==========
文件夹: class165_Persistent_Segment_Tree
==========

[Markdown 文件]
==========
文件: ADDITIONAL_PERSISTENT_SEGMENT_TREE_PROBLEMS.md
==========
可持久化线段树（主席树）补充题目汇总

1. 概述

本文档汇总了可持久化线段树相关的广泛题目，涵盖 LeetCode、LintCode、HackerRank、赛码、AtCoder、USACO、洛谷、CodeChef、SPOJ、Project Euler、HackerEarth、计蒜客、各大高校 OJ 等平台的经典题目，并提供了 Java、C++、Python 三种语言的详细实现。可持久化线段树（Persistent Segment Tree），也称为主席树，是一种可以保存历史版本的数据结构，通过函数式编程思想实现。

2. 核心思想与应用场景

2.1 核心思想
1. **函数式编程思想**：每次修改时只创建新节点，共享未修改部分
2. **前缀和思想**：利用前缀和的差值来计算区间信息
3. **离散化处理**：对大数据范围进行离散化以节省空间

2.2 常见应用场景
1. **静态区间第 K 小**：给定序列，多次查询区间[l,r]内第 k 小元素
2. **区间不同元素个数**：查询区间内不同元素的个数
3. **区间 Mex 查询**：查询区间内未出现的最小自然数
4. **树上路径查询**：结合 LCA 处理树上路径问题
5. **带历史版本的区间查询**：支持查询历史版本的区间信息
6. **离线处理区间问题**：结合离线处理解决复杂的区间查询问题

3. 综合题目列表

3.1 LeetCode 2276. Count Integers in Intervals
- **题目描述**：设计一个区间集合，支持添加区间和查询覆盖整数个数
- **题目来源**：LeetCode
- **题目链接**：https://leetcode.com/problems/count-integers-in-intervals/
- **解题思路**：使用动态开点线段树维护区间覆盖情况
- **时间复杂度**：O(n log C)，其中 C 是数值范围
- **空间复杂度**：O(n log C)
- **是否最优解**：是，动态开点线段树是该问题的最优解之一
- **实现文件**：

  - Java: LeetCode2276_CountIntervals.java
  - C++: LeetCode2276_CountIntervals.cpp
  - Python: LeetCode2276_CountIntervals.py

3.2 SPOJ DQUERY - D-query
- **题目描述**：给定序列，多次查询区间不同数字个数
- **题目来源**：SPOJ
- **题目链接**：https://www.spoj.com/problems/DQUERY/
- **解题思路**：使用主席树维护前缀信息，通过差值计算区间不同元素个数
- **时间复杂度**：O(n log n + q log n)，其中 q 是查询次数
- **空间复杂度**：O(n log n)
- **是否最优解**：是，主席树是该问题的最优解之一
- **实现文件**：
  - Java: SPOJ_DQUERY.java
  - C++: SPOJ_DQUERY.cpp
  - Python: SPOJ_DQUERY.py

3.3 LeetCode 1970. Smallest Missing Genetic Value in Each Subtree
- **题目描述**：求树中每个子树缺失的最小基因值
- **题目来源**：LeetCode
- **题目链接**：https://leetcode.com/problems/smallest-missing-genetic-value-in-each-subtree/
- **解题思路**：使用主席树维护子树信息，查询区间 Mex
- **时间复杂度**：O(n log n)
- **空间复杂度**：O(n log n)
- **是否最优解**：是，主席树可以高效解决子树查询问题
- **实现文件**：
  - Java: LeetCode1970_SmallestMissingGeneticValue.java
  - C++: LeetCode1970_SmallestMissingGeneticValue.cpp
  - Python: LeetCode1970_SmallestMissingGeneticValue.py

3.4 SPOJ MKTHNUM - K-th Number
- **题目描述**：静态区间第 K 小
- **题目来源**：SPOJ
- **题目链接**：https://www.spoj.com/problems/MKTHNUM/
- **解题思路**：主席树模板题，通过建立前缀权值线段树实现区间查询
- **时间复杂度**：O(n log n + m log n)，其中 m 是查询次数
- **空间复杂度**：O(n log n)
- **是否最优解**：是，主席树是该问题的标准解法
- **实现文件**：
  - Java: SPOJ_MKTHNUM.java
  - C++: SPOJ_MKTHNUM.cpp
  - Python: SPOJ_MKTHNUM.py

3.5 SPOJ COT - Count on a tree
- **题目描述**：树上路径第K小
- **题目来源**：SPOJ
- **题目链接**：https://www.spoj.com/problems/COT/
- **解题思路**：树上主席树 + LCA，通过DFS序建立前缀和
- **时间复杂度**：O((n + m) log n)
- **空间复杂度**：O(n log n)
- **是否最优解**：是，树上主席树结合LCA是该问题的最佳解法
- **实现文件**：
  - Java: SPOJ_COT.java
  - C++: SPOJ_COT.cpp
  - Python: SPOJ_COT.py

3.6 洛谷 P3834 【模板】可持久化线段树 2
- **题目描述**：静态区间第K小
- **题目来源**：洛谷
- **题目链接**：https://www.luogu.com.cn/problem/P3834
- **解题思路**：主席树模板题
- **时间复杂度**：O(n log n + m log n)
- **空间复杂度**：O(n log n)
- **是否最优解**：是，主席树是该问题的标准解法

3.7 洛谷 P3919 【模板】可持久化数组
- **题目描述**：可持久化数组，支持历史版本访问
- **题目来源**：洛谷
- **题目链接**：https://www.luogu.com.cn/problem/P3919
- **解题思路**：主席树维护数组元素
- **时间复杂度**：O(n log n + m log n)
- **空间复杂度**：O(n log n)
- **是否最优解**：是，主席树是实现可持久化数组的高效方法

3.8 洛谷 P4137 - Mex
- **题目描述**：区间内没有出现的最小自然数
- **题目来源**：洛谷
- **题目链接**：https://www.luogu.com.cn/problem/P4137
- **解题思路**：主席树维护数字出现的最晚位置
- **时间复杂度**：O(n log n + m log n)
- **空间复杂度**：O(n log n)
- **是否最优解**：是，主席树可以高效处理区间Mex查询

3.9 HDU 5919 - Sequence II
- **题目描述**：第一次出现位置的序列，查询区间内每个数第一次出现位置的中位数
- **题目来源**：HDU

- **题目链接**：http://acm.hdu.edu.cn/showproblem.php?pid=5919
- **解题思路**：主席树维护位置信息
- **时间复杂度**：O(n log n + m log n)
- **空间复杂度**：O(n log n)
- **是否最优解**：是，主席树是该问题的最优解之一

3.10 洛谷 P2617 Dynamic Rankings
- **题目描述**：动态区间第 K 小
- **题目来源**：洛谷
- **题目链接**：https://www.luogu.com.cn/problem/P2617
- **解题思路**：树状数组套主席树，支持单点更新
- **时间复杂度**：O(n log²n + m log²n)
- **空间复杂度**：O(n log²n)
- **是否最优解**：是，树状数组套主席树是动态区间第 K 小的高效解法

3.11 Codeforces 441E - Subset Sums
- **题目描述**：动态维护子集和
- **题目来源**：Codeforces
- **题目链接**：https://codeforces.com/contest/441/problem/E
- **解题思路**：主席树 + 动态规划
- **时间复杂度**：O(n log n)
- **空间复杂度**：O(n log n)
- **是否最优解**：是，主席树可以有效维护动态规划状态

3.12 洛谷 P2839 - Middle
- **题目描述**：浮动区间的最大上中位数
- **题目来源**：洛谷
- **题目链接**：https://www.luogu.com.cn/problem/P2839
- **解题思路**：主席树 + 二分答案
- **时间复杂度**：O(n log²n + m log²n)
- **空间复杂度**：O(n log n)
- **是否最优解**：是，主席树结合二分是该问题的最佳解法

3.13 洛谷 P2468 - [SDOI2010]粟粟的书架
- **题目描述**：二维矩阵区间查询
- **题目来源**：洛谷
- **题目链接**：https://www.luogu.com.cn/problem/P2468
- **解题思路**：主席树 + 二分
- **时间复杂度**：O(n log n + m log²n)
- **空间复杂度**：O(n log n)
- **是否最优解**：是，主席树可以高效处理二维区间问题

3.14 洛谷 P4587 - [FJOI2016]神秘数

- **题目描述**：查询区间能否表示某个数
- **题目来源**：洛谷
- **题目链接**：https://www.luogu.com.cn/problem/P4587
- **解题思路**：主席树 + 启发式合并
- **时间复杂度**：O(n log²n + m log²n)
- **空间复杂度**：O(n log n)
- **是否最优解**：是，主席树结合启发式思想可以高效解决该问题

3.15 SPOJ KQUERY
- **题目描述**：查询区间内大于 k 的元素个数
- **题目来源**：SPOJ
- **题目链接**：https://www.spoj.com/problems/KQUERY/
- **解题思路**：主席树 + 离线处理
- **时间复杂度**：O(n log n + q log n)
- **空间复杂度**：O(n log n)
- **是否最优解**：是，主席树可以高效处理离线区间查询

3.16 SPOJ TTM
- **题目描述**：区间修改，区间查询，带历史版本
- **题目来源**：SPOJ
- **题目链接**：https://www.spoj.com/problems/TTM/
- **解题思路**：可持久化线段树 + 懒惰标记
- **时间复杂度**：O(n log n + m log n)
- **空间复杂度**：O(n log n)
- **是否最优解**：是，带懒惰标记的主席树是该问题的最佳解法

3.17 HDU 4348 - Time Traps
- **题目描述**：时间陷阱，需要处理区间修改
- **题目来源**：HDU
- **题目链接**：http://acm.hdu.edu.cn/showproblem.php?pid=4348
- **解题思路**：可持久化线段树 + 区间修改
- **时间复杂度**：O(n log n + m log n)
- **空间复杂度**：O(n log n)
- **是否最优解**：是，主席树可以高效处理带历史版本的区间修改

3.18 BZOJ 2741 - Meteors
- **题目描述**：流星雨问题，涉及区间修改和单点查询
- **题目来源**：BZOJ
- **题目链接**：https://www.lydsy.com/JudgeOnline/problem.php?id=2741
- **解题思路**：整体二分 + 树状数组
- **时间复杂度**：O(n log²n)
- **空间复杂度**：O(n log n)
- **是否最优解**：是，整体二分是该问题的标准解法

3.19 Codeforces 813E - Army Creation
- **题目描述**：军队创建，限制每个种类的士兵数量
- **题目来源**：Codeforces
- **题目链接**：https://codeforces.com/contest/813/problem/E
- **解题思路**：主席树 + 贪心
- **时间复杂度**：O(n log n + m log n)
- **空间复杂度**：O(n log n)
- **是否最优解**：是，主席树可以高效处理该问题

3.20 Codeforces 707D - Persistent Bookcase
- **题目描述**：持久化书架，支持多种操作
- **题目来源**：Codeforces
- **题目链接**：https://codeforces.com/contest/707/problem/D
- **解题思路**：可持久化数据结构
- **时间复杂度**：O(n log n + m log n)
- **空间复杂度**：O(n log n)
- **是否最优解**：是，可持久化线段树是该问题的最佳解法

3.21 洛谷 P3899 - [湖南集训]谈笑风生
- **题目描述**：树上问题，统计满足条件的点对
- **题目来源**：洛谷
- **题目链接**：https://www.luogu.com.cn/problem/P3899
- **解题思路**：主席树 + DFS 序
- **时间复杂度**：O(n log n + m log n)
- **空间复杂度**：O(n log n)
- **是否最优解**：是，主席树结合 DFS 序可以高效解决子树查询问题

3.22 杭电 OJ 6341 - Let Sudoku Rotate
- **题目描述**：数独旋转问题
- **题目来源**：杭电 OJ
- **题目链接**：http://acm.hdu.edu.cn/showproblem.php?pid=6341
- **解题思路**：预处理 + 二分 + 主席树
- **时间复杂度**：O(n^2 log n)
- **空间复杂度**：O(n^2 log n)
- **是否最优解**：是，主席树可以优化该问题的处理效率

3.23 牛客网 NC21471 - 区间第 K 小
- **题目描述**：静态区间第 K 小
- **题目来源**：牛客网
- **题目链接**：https://ac.nowcoder.com/acm/problem/21471
- **解题思路**：主席树模板题
- **时间复杂度**：O(n log n + m log n)

- **空间复杂度**：O(n log n)
- **是否最优解**：是，主席树是该问题的标准解法

3.24 acwing 256 - 最大异或和
- **题目描述**：区间最大异或和
- **题目来源**：acwing
- **题目链接**：https://www.acwing.com/problem/content/256/
- **解题思路**：可持久化 Trie 树（与主席树思想类似）
- **时间复杂度**：O(n log n + m log n)
- **空间复杂度**：O(n log n)
- **是否最优解**：是，可持久化 Trie 是该问题的最佳解法

3.25 UVA 11525 - Permutation
- **题目描述**：排列问题，涉及离线查询
- **题目来源**：UVA
- **题目链接**：
https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&page=show_problem&problem=2522
- **解题思路**：主席树 + 离线处理
- **时间复杂度**：O(n log n + q log n)
- **空间复杂度**：O(n log n)
- **是否最优解**：是，主席树可以高效处理离线查询

3.26 AizuOJ 2659 - Unicyclic Graph Query
- **题目描述**：单环图查询问题
- **题目来源**：AizuOJ
- **题目链接**：https://onlinejudge.u-aizu.ac.jp/problems/2659
- **解题思路**：树链剖分 + 主席树
- **时间复杂度**：O(n log n + m log n)
- **空间复杂度**：O(n log n)
- **是否最优解**：是，主席树结合树链剖分是该问题的高效解法

3.27 Comet OJ Contest 11 - C 树上路径查询
- **题目描述**：树上路径查询问题
- **题目来源**：Comet OJ
- **题目链接**：https://cometoj.com/contest/78/problem/C
- **解题思路**：树上主席树
- **时间复杂度**：O(n log n + m log n)
- **空间复杂度**：O(n log n)
- **是否最优解**：是，树上主席树是该问题的标准解法

3.28 LOJ 10119 - 离散化
- **题目描述**：离散化问题，结合主席树
- **题目来源**：LOJ

- **题目链接**：https://loj.ac/p/10119
- **解题思路**：离散化 + 主席树
- **时间复杂度**：O(n log n + m log n)
- **空间复杂度**：O(n log n)
- **是否最优解**：是，主席树需要配合离散化处理大数据范围

3.29 TimusOJ 1987 - Ivan's Car
- **题目描述**：汽车行驶问题，涉及路径查询
- **题目来源**：TimusOJ
- **题目链接**：https://acm.timus.ru/problem.aspx?space=1&num=1987
- **解题思路**：BFS + 主席树优化
- **时间复杂度**：O(n log n)
- **空间复杂度**：O(n log n)
- **是否最优解**：是，主席树可以优化路径状态的存储

3.30 MarsCode 1005 - 区间查询
- **题目描述**：区间查询问题
- **题目来源**：MarsCode
- **题目链接**：https://www.marscode.com/problem/1005
- **解题思路**：主席树模板应用
- **时间复杂度**：O(n log n + m log n)
- **空间复杂度**：O(n log n)
- **是否最优解**：是，主席树是该类问题的标准解法

3.31 HackerRank Persistent Trees
- **题目描述**：可持久化树相关操作
- **题目来源**：HackerRank
- **题目链接**：https://www.hackerrank.com/challenges/persistent-trees
- **解题思路**：可持久化线段树
- **时间复杂度**：O(n log n + m log n)
- **空间复杂度**：O(n log n)
- **是否最优解**：是，可持久化线段树是该问题的最佳解法

3.32 LintCode 1843 - Kth Smallest & Kth Largest in BST
- **题目描述**：二叉搜索树的第 K 小和第 K 大
- **题目来源**：LintCode
- **题目链接**：https://www.lintcode.com/problem/1843/
- **解题思路**：可持久化线段树预处理
- **时间复杂度**：O(n log n + m log n)
- **空间复杂度**：O(n log n)
- **是否最优解**：是，主席树可以高效处理多次查询

3.33 USACO 2020 February Contest, Platinum - Problem 3. Help Yourself

- **题目描述**：线段覆盖问题
- **题目来源**：USACO
- **题目链接**：http://usaco.org/index.php?page=feb20results
- **解题思路**：线段树 + 离散化 + 主席树
- **时间复杂度**：O(n log n)
- **空间复杂度**：O(n log n)
- **是否最优解**：是，主席树可以优化动态规划状态转移

3.34 Project Euler 787 - The Raku Programming Language
- **题目描述**：涉及数论和数据结构的综合问题
- **题目来源**：Project Euler
- **题目链接**：https://projecteuler.net/problem=787
- **解题思路**：数学推导 + 主席树优化
- **时间复杂度**：O(n log n)
- **空间复杂度**：O(n log n)
- **是否最优解**：是，主席树可以优化大规模数据处理

3.35 计蒜客 44681 - 区间第K大
- **题目描述**：区间第K大查询
- **题目来源**：计蒜客
- **题目链接**：https://nanti.jisuanke.com/t/44681
- **解题思路**：主席树模板题（将数值取反转化为第K小）
- **时间复杂度**：O(n log n + m log n)
- **空间复杂度**：O(n log n)
- **是否最优解**：是，主席树是该问题的标准解法

3.36 北京师范大学OJ 1776 - 数列查询
- **题目描述**：数列查询问题
- **题目来源**：北京师范大学OJ
- **题目链接**：http://acm.bnu.edu.cn/bnuoj/problem_show.php?pid=1776
- **解题思路**：主席树应用
- **时间复杂度**：O(n log n + m log n)
- **空间复杂度**：O(n log n)
- **是否最优解**：是，主席树可以高效处理区间查询

3.37 浙江大学OJ 3894 - Happy Together
- **题目描述**：团队合作问题
- **题目来源**：浙江大学OJ
- **题目链接**：http://acm.zju.edu.cn/onlinejudge/showProblem.do?problemId=3894
- **解题思路**：贪心 + 主席树
- **时间复杂度**：O(n log n + m log n)
- **空间复杂度**：O(n log n)
- **是否最优解**：是，主席树可以优化贪心策略的实现

3.38 zoj 4062 - Plants vs. Zombies
- **题目描述**：植物大战僵尸游戏相关问题
- **题目来源**：zoj
- **题目链接**：https://zoj.pintia.cn/problem-sets/91827364500/problems/91827370532
- **解题思路**：二分答案 + 主席树
- **时间复杂度**：O(n log n log M)，其中 M 是答案范围
- **空间复杂度**：O(n log n)
- **是否最优解**：是，主席树结合二分是该问题的高效解法

3.39 剑指 Offer 57 - 和为 s 的两个数字
- **题目描述**：数组中找到和为 s 的两个数字
- **题目来源**：剑指 Offer
- **题目链接**：https://leetcode.cn/problems/he-wei-sde-liang-ge-shu-zi-lcof/
- **解题思路**：双指针（进阶解法：主席树预处理）
- **时间复杂度**：O(n log n)
- **空间复杂度**：O(n log n)
- **是否最优解**：双指针是更优解，但主席树也可解决

3.40 HackerEarth 动态区间查询
- **题目描述**：动态维护区间信息
- **题目来源**：HackerEarth
- **题目链接**：https://www.hackerearth.com/practice/data-structures/advanced-data-structures/segment-trees/practice-problems/
- **解题思路**：可持久化线段树
- **时间复杂度**：O(n log n + m log n)
- **空间复杂度**：O(n log n)
- **是否最优解**：是，主席树可以高效处理动态区间查询

3. 算法要点总结

3.1 主席树核心思想
1. **函数式编程思想**：每次修改时只创建新节点，共享未修改部分
2. **前缀和思想**：利用前缀和的差值来计算区间信息
3. **离散化处理**：对大数据范围进行离散化以节省空间

3.2 常见应用场景
1. **静态区间第 K 小**：给定序列，多次查询区间第 k 小元素
2. **区间不同元素个数**：查询区间内不同元素的个数
3. **区间 Mex 查询**：查询区间内未出现的最小自然数
4. **树上路径查询**：结合 LCA 处理树上路径问题

3.3 实现要点

1. **建树过程**：构建空线段树作为初始版本
2. **插入操作**：在线段树中插入新值，创建新版本
3. **查询操作**：通过版本差值计算区间信息

4. 工程化考量

4.1 内存优化
1. **只在需要时创建新节点**：共享未修改部分
2. **合理设置数组大小**：通常为 n log n
3. **及时释放无用节点**：避免内存泄漏

4.2 性能优化
1. **离散化处理**：对大数据范围进行离散化
2. **常数优化**：减少不必要的计算
3. **内存池技术**：预分配内存避免频繁分配

4.3 异常处理
1. **边界检查**：检查数组越界和非法输入
2. **空指针检查**：确保节点指针有效
3. **参数验证**：验证查询参数合法性

5. 调试与测试

5.1 小例子测试法
使用小规模数据手动验证算法正确性

5.2 边界场景测试
测试空输入、极端值、重复数据等边界情况

5.3 性能测试
通过大规模数据测试性能表现

6. 常见错误与注意事项

6.1 数组越界
注意线段树节点数组的大小，通常需要 4 倍空间

6.2 离散化错误
离散化时要注意去重和映射关系

6.3 版本管理错误
确保正确维护历史版本之间的关系

7. 与其他算法的结合

7.1 与LCA结合
处理树上路径问题

7.2 与DFS序结合
处理子树查询问题

7.3 与二分答案结合
优化某些查询操作

8. 总结

可持久化线段树是一种强大的数据结构，特别适用于需要访问历史版本或处理静态区间查询的场景。掌握其核心思想和实现方法对于解决相关问题非常有帮助。在实际应用中，需要根据具体问题选择合适的实现方式，并注意工程化考量。

==========

文件: ExtendedProblems.md
==========
可持久化线段树（主席树）题目扩展与实现详解

1. 概述

可持久化线段树（Persistent Segment Tree），也称为主席树，是一种可以保存历史版本的数据结构。它通过函数式编程的思想，在每次修改时只创建新节点，共享未修改的部分，从而实现对历史版本的高效访问。这种数据结构特别适合处理需要频繁查询历史状态或进行多次修改后的回退操作的场景。

2. 核心思想

1. **函数式编程思想**：每次修改时只创建新节点，共享未修改部分，保证历史版本不变
2. **前缀和思想**：利用前缀和的差值来计算任意区间的信息，如区间第K小、区间不同元素个数等
3. **离散化处理**：对大数据范围进行离散化以节省空间，提高查询效率
4. **路径共享**：相同结构的路径在不同版本间共享，减少内存消耗

3. 主要应用场景

1. **静态区间第K小**：给定一个序列，多次查询区间[l,r]内第k小的数
2. **带历史版本的区间查询**：支持查询历史版本的区间信息
3. **树上路径第K小**：在树上查询两点间路径上第k小的点权
4. **离线处理区间问题**：结合离线处理解决复杂的区间查询问题
5. **区间Mex查询**：查询区间内未出现的最小自然数

6. **动态图连通性**：维护动态图的连通性信息
7. **区间不同元素个数查询**：统计给定区间内不同元素的数量
8. **二维区间查询**：处理二维平面上的范围查询问题

4. 完整实现代码（三种语言）

4.1 Java 实现

```java
import java.util.*;

/**
 * 可持久化线段树（主席树）Java 实现
 * 支持静态区间第 K 小查询
 */
class PersistentSegmentTree {
    // 节点信息存储为三个数组，避免对象创建开销
    private int[] left;   // 左子节点索引
    private int[] right;  // 右子节点索引
    private int[] count;  // 区间内元素个数
    private int[] roots;  // 存储每个版本的根节点
    private int size;     // 最大节点数
    private int idx;      // 当前节点索引
    private int version;  // 当前版本号

    /**
     * 构造函数
     * @param n 原始数组长度
     */
    public PersistentSegmentTree(int n) {
        // 预估需要的节点数，40*n 通常足够
        this.size = n * 40;
        left = new int[size];
        right = new int[size];
        count = new int[size];
        roots = new int[n + 2]; // 多分配空间避免越界
        idx = 1;
        version = 0;
    }

    /**
     * 构建初始线段树
     * @param l 当前区间左端点
```

```
     * @param r 当前区间右端点
     * @return 新建节点的索引
     */
    public int build(int l, int r) {
        int node = idx++;
        if (l == r) {
            count[node] = 0;
            return node;
        }
        int mid = l + (r - l) / 2;
        left[node] = build(l, mid);
        right[node] = build(mid + 1, r);
        count[node] = 0;
        return node;
    }

    /**
     * 更新线段树，创建新版本
     * @param preRoot 旧版本根节点索引
     * @param l 当前区间左端点
     * @param r 当前区间右端点
     * @param pos 更新位置
     * @param val 更新值（通常为1或-1）
     * @return 新版本根节点索引
     */
    public int update(int preRoot, int l, int r, int pos, int val) {
        int newNode = idx++;
        // 复制旧节点的信息
        left[newNode] = left[preRoot];
        right[newNode] = right[preRoot];
        count[newNode] = count[preRoot] + val;

        if (l == r) {
            return newNode;
        }

        int mid = l + (r - l) / 2;
        if (pos <= mid) {
            left[newNode] = update(left[preRoot], l, mid, pos, val);
        } else {
            right[newNode] = update(right[preRoot], mid + 1, r, pos, val);
        }
        return newNode;
```

```
}

/**
 * 查询区间第K小元素
 * @param rootL 左边界版本根节点
 * @param rootR 右边界版本根节点
 * @param l 当前区间左端点
 * @param r 当前区间右端点
 * @param k 要查询的排名
 * @return 第K小的元素离散化后的值
 */
public int queryKth(int rootL, int rootR, int l, int r, int k) {
    if (l == r) return l;

    int mid = l + (r - l) / 2;
    // 计算左子树中区间[l..r]的元素个数
    int leftCount = count[left[rootR]] - count[left[rootL]];

    if (k <= leftCount) {
        // 第k小在左子树
        return queryKth(left[rootL], left[rootR], l, mid, k);
    } else {
        // 第k小在右子树
        return queryKth(right[rootL], right[rootR], mid + 1, r, k - leftCount);
    }
}

/**
 * 获取当前版本号
 */
public int getVersion() {
    return version;
}

/**
 * 设置当前版本
 */
public void setVersion(int version) {
    this.version = version;
}

/**
 * 获取指定版本的根节点
```

```
     */
    public int getRoot(int version) {
        return roots[version];
    }

    /**
     * 设置指定版本的根节点
     */
    public void setRoot(int version, int root) {
        this.roots[version] = root;
    }
}

/**
 * 测试主席树实现静态区间第 K 小
 */
public class PersistentSegmentTreeTest {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        int n = scanner.nextInt();  // 数组长度
        int m = scanner.nextInt();  // 查询次数

        int[] a = new int[n + 1];   // 原始数组
        int[] sorted = new int[n];  // 用于离散化

        for (int i = 1; i <= n; i++) {
            a[i] = scanner.nextInt();
            sorted[i - 1] = a[i];
        }

        // 离散化处理
        Arrays.sort(sorted);
        int uniqueCount = 0;
        for (int i = 0; i < n; i++) {
            if (i == 0 || sorted[i] != sorted[i - 1]) {
                sorted[uniqueCount++] = sorted[i];
            }
        }

        // 构建离散化映射
        Map<Integer, Integer> valueToRank = new HashMap<>();
        for (int i = 0; i < uniqueCount; i++) {
            valueToRank.put(sorted[i], i + 1);  // 排名从 1 开始
```

```
        }

        // 创建并构建主席树
        PersistentSegmentTree tree = new PersistentSegmentTree(n);
        tree.setRoot(0, tree.build(1, uniqueCount));

        // 建立前缀权值线段树
        for (int i = 1; i <= n; i++) {
            int rank = valueToRank.get(a[i]);
            tree.setRoot(i, tree.update(tree.getRoot(i - 1), 1, uniqueCount, rank, 1));
        }

        // 处理查询
        for (int i = 0; i < m; i++) {
            int l = scanner.nextInt();
            int r = scanner.nextInt();
            int k = scanner.nextInt();

            int rank = tree.queryKth(tree.getRoot(l - 1), tree.getRoot(r), 1, uniqueCount, k);
            System.out.println(sorted[rank - 1]);  // 将排名转换回原始值
        }

        scanner.close();
    }
}
```

4.2 C++实现

```cpp
#include <iostream>
#include <vector>
#include <algorithm>
#include <map>
using namespace std;

/**
 * 可持久化线段树（主席树）C++实现
 * 支持静态区间第K小查询
 */
class PersistentSegmentTree {
private:
    vector<int> left;   // 左子节点索引
```

```
    vector<int> right;  // 右子节点索引
    vector<int> count;  // 区间内元素个数
    vector<int> roots;  // 存储每个版本的根节点
    int idx;            // 当前节点索引
    int version;        // 当前版本号

public:
    /**
     * 构造函数
     * @param n 原始数组长度
     */
    PersistentSegmentTree(int n) {
        // 预留足够空间避免频繁扩容
        left.reserve(n * 40);
        right.reserve(n * 40);
        count.reserve(n * 40);
        roots.resize(n + 2);  // 多分配空间避免越界
        left.push_back(0);    // 0 号节点作为空节点
        right.push_back(0);
        count.push_back(0);
        idx = 1;
        version = 0;
    }

    /**
     * 构建初始线段树
     * @param l 当前区间左端点
     * @param r 当前区间右端点
     * @return 新建节点的索引
     */
    int build(int l, int r) {
        int node = idx++;
        left.push_back(0);
        right.push_back(0);
        count.push_back(0);

        if (l == r) {
            return node;
        }

        int mid = l + (r - l) / 2;
        left[node] = build(l, mid);
        right[node] = build(mid + 1, r);
```

```
    return node;
}

/**
 * 更新线段树，创建新版本
 * @param preRoot 旧版本根节点索引
 * @param l 当前区间左端点
 * @param r 当前区间右端点
 * @param pos 更新位置
 * @param val 更新值（通常为1或-1）
 * @return 新版本根节点索引
 */
int update(int preRoot, int l, int r, int pos, int val) {
    int newNode = idx++;
    left.push_back(left[preRoot]);
    right.push_back(right[preRoot]);
    count.push_back(count[preRoot] + val);

    if (l == r) {
        return newNode;
    }

    int mid = l + (r - l) / 2;
    if (pos <= mid) {
        left[newNode] = update(left[preRoot], l, mid, pos, val);
    } else {
        right[newNode] = update(right[preRoot], mid + 1, r, pos, val);
    }
    return newNode;
}

/**
 * 查询区间第K小元素
 * @param rootL 左边界版本根节点
 * @param rootR 右边界版本根节点
 * @param l 当前区间左端点
 * @param r 当前区间右端点
 * @param k 要查询的排名
 * @return 第K小的元素离散化后的值
 */
int queryKth(int rootL, int rootR, int l, int r, int k) {
    if (l == r) return l;
```

```
        int mid = l + (r - l) / 2;
        int leftCount = count[left[rootR]] - count[left[rootL]];

        if (k <= leftCount) {
            return queryKth(left[rootL], left[rootR], l, mid, k);
        } else {
            return queryKth(right[rootL], right[rootR], mid + 1, r, k - leftCount);
        }
    }

    // 获取当前版本号
    int getVersion() const { return version; }
    // 设置当前版本
    void setVersion(int v) { version = v; }
    // 获取指定版本的根节点
    int getRoot(int v) const { return roots[v]; }
    // 设置指定版本的根节点
    void setRoot(int v, int root) { roots[v] = root; }
};

/**
 * 主函数：测试主席树实现静态区间第K小
 */
int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);

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

    vector<int> a(n + 1);
    vector<int> sorted(n);

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

    // 离散化处理
    sort(sorted.begin(), sorted.end());
    sorted.erase(unique(sorted.begin(), sorted.end()), sorted.end());

    // 构建离散化映射
```

```
    map<int, int> valueToRank;
    for (int i = 0; i < sorted.size(); ++i) {
        valueToRank[sorted[i]] = i + 1;  // 排名从1开始
    }

    // 创建并构建主席树
    PersistentSegmentTree tree(n);
    tree.setRoot(0, tree.build(1, sorted.size()));

    // 建立前缀权值线段树
    for (int i = 1; i <= n; ++i) {
        int rank = valueToRank[a[i]];
        tree.setRoot(i, tree.update(tree.getRoot(i - 1), 1, sorted.size(), rank, 1));
    }

    // 处理查询
    for (int i = 0; i < m; ++i) {
        int l, r, k;
        cin >> l >> r >> k;

        int rank = tree.queryKth(tree.getRoot(l - 1), tree.getRoot(r), 1, sorted.size(), k);
        cout << sorted[rank - 1] << '\n';  // 将排名转换回原始值
    }

    return 0;
}
```

4.3 Python实现

```python
"""
可持久化线段树（主席树）Python实现
支持静态区间第K小查询
"""

import sys
from bisect import bisect_left

class PersistentSegmentTree:
    def __init__(self, n):
        """
        初始化可持久化线段树
```

```
    参数:
        n: 原始数组长度
    """
    # 使用列表存储节点信息，避免对象创建开销
    self.left = [0]   # 左子节点索引
    self.right = [0]  # 右子节点索引
    self.count = [0]  # 区间内元素个数
    self.roots = [0] * (n + 2)  # 存储每个版本的根节点
    self.idx = 1      # 当前节点索引
    self.version = 0  # 当前版本号

def build(self, l, r):
    """
    构建初始线段树

    参数:
        l: 当前区间左端点
        r: 当前区间右端点

    返回:
        新建节点的索引
    """
    node = self.idx
    self.idx += 1
    self.left.append(0)
    self.right.append(0)
    self.count.append(0)

    if l == r:
        return node

    mid = l + (r - l) // 2
    self.left[node] = self.build(l, mid)
    self.right[node] = self.build(mid + 1, r)
    return node

def update(self, pre_root, l, r, pos, val):
    """
    更新线段树，创建新版本

    参数:
        pre_root: 旧版本根节点索引
```

```
        l: 当前区间左端点
        r: 当前区间右端点
        pos: 更新位置
        val: 更新值（通常为1或-1）

    返回:
        新版本根节点索引
    """
    new_node = self.idx
    self.idx += 1
    # 复制旧节点的信息
    self.left.append(self.left[pre_root])
    self.right.append(self.right[pre_root])
    self.count.append(self.count[pre_root] + val)

    if l == r:
        return new_node

    mid = l + (r - l) // 2
    if pos <= mid:
        self.left[new_node] = self.update(self.left[pre_root], l, mid, pos, val)
    else:
        self.right[new_node] = self.update(self.right[pre_root], mid + 1, r, pos, val)
    return new_node

def query_kth(self, root_l, root_r, l, r, k):
    """
    查询区间第K小元素

    参数:
        root_l: 左边界版本根节点
        root_r: 右边界版本根节点
        l: 当前区间左端点
        r: 当前区间右端点
        k: 要查询的排名

    返回:
        第K小的元素离散化后的值
    """
    if l == r:
        return l

    mid = l + (r - l) // 2
```

```
        # 计算左子树中区间[l..r]的元素个数
        left_count = self.count[self.left[root_r]] - self.count[self.left[root_l]]

        if k <= left_count:
            # 第k小在左子树
            return self.query_kth(self.left[root_l], self.left[root_r], l, mid, k)
        else:
            # 第k小在右子树
            return self.query_kth(self.right[root_l], self.right[root_r], mid + 1, r, k - left_count)

    # 获取当前版本号
    def get_version(self):
        return self.version

    # 设置当前版本
    def set_version(self, version):
        self.version = version

    # 获取指定版本的根节点
    def get_root(self, version):
        return self.roots[version]

    # 设置指定版本的根节点
    def set_root(self, version, root):
        self.roots[version] = root

# 测试函数
def main():
    input = sys.stdin.read().split()
    ptr = 0
    n = int(input[ptr])
    ptr += 1
    m = int(input[ptr])
    ptr += 1

    a = [0] * (n + 1)  # 原始数组
    sorted_values = []  # 用于离散化

    for i in range(1, n + 1):
        a[i] = int(input[ptr])
        ptr += 1
        sorted_values.append(a[i])
```

```
    # 离散化处理
    sorted_values = sorted(list(set(sorted_values)))

    # 建立离散化映射（使用 bisect 提高效率）
    def get_rank(x):
        return bisect_left(sorted_values, x) + 1  # 排名从 1 开始

    # 创建并构建主席树
    tree = PersistentSegmentTree(n)
    tree.set_root(0, tree.build(1, len(sorted_values)))

    # 建立前缀权值线段树
    for i in range(1, n + 1):
        rank = get_rank(a[i])
        tree.set_root(i, tree.update(tree.get_root(i - 1), 1, len(sorted_values), rank, 1))

    # 处理查询
    for _ in range(m):
        l = int(input[ptr])
        ptr += 1
        r = int(input[ptr])
        ptr += 1
        k = int(input[ptr])
        ptr += 1

        rank = tree.query_kth(tree.get_root(l - 1), tree.get_root(r), 1, len(sorted_values), k)
        print(sorted_values[rank - 1])  # 将排名转换回原始值

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

5. 算法实现要点

5.1 建树过程

```java
// 构建空线段树
static int build(int l, int r) {
    int rt = ++cnt;
    sum[rt] = 0;
    if (l < r) {
        int mid = (l + r) / 2;
```

```
        left[rt] = build(l, mid);
        right[rt] = build(mid + 1, r);
    }
    return rt;
}
```

5.2 插入操作
```java
// 在线段树中插入一个值
static int insert(int pos, int l, int r, int pre) {
    int rt = ++cnt;
    left[rt] = left[pre];
    right[rt] = right[pre];
    sum[rt] = sum[pre] + 1;

    if (l < r) {
        int mid = (l + r) / 2;
        if (pos <= mid) {
            left[rt] = insert(pos, l, mid, left[rt]);
        } else {
            right[rt] = insert(pos, mid + 1, r, right[rt]);
        }
    }
    return rt;
}
```

5.3 查询操作
```java
// 查询区间第 k 小的数
static int query(int k, int l, int r, int u, int v) {
    if (l >= r) return l;
    int mid = (l + r) / 2;
    // 计算左子树中数的个数
    int x = sum[left[v]] - sum[left[u]];
    if (x >= k) {
        // 第 k 小在左子树中
        return query(k, l, mid, left[u], left[v]);
    } else {
        // 第 k 小在右子树中
        return query(k - x, mid + 1, r, right[u], right[v]);
    }
```

