==========
文件夹: class084_SegmentTree
==========

[Markdown 文件]
==========
文件: ADDITIONAL_SEGMENT_TREE_PROBLEMS.md
==========
更多线段树题目 - Additional Segment Tree Problems

LeetCode 题目

1. LeetCode 307. Range Sum Query - Mutable (区间求和 - 可变)
- **类型**: 单点更新 + 区间求和
- **难度**: Medium
- **测试链接**: https://leetcode.com/problems/range-sum-query-mutable/
- **核心思想**: 经典线段树应用
- **已实现**: Code18_RangeSumQueryMutable.java/.py/.cpp

2. LeetCode 315. Count of Smaller Numbers After Self (计算右侧小于当前元素的个数)
- **类型**: 离散化 + 单点更新 + 区间求和
- **难度**: Hard
- **测试链接**: https://leetcode.com/problems/count-of-smaller-numbers-after-self/
- **核心思想**: 使用线段树维护值域信息

3. LeetCode 493. Reverse Pairs (翻转对)
- **类型**: 离散化 + 单点更新 + 区间求和
- **难度**: Hard
- **测试链接**: https://leetcode.com/problems/reverse-pairs/
- **核心思想**: 计算满足条件的逆序对

4. LeetCode 327. Count of Range Sum (区间和的个数)
- **类型**: 离散化 + 单点更新 + 区间求和
- **难度**: Hard
- **测试链接**: https://leetcode.com/problems/count-of-range-sum/
- **核心思想**: 前缀和 + 线段树

5. LeetCode 1157. Online Majority Element In Subarray (子数组中占绝大多数的元素)
- **类型**: 区间查询 + 二分查找
- **难度**: Hard
- **测试链接**: https://leetcode.com/problems/online-majority-element-in-subarray/
- **核心思想**: 线段树维护区间众数信息

6. LeetCode 1526. Minimum Number of Increments on Subarrays to Form a Target Array (形成目标数组的子数组最少增加次数)
- **类型**: 差分 + 贪心 + 线段树
- **难度**: Hard
- **测试链接**: https://leetcode.com/problems/minimum-number-of-increments-on-subarrays-to-form-a-target-array/
- **核心思想**: 差分数组 + 线段树维护

7. LeetCode 715. Range Module (Range 模块)
- **类型**: 区间合并 + 线段树
- **难度**: Hard
- **测试链接**: https://leetcode.com/problems/range-module/
- **核心思想**: 维护区间覆盖状态

8. LeetCode 732. My Calendar III (我的日程安排表 III)
- **类型**: 区间最大重叠次数 + 线段树
- **难度**: Hard
- **测试链接**: https://leetcode.com/problems/my-calendar-iii/
- **核心思想**: 维护区间最大值

9. LeetCode 699. Falling Squares (掉落的方块)
- **类型**: 区间最值查询 + 离散化
- **难度**: Hard
- **测试链接**: https://leetcode.com/problems/falling-squares/
- **核心思想**: 坐标离散化 + 线段树维护区间最大值

10. LeetCode 850. Rectangle Area II (矩形面积 II)
- **类型**: 扫描线 + 线段树
- **难度**: Hard
- **测试链接**: https://leetcode.com/problems/rectangle-area-ii/
- **核心思想**: 扫描线算法 + 线段树维护区间长度

11. LeetCode 1099. Two Sum Less Than K (两数之和小于 K)
- **类型**: 排序 + 双指针 + 线段树
- **难度**: Easy
- **测试链接**: https://leetcode.com/problems/two-sum-less-than-k/
- **核心思想**: 可以用线段树优化查询

12. LeetCode 1155. Number of Dice Rolls With Target Sum (骰子的不同方法数)
- **类型**: 动态规划 + 线段树优化
- **难度**: Medium
- **测试链接**: https://leetcode.com/problems/number-of-dice-rolls-with-target-sum/
- **核心思想**: 线段树优化区间查询

13. LeetCode 1483. Kth Ancestor of a Tree Node (树节点的第 K 个祖先)
- **类型**: 倍增 + 线段树
- **难度**: Hard
- **测试链接**: https://leetcode.com/problems/kth-ancestor-of-a-tree-node/
- **核心思想**: 线段树维护祖先信息

14. LeetCode 1569. Number of Ways to Reorder Array to Get Same BST (重新排序得到相同二叉搜索树的方案数)
- **类型**: 组合数学 + 线段树
- **难度**: Hard
- **测试链接**: https://leetcode.com/problems/number-of-ways-to-reorder-array-to-get-same-bst/
- **核心思想**: 线段树维护区间信息

15. LeetCode 1649. Create Sorted Array through Instructions (通过指令创建有序数组)
- **类型**: 离散化 + 线段树
- **难度**: Hard
- **测试链接**: https://leetcode.com/problems/create-sorted-array-through-instructions/
- **核心思想**: 线段树维护计数信息

16. LeetCode 1744. Can You Eat Your Favorite Candy on Your Favorite Day? (你能在你最喜欢的那天吃到你最喜欢的糖果吗？)
- **类型**: 前缀和 + 线段树
- **难度**: Medium
- **测试链接**: https://leetcode.com/problems/can-you-eat-your-favorite-candy-on-your-favorite-day/
- **核心思想**: 线段树维护前缀和

17. LeetCode 1844. Replace All Digits with Characters (将所有数字用字符替换)
- **类型**: 字符串处理 + 线段树
- **难度**: Easy
- **测试链接**: https://leetcode.com/problems/replace-all-digits-with-characters/
- **核心思想**: 可以用线段树优化多模式匹配

18. LeetCode 1906. Minimum Absolute Difference Queries (查询的最小绝对差)
- **类型**: 前缀统计 + 线段树
- **难度**: Medium
- **测试链接**: https://leetcode.com/problems/minimum-absolute-difference-queries/
- **核心思想**: 线段树维护区间计数信息

19. LeetCode 1944. Number of Visible People in a Queue (队列中可以看到的人数)
- **类型**: 单调栈 + 线段树
- **难度**: Hard

- **测试链接**: https://leetcode.com/problems/number-of-visible-people-in-a-queue/
- **核心思想**: 线段树维护可见性信息

20. LeetCode 2013. Detect Squares (检测正方形)
- **类型**: 哈希表 + 线段树
- **难度**: Medium
- **测试链接**: https://leetcode.com/problems/detect-squares/
- **核心思想**: 线段树维护坐标信息

Codeforces 题目

1. Codeforces 339D. Xenia and Bit Operations
- **类型**: 位运算 + 线段树
- **难度**: Medium
- **测试链接**: https://codeforces.com/contest/339/problem/D
- **核心思想**: 线段树维护位运算结果

2. Codeforces 459D. Pashmak and Parmida's problem
- **类型**: 离散化 + 线段树
- **难度**: Hard
- **测试链接**: https://codeforces.com/contest/459/problem/D
- **核心思想**: 前缀统计 + 线段树查询

3. Codeforces 52C. Circular RMQ
- **类型**: 循环数组 + 线段树
- **难度**: Hard
- **测试链接**: https://codeforces.com/contest/52/problem/C
- **核心思想**: 处理循环数组的区间操作

4. Codeforces 369E. Valera and Queries
- **类型**: 离线处理 + 线段树
- **难度**: Hard
- **测试链接**: https://codeforces.com/contest/369/problem/E
- **核心思想**: 离线处理 + 线段树维护

5. Codeforces 620E. New Year Tree
- **类型**: 树链剖分 + 线段树
- **难度**: Hard
- **测试链接**: https://codeforces.com/contest/620/problem/E
- **核心思想**: 线段树维护子树信息

6. Codeforces 1110F. Nearest Leaf
- **类型**: 树链剖分 + 线段树

- **难度**: Hard
- **测试链接**: https://codeforces.com/contest/1110/problem/F
- **核心思想**: 线段树维护路径信息

7. Codeforces 1093E. Intersection of Permutations
- **类型**: 线段树套线段树
- **难度**: Hard
- **测试链接**: https://codeforces.com/contest/1093/problem/E
- **核心思想**: 二维线段树维护排列信息

8. Codeforces 1194F. Crossword Expert
- **类型**: 二分 + 线段树
- **难度**: Hard
- **测试链接**: https://codeforces.com/contest/1194/problem/F
- **核心思想**: 线段树维护时间信息

SPOJ 题目

1. SPOJ GSS1 - Can you answer these queries I
- **类型**: 区间最大子段和 + 线段树
- **难度**: Hard
- **测试链接**: https://www.spoj.com/problems/GSS1/
- **核心思想**: 线段树维护区间最大子段和信息
- **已实现**: Code19_GSS1.java/.py/.cpp

2. SPOJ GSS3 - Can you answer these queries III
- **类型**: 区间最大子段和 + 单点更新
- **难度**: Hard
- **测试链接**: https://www.spoj.com/problems/GSS3/
- **核心思想**: 支持单点更新的区间最大子段和

3. SPOJ GSS4 - Can you answer these queries IV
- **类型**: 区间开方 + 线段树
- **难度**: Hard
- **测试链接**: https://www.spoj.com/problems/GSS4/
- **核心思想**: 利用开方操作收敛性进行优化

4. SPOJ HORRIBLE - Horrible Queries
- **类型**: 区间更新 + 区间求和 + 懒惰传播
- **难度**: Hard
- **测试链接**: https://www.spoj.com/problems/HORRIBLE/
- **核心思想**: 带懒惰传播的线段树
- **已实现**: Code11_HorribleQueries.java/.py/.cpp

5. SPOJ BRCKTS - Brackets
- **类型**: 括号匹配 + 线段树
- **难度**: Medium
- **测试链接**: https://www.spoj.com/problems/BRCKTS/
- **核心思想**: 线段树维护括号匹配信息

6. SPOJ FREQUENT - Frequent values
- **类型**: 区间众数 + 线段树
- **难度**: Hard
- **测试链接**: https://www.spoj.com/problems/FREQUENT/
- **核心思想**: 线段树维护区间众数信息

7. SPOJ MKTHNUM - K-th number
- **类型**: 可持久化线段树（主席树）
- **难度**: Hard
- **测试链接**: https://www.spoj.com/problems/MKTHNUM/
- **核心思想**: 主席树求区间第 K 大

8. SPOJ COT - Count on a tree
- **类型**: 树链剖分 + 可持久化线段树
- **难度**: Hard
- **测试链接**: https://www.spoj.com/problems/COT/
- **核心思想**: 树上路径第 K 大

9. SPOJ RMQSQ - Range Minimum Query
- **类型**: 区间最小值查询
- **难度**: Easy
- **测试链接**: https://www.spoj.com/problems/RMQSQ/
- **核心思想**: 线段树维护区间最小值

AtCoder 题目

1. AtCoder ABC351 Practice J - Segment Tree
- **类型**: 基础线段树操作
- **难度**: Easy
- **测试链接**: https://atcoder.jp/contests/practice2/tasks/practice2_j
- **核心思想**: 线段树基础应用

2. AtCoder ARC159D - Yet Another ABC String
- **类型**: 线段树优化 DP
- **难度**: Hard
- **测试链接**: https://atcoder.jp/contests/arc159/tasks/arc159_d

- **核心思想**: 线段树优化动态规划

3. AtCoder ABC294F - Sugar Water 2
- **类型**: 二分答案 + 线段树
- **难度**: Medium
- **测试链接**: https://atcoder.jp/contests/abc294/tasks/abc294_f
- **核心思想**: 线段树维护计数信息

4. AtCoder ABC253F - Operations on a Matrix
- **类型**: 二维线段树
- **难度**: Hard
- **测试链接**: https://atcoder.jp/contests/abc253/tasks/abc253_f
- **核心思想**: 二维线段树维护矩阵信息

HDU 题目

1. HDU 1166. 敌兵布阵
- **类型**: 单点更新 + 区间求和
- **难度**: Medium
- **测试链接**: https://acm.hdu.edu.cn/showproblem.php?pid=1166
- **核心思想**: 经典线段树应用
- **已实现**: Code20_HDU1166.java/.py/.cpp

2. HDU 1754. I Hate It
- **类型**: 单点更新 + 区间最值
- **难度**: Medium
- **测试链接**: https://acm.hdu.edu.cn/showproblem.php?pid=1754
- **核心思想**: 线段树维护区间最大值

3. HDU 1698. Just a Hook
- **类型**: 区间更新 + 区间求和 + 懒惰传播
- **难度**: Medium
- **测试链接**: https://acm.hdu.edu.cn/showproblem.php?pid=1698
- **核心思想**: 带懒惰传播的线段树

4. HDU 2795. Billboard
- **类型**: 线段树优化二分
- **难度**: Medium
- **测试链接**: https://acm.hdu.edu.cn/showproblem.php?pid=2795
- **核心思想**: 线段树维护最大剩余空间

5. HDU 3308. LCIS
- **类型**: 线段树维护最长连续递增子序列

- **难度**: Hard
- **测试链接**: https://acm.hdu.edu.cn/showproblem.php?pid=3308
- **核心思想**: 线段树维护区间 LCIS 信息

6. HDU 4578. Transformation
- **类型**: 多种区间更新 + 线段树
- **难度**: Hard
- **测试链接**: https://acm.hdu.edu.cn/showproblem.php?pid=4578
- **核心思想**: 处理多种懒惰标记的优先级

7. HDU 5690. All X
- **类型**: 数学计算 + 线段树
- **难度**: Medium
- **测试链接**: https://acm.hdu.edu.cn/showproblem.php?pid=5690
- **核心思想**: 线段树维护数学表达式结果

POJ 题目

1. POJ 3468. A Simple Problem with Integers
- **类型**: 区间更新 + 区间求和 + 懒惰传播
- **难度**: Hard
- **测试链接**: http://poj.org/problem?id=3468
- **核心思想**: 经典的带懒惰传播线段树
- **已实现**: Code21_POJ3468.java/.py/.cpp

2. POJ 2528. Mayor's posters
- **类型**: 离散化 + 区间覆盖
- **难度**: Hard
- **测试链接**: http://poj.org/problem?id=2528
- **核心思想**: 坐标离散化 + 线段树维护区间覆盖

3. POJ 3264. Balanced Lineup
- **类型**: 区间最大最小值查询
- **难度**: Medium
- **测试链接**: http://poj.org/problem?id=3264
- **核心思想**: 线段树维护区间极值

4. POJ 2104. K-th Number
- **类型**: 可持久化线段树（主席树）
- **难度**: Hard
- **测试链接**: http://poj.org/problem?id=2104
- **核心思想**: 主席树求区间第 K 大

5. POJ 2828. Buy Tickets
- **类型**: 线段树逆序处理
- **难度**: Medium
- **测试链接**: http://poj.org/problem?id=2828
- **核心思想**: 线段树维护剩余位置

6. POJ 3250. Bad Hair Day
- **类型**: 单调栈 + 线段树
- **难度**: Medium
- **测试链接**: http://poj.org/problem?id=3250
- **核心思想**: 线段树维护可见性信息

洛谷(Luogu)题目

1. P3372 【模板】线段树 1
- **类型**: 区间更新 + 区间求和 + 懒惰传播
- **难度**: 模板
- **测试链接**: https://www.luogu.com.cn/problem/P3372
- **核心思想**: 线段树模板题
- **已实现**: Code22_LuoguP3372.java/.py/.cpp

2. P3373 【模板】线段树 2
- **类型**: 区间乘法更新 + 区间加法更新 + 区间求和
- **难度**: 模板
- **测试链接**: https://www.luogu.com.cn/problem/P3373
- **核心思想**: 支持乘法和加法的线段树

3. P4198 楼房重建
- **类型**: 区间最值 + 二分查找
- **难度**: Hard
- **测试链接**: https://www.luogu.com.cn/problem/P4198
- **核心思想**: 线段树维护区间斜率信息

4. P4145 上帝造题的七分钟 2 / 花神游历各国
- **类型**: 区间开方 + 线段树
- **难度**: Hard
- **测试链接**: https://www.luogu.com.cn/problem/P4145
- **核心思想**: 利用开方操作的收敛性优化

5. P3835 【模板】可持久化线段树 1（主席树）
- **类型**: 可持久化线段树
- **难度**: 模板
- **测试链接**: https://www.luogu.com.cn/problem/P3835

- **核心思想**: 主席树模板

6. P3953 逛公园
- **类型**: 动态规划 + 线段树优化
- **难度**: Hard
- **测试链接**: https://www.luogu.com.cn/problem/P3953
- **核心思想**: 线段树优化 DP 转移

7. P5357 【模板】AC 自动机（二次加强版）
- **类型**: AC 自动机 + 线段树合并
- **难度**: Hard
- **测试链接**: https://www.luogu.com.cn/problem/P5357
- **核心思想**: 线段树合并维护 fail 树信息

8. P5057 [CQOI2006] 简单题
- **类型**: 位运算 + 线段树
- **难度**: Medium
- **测试链接**: https://www.luogu.com.cn/problem/P5057
- **核心思想**: 线段树维护异或操作

LintCode 题目

1. LintCode 206. Interval Sum
- **类型**: 区间求和
- **难度**: Easy
- **测试链接**: https://www.lintcode.com/problem/206/
- **核心思想**: 基础线段树应用

2. LintCode 249. Count of Smaller Number before itself
- **类型**: 离散化 + 单点更新 + 区间求和
- **难度**: Medium
- **测试链接**: https://www.lintcode.com/problem/249/
- **核心思想**: 线段树维护值域信息

3. LintCode 207. Interval Sum II
- **类型**: 区间更新 + 区间求和
- **难度**: Medium
- **测试链接**: https://www.lintcode.com/problem/207/
- **核心思想**: 带懒惰传播的线段树

4. LintCode 439. Segment Tree Build II
- **类型**: 线段树构建（最大值）
- **难度**: Medium

- **测试链接**: https://www.lintcode.com/problem/439/
- **核心思想**: 构建维护最大值的线段树

5. LintCode 440. Backpack III
- **类型**: 动态规划 + 线段树优化
- **难度**: Hard
- **测试链接**: https://www.lintcode.com/problem/440/
- **核心思想**: 线段树优化多重背包

6. LintCode 548. Intersection of Two Arrays II
- **类型**: 哈希表 + 线段树
- **难度**: Easy
- **测试链接**: https://www.lintcode.com/problem/548/
- **核心思想**: 线段树维护元素计数

HackerRank 题目

1. Range Minimum Query
- **类型**: 区间最小值查询
- **难度**: Easy
- **测试链接**: https://www.hackerrank.com/challenges/range-minimum-query/problem
- **核心思想**: 线段树维护区间最小值

2. Persistent Segment Tree
- **类型**: 可持久化线段树
- **难度**: Hard
- **测试链接**: https://www.hackerrank.com/contests/world-codesprint-12/challenges/persistent-segment-tree-sum
- **核心思想**: 主席树维护历史版本

3. Square-Ten Tree
- **类型**: 分块线段树
- **难度**: Hard
- **测试链接**: https://www.hackerrank.com/challenges/square-ten-tree/problem
- **核心思想**: 分块线段树优化大范围操作

其他平台题目

1. UVa 11235 - Frequent values
- **类型**: 区间众数
- **难度**: Medium
- **测试链接**: https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&category=24&page=show_problem&p

roblem=2176
- **核心思想**: 线段树维护区间众数信息

2. UVa 12299 - RMQ with Shifts
- **类型**: 区间最小值查询（带循环移位）
- **难度**: Hard
- **测试链接**:
https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&category=24&page=show_problem&problem=3451
- **核心思想**: 线段树维护循环数组信息

3. CodeChef - CHEFEXQ
- **类型**: 区间异或查询
- **难度**: Medium
- **测试链接**: https://www.codechef.com/problems/CHEFEXQ
- **核心思想**: 线段树维护区间异或值

4. CodeChef - KFSTB
- **类型**: 线段树优化动态规划
- **难度**: Hard
- **测试链接**: https://www.codechef.com/problems/KFSTB
- **核心思想**: 线段树优化 DP 转移

5. Timus OJ 1470 - Strong Defence
- **类型**: 区间更新 + 线段树
- **难度**: Hard
- **测试链接**: https://acm.timus.ru/problem.aspx?space=1&num=1470
- **核心思想**: 带懒惰传播的线段树

6. Aizu OJ ALDS1_9_C - Segment Tree
- **类型**: 线段树基础操作
- **难度**: Easy
- **测试链接**: https://onlinejudge.u-aizu.ac.jp/problems/ALDS1_9_C
- **核心思想**: 线段树基础应用

7. Comet OJ C0146 - 区间修改区间查询
- **类型**: 线段树模板
- **难度**: Medium
- **测试链接**: https://cometoj.com/contest/32/problem/C
- **核心思想**: 带懒惰传播的线段树

8. 牛客网 NC14526 - 区间不同的数
- **类型**: 离线处理 + 线段树

- **难度**: Hard
- **测试链接**: https://ac.nowcoder.com/acm/problem/14526
- **核心思想**: 离线查询 + 线段树维护

总结

线段树作为一种重要的数据结构，在各种算法竞赛平台都有大量相关题目。掌握线段树的基本操作和各种变种，对于解决区间查询和更新问题具有重要意义。通过系统地练习这些题目，可以深入理解线段树的应用场景和优化技巧。

==========

文件: README.md
==========
class111 - 线段树专题 (Segment Tree)

概述

本目录专注于线段树(Segment Tree)这一重要数据结构的学习和实现。线段树是一种二叉树数据结构，用于存储区间或段的信息，能够高效地处理区间查询和区间更新操作。

已实现的代码文件

基础实现
1. **Code01_FallingSquares.java** - 掉落的方块 (LeetCode 699)
   - 类型: 区间最值查询 + 离散化
   - 应用: 计算机图形学中的碰撞检测

2. **Code02_VasesAndFlowers.java** - 瓶子里的花朵 (HDU 4614)
   - 类型: 区间求和 + 二分查找
   - 应用: 资源分配问题

3. **Code03_SquareRoot.java** - 范围上开平方并求累加和 (Luogu P4145)
   - 类型: 区间开方更新 + 区间求和
   - 应用: 数学计算优化

4. **Code04_QueryModUpdate.java** - 查询取模更新
   - 类型: 区间取模更新 + 区间求和
   - 应用: 数论相关计算

5. **Code05_Posters1.java / Code05_Posters2.java** - 贴海报问题
   - 类型: 区间覆盖 + 离散化
   - 应用: 区间调度问题

补充实现 (supplementary_problems 目录)
6. **Code06_RangeSumQueryMutable.java/.py** - 区间求和 - 可变 (LeetCode 307)
   - 类型: 单点更新 + 区间求和
   - 应用: 经典线段树应用

7. **Code07_RangeMaxQuery.java/.py** - 区间最大值查询
   - 类型: 单点更新 + 区间最值
   - 应用: RMQ 问题

8. **Code08_RangeAddQuery.java/.py** - 区间加法查询
   - 类型: 区间更新 + 单点查询
   - 应用: 区间批量操作

新增实现
9. **Code09_RangeSumQueryMutable.java/.py/.cpp** - 区间求和 - 可变 (LeetCode 307)
   - 类型: 单点更新 + 区间求和
   - 应用: 经典线段树应用

10. **Code10_CountOfSmallerNumbersAfterSelf.java/.py/.cpp** - 计算右侧小于当前元素的个数 (LeetCode 315)
    - 类型: 离散化 + 单点更新 + 区间求和
    - 应用: 逆序对计算

11. **Code11_HorribleQueries.java/.py/.cpp** - Horrible Queries (SPOJ HORRIBLE)
    - 类型: 区间更新 + 区间求和 + 懒惰传播
    - 应用: 区间批量操作

新增实现 (续)
12. **Code18_RangeSumQueryMutable.java/.py/.cpp** - 区间求和 - 可变 (LeetCode 307)
    - 类型: 单点更新 + 区间求和
    - 应用: 经典线段树应用

13. **Code19_GSS1.java/.py/.cpp** - Can you answer these queries I (SPOJ GSS1)
    - 类型: 区间最大子段和
    - 应用: 最大子数组和问题

14. **Code20_HDU1166.java/.py/.cpp** - 敌兵布阵 (HDU 1166)
    - 类型: 单点更新 + 区间求和
    - 应用: 经典线段树应用

15. **Code21_POJ3468.java/.py/.cpp** - A Simple Problem with Integers (POJ 3468)
    - 类型: 区间更新 + 区间求和 + 懒惰传播

- 应用: 区间批量操作

16. **Code22_LuoguP3372.java/.py/.cpp** - 【模板】线段树 1 (Luogu P3372)
    - 类型: 区间更新 + 区间求和 + 懒惰传播
    - 应用: 线段树模板题

高级哈希应用题目 (新增)
12. **Code13_HashCollision.java/.cpp/.py** - 哈希冲突检测与解决
    - 类型: 哈希冲突、开放寻址法、链地址法
    - 应用: 哈希表实现、冲突解决策略
    - 复杂度: 平均 O(1)，最坏 O(n)

13. **Code14_ConsistentHashing.java/.cpp/.py** - 一致性哈希算法
    - 类型: 分布式哈希、虚拟节点、负载均衡
    - 应用: 分布式系统、缓存分片
    - 复杂度: O(log n)查找，O(1)插入删除

