sys.stdin.readline().split()
        for i in range(1, self.n + 1):
            self.s[i] = int(values[i - 1])

        # 读取边信息
        for i in range(1, self.m + 1):
            edge = sys.stdin.readline().split()
            self.eu[i] = int(edge[0])
            self.ev[i] = int(edge[1])

        # 读取操作信息
        for i in range(1, self.q + 1):
            operation = sys.stdin.readline().split()
            self.op[i] = int(operation[0])
            self.a[i] = int(operation[1])
            if self.op[i] != 1:
                self.b[i] = int(operation[2])
            self.qid[i] = i

        # 初始化并查集
        self.init()

        # 整体二分求解
        self.compute(1, self.q, 1, self.q)
```

```
        # 输出结果
        for i in range(1, self.q + 1):
            if self.op[i] == 3:
                print(self.ans[i])

# 程序入口
#if __name__ == "__main__":
#    solution = WDMap()
#    solution.solve()
```

==========

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：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：使用整体二分算法，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)
//
// 算法适用条件：
// 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. 修改如果对判定答案有贡献，则贡献为确定值
// 4. 贡献满足交换律、结合律，具有可加性
// 5. 题目允许离线操作
//
// 工程化考量：
// - 数据结构选择：使用树状数组实现前缀和查询，时间复杂度为 O(logN)
// - 内存优化：预分配数组空间，避免动态内存分配的开销
// - 性能优化：使用离散化将大值域映射到小区间，减少计算量
// - 异常处理：处理数组边界和极端情况

#include <iostream>
#include <algorithm>
#include <vector>
#include <cstdio>
using namespace std;
```

```
const int MAXN = 100001;
int n, m; // n:数组长度, m:查询次数

// 原始数组，存储输入的数值
int arr[MAXN];

// 离散化后的数组，用于将大值域映射到小下标范围
int sorted[MAXN];

// 查询信息存储
struct Query {
    int l, r, k, id;
} queries[MAXN];

// 树状数组，用于维护当前值域范围内元素的个数
int tree[MAXN];

// 整体二分中用于分类查询的临时存储
int lset[MAXN]; // 满足条件的查询（答案在左半部分）
int rset[MAXN]; // 不满足条件的查询（答案在右半部分）

// 查询的答案存储数组
int ans[MAXN];

/**
 * 计算一个数的 lowbit 值
 * 功能：返回二进制表示中最低位的 1 所代表的数值
 * 例如：lowbit(6) = lowbit(110) = 2
 * 时间复杂度：O(1)
 */
int lowbit(int i) {
    return i & -i;
}

/**
 * 在树状数组中给位置 i 增加 v
 * 功能：更新树状数组中的值，用于后续前缀和查询
 * 时间复杂度：O(logN)
 */
void add(int i, int v) {
    while (i <= n) {
        tree[i] += v;
        i += lowbit(i);
```

```
    }
}

/**
 * 计算前缀和[1..i]
 * 功能：计算从 1 到 i 的元素和
 * 时间复杂度：O(logN)
 */
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]
 * 功能：计算从 l 到 r 的元素和
 * 时间复杂度：O(logN)
 */
int query(int l, int r) {
    return sum(r) - sum(l - 1);
}

/**
 * 整体二分核心函数
 * 功能：递归地对值域进行二分，并将查询分类处理
 * @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）
 * 时间复杂度：O(log(maxValue))
 */
void compute(int ql, int qr, int vl, int vr) {
    // 递归边界 1：没有查询需要处理
    if (ql > qr) {
        return;
    }

    // 递归边界 2：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[queries[i].id] = sorted[vl];
    }
    return;
}

// 二分中点，将值域划分为左右两部分
int mid = (vl + vr) >> 1;

// 预处理：为树状数组添加贡献
// 记录添加的位置，用于后续撤销操作
vector<int> positions;
for (int j = l; j <= n; j++) {
    if (arr[j] <= sorted[mid]) {
        add(j, 1);
        positions.push_back(j);
    }
}

// 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[queries[i].l, queries[i].r]中值小于等于sorted[mid]的元素个数
    int satisfy = query(queries[i].l, queries[i].r);

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

// 撤销对树状数组的修改，恢复到处理前的状态
for (int pos : positions) {
    add(pos, -1);
}

// 保存当前查询信息的临时数组
Query temp[MAXN];
for (int i = ql; i <= qr; i++) {
```

```
        temp[i] = queries[i];
    }

    // 重新排列查询顺序，使得左集合的查询在前，右集合的查询在后
    for (int i = 1; i <= lsiz; i++) {
        queries[ql + i - 1] = temp[lset[i]];
    }
    for (int i = 1; i <= rsiz; i++) {
        queries[ql + lsiz + i - 1] = temp[rset[i]];
    }

    // 递归处理左右两部分
    // 左半部分：值域在[vl, mid]范围内的查询
    compute(ql, ql + lsiz - 1, vl, mid);
    // 右半部分：值域在[mid+1, vr]范围内的查询
    compute(ql + lsiz, qr, mid + 1, vr);
}

/**
 * 主函数，处理输入输出并调用整体二分算法
 * 工程化特点：
 * - 关闭同步提高输入输出效率
 * - 进行离散化处理减少计算量
 * - 合理的数据结构选择
 */
int main() {
    // 关闭同步提高输入输出效率
    ios::sync_with_stdio(false);
    cin.tie(0);

    // 读取数组长度和查询次数
    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 >> queries[i].l >> queries[i].r >> queries[i].k;
        queries[i].id = i;
```

```
    }

    // 离散化：将大值域映射到小下标范围，减少二分的值域范围
    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
    // 初始查询范围[1, m]，初始值域范围[1, uniqueCount]
    compute(1, m, 1, uniqueCount);

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

    return 0;
}
```