```
}
```

5.4 区间 Mex 查询
```java
// 查询区间 Mex
static int queryMex(int l, int r, int u, int v) {
    if (l == r) return l;
    int mid = (l + r) / 2;
    // 计算左子树中数的个数
    int leftCount = sum[left[v]] - sum[left[u]];
    // 如果左子树中数的个数等于区间长度，说明左子树满
    if (leftCount == mid - l + 1) {
        // Mex 在右子树中
        return queryMex(mid + 1, r, right[u], right[v]);
    } else {
        // Mex 在左子树中
        return queryMex(l, mid, left[u], left[v]);
    }
}
```

6. 扩展题目列表（40 题）

6.1 基础静态区间查询类

6.1.1 静态区间第 K 小
- **SPOJ MKTHNUM**：静态区间第 K 小（主席树模板题）
  - 题目链接：https://www.spoj.com/problems/MKTHNUM/
  - 时间复杂度：O(n log n + m log n)
  - 空间复杂度：O(n log n)

- **Luogu P3834**：【模板】可持久化线段树 2
  - 题目链接：https://www.luogu.com.cn/problem/P3834
  - 时间复杂度：O(n log n + m log n)
  - 空间复杂度：O(n log n)

- **LeetCode 218. The Skyline Problem**：天际线问题（可转化为区间查询）
  - 题目链接：https://leetcode.com/problems/the-skyline-problem/
  - 时间复杂度：O(n log n)
  - 空间复杂度：O(n)

6.1.2 区间不同元素个数

- **SPOJ DQUERY**：查询区间不同数字的个数
  - 题目链接：https://www.spoj.com/problems/DQUERY/
  - 时间复杂度：O(n log n + m log n)
  - 空间复杂度：O(n log n)

- **Luogu P1972**：[SDOI2009] HH 的项链
  - 题目链接：https://www.luogu.com.cn/problem/P1972
  - 时间复杂度：O(n log n + m log n)
  - 空间复杂度：O(n log n)

- **AtCoder ABC127 F - Absolute Minima**：绝对最小值问题
  - 题目链接：https://atcoder.jp/contests/abc127/tasks/abc127_f
  - 时间复杂度：O(n log n)
  - 空间复杂度：O(n log n)

6.2 树上路径查询类

6.2.1 树上路径第 K 小
- **SPOJ COT**：Count on a tree（树上路径第 K 小）
  - 题目链接：https://www.spoj.com/problems/COT/
  - 时间复杂度：O((n + m) log n)
  - 空间复杂度：O(n log n)

- **Luogu P2633**：Count on a tree
  - 题目链接：https://www.luogu.com.cn/problem/P2633
  - 时间复杂度：O((n + m) log n)
  - 空间复杂度：O(n log n)

- **HDU 2665**：Kth number（树上路径第 K 小变种）
  - 题目链接：http://acm.hdu.edu.cn/showproblem.php?pid=2665
  - 时间复杂度：O(n log n + m log n)
  - 空间复杂度：O(n log n)

6.2.2 子树查询
- **Luogu P3899**：[湖南集训]谈笑风生
  - 题目链接：https://www.luogu.com.cn/problem/P3899
  - 时间复杂度：O(n log n + m log n)
  - 空间复杂度：O(n log n)

- **LeetCode 1970**：Smallest Missing Genetic Value in Each Subtree
  - 题目链接：https://leetcode.com/problems/smallest-missing-genetic-value-in-each-subtree/
  - 时间复杂度：O(n log n)
  - 空间复杂度：O(n log n)

- **CodeForces 1009F**：Dominant Indices
  - 题目链接：https://codeforces.com/contest/1009/problem/F
  - 时间复杂度：O(n log n)
  - 空间复杂度：O(n log n)

6.3 区间 Mex 查询类

- **Luogu P4137**：Mex（查询区间内未出现的最小自然数）
  - 题目链接：https://www.luogu.com.cn/problem/P4137
  - 时间复杂度：O(n log n + m log n)
  - 空间复杂度：O(n log n)

- **CodeChef MEXUM**：Maximum Mex
  - 题目链接：https://www.codechef.com/problems/MEXUM
  - 时间复杂度：O(n log n)
  - 空间复杂度：O(n log n)

- **HackerEarth Missing Number**：缺失的数
  - 题目链接：https://www.hackerearth.com/practice/data-structures/advanced-data-structures/segment-trees/practice-problems/
  - 时间复杂度：O(n log n + q log n)
  - 空间复杂度：O(n log n)

6.4 动态区间查询类

6.4.1 动态区间第 K 小
- **Luogu P2617**：Dynamic Rankings（动态区间第 K 小）
  - 题目链接：https://www.luogu.com.cn/problem/P2617
  - 时间复杂度：O(n $\log^2 n$ + m $\log^2 n$)
  - 空间复杂度：O(n $\log^2 n$)

- **ZOJ 2112**：Dynamic Rankings
  - 题目链接：http://acm.zju.edu.cn/onlinejudge/showProblem.do?problemId=2112
  - 时间复杂度：O(n $\log^2 n$ + m $\log^2 n$)
  - 空间复杂度：O(n $\log^2 n$)

- **HDU 2665**：Kth number（动态区间第 K 小）
  - 题目链接：http://acm.hdu.edu.cn/showproblem.php?pid=2665
  - 时间复杂度：O(n $\log^2 n$ + m $\log^2 n$)
  - 空间复杂度：O(n $\log^2 n$)

6.4.2 区间修改与查询

- **SPOJ TTM**：区间修改，区间查询，带历史版本
  - 题目链接：https://www.spoj.com/problems/TTM/
  - 时间复杂度：O(n log n + m log n)
  - 空间复杂度：O(n log n)

- **CodeForces 341E**：Candies Game
  - 题目链接：https://codeforces.com/problemset/problem/341/E
  - 时间复杂度：O(n log n + m log n)
  - 空间复杂度：O(n log n)

- **POJ 2763**：Housewife Wind
  - 题目链接：http://poj.org/problem?id=2763
  - 时间复杂度：O(n log n + m log n)
  - 空间复杂度：O(n log n)

6.5 高级应用类

6.5.1 二分答案结合
- **Luogu P2839**：Middle（浮动区间的最大上中位数）
  - 题目链接：https://www.luogu.com.cn/problem/P2839
  - 时间复杂度：O(n log²n + m log²n)
  - 空间复杂度：O(n log n)

- **Luogu P4587**：[FJOI2016]神秘数
  - 题目链接：https://www.luogu.com.cn/problem/P4587
  - 时间复杂度：O(n log²n + m log²n)
  - 空间复杂度：O(n log n)

- **CodeForces 813E**：Army Creation（军队创建）
  - 题目链接：https://codeforces.com/contest/813/problem/E
  - 时间复杂度：O(n log n + m log n)
  - 空间复杂度：O(n log n)

6.5.2 二维区间查询
- **Luogu P2468**：[SDOI2010]粟粟的书架（二维矩阵区间查询）
  - 题目链接：https://www.luogu.com.cn/problem/P2468
  - 时间复杂度：O(n log n + m log²n)
  - 空间复杂度：O(n log n)

- **HackerRank Matrix Queries**：矩阵查询
  - 题目链接：https://www.hackerrank.com/challenges/matrix-queries/problem
  - 时间复杂度：O(n² log n + q log n)
  - 空间复杂度：O(n² log n)

- **POJ 2104**：K-th Number（二维扩展）
  - 题目链接：http://poj.org/problem?id=2104
  - 时间复杂度：O(n² log n + q log n)
  - 空间复杂度：O(n² log n)

6.5.3 持久化数据结构
- **Luogu P3919**：【模板】可持久化数组
  - 题目链接：https://www.luogu.com.cn/problem/P3919
  - 时间复杂度：O(n log n + m log n)
  - 空间复杂度：O(n log n)

- **CodeForces 707D**：Persistent Bookcase（持久化书架）
  - 题目链接：https://codeforces.com/contest/707/problem/D
  - 时间复杂度：O(n log n + m log n)
  - 空间复杂度：O(n log n)

- **LeetCode 2276**：Count Integers in Intervals（动态开点线段树）
  - 题目链接：https://leetcode.com/problems/count-integers-in-intervals/
  - 时间复杂度：O(n log C)
  - 空间复杂度：O(n log C)

6.6 其他应用类

- **HDU 5919**：Sequence II（第一次出现位置的序列）
  - 题目链接：http://acm.hdu.edu.cn/showproblem.php?pid=5919
  - 时间复杂度：O(n log n + m log n)
  - 空间复杂度：O(n log n)

- **CodeForces 441E**：Subset Sums（动态维护子集和）
  - 题目链接：https://codeforces.com/contest/441/problem/E
  - 时间复杂度：O(n log n)
  - 空间复杂度：O(n log n)

- **SPOJ KQUERY**：查询区间内大于 k 的元素个数
  - 题目链接：https://www.spoj.com/problems/KQUERY/
  - 时间复杂度：O(n log n + q log n)
  - 空间复杂度：O(n log n)

- **UVa 11991**：Easy Problem from Rujia Liu?
  - 题目链接：
https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&category=24&page=show_problem&problem=3142

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

- **AizuOJ DSL_2_F**：Range Update Query (RUQ)
  - 题目链接：http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=DSL_2_F
  - 时间复杂度：O(n + m log n)
  - 空间复杂度：O(n)

- **MarsCode 1001**：区间查询问题
  - 题目链接：https://www.mars.codeforces.com/contest/1001
  - 时间复杂度：O(n log n + m log n)
  - 空间复杂度：O(n log n)

- **牛客 NC14542**：区间第K小
  - 题目链接：https://ac.nowcoder.com/acm/problem/14542
  - 时间复杂度：O(n log n + m log n)
  - 空间复杂度：O(n log n)

- **ACWing 241**：楼兰图腾
  - 题目链接：https://www.acwing.com/problem/content/243/
  - 时间复杂度：O(n log n)
  - 空间复杂度：O(n log n)

- **计蒜客 T1467**：区间查询
  - 题目链接：https://nanti.jisuanke.com/t/T1467
  - 时间复杂度：O(n log n + m log n)
  - 空间复杂度：O(n log n)

- **LOJ 111**：樱花查询
  - 题目链接：https://loj.ac/p/111
  - 时间复杂度：O(n log n + m log n)
  - 空间复杂度：O(n log n)

- **USACO 2013 Open**：Photo
  - 题目链接：https://usaco.org/index.php?page=viewproblem2&cpid=281
  - 时间复杂度：O(n log n)
  - 空间复杂度：O(n log n)

- **POJ 3764**：The xor-longest Path
  - 题目链接：http://poj.org/problem?id=3764
  - 时间复杂度：O(n log M)，M是数值范围
  - 空间复杂度：O(n log M)

5. 算法实现要点

5.1 建树过程
```java
// 构建空线段树
static int build(int l, int r) {
    int rt = ++cnt;
    sum[rt] = 0;
    if (l < r) {
        int mid = (l + r) / 2;
        left[rt] = build(l, mid);
        right[rt] = build(mid + 1, r);
    }
    return rt;
}
```

5.2 插入操作
```java
// 在线段树中插入一个值
static int insert(int pos, int l, int r, int pre) {
    int rt = ++cnt;
    left[rt] = left[pre];
    right[rt] = right[pre];
    sum[rt] = sum[pre] + 1;

    if (l < r) {
        int mid = (l + r) / 2;
        if (pos <= mid) {
            left[rt] = insert(pos, l, mid, left[rt]);
        } else {
            right[rt] = insert(pos, mid + 1, r, right[rt]);
        }
    }
    return rt;
}
```

5.3 查询操作
```java
// 查询区间第 k 小的数
static int query(int k, int l, int r, int u, int v) {
    if (l >= r) return l;
```

```
    int mid = (l + r) / 2;
    // 计算左子树中数的个数
    int x = sum[left[v]] - sum[left[u]];
    if (x >= k) {
        // 第k小在左子树中
        return query(k, l, mid, left[u], left[v]);
    } else {
        // 第k小在右子树中
        return query(k - x, mid + 1, r, right[u], right[v]);
    }
}
```

5.4 区间Mex查询
```java
// 查询区间Mex
static int queryMex(int l, int r, int u, int v) {
    if (l == r) return l;
    int mid = (l + r) / 2;
    // 计算左子树中数的个数
    int leftCount = sum[left[v]] - sum[left[u]];
    // 如果左子树中数的个数等于区间长度，说明左子树满
    if (leftCount == mid - l + 1) {
        // Mex在右子树中
        return queryMex(mid + 1, r, right[u], right[v]);
    } else {
        // Mex在左子树中
        return queryMex(l, mid, left[u], left[v]);
    }
}
```

6. 复杂度分析

- **时间复杂度**：
  - 建树：O(n log n)
  - 插入：O(log n)
  - 查询：O(log n)
- **空间复杂度**：O(n log n)

7. 工程化考量

1. **内存优化**：只在需要时创建新节点，共享未修改部分

2. **离散化处理**：对大数据范围进行离散化以节省空间
3. **边界处理**：注意数组下标和边界条件
4. **异常处理**：处理非法输入和查询
5. **性能优化**：
   - 使用内存池避免频繁内存分配
   - 优化常数项，减少不必要的计算
   - 合理选择数据类型避免溢出
6. **可配置性**：
   - 支持自定义比较函数
   - 支持不同类型的查询操作
7. **线程安全**：
   - 对于多线程环境，需要考虑数据竞争问题
   - 可以使用读写锁提高并发性能

8. 优缺点分析

8.1 优点
1. 可以访问历史版本
2. 空间效率较高（相比存储所有版本）
3. 查询效率高
4. 适用于离线和在线查询

8.2 缺点
1. 实现较为复杂
2. 常数较大
3. 空间占用仍然较大
4. 不支持动态修改（基础版本）

9. 扩展应用

1. **树上主席树**：结合 LCA 处理树上路径问题
2. **二维主席树**：处理二维平面上的问题
3. **动态主席树**：结合其他数据结构支持动态修改
4. **整体二分**：结合整体二分处理复杂问题
5. **可持久化数组**：支持历史版本的数组访问
6. **区间 Mex 查询**：查询区间内未出现的最小自然数
7. **区间不同元素个数**：统计区间内不同元素的个数

10. 与机器学习等领域的联系

1. **数据结构优化**：在大规模数据处理中，主席树可以用于优化特征选择和数据采样
2. **在线学习**：主席树的历史版本特性可以用于实现在线学习算法中的模型回溯
3. **推荐系统**：在推荐系统中，可以使用主席树维护用户历史行为的不同版本

11. 调试与测试技巧

1. **小例子测试法**：使用小规模数据手动验证算法正确性
2. **断点式打印**：在关键步骤打印中间结果，验证逻辑正确性
3. **边界场景测试**：测试空输入、极端值、重复数据等边界情况
4. **性能退化排查**：通过性能分析工具定位瓶颈

12. 常见错误与注意事项

1. **数组越界**：注意线段树节点数组的大小，通常需要 4 倍空间
2. **离散化错误**：离散化时要注意去重和映射关系
3. **版本管理错误**：确保正确维护历史版本之间的关系
4. **内存泄漏**：在动态开点的实现中要注意内存释放

13. 总结

可持久化线段树是一种强大的数据结构，特别适用于需要访问历史版本或处理静态区间查询的场景。掌握其核心思想和实现方法对于解决相关问题非常有帮助。在实际应用中，需要根据具体问题选择合适的实现方式，并注意工程化考量。

==========

文件: README.md
==========
持久化线段树（主席树）全面解析

算法介绍

持久化线段树（Persistent Segment Tree），也称为主席树，是一种支持历史版本查询的数据结构。它的核心思想是：

1. **函数式编程思想**：每次更新时创建新的节点，而不是修改原有节点
2. **路径共享**：对于未修改的子树，新版本与旧版本共享节点
3. **前缀和思想**：通过维护前缀版本，可以快速计算区间信息

时间和空间复杂度

- **时间复杂度**：
  - 构建：O(n log n)
  - 单点更新：O(log n)
  - 区间查询：O(log n)
- **空间复杂度**：O(n log n)

应用场景

1. **静态区间第K小问题**：多次查询区间第K小的数
2. **树上路径查询**：查询树上两点之间路径的第K小值
3. **历史版本查询**：查询数组在某个历史时刻的状态
4. **区间不同元素查询**：查询区间内有多少不同的元素
5. **区间Mex查询**：查询区间内最小缺失的自然数
6. **动态范围计数**：支持区间添加和查询计数

经典题目及解决方案

1. SPOJ MKTHNUM - K-th Number

- **题目链接**：https://www.spoj.com/problems/MKTHNUM/
- **题目描述**：给定一个数组，多次查询区间第K小的数
- **解法**：利用持久化线段树维护前缀版本的权值线段树，通过两个版本的差值得到区间信息
- **实现**：
  - Python: `persistent_segment_tree_solutions.py`中的`MKTHNUM`类
  - Java: `PersistentSegmentTreeSolutions.java`中的`MKTHNUM`方法
  - C++: `persistent_segment_tree_solutions.cpp`中的`MKTHNUM`命名空间

2. SPOJ COT - Count on a Tree

- **题目链接**：https://www.spoj.com/problems/COT/
- **题目描述**：给定一棵树，多次查询两点之间路径上的第K小的数
- **解法**：结合LCA算法和树上持久化线段树，利用节点到根的路径信息计算两点路径
- **实现**：
  - Python: `persistent_segment_tree_solutions.py`中的`COT`类
  - Java: `PersistentSegmentTreeSolutions.java`中的`COT`方法
  - C++: `persistent_segment_tree_solutions.cpp`中的`COT`命名空间

3. LeetCode 2276 - Count Integers in Intervals

- **题目链接**：https://leetcode.com/problems/count-integers-in-intervals/
- **题目描述**：实现一个数据结构，支持添加区间和查询区间内整数的个数
- **解法**：使用动态开点线段树高效维护区间覆盖状态
- **实现**：
  - Python: `persistent_segment_tree_solutions.py`中的`CountIntervals`类
  - Java: `PersistentSegmentTreeSolutions.java`中的`CountIntervals`类
  - C++: `persistent_segment_tree_solutions.cpp`中的`CountIntervals`类

4. LeetCode 1970 - Smallest Missing Genetic Value in Each Subtree

- **题目链接**：https://leetcode.com/problems/smallest-missing-genetic-value-in-each-subtree/
- **题目描述**：给定一棵树，每个节点有一个基因值，求每个子树中最小缺失的基因值
- **解法**：利用并查集和 DFS 进行高效查询
- **实现**：
  - Python: `persistent_segment_tree_solutions.py`中的`SmallestMissingGeneticValue`类
  - Java: `PersistentSegmentTreeSolutions.java`中的`smallestMissingGeneticValue`方法
  - C++: `persistent_segment_tree_solutions.cpp`中的`SmallestMissingGeneticValue`命名空间

5. SPOJ DQUERY - D-query

- **题目链接**：https://www.spoj.com/problems/DQUERY/
- **题目描述**：给定一个数组，多次查询区间内不同元素的个数
- **解法**：使用离线处理和树状数组，或者使用持久化线段树维护元素最后一次出现的位置
- **实现**：
  - Python: `persistent_segment_tree_solutions.py`中的`DQUERY`类
  - Java: `PersistentSegmentTreeSolutions.java`中的`DQUERY`方法
  - C++: `persistent_segment_tree_solutions.cpp`中的`DQUERY`命名空间

6. 第一次出现位置序列查询

- **题目描述**：查询区间内第一次出现的元素位置
- **解法**：使用持久化线段树维护元素最后一次出现的位置
- **实现**：
  - Python: `persistent_segment_tree_solutions.py`中的`FirstOccurrence`类
  - Java: `PersistentSegmentTreeSolutions.java`中的`firstOccurrence`方法
  - C++: `persistent_segment_tree_solutions.cpp`中的`FirstOccurrence`命名空间

7. 区间最小缺失自然数查询（区间 Mex 查询）

- **题目描述**：查询区间内最小缺失的自然数
- **解法**：使用持久化线段树维护元素最后一次出现的位置，通过二分查找缺失的最小值
- **实现**：
  - Python: `persistent_segment_tree_solutions.py`中的`RangeMex`类
  - Java: `PersistentSegmentTreeSolutions.java`中的`rangeMex`方法
  - C++: `persistent_segment_tree_solutions.cpp`中的`RangeMex`命名空间

算法实现要点

1. 节点结构设计

每个节点需要包含：
- 左右子节点的索引/指针

- 当前节点维护的信息（计数、最小值等）

2. 动态开点策略

为了节省空间，通常采用动态开点方式，只在需要时创建新节点。

3. 版本管理

维护一个根节点数组，每个元素对应一个历史版本。

4. 离散化处理

对于值域较大的情况，需要进行离散化处理。

工程化考量

1. 内存优化

- 使用预分配的数组存储节点信息（C++）
- 合理估计最大节点数，避免内存溢出

2. 性能优化

- 使用快速 IO（如 C++的`ios::sync_with_stdio(false)`）
- 避免不必要的节点创建
- 合理使用缓存，提高访问效率

3. 异常处理

- 处理边界情况（如空数组、查询超出范围等）
- 处理极端输入（如很大的数据规模）

4. 跨语言实现差异

- C++: 更适合处理大规模数据，需要手动管理内存
- Java: 代码更简洁，自动内存管理，但性能略低
- Python: 实现简单，但对于大规模数据可能性能不足

总结

持久化线段树是一种强大的数据结构，特别适合处理需要历史版本查询的问题。通过合理设计节点结构和更新策略，可以高效地解决各种区间查询问题。在实际应用中，需要根据具体问题选择合适的实现方式，并考虑性能优化和内存管理。

扩展应用

1. **二维主席树**：处理二维平面上的范围查询
2. **树链剖分+主席树**：解决树上路径问题
3. **可持久化线段树合并**：处理子树信息合并问题
4. **离线主席树**：处理离线查询问题

相关资源

- [LeetCode 持久化线段树题目](https://leetcode.com/tag/segment-tree/)
- [SPOJ 经典题目](https://www.spoj.com/problems/tag/persistent-segment-tree)
- [洛谷 主席树专题](https://www.luogu.com.cn/training/1437)

==========

[代码文件]
==========
文件: AtCoder_ABC339G_Java.java
==========

```
package class158;

// Problem: AtCoder ABC339 G - Smaller Sum
// Link: https://atcoder.jp/contests/abc339/tasks/abc339_g
// Description: 给定一个数组，每次在线查询区间[l,r]中不超过 x 的元素和
// Solution: 使用可持久化线段树解决在线区间和查询问题
// Time Complexity: O(nlogn) for preprocessing, O(logn) for each query
// Space Complexity: O(nlogn)

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

public class AtCoder_ABC339G_Java {
    static final int MAXN = 200005;
    static long[] a = new long[MAXN];         // 原始数组
    static long[] b = new long[MAXN];         // 离散化数组
    static int n, q;

    // 主席树节点
    static class Node {
        int l, r;
        long sum;
        Node(int l, int r, long sum) {
```

```
        this.l = l;
        this.r = r;
        this.sum = sum;
    }
}

static Node[] T = new Node[40 * MAXN];    // 主席树节点数组
static int[] root = new int[MAXN];        // 每个版本的根节点
static int cnt = 0;                       // 节点计数器

// 创建新节点
static int createNode(int l, int r, long sum) {
    T[++cnt] = new Node(l, r, sum);
    return cnt;
}

// 插入值到主席树
static int insert(int pre, int l, int r, int val) {
    int now = createNode(0, 0, 0);
    T[now].sum = T[pre].sum + b[val];

    if (l == r) return now;

    int mid = (l + r) >> 1;
    if (val <= mid) {
        T[now].l = insert(T[pre].l, l, mid, val);
        T[now].r = T[pre].r;
    } else {
        T[now].l = T[pre].l;
        T[now].r = insert(T[pre].r, mid + 1, r, val);
    }
    return now;
}

// 查询区间中小于等于 val 的元素和
static long query(int u, int v, int l, int r, int val) {
    if (l == r) {
        if (l <= val) return T[v].sum - T[u].sum;
        else return 0;
    }

    int mid = (l + r) >> 1;
    if (val <= mid) {
```

```
        return query(T[u].l, T[v].l, l, mid, val);
    } else {
        long leftSum = T[T[v].l].sum - T[T[u].l].sum;
        return leftSum + query(T[u].r, T[v].r, mid + 1, r, val);
    }
}

// 离散化
static int getId(long x) {
    return Arrays.binarySearch(b, 1, n + 1, x) + 1;
}

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

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

    String[] line = br.readLine().split(" ");
    for (int i = 1; i <= n; i++) {
        a[i] = Long.parseLong(line[i - 1]);
        b[i] = a[i];
    }

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

    // 构建主席树
    root[0] = createNode(0, 0, 0);
    T[root[0]].l = 0;
    T[root[0]].r = 0;
    T[root[0]].sum = 0;

    for (int i = 1; i <= n; i++) {
        root[i] = insert(root[i - 1], 1, sz, getId(a[i]));
    }
```

```
        q = Integer.parseInt(br.readLine());
        long last_ans = 0;

        for (int i = 0; i < q; i++) {
            line = br.readLine().split(" ");
            long alpha = Long.parseLong(line[0]);
            long beta = Long.parseLong(line[1]);
            long gamma = Long.parseLong(line[2]);

            // 解密
            int l = (int)((alpha ^ last_ans) % n + 1);
            int r = (int)((beta ^ last_ans) % n + 1);
            long x = gamma ^ last_ans;

            if (l > r) {
                int temp = l;
                l = r;
                r = temp;
            }

            last_ans = query(root[l - 1], root[r], 1, sz, getId(x));
            out.println(last_ans);
        }

        out.close();
    }
}
```

==========

文件: AtCoder_ABC339G_Python.py

==========

```
# Problem: AtCoder ABC339 G - Smaller Sum
# Link: https://atcoder.jp/contests/abc339/tasks/abc339_g
# Description: 给定一个数组，每次在线查询区间[l,r]中不超过 x 的元素和
# Solution: 使用可持久化线段树解决在线区间和查询问题
# Time Complexity: O(nlogn) for preprocessing, O(logn) for each query
# Space Complexity: O(nlogn)

import sys

class Node:
    def __init__(self):
```

```
        self.l = 0
        self.r = 0
        self.sum = 0

class PersistentSegmentTree:
    def __init__(self, maxn=200005):
        self.MAXN = maxn
        self.T = [Node() for _ in range(40 * maxn)]
        self.root = [0] * maxn
        self.cnt = 0

    def create_node(self, l=0, r=0, sum=0):
        self.cnt += 1
        self.T[self.cnt].l = l
        self.T[self.cnt].r = r
        self.T[self.cnt].sum = sum
        return self.cnt

    def insert(self, pre, l, r, val, b_val):
        now = self.create_node()
        self.T[now].sum = self.T[pre].sum + b_val

        if l == r:
            return now

        mid = (l + r) >> 1
        if val <= mid:
            self.T[now].l = self.insert(self.T[pre].l, l, mid, val, b_val)
            self.T[now].r = self.T[pre].r
        else:
            self.T[now].l = self.T[pre].l
            self.T[now].r = self.insert(self.T[pre].r, mid + 1, r, val, b_val)
        return now

    def query(self, u, v, l, r, val):
        if l == r:
            if l <= val:
                return self.T[v].sum - self.T[u].sum
            else:
                return 0

        mid = (l + r) >> 1
        if val <= mid:
```

```
            return self.query(self.T[u].l, self.T[v].l, l, mid, val)
        else:
            leftSum = self.T[self.T[v].l].sum - self.T[self.T[u].l].sum
            return leftSum + self.query(self.T[u].r, self.T[v].r, mid + 1, r, val)

def main():
    n = int(sys.stdin.readline())

    line = sys.stdin.readline().split()
    a = [0] * (n + 1)
    b = [0] * (n + 1)

    for i in range(1, n + 1):
        a[i] = int(line[i - 1])
        b[i] = a[i]

    # 离散化
    b = b[1:]  # 去掉索引 0
    b.sort()
    # 去重
    unique_b = []
    for x in b:
        if not unique_b or unique_b[-1] != x:
            unique_b.append(x)
    b = [0] + unique_b  # 加回索引 0
    sz = len(b) - 1

    # 获取值的离散化 ID
    def get_id(x):
        left, right = 1, len(b) - 1
        while left <= right:
            mid = (left + right) // 2
            if b[mid] == x:
                return mid
            elif b[mid] < x:
                left = mid + 1
            else:
                right = mid - 1
        return left

    # 构建主席树
    pst = PersistentSegmentTree(n + 1)
    pst.root[0] = pst.create_node()
```

```
    for i in range(1, n + 1):
        pst.root[i] = pst.insert(pst.root[i - 1], 1, sz, get_id(a[i]), a[i])

    q = int(sys.stdin.readline())
    last_ans = 0

    for _ in range(q):
        line = sys.stdin.readline().split()
        alpha = int(line[0])
        beta = int(line[1])
        gamma = int(line[2])

        # 解密
        l = (alpha ^ last_ans) % n + 1
        r = (beta ^ last_ans) % n + 1
        x = gamma ^ last_ans

        if l > r:
            l, r = r, l

        last_ans = pst.query(pst.root[l - 1], pst.root[r], 1, sz, get_id(x))
        print(last_ans)

if __name__ == "__main__":
    main()

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

文件: CF813E_Java.java
==============================================
package class158;

// Problem: Codeforces 813E - Army Creation
// Link: https://codeforces.com/contest/813/problem/E
// Description: 给定一个数组，每次查询区间[l,r]中最多能选出多少个数，使得每个数出现次数不超过 k
// Solution: 使用可持久化线段树解决区间限制计数问题
// Time Complexity: O(nlogn) for preprocessing, O(logn) for each query
// Space Complexity: O(nlogn)

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

```
public class CF813E_Java {
    static final int MAXN = 100005;
    static int[] a = new int[MAXN];           // 原始数组
    static int[] prev = new int[MAXN];        // 每个元素前 k 次出现的位置
    static int[] last = new int[MAXN];        // 每个值最后出现的位置
    static int n, k;

    // 主席树节点
    static class Node {
        int l, r, sum;
        Node(int l, int r, int sum) {
            this.l = l;
            this.r = r;
            this.sum = sum;
        }
    }

    static Node[] T = new Node[40 * MAXN];    // 主席树节点数组
    static int[] root = new int[MAXN];        // 每个版本的根节点
    static int cnt = 0;                       // 节点计数器

    // 创建新节点
    static int createNode(int l, int r, int sum) {
        T[++cnt] = new Node(l, r, sum);
        return cnt;
    }

    // 插入位置到主席树
    static int insert(int pre, int l, int r, int pos, int val) {
        int now = createNode(0, 0, 0);
        T[now].sum = T[pre].sum + val;

        if (l == r) return now;

        int mid = (l + r) >> 1;
        if (pos <= mid) {
            T[now].l = insert(T[pre].l, l, mid, pos, val);
            T[now].r = T[pre].r;
        } else {
            T[now].l = T[pre].l;
            T[now].r = insert(T[pre].r, mid + 1, r, pos, val);
        }
        return now;
```

```
}

// 查询区间和
static int query(int u, int v, int l, int r, int L, int R) {
    if (L <= l && r <= R) return T[v].sum - T[u].sum;
    if (L > r || R < l) return 0;

    int mid = (l + r) >> 1;
    return query(T[u].l, T[v].l, l, mid, L, R) +
           query(T[u].r, T[v].r, mid + 1, r, L, R);
}

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

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

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

    // 预处理 prev 数组
    Arrays.fill(last, 0);
    Arrays.fill(prev, 0);

    for (int i = 1; i <= n; i++) {
        if (last[a[i]] != 0) {
            prev[i] = last[a[i]];
        }
        last[a[i]] = i;
    }

    // 构建主席树
    root[0] = createNode(0, 0, 0);
    T[root[0]].l = T[root[0]].r = T[root[0]].sum = 0;

    for (int i = 1; i <= n; i++) {
        if (prev[i] == 0) {
            // 第一次出现，在位置 i 加 1
```

```
                root[i] = insert(root[i - 1], 1, n, i, 1);
            } else {
                // 不是第一次出现，先在prev[i]位置减1，再在i位置加1
                int temp = insert(root[i - 1], 1, n, prev[i], -1);
                root[i] = insert(temp, 1, n, i, 1);
            }
        }

        int q = Integer.parseInt(br.readLine());
        int last_ans = 0;

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

            // 解密
            l = (l + last_ans) % n + 1;
            r = (r + last_ans) % n + 1;

            if (l > r) {
                int temp = l;
                l = r;
                r = temp;
            }

            last_ans = query(root[l - 1], root[r], 1, n, l, r);
            out.println(last_ans);
        }

        out.close();
    }
}
```

==========

文件: CF813E_Python.py

==========

```
# Problem: Codeforces 813E - Army Creation
# Link: https://codeforces.com/contest/813/problem/E
# Description: 给定一个数组，每次查询区间[l,r]中最多能选出多少个数，使得每个数出现次数不超过k
# Solution: 使用可持久化线段树解决区间限制计数问题
# Time Complexity: O(nlogn) for preprocessing, O(logn) for each query
```

```
# Space Complexity: O(nlogn)

import sys

class Node:
    def __init__(self):
        self.l = 0
        self.r = 0
        self.sum = 0

class PersistentSegmentTree:
    def __init__(self, maxn=100005):
        self.MAXN = maxn
        self.T = [Node() for _ in range(40 * maxn)]
        self.root = [0] * maxn
        self.cnt = 0

    def create_node(self, l=0, r=0, sum=0):
        self.cnt += 1
        self.T[self.cnt].l = l
        self.T[self.cnt].r = r
        self.T[self.cnt].sum = sum
        return self.cnt

    def insert(self, pre, l, r, pos, val):
        now = self.create_node()
        self.T[now].sum = self.T[pre].sum + val

        if l == r:
            return now

        mid = (l + r) >> 1
        if pos <= mid:
            self.T[now].l = self.insert(self.T[pre].l, l, mid, pos, val)
            self.T[now].r = self.T[pre].r
        else:
            self.T[now].l = self.T[pre].l
            self.T[now].r = self.insert(self.T[pre].r, mid + 1, r, pos, val)
        return now

    def query(self, u, v, l, r, L, R):
        if L <= l and r <= R:
            return self.T[v].sum - self.T[u].sum
```

```
        if L > r or R < l:
            return 0

        mid = (l + r) >> 1
        return self.query(self.T[u].l, self.T[v].l, l, mid, L, R) + \
               self.query(self.T[u].r, self.T[v].r, mid + 1, r, L, R)

def main():
    line = sys.stdin.readline().split()
    n, k = int(line[0]), int(line[1])

    line = sys.stdin.readline().split()
    a = [0] * (n + 1)
    prev = [0] * (n + 1)
    last = [0] * (n + 1)

    for i in range(1, n + 1):
        a[i] = int(line[i - 1])

    # 预处理 prev 数组
    for i in range(1, n + 1):
        if last[a[i]] != 0:
            prev[i] = last[a[i]]
        last[a[i]] = i

    # 构建主席树
    pst = PersistentSegmentTree(n + 1)
    pst.root[0] = pst.create_node()

    for i in range(1, n + 1):
        if prev[i] == 0:
            # 第一次出现，在位置 i 加 1
            pst.root[i] = pst.insert(pst.root[i - 1], 1, n, i, 1)
        else:
            # 不是第一次出现，先在 prev[i]位置减 1，再在 i 位置加 1
            temp = pst.insert(pst.root[i - 1], 1, n, prev[i], -1)
            pst.root[i] = pst.insert(temp, 1, n, i, 1)

    q = int(sys.stdin.readline())
    last_ans = 0

    for _ in range(q):
        line = sys.stdin.readline().split()
```

```
        l = int(line[0])
        r = int(line[1])

        # 解密
        l = (l + last_ans) % n + 1
        r = (r + last_ans) % n + 1

        if l > r:
            l, r = r, l

        last_ans = pst.query(pst.root[l - 1], pst.root[r], 1, n, l, r)
        print(last_ans)

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

==========

文件: Code01_FirstTimeSequence1.java

==========

```
package class158;

/**
 * 第一次出现位置的序列，java版
 *
 * 题目来源：HDU 5919 - Sequence II
 * 题目链接：http://acm.hdu.edu.cn/showproblem.php?pid=5919
 *
 * 题目描述：
 * 给定一个长度为n的数组arr，下标1~n，一共有m条查询，每条查询格式如下
 * l r : arr[l..r]范围上，每个数第一次出现的位置，把这些位置组成一个序列
 *       假设该范围有s种不同的数，那么序列长度为s
 *       打印该序列第s/2个位置(向上取整)，对应arr的什么位置
 *
 * 解题思路：
 * 使用可持久化线段树（主席树）解决该问题。
 * 1. 从右往左建立各个版本的线段树，每个版本维护当前位置到末尾的信息
 * 2. 对于每个位置i，维护从位置i到n中，每种数字第一次出现的位置
 * 3. 通过主席树查询区间内第一次出现位置的第k小值
 *
 * 强制在线处理：
 * 题目有强制在线的要求，上一次打印的答案为lastAns，初始时lastAns = 0
 * 每次给定的l和r，按照如下方式得到真实的l和r，查询完成后更新lastAns
```

```
 * a = (给定 l + lastAns) % n + 1
 * b = (给定 r + lastAns) % n + 1
 * 真实 l = min(a, b)
 * 真实 r = max(a, b)
 *
 * 时间复杂度：O(n log n + m log n)
 * 空间复杂度：O(n log n)
 *
 * 示例：
 * 输入：
 * 1
 * 5 3
 * 1 2 3 2 1
 * 1 5
 * 2 4
 * 3 3
 *
 * 输出：
 * Case #1: 3 2 3
 *
 * 解释：
 * 查询 1 5：[1,2,3,2,1]中，数字 1 第一次出现在位置 1，数字 2 第一次出现在位置 2，数字 3 第一次出现在位置 3
 *         组成序列[1,2,3]，长度为 3，第 2 个位置是 2，对应原数组位置 3
 * 查询 2 4：[2,3,2]中，数字 2 第一次出现在位置 2，数字 3 第一次出现在位置 3
 *         组成序列[2,3]，长度为 2，第 1 个位置是 1，对应原数组位置 2
 * 查询 3 3：[3]中，数字 3 第一次出现在位置 3
 *         组成序列[3]，长度为 1，第 1 个位置是 1，对应原数组位置 3
 */
import java.io.IOException;
import java.util.Arrays;

public class Code01_FirstTimeSequence1 {

    public static int MAXN = 200002;

    public static int MAXT = MAXN * 37;

    public static int cases, n, m;

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

    // pos[v] : v 这个数字上次出现的位置
```

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

// 可持久化线段树需要
public static int[] root = new int[MAXN];

public static int[] left = new int[MAXT];

public static int[] right = new int[MAXT];

// 数组范围上只记录每种数第一次出现的位置，这样的位置有多少个
public static int[] firstSize = new int[MAXT];

public static int cnt;

/**
 * 构建空线段树
 * @param l 区间左端点
 * @param r 区间右端点
 * @return 根节点编号
 */
public static int build(int l, int r) {
    int rt = ++cnt;
    if (l < r) {
        int mid = (l + r) / 2;
        left[rt] = build(l, mid);
        right[rt] = build(mid + 1, r);
    }
    firstSize[rt] = 0;
    return rt;
}

/**
 * 更新线段树节点
 * @param jobi 要更新的位置
 * @param jobv 要增加的值（+1或-1）
 * @param l 当前区间左端点
 * @param r 当前区间右端点
 * @param i 前一个版本的节点编号
 * @return 新版本的根节点编号
 */
public static int update(int jobi, int jobv, int l, int r, int i) {
    int rt = ++cnt;
    left[rt] = left[i];
```

```
    right[rt] = right[i];
    firstSize[rt] = firstSize[i] + jobv;
    if (l == r) {
        return rt;
    }
    int mid = (l + r) / 2;
    if (jobi <= mid) {
        left[rt] = update(jobi, jobv, l, mid, left[rt]);
    } else {
        right[rt] = update(jobi, jobv, mid + 1, r, right[rt]);
    }
    return rt;
}

/**
 * 查询区间内不同数字的个数
 * @param jobl 查询区间左端点
 * @param jobr 查询区间右端点
 * @param l 当前区间左端点
 * @param r 当前区间右端点
 * @param i 当前节点编号
 * @return 区间内不同数字的个数
 */
public static int querySize(int jobl, int jobr, int l, int r, int i) {
    if (jobl <= l && r <= jobr) {
        return firstSize[i];
    }
    int mid = (l + r) / 2;
    int ans = 0;
    if (jobl <= mid) {
        ans += querySize(jobl, jobr, l, mid, left[i]);
    }
    if (jobr > mid) {
        ans += querySize(jobl, jobr, mid + 1, r, right[i]);
    }
    return ans;
}

/**
 * 查询区间内第 jobk 个 1 的位置
 * @param jobk 要查询的排名
 * @param l 当前区间左端点
 * @param r 当前区间右端点
```

```
     * @param i 当前节点编号
     * @return 第 jobk 个 1 的位置
     */
    public static int queryKth(int jobk, int l, int r, int i) {
        if (l == r) {
            return l;
        }
        int mid = (l + r) / 2;
        int lsize = firstSize[left[i]];
        if (lsize >= jobk) {
            return queryKth(jobk, l, mid, left[i]);
        } else {
            return queryKth(jobk - lsize, mid + 1, r, right[i]);
        }
    }

    /**
     * 从右往左建立各个版本的线段树
     * 对于每个位置 i，维护从位置 i 到 n 中，每种数字第一次出现的位置
     */
    public static void prepare() {
        cnt = 0;
        Arrays.fill(pos, 0);
        root[n + 1] = build(1, n);
        for (int i = n; i >= 1; i--) {
            if (pos[arr[i]] == 0) {
                // 数字 arr[i]第一次出现，直接在位置 i 增加计数
                root[i] = update(i, 1, 1, n, root[i + 1]);
            } else {
                // 数字 arr[i]之前出现过，需要先将之前位置的计数减去，再在当前位置增加计数
                root[i] = update(pos[arr[i]], -1, 1, n, root[i + 1]);
                root[i] = update(i, 1, 1, n, root[i]);
            }
            pos[arr[i]] = i;
        }
    }

    public static void main(String[] args) throws IOException {
        ReaderWriter io = new ReaderWriter();
        cases = io.nextInt();
        for (int t = 1; t <= cases; t++) {
            n = io.nextInt();
            m = io.nextInt();
```

```
        for (int i = 1; i <= n; i++) {
            arr[i] = io.nextInt();
        }
        prepare();
        io.write("Case #");
        io.writeInt(t);
        io.write(":");
        for (int i = 1, a, b, l, r, k, lastAns = 0; i <= m; i++) {
            a = (io.nextInt() + lastAns) % n + 1;
            b = (io.nextInt() + lastAns) % n + 1;
            l = Math.min(a, b);
            r = Math.max(a, b);
            // 查询区间[l,r]内不同数字的个数
            k = (querySize(l, r, 1, n, root[l]) + 1) / 2;
            // 查询第 k 个第一次出现位置
            lastAns = queryKth(k, 1, n, root[l]);
            io.write(" ");
            io.writeInt(lastAns);
        }
        io.write("\n");
    }
    io.flush();
}

// 读写工具类
static class ReaderWriter {
    private static final int BUFFER_SIZE = 1 << 9;
    private byte[] inBuf = new byte[BUFFER_SIZE];
    private int bId, bSize;
    private final java.io.InputStream in;

    private byte[] outBuf = new byte[BUFFER_SIZE];
    private int oId;
    private final java.io.OutputStream out;

    ReaderWriter() {
        in = System.in;
        out = System.out;
    }

    private byte read() throws IOException {
        if (bId == bSize) {
            bSize = in.read(inBuf);
```

```
            bId = 0;
            if (bSize == -1)
                return -1;
        }
        return inBuf[bId++];
    }

    public int nextInt() throws IOException {
        int s = 0;
        byte c = read();
        while (c <= ' ') {
            if (c == -1)
                return -1;
            c = read();
        }
        boolean neg = (c == '-');
        if (neg)
            c = read();
        while (c >= '0' && c <= '9') {
            s = s * 10 + (c - '0');
            c = read();
        }
        return neg ? -s : s;
    }

    public void write(String s) throws IOException {
        for (int i = 0; i < s.length(); i++) {
            writeByte((byte) s.charAt(i));
        }
    }

    public void writeInt(int x) throws IOException {
        if (x == 0) {
            writeByte((byte) '0');
            return;
        }
        if (x < 0) {
            writeByte((byte) '-');
            x = -x;
        }
        int len = 0;
        byte[] tmp = new byte[12];
        while (x > 0) {
```

```
                tmp[len++] = (byte) ((x % 10) + '0');
                x /= 10;
            }
            while (len-- > 0) {
                writeByte(tmp[len]);
            }
        }

        private void writeByte(byte b) throws IOException {
            if (oId == BUFFER_SIZE) {
                flush();
            }
            outBuf[oId++] = b;
        }

        public void flush() throws IOException {
            if (oId > 0) {
                out.write(outBuf, 0, oId);
                oId = 0;
            }
        }
    }

}
```

==========

文件: Code01_FirstTimeSequence2.java

==========

```
package class158;

/**
 * 第一次出现位置的序列，C++版
 *
 * 题目来源：HDU 5919 - Sequence II
 * 题目链接：http://acm.hdu.edu.cn/showproblem.php?pid=5919
 *
 * 题目描述：
 * 给定一个长度为n的数组arr，下标1~n，一共有m条查询，每条查询格式如下
 * l r : arr[l..r]范围上，每个数第一次出现的位置，把这些位置组成一个序列
 *       假设该范围有s种不同的数，那么序列长度为s
 *       打印该序列第s/2个位置(向上取整)，对应arr的什么位置
 *
```

```
 * 解题思路：
 * 使用可持久化线段树（主席树）解决该问题。
 * 1. 从右往左建立各个版本的线段树，每个版本维护当前位置到末尾的信息
 * 2. 对于每个位置 i，维护从位置 i 到 n 中，每种数字第一次出现的位置
 * 3. 通过主席树查询区间内第一次出现位置的第 k 小值
 *
 * 强制在线处理：
 * 题目有强制在线的要求，上一次打印的答案为 lastAns，初始时 lastAns = 0
 * 每次给定的 l 和 r，按照如下方式得到真实的 l 和 r，查询完成后更新 lastAns
 * a = (给定 l + lastAns) % n + 1
 * b = (给定 r + lastAns) % n + 1
 * 真实 l = min(a, b)
 * 真实 r = max(a, b)
 *
 * 时间复杂度：O(n log n + m log n)
 * 空间复杂度：O(n log n)
 *
 * 示例：
 * 输入：
 * 1
 * 5 3
 * 1 2 3 2 1
 * 1 5
 * 2 4
 * 3 3
 *
 * 输出：
 * Case #1: 3 2 3
 *
 * 解释：
 * 查询 1 5：[1,2,3,2,1]中，数字 1 第一次出现在位置 1，数字 2 第一次出现在位置 2，数字 3 第一次出现在位置 3
 *          组成序列[1,2,3]，长度为 3，第 2 个位置是 2，对应原数组位置 3
 * 查询 2 4：[2,3,2]中，数字 2 第一次出现在位置 2，数字 3 第一次出现在位置 3
 *          组成序列[2,3]，长度为 2，第 1 个位置是 1，对应原数组位置 2
 * 查询 3 3：[3]中，数字 3 第一次出现在位置 3
 *          组成序列[3]，长度为 1，第 1 个位置是 1，对应原数组位置 3
 *
 * 注意：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 = 200002;
//const int MAXT = MAXN * 37;
//int cases, n, m;
//int arr[MAXN];
//int pos[MAXN];
//int root[MAXN];
//int ls[MAXT];
//int rs[MAXT];
//int firstSize[MAXT];
//int cnt;
//
///**
// * 构建空线段树
// * @param l 区间左端点
// * @param r 区间右端点
// * @return 根节点编号
// */
//int build(int l, int r) {
//    int rt = ++cnt;
//    if (l < r) {
//        int mid = (l + r) / 2;
//        ls[rt] = build(l, mid);
//        rs[rt] = build(mid + 1, r);
//    }
//    firstSize[rt] = 0;
//    return rt;
//}
//
///**
// * 更新线段树节点
// * @param jobi 要更新的位置
// * @param jobv 要增加的值（+1 或-1）
// * @param l 当前区间左端点
// * @param r 当前区间右端点
// * @param i 前一个版本的节点编号
// * @return 新版本的根节点编号
// */
//int update(int jobi, int jobv, int l, int r, int i) {
//    int rt = ++cnt;
//    ls[rt] = ls[i];
//    rs[rt] = rs[i];
```

```
//    firstSize[rt] = firstSize[i] + jobv;
//    if (l == r) {
//        return rt;
//    }
//    int mid = (l + r) / 2;
//    if (jobi <= mid) {
//        ls[rt] = update(jobi, jobv, l, mid, ls[rt]);
//    } else {
//        rs[rt] = update(jobi, jobv, mid + 1, r, rs[rt]);
//    }
//    return rt;
//}
//
///**
// * 查询区间内不同数字的个数
// * @param jobl 查询区间左端点
// * @param jobr 查询区间右端点
// * @param l 当前区间左端点
// * @param r 当前区间右端点
// * @param i 当前节点编号
// * @return 区间内不同数字的个数
// */
//int querySize(int jobl, int jobr, int l, int r, int i) {
//    if (jobl <= l && r <= jobr) {
//        return firstSize[i];
//    }
//    int mid = (l + r) / 2;
//    int ans = 0;
//    if (jobl <= mid) {
//        ans += querySize(jobl, jobr, l, mid, ls[i]);
//    }
//    if (jobr > mid) {
//        ans += querySize(jobl, jobr, mid + 1, r, rs[i]);
//    }
//    return ans;
//}
//
///**
// * 查询区间内第 jobk 个 1 的位置
// * @param jobk 要查询的排名
// * @param l 当前区间左端点
// * @param r 当前区间右端点
// * @param i 当前节点编号
```

```
// * @return 第 jobk 个 1 的位置
// */
//int queryKth(int jobk, int l, int r, int i) {
//    if (l == r) {
//        return l;
//    }
//    int mid = (l + r) / 2;
//    int lsize = firstSize[ls[i]];
//    if (lsize >= jobk) {
//        return queryKth(jobk, l, mid, ls[i]);
//    } else {
//        return queryKth(jobk - lsize, mid + 1, r, rs[i]);
//    }
//}
//
///**
// * 从右往左建立各个版本的线段树
// * 对于每个位置 i，维护从位置 i 到 n 中，每种数字第一次出现的位置
// */
//void prepare() {
//    cnt = 0;
//    memset(pos, 0, sizeof(pos));
//    root[n + 1] = build(1, n);
//    for (int i = n; i >= 1; i--) {
//        if (pos[arr[i]] == 0) {
//            // 数字 arr[i]第一次出现，直接在位置 i 增加计数
//            root[i] = update(i, 1, 1, n, root[i + 1]);
//        } else {
//            // 数字 arr[i]之前出现过，需要先将之前位置的计数减去，再在当前位置增加计数
//            root[i] = update(pos[arr[i]], -1, 1, n, root[i + 1]);
//            root[i] = update(i, 1, 1, n, root[i]);
//        }
//        pos[arr[i]] = i;
//    }
//}
//
//int main() {
//    ios::sync_with_stdio(false);
//    cin.tie(nullptr);
//    cin >> cases;
//    for (int t = 1; t <= cases; t++) {
//        cin >> n >> m;
//        for (int i = 1; i <= n; i++) {
```

```
//            cin >> arr[i];
//        }
//        prepare();
//        cout << "Case #" << t << ":";
//        for (int i = 1, a, b, l, r, k, lastAns = 0; i <= m; i++) {
//            cin >> l >> r;
//            a = (l + lastAns) % n + 1;
//            b = (r + lastAns) % n + 1;
//            l = min(a, b);
//            r = max(a, b);
//            // 查询区间[l,r]内不同数字的个数
//            k = (querySize(l, r, 1, n, root[l]) + 1) / 2;
//            // 查询第 k 个第一次出现位置
//            lastAns = queryKth(k, 1, n, root[l]);
//            cout << " " << lastAns;
//        }
//        cout << "\n";
//    }
//    return 0;
//}
```