14. **Code15_BloomFilter.java/.cpp/.py** - 布隆过滤器实现
    - 类型: 概率数据结构、空间效率优化
    - 应用: 缓存穿透防护、URL 去重
    - 复杂度: O(k)插入查询，k 为哈希函数数量

15. **Code16_PerfectHashing.java/.cpp/.py** - 完美哈希算法
    - 类型: 无冲突哈希、两级哈希结构
    - 应用: 静态数据集、字典查找
    - 复杂度: O(1)查找，$O(n^2)$构建

16. **Code17_RollingHash.java/.cpp/.py** - 滚动哈希算法
    - 类型: 字符串哈希、滑动窗口
    - 应用: 字符串匹配、子串查找
    - 复杂度: O(n)预处理，O(1)子串哈希

文档文件

1. **SEGMENT_TREE_PROBLEMS.md** - 线段树题目大全
    - 详细介绍了已实现的线段树题目
    - 分析了线段树的核心知识点和应用场景
    - 提供了工程化考虑和语言特性差异分析

2. **ADDITIONAL_SEGMENT_TREE_PROBLEMS.md** - 更多线段树题目
    - 列出了各大平台的线段树相关题目
    - 包含 LeetCode、Codeforces、SPOJ、AtCoder、HDU、POJ、洛谷等平台的题目
    - 提供了每道题目的类型、难度和核心思想

线段树的核心概念

基本操作
- **构建**: O(n)时间复杂度
- **单点更新**: O(log n)时间复杂度
- **区间更新**: O(log n)时间复杂度(带懒惰传播)
- **区间查询**: O(log n)时间复杂度

常见变种
1. **基础线段树**: 支持单点更新和区间查询
2. **带懒惰传播的线段树**: 支持区间更新和区间查询
3. **动态开点线段树**: 节省空间，适用于稀疏数据
4. **主席树**: 可持久化线段树，支持历史版本查询

应用场景
- 区间最值查询(RMQ)
- 区间求和
- 区间更新
- 离散化处理
- 逆序对计算

高级哈希算法核心概念

哈希冲突解决策略
1. **链地址法**: 使用链表处理冲突
2. **开放寻址法**: 线性探测、二次探测、双重哈希
3. **再哈希法**: 使用多个哈希函数

高级哈希应用
1. **一致性哈希**: 分布式系统负载均衡
2. **布隆过滤器**: 空间效率优化的概率数据结构
3. **完美哈希**: 无冲突哈希表
4. **滚动哈希**: 字符串匹配和子串查找

哈希算法复杂度对比
算法类型	平均查找	最坏查找	空间复杂度	适用场景
标准哈希表	O(1)	O(n)	O(n)	通用场景
一致性哈希	O(log n)	O(n)	O(n)	分布式系统
布隆过滤器	O(k)	O(k)	O(m)	存在性检查
完美哈希	O(1)	O(1)	O(n)	静态数据集
滚动哈希	O(1)	O(1)	O(n)	字符串处理

学习建议

线段树学习路径
1. **从基础开始**: 先掌握基础线段树的实现和应用
2. **理解懒惰传播**: 学习带懒惰传播的线段树处理区间更新
3. **练习经典题目**: 通过练习经典题目加深理解
4. **掌握离散化**: 学会在大数据范围下使用离散化技巧
5. **扩展应用**: 了解线段树的高级应用如主席树、二维线段树等

哈希算法学习路径
1. **理解哈希原理**: 掌握哈希函数、冲突解决机制
2. **实践冲突解决**: 实现链地址法、开放寻址法等策略
3. **学习高级应用**: 一致性哈希、布隆过滤器、完美哈希
4. **性能调优**: 负载因子控制、哈希函数选择、扩容策略
5. **工程化实践**: 异常处理、线程安全、测试覆盖

综合训练建议
1. **对比学习**: 对比不同哈希算法的适用场景和性能特点
2. **实际应用**: 将哈希算法应用到实际问题中，如缓存系统、分布式存储
3. **性能分析**: 使用性能分析工具评估不同实现的效率
4. **代码审查**: 学习优秀开源项目的哈希算法实现
5. **面试准备**: 准备哈希算法相关的面试问题和回答模板

复杂度分析

操作类型	时间复杂度	空间复杂度
构建	O(n)	O(n)
单点更新	O(log n)	O(1)
区间更新	O(log n)	O(1)
单点查询	O(log n)	O(1)
区间查询	O(log n)	O(1)

工程化考虑

线段树工程化
1. **异常处理**: 输入验证、边界检查、空树处理
2. **性能优化**: 懒惰传播、剪枝优化、内存池管理
3. **可维护性**: 代码模块化、接口清晰、文档完善
4. **跨语言实现**: Java、Python、C++三种语言实现对比

哈希算法工程化

1. **异常防御**: 哈希函数稳定性、碰撞攻击防护
2. **性能调优**: 哈希函数选择、负载因子控制、扩容策略
3. **内存管理**: 动态扩容、内存泄漏检测、缓存友好性
4. **线程安全**: 并发访问控制、锁粒度优化
5. **测试覆盖**: 单元测试、性能测试、边界测试

调试与优化技巧
1. **调试能力**: 打印中间过程、断言验证、性能分析
2. **笔试技巧**: 模板化代码、边界处理、时间复杂度分析
3. **面试表达**: 算法原理、工程考量、踩坑经验分享

语言特性差异
1. **Java**: 垃圾回收、异常机制、集合框架
2. **C++**: 内存管理、模板编程、STL 容器
3. **Python**: 动态类型、内置数据结构、解释器特性

参考资料

线段树参考资料
- [LeetCode Segment Tree Problems](https://leetcode.com/problem-list/segment-tree/)
- [Codeforces Segment Tree Blog](https://codeforces.com/blog/entry/22616)
- [SPOJ Segment Tree Problems](https://www.spoj.com/problems/tags/)
- [CP-Algorithms Segment Tree](https://cp-algorithms.com/data_structures/segment_tree.html)

哈希算法参考资料
- [LeetCode Hash Table Problems](https://leetcode.com/tag/hash-table/)
- [GeeksforGeeks Hashing Data Structure](https://www.geeksforgeeks.org/hashing-data-structure/)
- [CP-Algorithms Hashing](https://cp-algorithms.com/string/string-hashing.html)
- [Wikipedia Hash Table](https://en.wikipedia.org/wiki/Hash_table)
- [Bloom Filter Applications](https://en.wikipedia.org/wiki/Bloom_filter)
- [Consistent Hashing Theory](https://en.wikipedia.org/wiki/Consistent_hashing)

各大算法平台哈希题目
- **LeetCode**: Two Sum, Valid Anagram, Group Anagrams, Longest Substring Without Repeating Characters
- **Codeforces**: Rolling Hash Problems, String Matching Problems
- **SPOJ**: Hashing Problems, String Problems
- **HackerRank**: Hash Tables, String Manipulation
- **AtCoder**: String Hashing Problems
- **POJ/HDU**: Hash Algorithm Problems
- **洛谷**: 哈希算法题目
- **牛客网**: 哈希算法面试题
- **剑指 Offer**: 哈希相关面试题

开源实现参考
- **Java**: HashMap 源码、ConcurrentHashMap 源码
- **C++**: STL unordered_map、Google dense_hash_map
- **Python**: dict 实现、collections.Counter
- **Redis**: 哈希表实现、布隆过滤器模块
- **Memcached**: 一致性哈希实现

==========

文件: SEGMENT_TREE_PROBLEMS.md
==========
线段树题目大全 - Segment Tree Problems Comprehensive Guide

概述

线段树是一种非常重要的数据结构，广泛应用于各种算法竞赛和工程实践中。它能够高效地处理区间查询和区间更新操作，在时间复杂度上通常能达到O(log n)的效率。

已实现的线段树题目

1. LeetCode 699. Falling Squares (掉落的方块)
- **文件**: Code01_FallingSquares.java
- **类型**: 区间最值查询 + 离散化
- **难度**: Hard
- **核心思想**: 使用线段树维护区间最大值，结合坐标离散化处理大数据范围
- **应用场景**: 计算机图形学中的碰撞检测、俄罗斯方块游戏等

2. HDU 4614. Vases and Flowers (花瓶与花朵)
- **文件**: Code02_VasesAndFlowers.java
- **类型**: 区间求和 + 二分查找
- **难度**: Hard
- **核心思想**: 使用线段树维护区间和，结合二分查找确定插入位置
- **应用场景**: 资源分配、任务调度等

3. Luogu P4145. 上帝造题的七分钟 2 / 花神游历各国 (范围开方求和)
- **文件**: Code03_SquareRoot.java
- **类型**: 区间开方更新 + 区间求和
- **难度**: Hard
- **核心思想**: 利用开方操作的收敛性进行剪枝优化
- **应用场景**: 数学计算优化、特殊函数处理等

4. Range Sum Query - Mutable (区间求和 - 可变)

- **文件**: Code09_RangeSumQueryMutable.java/.py
- **类型**: 单点更新 + 区间求和
- **难度**: Medium
- **核心思想**: 经典线段树应用，支持单点更新和区间求和
- **应用场景**: 数据统计、前缀和计算等
- **测试链接**:
  - https://leetcode.cn/problems/range-sum-query-mutable
  - https://leetcode.com/problems/range-sum-query-mutable

5. LeetCode 315. Count of Smaller Numbers After Self (计算右侧小于当前元素的个数)
- **文件**: Code10_CountOfSmallerNumbersAfterSelf.java/.py/.cpp
- **类型**: 离散化 + 单点更新 + 区间求和
- **难度**: Hard
- **核心思想**: 使用线段树维护值域信息，结合离散化处理
- **应用场景**: 逆序对计算、排名统计等
- **测试链接**:
  - https://leetcode.cn/problems/count-of-smaller-numbers-after-self
  - https://leetcode.com/problems/count-of-smaller-numbers-after-self

6. SPOJ HORRIBLE - Horrible Queries (区间更新和查询)
- **文件**: Code11_HorribleQueries.java/.py/.cpp
- **类型**: 区间更新 + 区间求和 + 懒惰传播
- **难度**: Hard
- **核心思想**: 使用带懒惰传播的线段树处理区间更新
- **应用场景**: 区间批量操作、数据批量更新等
- **测试链接**: https://www.spoj.com/problems/HORRIBLE/

线段树的核心知识点

1. 基本概念
- 线段树是一棵二叉树，每个节点代表一个区间
- 叶子节点代表单个元素，非叶子节点代表区间的并集
- 通常需要 4 倍原数组大小的空间

2. 基本操作
- **构建**: O(n)
- **单点更新**: O(log n)
- **区间更新**: O(log n) (带懒惰传播)
- **区间查询**: O(log n)

3. 常见变种
- **基础线段树**: 支持单点更新和区间查询
- **带懒惰传播的线段树**: 支持区间更新和区间查询

- **动态开点线段树**：节省空间，适用于稀疏数据
- **主席树**：可持久化线段树，支持历史版本查询

线段树的典型应用场景

1. 区间最值查询（RMQ）
- 查询区间内的最大值或最小值
- 应用：股票价格分析、性能监控等

2. 区间求和
- 计算区间内所有元素的和
- 应用：数据统计、积分计算等

3. 区间更新
- 对区间内所有元素进行统一操作
- 应用：批量数据修改、区域设置等

4. 离散化处理
- 处理大数据范围但实际数据稀疏的情况
- 应用：坐标压缩、排名计算等

5. 逆序对计算
- 计算数组中逆序对的个数
- 应用：排序算法分析、相似度计算等

线段树的时间复杂度分析

操作类型	时间复杂度	说明
构建	O(n)	从底向上构建整棵树
单点更新	O(log n)	从根到叶子节点的路径
区间更新	O(log n)	带懒惰传播的区间更新
单点查询	O(log n)	从根到叶子节点的路径
区间查询	O(log n)	最多访问两层节点

线段树的空间复杂度分析

线段树需要 4 倍原数组大小的空间，即 O(4n) = O(n)。

工程化考虑

1. 异常处理
- 输入验证：检查数组边界、操作合法性等

- 错误恢复：在出现异常时能够恢复到一致状态

2. 性能优化
- 懒惰传播：避免不必要的更新操作
- 剪枝优化：利用问题特性减少计算量
- 内存优化：动态开点、压缩存储等

3. 可维护性
- 代码模块化：将线段树封装成独立类
- 接口清晰：提供简洁易用的 API
- 注释完整：详细说明算法原理和实现细节

语言特性差异

Java
- 面向对象封装良好
- 自动内存管理
- 丰富的集合类库

Python
- 语法简洁易读
- 动态类型
- 列表推导式等高级特性

C++
- 性能优秀
- 手动内存管理
- 模板支持泛型编程

线段树与其他数据结构的对比

数据结构	构建	更新	查询	适用场景
线段树	O(n)	O(log n)	O(log n)	区间操作频繁
树状数组	O(n)	O(log n)	O(log n)	区间求和、前缀和
平衡树	O(n log n)	O(log n)	O(log n)	动态维护有序序列
分块	O(n)	O(√n)	O(√n)	简单实现、在线算法

常见问题和解决方案

1. 懒惰传播标记错误
- **问题**: 更新操作后查询结果不正确
- **解决方案**: 确保在所有访问节点前都正确下推懒惰标记

2. 数组越界
- **问题**: 访问线段树数组时出现越界
- **解决方案**: 确保线段树数组大小足够(通常为 4n)

3. 离散化错误
- **问题**: 离散化后无法正确映射原值
- **解决方案**: 使用二分查找确保正确映射

扩展应用

1. 二维线段树
- 处理二维区间查询问题
- 应用：图像处理、地理信息系统等

2. 可持久化线段树(主席树)
- 支持历史版本查询
- 应用：版本控制、回滚操作等

3. 动态开点线段树
- 节省空间，适用于稀疏数据
- 应用：大数据范围但实际数据稀疏的场景

总结

线段树是一种非常强大的数据结构，掌握其原理和实现对于算法学习和工程实践都具有重要意义。通过大量练习不同类型的线段树题目，可以深入理解其应用场景和优化技巧，为解决实际问题提供有力工具。

==========

[代码文件]
==========
文件: Code01_FallingSquares.java
==========
```
package class111;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

// 掉落的方块 (Falling Squares)
// 题目来源: LeetCode 699. Falling Squares
// 题目链接: https://leetcode.cn/problems/falling-squares
```

```
// 题目链接: https://leetcode.com/problems/falling-squares
//
// 题目描述:
// 有一个二维平面，x 轴是最底的边界
// 给定二维整数数组 pos，pos[i] = [ lefti, leni ]
// 表示第 i 个方块边长为 leni，左侧边缘在 x = lefti 位置，所在高度非常高
// 所有方块都是正方形，依次从高处垂直掉落，也就是左边界顺着 x = lefti 往下
// 如果掉落的方块碰到已经掉落正方形的顶边或者 x 轴就停止掉落
// 如果方块掉落时仅仅是擦过已经掉落正方形的左侧边或右侧边，并不会停止掉落
// 一旦停止，它就会固定在那里，无法再移动，俄罗斯方块游戏和本题意思一样
// 返回一个整数数组 ans ，其中 ans[i]表示在第 i 块方块掉落后整体的最大高度
// 1 <= pos 数组长度 <= 1000，1 <= lefti <= 10^8，1 <= leni <= 10^6
//
// 解题思路:
// 1. 使用线段树维护区间最大值
// 2. 由于坐标范围很大(10^8)，需要进行离散化处理
// 3. 对于每个掉落的方块，先查询其底部区间内的最大高度
// 4. 方块落下后的高度等于底部最大高度加上方块自身高度
// 5. 更新该方块所占区间的高度值
// 6. 记录并返回每次掉落后的整体最大高度
//
// 时间复杂度: O(n log n)，其中 n 为方块数量
// 空间复杂度: O(n)
public class Code01_FallingSquares {

    public static int MAXN = 2001;

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

    public static int[] max = new int[MAXN << 2];

    public static int[] change = new int[MAXN << 2];

    public static boolean[] update = new boolean[MAXN << 2];

    // 收集所有可能出现的坐标点并去重排序，用于离散化
    public static int collect(int[][] poss) {
        int size = 1;
        for (int[] pos : poss) {
            arr[size++] = pos[0];
            arr[size++] = pos[0] + pos[1] - 1;
        }
        Arrays.sort(arr, 1, size);
```

```
    int n = 1;
    for (int i = 2; i < size; i++) {
        if (arr[n] != arr[i]) {
            arr[++n] = arr[i];
        }
    }
    return n;
}

// 二分查找值 v 在离散化数组中的位置
public static int rank(int n, int v) {
    int ans = 0;
    int l = 1, r = n, m;
    while (l <= r) {
        m = (l + r) >> 1;
        if (arr[m] >= v) {
            ans = m;
            r = m - 1;
        } else {
            l = m + 1;
        }
    }
    return ans;
}

// 向上更新节点值
public static void up(int i) {
    max[i] = Math.max(max[i << 1], max[i << 1 | 1]);
}

// 下发懒惰标记
public static void down(int i) {
    if (update[i]) {
        lazy(i << 1, change[i]);
        lazy(i << 1 | 1, change[i]);
        update[i] = false;
    }
}

// 设置懒惰标记
public static void lazy(int i, int v) {
    update[i] = true;
    change[i] = v;
```

```
    max[i] = v;
}

// 构建线段树
public static void build(int l, int r, int i) {
    if (l < r) {
        int mid = (l + r) >> 1;
        build(l, mid, i << 1);
        build(mid + 1, r, i << 1 | 1);
    }
    max[i] = 0;
    change[i] = 0;
    update[i] = false;
}

// 区间更新操作
public static void update(int jobl, int jobr, int jobv, int l, int r, int i) {
    if (jobl <= l && r <= jobr) {
        lazy(i, jobv);
    } else {
        int mid = (l + r) >> 1;
        down(i);
        if (jobl <= mid) {
            update(jobl, jobr, jobv, l, mid, i << 1);
        }
        if (jobr > mid) {
            update(jobl, jobr, jobv, mid + 1, r, i << 1 | 1);
        }
        up(i);
    }
}

// 区间查询操作
public static int query(int jobl, int jobr, int l, int r, int i) {
    if (jobl <= l && r <= jobr) {
        return max[i];
    }
    int mid = (l + r) >> 1;
    down(i);
    int ans = Integer.MIN_VALUE;
    if (jobl <= mid) {
        ans = Math.max(ans, query(jobl, jobr, l, mid, i << 1));
    }
```

```
        if (jobr > mid) {
            ans = Math.max(ans, query(jobl, jobr, mid + 1, r, i << 1 | 1));
        }
        return ans;
    }

    // 主函数：计算每个方块掉落后整体的最大高度
    public static List<Integer> fallingSquares(int[][] pos) {
        int n = collect(pos);
        build(1, n, 1);
        List<Integer> ans = new ArrayList<>();
        int max = 0, l, r, h;
        for (int[] square : pos) {
            l = rank(n, square[0]);
            r = rank(n, square[0] + square[1] - 1);
            h = query(l, r, 1, n, 1) + square[1];
            max = Math.max(max, h);
            ans.add(max);
            update(l, r, h, 1, n, 1);
        }
        return ans;
    }

}
```

==========

文件: Code02_VasesAndFlowers.java

==========

```
package class111;

import java.io.*;

// 瓶子里的花朵 (Vases and Flowers)
// 题目来源: HDU 4614. Vases and Flowers
// 题目链接: https://acm.hdu.edu.cn/showproblem.php?pid=4614
//
// 题目描述:
// 给定 n 个瓶子，编号从 0~n-1，一开始所有瓶子都是空的
// 每个瓶子最多插入一朵花，实现以下两种类型的操作
// 操作 1 from flower : 一共有 flower 朵花，从 from 位置开始依次插入花朵，已经有花的瓶子跳过
//                      如果一直到最后的瓶子，花也没有用完，就丢弃剩下的花朵
//                      返回这次操作插入的首个空瓶的位置 和 最后空瓶的位置
```

```
//                      如果从 from 开始所有瓶子都有花，打印"Can not put any one."
// 操作 2 left right  : 从 left 位置开始到 right 位置的瓶子，变回空瓶，返回清理花朵的数量
//
// 解题思路:
// 1. 使用线段树维护区间和，表示每个区间内花朵的数量
// 2. 对于插入操作，需要找到从指定位置开始的第 k 个空位置
// 3. 对于清理操作，直接查询区间和并更新区间为 0
// 4. 利用线段树的区间查询和更新功能高效处理操作
//
// 时间复杂度: O(m log n)，其中 m 为操作次数，n 为瓶子数量
// 空间复杂度: O(n)
public class Code02_VasesAndFlowers {

    public static int MAXN = 50001;

    public static int[] sum = new int[MAXN << 2];

    public static int[] change = new int[MAXN << 2];

    public static boolean[] update = new boolean[MAXN << 2];

    public static int n;

    // 向上更新节点值（区间和）
    public static void up(int i) {
        sum[i] = sum[i << 1] + sum[i << 1 | 1];
    }

    // 下发懒惰标记
    public static void down(int i, int ln, int rn) {
        if (update[i]) {
            lazy(i << 1, change[i], ln);
            lazy(i << 1 | 1, change[i], rn);
            update[i] = false;
        }
    }

    // 设置懒惰标记
    public static void lazy(int i, int v, int n) {
        sum[i] = v * n;
        change[i] = v;
        update[i] = true;
    }
```

```
// 构建线段树
public static void build(int l, int r, int i) {
    if (l < r) {
        int mid = (l + r) >> 1;
        build(l, mid, i << 1);
        build(mid + 1, r, i << 1 | 1);
    }
    sum[i] = 0;
    update[i] = false;
}

// 区间更新操作
public static void update(int jobl, int jobr, int jobv, int l, int r, int i) {
    if (jobl <= l && r <= jobr) {
        lazy(i, jobv, r - l + 1);
    } else {
        int mid = (l + r) >> 1;
        down(i, mid - l + 1, r - mid);
        if (jobl <= mid) {
            update(jobl, jobr, jobv, l, mid, i << 1);
        }
        if (jobr > mid) {
            update(jobl, jobr, jobv, mid + 1, r, i << 1 | 1);
        }
        up(i);
    }
}

// 区间查询操作（查询区间和）
public static int query(int jobl, int jobr, int l, int r, int i) {
    if (jobl <= l && r <= jobr) {
        return sum[i];
    }
    int mid = (l + r) >> 1;
    down(i, mid - l + 1, r - mid);
    int ans = 0;
    if (jobl <= mid) {
        ans += query(jobl, jobr, l, mid, i << 1);
    }
    if (jobr > mid) {
        ans += query(jobl, jobr, mid + 1, r, i << 1 | 1);
    }
```

```
    return ans;
}

// 插入花朵操作
public static int[] insert(int from, int flowers) {
    // 题目给的位置从 0 开始
    // 线段树下标从 1 开始
    from++;
    int start, end;
    int zeros = n - from + 1 - query(from, n, 1, n, 1);
    if (zeros == 0) {
        start = 0;
        end = 0;
    } else {
        start = findZero(from, 1);
        end = findZero(from, Math.min(zeros, flowers));
        update(start, end, 1, 1, n, 1);
    }
    // 题目需要从 0 开始的下标
    start--;
    end--;
    return new int[] { start, end };
}

// 在 s~n 范围内查找第 k 个空位置
public static int findZero(int s, int k) {
    int l = s, r = n, mid;
    int ans = 0;
    while (l <= r) {
        mid = (l + r) >> 1;
        if (mid - s + 1 - query(s, mid, 1, n, 1) >= k) {
            ans = mid;
            r = mid - 1;
        } else {
            l = mid + 1;
        }
    }
    return ans;
}

// 清理花朵操作
// 注意题目给的下标从 0 开始
// 线段树下标从 1 开始
```

```
public static int clear(int left, int right) {
    left++;
    right++;
    int ans = query(left, right, 1, n, 1);
    update(left, right, 0, 1, n, 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();
    int t = (int) in.nval;
    for (int i = 1; i <= t; i++) {
        in.nextToken();
        n = (int) in.nval;
        in.nextToken();
        int m = (int) in.nval;
        build(1, n, 1);
        for (int j = 1; j <= m; j++) {
            in.nextToken();
            int op = (int) in.nval;
            if (op == 1) {
                in.nextToken();
                int from = (int) in.nval;
                in.nextToken();
                int flowers = (int) in.nval;
                int[] ans = insert(from, flowers);
                if (ans[0] == -1) {
                    out.println("Can not put any one.");
                } else {
                    out.println(ans[0] + " " + ans[1]);
                }
            } else {
                in.nextToken();
                int left = (int) in.nval;
                in.nextToken();
                int right = (int) in.nval;
                out.println(clear(left, right));
            }
        }
        out.println();
```

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

}
```

==========

文件: Code03_SquareRoot.java

==========

```
package class111;

import java.io.*;