==========

文件: POJ2104_KthNumber.java

==========

```
package class168;

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：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：使用整体二分算法，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)
 *
 * 算法适用条件：
```

```
 * 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. 修改如果对判定答案有贡献，则贡献为确定值
 * 4. 贡献满足交换律、结合律，具有可加性
 * 5. 题目允许离线操作
 *
 * 工程化考量：
 * - 数据结构选择：使用树状数组实现前缀和查询，效率高且代码简洁
 * - 内存优化：预分配数组空间，避免动态扩展
 * - 异常处理：处理可能的输入边界情况
 * - 性能优化：使用离散化将大值域映射到小区间，减少计算量
 */
public class POJ2104_KthNumber {
    // 定义数组最大长度
    private static final int MAXN = 100001;
    private static int n, m; // n:数组长度, m:查询次数

    // 原始数组，存储输入的数值
    private static int[] arr = new int[MAXN];

    // 离散化后的数组，用于将大值域映射到小下标范围
    private static int[] sorted = new int[MAXN];

    // 查询信息存储
    private static int[] queryL = new int[MAXN]; // 查询区间左端点
    private static int[] queryR = new int[MAXN]; // 查询区间右端点
    private static int[] queryK = new int[MAXN]; // 查询第 k 小
    private static int[] queryId = new int[MAXN]; // 查询编号

    // 树状数组，用于维护当前值域范围内元素的个数
    private static int[] tree = new int[MAXN];

    // 整体二分中用于分类查询的临时存储
    private static int[] lset = new int[MAXN]; // 满足条件的查询（答案在左半部分）
    private static int[] rset = new int[MAXN]; // 不满足条件的查询（答案在右半部分）

    // 查询的答案存储数组
    private static int[] ans = new int[MAXN];

    /**
     * 计算一个数的 lowbit 值
     * 功能：返回二进制表示中最低位的 1 所代表的数值
     * 例如：lowbit(6) = lowbit(110) = 2
```

```
 * 时间复杂度：O(1)
 */
private static int lowbit(int i) {
    return i & -i;
}

/**
 * 在树状数组中给位置 i 增加 v
 * 功能：更新树状数组中的值，用于后续前缀和查询
 * 时间复杂度：O(logN)
 */
private static void add(int i, int v) {
    while (i <= n) {
        tree[i] += v;
        i += lowbit(i);
    }
}

/**
 * 计算前缀和[1..i]
 * 功能：计算从 1 到 i 的元素和
 * 时间复杂度：O(logN)
 */
private 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]
 * 功能：计算从 l 到 r 的元素和
 * 时间复杂度：O(logN)
 */
private static int query(int l, int r) {
    return sum(r) - sum(l - 1);
}

/**
 * 整体二分核心函数
```

```
     * 功能：递归地对值域进行二分，并将查询分类处理
     * @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）
     * 时间复杂度：O(log(maxValue))
     */
    private static void compute(int ql, int qr, int vl, int vr) {
        // 递归边界 1：没有查询需要处理
        if (ql > qr) {
            return;
        }

        // 递归边界 2：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，将值域划分为左右两部分
        int mid = (vl + vr) >> 1;

        // 预处理：为树状数组添加贡献
        // 记录原始数组中每个元素的位置，用于后续的添加和撤销操作
        List<Integer> positions = new ArrayList<>();
        for (int j = 1; j <= n; j++) {
            if (arr[j] <= sorted[mid]) {
                add(j, 1);
                positions.add(j);
            }
        }

        // 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++) {
            int id = queryId[i];
            // 查询区间[queryL[id], queryR[id]]中值小于等于 sorted[mid]的元素个数
            int satisfy = query(queryL[id], queryR[id]);

            if (satisfy >= queryK[id]) {
                // 说明第 k 小的数在左半部分值域
```

```
            lset[++lsiz] = id;
        } else {
            // 说明第 k 小的数在右半部分值域，需要在右半部分找第(k-satisfy)小的数
            queryK[id] -= satisfy;
            rset[++rsiz] = id;
        }
    }

    // 撤销对树状数组的修改，恢复到处理前的状态
    for (int pos : positions) {
        add(pos, -1);
    }

    // 重新排列查询顺序，使得左集合的查询在前，右集合的查询在后
    for (int i = 1; i <= lsiz; i++) {