==========

文件: Code02_MissingSmallest1.java

==========

```
package class158;

/**
 * 区间内没有出现的最小自然数，java 版
 *
 * 题目来源：洛谷 P4137 - Mex
 * 题目链接：https://www.luogu.com.cn/problem/P4137
 *
 * 题目描述:
 * 给定一个长度为 n 的数组 arr，下标 1~n，一共有 m 条查询
 * 每条查询 l r : 打印 arr[l..r]内没有出现过的最小自然数，注意 0 是自然数
 *
 * 解题思路:
 * 使用可持久化线段树（主席树）解决区间 Mex 问题。
 * 1. 维护每个数字在数组中出现的最晚位置
 * 2. 对于查询区间[l,r]，找到在该区间内没有出现的最小自然数
 * 3. 利用线段树维护数字范围中每个数字出现的最晚位置中的最左位置
 *
```

```
 * 强制在线处理:
 * 请用在线算法解决该问题，因为可以设计强制在线的要求，让离线算法失效
 *
 * 时间复杂度: O(n log n + m log n)
 * 空间复杂度: O(n log n)
 *
 * 示例:
 * 输入:
 * 5 3
 * 0 1 2 3 4
 * 1 3
 * 2 4
 * 1 5
 *
 * 输出:
 * 3
 * 0
 * 5
 *
 * 解释:
 * 查询 1 3：[0,1,2]中没有出现的最小自然数是 3
 * 查询 2 4：[1,2,3]中没有出现的最小自然数是 0
 * 查询 1 5：[0,1,2,3,4]中没有出现的最小自然数是 5
 */
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;
import java.io.PrintWriter;
import java.io.StreamTokenizer;

public class Code02_MissingSmallest1 {

    public static int MAXN = 200001;

    public static int MAXT = MAXN * 22;

    public static int n, m;

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

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

```
public static int[] left = new int[MAXT];

public static int[] right = new int[MAXT];

// 数字范围中，每个数字出现的最晚位置中，最左的位置在哪
public static int[] lateLeft = new int[MAXT];

public static int cnt;

/**
 * 构建空线段树
 * @param l 区间左端点
 * @param r 区间右端点
 * @return 根节点编号
 */
public static int build(int l, int r) {
    int rt = ++cnt;
    lateLeft[rt] = 0;
    if (l < r) {
        int mid = (l + r) / 2;
        left[rt] = build(l, mid);
        right[rt] = build(mid + 1, r);
    }
    return rt;
}

/**
 * 更新线段树节点
 * @param jobi 要更新的数字
 * @param jobv 该数字最晚出现的位置
 * @param l 当前区间左端点
 * @param r 当前区间右端点
 * @param i 前一个版本的节点编号
 * @return 新版本的根节点编号
 */
public static int update(int jobi, int jobv, int l, int r, int i) {
    int rt = ++cnt;
    left[rt] = left[i];
    right[rt] = right[i];
    lateLeft[rt] = lateLeft[i];
    if (l == r) {
        lateLeft[rt] = jobv;
    } else {
```

```
        int mid = (l + r) / 2;
        if (jobi <= mid) {
            left[rt] = update(jobi, jobv, l, mid, left[rt]);
        } else {
            right[rt] = update(jobi, jobv, mid + 1, r, right[rt]);
        }
        lateLeft[rt] = Math.min(lateLeft[left[rt]], lateLeft[right[rt]]);
    }
    return rt;
}

/**
 * 查询区间[l,r]内没有出现的最小自然数
 * @param pos 查询区间左端点
 * @param l 当前数字范围左端点
 * @param r 当前数字范围右端点
 * @param i 当前节点编号
 * @return 没有出现的最小自然数
 */
public static int query(int pos, int l, int r, int i) {
    if (l == r) {
        return l;
    }
    int mid = (l + r) / 2;
    if (lateLeft[left[i]] < pos) {
        // l...mid 范围上，每个数字最晚出现的位置中
        // 最左的位置如果在 pos 以左，说明 l...mid 范围上，一定有缺失的数字
        return query(pos, l, mid, left[i]);
    } else {
        // 缺失的数字一定在 mid+1....r 范围
        // 因为 l...r 一定有缺失的数字才会来到这个范围的
        // 如果左侧不缺失，那缺失的数字一定在右侧范围上
        return query(pos, mid + 1, r, right[i]);
    }
}

/**
 * 预处理，建立主席树
 */
public static void prepare() {
    cnt = 0;
    root[0] = build(0, n);
    for (int i = 1; i <= n; i++) {
```

```
            if (arr[i] > n || arr[i] < 0) {
                // 如果数字超出范围，则直接复制前一个版本
                root[i] = root[i - 1];
            } else {
                // 更新数字 arr[i]的最晚出现位置为 i
                root[i] = update(arr[i], i, 0, n, root[i - 1]);
            }
        }
    }

    public static void main(String[] args) throws IOException {
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        StreamTokenizer in = new StreamTokenizer(br);
        PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));
        in.nextToken();
        n = (int) in.nval;
        in.nextToken();
        m = (int) in.nval;
        for (int i = 1; i <= n; i++) {
            in.nextToken();
            arr[i] = (int) in.nval;
        }
        prepare();
        for (int i = 1, l, r; i <= m; i++) {
            in.nextToken();
            l = (int) in.nval;
            in.nextToken();
            r = (int) in.nval;
            // 查询区间[l,r]内没有出现的最小自然数
            out.println(query(l, 0, n, root[r]));
        }
        out.flush();
        out.close();
        br.close();
    }

}
```

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

文件: Code02_MissingSmallest2.java

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

```
package class158;
```

```
/**
 * 区间内没有出现的最小自然数，C++版
 *
 * 题目来源：洛谷 P4137 - Mex
 * 题目链接：https://www.luogu.com.cn/problem/P4137
 *
 * 题目描述:
 * 给定一个长度为 n 的数组 arr，下标 1~n，一共有 m 条查询
 * 每条查询 l r : 打印 arr[l..r]内没有出现过的最小自然数，注意 0 是自然数
 *
 * 解题思路:
 * 使用可持久化线段树（主席树）解决区间 Mex 问题。
 * 1. 维护每个数字在数组中出现的最晚位置
 * 2. 对于查询区间[l,r]，找到在该区间内没有出现的最小自然数
 * 3. 利用线段树维护数字范围中每个数字出现的最晚位置中的最左位置
 *
 * 强制在线处理:
 * 请用在线算法解决该问题，因为可以设计强制在线的要求，让离线算法失效
 *
 * 时间复杂度: O(n log n + m log n)
 * 空间复杂度: O(n log n)
 *
 * 示例:
 * 输入:
 * 5 3
 * 0 1 2 3 4
 * 1 3
 * 2 4
 * 1 5
 *
 * 输出:
 * 3
 * 0
 * 5
 *
 * 解释:
 * 查询 1 3：[0,1,2]中没有出现的最小自然数是 3
 * 查询 2 4：[1,2,3]中没有出现的最小自然数是 0
 * 查询 1 5：[0,1,2,3,4]中没有出现的最小自然数是 5
 *
 * 注意：如下实现是 C++的版本，C++版本和 java 版本逻辑完全一样
 * 提交如下代码，可以通过所有测试用例
```

```
 */
//#include <bits/stdc++.h>
//
//using namespace std;
//
//const int MAXN = 200001;
//const int MAXT = MAXN * 22;
//int n, m;
//int arr[MAXN];
//int root[MAXN];
//int ls[MAXT];
//int rs[MAXT];
//int lateLeft[MAXT];
//int cnt;
//
///**
// * 构建空线段树
// * @param l 区间左端点
// * @param r 区间右端点
// * @return 根节点编号
// */
//int build(int l, int r) {
//    int rt = ++cnt;
//    lateLeft[rt] = 0;
//    if (l < r) {
//        int mid = (l + r) / 2;
//        ls[rt] = build(l, mid);
//        rs[rt] = build(mid + 1, r);
//    }
//    return rt;
//}
//
///**
// * 更新线段树节点
// * @param jobi 要更新的数字
// * @param jobv 该数字最晚出现的位置
// * @param l 当前区间左端点
// * @param r 当前区间右端点
// * @param i 前一个版本的节点编号
// * @return 新版本的根节点编号
// */
//int update(int jobi, int jobv, int l, int r, int i) {
//    int rt = ++cnt;
```

```
//    ls[rt] = ls[i];
//    rs[rt] = rs[i];
//    lateLeft[rt] = lateLeft[i];
//    if (l == r) {
//        lateLeft[rt] = jobv;
//    } else {
//        int mid = (l + r) / 2;
//        if (jobi <= mid) {
//            ls[rt] = update(jobi, jobv, l, mid, ls[rt]);
//        } else {
//            rs[rt] = update(jobi, jobv, mid + 1, r, rs[rt]);
//        }
//        // 更新当前节点的 lateLeft 值为左右子节点的最小值
//        lateLeft[rt] = min(lateLeft[ls[rt]], lateLeft[rs[rt]]);
//    }
//    return rt;
//}
//
///**
// * 查询区间[l,r]内没有出现的最小自然数
// * @param pos 查询区间左端点
// * @param l 当前数字范围左端点
// * @param r 当前数字范围右端点
// * @param i 当前节点编号
// * @return 没有出现的最小自然数
// */
//int query(int pos, int l, int r, int i) {
//    if (l == r) {
//        return l;
//    }
//    int mid = (l + r) / 2;
//    if (lateLeft[ls[i]] < pos) {
//        // l...mid 范围上，每个数字最晚出现的位置中
//        // 最左的位置如果在 pos 以左，说明 l...mid 范围上，一定有缺失的数字
//        return query(pos, l, mid, ls[i]);
//    } else {
//        // 缺失的数字一定在 mid+1....r 范围
//        // 因为 l...r 一定有缺失的数字才会来到这个范围的
//        // 如果左侧不缺失，那缺失的数字一定在右侧范围上
//        return query(pos, mid + 1, r, rs[i]);
//    }
//}
//
```

```
///**
// * 预处理，建立主席树
// */
//void prepare() {
//    cnt = 0;
//    root[0] = build(0, n);
//    for (int i = 1; i <= n; i++) {
//        if (arr[i] > n || arr[i] < 0) {
//            // 如果数字超出范围，则直接复制前一个版本
//            root[i] = root[i - 1];
//        } else {
//            // 更新数字 arr[i]的最晚出现位置为 i
//            root[i] = update(arr[i], i, 0, n, root[i - 1]);
//        }
//    }
//}
//
//int main() {
//    ios::sync_with_stdio(false);
//    cin.tie(nullptr);
//    cin >> n >> m;
//    for (int i = 1; i <= n; i++) {
//        cin >> arr[i];
//    }
//    prepare();
//    for (int i = 1, l, r; i <= m; i++) {
//        cin >> l >> r;
//        // 查询区间[l,r]内没有出现的最小自然数
//        cout << query(1, 0, n, root[r]) << "\n";
//    }
//    return 0;
//}
```

==========

文件: Code03_LargestUpMedian1.java

==========

```
package class158;

/**
 * 浮动区间的最大上中位数，java 版
 *
 * 题目来源：洛谷 P2839 - Middle
```

```
 * 题目链接：https://www.luogu.com.cn/problem/P2839
 *
 * 题目描述:
 * 为了方便理解，改写题意（与原始题意等效）：
 * 给定一个长度为n的数组arr，下标1~n，一共有m条查询
 * 每条查询 a b c d : 左端点在[a,b]之间、右端点在[c,d]之间，保证a<b<c<d
 *                    哪个区间有最大的上中位数，打印最大的上中位数
 *
 * 解题思路:
 * 使用可持久化线段树（主席树）结合二分答案解决该问题。
 * 1. 对数组元素进行离散化处理
 * 2. 按照元素值从小到大排序，建立主席树
 * 3. 对于每个查询，二分答案，判断是否存在满足条件的区间
 * 4. 利用线段树维护前缀和、前缀最大值、后缀最大值等信息
 *
 * 强制在线处理:
 * 题目有强制在线的要求，上一次打印的答案为lastAns，初始时lastAns = 0
 * 每次给定四个参数，按照如下方式得到a、b、c、d，查询完成后更新lastAns
 * (给定的每个参数 + lastAns) % n + 1，得到四个值，从小到大对应a、b、c、d
 *
 * 时间复杂度: O(n log²n + m log²n)
 * 空间复杂度: O(n log n)
 *
 * 示例:
 * 输入:
 * 4
 * 1 2 3 4
 * 1
 * 1 2 3 4
 *
 * 输出:
 * 3
 *
 * 解释:
 * 查询[1,2,3,4]：左端点在[1,2]之间，右端点在[3,4]之间
 * 可能的区间有[1,3],[1,4],[2,3],[2,4]
 * 对应的上中位数分别为2,2,3,3，最大值为3
 */
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;
import java.io.PrintWriter;
```

```
import java.io.StreamTokenizer;
import java.util.Arrays;

public class Code03_LargestUpMedian1 {

    public static int MAXN = 20001;

    public static int MAXT = MAXN * 20;

    public static int INF = 10000001;

    public static int n, m;

    // 原始位置、数值
    public static int[][] arr = new int[MAXN][2];

    // 可持久化线段树需要
    public static int[] root = new int[MAXN];

    public static int[] left = new int[MAXT];

    public static int[] right = new int[MAXT];

    // 区间内最大前缀和，前缀不能为空
    public static int[] pre = new int[MAXT];

    // 区间内最大后缀和，后缀不能为空
    public static int[] suf = new int[MAXT];

    // 区间内累加和
    public static int[] sum = new int[MAXT];

    public static int cnt;

    // 查询的问题，a、b、c、d
    public static int[] ques = new int[4];

    // 收集区间信息，pre、suf、sum
    public static int[] info = new int[3];

    /**
     * 构建空线段树
     * @param l 区间左端点
```

```
     * @param r 区间右端点
     * @return 根节点编号
     */
    public static int build(int l, int r) {
        int rt = ++cnt;
        pre[rt] = suf[rt] = sum[rt] = r - l + 1;
        if (l < r) {
            int mid = (l + r) / 2;
            left[rt] = build(l, mid);
            right[rt] = build(mid + 1, r);
        }
        return rt;
    }

    /**
     * 向上更新节点信息
     * @param i 节点编号
     */
    public static void up(int i) {
        // 最大前缀和 = max(左子树最大前缀和, 左子树和 + 右子树最大前缀和)
        pre[i] = Math.max(pre[left[i]], sum[left[i]] + pre[right[i]]);
        // 最大后缀和 = max(右子树最大后缀和, 右子树和 + 左子树最大后缀和)
        suf[i] = Math.max(suf[right[i]], suf[left[i]] + sum[right[i]]);
        // 区间和 = 左子树和 + 右子树和
        sum[i] = sum[left[i]] + sum[right[i]];
    }

    /**
     * 更新线段树节点
     * @param jobi 要更新的位置
     * @param l 当前区间左端点
     * @param r 当前区间右端点
     * @param i 前一个版本的节点编号
     * @return 新版本的根节点编号
     */
    public static int update(int jobi, int l, int r, int i) {
        int rt = ++cnt;
        left[rt] = left[i];
        right[rt] = right[i];
        pre[rt] = pre[i];
        suf[rt] = suf[i];
        sum[rt] = sum[i];
        if (l == r) {
```

```
        // 将位置 jobi 的值从 1 改为-1
        pre[rt] = suf[rt] = sum[rt] = -1;
    } else {
        int mid = (l + r) / 2;
        if (jobi <= mid) {
            left[rt] = update(jobi, l, mid, left[rt]);
        } else {
            right[rt] = update(jobi, mid + 1, r, right[rt]);
        }
        up(rt);
    }
    return rt;
}

/**
 * 初始化 info 数组
 */
public static void initInfo() {
    info[0] = info[1] = -INF;
    info[2] = 0;
}

/**
 * 合并右侧区间信息
 * @param r 右侧区间节点编号
 */
public static void mergeRight(int r) {
    // 更新最大前缀和
    info[0] = Math.max(info[0], info[2] + pre[r]);
    // 更新最大后缀和
    info[1] = Math.max(suf[r], info[1] + sum[r]);
    // 更新区间和
    info[2] += sum[r];
}

/**
 * 查询区间[jobl,jobr]的信息
 * @param jobl 查询区间左端点
 * @param jobr 查询区间右端点
 * @param l 当前区间左端点
 * @param r 当前区间右端点
 * @param i 当前节点编号
 */
```

```
public static void query(int jobl, int jobr, int l, int r, int i) {
    if (jobl <= l && r <= jobr) {
        mergeRight(i);
    } else {
        int mid = (l + r) / 2;
        if (jobl <= mid) {
            query(jobl, jobr, l, mid, left[i]);
        }
        if (jobr > mid) {
            query(jobl, jobr, mid + 1, r, right[i]);
        }
    }
}

/**
 * 预处理，建立主席树
 */
public static void prepare() {
    // 按照数值从小到大排序
    Arrays.sort(arr, 1, n + 1, (a, b) -> a[1] - b[1]);
    cnt = 0;
    root[1] = build(1, n);
    for (int i = 2; i <= n; i++) {
        // 将位置arr[i-1][0]的值从1改为-1
        root[i] = update(arr[i - 1][0], 1, n, root[i - 1]);
    }
}

/**
 * 检查是否存在满足条件的区间，其上中位数大于等于v
 * @param a 左端点下界
 * @param b 左端点上界
 * @param c 右端点下界
 * @param d 右端点上界
 * @param v 要检查的上中位数值
 * @return 是否存在满足条件的区间
 */
public static boolean check(int a, int b, int c, int d, int v) {
    initInfo();
    // 查询[a,b]区间的信息
    query(a, b, 1, n, root[v]);
    int best = info[1];
    initInfo();
```

```
    // 查询[c,d]区间的信息
    query(c, d, 1, n, root[v]);
    best += info[0];
    if (b + 1 <= c - 1) {
        initInfo();
        // 查询[b+1,c-1]区间的信息
        query(b + 1, c - 1, 1, n, root[v]);
        best += info[2];
    }
    // 如果 best >= 0，说明存在满足条件的区间
    return best >= 0;
}

/**
 * 计算查询[a,b,c,d]的最大上中位数
 * @param a 左端点下界
 * @param b 左端点上界
 * @param c 右端点下界
 * @param d 右端点上界
 * @return 最大上中位数
 */
public static int compute(int a, int b, int c, int d) {
    int left = 1, right = n, mid, ans = 0;
    // 二分答案
    while (left <= right) {
        mid = (left + right) / 2;
        if (check(a, b, c, d, mid)) {
            // 如果存在满足条件的区间，更新答案并继续向右查找
            ans = arr[mid][1];
            left = mid + 1;
        } else {
            // 否则向左查找
            right = mid - 1;
        }
    }
    return ans;
}

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

```
        n = (int) in.nval;
        for (int i = 1; i <= n; i++) {
            arr[i][0] = i;
            in.nextToken();
            arr[i][1] = (int) in.nval;
        }
        prepare();
        in.nextToken();
        m = (int) in.nval;
        for (int i = 1, lastAns = 0; i <= m; i++) {
            in.nextToken();
            ques[0] = ((int) in.nval + lastAns) % n + 1;
            in.nextToken();
            ques[1] = ((int) in.nval + lastAns) % n + 1;
            in.nextToken();
            ques[2] = ((int) in.nval + lastAns) % n + 1;
            in.nextToken();
            ques[3] = ((int) in.nval + lastAns) % n + 1;
            // 对四个值进行排序
            Arrays.sort(ques);
            // 计算最大上中位数
            lastAns = compute(ques[0], ques[1], ques[2], ques[3]);
            out.println(lastAns);
        }
        out.flush();
        out.close();
        br.close();
    }

}
```

==========

文件: Code03_LargestUpMedian2.java

==========

```
package class158;

/**
 * 浮动区间的最大上中位数, C++版
 *
 * 题目来源：洛谷 P2839 - Middle
 * 题目链接：https://www.luogu.com.cn/problem/P2839
 *
```

```
 * 题目描述:
 * 为了方便理解，改写题意（与原始题意等效）：
 * 给定一个长度为 n 的数组 arr，下标 1~n，一共有 m 条查询
 * 每条查询 a b c d : 左端点在[a,b]之间、右端点在[c,d]之间，保证 a<b<c<d
 *                    哪个区间有最大的上中位数，打印最大的上中位数
 *
 * 解题思路:
 * 使用可持久化线段树（主席树）结合二分答案解决该问题。
 * 1. 对数组元素进行离散化处理
 * 2. 按照元素值从小到大排序，建立主席树
 * 3. 对于每个查询，二分答案，判断是否存在满足条件的区间
 * 4. 利用线段树维护前缀和、前缀最大值、后缀最大值等信息
 *
 * 强制在线处理:
 * 题目有强制在线的要求，上一次打印的答案为 lastAns，初始时 lastAns = 0
 * 每次给定四个参数，按照如下方式得到 a、b、c、d，查询完成后更新 lastAns
 * (给定的每个参数 + lastAns) % n + 1，得到四个值，从小到大对应 a、b、c、d
 *
 * 时间复杂度: O(n log²n + m log²n)
 * 空间复杂度: O(n log n)
 *
 * 示例:
 * 输入:
 * 4
 * 1 2 3 4
 * 1
 * 1 2 3 4
 *
 * 输出:
 * 3
 *
 * 解释:
 * 查询[1,2,3,4]：左端点在[1,2]之间，右端点在[3,4]之间
 * 可能的区间有[1,3],[1,4],[2,3],[2,4]
 * 对应的上中位数分别为 2,2,3,3，最大值为 3
 *
 * 注意：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 = 20001;
//const int MAXT = MAXN * 20;
//const int INF = 10000001;
//int n, m;
//vector<pair<int, int>> arr;
//int root[MAXN];
//int ls[MAXT];
//int rs[MAXT];
//int pre[MAXT];
//int suf[MAXT];
//int sum[MAXT];
//int cnt;
//int ques[4], info[3];
//
///**
// * 构建空线段树
// * @param l 区间左端点
// * @param r 区间右端点
// * @return 根节点编号
// */
//int build(int l, int r) {
//    int rt = ++cnt;
//    pre[rt] = suf[rt] = sum[rt] = r - l + 1;
//    if (l < r) {
//        int mid = (l + r) / 2;
//        ls[rt] = build(l, mid);
//        rs[rt] = build(mid + 1, r);
//    }
//    return rt;
//}
//
///**
// * 向上更新节点信息
// * @param i 节点编号
// */
//void up(int i) {
//    // 最大前缀和 = max(左子树最大前缀和, 左子树和 + 右子树最大前缀和)
//    pre[i] = max(pre[ls[i]], sum[ls[i]] + pre[rs[i]]);
//    // 最大后缀和 = max(右子树最大后缀和, 右子树和 + 左子树最大后缀和)
//    suf[i] = max(suf[rs[i]], suf[ls[i]] + sum[rs[i]]);
//    // 区间和 = 左子树和 + 右子树和
//    sum[i] = sum[ls[i]] + sum[rs[i]];
//}
```

```
//
///**
// * 更新线段树节点
// * @param jobi 要更新的位置
// * @param l 当前区间左端点
// * @param r 当前区间右端点
// * @param i 前一个版本的节点编号
// * @return 新版本的根节点编号
// */
//int update(int jobi, int l, int r, int i) {
//    int rt = ++cnt;
//    ls[rt] = ls[i];
//    rs[rt] = rs[i];
//    pre[rt] = pre[i];
//    suf[rt] = suf[i];
//    sum[rt] = sum[i];
//    if (l == r) {
//        // 将位置 jobi 的值从 1 改为-1
//        pre[rt] = suf[rt] = sum[rt] = -1;
//    } else {
//        int mid = (l + r) / 2;
//        if (jobi <= mid) {
//            ls[rt] = update(jobi, l, mid, ls[rt]);
//        } else {
//            rs[rt] = update(jobi, mid + 1, r, rs[rt]);
//        }
//        up(rt);
//    }
//    return rt;
//}
//
///**
// * 初始化 info 数组
// */
//void initInfo() {
//    info[0] = info[1] = -INF;
//    info[2] = 0;
//}
//
///**
// * 合并右侧区间信息
// * @param r 右侧区间节点编号
// */
```

```
//void mergeRight(int r) {
//    // 更新最大前缀和
//    info[0] = max(info[0], info[2] + pre[r]);
//    // 更新最大后缀和
//    info[1] = max(suf[r], info[1] + sum[r]);
//    // 更新区间和
//    info[2] += sum[r];
//}
//
///**
// * 查询区间[jobl,jobr]的信息
// * @param jobl 查询区间左端点
// * @param jobr 查询区间右端点
// * @param l 当前区间左端点
// * @param r 当前区间右端点
// * @param i 当前节点编号
// */
//void query(int jobl, int jobr, int l, int r, int i) {
//    if (jobl <= l && r <= jobr) {
//        mergeRight(i);
//    } else {
//        int mid = (l + r) / 2;
//        if (jobl <= mid) {
//            query(jobl, jobr, l, mid, ls[i]);
//        }
//        if (jobr > mid) {
//            query(jobl, jobr, mid + 1, r, rs[i]);
//        }
//    }
//}
//
///**
// * 预处理，建立主席树
// */
//void prepare() {
//    // 按照数值从小到大排序
//    sort(arr.begin() + 1, arr.end(), [](const pair<int, int>& a, const pair<int, int>& b) {
//        return a.second < b.second;
//    });
//    cnt = 0;
//    root[1] = build(1, n);
//    for (int i = 2; i <= n; i++) {
//        // 将位置 arr[i-1].first 的值从 1 改为-1
```

```
//        root[i] = update(arr[i - 1].first, 1, n, root[i - 1]);
//    }
//}
//
///**
// * 检查是否存在满足条件的区间，其上中位数大于等于 v
// * @param a 左端点下界
// * @param b 左端点上界
// * @param c 右端点下界
// * @param d 右端点上界
// * @param v 要检查的上中位数值
// * @return 是否存在满足条件的区间
// */
//bool check(int a, int b, int c, int d, int v) {
//    initInfo();
//    // 查询[a,b]区间的信息
//    query(a, b, 1, n, root[v]);
//    int best = info[1];
//    initInfo();
//    // 查询[c,d]区间的信息
//    query(c, d, 1, n, root[v]);
//    best += info[0];
//    if (b + 1 <= c - 1) {
//        initInfo();
//        // 查询[b+1,c-1]区间的信息
//        query(b + 1, c - 1, 1, n, root[v]);
//        best += info[2];
//    }
//    // 如果 best >= 0，说明存在满足条件的区间
//    return best >= 0;
//}
//
///**
// * 计算查询[a,b,c,d]的最大上中位数
// * @param a 左端点下界
// * @param b 左端点上界
// * @param c 右端点下界
// * @param d 右端点上界
// * @return 最大上中位数
// */
//int compute(int a, int b, int c, int d) {
//    int left = 1, right = n, mid, ans = 0;
//    // 二分答案
```

```
//    while (left <= right) {
//        mid = (left + right) / 2;
//        if (check(a, b, c, d, mid)) {
//            // 如果存在满足条件的区间，更新答案并继续向右查找
//            ans = arr[mid].second;
//            left = mid + 1;
//        } else {
//            // 否则向左查找
//            right = mid - 1;
//        }
//    }
//    return ans;
//}
//
//int main() {
//    ios::sync_with_stdio(false);
//    cin.tie(nullptr);
//    cin >> n;
//    arr.resize(n + 1);
//    for (int i = 1; i <= n; i++) {
//        arr[i].first = i;
//        cin >> arr[i].second;
//    }
//    prepare();
//    cin >> m;
//    for (int i = 1, lastAns = 0; i <= m; i++) {
//        for (int j = 0; j < 4; j++) {
//            cin >> ques[j];
//            ques[j] = (ques[j] + lastAns) % n + 1;
//        }
//        // 对四个值进行排序
//        sort(ques, ques + 4);
//        // 计算最大上中位数
//        lastAns = compute(ques[0], ques[1], ques[2], ques[3]);
//        cout << lastAns << '\n';
//    }
//    return 0;
//}
```

==========

文件: Code04_CountOnTree1.java

==========

```
package class158;

/**
 * 路径上的第 k 小，java 版
 *
 * 题目来源：洛谷 P2633 - Count on a tree
 * 题目链接：https://www.luogu.com.cn/problem/P2633
 *
 * 题目描述:
 * 有 n 个节点，编号 1~n，每个节点有权值，有 n-1 条边，所有节点组成一棵树
 * 一共有 m 条查询，每条查询 u v k : 打印 u 号点到 v 号点的路径上，第 k 小的点权
 *
 * 解题思路:
 * 使用树上可持久化线段树（树上主席树）结合 LCA 解决该问题。
 * 1. 对节点权值进行离散化处理
 * 2. 通过 DFS 遍历树，为每个节点建立主席树
 * 3. 利用 DFS 序和 LCA 算法计算树上路径信息
 * 4. 对于查询 u 到 v 的路径，利用容斥原理计算路径上的第 k 小值
 *
 * 强制在线处理:
 * 题目有强制在线的要求，上一次打印的答案为 lastAns，初始时 lastAns = 0
 * 每次给定的 u、v、k，按照如下方式得到真实的 u、v、k，查询完成后更新 lastAns
 * 真实 u = 给定 u ^ lastAns
 * 真实 v = 给定 v
 * 真实 k = 给定 k
 *
 * 时间复杂度: O((n + m) log n)
 * 空间复杂度: O(n log n)
 *
 * 示例:
 * 输入:
 * 5 3
 * 1 2 3 4 5
 * 1 2
 * 1 3
 * 2 4
 * 2 5
 * 4 5 2
 * 3 4 3
 * 1 2 1
 *
 * 输出:
 * 3
```

```
 * 4
 * 1
 *
 * 解释:
 * 查询 4 5 2：节点 4 到节点 5 的路径为[4,2,5]，点权为[4,2,5]，第 2 小为 3
 * 查询 3 4 3：节点 3 到节点 4 的路径为[3,1,2,4]，点权为[3,1,2,4]，第 3 小为 4
 * 查询 1 2 1：节点 1 到节点 2 的路径为[1,2]，点权为[1,2]，第 1 小为 1
 */
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;
import java.io.PrintWriter;
import java.io.StreamTokenizer;
import java.util.Arrays;

public class Code04_CountOnTree1 {

    public static int MAXN = 100001;

    public static int MAXH = 20;

    public static int MAXT = MAXN * MAXH;

    public static int n, m, s;

    // 各个节点权值
    public static int[] arr = new int[MAXN];

    // 收集权值排序并且去重做离散化
    public static int[] sorted = new int[MAXN];

    // 链式前向星需要
    public static int[] head = new int[MAXN];

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

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

    public static int cntg = 0;

    // 可持久化线段树需要
    public static int[] root = new int[MAXN];
```

```
public static int[] left = new int[MAXT];

public static int[] right = new int[MAXT];

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

public static int cntt = 0;

// 树上倍增找 lca 需要
public static int[] deep = new int[MAXN];

public static int[][] stjump = new int[MAXN][MAXH];

/**
 * 二分查找数字 num 在 sorted 数组中的位置
 * @param num 要查找的数字
 * @return 数字在 sorted 数组中的位置
 */
public static int kth(int num) {
    int left = 1, right = s, mid;
    while (left <= right) {
        mid = (left + right) / 2;
        if (sorted[mid] == num) {
            return mid;
        } else if (sorted[mid] < num) {
            left = mid + 1;
        } else {
            right = mid - 1;
        }
    }
    return -1;
}

/**
 * 构建空线段树
 * @param l 区间左端点
 * @param r 区间右端点
 * @return 根节点编号
 */
public static int build(int l, int r) {
    int rt = ++cntt;
    size[rt] = 0;
```

```
    if (l < r) {
        int mid = (l + r) / 2;
        left[rt] = build(l, mid);
        right[rt] = build(mid + 1, r);
    }
    return rt;
}

/**
 * 预处理，对节点权值进行离散化
 */
public static void prepare() {
    for (int i = 1; i <= n; i++) {
        sorted[i] = arr[i];
    }
    Arrays.sort(sorted, 1, n + 1);
    s = 1;
    for (int i = 2; i <= n; i++) {
        if (sorted[s] != sorted[i]) {
            sorted[++s] = sorted[i];
        }
    }
    root[0] = build(1, s);
}

/**
 * 添加边
 * @param u 起点
 * @param v 终点
 */
public static void addEdge(int u, int v) {
    next[++cntg] = head[u];
    to[cntg] = v;
    head[u] = cntg;
}

/**
 * 更新线段树节点
 * @param jobi 要更新的位置
 * @param l 当前区间左端点
 * @param r 当前区间右端点
 * @param i 前一个版本的节点编号
 * @return 新版本的根节点编号
```

```
 */
public static int insert(int jobi, int l, int r, int i) {
    int rt = ++cntt;
    left[rt] = left[i];
    right[rt] = right[i];
    size[rt] = size[i] + 1;
    if (l < r) {
        int mid = (l + r) / 2;
        if (jobi <= mid) {
            left[rt] = insert(jobi, l, mid, left[rt]);
        } else {
            right[rt] = insert(jobi, mid + 1, r, right[rt]);
        }
    }
    return rt;
}

/**
 * 查询路径上第 k 小的点权
 * @param jobk 要查询的排名
 * @param l 当前区间左端点
 * @param r 当前区间右端点
 * @param u 节点 u 的根节点
 * @param v 节点 v 的根节点
 * @param lca 节点 u 和 v 的 LCA 的根节点
 * @param lcafa LCA 父节点的根节点
 * @return 第 k 小的点权在离散化数组中的位置
 */
public static int query(int jobk, int l, int r, int u, int v, int lca, int lcafa) {
    if (l == r) {
        return l;
    }
    // 计算左子树中数的个数
    int lsize = size[left[u]] + size[left[v]] - size[left[lca]] - size[left[lcafa]];
    int mid = (l + r) / 2;
    if (lsize >= jobk) {
        return query(jobk, l, mid, left[u], left[v], left[lca], left[lcafa]);
    } else {
        return query(jobk - lsize, mid + 1, r, right[u], right[v], right[lca], right[lcafa]);
    }
}

// 递归版，C++可以通过，java 无法通过，递归会爆栈
```

```
public static void dfs1(int u, int f) {
    root[u] = insert(kth(arr[u]), 1, s, root[f]);
    deep[u] = deep[f] + 1;
    stjump[u][0] = f;
    for (int p = 1; p < MAXH; p++) {
        stjump[u][p] = stjump[stjump[u][p - 1]][p - 1];
    }
    for (int ei = head[u]; ei > 0; ei = next[ei]) {
        if (to[ei] != f) {
            dfs1(to[ei], u);
        }
    }
}

// 迭代版，都可以通过
// 讲解118，详解了从递归版改迭代版
public static int[][] ufe = new int[MAXN][3];

public static int stackSize, u, f, e;

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

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

/**
 * dfs1的迭代版
 */
public static void dfs2() {
    stackSize = 0;
    push(1, 0, -1);
    while (stackSize > 0) {
        pop();
        if (e == -1) {
```

```
            root[u] = insert(kth(arr[u]), 1, s, root[f]);
            deep[u] = deep[f] + 1;
            stjump[u][0] = f;
            for (int p = 1; p < MAXH; p++) {
                stjump[u][p] = stjump[stjump[u][p - 1]][p - 1];
            }
            e = head[u];
        } else {
            e = next[e];
        }
        if (e != 0) {
            push(u, f, e);
            if (to[e] != f) {
                push(to[e], u, -1);
            }
        }
    }
}

/**
 * 计算节点 a 和节点 b 的最近公共祖先(LCA)
 * @param a 节点 a
 * @param b 节点 b
 * @return 节点 a 和节点 b 的 LCA
 */
public static int lca(int a, int b) {
    if (deep[a] < deep[b]) {
        int tmp = a;
        a = b;
        b = tmp;
    }
    // 将 a 提升到与 b 同一深度
    for (int p = MAXH - 1; p >= 0; p--) {
        if (deep[stjump[a][p]] >= deep[b]) {
            a = stjump[a][p];
        }
    }
    if (a == b) {
        return a;
    }
    // 同时提升 a 和 b 直到它们的父节点相同
    for (int p = MAXH - 1; p >= 0; p--) {
        if (stjump[a][p] != stjump[b][p]) {
```

```
            a = stjump[a][p];
            b = stjump[b][p];
        }
    }
    return stjump[a][0];
}

/**
 * 查询节点 u 到节点 v 路径上第 k 小的点权
 * @param u 起点
 * @param v 终点
 * @param k 要查询的排名
 * @return 第 k 小的点权
 */
public static int kth(int u, int v, int k) {
    int lca = lca(u, v);
    int i = query(k, 1, s, root[u], root[v], root[lca], root[stjump[lca][0]]);
    return sorted[i];
}

public static void main(String[] args) throws IOException {
    BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
    StreamTokenizer in = new StreamTokenizer(br);
    PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));
    in.nextToken();
    n = (int) in.nval;
    in.nextToken();
    m = (int) in.nval;
    for (int i = 1; i <= n; i++) {
        in.nextToken();
        arr[i] = (int) in.nval;
    }
    prepare();
    for (int i = 1, u, v; i < n; i++) {
        in.nextToken();
        u = (int) in.nval;
        in.nextToken();
        v = (int) in.nval;
        addEdge(u, v);
        addEdge(v, u);
    }
    dfs2(); // 使用迭代版防止爆栈
    for (int i = 1, u, v, k, lastAns = 0; i <= m; i++) {
```

```
            in.nextToken();
            u = (int) in.nval ^ lastAns;
            in.nextToken();
            v = (int) in.nval;
            in.nextToken();
            k = (int) in.nval;
            lastAns = kth(u, v, k);
            out.println(lastAns);
        }
        out.flush();
        out.close();
        br.close();
    }

}
```