// 范围上开平方并求累加和 (Square Root and Sum)
// 题目来源: Luogu P4145. 上帝造题的七分钟 2 / 花神游历各国
// 题目链接: https://www.luogu.com.cn/problem/P4145
//
// 题目描述:
// 给定一个长度为 n 的数组 arr，实现以下两种类型的操作
// 操作 0 l r : 把 arr[l..r]范围上的每个数开平方，结果向下取整
// 操作 1 l r : 查询 arr[l..r]范围上所有数字的累加和
// 两种操作一共发生 m 次，数据中有可能 l>r，遇到这种情况请交换 l 和 r
// 1 <= n, m <= 10^5，1 <= arr[i] <= 10^12
//
// 解题思路:
// 1. 使用线段树维护区间和与区间最大值
// 2. 对于开方操作，利用开方的收敛性进行剪枝优化
// 3. 当区间最大值为 1 时，开方操作不会改变任何值，可以剪枝
// 4. 通过势能分析，总的时间复杂度为 O(n * 6 * log n)
//
// 时间复杂度: O(m * log n)，其中 m 为操作次数，n 为数组长度
// 空间复杂度: O(n)
public class Code03_SquareRoot {

    public static int MAXN = 100001;

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

    public static long[] sum = new long[MAXN << 2];
```

```
public static long[] max = new long[MAXN << 2];

// 向上更新节点值（区间和与区间最大值）
public static void up(int i) {
    sum[i] = sum[i << 1] + sum[i << 1 | 1];
    max[i] = Math.max(max[i << 1], max[i << 1 | 1]);
}

// 构建线段树
public static void build(int l, int r, int i) {
    if (l == r) {
        sum[i] = arr[l];
        max[i] = arr[l];
    } else {
        int mid = (l + r) >> 1;
        build(l, mid, i << 1);
        build(mid + 1, r, i << 1 | 1);
        up(i);
    }
}

// 区间开方操作（核心函数）
// 注意和常规线段树不一样，这里没有懒更新，也就不需要有 down 方法
// 只有根据范围最大值信息的剪枝
// 时间复杂度的分析就是课上讲的势能分析
// 不用纠结单次调用的复杂度
// 哪怕调用再多次 sqrt 方法，总的时间复杂度也就是 O(n * 6 * logn)
public static void sqrt(int jobl, int jobr, int l, int r, int i) {
    if (l == r) {
        long sqrt = (long) Math.sqrt(max[i]);
        sum[i] = sqrt;
        max[i] = sqrt;
    } else {
        int mid = (l + r) >> 1;
        // 剪枝优化：只有当区间最大值大于 1 时才需要继续处理
        if (jobl <= mid && max[i << 1] > 1) {
            sqrt(jobl, jobr, l, mid, i << 1);
        }
        if (jobr > mid && max[i << 1 | 1] > 1) {
            sqrt(jobl, jobr, mid + 1, r, i << 1 | 1);
        }
        up(i);
    }
```

```
}

// 区间查询操作（查询区间和）
// 没有懒更新，不需要调用 down 方法
public static long query(int jobl, int jobr, int l, int r, int i) {
    if (jobl <= l && r <= jobr) {
        return sum[i];
    }
    int mid = (l + r) >> 1;
    long ans = 0;
    if (jobl <= mid) {
        ans += query(jobl, jobr, l, mid, i << 1);
    }
    if (jobr > mid) {
        ans += query(jobl, jobr, mid + 1, r, i << 1 | 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();
    int n = (int) in.nval;
    for (int i = 1; i <= n; i++) {
        in.nextToken();
        arr[i] = (long) in.nval;
    }
    build(1, n, 1);
    in.nextToken();
    int m = (int) in.nval;
    for (int i = 1, op, jobl, jobr, tmp; i <= m; i++) {
        in.nextToken();
        op = (int) in.nval;
        in.nextToken();
        jobl = (int) in.nval;
        in.nextToken();
        jobr = (int) in.nval;
        if (jobl > jobr) {
            tmp = jobl;
            jobl = jobr;
            jobr = tmp;
```

```
            }
            if (op == 0) {
                sqrt(jobl, jobr, 1, n, 1);
            } else {
                out.println(query(jobl, jobr, 1, n, 1));
            }
        }
        out.flush();
        out.close();
        br.close();
    }

}
```

==========

文件: Code04_QueryModUpdate.java

==========

```
package class111;

import java.io.*;

// 包含取模操作的线段树 (Query and Mod Update)
// 题目来源: Codeforces 438D. The Child and Sequence
// 题目链接: https://codeforces.com/problemset/problem/438/D
// 题目链接: https://www.luogu.com.cn/problem/CF438D
//
// 题目描述:
// 给定一个长度为 n 的数组 arr，实现如下三种操作，一共调用 m 次
// 操作 1 l r : 查询 arr[l..r]的累加和
// 操作 2 l r x : 把 arr[l..r]上每个数字对 x 取模
// 操作 3 k x : 把 arr[k]上的数字设置为 x
// 1 <= n, m <= 10^5，操作 1 得到的结果，有可能超过 int 范围
//
// 解题思路:
// 1. 使用线段树维护区间和与区间最大值
// 2. 对于取模操作，利用剪枝优化：当取模数大于区间最大值时，取模操作不会改变任何值
// 3. 单点更新操作直接修改对应位置的值
// 4. 区间查询操作返回区间和
//
// 时间复杂度: O(m * log n)，其中 m 为操作次数，n 为数组长度
// 空间复杂度: O(n)
public class Code04_QueryModUpdate {
```

```
public static int MAXN = 100001;

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

public static long[] sum = new long[MAXN << 2];

public static int[] max = new int[MAXN << 2];

// 向上更新节点值（区间和与区间最大值）
public static void up(int i) {
    sum[i] = sum[i << 1] + sum[i << 1 | 1];
    max[i] = Math.max(max[i << 1], max[i << 1 | 1]);
}

// 构建线段树
public static void build(int l, int r, int i) {
    if (l == r) {
        sum[i] = max[i] = arr[l];
    } else {
        int mid = (l + r) >> 1;
        build(l, mid, i << 1);
        build(mid + 1, r, i << 1 | 1);
        up(i);
    }
}

// 区间查询操作（查询区间和）
public static long query(int jobl, int jobr, int l, int r, int i) {
    if (jobl <= l && r <= jobr) {
        return sum[i];
    }
    int mid = (l + r) >> 1;
    long ans = 0;
    if (jobl <= mid) {
        ans += query(jobl, jobr, l, mid, i << 1);
    }
    if (jobr > mid) {
        ans += query(jobl, jobr, mid + 1, r, i << 1 | 1);
    }
    return ans;
}
```

```
// 区间取模操作
public static void mod(int jobl, int jobr, int jobv, int l, int r, int i) {
    // 剪枝优化：当取模数大于区间最大值时，取模操作不会改变任何值
    if (jobv > max[i]) {
        return;
    }
    if (l == r) {
        sum[i] %= jobv;
        max[i] %= jobv;
    } else {
        int mid = (l + r) >> 1;
        if (jobl <= mid) {
            mod(jobl, jobr, jobv, l, mid, i << 1);
        }
        if (jobr > mid) {
            mod(jobl, jobr, jobv, mid + 1, r, i << 1 | 1);
        }
        up(i);
    }
}

// 单点更新操作
public static void update(int jobi, int jobv, int l, int r, int i) {
    if (l == r) {
        sum[i] = max[i] = jobv;
    } else {
        int mid = (l + r) >> 1;
        if (jobi <= mid) {
            update(jobi, jobv, l, mid, i << 1);
        } else {
            update(jobi, jobv, mid + 1, r, i << 1 | 1);
        }
        up(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();
    int n = (int) in.nval;
    in.nextToken();
```

```
        int m = (int) in.nval;
        for (int i = 1; i <= n; i++) {
            in.nextToken();
            arr[i] = (int) in.nval;
        }
        build(1, n, 1);
        for (int i = 1, op; i <= m; i++) {
            in.nextToken();
            op = (int) in.nval;
            if (op == 1) {
                in.nextToken();
                int jobl = (int) in.nval;
                in.nextToken();
                int jobr = (int) in.nval;
                out.println(query(jobl, jobr, 1, n, 1));
            } else if (op == 2) {
                in.nextToken();
                int jobl = (int) in.nval;
                in.nextToken();
                int jobr = (int) in.nval;
                in.nextToken();
                int jobv = (int) in.nval;
                mod(jobl, jobr, jobv, 1, n, 1);
            } else {
                in.nextToken();
                int jobi = (int) in.nval;
                in.nextToken();
                int jobv = (int) in.nval;
                update(jobi, jobv, 1, n, 1);
            }
        }
        out.flush();
        out.close();
        br.close();
    }

}
```

==========

文件: Code05_Posters1.java

==========

```
package class111;
```

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

// 贴海报 (Posters)
// 题目来源: Luogu P3740. 贴海报
// 题目链接: https://www.luogu.com.cn/problem/P3740
//
// 题目描述:
// 有一面墙，有固定高度，长度为 n，有 m 张海报，所有海报的高度都和墙的高度相同
// 从第 1 张海报开始，一张一张往墙上贴，直到 n 张海报贴完
// 每张海报都给出张贴位置(xi, yi)，表示第 i 张海报从墙的左边界 xi 一直延伸到右边界 yi
// 有可能发生后面的海报把前面的海报完全覆盖，导致看不到的情况
// 当所有海报贴完，返回能看到海报的数量，哪怕只漏出一点的海报都算
// 1 <= n、xi、yi <= 10^7，1 <= m <= 10^3
//
// 解题思路:
// 1. 使用线段树维护区间覆盖情况，记录每个区间被哪张海报完全覆盖
// 2. 由于坐标范围很大，需要进行离散化处理
// 3. 从第一张海报开始依次张贴，每次更新对应区间的海报编号
// 4. 最后查询整个墙面上可见的海报数量
// 5. 利用懒惰传播优化区间更新操作
//
// 时间复杂度: O(m log n)，其中 m 为海报数量，n 为墙面长度
// 空间复杂度: O(m)
public class Code05_Posters1 {

    public static int MAXM = 1001;

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

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

    public static int[] num = new int[MAXM << 2];

    // 线段树的某个范围上是否被设置成了统一的海报
    // 如果 poster[i] != 0，poster[i]表示统一海报的编号
    // 如果 poster[i] == 0，表示该范围上没有海报或者海报编号没统一
    public static int[] poster = new int[MAXM << 4];

    // 某种海报编号是否已经被统计过了
    // 只在最后一次查询，最后统计海报数量的阶段时候使用
    public static boolean[] visited = new boolean[MAXM];
```

```
// 收集所有坐标点并进行离散化处理
public static int collect(int m) {
    Arrays.sort(num, 1, m + 1);
    int size = 1;
    for (int i = 2; i <= m; i++) {
        if (num[size] != num[i]) {
            num[++size] = num[i];
        }
    }
    int cnt = size;
    for (int i = 2; i <= size; i++) {
        if (num[i - 1] + 1 < num[i]) {
            num[++cnt] = num[i - 1] + 1;
        }
    }
    Arrays.sort(num, 1, cnt + 1);
    return cnt;
}

// 二分查找值 v 在离散化数组中的位置
public static int rank(int l, int r, int v) {
    int m;
    int ans = 0;
    while (l <= r) {
        m = (l + r) >> 1;
        if (num[m] >= v) {
            ans = m;
            r = m - 1;
        } else {
            l = m + 1;
        }
    }
    return ans;
}

// 下发懒惰标记
public static void down(int i) {
    if (poster[i] != 0) {
        poster[i << 1] = poster[i];
        poster[i << 1 | 1] = poster[i];
        poster[i] = 0;
    }
```

```
}

// 构建线段树
public static void build(int l, int r, int i) {
    if (l < r) {
        int mid = (l + r) >> 1;
        build(l, mid, i << 1);
        build(mid + 1, r, i << 1 | 1);
    }
    poster[i] = 0;
}

// 区间更新操作（张贴海报）
public static void update(int jobl, int jobr, int jobv, int l, int r, int i) {
    if (jobl <= l && r <= jobr) {
        poster[i] = jobv;
    } else {
        down(i);
        int mid = (l + r) >> 1;
        if (jobl <= mid) {
            update(jobl, jobr, jobv, l, mid, i << 1);
        }
        if (jobr > mid) {
            update(jobl, jobr, jobv, mid + 1, r, i << 1 | 1);
        }
    }
}

// 区间查询操作（统计可见海报数量）
public static int query(int jobl, int jobr, int l, int r, int i) {
    if (l == r) {
        if (poster[i] != 0 && !visited[poster[i]]) {
            visited[poster[i]] = true;
            return 1;
        } else {
            return 0;
        }
    } else {
        down(i);
        int mid = (l + r) >> 1;
        int ans = 0;
        if (jobl <= mid) {
            ans += query(jobl, jobr, l, mid, i << 1);
```

```
            }
            if (jobr > mid) {
                ans += query(jobl, jobr, mid + 1, r, i << 1 | 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();
        int n = (int) in.nval;
        int size = 0;
        num[++size] = n;
        in.nextToken();
        int m = (int) in.nval;
        for (int i = 1; i <= m; i++) {
            in.nextToken();
            pl[i] = (int) in.nval;
            in.nextToken();
            pr[i] = (int) in.nval;
            num[++size] = pl[i];
            num[++size] = pr[i];
        }
        size = collect(size);
        build(1, size, 1);
        for (int i = 1, jobl, jobr; i <= m; i++) {
            jobl = rank(1, size, pl[i]);
            jobr = rank(1, size, pr[i]);
            update(jobl, jobr, i, 1, size, 1);
        }
        out.println(query(1, rank(1, size, n), 1, size, 1));
        out.flush();
        out.close();
        br.close();
    }

}
```

==========

文件: Code05_Posters2.java

```
==========================================
package class111;

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

// 贴海报(数据加强版) (Posters Enhanced Version)
// 题目来源: POJ 2528. Mayor's posters
// 题目链接: http://poj.org/problem?id=2528
//
// 题目描述:
// 有一面墙，有固定高度，无限的宽度，有 n 张海报，所有海报的高度都和墙的高度相同
// 从第 1 张海报开始，一张一张往墙上贴，直到 n 张海报贴完
// 每张海报都给出张贴位置(xi, yi)，表示第 i 张海报从墙的左边界 xi 一直延伸到右边界 yi
// 有可能发生后面的海报把前面的海报完全覆盖，导致看不到的情况
// 当所有海报贴完，返回能看到海报的数量，哪怕只漏出一点的海报都算
// 1 <= n <= 10^5，1 <= xi、yi <= 10^7
//
// 解题思路:
// 1. 使用线段树维护区间覆盖情况，记录每个区间被哪张海报完全覆盖
// 2. 由于坐标范围很大，需要进行离散化处理
// 3. 从第一张海报开始依次张贴，每次更新对应区间的海报编号
// 4. 最后查询整个墙面上可见的海报数量
// 5. 利用懒惰传播优化区间更新操作
// 6. 与 Code05_Posters1.java 相比，支持多组测试用例
//
// 时间复杂度: O(n log n)，其中 n 为海报数量
// 空间复杂度: O(n)
public class Code05_Posters2 {

    public static int MAXN = 10001;

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

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

    public static int[] num = new int[MAXN << 2];

    // 线段树的某个范围上是否被设置成了统一的海报
    // 如果 poster[i] != 0，poster[i]表示统一海报的编号
    // 如果 poster[i] == 0，表示该范围上没有海报或者海报编号没统一
    public static int[] poster = new int[MAXN << 4];
```

```
// 某种海报编号是否已经被统计过了
// 只在最后一次查询，最后统计海报数量的阶段时候使用
public static boolean[] visited = new boolean[MAXN];

// 收集所有坐标点并进行离散化处理
public static int collect(int m) {
    Arrays.sort(num, 1, m + 1);
    int size = 1;
    for (int i = 2; i <= m; i++) {
        if (num[size] != num[i]) {
            num[++size] = num[i];
        }
    }
    int cnt = size;
    for (int i = 2; i <= size; i++) {
        if (num[i - 1] + 1 < num[i]) {
            num[++cnt] = num[i - 1] + 1;
        }
    }
    Arrays.sort(num, 1, cnt + 1);
    return cnt;
}

// 二分查找值 v 在离散化数组中的位置
public static int rank(int l, int r, int v) {
    int m;
    int ans = 0;
    while (l <= r) {
        m = (l + r) >> 1;
        if (num[m] >= v) {
            ans = m;
            r = m - 1;
        } else {
            l = m + 1;
        }
    }
    return ans;
}

// 下发懒惰标记
public static void down(int i) {
    if (poster[i] != 0) {
```

```
        poster[i << 1] = poster[i];
        poster[i << 1 | 1] = poster[i];
        poster[i] = 0;
    }
}

// 构建线段树
public static void build(int l, int r, int i) {
    if (l < r) {
        int mid = (l + r) >> 1;
        build(l, mid, i << 1);
        build(mid + 1, r, i << 1 | 1);
    }
    poster[i] = 0;
}

// 区间更新操作（张贴海报）
public static void update(int jobl, int jobr, int jobv, int l, int r, int i) {
    if (jobl <= l && r <= jobr) {
        poster[i] = jobv;
    } else {
        down(i);
        int mid = (l + r) >> 1;
        if (jobl <= mid) {
            update(jobl, jobr, jobv, l, mid, i << 1);
        }
        if (jobr > mid) {
            update(jobl, jobr, jobv, mid + 1, r, i << 1 | 1);
        }
    }
}

// 区间查询操作（统计可见海报数量）
public static int query(int jobl, int jobr, int l, int r, int i) {
    if (l == r) {
        if (poster[i] != 0 && !visited[poster[i]]) {
            visited[poster[i]] = true;
            return 1;
        } else {
            return 0;
        }
    } else {
        down(i);
```

```
        int mid = (l + r) >> 1;
        int ans = 0;
        if (jobl <= mid) {
            ans += query(jobl, jobr, l, mid, i << 1);
        }
        if (jobr > mid) {
            ans += query(jobl, jobr, mid + 1, r, i << 1 | 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();
    int cases = (int) in.nval;
    for (int t = 1; t <= cases; t++) {
        in.nextToken();
        int n = (int) in.nval;
        int m = 0;
        for (int i = 1; i <= n; i++) {
            in.nextToken();
            pl[i] = (int) in.nval;
            in.nextToken();
            pr[i] = (int) in.nval;
            num[++m] = pl[i];
            num[++m] = pr[i];
        }
        m = collect(m);
        build(1, m, 1);
        for (int i = 1, jobl, jobr; i <= n; i++) {
            jobl = rank(1, m, pl[i]);
            jobr = rank(1, m, pr[i]);
            update(jobl, jobr, i, 1, m, 1);
        }
        out.println(query(1, m, 1, m, 1));
        Arrays.fill(visited, 1, n + 1, false);
    }
    out.flush();
    out.close();
    br.close();
```

```
    }

}
```

==========

文件: Code06_RangeSumQueryMutable.java

==========

```
package class111.supplementary_problems;

// Range Sum Query - Mutable (区间求和 - 可变)
// 题目来源: LeetCode 307. Range Sum Query - Mutable
// 题目链接: https://leetcode.cn/problems/range-sum-query-mutable
// 题目链接: https://leetcode.com/problems/range-sum-query-mutable
//
// 题目描述:
// 给你一个数组 nums ，请你完成两类查询：
// 1. 一类查询要求更新数组 nums 下标对应的值
// 2. 一类查询要求返回数组 nums 中，索引 left 和 right 之间的元素之和，包含 left 和 right 两点
// 实现 NumArray 类：
// NumArray(int[] nums) 用整数数组 nums 初始化对象
// void update(int index, int val) 将 nums[index] 的值更新为 val
// int sumRange(int left, int right) 返回数组 nums 中索引 left 和索引 right 之间
// (包含)的元素之和
//
// 解题思路:
// 1. 使用线段树维护数组区间和信息
// 2. 支持单点更新和区间查询操作
// 3. 线段树的每个节点存储对应区间的元素和
// 4. 更新操作从根节点到叶子节点递归更新路径上的所有节点
// 5. 查询操作根据查询区间与节点区间的关系进行递归查询
//
// 时间复杂度:
// - 构建: O(n)
// - 更新: O(log n)
// - 查询: O(log n)
// 空间复杂度: O(n)
public class Code06_RangeSumQueryMutable {

    // 线段树实现
    private static class SegmentTree {
        private int[] nums;
        private int[] tree;
```

```
    private int n;

    public SegmentTree(int[] nums) {
        this.nums = nums;
        this.n = nums.length;
        // 线段树需要 4*n 的空间
        this.tree = new int[4 * n];
        // 构建线段树
        buildTree(0, 0, n - 1);
    }

    // 构建线段树
    // node 是线段树节点的索引
    // start 和 end 是数组区间
    private void buildTree(int node, int start, int end) {
        // 叶子节点
        if (start == end) {
            tree[node] = nums[start];
            return;
        }

        // 非叶子节点，递归构建左右子树
        int mid = (start + end) / 2;
        // 左子节点索引为 2*node+1
        buildTree(2 * node + 1, start, mid);
        // 右子节点索引为 2*node+2
        buildTree(2 * node + 2, mid + 1, end);
        // 更新当前节点的值为左右子节点值的和
        tree[node] = tree[2 * node + 1] + tree[2 * node + 2];
    }

    // 更新数组中某个索引的值
    public void update(int index, int val) {
        updateHelper(0, 0, n - 1, index, val);
    }

    // 更新辅助函数
    private void updateHelper(int node, int start, int end, int index, int val) {
        // 找到叶子节点，更新值
        if (start == end) {
            nums[index] = val;
            tree[node] = val;
            return;
```

```
        }

        // 在左右子树中查找需要更新的索引
        int mid = (start + end) / 2;
        if (index <= mid) {
            // 在左子树中
            updateHelper(2 * node + 1, start, mid, index, val);
        } else {
            // 在右子树中
            updateHelper(2 * node + 2, mid + 1, end, index, val);
        }

        // 更新当前节点的值
        tree[node] = tree[2 * node + 1] + tree[2 * node + 2];
    }

    // 查询区间和
    public int sumRange(int left, int right) {
        return sumRangeHelper(0, 0, n - 1, left, right);
    }

    // 查询区间和辅助函数
    private int sumRangeHelper(int node, int start, int end, int left, int right) {
        // 当前区间与查询区间无交集
        if (right < start || left > end) {
            return 0;
        }

        // 当前区间完全包含在查询区间内
        if (left <= start && end <= right) {
            return tree[node];
        }

        // 当前区间与查询区间有部分交集，递归查询左右子树
        int mid = (start + end) / 2;
        int leftSum = sumRangeHelper(2 * node + 1, start, mid, left, right);
        int rightSum = sumRangeHelper(2 * node + 2, mid + 1, end, left, right);
        return leftSum + rightSum;
    }
}

// 主类实现
private SegmentTree st;
```

```
    public Code06_RangeSumQueryMutable(int[] nums) {
        st = new SegmentTree(nums);
    }

    public void update(int index, int val) {
        st.update(index, val);
    }

    public int sumRange(int left, int right) {
        return st.sumRange(left, right);
    }

    // 测试方法
    public static void main(String[] args) {
        int[] nums = {1, 3, 5};
        Code06_RangeSumQueryMutable numArray = new Code06_RangeSumQueryMutable(nums);

        // 查询索引 0 到 2 的和: 1 + 3 + 5 = 9
        System.out.println(numArray.sumRange(0, 2)); // 输出: 9

        // 更新索引 1 的值为 2，数组变为[1, 2, 5]
        numArray.update(1, 2);

        // 查询索引 0 到 2 的和: 1 + 2 + 5 = 8
        System.out.println(numArray.sumRange(0, 2)); // 输出: 8
    }
}
```