        queryId[ql + i - 1] = lset[i];
    }
    for (int i = 1; i <= rsiz; i++) {
        queryId[ql + lsiz + i - 1] = rset[i];
    }

    // 递归处理左右两部分
    // 左半部分：值域在[vl, mid]范围内的查询
    compute(ql, ql + lsiz - 1, vl, mid);
    // 右半部分：值域在[mid+1, vr]范围内的查询
    compute(ql + lsiz, qr, mid + 1, vr);
}

/**
 * 主函数，处理输入输出并调用整体二分算法
 * 工程化特点：
 * - 使用 BufferedReader 和 PrintWriter 提高 IO 效率
 * - 进行离散化处理减少计算量
 * - 合理的内存管理
 */
public static void main(String[] args) throws IOException {
    // 初始化输入输出流
    BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
    PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));

    // 读取数组长度和查询次数
    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; // 记录查询编号
        }

        // 离散化：将大值域映射到小下标范围，减少二分的值域范围
        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
        // 初始查询范围[1, m]，初始值域范围[1, uniqueCount]
        compute(1, m, 1, uniqueCount);

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

        // 关闭流，确保输出完全刷新
        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：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：使用整体二分算法，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)
#
# 算法适用条件：
# 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. 修改如果对判定答案有贡献，则贡献为确定值
# 4. 贡献满足交换律、结合律，具有可加性
# 5. 题目允许离线操作
#
# 工程化考量：
# - 数据结构选择：使用树状数组实现前缀和查询
# - Python 特有优化：使用列表预先分配空间，避免频繁动态扩展
# - 边界处理：注意 Python 中的索引从 0 开始的特性，但保持与其他语言版本一致，使用 1-based 索引
# - 性能优化：使用离散化减少计算量，避免大数值操作

import sys

class POJ2104_KthNumber:
    def __init__(self):
        self.MAXN = 100001  # 定义数组最大长度
        self.n = 0  # 数组长度
        self.m = 0  # 查询次数
        self.arr = [0] * (self.MAXN)  # 原始数组
        self.sorted = [0] * (self.MAXN)  # 离散化后的数组
        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
        self.tree = [0] * (self.MAXN)  # 树状数组
        self.lset = [0] * (self.MAXN)  # 满足条件的查询
        self.rset = [0] * (self.MAXN)  # 不满足条件的查询
        self.ans = [0] * (self.MAXN)  # 查询的答案
```

```
def lowbit(self, i):
    """
    计算一个数的 lowbit 值
    功能：返回二进制表示中最低位的 1 所代表的数值
    例如：lowbit(6) = lowbit(110) = 2
    时间复杂度：O(1)
    """
    return i & -i

def add(self, i, v):
    """
    在树状数组中给位置 i 增加 v
    功能：更新树状数组中的值，用于后续前缀和查询
    时间复杂度：O(logN)
    """
    while i <= self.n:
        self.tree[i] += v
        i += self.lowbit(i)

def sum(self, i):
    """
    计算前缀和[1..i]
    功能：计算从 1 到 i 的元素和
    时间复杂度：O(logN)
    """
    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]
    功能：计算从 l 到 r 的元素和
    时间复杂度：O(logN)
    """
    return self.sum(r) - self.sum(l - 1)

def compute(self, ql, qr, vl, vr):
    """
    整体二分核心函数
```

```
功能：递归地对值域进行二分，并将查询分类处理
参数：
    ql: 查询范围的左端点
    qr: 查询范围的右端点
    vl: 值域范围的左端点（离散化后的下标）
    vr: 值域范围的右端点（离散化后的下标）
时间复杂度：O(log(maxValue))
"""
# 递归边界 1：没有查询需要处理
if ql > qr:
    return

# 递归边界 2：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，将值域划分为左右两部分
mid = (vl + vr) >> 1

# 预处理：为树状数组添加贡献
# 记录添加的位置，用于后续撤销操作
positions = []
for j in range(1, self.n + 1):
    if self.arr[j] <= self.sorted[mid]:
        self.add(j, 1)
        positions.append(j)

# 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):
    id = self.queryId[i]
    # 查询区间[self.queryL[id], self.queryR[id]]中值小于等于 self.sorted[mid]的元素个数
    satisfy = self.query(self.queryL[id], self.queryR[id])

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