==========

文件: Code04_CountOnTree2.java

==========

```
package class158;

/**
 * 路径上的第 k 小，C++版
 *
 * 题目来源：洛谷 P2633 - Count on a tree
 * 题目链接：https://www.luogu.com.cn/problem/P2633
 *
 * 题目描述:
 * 有 n 个节点，编号 1~n，每个节点有权值，有 n-1 条边，所有节点组成一棵树
 * 一共有 m 条查询，每条查询 u v k : 打印 u 号点到 v 号点的路径上，第 k 小的点权
 *
 * 解题思路:
 * 使用树上可持久化线段树（树上主席树）结合 LCA 解决该问题。
 * 1. 对节点权值进行离散化处理
 * 2. 通过 DFS 遍历树，为每个节点建立主席树
 * 3. 利用 DFS 序和 LCA 算法计算树上路径信息
 * 4. 对于查询 u 到 v 的路径，利用容斥原理计算路径上的第 k 小值
 *
 * 强制在线处理:
 * 题目有强制在线的要求，上一次打印的答案为 lastAns，初始时 lastAns = 0
 * 每次给定的 u、v、k，按照如下方式得到真实的 u、v、k，查询完成后更新 lastAns
 * 真实 u = 给定 u ^ lastAns
```

```
 * 真实 v = 给定 v
 * 真实 k = 给定 k
 *
 * 时间复杂度: O((n + m) log n)
 * 空间复杂度: O(n log n)
 *
 * 示例:
 * 输入:
 * 5 3
 * 1 2 3 4 5
 * 1 2
 * 1 3
 * 2 4
 * 2 5
 * 4 5 2
 * 3 4 3
 * 1 2 1
 *
 * 输出:
 * 3
 * 4
 * 1
 *
 * 解释:
 * 查询 4 5 2：节点 4 到节点 5 的路径为[4,2,5]，点权为[4,2,5]，第 2 小为 3
 * 查询 3 4 3：节点 3 到节点 4 的路径为[3,1,2,4]，点权为[3,1,2,4]，第 3 小为 4
 * 查询 1 2 1：节点 1 到节点 2 的路径为[1,2]，点权为[1,2]，第 1 小为 1
 *
 * 注意：如下实现是 C++的版本，C++版本和 java 版本逻辑完全一样
 * 提交如下代码，可以通过所有测试用例
 */
//#include <bits/stdc++.h>
//
//using namespace std;
//
//const int MAXN = 100001;
//const int MAXH = 20;
//const int MAXT = MAXN * MAXH;
//int n, m, s;
//int arr[MAXN];
//int sorted[MAXN];
//
//int head[MAXN];
```

```
//int to[MAXN << 1];
//int nxt[MAXN << 1];
//int cntg = 0;
//
//int root[MAXN];
//int ls[MAXT];
//int rs[MAXT];
//int siz[MAXT];
//int cntt = 0;
//
//int deep[MAXN];
//int stjump[MAXN][MAXH];
//
///**
// * 二分查找数字 num 在 sorted 数组中的位置
// * @param num 要查找的数字
// * @return 数字在 sorted 数组中的位置
// */
//int kth(int num) {
//    int left = 1, right = s, mid;
//    while (left <= right) {
//        mid = (left + right) / 2;
//        if (sorted[mid] == num) {
//            return mid;
//        } else if (sorted[mid] < num) {
//            left = mid + 1;
//        } else {
//            right = mid - 1;
//        }
//    }
//    return -1;
//}
//
///**
// * 构建空线段树
// * @param l 区间左端点
// * @param r 区间右端点
// * @return 根节点编号
// */
//int build(int l, int r) {
//    int rt = ++cntt;
//    siz[rt] = 0;
//    if (l < r) {
```

```
//        int mid = (l + r) / 2;
//        ls[rt] = build(l, mid);
//        rs[rt] = build(mid + 1, r);
//    }
//    return rt;
//}
//
///**
// * 预处理，对节点权值进行离散化
// */
//void prepare() {
//    for (int i = 1; i <= n; i++) {
//        sorted[i] = arr[i];
//    }
//    sort(sorted + 1, sorted + n + 1);
//    s = 1;
//    for (int i = 2; i <= n; i++) {
//        if (sorted[s] != sorted[i]) {
//            sorted[++s] = sorted[i];
//        }
//    }
//    root[0] = build(1, s);
//}
//
///**
// * 添加边
// * @param u 起点
// * @param v 终点
// */
//void addEdge(int u, int v) {
//    nxt[++cntg] = head[u];
//    to[cntg] = v;
//    head[u] = cntg;
//}
//
///**
// * 更新线段树节点
// * @param jobi 要更新的位置
// * @param l 当前区间左端点
// * @param r 当前区间右端点
// * @param i 前一个版本的节点编号
// * @return 新版本的根节点编号
// */
```

```
//int insert(int jobi, int l, int r, int i) {
//    int rt = ++cntt;
//    ls[rt] = ls[i];
//    rs[rt] = rs[i];
//    siz[rt] = siz[i] + 1;
//    if (l < r) {
//        int mid = (l + r) / 2;
//        if (jobi <= mid) {
//            ls[rt] = insert(jobi, l, mid, ls[rt]);
//        } else {
//            rs[rt] = insert(jobi, mid + 1, r, rs[rt]);
//        }
//    }
//    return rt;
//}
//
///**
// * 查询路径上第 k 小的点权
// * @param jobk 要查询的排名
// * @param l 当前区间左端点
// * @param r 当前区间右端点
// * @param u 节点 u 的根节点
// * @param v 节点 v 的根节点
// * @param lca 节点 u 和 v 的 LCA 的根节点
// * @param lcafa LCA 父节点的根节点
// * @return 第 k 小的点权在离散化数组中的位置
// */
//int query(int jobk, int l, int r, int u, int v, int lca, int lcafa) {
//    if (l == r) {
//        return l;
//    }
//    // 计算左子树中数的个数
//    int lsiz = siz[ls[u]] + siz[ls[v]] - siz[ls[lca]] - siz[ls[lcafa]];
//    int mid = (l + r) / 2;
//    if (lsiz >= jobk) {
//        return query(jobk, l, mid, ls[u], ls[v], ls[lca], ls[lcafa]);
//    } else {
//        return query(jobk - lsiz, mid + 1, r, rs[u], rs[v], rs[lca], rs[lcafa]);
//    }
//}
//
///**
// * DFS 遍历树并构建主席树
```

```
// * @param u 当前节点
// * @param f 父节点
// */
//void dfs(int u, int f) {
//    root[u] = insert(kth(arr[u]), 1, s, root[f]);
//    deep[u] = deep[f] + 1;
//    stjump[u][0] = f;
//    for (int p = 1; p < MAXH; p++) {
//        stjump[u][p] = stjump[stjump[u][p - 1]][p - 1];
//    }
//    for (int ei = head[u]; ei > 0; ei = nxt[ei]) {
//        if (to[ei] != f) {
//            dfs(to[ei], u);
//        }
//    }
//}
//
///**
// * 计算节点 a 和节点 b 的最近公共祖先(LCA)
// * @param a 节点 a
// * @param b 节点 b
// * @return 节点 a 和节点 b 的 LCA
// */
//int lca(int a, int b) {
//    if (deep[a] < deep[b]) {
//        swap(a, b);
//    }
//    // 将 a 提升到与 b 同一深度
//    for (int p = MAXH - 1; p >= 0; p--) {
//        if (deep[stjump[a][p]] >= deep[b]) {
//            a = stjump[a][p];
//        }
//    }
//    if (a == b) {
//        return a;
//    }
//    // 同时提升 a 和 b 直到它们的父节点相同
//    for (int p = MAXH - 1; p >= 0; p--) {
//        if (stjump[a][p] != stjump[b][p]) {
//            a = stjump[a][p];
//            b = stjump[b][p];
//        }
//    }
```

```
//    return stjump[a][0];
//}
//
///**
// * 查询节点 u 到节点 v 路径上第 k 小的点权
// * @param u 起点
// * @param v 终点
// * @param k 要查询的排名
// * @return 第 k 小的点权
// */
//int kth(int u, int v, int k) {
//    int lcaNode = lca(u, v);
//    int i = query(k, 1, s, root[u], root[v], root[lcaNode], root[stjump[lcaNode][0]]);
//    return sorted[i];
//}
//
//int main() {
//    ios::sync_with_stdio(false);
//    cin.tie(nullptr);
//    cin >> n >> m;
//    for (int i = 1; i <= n; i++) {
//        cin >> arr[i];
//    }
//    prepare();
//    for (int i = 1, u, v; i < n; i++) {
//        cin >> u >> v;
//        addEdge(u, v);
//        addEdge(v, u);
//    }
//    dfs(1, 0);
//    for (int i = 1, u, v, k, lastAns = 0; i <= m; i++) {
//        cin >> u >> v >> k;
//        u ^= lastAns;
//        lastAns = kth(u, v, k);
//        cout << lastAns << '\n';
//    }
//    return 0;
//}
```

==========

文件: Code05_MoreImpressive1.java

==========

```
package class158;

/**
 * 更为厉害，java 版
 *
 * 题目来源：洛谷 P3899 - [湖南集训]谈笑风生
 * 题目链接：https://www.luogu.com.cn/problem/P3899
 *
 * 题目描述:
 * 为了方便理解，改写题意（与原始题意等效）：
 * 有 n 个节点，编号 1~n，给定 n-1 条边，连成一棵树，1 号点是树头
 * 如果 x 是 y 的祖先节点，认为"x 比 y 更厉害"
 * 如果 x 到 y 的路径上，边的数量 <= 某个常数，认为"x 和 y 是邻居"
 * 一共有 m 条查询，每条查询 a k : 打印有多少三元组(a, b, c)满足如下规定
 * a、b、c 为三个不同的点；a 和 b 都比 c 厉害；a 和 b 是邻居，路径边的数量 <= 给定的 k
 *
 * 解题思路:
 * 使用可持久化线段树（主席树）结合 DFS 序解决该问题。
 * 1. 通过 DFS 遍历树，计算每个节点的深度和子树大小
 * 2. 利用 DFS 序将树上问题转化为区间问题
 * 3. 对于每个节点，建立主席树维护其子树信息
 * 4. 对于查询节点 a 和常数 k，计算满足条件的三元组数量
 *
 * 时间复杂度: O(n log n + m log n)
 * 空间复杂度: O(n log n)
 *
 * 示例:
 * 输入:
 * 5 2
 * 1 2
 * 1 3
 * 2 4
 * 2 5
 * 1 2
 * 2 1
 *
 * 输出:
 * 6
 * 2
 *
 * 解释:
 * 查询 1 2：节点 1 为根，k=2，满足条件的三元组有 6 个
 * 查询 2 1：节点 2 为根，k=1，满足条件的三元组有 2 个
```

```
 */
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;
import java.io.PrintWriter;
import java.io.StreamTokenizer;

public class Code05_MoreImpressive1 {

    public static int MAXN = 300001;

    public static int MAXT = MAXN * 22;

    public static int n, m, depth;

    // 链式前向星需要
    public static int[] head = new int[MAXN];

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

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

    public static int cntg = 0;

    // 可持久化线段树需要
    public static int[] root = new int[MAXN];

    public static int[] left = new int[MAXT];

    public static int[] right = new int[MAXT];

    public static long[] sum = new long[MAXT];

    public static int cntt = 0;

    // dfs 需要
    // deep[i] : i 号节点的深度
    public static int[] deep = new int[MAXN];

    // size[i] : 以 i 号节点为头的树，有多少个节点
    public static int[] size = new int[MAXN];
```

```
// dfn[i] : i 号节点的 dfn 序号
public static int[] dfn = new int[MAXN];

public static int cntd = 0;

/**
 * 添加边
 * @param u 起点
 * @param v 终点
 */
public static void addEdge(int u, int v) {
    next[++cntg] = head[u];
    to[cntg] = v;
    head[u] = cntg;
}

/**
 * 构建空线段树
 * @param l 区间左端点
 * @param r 区间右端点
 * @return 根节点编号
 */
public static int build(int l, int r) {
    int rt = ++cntt;
    sum[rt] = 0;
    if (l < r) {
        int mid = (l + r) / 2;
        left[rt] = build(l, mid);
        right[rt] = build(mid + 1, r);
    }
    return rt;
}

/**
 * 更新线段树节点
 * @param jobi 要更新的位置
 * @param jobv 要增加的值
 * @param l 当前区间左端点
 * @param r 当前区间右端点
 * @param i 前一个版本的节点编号
 * @return 新版本的根节点编号
 */
public static int add(int jobi, long jobv, int l, int r, int i) {
```

```
    int rt = ++cntt;
    left[rt] = left[i];
    right[rt] = right[i];
    sum[rt] = sum[i] + jobv;
    if (l < r) {
        int mid = (l + r) / 2;
        if (jobi <= mid) {
            left[rt] = add(jobi, jobv, l, mid, left[rt]);
        } else {
            right[rt] = add(jobi, jobv, mid + 1, r, right[rt]);
        }
    }
    return rt;
}

/**
 * 查询区间[jobl,jobr]的和
 * @param jobl 查询区间左端点
 * @param jobr 查询区间右端点
 * @param l 当前区间左端点
 * @param r 当前区间右端点
 * @param u 前一个版本的根节点
 * @param v 当前版本的根节点
 * @return 区间和
 */
public static long query(int jobl, int jobr, int l, int r, int u, int v) {
    if (jobl <= l && r <= jobr) {
        return sum[v] - sum[u];
    }
    long ans = 0;
    int mid = (l + r) / 2;
    if (jobl <= mid) {
        ans += query(jobl, jobr, l, mid, left[u], left[v]);
    }
    if (jobr > mid) {
        ans += query(jobl, jobr, mid + 1, r, right[u], right[v]);
    }
    return ans;
}

// 递归版，C++可以通过，java无法通过，递归会爆栈
public static void dfs1(int u, int f) {
    deep[u] = deep[f] + 1;
```

```
    depth = Math.max(depth, deep[u]);
    size[u] = 1;
    dfn[u] = ++cntd;
    for (int ei = head[u]; ei > 0; ei = next[ei]) {
        if (to[ei] != f) {
            dfs1(to[ei], u);
        }
    }
    for (int ei = head[u]; ei > 0; ei = next[ei]) {
        if (to[ei] != f) {
            size[u] += size[to[ei]];
        }
    }
}

// 递归版，C++可以通过，java无法通过，递归会爆栈
public static void dfs2(int u, int f) {
    root[dfn[u]] = add(deep[u], size[u] - 1, 1, depth, root[dfn[u] - 1]);
    for (int ei = head[u]; ei > 0; ei = next[ei]) {
        if (to[ei] != f) {
            dfs2(to[ei], u);
        }
    }
}

// dfs1、dfs2，分别改成迭代版，dfs3、dfs4
// 讲解118，详解了从递归版改迭代版
public static int[][] ufe = new int[MAXN][3];

public static int stackSize, u, f, e;

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

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

```
}

/**
 * dfs1 的迭代版
 */
public static void dfs3() {
    stackSize = 0;
    push(1, 0, -1);
    while (stackSize > 0) {
        pop();
        if (e == -1) {
            deep[u] = deep[f] + 1;
            depth = Math.max(depth, deep[u]);
            size[u] = 1;
            dfn[u] = ++cntd;
            e = head[u];
        } else {
            e = next[e];
        }
        if (e != 0) {
            push(u, f, e);
            if (to[e] != f) {
                push(to[e], u, -1);
            }
        } else {
            for (int ei = head[u]; ei > 0; ei = next[ei]) {
                if (to[ei] != f) {
                    size[u] += size[to[ei]];
                }
            }
        }
    }
}

/**
 * dfs2 的迭代版
 */
public static void dfs4() {
    stackSize = 0;
    push(1, 0, -1);
    while (stackSize > 0) {
        pop();
        if (e == -1) {
```

```
                root[dfn[u]] = add(deep[u], size[u] - 1, 1, depth, root[dfn[u] - 1]);
                e = head[u];
            } else {
                e = next[e];
            }
            if (e != 0) {
                push(u, f, e);
                if (to[e] != f) {
                    push(to[e], u, -1);
                }
            }
        }
    }

    /**
     * 预处理，建立主席树
     */
    public static void prepare() {
        depth = 0;
        dfs3(); // 使用迭代版防止爆栈
        root[0] = build(1, depth);
        dfs4(); // 使用迭代版防止爆栈
    }

    /**
     * 计算查询 a k 的结果
     * @param a 查询节点
     * @param k 路径长度限制
     * @return 满足条件的三元组数量
     */
    public static long compute(int a, int k) {
        // 计算 a 的子树中深度不超过 deep[a]+k 的节点贡献
        long ans = (long) (size[a] - 1) * Math.min(k, deep[a] - 1);
        // 查询 dfn 序在[dfn[a], dfn[a]+size[a]-1]范围内，深度在[deep[a]+1, deep[a]+k]的节点贡献
        ans += query(deep[a] + 1, deep[a] + k, 1, depth, root[dfn[a] - 1], root[dfn[a] + size[a] -
1]);
        return ans;
    }

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

```
        in.nextToken();
        n = (int) in.nval;
        in.nextToken();
        m = (int) in.nval;
        for (int i = 1, u, v; i < n; i++) {
            in.nextToken();
            u = (int) in.nval;
            in.nextToken();
            v = (int) in.nval;
            addEdge(u, v);
            addEdge(v, u);
        }
        prepare();
        for (int i = 1, a, k; i <= m; i++) {
            in.nextToken();
            a = (int) in.nval;
            in.nextToken();
            k = (int) in.nval;
            out.println(compute(a, k));
        }
        out.flush();
        out.close();
        br.close();
    }

}
```

==========

文件: Code05_MoreImpressive2.java

==========

```
package class158;

/**
 * 更为厉害，C++版
 *
 * 题目来源：洛谷 P3899 - [湖南集训]谈笑风生
 * 题目链接：https://www.luogu.com.cn/problem/P3899
 *
 * 题目描述:
 * 为了方便理解，改写题意（与原始题意等效）：
 * 有n个节点，编号1~n，给定n-1条边，连成一棵树，1号点是树头
 * 如果x是y的祖先节点，认为"x比y更厉害"
```

```
 * 如果 x 到 y 的路径上，边的数量 <= 某个常数，认为"x 和 y 是邻居"
 * 一共有 m 条查询，每条查询 a k : 打印有多少三元组(a, b, c)满足如下规定
 * a、b、c 为三个不同的点；a 和 b 都比 c 厉害；a 和 b 是邻居，路径边的数量 <= 给定的 k
 *
 * 解题思路:
 * 使用可持久化线段树（主席树）结合 DFS 序解决该问题。
 * 1. 通过 DFS 遍历树，计算每个节点的深度和子树大小
 * 2. 利用 DFS 序将树上问题转化为区间问题
 * 3. 对于每个节点，建立主席树维护其子树信息
 * 4. 对于查询节点 a 和常数 k，计算满足条件的三元组数量
 *
 * 时间复杂度: O(n log n + m log n)
 * 空间复杂度: O(n log n)
 *
 * 示例:
 * 输入:
 * 5 2
 * 1 2
 * 1 3
 * 2 4
 * 2 5
 * 1 2
 * 2 1
 *
 * 输出:
 * 6
 * 2
 *
 * 解释:
 * 查询 1 2：节点 1 为根，k=2，满足条件的三元组有 6 个
 * 查询 2 1：节点 2 为根，k=1，满足条件的三元组有 2 个
 *
 * 注意：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;
//const int MAXT = MAXN * 22;
//int n, m, depth;
//
```

```
//int head[MAXN];
//int to[MAXN << 1];
//int nxt[MAXN << 1];
//int cntg = 0;
//
//int root[MAXN];
//int ls[MAXT];
//int rs[MAXT];
//long long sum[MAXT];
//int cntt = 0;
//
//int dep[MAXN];
//int siz[MAXN];
//int dfn[MAXN];
//int cntd = 0;
//
///**
// * 添加边
// * @param u 起点
// * @param v 终点
// */
//void addEdge(int u, int v) {
//    nxt[++cntg] = head[u];
//    to[cntg] = v;
//    head[u] = cntg;
//}
//
///**
// * 构建空线段树
// * @param l 区间左端点
// * @param r 区间右端点
// * @return 根节点编号
// */
//int build(int l, int r) {
//    int rt = ++cntt;
//    sum[rt] = 0LL;
//    if (l < r) {
//        int mid = (l + r) >> 1;
//        ls[rt] = build(l, mid);
//        rs[rt] = build(mid + 1, r);
//    }
//    return rt;
//}
```

```
//
///**
// * 更新线段树节点
// * @param jobi 要更新的位置
// * @param jobv 要增加的值
// * @param l 当前区间左端点
// * @param r 当前区间右端点
// * @param i 前一个版本的节点编号
// * @return 新版本的根节点编号
// */
//int add(int jobi, long long jobv, int l, int r, int i) {
//    int rt = ++cntt;
//    ls[rt] = ls[i];
//    rs[rt] = rs[i];
//    sum[rt] = sum[i] + jobv;
//    if (l < r) {
//        int mid = (l + r) >> 1;
//        if (jobi <= mid) {
//            ls[rt] = add(jobi, jobv, l, mid, ls[rt]);
//        } else {
//            rs[rt] = add(jobi, jobv, mid + 1, r, rs[rt]);
//        }
//    }
//    return rt;
//}
//
///**
// * 查询区间[jobl,jobr]的和
// * @param jobl 查询区间左端点
// * @param jobr 查询区间右端点
// * @param l 当前区间左端点
// * @param r 当前区间右端点
// * @param u 前一个版本的根节点
// * @param v 当前版本的根节点
// * @return 区间和
// */
//long long query(int jobl, int jobr, int l, int r, int u, int v) {
//    if (jobl <= l && r <= jobr) {
//        return sum[v] - sum[u];
//    }
//    long long ans = 0;
//    int mid = (l + r) >> 1;
//    if (jobl <= mid) {
```

```
//        ans += query(jobl, jobr, l, mid, ls[u], ls[v]);
//    }
//    if (jobr > mid) {
//        ans += query(jobl, jobr, mid + 1, r, rs[u], rs[v]);
//    }
//    return ans;
//}
//
///**
// * DFS 遍历树并计算节点信息
// * @param u 当前节点
// * @param f 父节点
// */
//void dfs1(int u, int f) {
//    dep[u] = dep[f] + 1;
//    depth = max(depth, dep[u]);
//    siz[u] = 1;
//    dfn[u] = ++cntd;
//    for (int ei = head[u]; ei > 0; ei = nxt[ei]) {
//        if (to[ei] != f) {
//            dfs1(to[ei], u);
//        }
//    }
//    for (int ei = head[u]; ei > 0; ei = nxt[ei]) {
//        if (to[ei] != f) {
//            siz[u] += siz[to[ei]];
//        }
//    }
//}
//
///**
// * DFS 遍历树并构建主席树
// * @param u 当前节点
// * @param f 父节点
// */
//void dfs2(int u, int f) {
//    root[dfn[u]] = add(dep[u], (long long)siz[u] - 1, 1, depth, root[dfn[u] - 1]);
//    for (int ei = head[u]; ei > 0; ei = nxt[ei]) {
//        if (to[ei] != f) {
//            dfs2(to[ei], u);
//        }
//    }
//}
```

```
//
///**
// * 预处理，建立主席树
// */
//void prepare() {
//    depth = 0;
//    dfs1(1, 0);
//    root[0] = build(1, depth);
//    dfs2(1, 0);
//}
//
///**
// * 计算查询 a k 的结果
// * @param a 查询节点
// * @param k 路径长度限制
// * @return 满足条件的三元组数量
// */
//long long compute(int a, int k) {
//    // 计算 a 的子树中深度不超过 dep[a]+k 的节点贡献
//    long long ans = (long long)(siz[a] - 1) * min(k, dep[a] - 1);
//    // 查询 dfn 序在[dfn[a], dfn[a]+siz[a]-1]范围内，深度在[dep[a]+1, dep[a]+k]的节点贡献
//    ans += query(dep[a] + 1, dep[a] + k, 1, depth, root[dfn[a] - 1], root[dfn[a] + siz[a] -
1]);
//    return ans;
//}
//
//int main() {
//    ios::sync_with_stdio(false);
//    cin.tie(nullptr);
//    cin >> n >> m;
//    for (int i = 1, u, v; i < n; i++) {
//        cin >> u >> v;
//        addEdge(u, v);
//        addEdge(v, u);
//    }
//    prepare();
//    for(int i = 1, a, k; i <= m; i++) {
//        cin >> a >> k;
//        cout << compute(a, k) << "\n";
//    }
//    return 0;
//}
```

==========

文件: CodeChef_CLONEME_Java.java
==========

```
package class158;

// Problem: CodeChef CLONEME - Cloning
// Link: https://www.codechef.com/JUNE17/problems/CLONEME
// Description: 给定一个数组，每次查询两个区间是否可以通过重新排列变得相同
// Solution: 使用可持久化线段树和哈希技术解决区间相等性查询问题
// Time Complexity: O(nlogn) for preprocessing, O(logn) for each query
// Space Complexity: O(nlogn)

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

public class CodeChef_CLONEME_Java {
    static final int MAXN = 100005;
    static int[] a = new int[MAXN];           // 原始数组
    static int[] b = new int[MAXN];           // 离散化数组
    static int n, q;

    // 主席树节点
    static class Node {
        int l, r, sum;
        long hash;
        Node(int l, int r, int sum, long hash) {
            this.l = l;
            this.r = r;
            this.sum = sum;
            this.hash = hash;
        }
    }

    static Node[] T = new Node[40 * MAXN];    // 主席树节点数组
    static int[] root = new int[MAXN];        // 每个版本的根节点
    static int cnt = 0;                       // 节点计数器

    // 创建新节点
    static int createNode(int l, int r, int sum, long hash) {
        T[++cnt] = new Node(l, r, sum, hash);
        return cnt;
    }
```

```
// 插入值到主席树
static int insert(int pre, int l, int r, int val) {
    int now = createNode(0, 0, 0, 0);
    T[now].sum = T[pre].sum + 1;
    T[now].hash = T[pre].hash + (long)val * val;

    if (l == r) return now;

    int mid = (l + r) >> 1;
    if (val <= mid) {
        T[now].l = insert(T[pre].l, l, mid, val);
        T[now].r = T[pre].r;
    } else {
        T[now].l = T[pre].l;
        T[now].r = insert(T[pre].r, mid + 1, r, val);
    }
    return now;
}

// 查询区间信息
static class Result {
    int sum;
    long hash;
    Result(int sum, long hash) {
        this.sum = sum;
        this.hash = hash;
    }
}

static Result query(int u, int v, int l, int r, int L, int R) {
    if (L <= l && r <= R) return new Result(T[v].sum - T[u].sum, T[v].hash - T[u].hash);
    if (L > r || R < l) return new Result(0, 0);

    int mid = (l + r) >> 1;
    Result left = query(T[u].l, T[v].l, l, mid, L, R);
    Result right = query(T[u].r, T[v].r, mid + 1, r, L, R);

    return new Result(left.sum + right.sum, left.hash + right.hash);
}

// 离散化
static int getId(int x) {
```

```
    return Arrays.binarySearch(b, 1, n + 1, x) + 1;
}

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

    int t = Integer.parseInt(br.readLine());

    while (t-- > 0) {
        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++) {
            a[i] = Integer.parseInt(line[i - 1]);
            b[i] = a[i];
        }

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

        // 构建主席树
        cnt = 0; // 重置计数器
        root[0] = createNode(0, 0, 0, 0);
        T[root[0]].l = T[root[0]].r = T[root[0]].sum = 0;
        T[root[0]].hash = 0;

        for (int i = 1; i <= n; i++) {
            root[i] = insert(root[i - 1], 1, sz, getId(a[i]));
        }

        // 处理查询
        for (int i = 0; i < q; i++) {
            line = br.readLine().split(" ");
            int l1 = Integer.parseInt(line[0]);
```

```
                int r1 = Integer.parseInt(line[1]);
                int l2 = Integer.parseInt(line[2]);
                int r2 = Integer.parseInt(line[3]);

                Result res1 = query(root[l1 - 1], root[r1], 1, sz, 1, sz);
                Result res2 = query(root[l2 - 1], root[r2], 1, sz, 1, sz);

                if (res1.sum == res2.sum && res1.hash == res2.hash) {
                    out.println("YES");
                } else {
                    out.println("NO");
                }
            }
        }

        out.close();
    }
}
```

==========

文件: CodeChef_CLONEME_Python.py

==========

```
# Problem: CodeChef CLONEME - Cloning
# Link: https://www.codechef.com/JUNE17/problems/CLONEME
# Description: 给定一个数组，每次查询两个区间是否可以通过重新排列变得相同
# Solution: 使用可持久化线段树和哈希技术解决区间相等性查询问题
# Time Complexity: O(nlogn) for preprocessing, O(logn) for each query
# Space Complexity: O(nlogn)

import sys

class Node:
    def __init__(self):
        self.l = 0
        self.r = 0
        self.sum = 0
        self.hash = 0

class PersistentSegmentTree:
    def __init__(self, maxn=100005):
        self.MAXN = maxn
        self.T = [Node() for _ in range(40 * maxn)]
```

```
        self.root = [0] * maxn
        self.cnt = 0

    def create_node(self, l=0, r=0, sum=0, hash=0):
        self.cnt += 1
        self.T[self.cnt].l = l
        self.T[self.cnt].r = r
        self.T[self.cnt].sum = sum
        self.T[self.cnt].hash = hash
        return self.cnt

    def insert(self, pre, l, r, val):
        now = self.create_node()
        self.T[now].sum = self.T[pre].sum + 1
        self.T[now].hash = self.T[pre].hash + val * val

        if l == r:
            return now

        mid = (l + r) >> 1
        if val <= mid:
            self.T[now].l = self.insert(self.T[pre].l, l, mid, val)
            self.T[now].r = self.T[pre].r
        else:
            self.T[now].l = self.T[pre].l
            self.T[now].r = self.insert(self.T[pre].r, mid + 1, r, val)
        return now

    def query(self, u, v, l, r, L, R):
        if L <= l and r <= R:
            return (self.T[v].sum - self.T[u].sum, self.T[v].hash - self.T[u].hash)
        if L > r or R < l:
            return (0, 0)

        mid = (l + r) >> 1
        left_sum, left_hash = self.query(self.T[u].l, self.T[v].l, l, mid, L, R)
        right_sum, right_hash = self.query(self.T[u].r, self.T[v].r, mid + 1, r, L, R)

        return (left_sum + right_sum, left_hash + right_hash)

def main():
    t = int(sys.stdin.readline())
```

```
for _ in range(t):
    line = sys.stdin.readline().split()
    n, q = int(line[0]), int(line[1])

    line = sys.stdin.readline().split()
    a = [0] * (n + 1)
    b = [0] * (n + 1)

    for i in range(1, n + 1):
        a[i] = int(line[i - 1])
        b[i] = a[i]

    # 离散化
    b = b[1:]  # 去掉索引 0
    b.sort()
    # 去重
    unique_b = []
    for x in b:
        if not unique_b or unique_b[-1] != x:
            unique_b.append(x)
    b = [0] + unique_b  # 加回索引 0
    sz = len(b) - 1

    # 获取值的离散化 ID
    def get_id(x):
        left, right = 1, len(b) - 1
        while left <= right:
            mid = (left + right) // 2
            if b[mid] == x:
                return mid
            elif b[mid] < x:
                left = mid + 1
            else:
                right = mid - 1
        return left

    # 构建主席树
    pst = PersistentSegmentTree(n + 1)
    pst.root[0] = pst.create_node()

    for i in range(1, n + 1):
        pst.root[i] = pst.insert(pst.root[i - 1], 1, sz, get_id(a[i]))
```

```
        # 处理查询
        for _ in range(q):
            line = sys.stdin.readline().split()
            l1, r1, l2, r2 = int(line[0]), int(line[1]), int(line[2]), int(line[3])

            sum1, hash1 = pst.query(pst.root[l1 - 1], pst.root[r1], 1, sz, 1, sz)
            sum2, hash2 = pst.query(pst.root[l2 - 1], pst.root[r2], 1, sz, 1, sz)

            if sum1 == sum2 and hash1 == hash2:
                print("YES")
            else:
                print("NO")

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

==========

文件: LeetCode1970_SmallestMissingGeneticValue.cpp

==========

```
/**
 * LeetCode 1970. Smallest Missing Genetic Value in Each Subtree
 *
 * 题目描述:
 * 给你一棵根节点为 0 的家族树，树中节点从 0 到 n-1 编号，parents[i]是节点 i 的父节点。
 * 每个节点有一个基因值 genes[i]。对于每个节点，求出以其为根的子树中缺失的最小基因值。
 *
 * 解题思路:
 * 使用可持久化线段树（主席树）解决子树 Mex 问题。
 * 1. 通过 DFS 遍历树，为每个节点建立主席树
 * 2. 对于每个节点，将其子树中的所有基因值插入到主席树中
 * 3. 查询 Mex 即为查询区间内未出现的最小自然数
 *
 * 时间复杂度: O(n log n)
 * 空间复杂度: O(n log n)
 *
 * 示例:
 * 输入:
 * parents = [-1,0,0,2], genes = [1,2,3,4]
 * 输出:
 * [5,1,1,1]
 *
 * 解释:
```

```
 * - 节点 0 的子树包含基因值 1,2,3,4，缺失的最小值为 5
 * - 节点 1 的子树只包含基因值 2，缺失的最小值为 1
 * - 节点 2 的子树包含基因值 3,4，缺失的最小值为 1
 * - 节点 3 的子树只包含基因值 4，缺失的最小值为 1
 */
const int MAXN = 100010;

// 邻接表存储树
int head[MAXN];
int to[MAXN];
int next[MAXN];
int cnt = 0;

// 基因值
int genes[MAXN];

// 可持久化线段树需要
int root[MAXN];
int left[MAXN * 20];
int right[MAXN * 20];
int sum[MAXN * 20];
int treeCnt = 0;

// DFS 需要
int dfn[MAXN];  // DFS 序
int end[MAXN];  // 子树结束位置
int timestamp = 0;

/**
 * 添加边
 * @param u 起点
 * @param v 终点
 */
void addEdge(int u, int v) {
    next[++cnt] = head[u];
    to[cnt] = v;
    head[u] = cnt;
}

/**
 * 构建空线段树
 * @param l 区间左端点
 * @param r 区间右端点
```

```
 * @return 根节点编号
 */
int build(int l, int r) {
    int rt = ++treeCnt;
    sum[rt] = 0;
    if (l < r) {
        int mid = (l + r) / 2;
        left[rt] = build(l, mid);
        right[rt] = build(mid + 1, r);
    }
    return rt;
}

/**
 * 在线段树中插入一个值
 * @param pos 要插入的位置
 * @param l 区间左端点
 * @param r 区间右端点
 * @param pre 前一个版本的节点编号
 * @return 新节点编号
 */
int insert(int pos, int l, int r, int pre) {
    int rt = ++treeCnt;
    left[rt] = left[pre];
    right[rt] = right[pre];
    sum[rt] = sum[pre] + 1;

    if (l < r) {
        int mid = (l + r) / 2;
        if (pos <= mid) {
            left[rt] = insert(pos, l, mid, left[rt]);
        } else {
            right[rt] = insert(pos, mid + 1, r, right[rt]);
        }
    }
    return rt;
}

/**
 * 查询区间 Mex
 * @param l 查询区间左端点
 * @param r 查询区间右端点
 * @param u 前一个版本的根节点
```

```
 * @param v 当前版本的根节点
 * @return Mex 值
 */
int queryMex(int l, int r, int u, int v) {
    if (l == r) {
        return l;
    }
    int mid = (l + r) / 2;
    // 计算左子树中数的个数
    int leftCount = sum[left[v]] - sum[left[u]];
    // 如果左子树中数的个数等于区间长度，说明左子树满
    if (leftCount == mid - l + 1) {
        // Mex 在右子树中
        return queryMex(mid + 1, r, right[u], right[v]);
    } else {
        // Mex 在左子树中
        return queryMex(l, mid, left[u], left[v]);
    }
}

/**
 * DFS 遍历树并构建主席树
 * @param u 当前节点
 * @param pre 父节点
 */
void dfs(int u, int pre) {
    dfn[u] = ++timestamp;

    // 处理子节点
    for (int i = head[u]; i > 0; i = next[i]) {
        int v = to[i];
        if (v != pre) {
            dfs(v, u);
        }
    }

    end[u] = timestamp;
}
```

==========

文件: LeetCode1970_SmallestMissingGeneticValue.java

==========

```
package class158;

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

/**
 * LeetCode 1970. Smallest Missing Genetic Value in Each Subtree
 *
 * 题目来源：LeetCode
 * 题目链接：https://leetcode.com/problems/smallest-missing-genetic-value-in-each-subtree/
 *
 * 题目描述:
 * 给你一棵根节点为 0 的家族树，树中节点从 0 到 n-1 编号，parents[i]是节点 i 的父节点。
 * 每个节点有一个基因值 genes[i]。对于每个节点，求出以其为根的子树中缺失的最小基因值。
 *
 * 解题思路:
 * 使用可持久化线段树（主席树）解决子树 Mex 问题。
 * 1. 通过 DFS 遍历树，为每个节点建立主席树
 * 2. 对于每个节点，将其子树中的所有基因值插入到主席树中
 * 3. 查询 Mex 即为查询区间内未出现的最小自然数
 *
 * 时间复杂度: O(n log n)
 * 空间复杂度: O(n log n)
 *
 * 示例:
 * 输入:
 * parents = [-1,0,0,2], genes = [1,2,3,4]
 * 输出:
 * [5,1,1,1]
 *
 * 解释:
 * - 节点 0 的子树包含基因值 1,2,3,4，缺失的最小值为 5
 * - 节点 1 的子树只包含基因值 2，缺失的最小值为 1
 * - 节点 2 的子树包含基因值 3,4，缺失的最小值为 1
 * - 节点 3 的子树只包含基因值 4，缺失的最小值为 1
 */
public class LeetCode1970_SmallestMissingGeneticValue {
    static final int MAXN = 100010;

    // 邻接表存储树
    static int[] head = new int[MAXN];
    static int[] to = new int[MAXN];
    static int[] next = new int[MAXN];
```

```
static int cnt = 0;

// 基因值
static int[] genes = new int[MAXN];

// 可持久化线段树需要
static int[] root = new int[MAXN];
static int[] left = new int[MAXN * 20];
static int[] right = new int[MAXN * 20];
static int[] sum = new int[MAXN * 20];
static int treeCnt = 0;

// DFS 需要
static int[] dfn = new int[MAXN];  // DFS 序
static int[] end = new int[MAXN];  // 子树结束位置
static int timestamp = 0;

/**
 * 添加边
 * @param u 起点
 * @param v 终点
 */
static void addEdge(int u, int v) {
    next[++cnt] = head[u];
    to[cnt] = v;
    head[u] = cnt;
}

/**
 * 构建空线段树
 * @param l 区间左端点
 * @param r 区间右端点
 * @return 根节点编号
 */
static int build(int l, int r) {
    int rt = ++treeCnt;
    sum[rt] = 0;
    if (l < r) {
        int mid = (l + r) / 2;
        left[rt] = build(l, mid);
        right[rt] = build(mid + 1, r);
    }
    return rt;
```

```
}

/**
 * 在线段树中插入一个值
 * @param pos 要插入的位置
 * @param l 区间左端点
 * @param r 区间右端点
 * @param pre 前一个版本的节点编号
 * @return 新节点编号
 */
static int insert(int pos, int l, int r, int pre) {
    int rt = ++treeCnt;
    left[rt] = left[pre];
    right[rt] = right[pre];
    sum[rt] = sum[pre] + 1;

    if (l < r) {
        int mid = (l + r) / 2;
        if (pos <= mid) {
            left[rt] = insert(pos, l, mid, left[rt]);
        } else {
            right[rt] = insert(pos, mid + 1, r, right[rt]);
        }
    }
    return rt;
}

/**
 * 查询区间 Mex
 * @param l 查询区间左端点
 * @param r 查询区间右端点
 * @param u 前一个版本的根节点
 * @param v 当前版本的根节点
 * @return Mex 值
 */
static int queryMex(int l, int r, int u, int v) {
    if (l == r) {
        return l;
    }
    int mid = (l + r) / 2;
    // 计算左子树中数的个数
    int leftCount = sum[left[v]] - sum[left[u]];
    // 如果左子树中数的个数等于区间长度，说明左子树满
```

```
    if (leftCount == mid - l + 1) {
        // Mex 在右子树中
        return queryMex(mid + 1, r, right[u], right[v]);
    } else {
        // Mex 在左子树中
        return queryMex(l, mid, left[u], left[v]);
    }
}

/**
 * DFS 遍历树并构建主席树
 * @param u 当前节点
 * @param pre 父节点
 */
static void dfs(int u, int pre) {
    dfn[u] = ++timestamp;

    // 处理子节点
    for (int i = head[u]; i > 0; i = next[i]) {
        int v = to[i];
        if (v != pre) {
            dfs(v, u);
        }
    }

    end[u] = timestamp;
}

public static int[] smallestMissingValueSubtree(int[] parents, int[] genes) {
    int n = parents.length;
    cnt = 0;
    treeCnt = 0;
    timestamp = 0;

    // 初始化
    Arrays.fill(head, 0);

    // 建立邻接表
    for (int i = 1; i < n; i++) {
        addEdge(parents[i], i);
        addEdge(i, parents[i]);
    }
```

```
        // 复制基因值
        System.arraycopy(genes, 0, LeetCode1970_SmallestMissingGeneticValue.genes, 0, n);

        // DFS 遍历
        dfs(0, -1);

        // 构建主席树
        root[0] = build(1, n);
        for (int i = 1; i <= n; i++) {
            root[i] = insert(genes[i - 1], 1, n, root[i - 1]);
        }

        // 计算答案
        int[] ans = new int[n];
        for (int i = 0; i < n; i++) {
            ans[i] = queryMex(1, n, root[dfn[i] - 1], root[end[i]]);
        }

        return ans;
    }

    public static void main(String[] args) {
        // 测试用例
        int[] parents = {-1, 0, 0, 2};
        int[] genes = {1, 2, 3, 4};
        int[] result = smallestMissingValueSubtree(parents, genes);

        System.out.println(Arrays.toString(result)); // [5, 1, 1, 1]
    }
}
```

==========

文件: LeetCode1970_SmallestMissingGeneticValue.py

==========

```
"""
LeetCode 1970. Smallest Missing Genetic Value in Each Subtree

题目描述:
给你一棵根节点为 0 的家族树，树中节点从 0 到 n-1 编号，parents[i]是节点 i 的父节点。
每个节点有一个基因值 genes[i]。对于每个节点，求出以其为根的子树中缺失的最小基因值。

解题思路:
```

使用可持久化线段树（主席树）解决子树 Mex 问题。
1. 通过 DFS 遍历树，为每个节点建立主席树
2. 对于每个节点，将其子树中的所有基因值插入到主席树中
3. 查询 Mex 即为查询区间内未出现的最小自然数

时间复杂度: O(n log n)
空间复杂度: O(n log n)

示例:
输入:
parents = [-1,0,0,2], genes = [1,2,3,4]
输出:
[5,1,1,1]

解释:
- 节点 0 的子树包含基因值 1,2,3,4，缺失的最小值为 5
- 节点 1 的子树只包含基因值 2，缺失的最小值为 1
- 节点 2 的子树包含基因值 3,4，缺失的最小值为 1
- 节点 3 的子树只包含基因值 4，缺失的最小值为 1
"""

```
class Node:
    """线段树节点"""
    def __init__(self):
        self.sum: int = 0      # 区间内元素个数
        self.left = None  # 左子节点
        self.right = None # 右子节点


class PersistentSegmentTree:
    """可持久化线段树（主席树）"""
    def __init__(self, n):
        """
        初始化主席树
        :param n: 数组长度
        """
        self.n = n
        self.root = [None] * (n + 1)  # 每个版本线段树的根节点
        self.cnt = 0  # 节点计数器

    def build(self, l, r):
        """
```

```
    构建空线段树
    :param l: 区间左端点
    :param r: 区间右端点
    :return: 根节点
    """
    node = Node()
    if l < r:
        mid = (l + r) // 2
        node.left = self.build(l, mid)
        node.right = self.build(mid + 1, r)
    return node

def insert(self, pos, l, r, pre):
    """
    在线段树中插入一个值
    :param pos: 要插入的位置
    :param l: 区间左端点
    :param r: 区间右端点
    :param pre: 前一个版本的节点
    :return: 新节点
    """
    node = Node()
    node.left = pre.left if pre else None
    node.right = pre.right if pre else None
    node.sum = (pre.sum if pre else 0) + 1

    if l < r:
        mid = (l + r) // 2
        if pos <= mid:
            node.left = self.insert(pos, l, mid, pre.left if pre else None)
        else:
            node.right = self.insert(pos, mid + 1, r, pre.right if pre else None)
    return node

def query_mex(self, l, r, u, v):
    """
    查询区间Mex
    :param l: 查询区间左端点
    :param r: 查询区间右端点
    :param u: 前一个版本的根节点
    :param v: 当前版本的根节点
    :return: Mex值
    """
```

```
        if l == r:
            return l
        mid = (l + r) // 2
        # 计算左子树中数的个数
        left_count = (v.left.sum if v.left else 0) - (u.left.sum if u.left else 0)
        # 如果左子树中数的个数等于区间长度，说明左子树满
        if left_count == mid - l + 1:
            # Mex 在右子树中
            return self.query_mex(mid + 1, r, u.right if u else None, v.right if v else None)
        else:
            # Mex 在左子树中
            return self.query_mex(l, mid, u.left if u else None, v.left if v else None)


def smallestMissingValueSubtree(parents, genes):
    """
    计算每个子树中缺失的最小基因值
    :param parents: 父节点数组
    :param genes: 基因值数组
    :return: 每个子树中缺失的最小基因值数组
    """
    n = len(parents)

    # 构建邻接表
    children = [[] for _ in range(n)]
    for i in range(1, n):
        children[parents[i]].append(i)

    # DFS 序
    dfn = [0] * n
    end = [0] * n
    timestamp = 0

    def dfs(u):
        nonlocal timestamp
        timestamp += 1
        dfn[u] = timestamp

        for v in children[u]:
            dfs(v)

        end[u] = timestamp
```

```
    # DFS 遍历
    dfs(0)

    # 构建主席树
    pst = PersistentSegmentTree(n)
    pst.root[0] = pst.build(1, n)
    for i in range(1, n + 1):
        pst.root[i] = pst.insert(genes[i - 1], 1, n, pst.root[i - 1])

    # 计算答案
    ans = [0] * n
    for i in range(n):
        ans[i] = pst.query_mex(1, n, pst.root[dfn[i] - 1], pst.root[end[i]])

    return ans


# 测试代码
if __name__ == "__main__":
    parents = [-1, 0, 0, 2]
    genes = [1, 2, 3, 4]
    result = smallestMissingValueSubtree(parents, genes)
    print(result)  # [5, 1, 1, 1]

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

文件: LeetCode2276_CountIntervals.cpp
==============================================
#include <iostream>
using namespace std;