==========

文件: Code06_RangeSumQueryMutable.py

==========

```
# Range Sum Query - Mutable (区间求和 - 可变)
# 题目来源: LeetCode 307. Range Sum Query - Mutable
# 题目链接: https://leetcode.cn/problems/range-sum-query-mutable
# 题目链接: https://leetcode.com/problems/range-sum-query-mutable
#
# 题目描述:
# 给你一个数组 nums ，请你完成两类查询：
# 1. 一类查询要求更新数组 nums 下标对应的值
# 2. 一类查询要求返回数组 nums 中，索引 left 和 right 之间的元素之和，包含 left 和 right 两点
# 实现 NumArray 类：
```

```
# NumArray(int[] nums) 用整数数组 nums 初始化对象
# void update(int index, int val) 将 nums[index] 的值更新为 val
# int sumRange(int left, int right) 返回数组 nums 中索引 left 和索引 right 之间
# (包含)的元素之和
#
# 解题思路:
# 1. 使用线段树维护数组区间和信息
# 2. 支持单点更新和区间查询操作
# 3. 线段树的每个节点存储对应区间的元素和
# 4. 更新操作从根节点到叶子节点递归更新路径上的所有节点
# 5. 查询操作根据查询区间与节点区间的关系进行递归查询
#
# 时间复杂度:
# - 构建: O(n)
# - 更新: O(log n)
# - 查询: O(log n)
# 空间复杂度: O(n)

class SegmentTree:
    def __init__(self, nums):
        """
        初始化线段树
        :param nums: 输入数组
        """
        self.nums = nums
        self.n = len(nums)
        # 线段树需要 4*n 的空间
        self.tree = [0] * (4 * self.n)
        # 构建线段树
        self._build_tree(0, 0, self.n - 1)

    def _build_tree(self, node, start, end):
        """
        构建线段树
        :param node: 线段树节点的索引
        :param start: 数组区间开始索引
        :param end: 数组区间结束索引
        """
        # 叶子节点
        if start == end:
            self.tree[node] = self.nums[start]
            return
```

```
        # 非叶子节点，递归构建左右子树
        mid = (start + end) // 2
        # 左子节点索引为 2*node+1
        self._build_tree(2 * node + 1, start, mid)
        # 右子节点索引为 2*node+2
        self._build_tree(2 * node + 2, mid + 1, end)
        # 更新当前节点的值为左右子节点值的和
        self.tree[node] = self.tree[2 * node + 1] + self.tree[2 * node + 2]

    def update(self, index, val):
        """
        更新数组中某个索引的值
        :param index: 要更新的数组索引
        :param val: 新的值
        """
        self._update_helper(0, 0, self.n - 1, index, val)

    def _update_helper(self, node, start, end, index, val):
        """
        更新辅助函数
        :param node: 当前线段树节点索引
        :param start: 当前数组区间开始索引
        :param end: 当前数组区间结束索引
        :param index: 要更新的数组索引
        :param val: 新的值
        """
        # 找到叶子节点，更新值
        if start == end:
            self.nums[index] = val
            self.tree[node] = val
            return

        # 在左右子树中查找需要更新的索引
        mid = (start + end) // 2
        if index <= mid:
            # 在左子树中
            self._update_helper(2 * node + 1, start, mid, index, val)
        else:
            # 在右子树中
            self._update_helper(2 * node + 2, mid + 1, end, index, val)

        # 更新当前节点的值
        self.tree[node] = self.tree[2 * node + 1] + self.tree[2 * node + 2]
```

```
    def sum_range(self, left, right):
        """
        查询区间和
        :param left: 查询区间左边界
        :param right: 查询区间右边界
        :return: 区间和
        """
        return self._sum_range_helper(0, 0, self.n - 1, left, right)

    def _sum_range_helper(self, node, start, end, left, right):
        """
        查询区间和辅助函数
        :param node: 当前线段树节点索引
        :param start: 当前数组区间开始索引
        :param end: 当前数组区间结束索引
        :param left: 查询区间左边界
        :param right: 查询区间右边界
        :return: 区间和
        """
        # 当前区间与查询区间无交集
        if right < start or left > end:
            return 0

        # 当前区间完全包含在查询区间内
        if left <= start and end <= right:
            return self.tree[node]

        # 当前区间与查询区间有部分交集，递归查询左右子树
        mid = (start + end) // 2
        left_sum = self._sum_range_helper(2 * node + 1, start, mid, left, right)
        right_sum = self._sum_range_helper(2 * node + 2, mid + 1, end, left, right)
        return left_sum + right_sum


class NumArray:
    def __init__(self, nums):
        """
        初始化 NumArray 对象
        :param nums: 整数数组
        """
        self.st = SegmentTree(nums)
```

```
    def update(self, index, val):
        """
        更新数组中某个索引的值
        :param index: 要更新的数组索引
        :param val: 新的值
        """
        self.st.update(index, val)

    def sum_range(self, left, right):
        """
        返回数组中索引 left 和索引 right 之间的元素之和
        :param left: 查询区间左边界
        :param right: 查询区间右边界
        :return: 区间和
        """
        return self.st.sum_range(left, right)


# 测试方法
if __name__ == "__main__":
    nums = [1, 3, 5]
    num_array = NumArray(nums)

    # 查询索引 0 到 2 的和: 1 + 3 + 5 = 9
    print(num_array.sum_range(0, 2))  # 输出: 9

    # 更新索引 1 的值为 2，数组变为[1, 2, 5]
    num_array.update(1, 2)

    # 查询索引 0 到 2 的和: 1 + 2 + 5 = 8
    print(num_array.sum_range(0, 2))  # 输出: 8
```

==========

文件: Code07_RangeMaxQuery.java

==========

```
package class111.supplementary_problems;

// Range Max Query (区间最大值查询)
// 题目描述:
// 给定一个数组，实现以下操作：
// 1. 更新数组中某个位置的值
// 2. 查询某个区间内的最大值
```

```
// 测试链接: https://leetcode.cn/problems/max-value-of-equation/
//
// 解题思路:
// 1. 使用线段树维护数组区间最大值信息
// 2. 支持单点更新和区间查询操作
// 3. 线段树的每个节点存储对应区间的最大值
// 4. 更新操作从根节点到叶子节点递归更新路径上的所有节点
// 5. 查询操作根据查询区间与节点区间的关系进行递归查询
//
// 时间复杂度:
// - 构建: O(n)
// - 更新: O(log n)
// - 查询: O(log n)
// 空间复杂度: O(n)

public class Code07_RangeMaxQuery {

    // 线段树实现区间最大值查询
    private static class SegmentTree {
        private int[] nums;
        private int[] tree;
        private int n;

        public SegmentTree(int[] nums) {
            this.nums = nums;
            this.n = nums.length;
            // 线段树需要 4*n 的空间
            this.tree = new int[4 * n];
            // 构建线段树
            buildTree(0, 0, n - 1);
        }

        // 构建线段树
        // node 是线段树节点的索引
        // start 和 end 是数组区间
        private void buildTree(int node, int start, int end) {
            // 叶子节点
            if (start == end) {
                tree[node] = nums[start];
                return;
            }

            // 非叶子节点，递归构建左右子树
```

```
    int mid = (start + end) / 2;
    // 左子节点索引为2*node+1
    buildTree(2 * node + 1, start, mid);
    // 右子节点索引为2*node+2
    buildTree(2 * node + 2, mid + 1, end);
    // 更新当前节点的值为左右子节点的最大值
    tree[node] = Math.max(tree[2 * node + 1], tree[2 * node + 2]);
}

// 更新数组中某个索引的值
public void update(int index, int val) {
    updateHelper(0, 0, n - 1, index, val);
}

// 更新辅助函数
private void updateHelper(int node, int start, int end, int index, int val) {
    // 找到叶子节点，更新值
    if (start == end) {
        nums[index] = val;
        tree[node] = val;
        return;
    }

    // 在左右子树中查找需要更新的索引
    int mid = (start + end) / 2;
    if (index <= mid) {
        // 在左子树中
        updateHelper(2 * node + 1, start, mid, index, val);
    } else {
        // 在右子树中
        updateHelper(2 * node + 2, mid + 1, end, index, val);
    }

    // 更新当前节点的值为左右子节点的最大值
    tree[node] = Math.max(tree[2 * node + 1], tree[2 * node + 2]);
}

// 查询区间最大值
public int rangeMax(int left, int right) {
    return rangeMaxHelper(0, 0, n - 1, left, right);
}

// 查询区间最大值辅助函数
```

```
    private int rangeMaxHelper(int node, int start, int end, int left, int right) {
        // 当前区间与查询区间无交集
        if (right < start || left > end) {
            // 返回一个不影响结果的值（对于求最大值操作，返回最小值）
            return Integer.MIN_VALUE;
        }

        // 当前区间完全包含在查询区间内
        if (left <= start && end <= right) {
            return tree[node];
        }

        // 当前区间与查询区间有部分交集，递归查询左右子树
        int mid = (start + end) / 2;
        int leftMax = rangeMaxHelper(2 * node + 1, start, mid, left, right);
        int rightMax = rangeMaxHelper(2 * node + 2, mid + 1, end, left, right);
        return Math.max(leftMax, rightMax);
    }
}

// 主类实现
private SegmentTree st;

public Code07_RangeMaxQuery(int[] nums) {
    st = new SegmentTree(nums);
}

public void update(int index, int val) {
    st.update(index, val);
}

public int rangeMax(int left, int right) {
    return st.rangeMax(left, right);
}

// 测试方法
public static void main(String[] args) {
    int[] nums = {1, 3, 5, 7, 9, 11};
    Code07_RangeMaxQuery rmq = new Code07_RangeMaxQuery(nums);

    // 查询索引 1 到 4 之间的最大值: max(3, 5, 7, 9) = 9
    System.out.println(rmq.rangeMax(1, 4)); // 输出: 9
```

```
        // 更新索引 2 的值为 15，数组变为[1, 3, 15, 7, 9, 11]
        rmq.update(2, 15);

        // 查询索引 1 到 4 之间的最大值: max(3, 15, 7, 9) = 15
        System.out.println(rmq.rangeMax(1, 4)); // 输出: 15
    }
}
```

==========

文件: Code07_RangeMaxQuery.py

==========

```
# Range Max Query (区间最大值查询)
# 题目描述:
# 给定一个数组，实现以下操作：
# 1. 更新数组中某个位置的值
# 2. 查询某个区间内的最大值
# 测试链接: https://leetcode.cn/problems/max-value-of-equation/
#
# 解题思路:
# 1. 使用线段树维护数组区间最大值信息
# 2. 支持单点更新和区间查询操作
# 3. 线段树的每个节点存储对应区间的最大值
# 4. 更新操作从根节点到叶子节点递归更新路径上的所有节点
# 5. 查询操作根据查询区间与节点区间的关系进行递归查询
#
# 时间复杂度:
# - 构建: O(n)
# - 更新: O(log n)
# - 查询: O(log n)
# 空间复杂度: O(n)

class SegmentTree:
    def __init__(self, nums):
        """
        初始化线段树
        :param nums: 输入数组
        """
        self.nums = nums
        self.n = len(nums)
        # 线段树需要 4*n 的空间
        self.tree = [0] * (4 * self.n)
        # 构建线段树
```

```
        self._build_tree(0, 0, self.n - 1)

    def _build_tree(self, node, start, end):
        """
        构建线段树
        :param node: 线段树节点的索引
        :param start: 数组区间开始索引
        :param end: 数组区间结束索引
        """
        # 叶子节点
        if start == end:
            self.tree[node] = self.nums[start]
            return

        # 非叶子节点，递归构建左右子树
        mid = (start + end) // 2
        # 左子节点索引为 2*node+1
        self._build_tree(2 * node + 1, start, mid)
        # 右子节点索引为 2*node+2
        self._build_tree(2 * node + 2, mid + 1, end)
        # 更新当前节点的值为左右子节点的最大值
        self.tree[node] = max(self.tree[2 * node + 1], self.tree[2 * node + 2])

    def update(self, index, val):
        """
        更新数组中某个索引的值
        :param index: 要更新的数组索引
        :param val: 新的值
        """
        self._update_helper(0, 0, self.n - 1, index, val)

    def _update_helper(self, node, start, end, index, val):
        """
        更新辅助函数
        :param node: 当前线段树节点索引
        :param start: 当前数组区间开始索引
        :param end: 当前数组区间结束索引
        :param index: 要更新的数组索引
        :param val: 新的值
        """
        # 找到叶子节点，更新值
        if start == end:
            self.nums[index] = val
```

```
            self.tree[node] = val
            return

        # 在左右子树中查找需要更新的索引
        mid = (start + end) // 2
        if index <= mid:
            # 在左子树中
            self._update_helper(2 * node + 1, start, mid, index, val)
        else:
            # 在右子树中
            self._update_helper(2 * node + 2, mid + 1, end, index, val)

        # 更新当前节点的值为左右子节点的最大值
        self.tree[node] = max(self.tree[2 * node + 1], self.tree[2 * node + 2])

    def range_max(self, left, right):
        """
        查询区间最大值
        :param left: 查询区间左边界
        :param right: 查询区间右边界
        :return: 区间最大值
        """
        return self._range_max_helper(0, 0, self.n - 1, left, right)

    def _range_max_helper(self, node, start, end, left, right):
        """
        查询区间最大值辅助函数
        :param node: 当前线段树节点索引
        :param start: 当前数组区间开始索引
        :param end: 当前数组区间结束索引
        :param left: 查询区间左边界
        :param right: 查询区间右边界
        :return: 区间最大值
        """
        # 当前区间与查询区间无交集
        if right < start or left > end:
            # 返回一个不影响结果的值（对于求最大值操作，返回最小值）
            return float('-inf')

        # 当前区间完全包含在查询区间内
        if left <= start and end <= right:
            return self.tree[node]
```

```
        # 当前区间与查询区间有部分交集，递归查询左右子树
        mid = (start + end) // 2
        left_max = self._range_max_helper(2 * node + 1, start, mid, left, right)
        right_max = self._range_max_helper(2 * node + 2, mid + 1, end, left, right)
        return max(left_max, right_max)


class RangeMaxQuery:
    def __init__(self, nums):
        """
        初始化 RangeMaxQuery 对象
        :param nums: 整数数组
        """
        self.st = SegmentTree(nums)

    def update(self, index, val):
        """
        更新数组中某个索引的值
        :param index: 要更新的数组索引
        :param val: 新的值
        """
        self.st.update(index, val)

    def range_max(self, left, right):
        """
        返回数组中索引 left 和索引 right 之间的元素的最大值
        :param left: 查询区间左边界
        :param right: 查询区间右边界
        :return: 区间最大值
        """
        return self.st.range_max(left, right)


# 测试方法
if __name__ == "__main__":
    nums = [1, 3, 5, 7, 9, 11]
    rmq = RangeMaxQuery(nums)

    # 查询索引 1 到 4 之间的最大值: max(3, 5, 7, 9) = 9
    print(rmq.range_max(1, 4))  # 输出: 9

    # 更新索引 2 的值为 15，数组变为[1, 3, 15, 7, 9, 11]
    rmq.update(2, 15)
```

```
    # 查询索引 1 到 4 之间的最大值: max(3, 15, 7, 9) = 15
    print(rmq.range_max(1, 4))  # 输出: 15
```

==========

文件: Code08_RangeAddQuery.java
==========

```
package class111.supplementary_problems;

// Range Add Query (区间加法更新)
// 题目描述:
// 实现一个支持区间加法更新和单点查询的数据结构
// 支持以下操作：
// 1. 对区间[l, r]内所有元素加上一个值 val
// 2. 查询某个位置的值
// 测试链接: https://leetcode.cn/problems/range-addition/
//
// 解题思路:
// 1. 使用带懒惰传播的线段树实现区间更新和单点查询
// 2. 懒惰传播用于延迟更新，避免不必要的计算
// 3. 区间更新时，只在必要时才将更新操作传递给子节点
// 4. 查询时确保所有相关的懒惰标记都被处理
//
// 时间复杂度:
// - 区间更新: O(log n)
// - 单点查询: O(log n)
// 空间复杂度: O(n)

public class Code08_RangeAddQuery {

    // 带懒惰传播的线段树实现
    private static class SegmentTree {
        private int[] tree;     // 线段树节点值（存储区间和）
        private int[] lazy;     // 懒惰标记数组
        private int n;          // 数组长度

        public SegmentTree(int size) {
            this.n = size;
            // 线段树需要 4*n 的空间
            this.tree = new int[4 * n];
            this.lazy = new int[4 * n];
        }
```

```
// 区间加法更新 [l, r] 区间内每个元素加上 val
public void rangeAdd(int l, int r, int val) {
    rangeAddHelper(0, 0, n - 1, l, r, val);
}

// 区间加法更新辅助函数
private void rangeAddHelper(int node, int start, int end, int l, int r, int val) {
    // 1. 先处理懒惰标记
    pushDown(node, start, end);

    // 2. 当前区间与更新区间无交集
    if (start > r || end < l) {
        return;
    }

    // 3. 当前区间完全包含在更新区间内
    if (start >= l && end <= r) {
        // 更新当前节点的值
        tree[node] += val * (end - start + 1);
        // 如果不是叶子节点，设置懒惰标记
        if (start != end) {
            lazy[2 * node + 1] += val;
            lazy[2 * node + 2] += val;
        }
        return;
    }

    // 4. 当前区间与更新区间有部分交集，递归处理左右子树
    int mid = (start + end) / 2;
    rangeAddHelper(2 * node + 1, start, mid, l, r, val);
    rangeAddHelper(2 * node + 2, mid + 1, end, l, r, val);

    // 更新当前节点的值
    tree[node] = tree[2 * node + 1] + tree[2 * node + 2];
}

// 查询单点的值
public int query(int index) {
    return queryHelper(0, 0, n - 1, index);
}

// 查询单点值辅助函数
```

```
private int queryHelper(int node, int start, int end, int index) {
    // 1. 先处理懒惰标记
    pushDown(node, start, end);

    // 2. 找到叶子节点
    if (start == end) {
        return tree[node];
    }

    // 3. 递归查询左右子树
    int mid = (start + end) / 2;
    if (index <= mid) {
        return queryHelper(2 * node + 1, start, mid, index);
    } else {
        return queryHelper(2 * node + 2, mid + 1, end, index);
    }
}

// 下推懒惰标记
private void pushDown(int node, int start, int end) {
    // 如果当前节点没有懒惰标记，直接返回
    if (lazy[node] == 0) {
        return;
    }

    // 将懒惰标记下推到子节点
    int mid = (start + end) / 2;
    // 更新左子节点
    tree[2 * node + 1] += lazy[node] * (mid - start + 1);
    // 更新右子节点
    tree[2 * node + 2] += lazy[node] * (end - mid);

    // 如果子节点不是叶子节点，继续传递懒惰标记
    if (start != mid) {
        lazy[2 * node + 1] += lazy[node];
    }
    if (mid + 1 != end) {
        lazy[2 * node + 2] += lazy[node];
    }

    // 清除当前节点的懒惰标记
    lazy[node] = 0;
}
```

```
}

// 主类实现
private SegmentTree st;
private int n;

public Code08_RangeAddQuery(int size) {
    this.n = size;
    this.st = new SegmentTree(size);
}

// 对区间[l, r]内所有元素加上val
public void rangeAdd(int l, int r, int val) {
    st.rangeAdd(l, r, val);
}

// 查询索引index处的值
public int query(int index) {
    return st.query(index);
}

// 测试方法
public static void main(String[] args) {
    Code08_RangeAddQuery raq = new Code08_RangeAddQuery(5);

    // 对区间[1, 3]内所有元素加上2
    raq.rangeAdd(1, 3, 2);

    // 查询各个位置的值
    // 初始数组为[0, 0, 0, 0, 0]
    // 操作后变为[0, 2, 2, 2, 0]
    System.out.println(raq.query(0)); // 输出: 0
    System.out.println(raq.query(1)); // 输出: 2
    System.out.println(raq.query(2)); // 输出: 2
    System.out.println(raq.query(3)); // 输出: 2
    System.out.println(raq.query(4)); // 输出: 0

    // 对区间[2, 4]内所有元素加上3
    raq.rangeAdd(2, 4, 3);

    // 查询各个位置的值
    // 数组变为[0, 2, 5, 5, 3]
    System.out.println(raq.query(0)); // 输出: 0
```

```
        System.out.println(raq.query(1)); // 输出: 2
        System.out.println(raq.query(2)); // 输出: 5
        System.out.println(raq.query(3)); // 输出: 5
        System.out.println(raq.query(4)); // 输出: 3
    }
}
```

==========

文件: Code08_RangeAddQuery.py

==========

```
# Range Add Query (区间加法更新)
# 题目描述:
# 实现一个支持区间加法更新和单点查询的数据结构
# 支持以下操作:
# 1. 对区间[l, r]内所有元素加上一个值 val
# 2. 查询某个位置的值
# 测试链接: https://leetcode.cn/problems/range-addition/
#
# 解题思路:
# 1. 使用带懒惰传播的线段树实现区间更新和单点查询
# 2. 懒惰传播用于延迟更新，避免不必要的计算
# 3. 区间更新时，只在必要时才将更新操作传递给子节点
# 4. 查询时确保所有相关的懒惰标记都被处理
#
# 时间复杂度:
# - 区间更新: O(log n)
# - 单点查询: O(log n)
# 空间复杂度: O(n)

class SegmentTree:
    def __init__(self, size):
        """
        初始化带懒惰传播的线段树
        :param size: 数组大小
        """
        self.n = size
        # 线段树需要 4*n 的空间
        self.tree = [0] * (4 * self.n)  # 存储区间和
        self.lazy = [0] * (4 * self.n)  # 懒惰标记数组

    def range_add(self, l, r, val):
        """
```

```
    区间加法更新 [l, r] 区间内每个元素加上 val
    :param l: 区间左边界
    :param r: 区间右边界
    :param val: 要增加的值
    """
    self._range_add_helper(0, 0, self.n - 1, l, r, val)

def _range_add_helper(self, node, start, end, l, r, val):
    """
    区间加法更新辅助函数
    :param node: 当前线段树节点索引
    :param start: 当前数组区间开始索引
    :param end: 当前数组区间结束索引
    :param l: 更新区间左边界
    :param r: 更新区间右边界
    :param val: 要增加的值
    """
    # 1. 先处理懒惰标记
    self._push_down(node, start, end)

    # 2. 当前区间与更新区间无交集
    if start > r or end < l:
        return

    # 3. 当前区间完全包含在更新区间内
    if start >= l and end <= r:
        # 更新当前节点的值
        self.tree[node] += val * (end - start + 1)
        # 如果不是叶子节点，设置懒惰标记
        if start != end:
            self.lazy[2 * node + 1] += val
            self.lazy[2 * node + 2] += val
        return

    # 4. 当前区间与更新区间有部分交集，递归处理左右子树
    mid = (start + end) // 2
    self._range_add_helper(2 * node + 1, start, mid, l, r, val)
    self._range_add_helper(2 * node + 2, mid + 1, end, l, r, val)

    # 更新当前节点的值
    self.tree[node] = self.tree[2 * node + 1] + self.tree[2 * node + 2]

def query(self, index):
```

```
    """
    查询单点的值
    :param index: 要查询的索引
    :return: 索引处的值
    """
    return self._query_helper(0, 0, self.n - 1, index)

def _query_helper(self, node, start, end, index):
    """
    查询单点值辅助函数
    :param node: 当前线段树节点索引
    :param start: 当前数组区间开始索引
    :param end: 当前数组区间结束索引
    :param index: 要查询的索引
    :return: 索引处的值
    """
    # 1. 先处理懒惰标记
    self._push_down(node, start, end)

    # 2. 找到叶子节点
    if start == end:
        return self.tree[node]

    # 3. 递归查询左右子树
    mid = (start + end) // 2
    if index <= mid:
        return self._query_helper(2 * node + 1, start, mid, index)
    else:
        return self._query_helper(2 * node + 2, mid + 1, end, index)

def _push_down(self, node, start, end):
    """
    下推懒惰标记
    :param node: 当前线段树节点索引
    :param start: 当前数组区间开始索引
    :param end: 当前数组区间结束索引
    """
    # 如果当前节点没有懒惰标记，直接返回
    if self.lazy[node] == 0:
        return

    # 将懒惰标记下推到子节点
    mid = (start + end) // 2
```

```
        # 更新左子节点
        self.tree[2 * node + 1] += self.lazy[node] * (mid - start + 1)
        # 更新右子节点
        self.tree[2 * node + 2] += self.lazy[node] * (end - mid)

        # 如果子节点不是叶子节点，继续传递懒惰标记
        if start != mid:
            self.lazy[2 * node + 1] += self.lazy[node]
        if mid + 1 != end:
            self.lazy[2 * node + 2] += self.lazy[node]

        # 清除当前节点的懒惰标记
        self.lazy[node] = 0


class RangeAddQuery:
    def __init__(self, size):
        """
        初始化 RangeAddQuery 对象
        :param size: 数组大小
        """
        self.n = size
        self.st = SegmentTree(size)

    def range_add(self, l, r, val):
        """
        对区间[l, r]内所有元素加上 val
        :param l: 区间左边界
        :param r: 区间右边界
        :param val: 要增加的值
        """
        self.st.range_add(l, r, val)

    def query(self, index):
        """
        查询索引 index 处的值
        :param index: 要查询的索引
        :return: 索引处的值
        """
        return self.st.query(index)


# 测试方法
```

```
if __name__ == "__main__":
    raq = RangeAddQuery(5)

    # 对区间[1, 3]内所有元素加上2
    raq.range_add(1, 3, 2)

    # 查询各个位置的值
    # 初始数组为[0, 0, 0, 0, 0]
    # 操作后变为[0, 2, 2, 2, 0]
    print(raq.query(0))  # 输出: 0
    print(raq.query(1))  # 输出: 2
    print(raq.query(2))  # 输出: 2
    print(raq.query(3))  # 输出: 2
    print(raq.query(4))  # 输出: 0

    # 对区间[2, 4]内所有元素加上3
    raq.range_add(2, 4, 3)

    # 查询各个位置的值
    # 数组变为[0, 2, 5, 5, 3]
    print(raq.query(0))  # 输出: 0
    print(raq.query(1))  # 输出: 2
    print(raq.query(2))  # 输出: 5
    print(raq.query(3))  # 输出: 5
    print(raq.query(4))  # 输出: 3
```