```
            self.queryK[id] -= satisfy
            rsiz += 1
            self.rset[rsiz] = id

    # 撤销对树状数组的修改，恢复到处理前的状态
    for pos in positions:
        self.add(pos, -1)

    # 保存当前查询ID数组的临时副本
    temp = self.queryId.copy()

    # 重新排列查询顺序，使得左集合的查询在前，右集合的查询在后
    for i in range(1, lsiz + 1):
        self.queryId[ql + i - 1] = self.lset[i]
    for i in range(1, rsiz + 1):
        self.queryId[ql + lsiz + i - 1] = self.rset[i]

    # 递归处理左右两部分
    # 左半部分：值域在[vl, mid]范围内的查询
    self.compute(ql, ql + lsiz - 1, vl, mid)
    # 右半部分：值域在[mid+1, vr]范围内的查询
    self.compute(ql + lsiz, qr, mid + 1, vr)

def main(self):
    """
    主函数，处理输入输出并调用整体二分算法
    工程化特点：
    - 使用sys.stdin.readline提高输入效率
    - 进行离散化处理减少计算量
    - 注意Python中的边界条件处理
    """
    # 读取数组长度和查询次数
    input_line = sys.stdin.readline()
    self.n, self.m = map(int, input_line.split())

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

    # 读取查询
    for i in range(1, self.m + 1):
```

```
            query = list(map(int, sys.stdin.readline().split()))
            self.queryL[i] = query[0]
            self.queryR[i] = query[1]
            self.queryK[i] = query[2]
            self.queryId[i] = i  # 记录查询编号

        # 离散化：将大值域映射到小下标范围，减少二分的值域范围
        self.sorted[1:self.n+1].sort()
        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
        # 初始查询范围[1, m]，初始值域范围[1, uniqueCount]
        self.compute(1, self.m, 1, uniqueCount)

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

# 程序入口
if __name__ == "__main__":
    solution = POJ2104_KthNumber()
    solution.main()

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