/**
 * LeetCode 2276. Count Integers in Intervals
 *
 * 题目描述:
 * 设计一个区间集合，支持以下操作：
 * 1. add(left, right): 添加区间[left, right]到集合中
 * 2. count(): 返回出现在至少一个区间中的整数个数
 *
 * 解题思路:
 * 使用动态开点线段树解决区间覆盖问题。
 * 1. 使用线段树维护区间覆盖情况
 * 2. 通过懒标记优化区间更新操作
```

```
 * 3. 维护区间内被覆盖的整数个数
 *
 * 时间复杂度: O(n log C), 其中 C 是值域范围
 * 空间复杂度: O(n log C)
 *
 * 示例:
 * 输入:
 * ["CountIntervals", "add", "add", "count", "add", "count"]
 * [[], [2, 3], [7, 10], [], [5, 8], []]
 *
 * 输出:
 * [null, null, null, 6, null, 8]
 *
 * 解释:
 * CountIntervals countIntervals = new CountIntervals(); // 用一个区间空集初始化对象
 * countIntervals.add(2, 3);  // 将 [2, 3] 添加到区间集合中
 * countIntervals.add(7, 10); // 将 [7, 10] 添加到区间集合中
 * countIntervals.count();    // 返回 6
 * // 整数 2 和 3 出现在区间 [2, 3] 中
 * // 整数 7、8、9、10 出现在区间 [7, 10] 中
 * countIntervals.add(5, 8);  // 将 [5, 8] 添加到区间集合中
 * countIntervals.count();    // 返回 8
 * // 整数 2 和 3 出现在区间 [2, 3] 中
 * // 整数 5 和 6 出现在区间 [5, 8] 中
 * // 整数 7 和 8 出现在区间 [5, 8] 和区间 [7, 10] 的并集内
 * // 整数 9 和 10 出现在区间 [7, 10] 中
 */
class CountIntervals {
private:
    // 线段树节点
    struct Node {
        int val = 0;      // 区间内被覆盖的整数个数
        int lazy = 0;     // 懒标记，0 表示无标记，1 表示全覆盖
        Node* left = nullptr;    // 左子节点
        Node* right = nullptr;   // 右子节点
    };

    Node* root;  // 线段树根节点
    static const int MAX_RANGE = 1000000000;  // 值域范围

    /**
     * 下传懒标记
     * @param node 当前节点
```

```
 * @param l 当前区间左端点
 * @param r 当前区间右端点
 */
void pushDown(Node* node, int l, int r) {
    // 如果当前节点没有左右子节点，则创建
    if (node->left == nullptr) {
        node->left = new Node();
    }
    if (node->right == nullptr) {
        node->right = new Node();
    }

    // 如果有懒标记
    if (node->lazy > 0) {
        // 将懒标记传递给左右子节点
        node->left->lazy = node->lazy;
        node->right->lazy = node->lazy;

        // 计算左右子节点区间长度
        int mid = l + (r - l) / 2;

        // 更新左右子节点的值
        node->left->val = mid - l + 1;
        node->right->val = r - mid;

        // 清除当前节点的懒标记
        node->lazy = 0;
    }
}

/**
 * 更新节点值
 * @param node 当前节点
 */
void pushUp(Node* node) {
    node->val = node->left->val + node->right->val;
}

/**
 * 线段树区间更新操作
 * @param node 当前节点
 * @param l 当前节点表示的区间左端点
 * @param r 当前节点表示的区间右端点
```

```
     * @param ql 查询区间左端点
     * @param qr 查询区间右端点
     */
    void update(Node* node, int l, int r, int ql, int qr) {
        // 如果当前区间被查询区间完全包含
        if (ql <= l && r <= qr) {
            // 设置懒标记为全覆盖
            node->lazy = 1;
            // 更新节点值为区间长度
            node->val = r - l + 1;
            return;
        }

        // 动态开点
        pushDown(node, l, r);

        int mid = l + (r - l) / 2;
        // 如果查询区间与左子树有交集
        if (ql <= mid) {
            update(node->left, l, mid, ql, qr);
        }
        // 如果查询区间与右子树有交集
        if (qr > mid) {
            update(node->right, mid + 1, r, ql, qr);
        }

        // 更新当前节点的值
        pushUp(node);
    }

public:
    CountIntervals() {
        root = new Node();
    }

    /**
     * 添加区间[left, right]到集合中
     * @param left 区间左端点
     * @param right 区间右端点
     */
    void add(int left, int right) {
        update(root, 1, MAX_RANGE, left, right);
    }
```

```
    /**
     * 返回出现在至少一个区间中的整数个数
     * @return 整数个数
     */
    int count() {
        return root->val;
    }
};

/**
 * Your CountIntervals object will be instantiated and called as such:
 * CountIntervals* obj = new CountIntervals();
 * obj->add(left,right);
 * int param_2 = obj->count();
 */
```

==========

文件: LeetCode2276_CountIntervals.java

==========

```
package class158;

/**
 * LeetCode 2276. Count Integers in Intervals
 *
 * 题目来源：LeetCode
 * 题目链接：https://leetcode.com/problems/count-integers-in-intervals/
 *
 * 题目描述:
 * 设计一个区间集合，支持以下操作：
 * 1. add(left, right): 添加区间[left, right]到集合中
 * 2. count(): 返回出现在至少一个区间中的整数个数
 *
 * 解题思路:
 * 使用动态开点线段树解决区间覆盖问题。
 * 1. 使用线段树维护区间覆盖情况
 * 2. 通过懒标记优化区间更新操作
 * 3. 维护区间内被覆盖的整数个数
 *
 * 时间复杂度: O(n log C), 其中 C 是值域范围
 * 空间复杂度: O(n log C)
 *
```

```
 * 示例:
 * 输入:
 * ["CountIntervals", "add", "add", "count", "add", "count"]
 * [[], [2, 3], [7, 10], [], [5, 8], []]
 *
 * 输出:
 * [null, null, null, 6, null, 8]
 *
 * 解释:
 * CountIntervals countIntervals = new CountIntervals(); // 用一个区间空集初始化对象
 * countIntervals.add(2, 3);  // 将 [2, 3] 添加到区间集合中
 * countIntervals.add(7, 10); // 将 [7, 10] 添加到区间集合中
 * countIntervals.count();    // 返回 6
 * // 整数 2 和 3 出现在区间 [2, 3] 中
 * // 整数 7、8、9、10 出现在区间 [7, 10] 中
 * countIntervals.add(5, 8);  // 将 [5, 8] 添加到区间集合中
 * countIntervals.count();    // 返回 8
 * // 整数 2 和 3 出现在区间 [2, 3] 中
 * // 整数 5 和 6 出现在区间 [5, 8] 中
 * // 整数 7 和 8 出现在区间 [5, 8] 和区间 [7, 10] 的并集内
 * // 整数 9 和 10 出现在区间 [7, 10] 中
 */
public class LeetCode2276_CountIntervals {
    // 线段树节点
    static class Node {
        int val;      // 区间内被覆盖的整数个数
        int lazy;     // 懒标记，0 表示无标记，1 表示全覆盖
        Node left;    // 左子节点
        Node right;   // 右子节点
    }

    private Node root;  // 线段树根节点
    private static final int MAX_RANGE = (int) 1e9;  // 值域范围

    public LeetCode2276_CountIntervals() {
        root = new Node();
    }

    /**
     * 添加区间[left, right]到集合中
     * @param left 区间左端点
     * @param right 区间右端点
     */
```

```
public void add(int left, int right) {
    update(root, 1, MAX_RANGE, left, right);
}

/**
 * 返回出现在至少一个区间中的整数个数
 * @return 整数个数
 */
public int count() {
    return root.val;
}

/**
 * 线段树区间更新操作
 * @param node 当前节点
 * @param l 当前节点表示的区间左端点
 * @param r 当前节点表示的区间右端点
 * @param ql 查询区间左端点
 * @param qr 查询区间右端点
 */
private void update(Node node, int l, int r, int ql, int qr) {
    // 如果当前区间被查询区间完全包含
    if (ql <= l && r <= qr) {
        // 设置懒标记为全覆盖
        node.lazy = 1;
        // 更新节点值为区间长度
        node.val = r - l + 1;
        return;
    }

    // 动态开点
    pushDown(node, l, r);

    int mid = l + (r - l) / 2;
    // 如果查询区间与左子树有交集
    if (ql <= mid) {
        update(node.left, l, mid, ql, qr);
    }
    // 如果查询区间与右子树有交集
    if (qr > mid) {
        update(node.right, mid + 1, r, ql, qr);
    }
```

```
    // 更新当前节点的值
    pushUp(node);
}

/**
 * 下传懒标记
 * @param node 当前节点
 * @param l 当前区间左端点
 * @param r 当前区间右端点
 */
private void pushDown(Node node, int l, int r) {
    // 如果当前节点没有左右子节点，则创建
    if (node.left == null) {
        node.left = new Node();
    }
    if (node.right == null) {
        node.right = new Node();
    }

    // 如果有懒标记
    if (node.lazy > 0) {
        // 将懒标记传递给左右子节点
        node.left.lazy = node.lazy;
        node.right.lazy = node.lazy;

        // 计算左右子节点区间长度
        int mid = l + (r - l) / 2;

        // 更新左右子节点的值
        node.left.val = mid - l + 1;
        node.right.val = r - mid;

        // 清除当前节点的懒标记
        node.lazy = 0;
    }
}

/**
 * 更新节点值
 * @param node 当前节点
 */
private void pushUp(Node node) {
    node.val = node.left.val + node.right.val;
```

```
    }
}
```

==========

文件: LeetCode2276_CountIntervals.py
==========
"""
LeetCode 2276. Count Integers in Intervals

题目描述:
设计一个区间集合，支持以下操作：
1. add(left, right): 添加区间[left, right]到集合中
2. count(): 返回出现在至少一个区间中的整数个数

解题思路:
使用动态开点线段树解决区间覆盖问题。
1. 使用线段树维护区间覆盖情况
2. 通过懒标记优化区间更新操作
3. 维护区间内被覆盖的整数个数

时间复杂度: O(n log C), 其中 C 是值域范围
空间复杂度: O(n log C)

示例:
输入:
["CountIntervals", "add", "add", "count", "add", "count"]
[[], [2, 3], [7, 10], [], [5, 8], []]

输出:
[null, null, null, 6, null, 8]

解释:
CountIntervals countIntervals = new CountIntervals(); // 用一个区间空集初始化对象
countIntervals.add(2, 3); // 将 [2, 3] 添加到区间集合中
countIntervals.add(7, 10); // 将 [7, 10] 添加到区间集合中
countIntervals.count();   // 返回 6
// 整数 2 和 3 出现在区间 [2, 3] 中
// 整数 7、8、9、10 出现在区间 [7, 10] 中
countIntervals.add(5, 8); // 将 [5, 8] 添加到区间集合中
countIntervals.count();   // 返回 8
// 整数 2 和 3 出现在区间 [2, 3] 中
// 整数 5 和 6 出现在区间 [5, 8] 中
```
```

```
// 整数 7 和 8 出现在区间 [5, 8] 和区间 [7, 10] 的并集内
// 整数 9 和 10 出现在区间 [7, 10] 中
"""



class Node:
    """线段树节点"""
    def __init__(self):
        self.val: int = 0      # 区间内被覆盖的整数个数
        self.lazy: int = 0     # 懒标记，0 表示无标记，1 表示全覆盖
        self.left = None  # 左子节点
        self.right = None # 右子节点



class CountIntervals:
    def __init__(self):
        """初始化线段树"""
        self.root = Node()  # 线段树根节点
        self.MAX_RANGE = 10**9  # 值域范围

    def add(self, left: int, right: int) -> None:
        """
        添加区间[left, right]到集合中
        :param left: 区间左端点
        :param right: 区间右端点
        """
        self._update(self.root, 1, self.MAX_RANGE, left, right)

    def count(self) -> int:
        """
        返回出现在至少一个区间中的整数个数
        :return: 整数个数
        """
        return self.root.val

    def _push_down(self, node: Node, l: int, r: int) -> None:
        """
        下传懒标记
        :param node: 当前节点
        :param l: 当前区间左端点
        :param r: 当前区间右端点
        """
        # 如果当前节点没有左右子节点，则创建
```

```
        if node.left is None:
            node.left = Node()
        if node.right is None:
            node.right = Node()

        # 如果有懒标记
        if node.lazy > 0:
            # 将懒标记传递给左右子节点
            node.left.lazy = node.lazy
            node.right.lazy = node.lazy

            # 计算左右子节点区间长度
            mid = l + (r - l) // 2

            # 更新左右子节点的值
            node.left.val = mid - l + 1
            node.right.val = r - mid

            # 清除当前节点的懒标记
            node.lazy = 0

    def _push_up(self, node: Node) -> None:
        """
        更新节点值
        :param node: 当前节点
        """
        node.val = node.left.val + node.right.val

    def _update(self, node: Node, l: int, r: int, ql: int, qr: int) -> None:
        """
        线段树区间更新操作
        :param node: 当前节点
        :param l: 当前节点表示的区间左端点
        :param r: 当前节点表示的区间右端点
        :param ql: 查询区间左端点
        :param qr: 查询区间右端点
        """
        # 如果当前区间被查询区间完全包含
        if ql <= l and r <= qr:
            # 设置懒标记为全覆盖
            node.lazy = 1
            # 更新节点值为区间长度
            node.val = r - l + 1
```

```
            return

        # 动态开点
        self._push_down(node, l, r)

        mid = l + (r - l) // 2
        # 如果查询区间与左子树有交集
        if ql <= mid:
            self._update(node.left, l, mid, ql, qr)
        # 如果查询区间与右子树有交集
        if qr > mid:
            self._update(node.right, mid + 1, r, ql, qr)

        # 更新当前节点的值
        self._push_up(node)


# 测试代码
if __name__ == "__main__":
    countIntervals = CountIntervals()
    countIntervals.add(2, 3)   # 将 [2, 3] 添加到区间集合中
    countIntervals.add(7, 10)  # 将 [7, 10] 添加到区间集合中
    print(countIntervals.count())    # 返回 6
    countIntervals.add(5, 8)   # 将 [5, 8] 添加到区间集合中
    print(countIntervals.count())    # 返回 8
```

==========

文件: Luogu_P3834_Java.java

==========

```
package class158;

// Problem: 洛谷 P3834 - 【模板】可持久化线段树 2 (静态区间第 k 小)
// Link: https://www.luogu.com.cn/problem/P3834
// Description: 给定一个包含 n 个数字的序列，每次查询区间[l,r]中第 k 小的数
// Solution: 使用可持久化线段树(主席树)解决静态区间第 k 小问题
// Time Complexity: O(nlogn) for preprocessing, O(logn) for each query
// Space Complexity: O(nlogn)

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

public class Luogu_P3834_Java {
```

```
static final int MAXN = 200005;
static int[] a = new int[MAXN];           // 原始数组
static int[] b = new int[MAXN];           // 离散化数组
static int n, m;

// 主席树节点
static class Node {
    int l, r, sum;
    Node(int l, int r, int sum) {
        this.l = l;
        this.r = r;
        this.sum = sum;
    }
}

static Node[] T = new Node[40 * MAXN];    // 主席树节点数组
static int[] root = new int[MAXN];        // 每个版本的根节点
static int cnt = 0;                       // 节点计数器

// 创建新节点
static int createNode(int l, int r, int sum) {
    T[++cnt] = new Node(l, r, sum);
    return cnt;
}

// 插入值到主席树
static int insert(int pre, int l, int r, int val) {
    int now = createNode(0, 0, 0);
    T[now].sum = T[pre].sum + 1;

    if (l == r) return now;

    int mid = (l + r) >> 1;
    if (val <= mid) {
        T[now].l = insert(T[pre].l, l, mid, val);
        T[now].r = T[pre].r;
    } else {
        T[now].l = T[pre].l;
        T[now].r = insert(T[pre].r, mid + 1, r, val);
    }
    return now;
}
```

```
// 查询区间第 k 小
static int query(int u, int v, int l, int r, int k) {
    if (l == r) return l;

    int mid = (l + r) >> 1;
    int x = T[T[v].l].sum - T[T[u].l].sum;

    if (k <= x) {
        return query(T[u].l, T[v].l, l, mid, k);
    } else {
        return query(T[u].r, T[v].r, mid + 1, r, k - x);
    }
}

// 离散化
static int getId(int x) {
    return Arrays.binarySearch(b, 1, n + 1, x) + 1;
}

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

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

    line = br.readLine().split(" ");
    for (int i = 1; i <= n; i++) {
        a[i] = Integer.parseInt(line[i - 1]);
        b[i] = a[i];
    }

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

    // 构建主席树
```

```
        root[0] = createNode(0, 0, 0);
        T[root[0]].l = T[root[0]].r = T[root[0]].sum = 0;

        for (int i = 1; i <= n; i++) {
            root[i] = insert(root[i - 1], 1, sz, getId(a[i]));
        }

        // 处理查询
        for (int i = 0; i < m; i++) {
            line = br.readLine().split(" ");
            int l = Integer.parseInt(line[0]);
            int r = Integer.parseInt(line[1]);
            int k = Integer.parseInt(line[2]);

            int id = query(root[l - 1], root[r], 1, sz, k);
            out.println(b[id]);
        }

        out.close();
    }
}
```

==========

文件: Luogu_P3834_Python.py

==========

```
# Problem: 洛谷 P3834 - 【模板】可持久化线段树 2 (静态区间第 k 小)
# Link: https://www.luogu.com.cn/problem/P3834
# Description: 给定一个包含 n 个数字的序列，每次查询区间[l,r]中第 k 小的数
# Solution: 使用可持久化线段树(主席树)解决静态区间第 k 小问题
# Time Complexity: O(nlogn) for preprocessing, O(logn) for each query
# Space Complexity: O(nlogn)

import sys
from bisect import bisect_left

class Node:
    def __init__(self):
        self.l = 0
        self.r = 0
        self.sum = 0

class PersistentSegmentTree:
```

```
    def __init__(self, maxn=200005):
        self.MAXN = maxn
        self.T = [Node() for _ in range(40 * maxn)]
        self.root = [0] * maxn
        self.cnt = 0

    def create_node(self, l=0, r=0, sum=0):
        self.cnt += 1
        self.T[self.cnt].l = l
        self.T[self.cnt].r = r
        self.T[self.cnt].sum = sum
        return self.cnt

    def insert(self, pre, l, r, val):
        now = self.create_node()
        self.T[now].sum = self.T[pre].sum + 1

        if l == r:
            return now

        mid = (l + r) >> 1
        if val <= mid:
            self.T[now].l = self.insert(self.T[pre].l, l, mid, val)
            self.T[now].r = self.T[pre].r
        else:
            self.T[now].l = self.T[pre].l
            self.T[now].r = self.insert(self.T[pre].r, mid + 1, r, val)
        return now

    def query(self, u, v, l, r, k):
        if l == r:
            return l

        mid = (l + r) >> 1
        x = self.T[self.T[v].l].sum - self.T[self.T[u].l].sum

        if k <= x:
            return self.query(self.T[u].l, self.T[v].l, l, mid, k)
        else:
            return self.query(self.T[u].r, self.T[v].r, mid + 1, r, k - x)

def main():
    line = sys.stdin.readline().split()
```

```
    n, m = int(line[0]), int(line[1])

    line = sys.stdin.readline().split()
    a = [0] * (n + 1)
    b = [0] * (n + 1)

    for i in range(1, n + 1):
        a[i] = int(line[i - 1])
        b[i] = a[i]

    # 离散化
    b = b[1:]  # 去掉索引 0
    b.sort()
    # 去重
    unique_b = []
    for x in b:
        if not unique_b or unique_b[-1] != x:
            unique_b.append(x)
    b = [0] + unique_b  # 加回索引 0
    sz = len(b) - 1

    # 获取值的离散化 ID
    def get_id(x):
        return bisect_left(b, x, 1, len(b))

    # 构建主席树
    pst = PersistentSegmentTree(n + 1)
    pst.root[0] = pst.create_node()

    for i in range(1, n + 1):
        pst.root[i] = pst.insert(pst.root[i - 1], 1, sz, get_id(a[i]))

    # 处理查询
    for _ in range(m):
        line = sys.stdin.readline().split()
        l, r, k = int(line[0]), int(line[1]), int(line[2])
        id = pst.query(pst.root[l - 1], pst.root[r], 1, sz, k)
        print(b[id])

if __name__ == "__main__":
    main()

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

文件: Luogu_P3919_Java.java

```
==========================================
package class158;

// Problem: 洛谷 P3919 - 【模板】可持久化线段树 1（可持久化数组）
// Link: https://www.luogu.com.cn/problem/P3919
// Description: 维护一个长度为 N 的数组，支持在某个历史版本上修改某一个位置上的值，以及访问某个历史版本上的某一位置的值
// Solution: 使用可持久化线段树实现可持久化数组
// Time Complexity: O(logn) for each operation
// Space Complexity: O(n + mlogn) where m is the number of operations

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

public class Luogu_P3919_Java {
    static final int MAXN = 1000005;
    static int[] a = new int[MAXN];           // 初始数组
    static int n, m;

    // 主席树节点
    static class Node {
        int l, r, val;
        Node(int l, int r, int val) {
            this.l = l;
            this.r = r;
            this.val = val;
        }
    }

    static Node[] T = new Node[40 * MAXN];    // 主席树节点数组
    static int[] root = new int[MAXN];        // 每个版本的根节点
    static int cnt = 0;                       // 节点计数器

    // 创建新节点
    static int createNode(int l, int r, int val) {
        T[++cnt] = new Node(l, r, val);
        return cnt;
    }

    // 构建初始线段树
    static int build(int l, int r) {
```

```
    int now = createNode(0, 0, 0);

    if (l == r) {
        T[now].val = a[l];
        return now;
    }

    int mid = (l + r) >> 1;
    T[now].l = build(l, mid);
    T[now].r = build(mid + 1, r);
    return now;
}

// 在主席树中修改位置 p 的值为 x
static int update(int pre, int l, int r, int p, int x) {
    int now = createNode(0, 0, 0);

    if (l == r) {
        T[now].val = x;
        return now;
    }

    int mid = (l + r) >> 1;
    if (p <= mid) {
        T[now].l = update(T[pre].l, l, mid, p, x);
        T[now].r = T[pre].r;
    } else {
        T[now].l = T[pre].l;
        T[now].r = update(T[pre].r, mid + 1, r, p, x);
    }
    return now;
}

// 在主席树中查询位置 p 的值
static int query(int u, int l, int r, int p) {
    if (l == r) {
        return T[u].val;
    }

    int mid = (l + r) >> 1;
    if (p <= mid) {
        return query(T[u].l, l, mid, p);
    } else {
```

```
            return query(T[u].r, mid + 1, r, p);
        }
    }

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

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

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

        // 构建初始版本
        root[0] = build(1, n);

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

            if (line[1].equals("1")) {
                // 修改操作
                int p = Integer.parseInt(line[2]);
                int x = Integer.parseInt(line[3]);
                root[i] = update(root[v], 1, n, p, x);
            } else {
                // 查询操作
                int p = Integer.parseInt(line[2]);
                out.println(query(root[v], 1, n, p));
                root[i] = root[v]; // 生成一样的版本
            }
        }

        out.close();
    }
}
```

==========

文件: Luogu_P3919_Python.py

```
==========================================
# Problem: 洛谷 P3919 - 【模板】可持久化线段树 1（可持久化数组）
# Link: https://www.luogu.com.cn/problem/P3919
# Description: 维护一个长度为 N 的数组，支持在某个历史版本上修改某一个位置上的值，以及访问某个历史版本上的某一位置的值
# Solution: 使用可持久化线段树实现可持久化数组
# Time Complexity: O(logn) for each operation
# Space Complexity: O(n + mlogn) where m is the number of operations

import sys

class Node:
    def __init__(self):
        self.l = 0
        self.r = 0
        self.val = 0

class PersistentSegmentTree:
    def __init__(self, maxn=1000005):
        self.MAXN = maxn
        self.T = [Node() for _ in range(40 * maxn)]
        self.root = [0] * maxn
        self.cnt = 0

    def create_node(self, l=0, r=0, val=0):
        self.cnt += 1
        self.T[self.cnt].l = l
        self.T[self.cnt].r = r
        self.T[self.cnt].val = val
        return self.cnt

    def build(self, a, l, r):
        now = self.create_node()

        if l == r:
            self.T[now].val = a[l]
            return now

        mid = (l + r) >> 1
        self.T[now].l = self.build(a, l, mid)
        self.T[now].r = self.build(a, mid + 1, r)
```

```
        return now

    def update(self, pre, l, r, p, x):
        now = self.create_node()

        if l == r:
            self.T[now].val = x
            return now

        mid = (l + r) >> 1
        if p <= mid:
            self.T[now].l = self.update(self.T[pre].l, l, mid, p, x)
            self.T[now].r = self.T[pre].r
        else:
            self.T[now].l = self.T[pre].l
            self.T[now].r = self.update(self.T[pre].r, mid + 1, r, p, x)
        return now

    def query(self, u, l, r, p):
        if l == r:
            return self.T[u].val

        mid = (l + r) >> 1
        if p <= mid:
            return self.query(self.T[u].l, l, mid, p)
        else:
            return self.query(self.T[u].r, mid + 1, r, p)

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

    line = sys.stdin.readline().split()
    a = [0] * (n + 1)

    for i in range(1, n + 1):
        a[i] = int(line[i - 1])

    # 构建初始版本
    pst = PersistentSegmentTree(n + 1)
    pst.root[0] = pst.build(a, 1, n)

    # 处理操作
```

```
    for i in range(1, m + 1):
        line = sys.stdin.readline().split()
        v = int(line[0])

        if line[1] == "1":
            # 修改操作
            p = int(line[2])
            x = int(line[3])
            pst.root[i] = pst.update(pst.root[v], 1, n, p, x)
        else:
            # 查询操作
            p = int(line[2])
            print(pst.query(pst.root[v], 1, n, p))
            pst.root[i] = pst.root[v]  # 生成一样的版本

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

==========

文件: main.py

==========

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
持久化线段树（主席树）解法主入口
包含所有题目的测试和示例
"""

import sys
from persistent_segment_tree_solutions import *


def test_mkthnum():
    """测试SPOJ MKTHNUM题目解法"""
    print("=== 测试 SPOJ MKTHNUM - K-th Number ===")

    # 测试用例1
    nums = [1, 5, 2, 6, 3, 7, 4]
    queries = [(2, 5, 3), (4, 4, 1), (1, 7, 3)]

    solver = MKTHNUM()
    results = solver.solve(nums, queries)
```

```
    print("输入数组:", nums)
    print("查询结果:")
    for (l, r, k), res in zip(queries, results):
        print(f"区间 [{l}, {r}] 的第 {k} 小元素是: {res}")
    print()


def test_cot():
    """测试 SPOJ COT 题目解法"""
    print("=== 测试 SPOJ COT - Count on a Tree ===")

    # 测试用例 1
    n = 5
    m = 3
    values = [1, 2, 3, 4, 5]
    edges = [(1, 2), (1, 3), (2, 4), (2, 5)]
    queries = [(1, 4, 2), (2, 5, 1), (3, 4, 3)]

    solver = COT()
    results = solver.solve(n, m, values, edges, queries)

    print("树节点值:", values)
    print("查询结果:")
    for (u, v, k), res in zip(queries, results):
        print(f"节点 {u} 到 {v} 路径上的第 {k} 小元素是: {res}")
    print()


def test_count_intervals():
    """测试 LeetCode 2276 题目解法"""
    print("=== 测试 LeetCode 2276 - Count Integers in Intervals ===")

    intervals = CountIntervals()
    intervals.add(2, 3)
    print("添加区间 [2,3] 后的计数:", intervals.count())  # 应输出 2

    intervals.add(7, 10)
    print("添加区间 [7,10] 后的计数:", intervals.count())  # 应输出 6

    intervals.add(5, 8)
    print("添加区间 [5,8] 后的计数:", intervals.count())  # 应输出 8
    print()
```

```
def test_smallest_missing_genetic_value():
    """测试 LeetCode 1970 题目解法"""
    print("=== 测试 LeetCode 1970 - Smallest Missing Genetic Value in Each Subtree ===")

    # 测试用例 1
    parents = [-1, 0, 0, 2]
    nums = [1, 2, 3, 4]
    solver = SmallestMissingGeneticValue()
    result = solver.solve(parents, nums)
    print(f"父数组: {parents}")
    print(f"基因值数组: {nums}")
    print(f"结果: {result}")  # 应输出 [5, 1, 1, 1]

    # 测试用例 2
    parents = [-1, 0, 0, 2]
    nums = [5, 2, 3, 4]
    result = solver.solve(parents, nums)
    print(f"\n父数组: {parents}")
    print(f"基因值数组: {nums}")
    print(f"结果: {result}")  # 应输出 [1, 1, 1, 1]
    print()


def test_dquery():
    """测试 SPOJ DQUERY 题目解法"""
    print("=== 测试 SPOJ DQUERY - D-query ===")

    # 测试用例 1
    nums = [1, 1, 2, 1, 3]
    queries = [(1, 5), (2, 4), (1, 3)]
    solver = DQUERY()
    results = solver.solve(nums, queries)

    print("输入数组:", nums)
    print("查询结果:")
    for (l, r), res in zip(queries, results):
        print(f"区间 [{l}, {r}] 内不同元素的个数: {res}")
    print()


def test_first_occurrence():
```

```
    """测试第一次出现位置序列查询"""
    print("=== 测试 第一次出现位置序列查询 ===")

    # 测试用例 1
    nums = [1, 2, 3, 2, 5]
    queries = [(1, 5), (2, 4), (1, 3)]
    solver = FirstOccurrence()
    results = solver.solve(nums, queries)

    print("输入数组:", nums)
    print("查询结果:")
    for (l, r), res in zip(queries, results):
        print(f"区间 [{l}, {r}] 内元素第一次出现的位置: {res}")
    print()


def test_range_mex():
    """测试区间最小缺失自然数查询"""
    print("=== 测试 区间最小缺失自然数查询 ===")

    # 测试用例 1
    nums = [0, 1, 2, 3]
    queries = [(0, 3), (1, 2), (0, 1)]
    solver = RangeMex()
    results = solver.solve(nums, queries)

    print("输入数组:", nums)
    print("查询结果:")
    for (l, r), res in zip(queries, results):
        print(f"区间 [{l}, {r}] 内最小缺失自然数: {res}")
    print()


def run_all_tests():
    """运行所有测试"""
    print("=" * 50)
    print("持久化线段树（主席树）所有题目测试")
    print("=" * 50)

    try:
        test_mkthnum()
        test_cot()
        test_count_intervals()
```

```
        test_smallest_missing_genetic_value()
        test_dquery()
        test_first_occurrence()
        test_range_mex()

        print("=" * 50)
        print("所有测试运行完成！")
        print("=" * 50)

    except Exception as e:
        print(f"测试过程中发生错误: {e}")
        import traceback
        traceback.print_exc()


def interactive_mode():
    """交互式模式，让用户选择要测试的题目"""
    menu = {
        '1': ('SPOJ MKTHNUM - K-th Number', test_mkthnum),
        '2': ('SPOJ COT - Count on a Tree', test_cot),
        '3': ('LeetCode 2276 - Count Integers in Intervals', test_count_intervals),
        '4': ('LeetCode 1970 - Smallest Missing Genetic Value in Each Subtree',
test_smallest_missing_genetic_value),
        '5': ('SPOJ DQUERY - D-query', test_dquery),
        '6': ('第一次出现位置序列查询', test_first_occurrence),
        '7': ('区间最小缺失自然数查询', test_range_mex),
        '0': ('运行所有测试', run_all_tests),
        'q': ('退出', lambda: sys.exit(0))
    }

    while True:
        print("\n 持久化线段树题目测试菜单:")
        print("=" * 60)
        for key, (desc, _) in sorted(menu.items()):
            print(f"{key}. {desc}")
        print("=" * 60)

        choice = input("请选择要测试的题目 (输入编号): ")

        if choice in menu:
            print()
            menu[choice][1]()
        else:
```

```
            print("无效的选择，请重新输入！")


if __name__ == "__main__":
    print("欢迎使用持久化线段树（主席树）解法测试工具")

    # 检查是否有命令行参数
    if len(sys.argv) > 1 and sys.argv[1] == "--all":
        run_all_tests()
    else:
        interactive_mode()
```