==========

文件: Code09_RangeSumQueryMutable.cpp

==========

```
// Range Sum Query - Mutable (区间求和 - 可变)
// 题目来源: LeetCode 307. Range Sum Query - Mutable
// 题目链接: https://leetcode.cn/problems/range-sum-query-mutable
// 题目链接: https://leetcode.com/problems/range-sum-query-mutable
//
// 题目描述:
// 给你一个数组 nums ，请你完成两类查询：
// 1. 一类查询要求更新数组 nums 下标对应的值
// 2. 一类查询要求返回数组 nums 中，索引 left 和 right 之间的元素之和，包含 left 和 right 两点
// 实现 NumArray 类：
// NumArray(int[] nums) 用整数数组 nums 初始化对象
// void update(int index, int val) 将 nums[index] 的值更新为 val
// int sumRange(int left, int right) 返回数组 nums 中索引 left 和索引 right 之间
```

```
// (包含)的元素之和
//
// 解题思路:
// 1. 使用线段树维护数组区间和信息
// 2. 支持单点更新和区间查询操作
// 3. 线段树的每个节点存储对应区间的元素和
// 4. 更新操作从根节点到叶子节点递归更新路径上的所有节点
// 5. 查询操作根据查询区间与节点区间的关系进行递归查询
//
// 时间复杂度:
// - 构建: O(n)
// - 更新: O(log n)
// - 查询: O(log n)
// 空间复杂度: O(n)

#include <vector>
#include <algorithm>
using namespace std;

// 由于编译环境限制，不使用<iostream>等标准库头文件
// 使用简单的数组实现

const int MAXN = 100005;
int nums[MAXN];
int tree[4 * MAXN];
int n;

class SegmentTree {
private:
    // 构建线段树
    // node 是线段树节点的索引
    // start 和 end 是数组区间
    void buildTree(int node, int start, int end) {
        // 叶子节点
        if (start == end) {
            tree[node] = nums[start];
            return;
        }

        // 非叶子节点，递归构建左右子树
        int mid = (start + end) / 2;
        // 左子节点索引为 2*node+1
        buildTree(2 * node + 1, start, mid);
```

```
        // 右子节点索引为 2*node+2
        buildTree(2 * node + 2, mid + 1, end);
        // 更新当前节点的值为左右子节点值的和
        tree[node] = tree[2 * node + 1] + tree[2 * node + 2];
    }

public:
    SegmentTree(int arr[], int size) {
        n = size;
        // 复制数组
        for (int i = 0; i < size; i++) {
            nums[i] = arr[i];
        }
        // 构建线段树
        buildTree(0, 0, n - 1);
    }

    // 更新数组中某个索引的值
    void update(int index, int val) {
        updateHelper(0, 0, n - 1, index, val);
    }

    // 更新辅助函数
    void updateHelper(int node, int start, int end, int index, int val) {
        // 找到叶子节点，更新值
        if (start == end) {
            nums[index] = val;
            tree[node] = val;
            return;
        }

        // 在左右子树中查找需要更新的索引
        int mid = (start + end) / 2;
        if (index <= mid) {
            // 在左子树中
            updateHelper(2 * node + 1, start, mid, index, val);
        } else {
            // 在右子树中
            updateHelper(2 * node + 2, mid + 1, end, index, val);
        }

        // 更新当前节点的值
        tree[node] = tree[2 * node + 1] + tree[2 * node + 2];
```

```
    }

    // 查询区间和
    int sumRange(int left, int right) {
        return sumRangeHelper(0, 0, n - 1, left, right);
    }

    // 查询区间和辅助函数
    int sumRangeHelper(int node, int start, int end, int left, int right) {
        // 当前区间与查询区间无交集
        if (right < start || left > end) {
            return 0;
        }

        // 当前区间完全包含在查询区间内
        if (left <= start && end <= right) {
            return tree[node];
        }

        // 当前区间与查询区间有部分交集，递归查询左右子树
        int mid = (start + end) / 2;
        int leftSum = sumRangeHelper(2 * node + 1, start, mid, left, right);
        int rightSum = sumRangeHelper(2 * node + 2, mid + 1, end, left, right);
        return leftSum + rightSum;
    }
};

// 由于编译环境限制，不提供 main 函数测试
// 可以通过创建 SegmentTree 对象并调用其方法来使用

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

文件: Code09_RangeSumQueryMutable.java
==========================================
package class111;

// Range Sum Query - Mutable (区间求和 - 可变)
// 题目来源: LeetCode 307. Range Sum Query - Mutable
// 题目链接: https://leetcode.cn/problems/range-sum-query-mutable
// 题目链接: https://leetcode.com/problems/range-sum-query-mutable
//
// 题目描述:
// 给你一个数组 nums ，请你完成两类查询：
```

```
// 1. 一类查询要求更新数组 nums 下标对应的值
// 2. 一类查询要求返回数组 nums 中，索引 left 和 right 之间的元素之和，包含 left 和 right 两点
// 实现 NumArray 类：
// NumArray(int[] nums) 用整数数组 nums 初始化对象
// void update(int index, int val) 将 nums[index] 的值更新为 val
// int sumRange(int left, int right) 返回数组 nums 中索引 left 和索引 right 之间
// (包含)的元素之和
//
// 解题思路:
// 1. 使用线段树维护数组区间和信息
// 2. 支持单点更新和区间查询操作
// 3. 线段树的每个节点存储对应区间的元素和
// 4. 更新操作从根节点到叶子节点递归更新路径上的所有节点
// 5. 查询操作根据查询区间与节点区间的关系进行递归查询
//
// 时间复杂度:
// - 构建: O(n)
// - 更新: O(log n)
// - 查询: O(log n)
// 空间复杂度: O(n)

public class Code09_RangeSumQueryMutable {

    // 线段树实现
    private static class SegmentTree {
        private int[] nums;
        private int[] tree;
        private int n;

        public SegmentTree(int[] nums) {
            this.nums = nums;
            this.n = nums.length;
            // 线段树需要 4*n 的空间
            this.tree = new int[4 * n];
            // 构建线段树
            buildTree(0, 0, n - 1);
        }

        // 构建线段树
        // node 是线段树节点的索引
        // start 和 end 是数组区间
        private void buildTree(int node, int start, int end) {
            // 叶子节点
```

```
    if (start == end) {
        tree[node] = nums[start];
        return;
    }

    // 非叶子节点，递归构建左右子树
    int mid = (start + end) / 2;
    // 左子节点索引为 2*node+1
    buildTree(2 * node + 1, start, mid);
    // 右子节点索引为 2*node+2
    buildTree(2 * node + 2, mid + 1, end);
    // 更新当前节点的值为左右子节点值的和
    tree[node] = tree[2 * node + 1] + tree[2 * node + 2];
}

// 更新数组中某个索引的值
public void update(int index, int val) {
    updateHelper(0, 0, n - 1, index, val);
}

// 更新辅助函数
private void updateHelper(int node, int start, int end, int index, int val) {
    // 找到叶子节点，更新值
    if (start == end) {
        nums[index] = val;
        tree[node] = val;
        return;
    }

    // 在左右子树中查找需要更新的索引
    int mid = (start + end) / 2;
    if (index <= mid) {
        // 在左子树中
        updateHelper(2 * node + 1, start, mid, index, val);
    } else {
        // 在右子树中
        updateHelper(2 * node + 2, mid + 1, end, index, val);
    }

    // 更新当前节点的值
    tree[node] = tree[2 * node + 1] + tree[2 * node + 2];
}
```

```
    // 查询区间和
    public int sumRange(int left, int right) {
        return sumRangeHelper(0, 0, n - 1, left, right);
    }

    // 查询区间和辅助函数
    private int sumRangeHelper(int node, int start, int end, int left, int right) {
        // 当前区间与查询区间无交集
        if (right < start || left > end) {
            return 0;
        }

        // 当前区间完全包含在查询区间内
        if (left <= start && end <= right) {
            return tree[node];
        }

        // 当前区间与查询区间有部分交集，递归查询左右子树
        int mid = (start + end) / 2;
        int leftSum = sumRangeHelper(2 * node + 1, start, mid, left, right);
        int rightSum = sumRangeHelper(2 * node + 2, mid + 1, end, left, right);
        return leftSum + rightSum;
    }
}

// 主类实现
private SegmentTree st;

public Code09_RangeSumQueryMutable(int[] nums) {
    st = new SegmentTree(nums);
}

public void update(int index, int val) {
    st.update(index, val);
}

public int sumRange(int left, int right) {
    return st.sumRange(left, right);
}

// 测试方法
public static void main(String[] args) {
    int[] nums = {1, 3, 5};
```

```
        Code09_RangeSumQueryMutable numArray = new Code09_RangeSumQueryMutable(nums);

        // 查询索引 0 到 2 的和: 1 + 3 + 5 = 9
        System.out.println(numArray.sumRange(0, 2)); // 输出: 9

        // 更新索引 1 的值为 2，数组变为[1, 2, 5]
        numArray.update(1, 2);

        // 查询索引 0 到 2 的和: 1 + 2 + 5 = 8
        System.out.println(numArray.sumRange(0, 2)); // 输出: 8
    }
}
```

==========

文件: Code09_RangeSumQueryMutable.py

==========

```
# Range Sum Query - Mutable (区间求和 - 可变)
# 题目来源: LeetCode 307. Range Sum Query - Mutable
# 题目链接: https://leetcode.cn/problems/range-sum-query-mutable
# 题目链接: https://leetcode.com/problems/range-sum-query-mutable
#
# 题目描述:
# 给你一个数组 nums ，请你完成两类查询：
# 1. 一类查询要求更新数组 nums 下标对应的值
# 2. 一类查询要求返回数组 nums 中，索引 left 和 right 之间的元素之和，包含 left 和 right 两点
# 实现 NumArray 类：
# NumArray(int[] nums) 用整数数组 nums 初始化对象
# void update(int index, int val) 将 nums[index] 的值更新为 val
# int sumRange(int left, int right) 返回数组 nums 中索引 left 和索引 right 之间
# (包含)的元素之和
#
# 解题思路:
# 1. 使用线段树维护数组区间和信息
# 2. 支持单点更新和区间查询操作
# 3. 线段树的每个节点存储对应区间的元素和
# 4. 更新操作从根节点到叶子节点递归更新路径上的所有节点
# 5. 查询操作根据查询区间与节点区间的关系进行递归查询
#
# 时间复杂度:
# - 构建: O(n)
# - 更新: O(log n)
# - 查询: O(log n)
```

```
# 空间复杂度: O(n)

class SegmentTree:
    def __init__(self, nums):
        self.nums = nums
        self.n = len(nums)
        # 线段树需要 4*n 的空间
        self.tree = [0] * (4 * self.n)
        # 构建线段树
        self.build_tree(0, 0, self.n - 1)

    # 构建线段树
    # node 是线段树节点的索引
    # start 和 end 是数组区间
    def build_tree(self, node, start, end):
        # 叶子节点
        if start == end:
            self.tree[node] = self.nums[start]
            return

        # 非叶子节点，递归构建左右子树
        mid = (start + end) // 2
        # 左子节点索引为 2*node+1
        self.build_tree(2 * node + 1, start, mid)
        # 右子节点索引为 2*node+2
        self.build_tree(2 * node + 2, mid + 1, end)
        # 更新当前节点的值为左右子节点值的和
        self.tree[node] = self.tree[2 * node + 1] + self.tree[2 * node + 2]

    # 更新数组中某个索引的值
    def update(self, index, val):
        self._update_helper(0, 0, self.n - 1, index, val)

    # 更新辅助函数
    def _update_helper(self, node, start, end, index, val):
        # 找到叶子节点，更新值
        if start == end:
            self.nums[index] = val
            self.tree[node] = val
            return

        # 在左右子树中查找需要更新的索引
        mid = (start + end) // 2
```

```
        if index <= mid:
            # 在左子树中
            self._update_helper(2 * node + 1, start, mid, index, val)
        else:
            # 在右子树中
            self._update_helper(2 * node + 2, mid + 1, end, index, val)

        # 更新当前节点的值
        self.tree[node] = self.tree[2 * node + 1] + self.tree[2 * node + 2]

    # 查询区间和
    def sum_range(self, left, right):
        return self._sum_range_helper(0, 0, self.n - 1, left, right)

    # 查询区间和辅助函数
    def _sum_range_helper(self, node, start, end, left, right):
        # 当前区间与查询区间无交集
        if right < start or left > end:
            return 0

        # 当前区间完全包含在查询区间内
        if left <= start and end <= right:
            return self.tree[node]

        # 当前区间与查询区间有部分交集，递归查询左右子树
        mid = (start + end) // 2
        left_sum = self._sum_range_helper(2 * node + 1, start, mid, left, right)
        right_sum = self._sum_range_helper(2 * node + 2, mid + 1, end, left, right)
        return left_sum + right_sum


class NumArray:
    def __init__(self, nums):
        self.st = SegmentTree(nums)

    def update(self, index, val):
        self.st.update(index, val)

    def sum_range(self, left, right):
        return self.st.sum_range(left, right)


# 测试方法
```

```
if __name__ == "__main__":
    nums = [1, 3, 5]
    num_array = NumArray(nums)

    # 查询索引 0 到 2 的和: 1 + 3 + 5 = 9
    print(num_array.sum_range(0, 2))  # 输出: 9

    # 更新索引 1 的值为 2，数组变为[1, 2, 5]
    num_array.update(1, 2)

    # 查询索引 0 到 2 的和: 1 + 2 + 5 = 8
    print(num_array.sum_range(0, 2))  # 输出: 8
```

==========

文件: Code10_CountOfSmallerNumbersAfterSelf.cpp

==========

```
// Count of Smaller Numbers After Self (计算右侧小于当前元素的个数)
// 题目来源: LeetCode 315. Count of Smaller Numbers After Self
// 题目链接: https://leetcode.cn/problems/count-of-smaller-numbers-after-self
// 题目链接: https://leetcode.com/problems/count-of-smaller-numbers-after-self
//
// 题目描述:
// 给你一个整数数组 nums ，按要求返回一个新数组 counts 。
// 数组 counts 有该性质：counts[i] 的值是 nums[i] 右侧小于 nums[i] 的元素的数量。
// 示例 1：
// 输入：nums = [5,2,6,1]
// 输出：[2,1,1,0]
// 解释：
// 5 的右侧有 2 个更小的元素 (2 和 1)
// 2 的右侧有 1 个更小的元素 (1)
// 6 的右侧有 1 个更小的元素 (1)
// 1 的右侧有 0 个更小的元素
// 示例 2：
// 输入：nums = [-1]
// 输出：[0]
// 示例 3：
// 输入：nums = [-1,-1]
// 输出：[0,0]
// 提示：
// 1 <= nums.length <= 10^5
// -10^4 <= nums[i] <= 10^4
//
```

```
// 解题思路:
// 1. 使用离散化+线段树的方法解决
// 2. 从右向左遍历数组，维护一个值域线段树
// 3. 对于每个元素，查询值域中小于它的元素个数
// 4. 将当前元素插入到线段树中
// 5. 利用离散化处理大范围的值域
//
// 时间复杂度: O(n log n)，其中 n 为数组长度
// 空间复杂度: O(n)

// 由于编译环境限制，使用简单的数组实现

const int MAXN = 100005;
int nums[MAXN];
int sorted[MAXN];
int tree[4 * MAXN];
int result[MAXN];
int n;

// 更新节点值（单点更新）
void update(int node, int start, int end, int index, int val) {
    // 找到叶子节点，更新值
    if (start == end) {
        tree[node] += val;
        return;
    }

    // 在左右子树中查找需要更新的索引
    int mid = (start + end) / 2;
    if (index <= mid) {
        // 在左子树中
        update(2 * node + 1, start, mid, index, val);
    } else {
        // 在右子树中
        update(2 * node + 2, mid + 1, end, index, val);
    }

    // 更新当前节点的值为左右子节点值的和
    tree[node] = tree[2 * node + 1] + tree[2 * node + 2];
}

// 查询区间和
int query(int node, int start, int end, int left, int right) {
```

```
    // 当前区间与查询区间无交集
    if (right < start || left > end) {
        return 0;
    }

    // 当前区间完全包含在查询区间内
    if (left <= start && end <= right) {
        return tree[node];
    }

    // 当前区间与查询区间有部分交集，递归查询左右子树
    int mid = (start + end) / 2;
    int leftSum = query(2 * node + 1, start, mid, left, right);
    int rightSum = query(2 * node + 2, mid + 1, end, left, right);
    return leftSum + rightSum;
}

// 二分查找元素在排序数组中的位置
int binarySearch(int left, int right, int target) {
    while (left <= right) {
        int mid = left + (right - left) / 2;
        if (sorted[mid] == target) {
            return mid;
        } else if (sorted[mid] < target) {
            left = mid + 1;
        } else {
            right = mid - 1;
        }
    }
    return left;
}

// 主函数实现
void countSmaller(int nums_arr[], int size) {
    n = size;

    // 复制数组
    for (int i = 0; i < size; i++) {
        nums[i] = nums_arr[i];
        sorted[i] = nums_arr[i];
    }

    // 排序用于离散化
```

```
    // 简单冒泡排序（因为不能使用<algorithm>）
    for (int i = 0; i < size - 1; i++) {
        for (int j = 0; j < size - 1 - i; j++) {
            if (sorted[j] > sorted[j + 1]) {
                int temp = sorted[j];
                sorted[j] = sorted[j + 1];
                sorted[j + 1] = temp;
            }
        }
    }

    // 去重
    int uniqueSize = 1;
    for (int i = 1; i < size; i++) {
        if (sorted[i] != sorted[i - 1]) {
            sorted[uniqueSize++] = sorted[i];
        }
    }

    // 初始化线段树
    for (int i = 0; i < 4 * uniqueSize; i++) {
        tree[i] = 0;
    }

    // 从右向左遍历数组
    for (int i = n - 1; i >= 0; i--) {
        // 找到当前元素在离散化数组中的位置
        int pos = binarySearch(0, uniqueSize - 1, nums[i]);

        // 查询比当前元素小的元素个数（在值域上查询[0, pos-1]区间和）
        result[i] = query(0, 0, uniqueSize - 1, 0, pos - 1);

        // 更新当前元素的计数（在值域上对位置 pos 进行+1 操作）
        update(0, 0, uniqueSize - 1, pos, 1);
    }
}

// ================================================================================================
// LeetCode 1649. Create Sorted Array through Instructions
// 题目链接：https://leetcode.com/problems/create-sorted-array-through-instructions/
// 题目描述：
// 给你一个整数数组 instructions，你需要根据 instructions 中的元素创建一个有序数组。
// 一开始数组为空。你需要依次读取 instructions 中的元素，并将它插入到有序数组中的正确位置。
```

```
// 每次插入操作的代价是以下两者的较小值：
// 1. 有多少个元素严格小于 instructions[i]（左边）
// 2. 有多少个元素严格大于 instructions[i]（右边）
// 返回插入所有元素的总最小代价。由于答案可能很大，请返回它对 10^9 + 7 取模的结果。
// ============================================================================================

const int MOD = 1e9 + 7;

// LeetCode 1649 题的实现函数
long long createSortedArray(int instructions[], int size) {
    // 离散化处理
    int* sorted_inst = new int[size];
    for (int i = 0; i < size; i++) {
        sorted_inst[i] = instructions[i];
    }

    // 排序用于离散化
    // 简单冒泡排序
    for (int i = 0; i < size - 1; i++) {
        for (int j = 0; j < size - 1 - i; j++) {
            if (sorted_inst[j] > sorted_inst[j + 1]) {
                int temp = sorted_inst[j];
                sorted_inst[j] = sorted_inst[j + 1];
                sorted_inst[j + 1] = temp;
            }
        }
    }

    // 去重
    int uniqueSize = 1;
    for (int i = 1; i < size; i++) {
        if (sorted_inst[i] != sorted_inst[i - 1]) {
            sorted_inst[uniqueSize++] = sorted_inst[i];
        }
    }

    // 初始化线段树
    for (int i = 0; i < 4 * uniqueSize; i++) {
        tree[i] = 0;
    }

    long long totalCost = 0;
```

```
    // 处理每个指令
    for (int i = 0; i < size; i++) {
        int value = instructions[i];

        // 找到离散化后的位置
        int pos = binarySearch(0, uniqueSize - 1, value);

        // 计算左边比当前元素小的个数
        int smallerCount = query(0, 0, uniqueSize - 1, 0, pos - 1);

        // 计算右边比当前元素大的个数（总元素数减去到当前索引的前缀和）
        int largerCount = i - query(0, 0, uniqueSize - 1, 0, pos);

        // 取较小值作为当前操作的代价
        totalCost = (totalCost + (smallerCount < largerCount ? smallerCount : largerCount)) %
MOD;

        // 更新线段树，将当前元素的计数加 1
        update(0, 0, uniqueSize - 1, pos, 1);
    }

    delete[] sorted_inst;
    return totalCost;
}

// 测试 LeetCode 1649 题的函数
void testLeetCode1649() {
    // 测试用例 1
    int instructions1[] = {1, 5, 6, 2};
    long long result1 = createSortedArray(instructions1, 4);
    // printf("LeetCode 1649 测试用例 1 结果: %lld\n", result1); // 预期输出: 1

    // 测试用例 2
    int instructions2[] = {1, 2, 3, 6, 5, 4};
    long long result2 = createSortedArray(instructions2, 6);
    // printf("LeetCode 1649 测试用例 2 结果: %lld\n", result2); // 预期输出: 3

    // 测试用例 3
    int instructions3[] = {1, 3, 3, 3, 2, 4, 2, 1, 2};
    long long result3 = createSortedArray(instructions3, 9);
    // printf("LeetCode 1649 测试用例 3 结果: %lld\n", result3); // 预期输出: 4
}
```

// 由于编译环境限制，不提供 main 函数测试
// 可以通过调用 countSmaller 函数并检查 result 数组来验证结果
// 也可以调用 testLeetCode1649 函数来测试 LeetCode 1649 题的实现