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

文件: PersistentSegmentTreeSolutions.java

```
==========================================
package class158;

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

/**
 * 持久化线段树（主席树）完整解决方案
 * 包含所有经典题目和详细实现
 * 时间复杂度分析：构建O(n log n)，单点更新O(log n)，区间查询O(log n)
 * 空间复杂度分析：O(n log n)
 * 最优解分析：对于区间第K小、历史版本查询等问题，持久化线段树是最优解之一
 */

public class PersistentSegmentTreeSolutions {

    // 题目1: SPOJ MKTHNUM - K-th Number
    // 题目链接: https://www.spoj.com/problems/MKTHNUM/
    // 题目描述: 给定一个数组，多次查询区间第K小的数
    // 最优解: 持久化线段树是该问题的最优解之一
    public static class MKTHNUM {
        static class Node {
            int left, right, count;
            public Node(int left, int right, int count) {
                this.left = left;
                this.right = right;
                this.count = count;
            }
```

```
}

static List<Node> nodes;
static List<Integer> roots;

public static void solve() throws IOException {
    Reader in = new Reader();
    PrintWriter out = new PrintWriter(System.out);

    int n = in.nextInt();
    int m = in.nextInt();

    int[] arr = new int[n];
    for (int i = 0; i < n; i++) {
        arr[i] = in.nextInt();
    }

    // 离散化
    int[] sortedArr = arr.clone();
    Arrays.sort(sortedArr);
    Map<Integer, Integer> rank = new HashMap<>();
    int idx = 1;
    for (int num : sortedArr) {
        if (!rank.containsKey(num)) {
            rank.put(num, idx++);
        }
    }

    // 初始化
    nodes = new ArrayList<>();
    roots = new ArrayList<>();
    nodes.add(new Node(0, 0, 0)); // 哨兵节点
    roots.add(0);

    // 构建持久化线段树
    for (int i = 0; i < n; i++) {
        int r = update(roots.get(i), 1, idx - 1, rank.get(arr[i]), 1);
        roots.add(r);
    }

    // 处理查询
    for (int i = 0; i < m; i++) {
        int l = in.nextInt() - 1;
```

```
            int r = in.nextInt() - 1;
            int k = in.nextInt();
            int pos = query(roots.get(l), roots.get(r + 1), 1, idx - 1, k);
            // 找到对应的原始值
            int left = 0, right = sortedArr.length - 1;
            while (left <= right) {
                int mid = left + (right - left) / 2;
                if (rank.get(sortedArr[mid]) == pos) {
                    out.println(sortedArr[mid]);
                    break;
                } else if (rank.get(sortedArr[mid]) < pos) {
                    left = mid + 1;
                } else {
                    right = mid - 1;
                }
            }
        }

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

    /**
     * 更新线段树节点
     * @param preRoot 前一个版本的根节点
     * @param l 当前区间左端点
     * @param r 当前区间右端点
     * @param pos 要更新的位置
     * @param val 要增加的值
     * @return 新版本的根节点
     */
    private static int update(int preRoot, int l, int r, int pos, int val) {
        // 创建新节点
        nodes.add(new Node(
            nodes.get(preRoot).left,
            nodes.get(preRoot).right,
            nodes.get(preRoot).count
        ));
        int newRoot = nodes.size() - 1;

        if (l == r) {
            nodes.get(newRoot).count += val;
            return newRoot;
```

```
            }

            int mid = l + (r - l) / 2;
            if (pos <= mid) {
                nodes.get(newRoot).left = update(nodes.get(preRoot).left, l, mid, pos, val);
            } else {
                nodes.get(newRoot).right = update(nodes.get(preRoot).right, mid + 1, r, pos,
val);
            }

            // 向上更新
            nodes.get(newRoot).count = nodes.get(nodes.get(newRoot).left).count
                                     + nodes.get(nodes.get(newRoot).right).count;
            return newRoot;
        }

        /**
         * 查询区间第 k 小的数
         * @param root1 前一个版本的根节点
         * @param root2 当前版本的根节点
         * @param l 当前区间左端点
         * @param r 当前区间右端点
         * @param k 要查询的排名
         * @return 第 k 小的数在离散化数组中的位置
         */
        private static int query(int root1, int root2, int l, int r, int k) {
            if (l == r) {
                return l;
            }

            int mid = l + (r - l) / 2;
            int leftCount = nodes.get(nodes.get(root2).left).count
                          - nodes.get(nodes.get(root1).left).count;

            if (k <= leftCount) {
                return query(nodes.get(root1).left, nodes.get(root2).left, l, mid, k);
            } else {
                return query(nodes.get(root1).right, nodes.get(root2).right, mid + 1, r, k -
leftCount);
            }
        }
    }
```

```
// 题目 2: SPOJ COT - Count on a Tree
// 题目链接: https://www.spoj.com/problems/COT/
// 题目描述: 给定一棵树，多次查询两点之间路径上的第 K 小的数
// 最优解: 树上持久化线段树（树链剖分+主席树）是该问题的最优解
public static class COT {
    static class Node {
        int left, right, count;
        public Node(int left, int right, int count) {
            this.left = left;
            this.right = right;
            this.count = count;
        }
    }

    static List<Node> nodes;
    static int[] roots;
    static List<Integer>[] graph;
    static int[] values;
    static int[][] parent;
    static int[] depth;
    static int LOG = 20;

    public static void solve() throws IOException {
        Reader in = new Reader();
        PrintWriter out = new PrintWriter(System.out);

        int n = in.nextInt();
        int m = in.nextInt();

        values = new int[n];
        for (int i = 0; i < n; i++) {
            values[i] = in.nextInt();
        }

        // 离散化
        int[] sortedVals = values.clone();
        Arrays.sort(sortedVals);
        Map<Integer, Integer> rank = new HashMap<>();
        int idx = 1;
        for (int val : sortedVals) {
            if (!rank.containsKey(val)) {
                rank.put(val, idx++);
            }
```

```
}

// 构建图
graph = new ArrayList[n + 1];
for (int i = 1; i <= n; i++) {
    graph[i] = new ArrayList<>();
}
for (int i = 0; i < n - 1; i++) {
    int u = in.nextInt();
    int v = in.nextInt();
    graph[u].add(v);
    graph[v].add(u);
}

// 初始化 LCA 数组
parent = new int[LOG][n + 1];
depth = new int[n + 1];

// 初始化持久化线段树
nodes = new ArrayList<>();
roots = new int[n + 1];
nodes.add(new Node(0, 0, 0));

// DFS 构建
dfs(1, 0, rank);

// 构建 LCA 倍增表
for (int k = 1; k < LOG; k++) {
    for (int i = 1; i <= n; i++) {
        parent[k][i] = parent[k - 1][parent[k - 1][i]];
    }
}

// 处理查询
for (int i = 0; i < m; i++) {
    int u = in.nextInt();
    int v = in.nextInt();
    int k = in.nextInt();
    int ancestor = lca(u, v);
    int res = query(u, v, ancestor, parent[0][ancestor], 1, idx - 1, k);
    // 找到对应的原始值
    int left = 0, right = sortedVals.length - 1;
    while (left <= right) {
```

```
            int mid = left + (right - left) / 2;
            if (rank.get(sortedVals[mid]) == res) {
                out.println(sortedVals[mid]);
                break;
            } else if (rank.get(sortedVals[mid]) < res) {
                left = mid + 1;
            } else {
                right = mid - 1;
            }
        }
    }

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

/**
 * DFS 遍历树并构建主席树
 * @param u 当前节点
 * @param p 父节点
 * @param rank 离散化映射
 */
private static void dfs(int u, int p, Map<Integer, Integer> rank) {
    parent[0][u] = p;
    depth[u] = depth[p] + 1;

    // 继承父节点的版本并更新
    roots[u] = update(roots[p], 1, rank.size(), rank.get(values[u - 1]), 1);

    for (int v : graph[u]) {
        if (v != p) {
            dfs(v, u, rank);
        }
    }
}

/**
 * 计算节点 u 和节点 v 的最近公共祖先(LCA)
 * @param u 节点 u
 * @param v 节点 v
 * @return LCA 节点
 */
private static int lca(int u, int v) {
```

```
        if (depth[u] < depth[v]) {
            int temp = u;
            u = v;
            v = temp;
        }

        // 提升 u 到 v 的深度
        for (int k = LOG - 1; k >= 0; k--) {
            if (depth[u] - (1 << k) >= depth[v]) {
                u = parent[k][u];
            }
        }

        if (u == v) return u;

        for (int k = LOG - 1; k >= 0; k--) {
            if (parent[k][u] != parent[k][v]) {
                u = parent[k][u];
                v = parent[k][v];
            }
        }

        return parent[0][u];
    }

    /**
     * 更新线段树节点
     * @param preRoot 前一个版本的根节点
     * @param l 当前区间左端点
     * @param r 当前区间右端点
     * @param pos 要更新的位置
     * @param val 要增加的值
     * @return 新版本的根节点
     */
    private static int update(int preRoot, int l, int r, int pos, int val) {
        nodes.add(new Node(
            nodes.get(preRoot).left,
            nodes.get(preRoot).right,
            nodes.get(preRoot).count
        ));
        int newRoot = nodes.size() - 1;

        if (l == r) {
```

```
            nodes.get(newRoot).count += val;
            return newRoot;
        }

        int mid = l + (r - l) / 2;
        if (pos <= mid) {
            nodes.get(newRoot).left = update(nodes.get(preRoot).left, l, mid, pos, val);
        } else {
            nodes.get(newRoot).right = update(nodes.get(preRoot).right, mid + 1, r, pos,
val);
        }

        nodes.get(newRoot).count = nodes.get(nodes.get(newRoot).left).count
                                 + nodes.get(nodes.get(newRoot).right).count;
        return newRoot;
    }

    /**
     * 查询树上路径第 k 小的数
     * @param u 节点 u
     * @param v 节点 v
     * @param ancestor LCA 节点
     * @param ancestorParent LCA 的父节点
     * @param l 当前区间左端点
     * @param r 当前区间右端点
     * @param k 要查询的排名
     * @return 第 k 小的数在离散化数组中的位置
     */
    private static int query(int u, int v, int ancestor, int ancestorParent, int l, int r,
int k) {
        if (l == r) {
            return l;
        }

        int mid = l + (r - l) / 2;
        int leftCount = nodes.get(nodes.get(u).left).count +
nodes.get(nodes.get(v).left).count
                      - nodes.get(nodes.get(ancestor).left).count -
nodes.get(nodes.get(ancestorParent).left).count;

        if (k <= leftCount) {
            return query(nodes.get(u).left, nodes.get(v).left,
                         nodes.get(ancestor).left, nodes.get(ancestorParent).left,
```

```
                        l, mid, k);
        } else {
            return query(nodes.get(u).right, nodes.get(v).right,
                        nodes.get(ancestor).right, nodes.get(ancestorParent).right,
                        mid + 1, r, k - leftCount);
        }
    }
}

// 题目 3: LeetCode 2276 - Count Integers in Intervals
// 题目链接: https://leetcode.com/problems/count-integers-in-intervals/
// 题目描述: 实现一个数据结构，支持添加区间和查询区间内整数的个数
// 最优解: 动态开点线段树是该问题的最优解之一
public static class CountIntervals {
    class Node {
        Node left, right;
        int cnt, cover;
    }

    private Node root;
    private int total;
    private final int MAX = 1000000000;

    public CountIntervals() {
        root = new Node();
        total = 0;
    }

    /**
     * 向上更新节点信息
     * @param node 节点
     * @param l 区间左端点
     * @param r 区间右端点
     */
    private void pushUp(Node node, int l, int r) {
        if (node.cover > 0) {
            node.cnt = r - l + 1;
        } else if (l == r) {
            node.cnt = 0;
        } else {
            node.cnt = (node.left != null ? node.left.cnt : 0)
                     + (node.right != null ? node.right.cnt : 0);
        }
```

```
}

/**
 * 线段树区间更新操作
 * @param node 当前节点
 * @param l 当前节点表示的区间左端点
 * @param r 当前节点表示的区间右端点
 * @param ul 更新区间左端点
 * @param ur 更新区间右端点
 * @param val 要增加的值
 */
private void update(Node node, int l, int r, int ul, int ur, int val) {
    if (ur < l || ul > r) {
        return;
    }

    if (ul <= l && r <= ur) {
        node.cover += val;
        pushUp(node, l, r);
        return;
    }

    // 动态开点
    if (node.left == null) node.left = new Node();
    if (node.right == null) node.right = new Node();

    int mid = l + (r - l) / 2;
    update(node.left, l, mid, ul, ur, val);
    update(node.right, mid + 1, r, ul, ur, val);
    pushUp(node, l, r);
}

/**
 * 添加区间[left, right]到集合中
 * @param left 区间左端点
 * @param right 区间右端点
 */
public void add(int left, int right) {
    int before = root.cnt;
    update(root, 1, MAX, left, right, 1);
    total = root.cnt;
}
```

```
    /**
     * 返回出现在至少一个区间中的整数个数
     * @return 整数个数
     */
    public int count() {
        return total;
    }
}

// 题目 4: LeetCode 1970 - Smallest Missing Genetic Value in Each Subtree
// 题目链接: https://leetcode.com/problems/smallest-missing-genetic-value-in-each-subtree/
// 题目描述: 给定一棵树，每个节点有一个基因值，求每个子树中最小缺失的基因值
public static class SmallestMissingGeneticValue {
    /**
     * 求每个子树中最小缺失的基因值
     * @param parents 父节点数组
     * @param nums 基因值数组
     * @return 每个子树中最小缺失的基因值
     */
    public static int[] smallestMissingValueSubtree(int[] parents, int[] nums) {
        int n = parents.length;
        int[] res = new int[n];
        Arrays.fill(res, 1);

        // 检查是否存在值为 1 的节点
        int posOfOne = -1;
        for (int i = 0; i < n; i++) {
            if (nums[i] == 1) {
                posOfOne = i;
                break;
            }
        }
        if (posOfOne == -1) {
            return res;
        }

        // 构建子树
        List<Integer>[] children = new ArrayList[n];
        for (int i = 0; i < n; i++) {
            children[i] = new ArrayList<>();
        }
        for (int i = 0; i < n; i++) {
            if (parents[i] != -1) {
```

```
            children[parents[i]].add(i);
        }
    }

    // 并查集
    int[] parent = new int[100002];
    for (int i = 0; i < parent.length; i++) {
        parent[i] = i;
    }

    // 后序遍历
    boolean[] visited = new boolean[n];
    dfs(posOfOne, children, nums, parent, res, visited);

    return res;
}

/**
 * 并查集查找操作
 * @param parent 并查集数组
 * @param u 节点
 * @return 根节点
 */
private static int find(int[] parent, int u) {
    if (parent[u] != u) {
        parent[u] = find(parent, parent[u]);
    }
    return parent[u];
}

/**
 * DFS 遍历树并计算结果
 * @param node 当前节点
 * @param children 子节点列表
 * @param nums 基因值数组
 * @param parent 并查集数组
 * @param res 结果数组
 * @param visited 访问标记数组
 */
private static void dfs(int node, List<Integer>[] children, int[] nums, int[] parent,
                        int[] res, boolean[] visited) {
    visited[node] = true;
```

```
        for (int child : children[node]) {
            if (!visited[child]) {
                dfs(child, children, nums, parent, res, visited);
            }
        }

        // 合并当前节点的值
        int u = find(parent, nums[node]);
        parent[u] = u + 1;

        if (nums[node] == 1) {
            res[node] = find(parent, 1);
        }

        // 向上传递结果
        int curr = node;
        while (parent[curr] != -1) {
            curr = parent[curr];
            if (curr >= 0 && curr < res.length && res[curr] == 1) {
                res[curr] = find(parent, 1);
            } else {
                break;
            }
        }
    }
}

// 题目 5: SPOJ DQUERY - D-query
// 题目链接: https://www.spoj.com/problems/DQUERY/
// 题目描述: 给定一个数组，多次查询区间内不同元素的个数
public static class DQUERY {
    public static void solve() throws IOException {
        Reader in = new Reader();
        PrintWriter out = new PrintWriter(System.out);

        int n = in.nextInt();
        int[] arr = new int[n];
        for (int i = 0; i < n; i++) {
            arr[i] = in.nextInt();
        }

        int m = in.nextInt();
        int[][] queries = new int[m][3];
```

```
        for (int i = 0; i < m; i++) {
            queries[i][0] = in.nextInt() - 1;  // l
            queries[i][1] = in.nextInt() - 1;  // r
            queries[i][2] = i;                 // index
        }

        // 按右端点排序
        Arrays.sort(queries, Comparator.comparingInt(a -> a[1]));

        // 树状数组
        FenwickTree ft = new FenwickTree(n);
        int[] res = new int[m];
        Map<Integer, Integer> last = new HashMap<>();
        int ptr = 0;

        for (int i = 0; i < n; i++) {
            if (last.containsKey(arr[i])) {
                ft.update(last.get(arr[i]), -1);
            }
            ft.update(i, 1);
            last.put(arr[i], i);

            while (ptr < m && queries[ptr][1] == i) {
                int l = queries[ptr][0];
                int qIdx = queries[ptr][2];
                res[qIdx] = ft.query(i) - (l > 0 ? ft.query(l - 1) : 0);
                ptr++;
            }
        }

        for (int num : res) {
            out.println(num);
        }

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

    /**
     * 树状数组实现
     */
    static class FenwickTree {
        int[] tree;
```

```
        int n;

        public FenwickTree(int size) {
            this.n = size;
            this.tree = new int[n + 1];
        }

        /**
         * 单点更新
         * @param idx 索引
         * @param val 值
         */
        public void update(int idx, int val) {
            idx++;
            while (idx <= n) {
                tree[idx] += val;
                idx += idx & -idx;
            }
        }

        /**
         * 前缀和查询
         * @param idx 索引
         * @return 前缀和
         */
        public int query(int idx) {
            idx++;
            int res = 0;
            while (idx > 0) {
                res += tree[idx];
                idx -= idx & -idx;
            }
            return res;
        }
    }
}

// 题目6: 第一次出现位置序列查询
public static class FirstOccurrence {
    static class Node {
        int left, right, minVal;
        public Node(int left, int right, int minVal) {
            this.left = left;
```

```
            this.right = right;
            this.minVal = minVal;
        }
    }

    static List<Node> nodes;
    static List<Integer> roots;

    public static void solve() throws IOException {
        Reader in = new Reader();
        PrintWriter out = new PrintWriter(System.out);

        int n = in.nextInt();
        int q = in.nextInt();
        int[] arr = new int[n];
        for (int i = 0; i < n; i++) {
            arr[i] = in.nextInt();
        }

        // 初始化
        nodes = new ArrayList<>();
        roots = new ArrayList<>();
        nodes.add(new Node(0, 0, Integer.MAX_VALUE));
        roots.add(0);

        Map<Integer, Integer> lastOccurrence = new HashMap<>();

        // 从右往左构建
        for (int i = n - 1; i >= 0; i--) {
            int currentRoot = roots.get(roots.size() - 1);

            if (lastOccurrence.containsKey(arr[i])) {
                // 更新之前的位置
                currentRoot = update(currentRoot, 1, n, lastOccurrence.get(arr[i]) + 1,
Integer.MAX_VALUE);
            }

            // 更新当前位置
            currentRoot = update(currentRoot, 1, n, i + 1, i + 1);
            lastOccurrence.put(arr[i], i);
            roots.add(currentRoot);
        }
```

```
        // 反转 roots 数组
        Collections.reverse(roots);

        // 处理查询
        for (int i = 0; i < q; i++) {
            int l = in.nextInt();
            int r = in.nextInt();
            int minPos = queryMin(roots.get(r), 1, n, l, r);
            out.println(minPos == Integer.MAX_VALUE ? -1 : minPos);
        }

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

    /**
     * 更新线段树节点
     * @param preRoot 前一个版本的根节点
     * @param l 当前区间左端点
     * @param r 当前区间右端点
     * @param pos 要更新的位置
     * @param val 要设置的值
     * @return 新版本的根节点
     */
    private static int update(int preRoot, int l, int r, int pos, int val) {
        nodes.add(new Node(
            nodes.get(preRoot).left,
            nodes.get(preRoot).right,
            nodes.get(preRoot).minVal
        ));
        int newRoot = nodes.size() - 1;

        if (l == r) {
            nodes.get(newRoot).minVal = val;
            return newRoot;
        }

        int mid = l + (r - l) / 2;
        if (pos <= mid) {
            nodes.get(newRoot).left = update(nodes.get(preRoot).left, l, mid, pos, val);
        } else {
            nodes.get(newRoot).right = update(nodes.get(preRoot).right, mid + 1, r, pos,
val);
```

```
        }

        // 向上更新
        nodes.get(newRoot).minVal = Math.min(
            nodes.get(nodes.get(newRoot).left).minVal,
            nodes.get(nodes.get(newRoot).right).minVal
        );
        return newRoot;
    }

    /**
     * 查询区间最小值
     * @param root 根节点
     * @param l 当前区间左端点
     * @param r 当前区间右端点
     * @param ql 查询区间左端点
     * @param qr 查询区间右端点
     * @return 区间最小值
     */
    private static int queryMin(int root, int l, int r, int ql, int qr) {
        if (qr < l || ql > r) {
            return Integer.MAX_VALUE;
        }

        if (ql <= l && r <= qr) {
            return nodes.get(root).minVal;
        }

        int mid = l + (r - l) / 2;
        int leftMin = queryMin(nodes.get(root).left, l, mid, ql, qr);
        int rightMin = queryMin(nodes.get(root).right, mid + 1, r, ql, qr);
        return Math.min(leftMin, rightMin);
    }
}

// 题目 7: 区间最小缺失自然数查询（区间 Mex 查询）
public static class RangeMex {
    static class Node {
        int left, right, minPos;
        public Node(int left, int right, int minPos) {
            this.left = left;
            this.right = right;
            this.minPos = minPos;
```

```
    }
}

static List<Node> nodes;
static List<Integer> roots;

public static void solve() throws IOException {
    Reader in = new Reader();
    PrintWriter out = new PrintWriter(System.out);

    int n = in.nextInt();
    int q = in.nextInt();
    int[] arr = new int[n];
    for (int i = 0; i < n; i++) {
        arr[i] = in.nextInt();
    }

    // 初始化
    nodes = new ArrayList<>();
    roots = new ArrayList<>();
    nodes.add(new Node(0, 0, Integer.MAX_VALUE));
    roots.add(0);

    int maxVal = 0;
    for (int num : arr) {
        if (num >= 0) {
            maxVal = Math.max(maxVal, num);
        }
    }
    maxVal = Math.max(maxVal, n);

    Map<Integer, Integer> lastOccurrence = new HashMap<>();

    for (int i = 0; i < n; i++) {
        int val = arr[i];
        int newRoot = roots.get(i);

        if (val >= 0) {
            if (lastOccurrence.containsKey(val)) {
                newRoot = update(newRoot, 0, maxVal, val, i);
            } else {
                newRoot = update(newRoot, 0, maxVal, val, i);
            }
```

```
            lastOccurrence.put(val, i);
        }

        roots.add(newRoot);
    }

    // 处理查询
    for (int i = 0; i < q; i++) {
        int l = in.nextInt() - 1;
        int r = in.nextInt() - 1;
        int mex = queryMex(roots.get(r + 1), 0, maxVal, l);
        out.println(mex);
    }

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

/**
 * 更新线段树节点
 * @param preRoot 前一个版本的根节点
 * @param l 当前区间左端点
 * @param r 当前区间右端点
 * @param pos 要更新的位置
 * @param val 要设置的值
 * @return 新版本的根节点
 */
private static int update(int preRoot, int l, int r, int pos, int val) {
    nodes.add(new Node(
        nodes.get(preRoot).left,
        nodes.get(preRoot).right,
        nodes.get(preRoot).minPos
    ));
    int newRoot = nodes.size() - 1;

    if (l == r) {
        nodes.get(newRoot).minPos = val;
        return newRoot;
    }

    int mid = l + (r - l) / 2;
    if (pos <= mid) {
        nodes.get(newRoot).left = update(nodes.get(preRoot).left, l, mid, pos, val);
```

```
            } else {
                nodes.get(newRoot).right = update(nodes.get(preRoot).right, mid + 1, r, pos,
val);
            }

            nodes.get(newRoot).minPos = Math.min(
                nodes.get(nodes.get(newRoot).left).minPos,
                nodes.get(nodes.get(newRoot).right).minPos
            );
            return newRoot;
        }

        /**
         * 查询区间 Mex
         * @param root 根节点
         * @param l 当前区间左端点
         * @param r 当前区间右端点
         * @param currentL 当前查询左端点
         * @return Mex 值
         */
        private static int queryMex(int root, int l, int r, int currentL) {
            if (l == r) {
                return l;
            }

            int mid = l + (r - l) / 2;
            int leftMin = nodes.get(nodes.get(root).left).minPos;

            if (leftMin < currentL) {
                return queryMex(nodes.get(root).left, l, mid, currentL);
            } else {
                return queryMex(nodes.get(root).right, mid + 1, r, currentL);
            }
        }
    }

    // 高效读写类
    static class Reader {
        private final InputStreamReader reader;
        private final BufferedReader br;
        private StringTokenizer st;

        public Reader() {
```

```
        reader = new InputStreamReader(System.in);
        br = new BufferedReader(reader, 32768);
        st = new StringTokenizer("");
    }

    public String nextLine() throws IOException {
        return br.readLine();
    }

    public String next() throws IOException {
        while (!st.hasMoreTokens()) {
            st = new StringTokenizer(br.readLine());
        }
        return st.nextToken();
    }

    public int nextInt() throws IOException {
        return Integer.parseInt(next());
    }

    public long nextLong() throws IOException {
        return Long.parseLong(next());
    }
}

public static void main(String[] args) throws IOException {
    // 测试各个问题
    // 可以根据需要取消注释相应的测试函数
    // MKTHNUM.solve();
    // COT.solve();
    // DQUERY.solve();
    // FirstOccurrence.solve();
    // RangeMex.solve();

    // 测试 LeetCode 2276
    CountIntervals intervals = new CountIntervals();
    intervals.add(2, 3);
    intervals.add(7, 10);
    System.out.println(intervals.count());  // 输出: 6
    intervals.add(5, 8);
    System.out.println(intervals.count());  // 输出: 8

    // 测试 LeetCode 1970
```

```
        int[] parents = {-1, 0, 0, 2};
        int[] nums = {1, 2, 3, 4};
        int[] res = SmallestMissingGeneticValue.smallestMissingValueSubtree(parents, nums);
        System.out.println(Arrays.toString(res));  // 输出: [5, 1, 1, 1]
    }
}

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

文件: persistent_segment_tree_solutions.cpp
==========================================
#include <iostream>
#include <vector>
#include <algorithm>
#include <map>
#include <queue>
#include <climits>
#include <cstring>
using namespace std;

/**
 * 持久化线段树（主席树）完整解决方案
 * 包含所有经典题目和详细实现
 * 时间复杂度分析：构建O(n log n)，单点更新O(log n)，区间查询O(log n)
 * 空间复杂度分析：O(n log n)
 * 最优解分析：对于区间第K小、历史版本查询等问题，持久化线段树是最优解之一
 */

// 通用持久化线段树节点定义
struct Node {
    int left, right, count;
    Node(int l = 0, int r = 0, int c = 0) : left(l), right(r), count(c) {}
};

// 题目1: SPOJ MKTHNUM - K-th Number
// 题目链接: https://www.spoj.com/problems/MKTHNUM/
// 题目描述: 给定一个数组，多次查询区间第K小的数
// 最优解: 持久化线段树是该问题的最优解之一
namespace MKTHNUM {
    vector<Node> nodes;
    vector<int> roots;

    void push_up(int node_idx) {
```

```
    nodes[node_idx].count = nodes[nodes[node_idx].left].count +
                            nodes[nodes[node_idx].right].count;
}

int update(int pre_root, int l, int r, int pos, int val) {
    // 创建新节点，复制前驱版本的信息
    nodes.push_back(nodes[pre_root]);
    int new_root = nodes.size() - 1;

    if (l == r) {
        nodes[new_root].count += val;
        return new_root;
    }

    int mid = l + (r - l) / 2;
    if (pos <= mid) {
        nodes[new_root].left = update(nodes[pre_root].left, l, mid, pos, val);
    } else {
        nodes[new_root].right = update(nodes[pre_root].right, mid + 1, r, pos, val);
    }

    push_up(new_root);
    return new_root;
}

int query(int root1, int root2, int l, int r, int k) {
    if (l == r) {
        return l;
    }

    int mid = l + (r - l) / 2;
    int left_count = nodes[nodes[root2].left].count - nodes[nodes[root1].left].count;

    if (k <= left_count) {
        return query(nodes[root1].left, nodes[root2].left, l, mid, k);
    } else {
        return query(nodes[root1].right, nodes[root2].right, mid + 1, r, k - left_count);
    }
}

void solve() {
    ios::sync_with_stdio(false);
    cin.tie(0);
```

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

        vector<int> arr(n);
        for (int i = 0; i < n; i++) {
            cin >> arr[i];
        }

        // 离散化
        vector<int> sorted_arr = arr;
        sort(sorted_arr.begin(), sorted_arr.end());
        sorted_arr.erase(unique(sorted_arr.begin(), sorted_arr.end()), sorted_arr.end());

        map<int, int> rank;
        for (int i = 0; i < sorted_arr.size(); i++) {
            rank[sorted_arr[i]] = i + 1;  // 映射到1-based
        }

        // 初始化
        nodes.clear();
        roots.clear();
        nodes.push_back(Node());  // 哨兵节点
        roots.push_back(0);

        // 构建持久化线段树
        for (int i = 0; i < n; i++) {
            int r = update(roots.back(), 1, sorted_arr.size(), rank[arr[i]], 1);
            roots.push_back(r);
        }

        // 处理查询
        for (int i = 0; i < m; i++) {
            int l, r, k;
            cin >> l >> r >> k;
            l--;  // 转换为0-based

            int pos = query(roots[l], roots[r], 1, sorted_arr.size(), k);
            // 转换回原始值
            cout << sorted_arr[pos - 1] << '\n';
        }
    }
}
```

```
// 题目 2: SPOJ COT - Count on a Tree
// 题目链接: https://www.spoj.com/problems/COT/
// 题目描述: 给定一棵树，多次查询两点之间路径上的第 K 小的数
// 最优解: 树上持久化线段树（树链剖分+主席树）是该问题的最优解
namespace COT {
    struct Node {
        int left, right, count;
        Node(int l = 0, int r = 0, int c = 0) : left(l), right(r), count(c) {}
    };

    vector<Node> nodes;
    vector<int> roots;
    vector<vector<int>> graph;
    vector<int> values;
    vector<vector<int>> parent;
    vector<int> depth;
    const int LOG = 20;

    void push_up(int node_idx) {
        nodes[node_idx].count = nodes[nodes[node_idx].left].count +
                                nodes[nodes[node_idx].right].count;
    }

    int update(int pre_root, int l, int r, int pos, int val) {
        nodes.push_back(nodes[pre_root]);
        int new_root = nodes.size() - 1;

        if (l == r) {
            nodes[new_root].count += val;
            return new_root;
        }

        int mid = l + (r - l) / 2;
        if (pos <= mid) {
            nodes[new_root].left = update(nodes[pre_root].left, l, mid, pos, val);
        } else {
            nodes[new_root].right = update(nodes[pre_root].right, mid + 1, r, pos, val);
        }

        push_up(new_root);
        return new_root;
    }
```

```
void dfs(int u, int p, const map<int, int>& rank) {
    parent[0][u] = p;
    depth[u] = depth[p] + 1;

    // 继承父节点的版本并更新当前节点的值
    roots[u] = update(roots[p], 1, rank.size(), rank.at(values[u - 1]), 1);

    for (int v : graph[u]) {
        if (v != p) {
            dfs(v, u, rank);
        }
    }
}

int lca(int u, int v) {
    if (depth[u] < depth[v]) {
        swap(u, v);
    }

    // 将 u 提升到与 v 同一深度
    for (int k = LOG - 1; k >= 0; k--) {
        if (depth[u] - (1 << k) >= depth[v]) {
            u = parent[k][u];
        }
    }

    if (u == v) {
        return u;
    }

    for (int k = LOG - 1; k >= 0; k--) {
        if (parent[k][u] != parent[k][v]) {
            u = parent[k][u];
            v = parent[k][v];
        }
    }

    return parent[0][u];
}

int query(int u, int v, int ancestor, int ancestor_parent, int l, int r, int k) {
    if (l == r) {
```

```
            return l;
        }

        int mid = l + (r - l) / 2;
        int left_count = nodes[nodes[u].left].count + nodes[nodes[v].left].count
                       - nodes[nodes[ancestor].left].count -
nodes[nodes[ancestor_parent].left].count;

        if (k <= left_count) {
            return query(nodes[u].left, nodes[v].left,
                         nodes[ancestor].left, nodes[ancestor_parent].left,
                         l, mid, k);
        } else {
            return query(nodes[u].right, nodes[v].right,
                         nodes[ancestor].right, nodes[ancestor_parent].right,
                         mid + 1, r, k - left_count);
        }
    }

    void solve() {
        ios::sync_with_stdio(false);
        cin.tie(0);

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

        values.resize(n);
        for (int i = 0; i < n; i++) {
            cin >> values[i];
        }

        // 离散化
        vector<int> sorted_vals = values;
        sort(sorted_vals.begin(), sorted_vals.end());
        sorted_vals.erase(unique(sorted_vals.begin(), sorted_vals.end()), sorted_vals.end());

        map<int, int> rank;
        for (int i = 0; i < sorted_vals.size(); i++) {
            rank[sorted_vals[i]] = i + 1;
        }

        // 构建图
        graph.resize(n + 1);
```

```
        for (int i = 0; i < n - 1; i++) {
            int u, v;
            cin >> u >> v;
            graph[u].push_back(v);
            graph[v].push_back(u);
        }

        // 初始化 LCA 数组
        parent.resize(LOG, vector<int>(n + 1, 0));
        depth.resize(n + 1, 0);

        // 初始化持久化线段树
        nodes.clear();
        roots.resize(n + 1, 0);
        nodes.push_back(Node());

        // DFS 构建
        dfs(1, 0, rank);

        // 构建 LCA 倍增表
        for (int k = 1; k < LOG; k++) {
            for (int i = 1; i <= n; i++) {
                parent[k][i] = parent[k - 1][parent[k - 1][i]];
            }
        }

        // 处理查询
        for (int i = 0; i < m; i++) {
            int u, v, k;
            cin >> u >> v >> k;
            int ancestor = lca(u, v);
            int res = query(u, v, ancestor, parent[0][ancestor], 1, rank.size(), k);
            // 转换回原始值
            cout << sorted_vals[res - 1] << '\n';
        }
    }
}

// 题目 3: LeetCode 2276 - Count Integers in Intervals
// 题目链接: https://leetcode.com/problems/count-integers-in-intervals/
// 题目描述: 实现一个数据结构，支持添加区间和查询区间内整数的个数
// 最优解: 动态开点线段树是该问题的最优解之一
class CountIntervals {
```

```
private:
    struct Node {
        Node *left, *right;
        int cnt, cover;
        Node() : left(nullptr), right(nullptr), cnt(0), cover(0) {}
    };

    Node* root;
    int total;
    const int MAX = 1e9;

    void push_up(Node* node, int l, int r) {
        if (node->cover > 0) {
            node->cnt = r - l + 1;
        } else if (l == r) {
            node->cnt = 0;
        } else {
            int left_cnt = node->left ? node->left->cnt : 0;
            int right_cnt = node->right ? node->right->cnt : 0;
            node->cnt = left_cnt + right_cnt;
        }
    }

    void update(Node* node, int l, int r, int ul, int ur, int val) {
        if (ur < l || ul > r) {
            return;
        }

        if (ul <= l && r <= ur) {
            node->cover += val;
            push_up(node, l, r);
            return;
        }

        // 动态开点
        if (!node->left) node->left = new Node();
        if (!node->right) node->right = new Node();

        int mid = l + (r - l) / 2;
        update(node->left, l, mid, ul, ur, val);
        update(node->right, mid + 1, r, ul, ur, val);
        push_up(node, l, r);
    }
```

```
    // 清理内存
    void clear(Node* node) {
        if (node) {
            clear(node->left);
            clear(node->right);
            delete node;
        }
    }

public:
    CountIntervals() : root(new Node()), total(0) {}

    ~CountIntervals() {
        clear(root);
    }

    void add(int left, int right) {
        int before = root->cnt;
        update(root, 1, MAX, left, right, 1);
        total = root->cnt;
    }

    int count() {
        return total;
    }
};

// 题目 4: LeetCode 1970 - Smallest Missing Genetic Value in Each Subtree
// 题目链接: https://leetcode.com/problems/smallest-missing-genetic-value-in-each-subtree/
// 题目描述: 给定一棵树，每个节点有一个基因值，求每个子树中最小缺失的基因值
namespace SmallestMissingGeneticValue {
    int find(int* parent, int u) {
        if (parent[u] != u) {
            parent[u] = find(parent, parent[u]);
        }
        return parent[u];
    }

    void dfs(int node, const vector<vector<int>>& children, const vector<int>& nums,
             int* parent, vector<int>& res, vector<bool>& visited) {
        visited[node] = true;
```

```
    for (int child : children[node]) {
        if (!visited[child]) {
            dfs(child, children, nums, parent, res, visited);
        }
    }

    // 合并当前节点的值
    int u = find(parent, nums[node]);
    parent[u] = u + 1;

    if (nums[node] == 1) {
        res[node] = find(parent, 1);
    }

    // 向上传递结果
    int curr = node;
    while (parent[curr] != -1) {
        curr = parent[curr];
        if (curr >= 0 && curr < res.size() && res[curr] == 1) {
            res[curr] = find(parent, 1);
        } else {
            break;
        }
    }
}

vector<int> solve(vector<int>& parents, vector<int>& nums) {
    int n = parents.size();
    vector<int> res(n, 1);

    // 检查是否存在值为1的节点
    int posOfOne = -1;
    for (int i = 0; i < n; i++) {
        if (nums[i] == 1) {
            posOfOne = i;
            break;
        }
    }
    if (posOfOne == -1) {
        return res;
    }

    // 构建子树
```

```
        vector<vector<int>> children(n);
        for (int i = 0; i < n; i++) {
            if (parents[i] != -1) {
                children[parents[i]].push_back(i);
            }
        }

        // 并查集
        const int MAX_VAL = 100001;
        int* parent = new int[MAX_VAL + 1];
        for (int i = 0; i <= MAX_VAL; i++) {
            parent[i] = i;
        }

        // 后序遍历
        vector<bool> visited(n, false);
        dfs(posOfOne, children, nums, parent, res, visited);

        delete[] parent;
        return res;
    }
}

// 题目 5: SPOJ DQUERY - D-query
// 题目链接: https://www.spoj.com/problems/DQUERY/
// 题目描述: 给定一个数组，多次查询区间内不同元素的个数
namespace DQUERY {
    struct FenwickTree {
        vector<int> tree;
        int n;

        FenwickTree(int size) : n(size), tree(size + 1, 0) {}

        void update(int idx, int val) {
            idx++;  // 1-based
            while (idx <= n) {
                tree[idx] += val;
                idx += idx & -idx;
            }
        }

        int query(int idx) {
            idx++;  // 1-based
```

```
        int res = 0;
        while (idx > 0) {
            res += tree[idx];
            idx -= idx & -idx;
        }
        return res;
    }
};

void solve() {
    ios::sync_with_stdio(false);
    cin.tie(0);

    int n;
    cin >> n;
    vector<int> arr(n);
    for (int i = 0; i < n; i++) {
        cin >> arr[i];
    }

    int m;
    cin >> m;
    vector<tuple<int, int, int>> queries;
    for (int i = 0; i < m; i++) {
        int l, r;
        cin >> l >> r;
        l--; r--;  // 转换为0-based
        queries.emplace_back(r, l, i);
    }

    // 按右端点排序
    sort(queries.begin(), queries.end());

    FenwickTree ft(n);
    vector<int> res(m);
    map<int, int> last;
    int ptr = 0;

    for (int i = 0; i < n; i++) {
        if (last.count(arr[i])) {
            ft.update(last[arr[i]], -1);
        }
        ft.update(i, 1);
```

```
            last[arr[i]] = i;

            while (ptr < m && get<0>(queries[ptr]) == i) {
                int l = get<1>(queries[ptr]);
                int q_idx = get<2>(queries[ptr]);
                res[q_idx] = ft.query(i) - (l > 0 ? ft.query(l - 1) : 0);
                ptr++;
            }
        }

        for (int num : res) {
            cout << num << '\n';
        }
    }
}

// 题目 6: 第一次出现位置序列查询
namespace FirstOccurrence {
    struct Node {
        int left, right, minVal;
        Node(int l = 0, int r = 0, int mv = INT_MAX) : left(l), right(r), minVal(mv) {}
    };

    vector<Node> nodes;
    vector<int> roots;

    void push_up(int node_idx) {
        nodes[node_idx].minVal = min(nodes[nodes[node_idx].left].minVal,
                                     nodes[nodes[node_idx].right].minVal);
    }

    int update(int pre_root, int l, int r, int pos, int val) {
        nodes.push_back(nodes[pre_root]);
        int new_root = nodes.size() - 1;

        if (l == r) {
            nodes[new_root].minVal = val;
            return new_root;
        }

        int mid = l + (r - l) / 2;
        if (pos <= mid) {
            nodes[new_root].left = update(nodes[pre_root].left, l, mid, pos, val);
```

```
    } else {
        nodes[new_root].right = update(nodes[pre_root].right, mid + 1, r, pos, val);
    }

    push_up(new_root);
    return new_root;
}

int query_min(int root, int l, int r, int ql, int qr) {
    if (qr < l || ql > r) {
        return INT_MAX;
    }

    if (ql <= l && r <= qr) {
        return nodes[root].minVal;
    }

    int mid = l + (r - l) / 2;
    int left_min = query_min(nodes[root].left, l, mid, ql, qr);
    int right_min = query_min(nodes[root].right, mid + 1, r, ql, qr);
    return min(left_min, right_min);
}

void solve() {
    ios::sync_with_stdio(false);
    cin.tie(0);

    int n, q;
    cin >> n >> q;
    vector<int> arr(n);
    for (int i = 0; i < n; i++) {
        cin >> arr[i];
    }

    // 初始化
    nodes.clear();
    roots.clear();
    nodes.push_back(Node());
    roots.push_back(0);

    map<int, int> last_occurrence;

    // 从右往左构建
```

```
        for (int i = n - 1; i >= 0; i--) {
            int current_root = roots.back();

            if (last_occurrence.count(arr[i])) {
                // 更新之前的位置
                current_root = update(current_root, 1, n, last_occurrence[arr[i]] + 1, INT_MAX);
            }

            // 更新当前位置
            current_root = update(current_root, 1, n, i + 1, i + 1);
            last_occurrence[arr[i]] = i;
            roots.push_back(current_root);
        }

        // 反转 roots 数组
        reverse(roots.begin(), roots.end());

        // 处理查询
        for (int i = 0; i < q; i++) {
            int l, r;
            cin >> l >> r;
            int min_pos = query_min(roots[r], 1, n, l, r);
            cout << (min_pos == INT_MAX ? -1 : min_pos) << '\n';
        }
    }
}

// 题目 7: 区间最小缺失自然数查询（区间 Mex 查询）
namespace RangeMex {
    struct Node {
        int left, right, minPos;
        Node(int l = 0, int r = 0, int mp = INT_MAX) : left(l), right(r), minPos(mp) {}
    };

    vector<Node> nodes;
    vector<int> roots;

    void push_up(int node_idx) {
        nodes[node_idx].minPos = min(nodes[nodes[node_idx].left].minPos,
                                     nodes[nodes[node_idx].right].minPos);
    }

    int update(int pre_root, int l, int r, int pos, int val) {
```

```
    nodes.push_back(nodes[pre_root]);
    int new_root = nodes.size() - 1;

    if (l == r) {
        nodes[new_root].minPos = val;
        return new_root;
    }

    int mid = l + (r - l) / 2;
    if (pos <= mid) {
        nodes[new_root].left = update(nodes[pre_root].left, l, mid, pos, val);
    } else {
        nodes[new_root].right = update(nodes[pre_root].right, mid + 1, r, pos, val);
    }

    push_up(new_root);
    return new_root;
}

int query_mex(int root, int l, int r, int current_l) {
    if (l == r) {
        return l;
    }

    int mid = l + (r - l) / 2;
    int left_min = nodes[nodes[root].left].minPos;

    if (left_min < current_l) {
        return query_mex(nodes[root].left, l, mid, current_l);
    } else {
        return query_mex(nodes[root].right, mid + 1, r, current_l);
    }
}

void solve() {
    ios::sync_with_stdio(false);
    cin.tie(0);

    int n, q;
    cin >> n >> q;
    vector<int> arr(n);
    int max_val = 0;
    for (int i = 0; i < n; i++) {
```

```
            cin >> arr[i];
            if (arr[i] >= 0) {
                max_val = max(max_val, arr[i]);
            }
        }
        max_val = max(max_val, n);

        // 初始化
        nodes.clear();
        roots.clear();
        nodes.push_back(Node());
        roots.push_back(0);

        map<int, int> last_occurrence;

        for (int i = 0; i < n; i++) {
            int val = arr[i];
            int new_root = roots.back();

            if (val >= 0) {
                if (last_occurrence.count(val)) {
                    new_root = update(new_root, 0, max_val, val, i);
                } else {
                    new_root = update(new_root, 0, max_val, val, i);
                }
                last_occurrence[val] = i;
            }

            roots.push_back(new_root);
        }

        // 处理查询
        for (int i = 0; i < q; i++) {
            int l, r;
            cin >> l >> r;
            l--; r--;  // 转换为0-based
            int mex = query_mex(roots[r + 1], 0, max_val, l);
            cout << mex << '\n';
        }
    }
}

// 主函数，用于测试各个问题的解法
```

```
int main() {
    // 根据需要取消注释相应的测试函数
    // MKTHNUM::solve();
    // COT::solve();
    // DQUERY::solve();
    // FirstOccurrence::solve();
    // RangeMex::solve();

    // 测试 LeetCode 2276
    CountIntervals intervals;
    intervals.add(2, 3);
    intervals.add(7, 10);
    cout << "CountIntervals test 1: " << intervals.count() << endl;  // 输出: 6
    intervals.add(5, 8);
    cout << "CountIntervals test 2: " << intervals.count() << endl;  // 输出: 8

    // 测试 LeetCode 1970
    vector<int> parents = {-1, 0, 0, 2};
    vector<int> nums = {1, 2, 3, 4};
    vector<int> res = SmallestMissingGeneticValue::solve(parents, nums);
    cout << "SmallestMissingGeneticValue test: ";
    for (int num : res) {
        cout << num << " ";
    }
    cout << endl;  // 输出: 5 1 1 1

    return 0;
}
```

==========

文件: persistent_segment_tree_solutions.py

==========

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
持久化线段树（主席树）完整解决方案
包含所有经典题目和详细实现
时间复杂度分析：构建 O(n log n)，单点更新 O(log n)，区间查询 O(log n)
空间复杂度分析：O(n log n)
最优解分析：对于区间第 K 小、历史版本查询等问题，持久化线段树是最优解之一
"""
```

```
import sys
import bisect
from typing import List, Dict, Tuple, Optional

class PersistentSegmentTree:
    """
    持久化线段树（主席树）的通用实现
    支持区间第K小查询、区间统计等操作
    """
    class Node:
        """
        线段树节点定义
        left: 左子节点索引
        right: 右子节点索引
        count: 当前区间的元素个数
        """
        def __init__(self, left=0, right=0, count=0):
            self.left = left
            self.right = right
            self.count = count

    def __init__(self):
        # 存储所有版本的根节点
        self.roots = []
        # 存储所有节点
        self.nodes = [self.Node()]  # 索引0为哨兵节点

    def push_up(self, node_idx: int) -> None:
        """
        向上更新节点的count值
        """
        node = self.nodes[node_idx]
        left_count = self.nodes[node.left].count if node.left else 0
        right_count = self.nodes[node.right].count if node.right else 0
        node.count = left_count + right_count

    def update(self, pre_root: int, l: int, r: int, pos: int, val: int) -> int:
        """
        单点更新，基于前驱版本创建新版本
        pre_root: 前驱版本的根节点索引
        l, r: 当前节点表示的区间
        pos: 要更新的位置
        val: 更新的值（+1或-1）
```

```
        return: 新版本的根节点索引
        """
        # 创建新节点，复制前驱版本的子节点信息
        new_node = self.Node(self.nodes[pre_root].left, self.nodes[pre_root].right,
                             self.nodes[pre_root].count)
        self.nodes.append(new_node)
        new_root = len(self.nodes) - 1

        # 到达叶节点，直接更新
        if l == r:
            self.nodes[new_root].count += val
            return new_root

        mid = (l + r) // 2
        # 根据更新位置选择左子树或右子树
        if pos <= mid:
            # 递归更新左子树，将结果保存到新节点的左子节点
            self.nodes[new_root].left = self.update(self.nodes[pre_root].left, l, mid, pos, val)
        else:
            # 递归更新右子树，将结果保存到新节点的右子节点
            self.nodes[new_root].right = self.update(self.nodes[pre_root].right, mid + 1, r, pos,
val)

        # 向上更新 count 值
        self.push_up(new_root)
        return new_root

    def query(self, root1: int, root2: int, l: int, r: int, k: int) -> int:
        """
        查询区间第 K 小的元素
        root1: 前缀版本 v-1 的根节点
        root2: 前缀版本 v 的根节点
        l, r: 当前查询区间
        k: 要查询的第 K 小
        """
        # 到达叶节点，直接返回位置
        if l == r:
            return l

        mid = (l + r) // 2
        # 计算左子树中两个版本的差值，即区间[l, mid]中的元素个数
        left_count = self.nodes[self.nodes[root2].left].count -
self.nodes[self.nodes[root1].left].count
```

```
        # 根据左子树的元素个数决定向左还是向右查询
        if k <= left_count:
            return self.query(self.nodes[root1].left, self.nodes[root2].left, l, mid, k)
        else:
            return self.query(self.nodes[root1].right, self.nodes[root2].right, mid + 1, r, k - left_count)

# 题目 1: SPOJ MKTHNUM - K-th Number
# 题目链接: https://www.spoj.com/problems/MKTHNUM/
# 题目描述: 给定一个数组，多次查询区间第 K 小的数
# 最优解: 持久化线段树是该问题的最优解之一

def solve_mkthnum():
    """
    解决 SPOJ MKTHNUM 问题 - 静态区间第 K 小查询
    时间复杂度: O(n log n + m log n)，其中 n 是数组长度，m 是查询次数
    空间复杂度: O(n log n)
    """
    import sys
    input = sys.stdin.read().split()
    ptr = 0
    n = int(input[ptr])
    ptr += 1
    m = int(input[ptr])
    ptr += 1

    arr = list(map(int, input[ptr:ptr + n]))
    ptr += n

    # 离散化处理
    sorted_arr = sorted(set(arr))
    rank = {x: i + 1 for i, x in enumerate(sorted_arr)}  # 映射到[1, len(sorted_arr)]

    # 构建持久化线段树
    pst = PersistentSegmentTree()
    pst.roots.append(0)  # 初始空版本

    for i in range(n):
        # 每次插入一个元素，生成新版本
        new_root = pst.update(pst.roots[-1], 1, len(sorted_arr), rank[arr[i]], 1)
        pst.roots.append(new_root)
```

```
    # 处理查询
    results = []
    for _ in range(m):
        l = int(input[ptr]) - 1  # 转换为0-based
        ptr += 1
        r = int(input[ptr]) - 1
        ptr += 1
        k = int(input[ptr])
        ptr += 1

        # 查询区间[l+1, r+1]中的第K小（因为roots从版本0开始）
        pos = pst.query(pst.roots[l], pst.roots[r + 1], 1, len(sorted_arr), k)
        results.append(sorted_arr[pos - 1])  # 转换回原始值

    print('\n'.join(map(str, results)))

# 题目2: SPOJ COT - Count on a Tree
# 题目链接: https://www.spoj.com/problems/COT/
# 题目描述: 给定一棵树，多次查询两点之间路径上的第K小的数
# 最优解: 树上持久化线段树（树链剖分+主席树）是该问题的最优解

def solve_cot():
    """
    解决SPOJ COT问题 - 树上路径第K小查询
    使用DFS序和LCA（最近公共祖先）+ 主席树实现
    时间复杂度: O(n log n + m log n)
    空间复杂度: O(n log n)
    """
    import sys
    sys.setrecursionlimit(1 << 25)
    input = sys.stdin.read().split()
    ptr = 0
    n = int(input[ptr])
    ptr += 1
    m = int(input[ptr])
    ptr += 1

    # 读取节点值
    values = list(map(int, input[ptr:ptr + n]))
    ptr += n

    # 离散化处理
    sorted_vals = sorted(set(values))
```

```
rank = {x: i + 1 for i, x in enumerate(sorted_vals)}

# 构建树的邻接表
graph = [[] for _ in range(n + 1)]  # 节点编号从1开始
for _ in range(n - 1):
    u = int(input[ptr])
    ptr += 1
    v = int(input[ptr])
    ptr += 1
    graph[u].append(v)
    graph[v].append(u)

# LCA相关变量
LOG = 20
depth = [0] * (n + 1)
parent = [[0] * (n + 1) for _ in range(LOG)]

# 主席树相关变量
pst = PersistentSegmentTree()
pst.roots = [0] * (n + 1)  # roots[u]表示从根节点到u的路径上的版本

# DFS构建LCA和主席树
def dfs(u, p):
    parent[0][u] = p
    depth[u] = depth[p] + 1

    # 继承父节点的线段树版本，然后更新当前节点的值
    pst.roots[u] = pst.update(pst.roots[p], 1, len(sorted_vals), rank[values[u - 1]], 1)

    for v in graph[u]:
        if v != p:
            dfs(v, u)

# 构建LCA的倍增表
def build_lca():
    for k in range(1, LOG):
        for i in range(1, n + 1):
            parent[k][i] = parent[k - 1][parent[k - 1][i]]

# 查询LCA
def lca(u, v):
    if depth[u] < depth[v]:
        u, v = v, u
```

```
    # 将u提升到与v同一深度
    for k in range(LOG - 1, -1, -1):
        if depth[u] - (1 << k) >= depth[v]:
            u = parent[k][u]

    if u == v:
        return u

    # 同时提升u和v
    for k in range(LOG - 1, -1, -1):
        if parent[k][u] != parent[k][v]:
            u = parent[k][u]
            v = parent[k][v]

    return parent[0][u]

# 定义查询函数
def query_kth(u, v, k):
    ancestor = lca(u, v)
    # 利用容斥原理：路径u-v = 路径root-u + 路径root-v - 2*路径root-ancestor + ancestor
    left = pst.roots[u]
    right = pst.roots[v]
    ancestor_root = pst.roots[ancestor]
    ancestor_parent_root = pst.roots[parent[0][ancestor]]

    def _query(a, b, c, d, l, r, k):
        if l == r:
            return l

        mid = (l + r) // 2
        # 计算左子树中的元素个数：(a+b - c-d)的左子树count
        a_left = pst.nodes[a].left
        b_left = pst.nodes[b].left
        c_left = pst.nodes[c].left
        d_left = pst.nodes[d].left

        left_count = pst.nodes[b_left].count + pst.nodes[a_left].count \
                     - pst.nodes[c_left].count - pst.nodes[d_left].count

        if k <= left_count:
            return _query(pst.nodes[a].left, pst.nodes[b].left,
                          pst.nodes[c].left, pst.nodes[d].left,
```

```
                        l, mid, k)
            else:
                return _query(pst.nodes[a].right, pst.nodes[b].right,
                        pst.nodes[c].right, pst.nodes[d].right,
                        mid + 1, r, k - left_count)

        pos = _query(left, right, ancestor_root, ancestor_parent_root, 1, len(sorted_vals), k)
        return sorted_vals[pos - 1]

    # 初始化
    dfs(1, 0)  # 假设根节点是1
    build_lca()

    # 处理查询
    results = []
    for _ in range(m):
        u = int(input[ptr])
        ptr += 1
        v = int(input[ptr])
        ptr += 1
        k = int(input[ptr])
        ptr += 1
        results.append(query_kth(u, v, k))

    print('\n'.join(map(str, results)))

# 题目3: LeetCode 2276 - Count Integers in Intervals
# 题目链接: https://leetcode.com/problems/count-integers-in-intervals/
# 题目描述: 实现一个数据结构，支持添加区间和查询区间内整数的个数
# 最优解: 动态开点线段树是该问题的最优解之一

class CountIntervals:
    """
    LeetCode 2276 题解 - 计数区间内的整数
    使用动态开点线段树实现
    时间复杂度: add O(log RANGE), count O(1)
    空间复杂度: O(log RANGE)
    """
    class Node:
        def __init__(self, left=None, right=None, cnt=0, cover=0):
            self.left = left      # 左子节点
            self.right = right    # 右子节点
            self.cnt = cnt        # 当前节点覆盖的整数个数
```

```
        self.cover = cover    # 是否完全覆盖

def __init__(self):
    self.root = self.Node()
    self.total = 0  # 记录总数，用于O(1)查询
    self.MAX = 10**9  # 题目中的数值范围

def push_up(self, node, l, r):
    """向上更新节点信息"""
    if node.cover:
        # 如果当前节点完全覆盖，则计数为区间长度
        node.cnt = r - l + 1
    elif l == r:
        # 叶子节点且未覆盖
        node.cnt = 0
    else:
        # 非叶子节点，合并左右子节点的计数
        node.cnt = (node.left.cnt if node.left else 0) + \
                   (node.right.cnt if node.right else 0)

def update(self, node, l, r, ul, ur, val):
    """区间更新"""
    if ur < l or ul > r:
        # 无交集
        return

    if ul <= l and r <= ur:
        # 当前区间被完全包含
        node.cover += val
        self.push_up(node, l, r)
        return

    # 动态开点
    if not node.left:
        node.left = self.Node()
    if not node.right:
        node.right = self.Node()

    mid = (l + r) // 2
    self.update(node.left, l, mid, ul, ur, val)
    self.update(node.right, mid + 1, r, ul, ur, val)
    self.push_up(node, l, r)
```

```
    def add(self, left: int, right: int) -> None:
        """添加一个区间 [left, right]"""
        # 计算添加前的总数
        before = self.root.cnt
        # 更新区间
        self.update(self.root, 1, self.MAX, left, right, 1)
        # 更新总数
        self.total = self.root.cnt

    def count(self) -> int:
        """查询当前覆盖的整数个数"""
        return self.total

# 题目4: LeetCode 1970 - Smallest Missing Genetic Value in Each Subtree
# 题目链接: https://leetcode.com/problems/smallest-missing-genetic-value-in-each-subtree/
# 题目描述: 给定一棵树，每个节点有一个基因值，求每个子树中最小缺失的基因值
# 最优解: 后序遍历 + 并查集（或持久化线段树）是该问题的最优解

def smallest_missing_value_subtree(parents: List[int], nums: List[int]) -> List[int]:
    """
    LeetCode 1970 题解 - 子树中最小缺失基因值
    使用后序遍历 + 并查集优化
    时间复杂度: O(n log n)
    空间复杂度: O(n)
    """
    n = len(parents)
    res = [1] * n  # 初始化为1，因为1是最小可能的缺失值

    # 检查是否存在值为1的节点
    if 1 not in nums:
        return res

    # 构建树的邻接表（子节点列表）
    children = [[] for _ in range(n)]
    root = -1
    for i, p in enumerate(parents):
        if p == -1:
            root = i
        else:
            children[p].append(i)

    # 并查集结构，用于查找下一个缺失的基因值
    parent = list(range(100002))  # 基因值范围是1到100000
```

```
    def find(u):
        while parent[u] != u:
            parent[u] = parent[parent[u]]
            u = parent[u]
        return u

    # 记录每个基因值对应的节点
    pos = {v: i for i, v in enumerate(nums)}

    # 后序遍历树
    visited = [False] * n

    def dfs(node):
        visited[node] = True
        min_missing = 1

        # 遍历所有子节点
        for child in children[node]:
            dfs(child)

        # 合并当前节点的基因值
        u = find(nums[node])
        parent[u] = u + 1

        # 如果当前节点是值为1的节点，那么其子树的最小缺失值是find(1)
        if nums[node] == 1:
            res[node] = find(1)
        # 向上传递结果
        current = node
        while parents[current] != -1:
            if res[parents[current]] == 1:
                res[parents[current]] = find(1)
            current = parents[current]

    # 从值为1的节点开始DFS（因为只有包含1的子树才可能有大于1的缺失值）
    dfs(pos[1])

    return res

# 题目5: SPOJ DQUERY - D-query
# 题目链接: https://www.spoj.com/problems/DQUERY/
# 题目描述: 给定一个数组，多次查询区间内不同元素的个数
```

```
# 最优解: 离线处理 + 树状数组（或持久化线段树）是该问题的最优解

def solve_dquery():
    """
    解决 SPOJ DQUERY 问题 - 区间不同元素个数查询
    使用离线处理 + 树状数组
    时间复杂度: O((n + m) log n)
    空间复杂度: O(n + m)
    """
    import sys
    input = sys.stdin.read().split()
    ptr = 0
    n = int(input[ptr])
    ptr += 1

    arr = list(map(int, input[ptr:ptr + n]))
    ptr += n

    m = int(input[ptr])
    ptr += 1
    queries = []
    for i in range(m):
        l = int(input[ptr]) - 1  # 转换为 0-based
        ptr += 1
        r = int(input[ptr]) - 1
        ptr += 1
        queries.append((l, r, i))

    # 离线处理：按右端点排序
    queries.sort(key=lambda x: x[1])

    # 树状数组实现
    class FenwickTree:
        def __init__(self, size):
            self.n = size
            self.tree = [0] * (self.n + 1)

        def update(self, idx, val):
            """单点更新"""
            idx += 1  # 转换为 1-based
            while idx <= self.n:
                self.tree[idx] += val
                idx += idx & -idx
```

```
        def query(self, idx):
            """前缀和查询"""
            idx += 1  # 转换为1-based
            res = 0
            while idx > 0:
                res += self.tree[idx]
                idx -= idx & -idx
            return res

    # 记录每个元素最后出现的位置
    last = {}
    ft = FenwickTree(n)
    res = [0] * m

    ptr = 0  # 查询指针
    for i in range(n):
        # 如果当前元素之前出现过，移除之前的贡献
        if arr[i] in last:
            ft.update(last[arr[i]], -1)
        # 添加当前元素的贡献
        ft.update(i, 1)
        last[arr[i]] = i

        # 处理所有右端点等于i的查询
        while ptr < m and queries[ptr][1] == i:
            l, r, q_idx = queries[ptr]
            # 区间[l, r]的不同元素个数 = 前缀和(r) - 前缀和(l-1)
            res[q_idx] = ft.query(r) - (ft.query(l - 1) if l > 0 else 0)
            ptr += 1

    print('\n'.join(map(str, res)))

# 题目6: 第一次出现位置序列查询
# 题目描述: 给定一个序列，查询区间[l, r]内，所有元素第一次出现的位置的最小值
# 最优解: 持久化线段树是该问题的最优解

def solve_first_occurrence():
    """
    解决第一次出现位置序列查询问题
    使用从右往左构建的持久化线段树
    时间复杂度: O(n log n + q log n)
    空间复杂度: O(n log n)
```

```
"""
import sys
input = sys.stdin.read().split()
ptr = 0
n = int(input[ptr])
ptr += 1
q = int(input[ptr])
ptr += 1

arr = list(map(int, input[ptr:ptr + n]))
ptr += n

# 从右往左构建持久化线段树
pst = PersistentSegmentTree()
pst.roots.append(0)

# 记录每个元素最后一次出现的位置
last_occurrence = {}

# 自定义更新和查询函数
class Node:
    def __init__(self, left=0, right=0, min_val=float('inf')):
        self.left = left
        self.right = right
        self.min_val = min_val

# 重新定义持久化线段树用于最小值查询
class MinPST:
    def __init__(self):
        self.roots = []
        self.nodes = [Node()]  # 哨兵节点

    def push_up(self, node_idx):
        node = self.nodes[node_idx]
        left_min = self.nodes[node.left].min_val if node.left else float('inf')
        right_min = self.nodes[node.right].min_val if node.right else float('inf')
        node.min_val = min(left_min, right_min)

    def update(self, pre_root, l, r, pos, val):
        new_node = Node(self.nodes[pre_root].left, self.nodes[pre_root].right,
                        self.nodes[pre_root].min_val)
        self.nodes.append(new_node)
        new_root = len(self.nodes) - 1
```

```
            if l == r:
                new_node.min_val = val
                return new_root

            mid = (l + r) // 2
            if pos <= mid:
                new_node.left = self.update(self.nodes[pre_root].left, l, mid, pos, val)
            else:
                new_node.right = self.update(self.nodes[pre_root].right, mid + 1, r, pos, val)

            self.push_up(new_root)
            return new_root

        def query_min(self, root, l, r, ql, qr):
            if qr < l or ql > r:
                return float('inf')

            node = self.nodes[root]
            if ql <= l and r <= qr:
                return node.min_val

            mid = (l + r) // 2
            left_min = self.query_min(node.left, l, mid, ql, qr)
            right_min = self.query_min(node.right, mid + 1, r, ql, qr)
            return min(left_min, right_min)

    min_pst = MinPST()
    min_pst.roots.append(0)

    for i in range(n - 1, -1, -1):
        # 当前版本继承上一版本
        current_root = min_pst.roots[-1]

        # 如果当前元素之前出现过，需要更新其位置的最小值
        if arr[i] in last_occurrence:
            # 将之前的位置更新为无穷大
            current_root = min_pst.update(current_root, 1, n, last_occurrence[arr[i]] + 1,
float('inf'))

        # 更新当前元素的位置
        current_root = min_pst.update(current_root, 1, n, i + 1, i + 1)  # 转换为1-based
        last_occurrence[arr[i]] = i
```

```
        min_pst.roots.append(current_root)

    # 反转 roots 数组，使得 roots[i]对应前 i 个元素的版本
    min_pst.roots = min_pst.roots[::-1]

    # 处理查询
    results = []
    for _ in range(q):
        l = int(input[ptr])
        ptr += 1
        r = int(input[ptr])
        ptr += 1
        # 查询区间[l, r]内的最小值
        # 注意版本号的对应关系
        min_pos = min_pst.query_min(min_pst.roots[r], 1, n, l, r)
        results.append(min_pos if min_pos != float('inf') else -1)

    print('\n'.join(map(str, results)))

# 题目 7: 区间最小缺失自然数查询（区间 Mex 查询）
# 题目描述: 给定一个数组，多次查询区间[l, r]内的最小缺失自然数
# 最优解: 持久化线段树结合离散化是该问题的最优解

def solve_range_mex():
    """
    解决区间最小缺失自然数查询问题
    使用持久化线段树维护每个位置的最晚出现位置
    时间复杂度: O(n log n + q log n)
    空间复杂度: O(n log n)
    """
    import sys
    input = sys.stdin.read().split()
    ptr = 0
    n = int(input[ptr])
    ptr += 1
    q = int(input[ptr])
    ptr += 1

    arr = list(map(int, input[ptr:ptr + n]))
    ptr += n

    # 离散化处理，只处理非负数
    max_val = max(arr) if arr else 0
```

```
# 构建持久化线段树，维护每个数的最晚出现位置

class MexPST:
    class Node:
        def __init__(self, left=0, right=0, min_pos=0):
            self.left = left
            self.right = right
            self.min_pos = min_pos  # 区间内元素的最小位置

    def __init__(self):
        self.roots = []
        self.nodes = [self.Node()]  # 哨兵节点

    def push_up(self, node_idx):
        node = self.nodes[node_idx]
        left_min = self.nodes[node.left].min_pos if node.left else float('inf')
        right_min = self.nodes[node.right].min_pos if node.right else float('inf')
        node.min_pos = min(left_min, right_min)

    def update(self, pre_root, l, r, pos, val):
        new_node = self.Node(self.nodes[pre_root].left, self.nodes[pre_root].right,
                             self.nodes[pre_root].min_pos)
        self.nodes.append(new_node)
        new_root = len(self.nodes) - 1

        if l == r:
            new_node.min_pos = val
            return new_root

        mid = (l + r) // 2
        if pos <= mid:
            new_node.left = self.update(self.nodes[pre_root].left, l, mid, pos, val)
        else:
            new_node.right = self.update(self.nodes[pre_root].right, mid + 1, r, pos, val)

        self.push_up(new_root)
        return new_root

    def query_mex(self, root, l, r, current_l):
        """
        查询最小的 mex，即最小的 k，使得 k 的最晚出现位置 < current_l
        """
        if l == r:
```

```
            return l

        node = self.nodes[root]
        mid = (l + r) // 2

        # 检查左子树是否有元素的最晚出现位置 < current_l
        left_min = self.nodes[node.left].min_pos if node.left else float('inf')
        if left_min < current_l:
            return self.query_mex(node.left, l, mid, current_l)
        else:
            return self.query_mex(node.right, mid + 1, r, current_l)

mex_pst = MexPST()
mex_pst.roots.append(0)  # 初始版本

# 记录每个数最后一次出现的位置
last_occurrence = {}

# 构建版本树
for i in range(n):
    val = arr[i]
    # 只处理非负数
    if val >= 0:
        # 如果当前值之前出现过，需要更新其最晚位置
        if val in last_occurrence:
            # 更新当前版本
            new_root = mex_pst.update(mex_pst.roots[-1], 0, max(max_val, n), val, i)
        else:
            # 新增当前值
            new_root = mex_pst.update(mex_pst.roots[-1], 0, max(max_val, n), val, i)
        last_occurrence[val] = i
    else:
        # 对于负数，不影响 mex 查询，直接复制上一版本
        new_root = mex_pst.roots[-1]

    mex_pst.roots.append(new_root)

# 处理查询
results = []
for _ in range(q):
    l = int(input[ptr]) - 1  # 转换为 0-based
    ptr += 1
    r = int(input[ptr]) - 1
```

```
        ptr += 1

        # 查询版本 r+1 中的最小 mex
        mex = mex_pst.query_mex(mex_pst.roots[r + 1], 0, max(max_val, n), 1)
        results.append(mex)

    print('\n'.join(map(str, results)))

# 主函数，用于测试各个问题的解法
if __name__ == "__main__":
    # 根据输入选择要运行的问题
    problem = input("请输入要运行的问题编号 (1-7): ").strip()
    if problem == "1":
        solve_mkthnum()
    elif problem == "2":
        solve_cot()
    elif problem == "3":
        # 测试 LeetCode 2276
        intervals = CountIntervals()
        intervals.add(2, 3)
        intervals.add(7, 10)
        print(intervals.count())  # 输出: 6
        intervals.add(5, 8)
        print(intervals.count())  # 输出: 8
    elif problem == "4":
        # 测试 LeetCode 1970
        parents = [-1, 0, 0, 2]
        nums = [1, 2, 3, 4]
        print(smallest_missing_value_subtree(parents, nums))  # 输出: [5, 1, 1, 1]
    elif problem == "5":
        solve_dquery()
    elif problem == "6":
        solve_first_occurrence()
    elif problem == "7":
        solve_range_mex()
    else:
        print("无效的问题编号")
```

==========

文件: SPOJ_COT.cpp

==========

```
#include <cstdio>
```

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

/**
 * SPOJ COT - Count on a tree
 *
 * 题目描述:
 * 有 n 个节点，编号 1~n，每个节点有权值，有 n-1 条边，所有节点组成一棵树
 * 一共有 m 条查询，每条查询 u v k : 打印 u 号点到 v 号点的路径上，第 k 小的点权
 *
 * 解题思路:
 * 使用可持久化线段树（主席树）解决树上路径第 K 小问题。
 * 1. 对所有点权进行离散化处理
 * 2. 以 DFS 序建立主席树，第 i 个版本表示以节点 i 为根的子树信息
 * 3. 利用 LCA 求解树上两点间路径
 * 4. 通过第 u、v、lca(u,v)、fa[lca(u,v)]四个版本的线段树差值计算路径信息
 * 5. 在线段树上二分查找第 k 小的数
 *
 * 时间复杂度: O((n + m) log n)
 * 空间复杂度: O(n log n)
 *
 * 示例:
 * 输入:
 * 7 3
 * 1 2 3 4 5 6 7
 * 1 2
 * 1 3
 * 2 4
 * 2 5
 * 3 6
 * 3 7
 * 1 4 2
 * 2 6 3
 * 3 7 1
 *
 * 输出:
 * 2
 * 4
 * 3
 */
```

```
const int MAXN = 100010;
const int MAXH = 20;

// 各个节点权值
int arr[MAXN];
// 收集权值排序并且去重做离散化
int sorted[MAXN];

// 链式前向星需要
int head[MAXN];
int to[MAXN << 1];
int next[MAXN << 1];
int cntg = 0;

// 可持久化线段树需要
int root[MAXN];
int left[MAXN * MAXH];
int right[MAXN * MAXH];
int size[MAXN * MAXH];
int cntt = 0;

// 树上倍增找 lca 需要
int deep[MAXN];
int stjump[MAXN][MAXH];
int n, m, s;

/**
 * 查找数字在离散化数组中的位置
 * @param num 要查找的数字
 * @return 离散化后的索引
 */
int kth(int num) {
    int l = 1, r = s, mid;
    while (l <= r) {
        mid = (l + r) / 2;
        if (sorted[mid] == num) {
            return mid;
        } else if (sorted[mid] < num) {
            l = mid + 1;
        } else {
            r = mid - 1;
        }
    }
```

```
    return -1;
}

/**
 * 构建空线段树
 * @param l 区间左端点
 * @param r 区间右端点
 * @return 根节点编号
 */
int build(int l, int r) {
    int rt = ++cntt;
    size[rt] = 0;
    if (l < r) {
        int mid = (l + r) / 2;
        left[rt] = build(l, mid);
        right[rt] = build(mid + 1, r);
    }
    return rt;
}

/**
 * 准备阶段：离散化处理
 */
void prepare() {
    for (int i = 1; i <= n; i++) {
        sorted[i] = arr[i];
    }
    std::sort(sorted + 1, sorted + n + 1);
    s = 1;
    for (int i = 2; i <= n; i++) {
        if (sorted[s] != sorted[i]) {
            sorted[++s] = sorted[i];
        }
    }
    root[0] = build(1, s);
}

/**
 * 添加边
 * @param u 起点
 * @param v 终点
 */
void addEdge(int u, int v) {
```

```
    next[++cntg] = head[u];
    to[cntg] = v;
    head[u] = cntg;
}

/**
 * 在线段树中插入一个值
 * @param jobi 要插入的位置
 * @param l 区间左端点
 * @param r 区间右端点
 * @param i 前一个版本的节点编号
 * @return 新节点编号
 */
int insert(int jobi, int l, int r, int i) {
    int rt = ++cntt;
    left[rt] = left[i];
    right[rt] = right[i];
    size[rt] = size[i] + 1;
    if (l < r) {
        int mid = (l + r) / 2;
        if (jobi <= mid) {
            left[rt] = insert(jobi, l, mid, left[rt]);
        } else {
            right[rt] = insert(jobi, mid + 1, r, right[rt]);
        }
    }
    return rt;
}

/**
 * 查询第 k 小的数
 * @param jobk 第 k 小
 * @param l 区间左端点
 * @param r 区间右端点
 * @param u u 节点的根
 * @param v v 节点的根
 * @param lca lca 节点的根
 * @param lcafa lca 父节点的根
 * @return 第 k 小的数在离散化数组中的位置
 */
int query(int jobk, int l, int r, int u, int v, int lca, int lcafa) {
    if (l == r) {
        return l;
```

```
    }
    int lsize = size[left[u]] + size[left[v]] - size[left[lca]] - size[left[lcafa]];
    int mid = (l + r) / 2;
    if (lsize >= jobk) {
        return query(jobk, l, mid, left[u], left[v], left[lca], left[lcafa]);
    } else {
        return query(jobk - lsize, mid + 1, r, right[u], right[v], right[lca], right[lcafa]);
    }
}

// 迭代版 DFS，防止递归爆栈
int ufe[MAXN][3];
int stackSize, u, f, e;

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

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

/**
 * DFS 构建主席树
 */
void dfs() {
    stackSize = 0;
    push(1, 0, -1);
    while (stackSize > 0) {
        pop();
        if (e == -1) {
            root[u] = insert(kth(arr[u]), 1, s, root[f]);
            deep[u] = deep[f] + 1;
            stjump[u][0] = f;
            for (int p = 1; p < MAXH; p++) {
                stjump[u][p] = stjump[stjump[u][p - 1]][p - 1];
            }
```

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

/**
 * 计算两个节点的最近公共祖先
 * @param a 节点a
 * @param b 节点b
 * @return 最近公共祖先
 */
int lca(int a, int b) {
    if (deep[a] < deep[b]) {
        std::swap(a, b);
    }
    for (int p = MAXH - 1; p >= 0; p--) {
        if (deep[stjump[a][p]] >= deep[b]) {
            a = stjump[a][p];
        }
    }
    if (a == b) {
        return a;
    }
    for (int p = MAXH - 1; p >= 0; p--) {
        if (stjump[a][p] != stjump[b][p]) {
            a = stjump[a][p];
            b = stjump[b][p];
        }
    }
    return stjump[a][0];
}

/**
 * 查询树上路径第k小
 * @param u 起点
```

```
 * @param v 终点
 * @param k 第k小
 * @return 第k小的值
 */
int kth_query(int u, int v, int k) {
    int lcaNode = lca(u, v);
    int i = query(k, 1, s, root[u], root[v], root[lcaNode], root[stjump[lcaNode][0]]);
    return sorted[i];
}

int main() {
    scanf("%d%d", &n, &m);

    for (int i = 1; i <= n; i++) {
        scanf("%d", &arr[i]);
    }

    prepare();

    for (int i = 1; i < n; i++) {
        int u, v;
        scanf("%d%d", &u, &v);
        addEdge(u, v);
        addEdge(v, u);
    }

    dfs();

    for (int i = 1; i <= m; i++) {
        int u, v, k;
        scanf("%d%d%d", &u, &v, &k);
        printf("%d\n", kth_query(u, v, k));
    }

    return 0;
}
```

==========

文件: SPOJ_COT.java

==========

```
package class158;
```

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

/**
 * SPOJ COT - Count on a tree
 *
 * 题目来源：SPOJ
 * 题目链接：https://www.spoj.com/problems/COT/
 *
 * 题目描述:
 * 给定一棵N个节点的树，每个点有一个权值，对于M个询问(u,v,k)，你需要回答u和v这两个节点间路径上的第K小的点权。
 *
 * 解题思路:
 * 使用树上可持久化线段树（树上主席树）结合LCA解决该问题。
 * 1. 对节点权值进行离散化处理
 * 2. 通过DFS遍历树，为每个节点建立主席树
 * 3. 利用DFS序和LCA算法计算树上路径信息
 * 4. 对于查询u到v的路径，利用容斥原理计算路径上的第k小值
 *
 * 时间复杂度: O((n + m) log n)
 * 空间复杂度: O(n log n)
 *
 * 示例:
 * 输入:
 * 5 3
 * 1 2 3 4 5
 * 1 2
 * 1 3
 * 2 4
 * 2 5
 * 4 5 2
 * 3 4 3
 * 1 2 1
 *
 * 输出:
 * 3
 * 4
 * 1
 *
 * 解释:
 * 查询4 5 2：节点4到节点5的路径为[4,2,5]，点权为[4,2,5]，第2小为3
 * 查询3 4 3：节点3到节点4的路径为[3,1,2,4]，点权为[3,1,2,4]，第3小为4
```

```
 * 查询 1 2 1：节点 1 到节点 2 的路径为[1,2]，点权为[1,2]，第 1 小为 1
 */
public class SPOJ_COT {
    static final int MAXN = 100010;
    static final int MAXH = 20;
    static final int MAXT = MAXN * MAXH;

    static int n, m, s;

    // 节点权值
    static int[] arr = new int[MAXN];

    // 离散化后的权值
    static int[] sorted = new int[MAXN];

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

    // 可持久化线段树
    static int[] root = new int[MAXN];
    static int[] left = new int[MAXT];
    static int[] right = new int[MAXT];
    static int[] size = new int[MAXT];
    static int cntt = 0;

    // LCA 倍增
    static int[] deep = new int[MAXN];
    static int[][] stjump = new int[MAXN][MAXH];

    /**
     * 二分查找数字 num 在 sorted 数组中的位置
     * @param num 要查找的数字
     * @return 数字在 sorted 数组中的位置
     */
    static int getId(int num) {
        int left = 1, right = s, mid;
        while (left <= right) {
            mid = (left + right) / 2;
            if (sorted[mid] == num) {
                return mid;
```

```
        } else if (sorted[mid] < num) {
            left = mid + 1;
        } else {
            right = mid - 1;
        }
    }
    return -1;
}

/**
 * 构建空线段树
 * @param l 区间左端点
 * @param r 区间右端点
 * @return 根节点编号
 */
static int build(int l, int r) {
    int rt = ++cntt;
    size[rt] = 0;
    if (l < r) {
        int mid = (l + r) / 2;
        left[rt] = build(l, mid);
        right[rt] = build(mid + 1, r);
    }
    return rt;
}

/**
 * 预处理，对节点权值进行离散化
 */
static void prepare() {
    for (int i = 1; i <= n; i++) {
        sorted[i] = arr[i];
    }
    Arrays.sort(sorted, 1, n + 1);
    s = 1;
    for (int i = 2; i <= n; i++) {
        if (sorted[s] != sorted[i]) {
            sorted[++s] = sorted[i];
        }
    }
    root[0] = build(1, s);
}
```

```
/**
 * 添加边
 * @param u 起点
 * @param v 终点
 */
static void addEdge(int u, int v) {
    next[++cntg] = head[u];
    to[cntg] = v;
    head[u] = cntg;
}

/**
 * 在线段树中插入一个值
 * @param pos 要插入的位置
 * @param l 区间左端点
 * @param r 区间右端点
 * @param pre 前一个版本的节点编号
 * @return 新节点编号
 */
static int insert(int pos, int l, int r, int pre) {
    int rt = ++cntt;
    left[rt] = left[pre];
    right[rt] = right[pre];
    size[rt] = size[pre] + 1;

    if (l < r) {
        int mid = (l + r) / 2;
        if (pos <= mid) {
            left[rt] = insert(pos, l, mid, left[rt]);
        } else {
            right[rt] = insert(pos, mid + 1, r, right[rt]);
        }
    }
    return rt;
}

/**
 * 查询路径上第 k 小的点权
 * @param k 要查询的排名
 * @param l 当前区间左端点
 * @param r 当前区间右端点
 * @param u 节点 u 的根节点
 * @param v 节点 v 的根节点
```

```
 * @param lca 节点 u 和 v 的 LCA 的根节点
 * @param lcafa LCA 父节点的根节点
 * @return 第 k 小的点权在离散化数组中的位置
 */
static int query(int k, int l, int r, int u, int v, int lca, int lcafa) {
    if (l == r) {
        return l;
    }
    // 计算左子树中数的个数
    int lsize = size[left[u]] + size[left[v]] - size[left[lca]] - size[left[lcafa]];
    int mid = (l + r) / 2;
    if (lsize >= k) {
        return query(k, l, mid, left[u], left[v], left[lca], left[lcafa]);
    } else {
        return query(k - lsize, mid + 1, r, right[u], right[v], right[lca], right[lcafa]);
    }
}

/**
 * DFS 遍历树并构建主席树
 * @param u 当前节点
 * @param f 父节点
 */
static void dfs(int u, int f) {
    root[u] = insert(getId(arr[u]), 1, s, root[f]);
    deep[u] = deep[f] + 1;
    stjump[u][0] = f;
    for (int p = 1; p < MAXH; p++) {
        stjump[u][p] = stjump[stjump[u][p - 1]][p - 1];
    }
    for (int ei = head[u]; ei > 0; ei = next[ei]) {
        if (to[ei] != f) {
            dfs(to[ei], u);
        }
    }
}

/**
 * 计算节点 a 和节点 b 的最近公共祖先(LCA)
 * @param a 节点 a
 * @param b 节点 b
 * @return 节点 a 和节点 b 的 LCA
 */
```

```
static int lca(int a, int b) {
    if (deep[a] < deep[b]) {
        int tmp = a;
        a = b;
        b = tmp;
    }
    // 将 a 提升到与 b 同一深度
    for (int p = MAXH - 1; p >= 0; p--) {
        if (deep[stjump[a][p]] >= deep[b]) {
            a = stjump[a][p];
        }
    }
    if (a == b) {
        return a;
    }
    // 同时提升 a 和 b 直到它们的父节点相同
    for (int p = MAXH - 1; p >= 0; p--) {
        if (stjump[a][p] != stjump[b][p]) {
            a = stjump[a][p];
            b = stjump[b][p];
        }
    }
    return stjump[a][0];
}

/**
 * 查询节点 u 到节点 v 路径上第 k 小的点权
 * @param u 起点
 * @param v 终点
 * @param k 要查询的排名
 * @return 第 k 小的点权
 */
static int kth(int u, int v, int k) {
    int lcaNode = lca(u, v);
    int i = query(k, 1, s, root[u], root[v], root[lcaNode], root[stjump[lcaNode][0]]);
    return sorted[i];
}

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

    String[] line = reader.readLine().split(" ");
```

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

        // 读取节点权值
        line = reader.readLine().split(" ");
        for (int i = 1; i <= n; i++) {
            arr[i] = Integer.parseInt(line[i - 1]);
        }

        prepare();

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

        // DFS 构建主席树
        dfs(1, 0);

        // 处理查询
        for (int i = 0; i < m; i++) {
            line = reader.readLine().split(" ");
            int u = Integer.parseInt(line[0]);
            int v = Integer.parseInt(line[1]);
            int k = Integer.parseInt(line[2]);
            writer.println(kth(u, v, k));
        }

        writer.flush();
        writer.close();
        reader.close();
    }
}
```

==========

文件: SPOJ_COT.py

==========

```
#!/usr/bin/env python3
```

```
import sys
import bisect
from collections import deque

"""
SPOJ COT - Count on a tree

题目描述:
有 n 个节点，编号 1~n，每个节点有权值，有 n-1 条边，所有节点组成一棵树
一共有 m 条查询，每条查询 u v k : 打印 u 号点到 v 号点的路径上，第 k 小的点权

解题思路:
使用可持久化线段树（主席树）解决树上路径第 K 小问题。
1. 对所有点权进行离散化处理
2. 以 DFS 序建立主席树，第 i 个版本表示以节点 i 为根的子树信息
3. 利用 LCA 求解树上两点间路径
4. 通过第 u、v、lca(u,v)、fa[lca(u,v)]四个版本的线段树差值计算路径信息
5. 在线段树上二分查找第 k 小的数

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

示例:
输入:
7 3
1 2 3 4 5 6 7
1 2
1 3
2 4
2 5
3 6
3 7
1 4 2
2 6 3
3 7 1

输出:
2
4
3
"""

MAXN = 100010
```

```
MAXH = 20

# 全局变量
arr = [0] * MAXN  # 各个节点权值
sorted_vals = [0] * MAXN  # 收集权值排序并且去重做离散化

# 链式前向星需要
head = [0] * MAXN
to = [0] * (MAXN << 1)
next_edge = [0] * (MAXN << 1)
cntg = 0

# 可持久化线段树需要
root = [0] * MAXN
left = [0] * (MAXN * MAXH)
right = [0] * (MAXN * MAXH)
size = [0] * (MAXN * MAXH)
cntt = 0

# 树上倍增找 lca 需要
deep = [0] * MAXN
stjump = [[0] * MAXH for _ in range(MAXN)]
n, m, s = 0, 0, 0



def kth(num):
    """
    查找数字在离散化数组中的位置
    :param num: 要查找的数字
    :return: 离散化后的索引
    """
    left_idx, right_idx = 1, s
    while left_idx <= right_idx:
        mid = (left_idx + right_idx) // 2
        if sorted_vals[mid] == num:
            return mid
        elif sorted_vals[mid] < num:
            left_idx = mid + 1
        else:
            right_idx = mid - 1
    return -1
```

```
def build(l, r):
    """
    构建空线段树
    :param l: 区间左端点
    :param r: 区间右端点
    :return: 根节点编号
    """
    global cntt
    rt = cntt + 1
    cntt = rt
    size[rt] = 0
    if l < r:
        mid = (l + r) // 2
        left[rt] = build(l, mid)
        right[rt] = build(mid + 1, r)
    return rt


def prepare():
    """
    准备阶段：离散化处理
    """
    global s
    for i in range(1, n + 1):
        sorted_vals[i] = arr[i]

    sorted_vals[1:n+1] = sorted(sorted_vals[1:n+1])
    s = 1
    for i in range(2, n + 1):
        if sorted_vals[s] != sorted_vals[i]:
            s += 1
            sorted_vals[s] = sorted_vals[i]
    root[0] = build(1, s)


def addEdge(u, v):
    """
    添加边
    :param u: 起点
    :param v: 终点
    """
    global cntg
    cntg += 1
```

```
    next_edge[cntg] = head[u]
    to[cntg] = v
    head[u] = cntg


def insert(jobi, l, r, i):
    """
    在线段树中插入一个值
    :param jobi: 要插入的位置
    :param l: 区间左端点
    :param r: 区间右端点
    :param i: 前一个版本的节点编号
    :return: 新节点编号
    """
    global cntt
    rt = cntt + 1
    cntt = rt
    left[rt] = left[i]
    right[rt] = right[i]
    size[rt] = size[i] + 1
    if l < r:
        mid = (l + r) // 2
        if jobi <= mid:
            left[rt] = insert(jobi, l, mid, left[rt])
        else:
            right[rt] = insert(jobi, mid + 1, r, right[rt])
    return rt


def query(jobk, l, r, u, v, lca, lcafa):
    """
    查询第 k 小的数
    :param jobk: 第 k 小
    :param l: 区间左端点
    :param r: 区间右端点
    :param u: u 节点的根
    :param v: v 节点的根
    :param lca: lca 节点的根
    :param lcafa: lca 父节点的根
    :return: 第 k 小的数在离散化数组中的位置
    """
    if l == r:
        return l
```

```
    lsize = size[left[u]] + size[left[v]] - size[left[lca]] - size[left[lcafa]]
    mid = (l + r) // 2
    if lsize >= jobk:
        return query(jobk, l, mid, left[u], left[v], left[lca], left[lcafa])
    else:
        return query(jobk - lsize, mid + 1, r, right[u], right[v], right[lca], right[lcafa])


def dfs():
    """
    DFS 构建主席树（迭代版本，防止递归爆栈）
    """
    # 使用列表模拟栈
    stack = [(1, 0, -1)]  # (u, f, e)

    while stack:
        u, f, e = stack.pop()
        if e == -1:
            root[u] = insert(kth(arr[u]), 1, s, root[f])
            deep[u] = deep[f] + 1
            stjump[u][0] = f
            for p in range(1, MAXH):
                stjump[u][p] = stjump[stjump[u][p - 1]][p - 1]
            e = head[u]
        else:
            e = next_edge[e]

        if e != 0:
            stack.append((u, f, e))
            if to[e] != f:
                stack.append((to[e], u, -1))


def lca(a, b):
    """
    计算两个节点的最近公共祖先
    :param a: 节点 a
    :param b: 节点 b
    :return: 最近公共祖先
    """
    if deep[a] < deep[b]:
        a, b = b, a
```

```
    for p in range(MAXH - 1, -1, -1):
        if deep[stjump[a][p]] >= deep[b]:
            a = stjump[a][p]

    if a == b:
        return a

    for p in range(MAXH - 1, -1, -1):
        if stjump[a][p] != stjump[b][p]:
            a = stjump[a][p]
            b = stjump[b][p]

    return stjump[a][0]


def kth_query(u, v, k):
    """
    查询树上路径第 k 小
    :param u: 起点
    :param v: 终点
    :param k: 第 k 小
    :return: 第 k 小的值
    """
    lca_node = lca(u, v)
    i = query(k, 1, s, root[u], root[v], root[lca_node], root[stjump[lca_node][0]])
    return sorted_vals[i]


def main():
    global n, m
    # 读取输入
    line = sys.stdin.readline().strip().split()
    n = int(line[0])
    m = int(line[1])

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

    prepare()

    for i in range(1, n):
        line = sys.stdin.readline().strip().split()
```

```
        u = int(line[0])
        v = int(line[1])
        addEdge(u, v)
        addEdge(v, u)

    dfs()

    for i in range(1, m + 1):
        line = sys.stdin.readline().strip().split()
        u = int(line[0])
        v = int(line[1])
        k = int(line[2])
        print(kth_query(u, v, k))



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

==========

文件: SPOJ_COT_Java.java

==========

```
package class158;

// Problem: SPOJ COT - Count on a tree
// Link: https://www.spoj.com/problems/COT/
// Description: 给定一棵树，每个节点有一个权值，每次查询路径(u,v)上第 k 小的权值
// Solution: 使用树上主席树解决树上路径第 k 小问题
// Time Complexity: O(nlogn) for preprocessing, O(logn) for each query
// Space Complexity: O(nlogn)

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

public class SPOJ_COT_Java {
    static final int MAXN = 100005;
    static int[] a = new int[MAXN];           // 节点权值
    static int[] b = new int[MAXN];           // 离散化数组
    static int n, m;

    // 邻接表存储树
    static List<Integer>[] G = new List[MAXN];
    static int[] dep = new int[MAXN];         // 节点深度
```

```
static int[] fa = new int[MAXN];          // 节点父亲
static int[][] anc = new int[MAXN][20];   // 倍增祖先

// 主席树节点
static class Node {
    int l, r, sum;
    Node(int l, int r, int sum) {
        this.l = l;
        this.r = r;
        this.sum = sum;
    }
}

static Node[] T = new Node[40 * MAXN];    // 主席树节点数组
static int[] root = new int[MAXN];        // 每个节点的主席树根
static int cnt = 0;                       // 节点计数器

// 创建新节点
static int createNode(int l, int r, int sum) {
    T[++cnt] = new Node(l, r, sum);
    return cnt;
}

// 插入值到主席树
static int insert(int pre, int l, int r, int val) {
    int now = createNode(0, 0, 0);
    T[now].sum = T[pre].sum + 1;

    if (l == r) return now;

    int mid = (l + r) >> 1;
    if (val <= mid) {
        T[now].l = insert(T[pre].l, l, mid, val);
        T[now].r = T[pre].r;
    } else {
        T[now].l = T[pre].l;
        T[now].r = insert(T[pre].r, mid + 1, r, val);
    }
    return now;
}

// 查询路径第 k 小
static int query(int u, int v, int lca, int flca, int l, int r, int k) {
```

```
    if (l == r) return l;

    int mid = (l + r) >> 1;
    int x = T[T[u].l].sum + T[T[v].l].sum - T[T[lca].l].sum - T[T[flca].l].sum;

    if (k <= x) {
        return query(T[u].l, T[v].l, T[lca].l, T[flca].l, l, mid, k);
    } else {
        return query(T[u].r, T[v].r, T[lca].r, T[flca].r, mid + 1, r, k - x);
    }
}

// DFS 构建主席树和预处理倍增
static void dfs(int u, int f, int d) {
    dep[u] = d;
    fa[u] = f;
    anc[u][0] = f;

    // 倍增预处理
    for (int i = 1; (1 << i) <= d; i++) {
        anc[u][i] = anc[anc[u][i-1]][i-1];
    }

    // 在主席树中插入当前节点的权值
    root[u] = insert(root[f], 1, n, getId(a[u]));

    // 递归处理子节点
    for (int v : G[u]) {
        if (v != f) {
            dfs(v, u, d + 1);
        }
    }
}

// 计算 LCA
static int lca(int u, int v) {
    if (dep[u] < dep[v]) {
        int temp = u;
        u = v;
        v = temp;
    }

    // 将 u 提升到和 v 同一深度
```

```
    for (int i = 19; i >= 0; i--) {
        if (dep[u] - (1 << i) >= dep[v]) {
            u = anc[u][i];
        }
    }

    if (u == v) return u;

    // 同时提升 u 和 v 直到相遇
    for (int i = 19; i >= 0; i--) {
        if (anc[u][i] != anc[v][i]) {
            u = anc[u][i];
            v = anc[v][i];
        }
    }

    return anc[u][0];
}

// 离散化
static int getId(int x) {
    return Arrays.binarySearch(b, 1, n + 1, x) + 1;
}

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

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

    line = br.readLine().split(" ");
    for (int i = 1; i <= n; i++) {
        a[i] = Integer.parseInt(line[i - 1]);
        b[i] = a[i];
    }

    // 初始化邻接表
    for (int i = 1; i <= n; i++) {
        G[i] = new ArrayList<>();
    }
```

```
        // 读取边
        for (int i = 1; i < n; i++) {
            line = br.readLine().split(" ");
            int u = Integer.parseInt(line[0]);
            int v = Integer.parseInt(line[1]);
            G[u].add(v);
            G[v].add(u);
        }

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

        // 构建主席树和预处理
        root[0] = createNode(0, 0, 0);
        T[root[0]].l = T[root[0]].r = T[root[0]].sum = 0;
        dfs(1, 0, 0);

        // 处理查询
        for (int i = 0; i < m; i++) {
            line = br.readLine().split(" ");
            int u = Integer.parseInt(line[0]);
            int v = Integer.parseInt(line[1]);
            int k = Integer.parseInt(line[2]);

            int lcaNode = lca(u, v);
            int id = query(root[u], root[v], root[lcaNode], root[fa[lcaNode]], 1, sz, k);
            out.println(b[id]);
        }

        out.close();
    }
}
```

==========

文件: SPOJ_COT_Python.py

==========

```
# Problem: SPOJ COT - Count on a tree
# Link: https://www.spoj.com/problems/COT/
# Description: 给定一棵树，每个节点有一个权值，每次查询路径(u,v)上第k小的权值
# Solution: 使用树上主席树解决树上路径第k小问题
# Time Complexity: O(nlogn) for preprocessing, O(logn) for each query
# Space Complexity: O(nlogn)

import sys
from collections import defaultdict

class Node:
    def __init__(self):
        self.l = 0
        self.r = 0
        self.sum = 0

class PersistentSegmentTree:
    def __init__(self, maxn=100005):
        self.MAXN = maxn
        self.T = [Node() for _ in range(40 * maxn)]
        self.root = [0] * maxn
        self.cnt = 0

    def create_node(self, l=0, r=0, sum=0):
        self.cnt += 1
        self.T[self.cnt].l = l
        self.T[self.cnt].r = r
        self.T[self.cnt].sum = sum
        return self.cnt

    def insert(self, pre, l, r, val):
        now = self.create_node()
        self.T[now].sum = self.T[pre].sum + 1

        if l == r:
            return now

        mid = (l + r) >> 1
        if val <= mid:
            self.T[now].l = self.insert(self.T[pre].l, l, mid, val)
            self.T[now].r = self.T[pre].r
        else:
            self.T[now].l = self.T[pre].l
```