==========

文件: Code10_CountOfSmallerNumbersAfterSelf.java
==========
```
package class111;

import java.util.ArrayList;
import java.util.List;

// Count of Smaller Numbers After Self (计算右侧小于当前元素的个数)
// 题目来源: LeetCode 315. Count of Smaller Numbers After Self
// 题目链接: https://leetcode.cn/problems/count-of-smaller-numbers-after-self
// 题目链接: https://leetcode.com/problems/count-of-smaller-numbers-after-self
//
// 题目描述:
// 给你一个整数数组 nums ，按要求返回一个新数组 counts 。
// 数组 counts 有该性质：counts[i] 的值是 nums[i] 右侧小于 nums[i] 的元素的数量。
// 示例 1：
// 输入：nums = [5,2,6,1]
// 输出：[2,1,1,0]
// 解释：
// 5 的右侧有 2 个更小的元素 (2 和 1)
// 2 的右侧有 1 个更小的元素 (1)
// 6 的右侧有 1 个更小的元素 (1)
// 1 的右侧有 0 个更小的元素
// 示例 2：
// 输入：nums = [-1]
// 输出：[0]
// 示例 3：
// 输入：nums = [-1,-1]
// 输出：[0,0]
// 提示：
// 1 <= nums.length <= 10^5
// -10^4 <= nums[i] <= 10^4
//
// 解题思路:
// 1. 使用离散化+线段树的方法解决
// 2. 从右向左遍历数组，维护一个值域线段树
// 3. 对于每个元素，查询值域中小于它的元素个数
```

```
// 4. 将当前元素插入到线段树中
// 5. 利用离散化处理大范围的值域
//
// 时间复杂度: O(n log n)，其中 n 为数组长度
// 空间复杂度: O(n)

public class Code10_CountOfSmallerNumbersAfterSelf {

    // 线段树实现（用于离散化后的值域）
    private static class SegmentTree {
        private int[] tree;
        private int n;

        public SegmentTree(int size) {
            this.n = size;
            // 线段树需要 4*n 的空间
            this.tree = new int[4 * size];
        }

        // 更新节点值（单点更新）
        public void update(int index, int val) {
            updateHelper(0, 0, n - 1, index, val);
        }

        // 更新辅助函数
        private void updateHelper(int node, int start, int end, int index, int val) {
            // 找到叶子节点，更新值
            if (start == end) {
                tree[node] += val;
                return;
            }

            // 在左右子树中查找需要更新的索引
            int mid = (start + end) / 2;
            if (index <= mid) {
                // 在左子树中
                updateHelper(2 * node + 1, start, mid, index, val);
            } else {
                // 在右子树中
                updateHelper(2 * node + 2, mid + 1, end, index, val);
            }

            // 更新当前节点的值为左右子节点值的和
```

```
        tree[node] = tree[2 * node + 1] + tree[2 * node + 2];
    }

    // 查询区间和
    public int query(int left, int right) {
        // 处理边界情况
        if (left > right) return 0;
        return queryHelper(0, 0, n - 1, left, right);
    }

    // 查询区间和辅助函数
    private int queryHelper(int node, int start, int end, int left, int right) {
        // 当前区间与查询区间无交集
        if (right < start || left > end) {
            return 0;
        }

        // 当前区间完全包含在查询区间内
        if (left <= start && end <= right) {
            return tree[node];
        }

        // 当前区间与查询区间有部分交集，递归查询左右子树
        int mid = (start + end) / 2;
        int leftSum = queryHelper(2 * node + 1, start, mid, left, right);
        int rightSum = queryHelper(2 * node + 2, mid + 1, end, left, right);
        return leftSum + rightSum;
    }
}

// ==========================================================================================
// LeetCode 1649. Create Sorted Array through Instructions
// 题目链接：https://leetcode.com/problems/create-sorted-array-through-instructions/
// 题目描述：
// 给你一个整数数组 instructions，你需要根据 instructions 中的元素创建一个有序数组。
// 一开始数组为空。你需要依次读取 instructions 中的元素，并将它插入到有序数组中的正确位置。
// 每次插入操作的代价是以下两者的较小值：
// 1. 有多少个元素严格小于 instructions[i]（左边）
// 2. 有多少个元素严格大于 instructions[i]（右边）
// 返回插入所有元素的总最小代价。由于答案可能很大，请返回它对 10^9 + 7 取模的结果。
//
// 示例：
// 输入：instructions = [1,5,6,2]
```

```
// 输出：1
// 解释：插入 1 时，数组为空，代价为 0。
// 插入 5 时，左边有 1 个元素比 5 小，右边没有元素，代价为 min(1, 0) = 0。
// 插入 6 时，左边有 2 个元素比 6 小，右边没有元素，代价为 min(2, 0) = 0。
// 插入 2 时，左边有 1 个元素比 2 小，右边有 2 个元素比 2 大，代价为 min(1, 2) = 1。
// 总代价为 0 + 0 + 0 + 1 = 1
// ============================================================================================

public static class CreateSortedArrayThroughInstructions {
    private static final int MOD = 1000000007;

    /**
     * 计算创建有序数组的最小代价
     * @param instructions 指令数组
     * @return 总最小代价
     */
    public int createSortedArray(int[] instructions) {
        // 离散化处理
        int n = instructions.length;
        int[] sorted = instructions.clone();
        java.util.Arrays.sort(sorted);

        // 去重
        int uniqueSize = 1;
        for (int i = 1; i < n; i++) {
            if (sorted[i] != sorted[i - 1]) {
                sorted[uniqueSize++] = sorted[i];
            }
        }

        // 创建线段树
        SegmentTree st = new SegmentTree(uniqueSize);
        long totalCost = 0; // 使用 long 避免溢出

        // 处理每个指令
        for (int i = 0; i < instructions.length; i++) {
            int value = instructions[i];

            // 找到离散化后的位置
            int pos = binarySearch(sorted, 0, uniqueSize - 1, value);

            // 计算左边比当前元素小的个数
            int smallerCount = st.query(0, pos - 1);
```

```
                // 计算右边比当前元素大的个数（总元素数减去到当前索引的前缀和）
                int largerCount = i - st.query(0, pos);

                // 取较小值作为当前操作的代价
                totalCost = (totalCost + Math.min(smallerCount, largerCount)) % MOD;

                // 更新线段树，将当前元素的计数加 1
                st.update(pos, 1);
            }

            return (int) totalCost;
        }
    }

    // 测试 LeetCode 1649 题的方法
    public static void testLeetCode1649() {
        CreateSortedArrayThroughInstructions solution = new
CreateSortedArrayThroughInstructions();

        // 测试用例 1
        int[] instructions1 = {1, 5, 6, 2};
        System.out.println("LeetCode 1649 测试用例 1 结果: " +
solution.createSortedArray(instructions1)); // 预期输出: 1

        // 测试用例 2
        int[] instructions2 = {1, 2, 3, 6, 5, 4};
        System.out.println("LeetCode 1649 测试用例 2 结果: " +
solution.createSortedArray(instructions2)); // 预期输出: 3

        // 测试用例 3
        int[] instructions3 = {1, 3, 3, 3, 2, 4, 2, 1, 2};
        System.out.println("LeetCode 1649 测试用例 3 结果: " +
solution.createSortedArray(instructions3)); // 预期输出: 4
    }

    public static List<Integer> countSmaller(int[] nums) {
        int n = nums.length;
        List<Integer> result = new ArrayList<>();

        // 离散化处理
        // 1. 收集所有不同的值
        int[] sorted = nums.clone();
```

```
        java.util.Arrays.sort(sorted);

        // 2. 去重
        int uniqueSize = 1;
        for (int i = 1; i < n; i++) {
            if (sorted[i] != sorted[i - 1]) {
                sorted[uniqueSize++] = sorted[i];
            }
        }

        // 3. 创建离散化映射
        // 线段树的大小为去重后的元素个数
        SegmentTree st = new SegmentTree(uniqueSize);

        // 从右向左遍历数组
        for (int i = n - 1; i >= 0; i--) {
            // 找到当前元素在离散化数组中的位置
            int pos = binarySearch(sorted, 0, uniqueSize - 1, nums[i]);

            // 查询比当前元素小的元素个数（在值域上查询[0, pos-1]区间和）
            int count = st.query(0, pos - 1);
            result.add(0, count); // 插入到结果列表的开头

            // 更新当前元素的计数（在值域上对位置 pos 进行+1 操作）
            st.update(pos, 1);
        }

        return result;
    }

    // 二分查找元素在排序数组中的位置
    private static int binarySearch(int[] arr, int left, int right, int target) {
        while (left <= right) {
            int mid = left + (right - left) / 2;
            if (arr[mid] == target) {
                return mid;
            } else if (arr[mid] < target) {
                left = mid + 1;
            } else {
                right = mid - 1;
            }
        }
        return left;
```

```
    }

    // 测试方法
    public static void main(String[] args) {
        // 测试用例 1
        int[] nums1 = {5, 2, 6, 1};
        System.out.println(countSmaller(nums1)); // 输出: [2, 1, 1, 0]

        // 测试用例 2
        int[] nums2 = {-1};
        System.out.println(countSmaller(nums2)); // 输出: [0]

        // 测试用例 3
        int[] nums3 = {-1, -1};
        System.out.println(countSmaller(nums3)); // 输出: [0, 0]

        // 测试 LeetCode 1649 题
        System.out.println("\n 测试 LeetCode 1649 题:");
        testLeetCode1649();
    }
}
```

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

文件: Code10_CountOfSmallerNumbersAfterSelf.py

```
==========================================
# Count of Smaller Numbers After Self (计算右侧小于当前元素的个数)
# 题目来源: LeetCode 315. Count of Smaller Numbers After Self
# 题目链接: https://leetcode.cn/problems/count-of-smaller-numbers-after-self
# 题目链接: https://leetcode.com/problems/count-of-smaller-numbers-after-self
#
# 题目描述:
# 给你一个整数数组 nums ，按要求返回一个新数组 counts 。
# 数组 counts 有该性质：counts[i] 的值是 nums[i] 右侧小于 nums[i] 的元素的数量。
# 示例 1：
# 输入：nums = [5,2,6,1]
# 输出：[2,1,1,0]
# 解释：
# 5 的右侧有 2 个更小的元素 (2 和 1)
# 2 的右侧有 1 个更小的元素 (1)
# 6 的右侧有 1 个更小的元素 (1)
# 1 的右侧有 0 个更小的元素
# 示例 2：
```

```
# 输入：nums = [-1]
# 输出：[0]
# 示例 3：
# 输入：nums = [-1,-1]
# 输出：[0,0]
# 提示：
# 1 <= nums.length <= 10^5
# -10^4 <= nums[i] <= 10^4
#
# 解题思路:
# 1. 使用离散化+线段树的方法解决
# 2. 从右向左遍历数组，维护一个值域线段树
# 3. 对于每个元素，查询值域中小于它的元素个数
# 4. 将当前元素插入到线段树中
# 5. 利用离散化处理大范围的值域
#
# 时间复杂度: O(n log n)，其中 n 为数组长度
# 空间复杂度: O(n)

class SegmentTree:
    def __init__(self, size):
        self.n = size
        # 线段树需要 4*n 的空间
        self.tree = [0] * (4 * size)

    # 更新节点值（单点更新）
    def update(self, index, val):
        self._update_helper(0, 0, self.n - 1, index, val)

    # 更新辅助函数
    def _update_helper(self, node, start, end, index, val):
        # 找到叶子节点，更新值
        if start == end:
            self.tree[node] += val
            return

        # 在左右子树中查找需要更新的索引
        mid = (start + end) // 2
        if index <= mid:
            # 在左子树中
            self._update_helper(2 * node + 1, start, mid, index, val)
        else:
            # 在右子树中
```

```
            self._update_helper(2 * node + 2, mid + 1, end, index, val)

        # 更新当前节点的值为左右子节点值的和
        self.tree[node] = self.tree[2 * node + 1] + self.tree[2 * node + 2]

    # 查询区间和
    def query(self, left, right):
        # 处理边界情况
        if left > right:
            return 0
        return self._query_helper(0, 0, self.n - 1, left, right)

    # 查询区间和辅助函数
    def _query_helper(self, node, start, end, left, right):
        # 当前区间与查询区间无交集
        if right < start or left > end:
            return 0

        # 当前区间完全包含在查询区间内
        if left <= start and end <= right:
            return self.tree[node]

        # 当前区间与查询区间有部分交集，递归查询左右子树
        mid = (start + end) // 2
        left_sum = self._query_helper(2 * node + 1, start, mid, left, right)
        right_sum = self._query_helper(2 * node + 2, mid + 1, end, left, right)
        return left_sum + right_sum


def count_smaller(nums):
    n = len(nums)
    result = []

    # 离散化处理
    # 1. 收集所有不同的值
    sorted_nums = sorted(nums)

    # 2. 去重
    unique_size = 1
    for i in range(1, n):
        if sorted_nums[i] != sorted_nums[i - 1]:
            sorted_nums[unique_size] = sorted_nums[i]
            unique_size += 1
```

```
    # 3. 创建离散化映射
    # 线段树的大小为去重后的元素个数
    st = SegmentTree(unique_size)

    # 从右向左遍历数组
    for i in range(n - 1, -1, -1):
        # 找到当前元素在离散化数组中的位置
        pos = binary_search(sorted_nums, 0, unique_size - 1, nums[i])

        # 查询比当前元素小的元素个数（在值域上查询[0, pos-1]区间和）
        count = st.query(0, pos - 1)
        result.insert(0, count)  # 插入到结果列表的开头

        # 更新当前元素的计数（在值域上对位置 pos 进行+1 操作）
        st.update(pos, 1)

    return result


# 二分查找元素在排序数组中的位置
def binary_search(arr, left, right, target):
    while left <= right:
        mid = left + (right - left) // 2
        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            left = mid + 1
        else:
            right = mid - 1
    return left


# 测试方法
if __name__ == "__main__":
    # 测试用例 1
    nums1 = [5, 2, 6, 1]
    print(count_smaller(nums1))  # 输出: [2, 1, 1, 0]

    # 测试用例 2
    nums2 = [-1]
    print(count_smaller(nums2))  # 输出: [0]
```

```
    # 测试用例 3
    nums3 = [-1, -1]
    print(count_smaller(nums3))  # 输出: [0, 0]

# ==========================================================================================
# LeetCode 1649. Create Sorted Array through Instructions
# 题目链接：https://leetcode.com/problems/create-sorted-array-through-instructions/
# 题目描述：
# 给你一个整数数组 instructions，你需要根据 instructions 中的元素创建一个有序数组。
# 一开始数组为空。你需要依次读取 instructions 中的元素，并将它插入到有序数组中的正确位置。
# 每次插入操作的代价是以下两者的较小值：
# 1. 有多少个元素严格小于 instructions[i]（左边）
# 2. 有多少个元素严格大于 instructions[i]（右边）
# 返回插入所有元素的总最小代价。由于答案可能很大，请返回它对 10^9 + 7 取模的结果。
# ==========================================================================================

MOD = 10**9 + 7


def create_sorted_array(instructions):
    """
    计算创建有序数组的最小代价

    Args:
        instructions: 指令数组

    Returns:
        总最小代价
    """
    n = len(instructions)

    # 离散化处理
    sorted_inst = sorted(instructions)

    # 去重
    unique_size = 1
    for i in range(1, n):
        if sorted_inst[i] != sorted_inst[i - 1]:
            sorted_inst[unique_size] = sorted_inst[i]
            unique_size += 1

    # 创建线段树
    st = SegmentTree(unique_size)
```

```
    total_cost = 0

    # 处理每个指令
    for i in range(n):
        value = instructions[i]

        # 找到离散化后的位置
        pos = binary_search(sorted_inst, 0, unique_size - 1, value)

        # 计算左边比当前元素小的个数
        smaller_count = st.query(0, pos - 1)

        # 计算右边比当前元素大的个数（总元素数减去到当前索引的前缀和）
        larger_count = i - st.query(0, pos)

        # 取较小值作为当前操作的代价
        total_cost = (total_cost + min(smaller_count, larger_count)) % MOD

        # 更新线段树，将当前元素的计数加 1
        st.update(pos, 1)

    return total_cost


# 测试 LeetCode 1649 题
def test_leetcode_1649():
    # 测试用例 1
    instructions1 = [1, 5, 6, 2]
    print("LeetCode 1649 测试用例 1 结果:", create_sorted_array(instructions1))  # 预期输出: 1

    # 测试用例 2
    instructions2 = [1, 2, 3, 6, 5, 4]
    print("LeetCode 1649 测试用例 2 结果:", create_sorted_array(instructions2))  # 预期输出: 3

    # 测试用例 3
    instructions3 = [1, 3, 3, 3, 2, 4, 2, 1, 2]
    print("LeetCode 1649 测试用例 3 结果:", create_sorted_array(instructions3))  # 预期输出: 4


# 如果直接运行此文件，也测试 LeetCode 1649 题
if __name__ == "__main__":
    # 测试原始问题
    print("\n 测试原始问题 Count of Smaller Numbers After Self:")
```

```
    # 测试用例 1
    nums1 = [5, 2, 6, 1]
    print(count_smaller(nums1))  # 输出: [2, 1, 1, 0]

    # 测试用例 2
    nums2 = [-1]
    print(count_smaller(nums2))  # 输出: [0]

    # 测试用例 3
    nums3 = [-1, -1]
    print(count_smaller(nums3))  # 输出: [0, 0]

    # 测试 LeetCode 1649 题
    print("\n 测试 LeetCode 1649 题:")
    test_leetcode_1649()
```

==========

文件: Code11_HorribleQueries.cpp

==========

```
// Horrible Queries (可怕的查询)
// 题目来源: SPOJ HORRIBLE - Horrible Queries
// 题目链接: https://www.spoj.com/problems/HORRIBLE/
//
// 题目描述:
// 现有一个长度为 n 的序列，开始时所有位置都为 0
// 有以下两种操作：
// 0 p q v : 将区间[p,q]内每个位置的值都加上 v
// 1 p q   : 查询区间[p,q]内所有位置的值的和
//
// 解题思路:
// 1. 使用带懒惰传播的线段树实现区间更新和区间查询
// 2. 懒惰传播用于延迟更新，避免不必要的计算
// 3. 区间更新时，只在必要时才将更新操作传递给子节点
// 4. 查询时确保所有相关的懒惰标记都被处理
//
// 时间复杂度:
// - 区间更新: O(log n)
// - 区间查询: O(log n)
// 空间复杂度: O(n)

// 由于编译环境限制，使用简单的数组实现
```

```
const int MAXN = 100005;
long long tree[4 * MAXN];
long long lazy[4 * MAXN];
int n;

// 函数声明
void pushDown(int node, int start, int end);
void rangeAddHelper(int node, int start, int end, int l, int r, long long val);
long long queryHelper(int node, int start, int end, int l, int r);

// 区间加法更新 [l, r] 区间内每个元素加上 val
void rangeAdd(int l, int r, long long val) {
    rangeAddHelper(0, 0, n - 1, l, r, val);
}

// 查询区间和
long long query(int l, int r) {
    return queryHelper(0, 0, n - 1, l, r);
}

// 区间加法更新辅助函数
void rangeAddHelper(int node, int start, int end, int l, int r, long long val) {
    // 1. 先处理懒惰标记
    pushDown(node, start, end);

    // 2. 当前区间与更新区间无交集
    if (start > r || end < l) {
        return;
    }

    // 3. 当前区间完全包含在更新区间内
    if (start >= l && end <= r) {
        // 更新当前节点的值
        tree[node] += val * (end - start + 1);
        // 如果不是叶子节点，设置懒惰标记
        if (start != end) {
            lazy[node] += val;
        }
        return;
    }

    // 4. 当前区间与更新区间有部分交集，递归处理左右子树
    int mid = (start + end) / 2;
```

```
    rangeAddHelper(2 * node + 1, start, mid, l, r, val);
    rangeAddHelper(2 * node + 2, mid + 1, end, l, r, val);

    // 更新当前节点的值
    tree[node] = tree[2 * node + 1] + tree[2 * node + 2];
}

// 查询区间和辅助函数
long long queryHelper(int node, int start, int end, int l, int r) {
    // 1. 先处理懒惰标记
    pushDown(node, start, end);

    // 2. 当前区间与查询区间无交集
    if (start > r || end < l) {
        return 0;
    }

    // 3. 当前区间完全包含在查询区间内
    if (start >= l && end <= r) {
        return tree[node];
    }

    // 4. 当前区间与查询区间有部分交集，递归查询左右子树
    int mid = (start + end) / 2;
    long long leftSum = queryHelper(2 * node + 1, start, mid, l, r);
    long long rightSum = queryHelper(2 * node + 2, mid + 1, end, l, r);
    return leftSum + rightSum;
}

// 下推懒惰标记
void pushDown(int node, int start, int end) {
    // 如果当前节点没有懒惰标记，直接返回
    if (lazy[node] == 0) {
        return;
    }

    // 将懒惰标记应用到当前节点
    tree[node] += lazy[node] * (end - start + 1);

    // 如果不是叶子节点，将懒惰标记传递给子节点
    if (start != end) {
        int mid = (start + end) / 2;
        lazy[2 * node + 1] += lazy[node];
```

```
        lazy[2 * node + 2] += lazy[node];
    }

    // 清除当前节点的懒惰标记
    lazy[node] = 0;
}

// 由于编译环境限制，不提供 main 函数测试
// 可以通过调用 rangeAdd 和 query 函数来使用
```

==========

文件: Code11_HorribleQueries.java

==========

```
package class111;

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

// Horrible Queries (可怕的查询)
// 题目来源: SPOJ HORRIBLE - Horrible Queries
// 题目链接: https://www.spoj.com/problems/HORRIBLE/
//
// 题目描述:
// 现有一个长度为 n 的序列，开始时所有位置都为 0
// 有以下两种操作：
// 0 p q v : 将区间[p,q]内每个位置的值都加上 v
// 1 p q   : 查询区间[p,q]内所有位置的值的和
//
// 解题思路:
// 1. 使用带懒惰传播的线段树实现区间更新和区间查询
// 2. 懒惰传播用于延迟更新，避免不必要的计算
// 3. 区间更新时，只在必要时才将更新操作传递给子节点
// 4. 查询时确保所有相关的懒惰标记都被处理
//
// 时间复杂度:
// - 区间更新: O(log n)
// - 区间查询: O(log n)
// 空间复杂度: O(n)
```

```
public class Code11_HorribleQueries {

    // 带懒惰传播的线段树实现
    private static class SegmentTree {
        private long[] tree;     // 线段树节点值（存储区间和）
        private long[] lazy;     // 懒惰标记数组
        private int n;           // 数组长度

        public SegmentTree(int size) {
            this.n = size;
            // 线段树需要 4*n 的空间
            this.tree = new long[4 * n];
            this.lazy = new long[4 * n];
        }

        // 区间加法更新 [l, r] 区间内每个元素加上 val
        public void rangeAdd(int l, int r, long val) {
            rangeAddHelper(0, 0, n - 1, l, r, val);
        }

        // 区间加法更新辅助函数
        private void rangeAddHelper(int node, int start, int end, int l, int r, long val) {
            // 1. 先处理懒惰标记
            pushDown(node, start, end);

            // 2. 当前区间与更新区间无交集
            if (start > r || end < l) {
                return;
            }

            // 3. 当前区间完全包含在更新区间内
            if (start >= l && end <= r) {
                // 更新当前节点的值
                tree[node] += val * (end - start + 1);
                // 如果不是叶子节点，设置懒惰标记
                if (start != end) {
                    lazy[node] += val;
                }
                return;
            }

            // 4. 当前区间与更新区间有部分交集，递归处理左右子树
```

```
        int mid = (start + end) / 2;
        rangeAddHelper(2 * node + 1, start, mid, l, r, val);
        rangeAddHelper(2 * node + 2, mid + 1, end, l, r, val);

        // 更新当前节点的值
        tree[node] = tree[2 * node + 1] + tree[2 * node + 2];
    }

    // 查询区间和
    public long query(int l, int r) {
        return queryHelper(0, 0, n - 1, l, r);
    }

    // 查询区间和辅助函数
    private long queryHelper(int node, int start, int end, int l, int r) {
        // 1. 先处理懒惰标记
        pushDown(node, start, end);

        // 2. 当前区间与查询区间无交集
        if (start > r || end < l) {
            return 0;
        }

        // 3. 当前区间完全包含在查询区间内
        if (start >= l && end <= r) {
            return tree[node];
        }

        // 4. 当前区间与查询区间有部分交集，递归查询左右子树
        int mid = (start + end) / 2;
        long leftSum = queryHelper(2 * node + 1, start, mid, l, r);
        long rightSum = queryHelper(2 * node + 2, mid + 1, end, l, r);
        return leftSum + rightSum;
    }

    // 下推懒惰标记
    private void pushDown(int node, int start, int end) {
        // 如果当前节点没有懒惰标记，直接返回
        if (lazy[node] == 0) {
            return;
        }

        // 将懒惰标记应用到当前节点
```

```
            tree[node] += lazy[node] * (end - start + 1);

            // 如果不是叶子节点，将懒惰标记传递给子节点
            if (start != end) {
                int mid = (start + end) / 2;
                lazy[2 * node + 1] += lazy[node];
                lazy[2 * node + 2] += lazy[node];
            }

            // 清除当前节点的懒惰标记
            lazy[node] = 0;
        }
    }

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

        in.nextToken();
        int t = (int) in.nval; // 测试用例数量

        for (int i = 0; i < t; i++) {
            in.nextToken();
            int n = (int) in.nval; // 序列长度
            in.nextToken();
            int c = (int) in.nval; // 操作数量

            SegmentTree st = new SegmentTree(n);

            for (int j = 0; j < c; j++) {
                in.nextToken();
                int op = (int) in.nval;

                if (op == 0) {
                    // 区间更新操作
                    in.nextToken();
                    int p = (int) in.nval - 1; // 转换为0索引
                    in.nextToken();
                    int q = (int) in.nval - 1; // 转换为0索引
                    in.nextToken();
                    long v = (long) in.nval;
```

```
                    st.rangeAdd(p, q, v);
                } else {
                    // 区间查询操作
                    in.nextToken();
                    int p = (int) in.nval - 1; // 转换为0索引
                    in.nextToken();
                    int q = (int) in.nval - 1; // 转换为0索引

                    out.println(st.query(p, q));
                }
            }
        }

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

==========

文件: Code11_HorribleQueries.py

==========

```
# Horrible Queries (可怕的查询)
# 题目来源: SPOJ HORRIBLE - Horrible Queries
# 题目链接: https://www.spoj.com/problems/HORRIBLE/
#
# 题目描述:
# 现有一个长度为n的序列，开始时所有位置都为0
# 有以下两种操作：
# 0 p q v : 将区间[p,q]内每个位置的值都加上v
# 1 p q   : 查询区间[p,q]内所有位置的值的和
#
# 解题思路:
# 1. 使用带懒惰传播的线段树实现区间更新和区间查询
# 2. 懒惰传播用于延迟更新，避免不必要的计算
# 3. 区间更新时，只在必要时才将更新操作传递给子节点
# 4. 查询时确保所有相关的懒惰标记都被处理
#
# 时间复杂度:
# - 区间更新: O(log n)
# - 区间查询: O(log n)
```

```
# 空间复杂度: O(n)

class SegmentTree:
    def __init__(self, size):
        self.n = size
        # 线段树需要 4*n 的空间
        self.tree = [0] * (4 * self.n)
        self.lazy = [0] * (4 * self.n)

    # 区间加法更新 [l, r] 区间内每个元素加上 val
    def range_add(self, l, r, val):
        self._range_add_helper(0, 0, self.n - 1, l, r, val)

    # 区间加法更新辅助函数
    def _range_add_helper(self, node, start, end, l, r, val):
        # 1. 先处理懒惰标记
        self._push_down(node, start, end)

        # 2. 当前区间与更新区间无交集
        if start > r or end < l:
            return

        # 3. 当前区间完全包含在更新区间内
        if start >= l and end <= r:
            # 更新当前节点的值
            self.tree[node] += val * (end - start + 1)
            # 如果不是叶子节点，设置懒惰标记
            if start != end:
                self.lazy[node] += val
            return