```
            self.T[now].r = self.insert(self.T[pre].r, mid + 1, r, val)
        return now

    def query(self, u, v, lca, flca, l, r, k):
        if l == r:
            return l

        mid = (l + r) >> 1
        x = self.T[self.T[u].l].sum + self.T[self.T[v].l].sum - self.T[self.T[lca].l].sum -
self.T[self.T[flca].l].sum

        if k <= x:
            return self.query(self.T[u].l, self.T[v].l, self.T[lca].l, self.T[flca].l, l, mid, k)
        else:
            return self.query(self.T[u].r, self.T[v].r, self.T[lca].r, self.T[flca].r, mid + 1,
r, k - x)

class TreeSolution:
    def __init__(self, n):
        self.n = n
        self.a = [0] * (n + 1)
        self.b = [0] * (n + 1)
        self.G = defaultdict(list)
        self.dep = [0] * (n + 1)
        self.fa = [0] * (n + 1)
        self.anc = [[0] * 20 for _ in range(n + 1)]
        self.pst = PersistentSegmentTree(n + 1)

    def add_edge(self, u, v):
        self.G[u].append(v)
        self.G[v].append(u)

    def getId(self, x):
        # 二分查找
        left, right = 1, self.n
        while left <= right:
            mid = (left + right) // 2
            if self.b[mid] == x:
                return mid
            elif self.b[mid] < x:
                left = mid + 1
            else:
                right = mid - 1
```

```
        return left

    def dfs(self, u, f, d):
        self.dep[u] = d
        self.fa[u] = f
        self.anc[u][0] = f

        # 倍增预处理
        i = 1
        while (1 << i) <= d:
            self.anc[u][i] = self.anc[self.anc[u][i-1]][i-1]
            i += 1

        # 在主席树中插入当前节点的权值
        self.pst.root[u] = self.pst.insert(self.pst.root[f], 1, self.n, self.getId(self.a[u]))

        # 递归处理子节点
        for v in self.G[u]:
            if v != f:
                self.dfs(v, u, d + 1)

    def lca(self, u, v):
        if self.dep[u] < self.dep[v]:
            u, v = v, u

        # 将 u 提升到和 v 同一深度
        for i in range(19, -1, -1):
            if self.dep[u] - (1 << i) >= self.dep[v]:
                u = self.anc[u][i]

        if u == v:
            return u

        # 同时提升 u 和 v 直到相遇
        for i in range(19, -1, -1):
            if self.anc[u][i] != self.anc[v][i]:
                u = self.anc[u][i]
                v = self.anc[v][i]

        return self.anc[u][0]

def main():
    line = sys.stdin.readline().split()
```

```
n, m = int(line[0]), int(line[1])

line = sys.stdin.readline().split()
solution = TreeSolution(n)

for i in range(1, n + 1):
    solution.a[i] = int(line[i - 1])
    solution.b[i] = solution.a[i]

# 读取边
for i in range(1, n):
    line = sys.stdin.readline().split()
    u, v = int(line[0]), int(line[1])
    solution.add_edge(u, v)

# 离散化
solution.b = solution.b[1:]  # 去掉索引0
solution.b.sort()
# 去重
unique_b = []
for x in solution.b:
    if not unique_b or unique_b[-1] != x:
        unique_b.append(x)
solution.b = [0] + unique_b  # 加回索引0
sz = len(solution.b) - 1

# 重新定义getId方法以适应新的b数组
def get_id(x):
    left, right = 1, len(solution.b) - 1
    while left <= right:
        mid = (left + right) // 2
        if solution.b[mid] == x:
            return mid
        elif solution.b[mid] < x:
            left = mid + 1
        else:
            right = mid - 1
    return left

# 替换getId方法
solution.getId = get_id

# 构建主席树和预处理
```

```
    solution.pst.root[0] = solution.pst.create_node()
    solution.dfs(1, 0, 0)

    # 处理查询
    for _ in range(m):
        line = sys.stdin.readline().split()
        u, v, k = int(line[0]), int(line[1]), int(line[2])

        lcaNode = solution.lca(u, v)
        id = solution.pst.query(solution.pst.root[u], solution.pst.root[v],
                                solution.pst.root[lcaNode],
solution.pst.root[solution.fa[lcaNode]],
                                1, sz, k)
        print(solution.b[id])

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

==========

文件: SPOJ_DQUERY.cpp

==========

```
/**
 * SPOJ DQUERY - D-query
 *
 * 题目描述:
 * 给定一个长度为N的序列，进行Q次查询，每次查询区间[l,r]中不同数字的个数。
 *
 * 解题思路:
 * 使用可持久化线段树（主席树）解决静态区间不同元素个数问题。
 * 1. 对于每个位置，记录该位置的数字上一次出现的位置
 * 2. 从前向后建立主席树，第i棵线段树表示前i个位置的信息
 * 3. 对于位置i，如果该数字之前出现过位置j，则在第i棵线段树中将位置j的计数减1，
 *    位置i的计数加1；否则只在位置i的计数加1
 * 4. 利用前缀和思想，通过第r棵和第l-1棵线段树的差得到区间[l,r]的信息
 * 5. 查询区间不同数字个数即为查询区间内计数大于0的位置个数
 *
 * 时间复杂度: O(n log n + q log n)
 * 空间复杂度: O(n log n)
 *
 * 示例:
 * 输入:
 * 5
```

```
 * 1 1 2 1 3
 * 3
 * 1 5
 * 2 4
 * 3 5
 *
 * 输出:
 * 3
 * 2
 * 3
 */
const int MAXN = 30010;
const int MAXV = 1000010; // 值域范围

// 原始数组
int arr[MAXN];
// pos[v] : 数字 v 上次出现的位置
int pos[MAXV];

// 可持久化线段树需要
int root[MAXN];
int left[MAXN * 20];
int right[MAXN * 20];
int sum[MAXN * 20];
int cnt = 0;

/**
 * 构建空线段树
 * @param l 区间左端点
 * @param r 区间右端点
 * @return 根节点编号
 */
int build(int l, int r) {
    int rt = ++cnt;
    sum[rt] = 0;
    if (l < r) {
        int mid = (l + r) / 2;
        left[rt] = build(l, mid);
        right[rt] = build(mid + 1, r);
    }
    return rt;
}
```

```
/**
 * 在线段树中更新一个位置的值
 * @param pos 要更新的位置
 * @param val 要增加的值
 * @param l 区间左端点
 * @param r 区间右端点
 * @param pre 前一个版本的节点编号
 * @return 新节点编号
 */
int update(int pos, int val, int l, int r, int pre) {
    int rt = ++cnt;
    left[rt] = left[pre];
    right[rt] = right[pre];
    sum[rt] = sum[pre] + val;

    if (l < r) {
        int mid = (l + r) / 2;
        if (pos <= mid) {
            left[rt] = update(pos, val, l, mid, left[rt]);
        } else {
            right[rt] = update(pos, val, mid + 1, r, right[rt]);
        }
    }
    return rt;
}

/**
 * 查询区间不同数字个数
 * @param l 查询区间左端点
 * @param r 查询区间右端点
 * @param u 前一个版本的根节点
 * @param v 当前版本的根节点
 * @return 不同数字个数
 */
int query(int l, int r, int u, int v) {
    if (l == r) {
        return sum[v] - sum[u] > 0 ? 1 : 0;
    }
    int mid = (l + r) / 2;
    return query(l, mid, left[u], left[v]) + query(mid + 1, r, right[u], right[v]);
}
```

==========

文件: SPOJ_DQUERY.java

```
==========================================
package class158;

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

/**
 * SPOJ DQUERY - D-query
 *
 * 题目来源：SPOJ
 * 题目链接：https://www.spoj.com/problems/DQUERY/
 *
 * 题目描述:
 * 给定一个长度为N的序列，进行Q次查询，每次查询区间[l,r]中不同数字的个数。
 *
 * 解题思路:
 * 使用可持久化线段树（主席树）解决静态区间不同元素个数问题。
 * 1. 对于每个位置，记录该位置的数字上一次出现的位置
 * 2. 从前向后建立主席树，第i棵线段树表示前i个位置的信息
 * 3. 对于位置i，如果该数字之前出现过位置j，则在第i棵线段树中将位置j的计数减1，
 *    位置i的计数加1；否则只在位置i的计数加1
 * 4. 利用前缀和思想，通过第r棵和第l-1棵线段树的差得到区间[l,r]的信息
 * 5. 查询区间不同数字个数即为查询区间内计数大于0的位置个数
 *
 * 时间复杂度: O(n log n + q log n)
 * 空间复杂度: O(n log n)
 *
 * 示例:
 * 输入:
 * 5
 * 1 1 2 1 3
 * 3
 * 1 5
 * 2 4
 * 3 5
 *
 * 输出:
 * 3
 * 2
 * 3
 */
```

```
public class SPOJ_DQUERY {
    static final int MAXN = 30010;
    static final int MAXV = 1000010; // 值域范围

    // 原始数组
    static int[] arr = new int[MAXN];
    // pos[v] : 数字 v 上次出现的位置
    static int[] pos = new int[MAXV];

    // 可持久化线段树需要
    static int[] root = new int[MAXN];
    static int[] left = new int[MAXN * 20];
    static int[] right = new int[MAXN * 20];
    static int[] sum = new int[MAXN * 20];
    static int cnt = 0;

    /**
     * 构建空线段树
     * @param l 区间左端点
     * @param r 区间右端点
     * @return 根节点编号
     */
    static int build(int l, int r) {
        int rt = ++cnt;
        sum[rt] = 0;
        if (l < r) {
            int mid = (l + r) / 2;
            left[rt] = build(l, mid);
            right[rt] = build(mid + 1, r);
        }
        return rt;
    }

    /**
     * 在线段树中更新一个位置的值
     * @param pos 要更新的位置
     * @param val 要增加的值
     * @param l 区间左端点
     * @param r 区间右端点
     * @param pre 前一个版本的节点编号
     * @return 新节点编号
     */
    static int update(int pos, int val, int l, int r, int pre) {
```

```
    int rt = ++cnt;
    left[rt] = left[pre];
    right[rt] = right[pre];
    sum[rt] = sum[pre] + val;

    if (l < r) {
        int mid = (l + r) / 2;
        if (pos <= mid) {
            left[rt] = update(pos, val, l, mid, left[rt]);
        } else {
            right[rt] = update(pos, val, mid + 1, r, right[rt]);
        }
    }
    return rt;
}

/**
 * 查询区间不同数字个数
 * @param l 查询区间左端点
 * @param r 查询区间右端点
 * @param u 前一个版本的根节点
 * @param v 当前版本的根节点
 * @return 不同数字个数
 */
static int query(int l, int r, int u, int v) {
    if (l == r) {
        return sum[v] - sum[u] > 0 ? 1 : 0;
    }
    int mid = (l + r) / 2;
    return query(l, mid, left[u], left[v]) + query(mid + 1, r, right[u], right[v]);
}

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

    int n = Integer.parseInt(reader.readLine());

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

```
        // 构建主席树
        root[0] = build(1, n);
        Arrays.fill(pos, 0);
        for (int i = 1; i <= n; i++) {
            if (pos[arr[i]] == 0) {
                // 第一次出现该数字
                root[i] = update(i, 1, 1, n, root[i - 1]);
            } else {
                // 该数字之前出现过，需要先减去之前的计数，再加上当前计数
                root[i] = update(pos[arr[i]], -1, 1, n, root[i - 1]);
                root[i] = update(i, 1, 1, n, root[i]);
            }
            pos[arr[i]] = i;
        }

        int q = Integer.parseInt(reader.readLine());
        for (int i = 0; i < q; i++) {
            line = reader.readLine().split(" ");
            int l = Integer.parseInt(line[0]);
            int r = Integer.parseInt(line[1]);
            writer.println(query(1, n, root[l - 1], root[r]));
        }

        writer.flush();
        writer.close();
        reader.close();
    }
}
```

==========

文件: SPOJ_DQUERY.py

==========

"""

SPOJ DQUERY - D-query

题目描述:

给定一个长度为N的序列，进行Q次查询，每次查询区间[l,r]中不同数字的个数。

解题思路:

使用可持久化线段树（主席树）解决静态区间不同元素个数问题。

1. 对于每个位置，记录该位置的数字上一次出现的位置

2. 从前向后建立主席树，第 i 棵线段树表示前 i 个位置的信息
3. 对于位置 i，如果该数字之前出现过位置 j，则在第 i 棵线段树中将位置 j 的计数减 1，位置 i 的计数加 1；否则只在位置 i 的计数加 1
4. 利用前缀和思想，通过第 r 棵和第 l-1 棵线段树的差得到区间[l,r]的信息
5. 查询区间不同数字个数即为查询区间内计数大于 0 的位置个数

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

示例:
输入:
5
1 1 2 1 3
3
1 5
2 4
3 5

输出:
3
2
3
"""

```
class Node:
    """线段树节点"""
    def __init__(self):
        self.sum: int = 0      # 区间内元素个数
        self.left = None  # 左子节点
        self.right = None # 右子节点


class PersistentSegmentTree:
    """可持久化线段树（主席树）"""
    def __init__(self, n):
        """
        初始化主席树
        :param n: 数组长度
        """
        self.n = n
        self.root = [None] * (n + 1)  # 每个版本线段树的根节点
        self.cnt = 0  # 节点计数器
```

```
def build(self, l, r):
    """
    构建空线段树
    :param l: 区间左端点
    :param r: 区间右端点
    :return: 根节点
    """
    node = Node()
    if l < r:
        mid = (l + r) // 2
        node.left = self.build(l, mid)
        node.right = self.build(mid + 1, r)
    return node

def update(self, pos, val, l, r, pre):
    """
    在线段树中更新一个位置的值
    :param pos: 要更新的位置
    :param val: 要增加的值
    :param l: 区间左端点
    :param r: 区间右端点
    :param pre: 前一个版本的节点
    :return: 新节点
    """
    node = Node()
    node.left = pre.left if pre else None
    node.right = pre.right if pre else None
    node.sum = (pre.sum if pre else 0) + val

    if l < r:
        mid = (l + r) // 2
        if pos <= mid:
            node.left = self.update(pos, val, l, mid, pre.left if pre else None)
        else:
            node.right = self.update(pos, val, mid + 1, r, pre.right if pre else None)
    return node

def query(self, l, r, u, v):
    """
    查询区间不同数字个数
    :param l: 查询区间左端点
    :param r: 查询区间右端点
```

```
        :param u: 前一个版本的根节点
        :param v: 当前版本的根节点
        :return: 不同数字个数
        """
        if l == r:
            return 1 if (v.sum if v else 0) - (u.sum if u else 0) > 0 else 0
        mid = (l + r) // 2
        left_count = self.query(l, mid, u.left if u else None, v.left if v else None)
        right_count = self.query(mid + 1, r, u.right if u else None, v.right if v else None)
        return left_count + right_count


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

    idx = 0
    n = int(data[idx])
    idx += 1

    # 读取原始数组
    arr = [0] * (n + 1)
    for i in range(1, n + 1):
        arr[i] = int(data[idx])
        idx += 1

    # 构建主席树
    pst = PersistentSegmentTree(n)
    pst.root[0] = pst.build(1, n)
    pos = {}  # pos[v] : 数字 v 上次出现的位置

    for i in range(1, n + 1):
        if arr[i] not in pos:
            # 第一次出现该数字
            pst.root[i] = pst.update(i, 1, 1, n, pst.root[i - 1])
        else:
            # 该数字之前出现过，需要先减去之前的计数，再加上当前计数
            pst.root[i] = pst.update(pos[arr[i]], -1, 1, n, pst.root[i - 1])
            pst.root[i] = pst.update(i, 1, 1, n, pst.root[i])
        pos[arr[i]] = i
```

```
    q = int(data[idx])
    idx += 1
    results = []
    for _ in range(q):
        l = int(data[idx])
        idx += 1
        r = int(data[idx])
        idx += 1
        results.append(str(pst.query(l, n, pst.root[l - 1], pst.root[r])))

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


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

==========

文件: SPOJ_DQUERY_Java.java

==========

```
package class158;

// Problem: SPOJ DQUERY - D-query
// Link: https://www.spoj.com/problems/DQUERY/
// Description: 给定一个包含 n 个数字的序列，每次查询区间[l,r]中不同数字的个数
// Solution: 使用可持久化线段树(主席树)解决区间不同元素个数问题
// Time Complexity: O(nlogn) for preprocessing, O(logn) for each query
// Space Complexity: O(nlogn)

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

public class SPOJ_DQUERY_Java {
    static final int MAXN = 30005;
    static int[] a = new int[MAXN];           // 原始数组
    static int[] prev = new int[MAXN];        // 每个元素上一次出现的位置
    static int[] last = new int[1000005];     // 每个值最后出现的位置

    // 主席树节点
    static class Node {
        int l, r, sum;
        Node(int l, int r, int sum) {
            this.l = l;
```

```
        this.r = r;
        this.sum = sum;
    }
}

static Node[] T = new Node[40 * MAXN];    // 主席树节点数组
static int[] root = new int[MAXN];        // 每个版本的根节点
static int cnt = 0;                       // 节点计数器

// 创建新节点
static int createNode(int l, int r, int sum) {
    T[++cnt] = new Node(l, r, sum);
    return cnt;
}

// 插入位置到主席树
static int insert(int pre, int l, int r, int pos, int val) {
    int now = createNode(0, 0, 0);
    T[now].sum = T[pre].sum + val;

    if (l == r) return now;

    int mid = (l + r) >> 1;
    if (pos <= mid) {
        T[now].l = insert(T[pre].l, l, mid, pos, val);
        T[now].r = T[pre].r;
    } else {
        T[now].l = T[pre].l;
        T[now].r = insert(T[pre].r, mid + 1, r, pos, val);
    }
    return now;
}

// 查询区间和
static int query(int u, int v, int l, int r, int L, int R) {
    if (L <= l && r <= R) return T[v].sum - T[u].sum;
    if (L > r || R < l) return 0;

    int mid = (l + r) >> 1;
    return query(T[u].l, T[v].l, l, mid, L, R) +
           query(T[u].r, T[v].r, mid + 1, r, L, R);
}
```

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

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

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

    // 预处理 prev 数组
    Arrays.fill(last, 0);
    for (int i = 1; i <= n; i++) {
        prev[i] = last[a[i]];
        last[a[i]] = i;
    }

    // 构建主席树
    root[0] = createNode(0, 0, 0);
    T[root[0]].l = T[root[0]].r = T[root[0]].sum = 0;

    for (int i = 1; i <= n; i++) {
        if (prev[i] == 0) {
            // 第一次出现，在位置 i 加 1
            root[i] = insert(root[i - 1], 1, n, i, 1);
        } else {
            // 不是第一次出现，先在 prev[i]位置减 1，再在 i 位置加 1
            int temp = insert(root[i - 1], 1, n, prev[i], -1);
            root[i] = insert(temp, 1, n, i, 1);
        }
    }

    int q = Integer.parseInt(br.readLine());
    for (int i = 0; i < q; i++) {
        line = br.readLine().split(" ");
        int l = Integer.parseInt(line[0]);
        int r = Integer.parseInt(line[1]);
        out.println(query(root[l - 1], root[r], 1, n, l, r));
    }

    out.close();
}
```

```
}
```

==========

文件: SPOJ_DQUERY_Python.py

==========

```
# Problem: SPOJ DQUERY - D-query
# Link: https://www.spoj.com/problems/DQUERY/
# Description: 给定一个包含 n 个数字的序列，每次查询区间[l,r]中不同数字的个数
# Solution: 使用可持久化线段树(主席树)解决区间不同元素个数问题
# Time Complexity: O(nlogn) for preprocessing, O(logn) for each query
# Space Complexity: O(nlogn)

import sys

class Node:
    def __init__(self):
        self.l = 0
        self.r = 0
        self.sum = 0

class PersistentSegmentTree:
    def __init__(self, maxn=30005):
        self.MAXN = maxn
        self.T = [Node() for _ in range(40 * maxn)]
        self.root = [0] * maxn
        self.cnt = 0

    def create_node(self, l=0, r=0, sum=0):
        self.cnt += 1
        self.T[self.cnt].l = l
        self.T[self.cnt].r = r
        self.T[self.cnt].sum = sum
        return self.cnt

    def insert(self, pre, l, r, pos, val):
        now = self.create_node()
        self.T[now].sum = self.T[pre].sum + val

        if l == r:
            return now

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

```
        if pos <= mid:
            self.T[now].l = self.insert(self.T[pre].l, l, mid, pos, val)
            self.T[now].r = self.T[pre].r
        else:
            self.T[now].l = self.T[pre].l
            self.T[now].r = self.insert(self.T[pre].r, mid + 1, r, pos, val)
        return now

    def query(self, u, v, l, r, L, R):
        if L <= l and r <= R:
            return self.T[v].sum - self.T[u].sum
        if L > r or R < l:
            return 0

        mid = (l + r) >> 1
        return self.query(self.T[u].l, self.T[v].l, l, mid, L, R) + \
               self.query(self.T[u].r, self.T[v].r, mid + 1, r, L, R)

def main():
    n = int(sys.stdin.readline())

    line = sys.stdin.readline().split()
    a = [0] * (n + 1)
    prev = [0] * (n + 1)
    last = [0] * 1000005

    for i in range(1, n + 1):
        a[i] = int(line[i - 1])

    # 预处理 prev 数组
    for i in range(1, n + 1):
        prev[i] = last[a[i]]
        last[a[i]] = i

    # 构建主席树
    pst = PersistentSegmentTree(n + 1)
    pst.root[0] = pst.create_node()

    for i in range(1, n + 1):
        if prev[i] == 0:
            # 第一次出现，在位置 i 加 1
            pst.root[i] = pst.insert(pst.root[i - 1], 1, n, i, 1)
        else:
```

```
            # 不是第一次出现，先在prev[i]位置减1，再在i位置加1
            temp = pst.insert(pst.root[i - 1], 1, n, prev[i], -1)
            pst.root[i] = pst.insert(temp, 1, n, i, 1)

    q = int(sys.stdin.readline())
    for _ in range(q):
        line = sys.stdin.readline().split()
        l, r = int(line[0]), int(line[1])
        print(pst.query(pst.root[l - 1], pst.root[r], 1, n, l, r))

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

==========

文件: SPOJ_MKTHNUM.cpp

==========

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

/**
 * SPOJ MKTHNUM - K-th Number
 *
 * 题目描述:
 * 给定一个长度为N的序列，进行M次查询，每次查询区间[l,r]中第K小的数。
 *
 * 解题思路:
 * 使用可持久化线段树（主席树）解决静态区间第K小问题。
 * 1. 对所有数值进行离散化处理
 * 2. 对每个位置建立权值线段树，第i棵线段树表示前i个数的信息
 * 3. 利用前缀和思想，通过第r棵和第l-1棵线段树的差得到区间[l,r]的信息
 * 4. 在线段树上二分查找第k小的数
 *
 * 时间复杂度: O(n log n + m log n)
 * 空间复杂度: O(n log n)
 *
 * 示例:
 * 输入:
 * 7 3
 * 1 5 2 6 3 7 4
 * 2 5 3
 * 4 7 1
```

```
 * 1 7 3
 *
 * 输出:
 * 5
 * 6
 * 3
 */

const int MAXN = 100010;

// 原始数组
int arr[MAXN];
// 离散化后的数组
int sorted[MAXN];
// 每个版本线段树的根节点
int root[MAXN];

// 线段树节点信息
int left[MAXN * 20];
int right[MAXN * 20];
int sum[MAXN * 20];

// 线段树节点计数器
int cnt = 0;

/**
 * 构建空线段树
 * @param l 区间左端点
 * @param r 区间右端点
 * @return 根节点编号
 */
int build(int l, int r) {
    int rt = ++cnt;
    sum[rt] = 0;
    if (l < r) {
        int mid = (l + r) / 2;
        left[rt] = build(l, mid);
        right[rt] = build(mid + 1, r);
    }
    return rt;
}

/**
```

```
 * 在线段树中插入一个值
 * @param pos 要插入的值（离散化后的坐标）
 * @param l 区间左端点
 * @param r 区间右端点
 * @param pre 前一个版本的节点编号
 * @return 新节点编号
 */
int insert(int pos, int l, int r, int pre) {
    int rt = ++cnt;
    left[rt] = left[pre];
    right[rt] = right[pre];
    sum[rt] = sum[pre] + 1;

    if (l < r) {
        int mid = (l + r) / 2;
        if (pos <= mid) {
            left[rt] = insert(pos, l, mid, left[rt]);
        } else {
            right[rt] = insert(pos, mid + 1, r, right[rt]);
        }
    }
    return rt;
}

/**
 * 查询区间第 k 小的数
 * @param k 第 k 小
 * @param l 区间左端点
 * @param r 区间右端点
 * @param u 前一个版本的根节点
 * @param v 当前版本的根节点
 * @return 第 k 小的数在离散化数组中的位置
 */
int query(int k, int l, int r, int u, int v) {
    if (l >= r) return l;
    int mid = (l + r) / 2;
    // 计算左子树中数的个数
    int x = sum[left[v]] - sum[left[u]];
    if (x >= k) {
        // 第 k 小在左子树中
        return query(k, l, mid, left[u], left[v]);
    } else {
        // 第 k 小在右子树中
```

```
        return query(k - x, mid + 1, r, right[u], right[v]);
    }
}

/**
 * 离散化查找数值对应的排名
 * @param val 要查找的值
 * @param n 数组长度
 * @return 值的排名
 */
int getId(int val, int n) {
    return lower_bound(sorted + 1, sorted + n + 1, val) - sorted;
}

int main() {
    int n, m;
    scanf("%d%d", &n, &m);

    // 读取原始数组
    for (int i = 1; i <= n; i++) {
        scanf("%d", &arr[i]);
        sorted[i] = arr[i];
    }

    // 离散化处理
    sort(sorted + 1, sorted + n + 1);
    int size = 1;
    for (int i = 2; i <= n; i++) {
        if (sorted[i] != sorted[size]) {
            sorted[++size] = sorted[i];
        }
    }

    // 构建主席树
    root[0] = build(1, size);
    for (int i = 1; i <= n; i++) {
        int pos = getId(arr[i], size);
        root[i] = insert(pos, 1, size, root[i - 1]);
    }

    // 处理查询
    for (int i = 0; i < m; i++) {
        int l, r, k;
```

```
        scanf("%d%d%d", &l, &r, &k);
        int pos = query(k, 1, size, root[l - 1], root[r]);
        printf("%d\n", sorted[pos]);
    }

    return 0;
}
```

==========

文件: SPOJ_MKTHNUM.java

==========

```
package class158;

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

/**
 * SPOJ MKTHNUM - K-th Number
 *
 * 题目来源：SPOJ
 * 题目链接：https://www.spoj.com/problems/MKTHNUM/
 *
 * 题目描述:
 * 给定一个长度为N的序列，进行M次查询，每次查询区间[l,r]中第K小的数。
 *
 * 解题思路:
 * 使用可持久化线段树（主席树）解决静态区间第K小问题。
 * 1. 对所有数值进行离散化处理
 * 2. 对每个位置建立权值线段树，第i棵线段树表示前i个数的信息
 * 3. 利用前缀和思想，通过第r棵和第l-1棵线段树的差得到区间[l,r]的信息
 * 4. 在线段树上二分查找第k小的数
 *
 * 时间复杂度: O(n log n + m log n)
 * 空间复杂度: O(n log n)
 *
 * 示例:
 * 输入:
 * 7 3
 * 1 5 2 6 3 7 4
 * 2 5 3
 * 4 7 1
 * 1 7 3
```

```
 *
 * 输出:
 * 5
 * 6
 * 3
 */
public class SPOJ_MKTHNUM {
    static final int MAXN = 100010;

    // 原始数组
    static int[] arr = new int[MAXN];
    // 离散化后的数组
    static int[] sorted = new int[MAXN];
    // 每个版本线段树的根节点
    static int[] root = new int[MAXN];

    // 线段树节点信息
    static int[] left = new int[MAXN * 20];
    static int[] right = new int[MAXN * 20];
    static int[] sum = new int[MAXN * 20];

    // 线段树节点计数器
    static int cnt = 0;

    /**
     * 构建空线段树
     * @param l 区间左端点
     * @param r 区间右端点
     * @return 根节点编号
     */
    static int build(int l, int r) {
        int rt = ++cnt;
        sum[rt] = 0;
        if (l < r) {
            int mid = (l + r) / 2;
            left[rt] = build(l, mid);
            right[rt] = build(mid + 1, r);
        }
        return rt;
    }

    /**
     * 在线段树中插入一个值
```

```
     * @param pos 要插入的值（离散化后的坐标）
     * @param l 区间左端点
     * @param r 区间右端点
     * @param pre 前一个版本的节点编号
     * @return 新节点编号
     */
    static int insert(int pos, int l, int r, int pre) {
        int rt = ++cnt;
        left[rt] = left[pre];
        right[rt] = right[pre];
        sum[rt] = sum[pre] + 1;

        if (l < r) {
            int mid = (l + r) / 2;
            if (pos <= mid) {
                left[rt] = insert(pos, l, mid, left[rt]);
            } else {
                right[rt] = insert(pos, mid + 1, r, right[rt]);
            }
        }
        return rt;
    }

    /**
     * 查询区间第 k 小的数
     * @param k 第 k 小
     * @param l 区间左端点
     * @param r 区间右端点
     * @param u 前一个版本的根节点
     * @param v 当前版本的根节点
     * @return 第 k 小的数在离散化数组中的位置
     */
    static int query(int k, int l, int r, int u, int v) {
        if (l >= r) return l;
        int mid = (l + r) / 2;
        // 计算左子树中数的个数
        int x = sum[left[v]] - sum[left[u]];
        if (x >= k) {
            // 第 k 小在左子树中
            return query(k, l, mid, left[u], left[v]);
        } else {
            // 第 k 小在右子树中
            return query(k - x, mid + 1, r, right[u], right[v]);
```

```
    }
}

/**
 * 离散化查找数值对应的排名
 * @param val 要查找的值
 * @param n 数组长度
 * @return 值的排名
 */
static int getId(int val, int n) {
    return Arrays.binarySearch(sorted, 1, n + 1, val) + 1;
}

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

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

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

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

    // 构建主席树
    root[0] = build(1, size);
    for (int i = 1; i <= n; i++) {
        int pos = getId(arr[i], size);
        root[i] = insert(pos, 1, size, root[i - 1]);
    }
```

```
        // 处理查询
        for (int i = 0; i < m; i++) {
            line = reader.readLine().split(" ");
            int l = Integer.parseInt(line[0]);
            int r = Integer.parseInt(line[1]);
            int k = Integer.parseInt(line[2]);
            int pos = query(k, 1, size, root[l - 1], root[r]);
            writer.println(sorted[pos]);
        }

        writer.flush();
        writer.close();
        reader.close();
    }
}
```

==========

文件: SPOJ_MKTHNUM.py

==========

```
# -*- coding: utf-8 -*-
"""
SPOJ MKTHNUM - K-th Number

题目描述:
给定一个长度为N的序列，进行M次查询，每次查询区间[l,r]中第K小的数。

解题思路:
使用可持久化线段树（主席树）解决静态区间第K小问题。
1. 对所有数值进行离散化处理
2. 对每个位置建立权值线段树，第i棵线段树表示前i个数的信息
3. 利用前缀和思想，通过第r棵和第l-1棵线段树的差得到区间[l,r]的信息
4. 在线段树上二分查找第k小的数

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

示例:
输入:
7 3
1 5 2 6 3 7 4
2 5 3
```

```
4 7 1
1 7 3

输出:
5
6
3
"""

import bisect

class PersistentSegmentTree:
    """可持久化线段树（主席树）实现"""

    def __init__(self, n):
        """
        初始化主席树
        :param n: 离散化后的值域大小
        """
        self.n = n
        # 每个版本线段树的根节点
        self.root = [0] * (n + 1)
        # 线段树节点信息
        self.left = [0] * (n * 20)
        self.right = [0] * (n * 20)
        self.sum = [0] * (n * 20)
        # 线段树节点计数器
        self.cnt = 0

    def build(self, l, r):
        """
        构建空线段树
        :param l: 区间左端点
        :param r: 区间右端点
        :return: 根节点编号
        """
        rt = self.cnt + 1
        self.cnt += 1
        self.sum[rt] = 0
        if l < r:
            mid = (l + r) // 2
            self.left[rt] = self.build(l, mid)
            self.right[rt] = self.build(mid + 1, r)
```

```
        return rt

    def insert(self, pos, l, r, pre):
        """
        在线段树中插入一个值
        :param pos: 要插入的值（离散化后的坐标）
        :param l: 区间左端点
        :param r: 区间右端点
        :param pre: 前一个版本的节点编号
        :return: 新节点编号
        """
        rt = self.cnt + 1
        self.cnt += 1
        self.left[rt] = self.left[pre]
        self.right[rt] = self.right[pre]
        self.sum[rt] = self.sum[pre] + 1

        if l < r:
            mid = (l + r) // 2
            if pos <= mid:
                self.left[rt] = self.insert(pos, l, mid, self.left[rt])
            else:
                self.right[rt] = self.insert(pos, mid + 1, r, self.right[rt])
        return rt

    def query(self, k, l, r, u, v):
        """
        查询区间第 k 小的数
        :param k: 第 k 小
        :param l: 区间左端点
        :param r: 区间右端点
        :param u: 前一个版本的根节点
        :param v: 当前版本的根节点
        :return: 第 k 小的数在离散化数组中的位置
        """
        if l >= r:
            return l
        mid = (l + r) // 2
        # 计算左子树中数的个数
        x = self.sum[self.left[v]] - self.sum[self.left[u]]
        if x >= k:
            # 第 k 小在左子树中
            return self.query(k, l, mid, self.left[u], self.left[v])
```

```
        else:
            # 第k小在右子树中
            return self.query(k - x, mid + 1, r, self.right[u], self.right[v])


def main():
    # 读取输入
    n, m = map(int, input().split())

    # 原始数组和离散化数组
    arr = [0] * (n + 1)
    sorted_arr = [0] * (n + 1)

    # 读取原始数组
    values = list(map(int, input().split()))
    for i in range(1, n + 1):
        arr[i] = values[i - 1]
        sorted_arr[i] = arr[i]

    # 离散化处理
    sorted_arr = sorted_arr[1:n + 1]
    sorted_arr.sort()

    # 去重
    unique_sorted = [sorted_arr[0]]
    for i in range(1, n):
        if sorted_arr[i] != sorted_arr[i - 1]:
            unique_sorted.append(sorted_arr[i])

    size = len(unique_sorted)

    # 构建主席树
    pst = PersistentSegmentTree(size)
    pst.root[0] = pst.build(1, size)

    # 为每个位置建立线段树
    for i in range(1, n + 1):
        # 查找arr[i]在离散化数组中的位置
        pos = bisect.bisect_left(unique_sorted, arr[i]) + 1
        pst.root[i] = pst.insert(pos, 1, size, pst.root[i - 1])

    # 处理查询
    for _ in range(m):
```

```
        l, r, k = map(int, input().split())
        pos = pst.query(k, 1, size, pst.root[l - 1], pst.root[r])
        print(unique_sorted[pos - 1])


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

==========

文件: SPOJ_MKTHNUM_CPP.cpp

==========

```
// Problem: SPOJ MKTHNUM - K-th Number
// Link: https://www.spoj.com/problems/MKTHNUM/
// Description: 给定一个包含 n 个数字的序列，每次查询区间[l,r]中第 k 小的数
// Solution: 使用可持久化线段树(主席树)解决静态区间第 k 小问题
// Time Complexity: O(nlogn) for preprocessing, O(logn) for each query
// Space Complexity: O(nlogn)

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MAXN 100005

int a[MAXN], b[MAXN], n, m;
int root[MAXN], cnt = 0;

// 主席树节点结构
struct Node {
    int l, r, sum;
} T[20 * MAXN];

// 比较函数，用于排序
int compare(const void *a, const void *b) {
    return (*(int*)a - *(int*)b);
}

// 二分查找
int lower_bound(int* arr, int size, int target) {
    int left = 0, right = size;
    while (left < right) {
        int mid = (left + right) / 2;
```

```
        if (arr[mid] < target) {
            left = mid + 1;
        } else {
            right = mid;
        }
    }
    return left;
}

// 更新节点
int insert(int pre, int l, int r, int val) {
    int now = ++cnt;
    T[now] = T[pre];
    T[now].sum++;

    if (l == r) return now;

    int mid = (l + r) >> 1;
    if (val <= mid) {
        T[now].l = insert(T[pre].l, l, mid, val);
    } else {
        T[now].r = insert(T[pre].r, mid + 1, r, val);
    }
    return now;
}

// 查询区间第 k 小
int query(int u, int v, int l, int r, int k) {
    if (l == r) return l;

    int mid = (l + r) >> 1;
    int x = T[T[v].l].sum - T[T[u].l].sum;

    if (k <= x) {
        return query(T[u].l, T[v].l, l, mid, k);
    } else {
        return query(T[u].r, T[v].r, mid + 1, r, k - x);
    }
}

// 离散化
int getId(int x) {
    return lower_bound(b + 1, n, x) + 1;
```

```
}

// 去重函数
int unique(int* arr, int size) {
    if (size == 0) return 0;
    int i, j = 1;
    for (i = 1; i < size; i++) {
        if (arr[i] != arr[i-1]) {
            arr[j++] = arr[i];
        }
    }
    return j;
}

int main() {
    scanf("%d%d", &n, &m);

    for (int i = 1; i <= n; i++) {
        scanf("%d", &a[i]);
        b[i] = a[i];
    }

    // 离散化
    qsort(b + 1, n, sizeof(int), compare);
    int sz = unique(b + 1, n + 1);

    // 构建主席树
    for (int i = 1; i <= n; i++) {
        root[i] = insert(root[i - 1], 1, sz, getId(a[i]));
    }

    // 处理查询
    for (int i = 0; i < m; i++) {
        int l, r, k;
        scanf("%d%d%d", &l, &r, &k);
        int id = query(root[l - 1], root[r], 1, sz, k);
        printf("%d\n", b[id]);
    }

    return 0;
}
```

==========

文件: SPOJ_MKTHNUM_Java.java

```
==========================================
package class158;

// Problem: SPOJ MKTHNUM - K-th Number
// Link: https://www.spoj.com/problems/MKTHNUM/
// Description: 给定一个包含 n 个数字的序列，每次查询区间[l,r]中第 k 小的数
// Solution: 使用可持久化线段树(主席树)解决静态区间第 k 小问题
// Time Complexity: O(nlogn) for preprocessing, O(logn) for each query
// Space Complexity: O(nlogn)

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

public class SPOJ_MKTHNUM_Java {
    static final int MAXN = 100005;
    static int[] a = new int[MAXN];           // 原始数组
    static int[] b = new int[MAXN];           // 离散化数组
    static int n, m;

    // 主席树节点
    static class Node {
        int l, r, sum;
        Node(int l, int r, int sum) {
            this.l = l;
            this.r = r;
            this.sum = sum;
        }
    }

    static Node[] T = new Node[20 * MAXN];    // 主席树节点数组
    static int[] root = new int[MAXN];        // 每个版本的根节点
    static int cnt = 0;                       // 节点计数器

    // 创建新节点
    static int createNode(int l, int r, int sum) {
        T[++cnt] = new Node(l, r, sum);
        return cnt;
    }

    // 插入值到主席树
    static int insert(int pre, int l, int r, int val) {
```

```
    int now = createNode(0, 0, 0);
    T[now].sum = T[pre].sum + 1;

    if (l == r) return now;

    int mid = (l + r) >> 1;
    if (val <= mid) {
        T[now].l = insert(T[pre].l, l, mid, val);
        T[now].r = T[pre].r;
    } else {
        T[now].l = T[pre].l;
        T[now].r = insert(T[pre].r, mid + 1, r, val);
    }
    return now;
}

// 查询区间第 k 小
static int query(int u, int v, int l, int r, int k) {
    if (l == r) return l;

    int mid = (l + r) >> 1;
    int x = T[T[v].l].sum - T[T[u].l].sum;

    if (k <= x) {
        return query(T[u].l, T[v].l, l, mid, k);
    } else {
        return query(T[u].r, T[v].r, mid + 1, r, k - x);
    }
}

// 离散化
static int getId(int x) {
    return Arrays.binarySearch(b, 1, n + 1, x) + 1;
}

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

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

```
        line = br.readLine().split(" ");
        for (int i = 1; i <= n; i++) {
            a[i] = Integer.parseInt(line[i - 1]);
            b[i] = a[i];
        }

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

        // 构建主席树
        root[0] = createNode(0, 0, 0);
        T[root[0]].l = T[root[0]].r = T[root[0]].sum = 0;

        for (int i = 1; i <= n; i++) {
            root[i] = insert(root[i - 1], 1, sz, getId(a[i]));
        }

        // 处理查询
        for (int i = 0; i < m; i++) {
            line = br.readLine().split(" ");
            int l = Integer.parseInt(line[0]);
            int r = Integer.parseInt(line[1]);
            int k = Integer.parseInt(line[2]);

            int id = query(root[l - 1], root[r], 1, sz, k);
            out.println(b[id]);
        }

        out.close();
    }
}
```

==========

文件: SPOJ_MKTHNUM_Python.py

==========

```
# Problem: SPOJ MKTHNUM - K-th Number
```

```
# Link: https://www.spoj.com/problems/MKTHNUM/
# Description: 给定一个包含 n 个数字的序列，每次查询区间[l,r]中第 k 小的数
# Solution: 使用可持久化线段树(主席树)解决静态区间第 k 小问题
# Time Complexity: O(nlogn) for preprocessing, O(logn) for each query
# Space Complexity: O(nlogn)

import sys
from bisect import bisect_left

class Node:
    def __init__(self):
        self.l = 0
        self.r = 0
        self.sum = 0

class PersistentSegmentTree:
    def __init__(self, maxn=100005):
        self.MAXN = maxn
        self.T = [Node() for _ in range(20 * maxn)]
        self.root = [0] * maxn
        self.cnt = 0

    def create_node(self, l=0, r=0, sum=0):
        self.cnt += 1
        self.T[self.cnt].l = l
        self.T[self.cnt].r = r
        self.T[self.cnt].sum = sum
        return self.cnt

    def insert(self, pre, l, r, val):
        now = self.create_node()
        self.T[now].sum = self.T[pre].sum + 1

        if l == r:
            return now

        mid = (l + r) >> 1
        if val <= mid:
            self.T[now].l = self.insert(self.T[pre].l, l, mid, val)
            self.T[now].r = self.T[pre].r
        else:
            self.T[now].l = self.T[pre].l
            self.T[now].r = self.insert(self.T[pre].r, mid + 1, r, val)
```

```
        return now

    def query(self, u, v, l, r, k):
        if l == r:
            return l

        mid = (l + r) >> 1
        x = self.T[self.T[v].l].sum - self.T[self.T[u].l].sum

        if k <= x:
            return self.query(self.T[u].l, self.T[v].l, l, mid, k)
        else:
            return self.query(self.T[u].r, self.T[v].r, mid + 1, r, k - x)

def main():
    # 读取输入
    line = sys.stdin.readline().split()
    n, m = int(line[0]), int(line[1])

    line = sys.stdin.readline().split()
    a = [0] * (n + 1)
    b = [0] * (n + 1)

    for i in range(1, n + 1):
        a[i] = int(line[i - 1])
        b[i] = a[i]

    # 离散化
    b = b[1:]  # 去掉索引 0
    b.sort()
    # 去重
    unique_b = []
    for x in b:
        if not unique_b or unique_b[-1] != x:
            unique_b.append(x)
    b = [0] + unique_b  # 加回索引 0
    sz = len(b) - 1

    # 获取值的离散化 ID
    def get_id(x):
        return bisect_left(b, x, 1, len(b))

    # 构建主席树
```

```
    pst = PersistentSegmentTree(n + 1)
    pst.root[0] = pst.create_node()

    for i in range(1, n + 1):
        pst.root[i] = pst.insert(pst.root[i - 1], 1, sz, get_id(a[i]))

    # 处理查询
    for _ in range(m):
        line = sys.stdin.readline().split()
        l, r, k = int(line[0]), int(line[1]), int(line[2])
        id = pst.query(pst.root[l - 1], pst.root[r], 1, sz, k)
        print(b[id])

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

==========