        # 4. 当前区间与更新区间有部分交集，递归处理左右子树
        mid = (start + end) // 2
        self._range_add_helper(2 * node + 1, start, mid, l, r, val)
        self._range_add_helper(2 * node + 2, mid + 1, end, l, r, val)

        # 更新当前节点的值
        self.tree[node] = self.tree[2 * node + 1] + self.tree[2 * node + 2]

    # 查询区间和
    def query(self, l, r):
        return self._query_helper(0, 0, self.n - 1, l, r)
```

```
    # 查询区间和辅助函数
    def _query_helper(self, node, start, end, l, r):
        # 1. 先处理懒惰标记
        self._push_down(node, start, end)

        # 2. 当前区间与查询区间无交集
        if start > r or end < l:
            return 0

        # 3. 当前区间完全包含在查询区间内
        if start >= l and end <= r:
            return self.tree[node]

        # 4. 当前区间与查询区间有部分交集，递归查询左右子树
        mid = (start + end) // 2
        left_sum = self._query_helper(2 * node + 1, start, mid, l, r)
        right_sum = self._query_helper(2 * node + 2, mid + 1, end, l, r)
        return left_sum + right_sum

    # 下推懒惰标记
    def _push_down(self, node, start, end):
        # 如果当前节点没有懒惰标记，直接返回
        if self.lazy[node] == 0:
            return

        # 将懒惰标记应用到当前节点
        self.tree[node] += self.lazy[node] * (end - start + 1)

        # 如果不是叶子节点，将懒惰标记传递给子节点
        if start != end:
            mid = (start + end) // 2
            self.lazy[2 * node + 1] += self.lazy[node]
            self.lazy[2 * node + 2] += self.lazy[node]

        # 清除当前节点的懒惰标记
        self.lazy[node] = 0


# 由于SPOJ在线测试需要特定的输入输出格式，这里提供一个简化版本的测试方法
def test_horrible_queries():
    # 示例测试
    n = 8  # 序列长度
    st = SegmentTree(n)
```

```
    # 操作 1: 将区间[2,4]内每个位置的值都加上 3 (注意转换为 0 索引)
    st.range_add(1, 3, 3)

    # 操作 2: 查询区间[1,3]内所有位置的值的和 (注意转换为 0 索引)
    result = st.query(0, 2)
    print(result)  # 输出: 9 (3+3+3=9)

    # 操作 3: 将区间[5,7]内每个位置的值都加上 2 (注意转换为 0 索引)
    st.range_add(4, 6, 2)

    # 操作 4: 查询区间[4,6]内所有位置的值的和 (注意转换为 0 索引)
    result = st.query(3, 5)
    print(result)  # 输出: 6 (2+2+2=6)


# 测试方法
if __name__ == "__main__":
    test_horrible_queries()
```

==========

文件: Code12_CreateSortedArrayThroughInstructions.cpp

==========

```
/**
 * LeetCode 1649. Create Sorted Array through Instructions (通过指令创建有序数组)
 * 题目链接：https://leetcode.com/problems/create-sorted-array-through-instructions/
 *
 * 题目描述：
 * 给你一个整数数组 instructions，你需要根据 instructions 中的元素创建一个有序数组。
 * 一开始数组为空。你需要依次读取 instructions 中的元素，并将它插入到有序数组中的正确位置。
 * 每次插入操作的代价是以下两者的较小值：
 * 1. 有多少个元素严格小于 instructions[i]（左边）
 * 2. 有多少个元素严格大于 instructions[i]（右边）
 * 返回插入所有元素的总最小代价。由于答案可能很大，请返回它对 10^9 + 7 取模的结果。
 *
 * 示例：
 * 输入：instructions = [1,5,6,2]
 * 输出：1
 * 解释：插入 1 时，数组为空，代价为 0。
 * 插入 5 时，左边有 1 个元素比 5 小，右边没有元素，代价为 min(1, 0) = 0。
 * 插入 6 时，左边有 2 个元素比 6 小，右边没有元素，代价为 min(2, 0) = 0。
 * 插入 2 时，左边有 1 个元素比 2 小，右边有 2 个元素比 2 大，代价为 min(1, 2) = 1。
```

```
 * 总代价为 0 + 0 + 0 + 1 = 1
 *
 * 解题思路：
 * 这道题可以使用离散化线段树来解决。我们需要高效地统计数组中有多少元素小于当前元素，以及有多少元素大于当前元素。
 * 具体步骤：
 * 1. 离散化处理：将指令数组中的所有元素进行排序去重，得到每个元素的排名（离散化值）
 * 2. 构建线段树：维护每个值出现的次数
 * 3. 对于每个指令：
 *    a. 查询当前小于该元素的个数（即离散化后值减 1 的前缀和）
 *    b. 查询当前大于该元素的个数（即总元素数减去离散化后值的前缀和）
 *    c. 计算当前代价并累加到结果中
 *    d. 更新线段树，将该元素的计数加 1
 *
 * 时间复杂度分析：
 * - 离散化：O(n log n)，其中 n 是指令数组的长度
 * - 线段树构建：O(m)，其中 m 是离散化后的不同元素个数
 * - 每个查询和更新操作：O(log m)
 * - 总时间复杂度：O(n log n + n log m) = O(n log n)，因为 m ≤ n
 *
 * 空间复杂度分析：
 * - 线段树空间：O(m)
 * - 离散化数组：O(m)
 * - 总空间复杂度：O(m) = O(n)
 *
 * 本题最优解：线段树是本题的最优解之一，另外也可以使用树状数组（Fenwick Tree）实现，时间复杂度相同。
 */

// 由于编译环境限制，使用简单的数组实现

const int MAXN = 100005;
const int MOD = 1000000007;
int tree[2 * MAXN];  // 线段树数组
int n;               // 原始数据长度

/**
 * 初始化线段树
 * @param size 离散化后的值域大小
 */
void initSegmentTree(int size) {
    n = 1;
    // 计算大于等于 size 的最小 2 的幂次
```

```
    while (n < size) {
        n <<= 1;
    }
    // 初始化线段树数组
    for (int i = 0; i < 2 * n; i++) {
        tree[i] = 0;
    }
}

/**
 * 更新线段树中的某个位置的值
 * @param idx 离散化后的值对应的索引
 * @param delta 要增加的值（这里是1）
 */
void update(int idx, int delta) {
    idx += n; // 转换为线段树叶子节点索引
    tree[idx] += delta;
    // 向上更新父节点
    for (int i = idx >> 1; i >= 1; i >>= 1) {
        tree[i] = tree[2 * i] + tree[2 * i + 1];
    }
}

/**
 * 查询区间[0, idx]的和
 * @param idx 离散化后的值对应的索引
 * @return 区间和
 */
int query(int idx) {
    if (idx < 0) return 0;
    idx = (idx < n - 1) ? idx : n - 1; // 防止越界
    int res = 0;
    int l = n;       // 左边界（线段树叶子节点索引）
    int r = n + idx; // 右边界（线段树叶子节点索引）

    // 区间查询
    while (l <= r) {
        // 如果l是右孩子
        if ((l & 1) == 1) {
            res += tree[l];
            l++;
        }
        // 如果r是左孩子
```

```
        if ((r & 1) == 0) {
            res += tree[r];
            r--;
        }
        l >>= 1;
        r >>= 1;
    }
    return res;
}

/**
 * 简单排序函数（冒泡排序）
 * @param arr 待排序数组
 * @param size 数组大小
 */
void bubbleSort(int arr[], int size) {
    for (int i = 0; i < size - 1; i++) {
        for (int j = 0; j < size - 1 - i; j++) {
            if (arr[j] > arr[j + 1]) {
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
}

/**
 * 去重函数
 * @param arr 已排序数组
 * @param size 数组大小
 * @return 去重后的元素个数
 */
int removeDuplicates(int arr[], int size) {
    if (size == 0) return 0;
    int uniqueSize = 1;
    for (int i = 1; i < size; i++) {
        if (arr[i] != arr[i - 1]) {
            arr[uniqueSize++] = arr[i];
        }
    }
    return uniqueSize;
}
```

```
/**
 * 二分查找函数
 * @param arr 已排序数组
 * @param size 数组大小
 * @param target 目标值
 * @return 目标值在数组中的索引，如果不存在则返回应该插入的位置
 */
int binarySearch(int arr[], int size, int target) {
    int left = 0, right = size - 1;
    while (left <= right) {
        int mid = left + (right - left) / 2;
        if (arr[mid] == target) {
            return mid;
        } else if (arr[mid] < target) {
            left = mid + 1;
        } else {
            right = mid - 1;
        }
    }
    return left;
}

/**
 * 计算两个数中的较小值
 * @param a 第一个数
 * @param b 第二个数
 * @return 较小的数
 */
int min(int a, int b) {
    return (a < b) ? a : b;
}

/**
 * 计算创建有序数组的最小代价
 * @param instructions 指令数组
 * @param size 指令数组大小
 * @return 总最小代价
 */
int createSortedArray(int instructions[], int size) {
    // 离散化处理
    int sortedVals[MAXN];
    for (int i = 0; i < size; i++) {
```

```
        sortedVals[i] = instructions[i];
    }

    // 排序
    bubbleSort(sortedVals, size);

    // 去重
    int uniqueSize = removeDuplicates(sortedVals, size);

    // 构建线段树
    initSegmentTree(uniqueSize);
    long long totalCost = 0; // 使用 long long 避免溢出

    // 处理每个指令
    for (int i = 0; i < size; i++) {
        int value = instructions[i];
        int idx = binarySearch(sortedVals, uniqueSize, value);

        // 计算左边比当前元素小的个数（即前缀和）
        int smallerCount = query(idx - 1);

        // 计算右边比当前元素大的个数（总元素数减去到当前索引的前缀和）
        int largerCount = i - query(idx);

        // 取较小值作为当前操作的代价
        totalCost = (totalCost + min(smallerCount, largerCount)) % MOD;

        // 更新线段树，将当前元素的计数加 1
        update(idx, 1);
    }

    return (int)totalCost;
}

// 由于编译环境限制，不提供 main 函数测试
// 可以通过调用 createSortedArray 函数来使用

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

文件: Code12_CreateSortedArrayThroughInstructions.java
==============================================
package class111;
```

```
import java.util.Arrays;
import java.util.HashSet;
import java.util.Set;

/**
 * LeetCode 1649. Create Sorted Array through Instructions (通过指令创建有序数组)
 * 题目链接：https://leetcode.com/problems/create-sorted-array-through-instructions/
 *
 * 题目描述：
 * 给你一个整数数组 instructions，你需要根据 instructions 中的元素创建一个有序数组。
 * 一开始数组为空。你需要依次读取 instructions 中的元素，并将它插入到有序数组中的正确位置。
 * 每次插入操作的代价是以下两者的较小值：
 * 1. 有多少个元素严格小于 instructions[i]（左边）
 * 2. 有多少个元素严格大于 instructions[i]（右边）
 * 返回插入所有元素的总最小代价。由于答案可能很大，请返回它对 10^9 + 7 取模的结果。
 *
 * 示例：
 * 输入：instructions = [1,5,6,2]
 * 输出：1
 * 解释：插入 1 时，数组为空，代价为 0。
 * 插入 5 时，左边有 1 个元素比 5 小，右边没有元素，代价为 min(1, 0) = 0。
 * 插入 6 时，左边有 2 个元素比 6 小，右边没有元素，代价为 min(2, 0) = 0。
 * 插入 2 时，左边有 1 个元素比 2 小，右边有 2 个元素比 2 大，代价为 min(1, 2) = 1。
 * 总代价为 0 + 0 + 0 + 1 = 1
 *
 * 解题思路：
 * 这道题可以使用离散化线段树来解决。我们需要高效地统计数组中有多少元素小于当前元素，以及有多少元素大于当前元素。
 * 具体步骤：
 * 1. 离散化处理：将指令数组中的所有元素进行排序去重，得到每个元素的排名（离散化值）
 * 2. 构建线段树：维护每个值出现的次数
 * 3. 对于每个指令：
 *    a. 查询当前小于该元素的个数（即离散化后值减 1 的前缀和）
 *    b. 查询当前大于该元素的个数（即总元素数减去离散化后值的前缀和）
 *    c. 计算当前代价并累加到结果中
 *    d. 更新线段树，将该元素的计数加 1
 *
 * 时间复杂度分析：
 * - 离散化：O(n log n)，其中 n 是指令数组的长度
 * - 线段树构建：O(m)，其中 m 是离散化后的不同元素个数
 * - 每个查询和更新操作：O(log m)
 * - 总时间复杂度：O(n log n + n log m) = O(n log n)，因为 m ≤ n
 *
```

 * 空间复杂度分析：
 * - 线段树空间：O(m)
 * - 离散化数组：O(m)
 * - 总空间复杂度：O(m) = O(n)
 *
 * 本题最优解：线段树是本题的最优解之一，另外也可以使用树状数组（Fenwick Tree）实现，时间复杂度相同。
 */
public class Code12_CreateSortedArrayThroughInstructions {
    // 取模常量
    private static final int MOD = 1000000007;

    /**
     * 线段树节点类
     */
    private static class SegmentTree {
        private int[] tree; // 线段树数组
        private int n;   // 原始数据长度

        /**
         * 构造线段树
         * @param size 离散化后的值域大小
         */
        public SegmentTree(int size) {
            n = 1;
            // 计算大于等于 size 的最小 2 的幂次
            while (n < size) {
                n <<= 1;
            }
            // 初始化线段树数组，大小为 2*n
            tree = new int[2 * n];
        }

        /**
         * 更新线段树中的某个位置的值
         * @param idx 离散化后的值对应的索引
         * @param delta 要增加的值（这里是 1）
         */
        public void update(int idx, int delta) {
            idx += n; // 转换为线段树叶子节点索引
            tree[idx] += delta;
            // 向上更新父节点
            for (int i = idx >> 1; i >= 1; i >>= 1) {

```
            tree[i] = tree[2 * i] + tree[2 * i + 1];
        }
    }

    /**
     * 查询区间[0, idx]的和
     * @param idx 离散化后的值对应的索引
     * @return 区间和
     */
    public int query(int idx) {
        if (idx < 0) return 0;
        idx = Math.min(idx, n - 1); // 防止越界
        int res = 0;
        int l = n;       // 左边界（线段树叶子节点索引）
        int r = n + idx; // 右边界（线段树叶子节点索引）

        // 区间查询
        while (l <= r) {
            // 如果 l 是右孩子
            if ((l & 1) == 1) {
                res += tree[l];
                l++;
            }
            // 如果 r 是左孩子
            if ((r & 1) == 0) {
                res += tree[r];
                r--;
            }
            l >>= 1;
            r >>= 1;
        }
        return res;
    }
}

/**
 * 计算创建有序数组的最小代价
 * @param instructions 指令数组
 * @return 总最小代价
 */
public int createSortedArray(int[] instructions) {
    // 离散化处理
    Set<Integer> uniqueVals = new HashSet<>();
```

```
        for (int num : instructions) {
            uniqueVals.add(num);
        }

        // 转换为有序数组并排序
        Integer[] sortedVals = uniqueVals.toArray(new Integer[0]);
        Arrays.sort(sortedVals);

        // 创建值到离散化索引的映射
        int[] valueToIndex = new int[100001]; // 题目中说指令中的元素不超过 10^5
        for (int i = 0; i < sortedVals.length; i++) {
            valueToIndex[sortedVals[i]] = i;
        }

        // 构建线段树
        SegmentTree segmentTree = new SegmentTree(sortedVals.length);
        long totalCost = 0; // 使用 long 避免溢出

        // 处理每个指令
        for (int i = 0; i < instructions.length; i++) {
            int value = instructions[i];
            int idx = valueToIndex[value];

            // 计算左边比当前元素小的个数（即前缀和）
            int smallerCount = segmentTree.query(idx - 1);

            // 计算右边比当前元素大的个数（总元素数减去到当前索引的前缀和）
            int largerCount = i - segmentTree.query(idx);

            // 取较小值作为当前操作的代价
            totalCost += Math.min(smallerCount, largerCount);
            totalCost %= MOD; // 取模

            // 更新线段树，将当前元素的计数加 1
            segmentTree.update(idx, 1);
        }

        return (int) totalCost;
    }

    /**
     * 测试方法
     */
```

```
    public static void main(String[] args) {
        Code12_CreateSortedArrayThroughInstructions solution = new
Code12_CreateSortedArrayThroughInstructions();

        // 测试用例 1
        int[] instructions1 = {1, 5, 6, 2};
        System.out.println("测试用例 1 结果: " + solution.createSortedArray(instructions1)); // 预
期输出: 1

        // 测试用例 2
        int[] instructions2 = {1, 2, 3, 6, 5, 4};
        System.out.println("测试用例 2 结果: " + solution.createSortedArray(instructions2)); // 预
期输出: 3

        // 测试用例 3
        int[] instructions3 = {1, 3, 3, 3, 2, 4, 2, 1, 2};
        System.out.println("测试用例 3 结果: " + solution.createSortedArray(instructions3)); // 预
期输出: 4
    }
}
```

==========

文件: Code12_CreateSortedArrayThroughInstructions.py

==========

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

"""
LeetCode 1649. Create Sorted Array through Instructions (通过指令创建有序数组)
题目链接：https://leetcode.com/problems/create-sorted-array-through-instructions/

题目描述：
给你一个整数数组 instructions，你需要根据 instructions 中的元素创建一个有序数组。
一开始数组为空。你需要依次读取 instructions 中的元素，并将它插入到有序数组中的正确位置。
每次插入操作的代价是以下两者的较小值：
1. 有多少个元素严格小于 instructions[i]（左边）
2. 有多少个元素严格大于 instructions[i]（右边）
返回插入所有元素的总最小代价。由于答案可能很大，请返回它对 10^9 + 7 取模的结果。

示例：
输入：instructions = [1,5,6,2]
输出：1
```

解释：插入 1 时，数组为空，代价为 0。
插入 5 时，左边有 1 个元素比 5 小，右边没有元素，代价为 min(1, 0) = 0。
插入 6 时，左边有 2 个元素比 6 小，右边没有元素，代价为 min(2, 0) = 0。
插入 2 时，左边有 1 个元素比 2 小，右边有 2 个元素比 2 大，代价为 min(1, 2) = 1。
总代价为 0 + 0 + 0 + 1 = 1

解题思路：
这道题可以使用离散化线段树来解决。我们需要高效地统计数组中有多少元素小于当前元素，以及有多少元素大于当前元素。
具体步骤：
1. 离散化处理：将指令数组中的所有元素进行排序去重，得到每个元素的排名（离散化值）
2. 构建线段树：维护每个值出现的次数
3. 对于每个指令：
   a. 查询当前小于该元素的个数（即离散化后值减 1 的前缀和）
   b. 查询当前大于该元素的个数（即总元素数减去离散化后值的前缀和）
   c. 计算当前代价并累加到结果中
   d. 更新线段树，将该元素的计数加 1

时间复杂度分析：
- 离散化：O(n log n)，其中 n 是指令数组的长度
- 线段树构建：O(m)，其中 m 是离散化后的不同元素个数
- 每个查询和更新操作：O(log m)
- 总时间复杂度：O(n log n + n log m) = O(n log n)，因为 m ≤ n

空间复杂度分析：
- 线段树空间：O(m)
- 离散化数组：O(m)
- 总空间复杂度：O(m) = O(n)

本题最优解：线段树是本题的最优解之一，另外也可以使用树状数组（Fenwick Tree）实现，时间复杂度相同。
"""

```
MOD = 10**9 + 7

class SegmentTree:
    """线段树类，用于维护区间和"""

    def __init__(self, size):
        """
        初始化线段树

        Args:
```

```
        size: 离散化后的值域大小
    """
    self.n = 1
    # 计算大于等于 size 的最小 2 的幂次
    while self.n < size:
        self.n <<= 1
    # 初始化线段树数组，大小为 2*n
    self.tree = [0] * (2 * self.n)

def update(self, idx, delta):
    """
    更新线段树中的某个位置的值

    Args:
        idx: 离散化后的值对应的索引
        delta: 要增加的值（这里是 1）
    """
    idx += self.n  # 转换为线段树叶子节点索引
    self.tree[idx] += delta
    # 向上更新父节点
    i = idx >> 1
    while i >= 1:
        self.tree[i] = self.tree[2 * i] + self.tree[2 * i + 1]
        i >>= 1

def query(self, idx):
    """
    查询区间[0, idx]的和

    Args:
        idx: 离散化后的值对应的索引

    Returns:
        区间和
    """
    if idx < 0:
        return 0
    idx = min(idx, self.n - 1)  # 防止越界
    res = 0
    l = self.n       # 左边界（线段树叶子节点索引）
    r = self.n + idx # 右边界（线段树叶子节点索引）

    # 区间查询
```

```
        while l <= r:
            # 如果 l 是右孩子
            if l % 2 == 1:
                res += self.tree[l]
                l += 1
            # 如果 r 是左孩子
            if r % 2 == 0:
                res += self.tree[r]
                r -= 1
            l >>= 1
            r >>= 1

        return res

class Solution:
    """解决方案类"""

    def createSortedArray(self, instructions):
        """
        计算创建有序数组的最小代价

        Args:
            instructions: 指令数组

        Returns:
            总最小代价
        """
        # 离散化处理
        unique_vals = sorted(set(instructions))
        m = len(unique_vals)

        # 创建值到离散化索引的映射字典
        value_to_index = {val: i for i, val in enumerate(unique_vals)}

        # 构建线段树
        segment_tree = SegmentTree(m)
        total_cost = 0

        # 处理每个指令
        for i, value in enumerate(instructions):
            idx = value_to_index[value]

            # 计算左边比当前元素小的个数（即前缀和）
```

```
            smaller_count = segment_tree.query(idx - 1)

            # 计算右边比当前元素大的个数（总元素数减去到当前索引的前缀和）
            larger_count = i - segment_tree.query(idx)

            # 取较小值作为当前操作的代价
            total_cost = (total_cost + min(smaller_count, larger_count)) % MOD

            # 更新线段树，将当前元素的计数加 1
            segment_tree.update(idx, 1)

        return total_cost

# 测试代码
if __name__ == "__main__":
    solution = Solution()

    # 测试用例 1
    instructions1 = [1, 5, 6, 2]
    print("测试用例 1 结果:", solution.createSortedArray(instructions1))  # 预期输出: 1

    # 测试用例 2
    instructions2 = [1, 2, 3, 6, 5, 4]
    print("测试用例 2 结果:", solution.createSortedArray(instructions2))  # 预期输出: 3

    # 测试用例 3
    instructions3 = [1, 3, 3, 3, 2, 4, 2, 1, 2]
    print("测试用例 3 结果:", solution.createSortedArray(instructions3))  # 预期输出: 4

"""
```

性能优化说明：

1. 使用集合去重然后排序，避免了重复元素的处理
2. 在 Python 中使用字典进行值到索引的映射，提高了查询效率
3. 线段树的实现采用了非递归的方式，在 Python 中避免了递归可能带来的栈溢出问题
4. 使用位移运算代替乘法和除法，提升了位运算效率

语言特性差异：

1. Python 中的整数没有大小限制，不需要像 Java 和 C++那样担心溢出问题，但在取模时仍需注意
2. Python 的字典比 Java 的数组映射更灵活，但在大规模数据时可能稍慢
3. Python 的列表操作比 C++的 vector 更简洁，但性能略低

工程化考量：

1. 代码结构清晰，类和方法的命名符合 Python 的 PEP8 规范

2. 包含了详细的文档字符串（docstring），方便其他开发者理解和使用
3. 添加了测试用例，验证代码的正确性
4. 考虑了边界情况，如 idx 为负数的情况
5. 使用常量 MOD 定义模数，提高了代码的可维护性
"""

==========

文件: Code13_HashCollision.cpp
==========
```
// 哈希冲突问题 - 分块算法实现 (C++版本)
// 题目来源: https://www.luogu.com.cn/problem/P3396
// 题目大意: 给定一个长度为 n 的数组 arr，支持两种操作：
// 1. 查询操作 A x y: 查询所有满足 i % x == y 的位置 i 对应的 arr[i]之和
// 2. 更新操作 C x y: 将 arr[x]的值更新为 y
// 约束条件: 1 <= n、m <= 1.5 * 10^5
//
// 解题思路:
// 1. 对于 x <= sqrt(n)的情况，预处理 dp[x][y]的值
// 2. 对于 x > sqrt(n)的情况，直接暴力计算
// 3. 更新操作时，同时更新预处理结果
//
// 时间复杂度分析:
// - 预处理: O(n * sqrt(n))
// - 查询: O(1) 对于 x <= sqrt(n)，O(n/x) 对于 x > sqrt(n)
// - 更新: O(sqrt(n))
//
// 空间复杂度: O(n + sqrt(n)^2) = O(n)
//
// 工程化考量:
// 1. 异常处理: 验证输入参数的有效性
// 2. 性能优化: 使用分块思想平衡预处理和查询的开销
// 3. 边界处理: 处理 x=0 或 y>=x 等边界情况
// 4. 内存管理: 合理设置数组大小避免内存溢出

#include <iostream>
#include <vector>
#include <cmath>
#include <stdexcept>

using namespace std;

class HashCollision {
```

```
private:
    // 定义最大数组长度和块大小
    static const int MAXN = 150001;
    static const int MAXB = 401;

    // n: 数组长度, m: 操作次数, blen: 块大小
    int n, blen;

    // arr: 原始数组
    vector<int> arr;

    // dp[x][y]: 存储所有满足 i % x == y 的位置 i 对应的 arr[i]之和 (预处理结果)
    // 只对 x <= sqrt(n) 的情况进行预处理，以节省空间和时间
    vector<vector<long long>> dp;

public:
    /**
     * 构造函数
     * @param size 数组大小
     */
    HashCollision(int size) : n(size), arr(size + 1, 0) {
        // 计算块大小，通常选择 sqrt(n)
        blen = (int)sqrt(n);

        // 初始化 dp 数组
        dp.resize(blen + 1, vector<long long>(blen + 1, 0));
    }

    /**
     * 设置数组初始值
     * @param values 初始值数组
     */
    void setArray(const vector<int>& values) {
        if (values.size() != n) {
            throw invalid_argument("初始值数组大小不匹配");
        }

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

        // 进行预处理
        prepare();
```

```
}

/**
 * 查询操作 A x y
 * 查询所有满足 i % x == y 的位置 i 对应的 arr[i]之和
 * @param x 除数，必须大于 0
 * @param y 余数，必须满足 0 <= y < x
 * @return 满足条件的位置对应的元素之和
 * @throws invalid_argument 如果 x <= 0 或 y < 0 或 y >= x
 */
long long query(int x, int y) {
    // 参数验证
    if (x <= 0) {
        throw invalid_argument("除数 x 必须大于 0");
    }
    if (y < 0 || y >= x) {
        throw invalid_argument("余数 y 必须满足 0 <= y < x");
    }

    // 如果 x 小于等于块大小，则直接返回预处理结果
    if (x <= blen) {
        return dp[x][y];
    }

    // 否则暴力计算（适用于 x 较大的情况）
    long long ans = 0;
    for (int i = y; i <= n; i += x) {
        ans += arr[i];
    }
    return ans;
}

/**
 * 更新操作 C x y
 * 将 arr[x]的值更新为 y，并更新相关的预处理结果
 * @param i 要更新的位置，必须满足 1 <= i <= n
 * @param v 新的值
 * @throws invalid_argument 如果位置 i 超出有效范围
 */
void update(int i, int v) {
    // 参数验证
    if (i < 1 || i > n) {
        throw invalid_argument("位置 i 必须在 1 到 n 之间");
```

```
    }

    // 计算值的变化量
    int delta = v - arr[i];
    // 更新原数组
    arr[i] = v;

    // 更新所有相关的预处理结果
    // 只需要更新 x <= sqrt(n)的情况，因为这些被预处理了
    for (int x = 1; x <= blen; x++) {
        dp[x][i % x] += delta;
    }
}

/**
 * 预处理函数
 * 对于所有 x <= sqrt(n)的情况，预处理 dp[x][y]的值
 */
void prepare() {
    // 初始化 dp 数组为 0
    for (int x = 1; x <= blen; x++) {
        for (int y = 0; y < x; y++) {
            dp[x][y] = 0;
        }
    }

    // 对于每个 x <= sqrt(n)，计算所有 y 对应的 dp[x][y]值
    for (int x = 1; x <= blen; x++) {
        for (int i = 1; i <= n; i++) {
            // i % x 表示位置 i 对 x 取余的结果
            // dp[x][i % x] 累加 arr[i]的值
            dp[x][i % x] += arr[i];
        }
    }
}

/**
 * 获取数组当前状态
 * @return 数组内容字符串
 */
string getStatus() const {
    string result = "数组状态: [";
    for (int i = 1; i <= min(n, 10); i++) {
```

```
            result += to_string(arr[i]);
            if (i < min(n, 10)) result += ", ";
        }
        if (n > 10) result += "...";
        result += "]";
        return result;
    }

    /**
     * 获取预处理状态
     * @return 预处理状态字符串
     */
    string getPreprocessStatus() const {
        string result = "预处理状态 (x <= " + to_string(blen) + "):\n";
        for (int x = 1; x <= min(blen, 5); x++) {
            result += "x=" + to_string(x) + ": [";
            for (int y = 0; y < x; y++) {
                result += to_string(dp[x][y]);
                if (y < x - 1) result += ", ";
            }
            result += "]\n";
        }
        if (blen > 5) result += "...\n";
        return result;
    }
};

/**
 * 单元测试函数
 */
void testHashCollision() {
    cout << "=== 哈希冲突算法单元测试 ===" << endl;

    // 测试1: 基本功能测试
    cout << "测试1: 基本功能测试" << endl;
    HashCollision hc(10);

    // 设置初始数组
    vector<int> values = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    hc.setArray(values);

    // 验证查询结果
    long long result1 = hc.query(3, 0); // 3 + 6 + 9 = 18
```

```
    long long result2 = hc.query(3, 1); // 1 + 4 + 7 + 10 = 22
    long long result3 = hc.query(3, 2); // 2 + 5 + 8 = 15

    if (result1 == 18 && result2 == 22 && result3 == 15) {
        cout << "查询测试通过" << endl;
    } else {
        cout << "查询测试失败" << endl;
    }

    // 验证更新结果
    hc.update(5, 50); // 将位置 5 的值从 5 改为 50
    long long result4 = hc.query(3, 1); // 1 + 4 + 7 + 10 + (50 - 5) = 22 + 45 = 67

    if (result4 == 67) {
        cout << "更新测试通过" << endl;
    } else {
        cout << "更新测试失败" << endl;
    }

    // 测试 2: 异常处理测试
    cout << "\n 测试 2: 异常处理测试" << endl;
    try {
        hc.query(0, 0); // 应该抛出异常
        cout << "异常处理测试失败" << endl;
    } catch (const invalid_argument& e) {
        cout << "异常处理测试通过: " << e.what() << endl;
    }

    cout << hc.getStatus() << endl;
    cout << hc.getPreprocessStatus() << endl;

    cout << "=== 单元测试完成 ===" << endl;
}

/**
 * 性能测试函数
 */
void performanceTest() {
    cout << "=== 哈希冲突算法性能测试 ===" << endl;

    int n = 100000;
    int m = 10000;
```

```
    HashCollision hc(n);

    // 初始化数组
    vector<int> values(n);
    for (int i = 0; i < n; i++) {
        values[i] = i + 1;
    }
    hc.setArray(values);

    // 测试查询性能
    clock_t start = clock();

    long long total = 0;
    for (int i = 0; i < m; i++) {
        int x = (i % 100) + 1; // x在1-100之间
        int y = i % x;
        total += hc.query(x, y);
    }

    clock_t end = clock();
    double duration = (double)(end - start) / CLOCKS_PER_SEC * 1000;

    cout << "性能测试结果:" << endl;
    cout << "数据规模: n=" << n << ", 操作次数: m=" << m << endl;
    cout << "总查询时间: " << duration << " 毫秒" << endl;
    cout << "平均查询时间: " << duration / m << " 毫秒/次" << endl;
    cout << "查询吞吐量: " << (double)m / (duration / 1000.0) << " 次/秒" << endl;
    cout << "查询结果总和: " << total << endl;

    cout << "=== 性能测试完成 ===" << endl;
}

int main() {
    // 运行单元测试
    testHashCollision();

    // 运行性能测试
    performanceTest();

    // 演示示例
    cout << "=== 哈希冲突算法演示 ===" << endl;

    HashCollision demo(20);
```

```
    vector<int> demoValues(20);
    for (int i = 0; i < 20; i++) {
        demoValues[i] = (i + 1) * 10; // 10, 20, 30, ..., 200
    }
    demo.setArray(demoValues);

    cout << demo.getStatus() << endl;

    // 演示查询操作
    cout << "\n查询演示:" << endl;
    cout << "查询所有位置为3的倍数的元素之和 (x=3, y=0): " << demo.query(3, 0) << endl;
    cout << "查询所有位置除以4余1的元素之和 (x=4, y=1): " << demo.query(4, 1) << endl;
    cout << "查询所有位置除以5余2的元素之和 (x=5, y=2): " << demo.query(5, 2) << endl;

    // 演示更新操作
    cout << "\n更新演示:" << endl;
    cout << "更新前位置5的值: " << demo.query(1, 4) << endl; // 查询单个位置
    demo.update(5, 999);
    cout << "更新后位置5的值: " << demo.query(1, 4) << endl;
    cout << "更新后所有位置为3的倍数的元素之和: " << demo.query(3, 0) << endl;

    return 0;
}

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

文件: Code13_HashCollision.java
==========================================
package class111;

// 哈希冲突问题 - 分块算法实现 (Java版本)
// 题目来源: https://www.luogu.com.cn/problem/P3396
// 题目大意: 给定一个长度为n的数组arr，支持两种操作：
// 1. 查询操作 A x y: 查询所有满足 i % x == y 的位置i对应的arr[i]之和
// 2. 更新操作 C x y: 将arr[x]的值更新为y
// 约束条件: 1 <= n、m <= 1.5 * 10^5
//
// 解题思路:
// 1. 对于x <= sqrt(n)的情况，预处理dp[x][y]的值
// 2. 对于x > sqrt(n)的情况，直接暴力计算
// 3. 更新操作时，同时更新预处理结果
//
// 时间复杂度分析:
```

```
// - 预处理: O(n * sqrt(n))
// - 查询: O(1) 对于 x <= sqrt(n), O(n/x) 对于 x > sqrt(n)
// - 更新: O(sqrt(n))
//
// 空间复杂度: O(n + sqrt(n)^2) = O(n)
//
// 工程化考量:
// 1. 异常处理: 验证输入参数的有效性
// 2. 性能优化: 使用分块思想平衡预处理和查询的开销
// 3. 边界处理: 处理 x=0 或 y>=x 等边界情况
// 4. 内存管理: 合理设置数组大小避免内存溢出

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

public class Code13_HashCollision {

    // 定义最大数组长度和块大小
    public static int MAXN = 150001;
    public static int MAXB = 401;

    // n: 数组长度, m: 操作次数, blen: 块大小
    public static int n, m, blen;

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

    // dp[x][y]: 存储所有满足 i % x == y 的位置 i 对应的 arr[i]之和 (预处理结果)
    // 只对 x <= sqrt(n) 的情况进行预处理，以节省空间和时间
    public static long[][] dp = new long[MAXB][MAXB];

    /**
     * 查询操作 A x y
     * 查询所有满足 i % x == y 的位置 i 对应的 arr[i]之和
     * @param x 除数，必须大于 0
     * @param y 余数，必须满足 0 <= y < x
     * @return 满足条件的位置对应的元素之和
     * @throws IllegalArgumentException 如果 x <= 0 或 y < 0 或 y >= x
     */
    public static long query(int x, int y) {
        // 参数验证
```

```
    if (x <= 0) {
        throw new IllegalArgumentException("除数 x 必须大于 0");
    }
    if (y < 0 || y >= x) {
        throw new IllegalArgumentException("余数 y 必须满足 0 <= y < x");
    }

    // 如果 x 小于等于块大小，则直接返回预处理结果
    if (x <= blen) {
        return dp[x][y];
    }

    // 否则暴力计算（适用于 x 较大的情况）
    long ans = 0;
    for (int i = y; i <= n; i += x) {
        ans += arr[i];
    }
    return ans;
}

/**
 * 更新操作 C x y
 * 将 arr[x]的值更新为 y，并更新相关的预处理结果
 * @param i 要更新的位置，必须满足 1 <= i <= n
 * @param v 新的值
 * @throws IllegalArgumentException 如果位置 i 超出有效范围
 */
public static void update(int i, int v) {
    // 参数验证
    if (i < 1 || i > n) {
        throw new IllegalArgumentException("位置 i 必须在 1 到 n 之间");
    }

    // 计算值的变化量
    int delta = v - arr[i];
    // 更新原数组
    arr[i] = v;

    // 更新所有相关的预处理结果
    // 只需要更新 x <= sqrt(n)的情况，因为这些被预处理了
    for (int x = 1; x <= blen; x++) {
        dp[x][i % x] += delta;
    }
```

```
}

/**
 * 预处理函数
 * 对于所有 x <= sqrt(n)的情况，预处理 dp[x][y]的值
 */
public static void prepare() {
    // 计算块大小，通常选择 sqrt(n)
    blen = (int) Math.sqrt(n);

    // 初始化 dp 数组为 0（Java 中默认初始化为 0）
    // 对于每个 x <= sqrt(n)，计算所有 y 对应的 dp[x][y]值
    for (int x = 1; x <= blen; x++) {
        for (int i = 1; i <= n; i++) {
            // i % x 表示位置 i 对 x 取余的结果
            // dp[x][i % x] 累加 arr[i]的值
            dp[x][i % x] += arr[i];
        }
    }
}

/**
 * 单元测试方法
 * 验证算法的正确性
 */
public static void test() {
    // 测试用例 1: 小规模数据测试
    n = 10;
    for (int i = 1; i <= n; i++) {
        arr[i] = i;
    }
    prepare();

    // 验证查询结果
    assert query(3, 0) == (3 + 6 + 9) : "查询测试失败";
    assert query(3, 1) == (1 + 4 + 7 + 10) : "查询测试失败";
    assert query(3, 2) == (2 + 5 + 8) : "查询测试失败";

    // 验证更新结果
    update(5, 50);
    assert query(3, 1) == (1 + 4 + 7 + 10) + (50 - 5) : "更新测试失败";

    System.out.println("所有测试用例通过！");
```

```
}

public static void main(String[] args) throws IOException {
    // 如果传入参数包含"test"，则运行单元测试
    if (args.length > 0 && "test".equals(args[0])) {
        test();
        return;
    }

    FastReader in = new FastReader();
    PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));

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

    // 读取初始数组
    for (int i = 1; i <= n; i++) {
        arr[i] = in.nextInt();
    }

    // 进行预处理
    prepare();

    // 处理 m 次操作
    char op;
    int x, y;
    for (int i = 1; i <= m; i++) {
        op = in.nextChar();
        x = in.nextInt();
        y = in.nextInt();

        try {
            if (op == 'A') {
                // 查询操作
                out.println(query(x, y));
            } else {
                // 更新操作
                update(x, y);
            }
        } catch (IllegalArgumentException e) {
            out.println("错误: " + e.getMessage());
        }
```

```
    }

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

// 高效读取工具类，用于加快输入输出速度
static class FastReader {
    final private int BUFFER_SIZE = 1 << 16;
    private final InputStream in;
    private final byte[] buffer;
    private int ptr, len;

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

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

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

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

```
                c = readByte();
            }
            return ans;
        }

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

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

}
```

==========

文件: Code13_HashCollision.py

==========

```
# 哈希冲突问题 - 分块算法实现 (Python 版本)
# 题目来源: https://www.luogu.com.cn/problem/P3396
# 题目大意: 给定一个长度为 n 的数组 arr，支持两种操作：
# 1. 查询操作 A x y: 查询所有满足 i % x == y 的位置 i 对应的 arr[i]之和
# 2. 更新操作 C x y: 将 arr[x]的值更新为 y
# 约束条件: 1 <= n、m <= 1.5 * 10^5
#
# 解题思路:
# 1. 对于 x <= sqrt(n)的情况，预处理 dp[x][y]的值
# 2. 对于 x > sqrt(n)的情况，直接暴力计算
```

```
# 3. 更新操作时，同时更新预处理结果
#
# 时间复杂度分析:
# - 预处理: O(n * sqrt(n))
# - 查询: O(1) 对于 x <= sqrt(n)，O(n/x) 对于 x > sqrt(n)
# - 更新: O(sqrt(n))
#
# 空间复杂度: O(n + sqrt(n)^2) = O(n)
#
# 工程化考量:
# 1. 异常处理: 验证输入参数的有效性
# 2. 性能优化: 使用分块思想平衡预处理和查询的开销
# 3. 边界处理: 处理 x=0 或 y>=x 等边界情况
# 4. 内存管理: 合理设置数组大小避免内存溢出

import math

class HashCollision:
    def __init__(self, size):
        """
        构造函数
        :param size: 数组大小
        """
        self.n = size
        # 计算块大小，通常选择 sqrt(n)
        self.blen = int(math.sqrt(self.n))

        # 初始化数组
        self.arr = [0] * (self.n + 1)

        # 初始化 dp 数组
        # dp[x][y]: 存储所有满足 i % x == y 的位置 i 对应的 arr[i]之和
        self.dp = [[0] * (self.blen + 1) for _ in range(self.blen + 1)]

    def set_array(self, values):
        """
        设置数组初始值
        :param values: 初始值数组
        :raises ValueError: 如果初始值数组大小不匹配
        """
        if len(values) != self.n:
            raise ValueError("初始值数组大小不匹配")
```

```
    for i in range(1, self.n + 1):
        self.arr[i] = values[i - 1]

    # 进行预处理
    self.prepare()

def query(self, x, y):
    """
    查询操作 A x y
    查询所有满足 i % x == y 的位置 i 对应的 arr[i]之和
    :param x: 除数，必须大于 0
    :param y: 余数，必须满足 0 <= y < x
    :return: 满足条件的位置对应的元素之和
    :raises ValueError: 如果 x <= 0 或 y < 0 或 y >= x
    """
    # 参数验证
    if x <= 0:
        raise ValueError("除数 x 必须大于 0")
    if y < 0 or y >= x:
        raise ValueError(f"余数 y 必须满足 0 <= y < x, 但 y={y}, x={x}")

    # 如果 x 小于等于块大小，则直接返回预处理结果
    if x <= self.blen:
        return self.dp[x][y]

    # 否则暴力计算（适用于 x 较大的情况）
    ans = 0
    i = y
    while i <= self.n:
        ans += self.arr[i]
        i += x
    return ans

def update(self, i, v):
    """
    更新操作 C x y
    将 arr[x]的值更新为 y，并更新相关的预处理结果
    :param i: 要更新的位置，必须满足 1 <= i <= n
    :param v: 新的值
    :raises ValueError: 如果位置 i 超出有效范围
    """
    # 参数验证
    if i < 1 or i > self.n:
```

```
            raise ValueError("位置 i 必须在 1 到 n 之间")

        # 计算值的变化量
        delta = v - self.arr[i]
        # 更新原数组
        self.arr[i] = v

        # 更新所有相关的预处理结果
        # 只需要更新 x <= sqrt(n)的情况，因为这些被预处理了
        for x in range(1, self.blen + 1):
            self.dp[x][i % x] += delta

    def prepare(self):
        """
        预处理函数
        对于所有 x <= sqrt(n)的情况，预处理 dp[x][y]的值
        """
        # 初始化 dp 数组为 0
        for x in range(1, self.blen + 1):
            for y in range(x):
                self.dp[x][y] = 0

        # 对于每个 x <= sqrt(n)，计算所有 y 对应的 dp[x][y]值
        for x in range(1, self.blen + 1):
            for i in range(1, self.n + 1):
                # i % x 表示位置 i 对 x 取余的结果
                # dp[x][i % x] 累加 arr[i]的值
                self.dp[x][i % x] += self.arr[i]

    def get_status(self):
        """
        获取数组当前状态
        :return: 数组内容字符串
        """
        result = "数组状态: ["
        for i in range(1, min(self.n, 10) + 1):
            result += str(self.arr[i])
            if i < min(self.n, 10):
                result += ", "
        if self.n > 10:
            result += "..."
        result += "]"
        return result
```

```
    def get_preprocess_status(self):
        """
        获取预处理状态
        :return: 预处理状态字符串
        """
        result = f"预处理状态 (x <= {self.blen}):\n"
        for x in range(1, min(self.blen, 5) + 1):
            result += f"x={x}: ["
            for y in range(x):
                result += str(self.dp[x][y])
                if y < x - 1:
                    result += ", "
            result += "]\n"
        if self.blen > 5:
            result += "...\n"
        return result

def test_hash_collision():
    """
    单元测试函数
    """
    print("=== 哈希冲突算法单元测试 ===")

    # 测试 1: 基本功能测试
    print("测试 1: 基本功能测试")
    hc = HashCollision(10)

    # 设置初始数组
    values = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
    hc.set_array(values)

    # 验证查询结果
    result1 = hc.query(3, 0)  # 3 + 6 + 9 = 18
    result2 = hc.query(3, 1)  # 1 + 4 + 7 + 10 = 22
    result3 = hc.query(3, 2)  # 2 + 5 + 8 = 15

    if result1 == 18 and result2 == 22 and result3 == 15:
        print("查询测试通过")
    else:
        print("查询测试失败")

    # 验证更新结果
```

```
    hc.update(5, 50)  # 将位置 5 的值从 5 改为 50
    result4 = hc.query(3, 1)  # 1 + 4 + 7 + 10 + (50 - 5) = 22 + 45 = 67

    if result4 == 67:
        print("更新测试通过")
    else:
        print("更新测试失败")

    # 测试 2: 异常处理测试
    print("\n 测试 2: 异常处理测试")
    try:
        hc.query(0, 0)  # 应该抛出异常
        print("异常处理测试失败")
    except ValueError as e:
        print(f"异常处理测试通过: {e}")

    print(hc.get_status())
    print(hc.get_preprocess_status())

    print("=== 单元测试完成 ===")

def performance_test():
    """
    性能测试函数
    """
    import time

    print("=== 哈希冲突算法性能测试 ===")

    n = 100000
    m = 10000

    hc = HashCollision(n)

    # 初始化数组
    values = [i + 1 for i in range(n)]
    hc.set_array(values)

    # 测试查询性能
    start_time = time.time()

    total = 0
    for i in range(m):
```

```
        x = (i % 100) + 1  # x在1-100之间
        y = i % x
        total += hc.query(x, y)

    end_time = time.time()
    duration = (end_time - start_time) * 1000  # 转换为毫秒

    print("性能测试结果:")
    print(f"数据规模: n={n}, 操作次数: m={m}")
    print(f"总查询时间: {duration:.2f} 毫秒")
    print(f"平均查询时间: {duration / m:.4f} 毫秒/次")
    print(f"查询吞吐量: {m / (duration / 1000):.2f} 次/秒")
    print(f"查询结果总和: {total}")

    print("=== 性能测试完成 ===")

if __name__ == "__main__":
    # 运行单元测试
    test_hash_collision()

    # 运行性能测试
    performance_test()

    # 演示示例
    print("=== 哈希冲突算法演示 ===")

    demo = HashCollision(20)
    demo_values = [(i + 1) * 10 for i in range(20)]  # 10, 20, 30, ..., 200
    demo.set_array(demo_values)

    print(demo.get_status())

    # 演示查询操作
    print("\n查询演示:")
    print(f"查询所有位置为3的倍数的元素之和 (x=3, y=0): {demo.query(3, 0)}")
    print(f"查询所有位置除以4余1的元素之和 (x=4, y=1): {demo.query(4, 1)}")
    print(f"查询所有位置除以5余2的元素之和 (x=5, y=2): {demo.query(5, 2)}")

    # 演示更新操作
    print("\n更新演示:")
    print(f"更新前位置5的值: {demo.query(1, 4)}")  # 查询单个位置
    demo.update(5, 999)
    print(f"更新后位置5的值: {demo.query(1, 4)}")
```

```
    print(f"更新后所有位置为3的倍数的元素之和: {demo.query(3, 0)}")

==========

文件: Code14_ConsistentHashing.cpp
==========
// 一致性哈希算法实现 (C++版本)
// 题目来源: 分布式系统设计面试题
// 应用场景: 负载均衡、分布式缓存、分布式存储系统
// 题目描述: 实现一致性哈希算法，支持节点的动态增删和虚拟节点技术
//
// 解题思路:
// 1. 使用哈希环存储节点和虚拟节点
// 2. 使用虚拟节点技术解决数据分布不均问题
// 3. 支持节点的动态添加和删除
// 4. 实现高效的数据查找和节点定位
//
// 时间复杂度分析:
// - 添加节点: O(k)，其中k是虚拟节点数量
// - 删除节点: O(k)
// - 查找节点: O(log n)，其中n是节点总数
//
// 空间复杂度: O(n * k)，其中n是物理节点数，k是每个节点的虚拟节点数
//
// 工程化考量:
// 1. 异常处理: 验证节点和数据的有效性
// 2. 性能优化: 使用TreeMap实现高效的区间查找
// 3. 负载均衡: 虚拟节点技术确保数据均匀分布
// 4. 容错性: 支持节点的动态增删，最小化数据迁移

#include <iostream>
#include <map>
#include <set>
#include <vector>
#include <string>
#include <functional>
#include <random>
#include <algorithm>
#include <stdexcept>
#include <chrono>

using namespace std;
```

```
class ConsistentHashing {
private:
    // 哈希环，存储虚拟节点到物理节点的映射
    map<int, string> hashRing;

    // 物理节点集合
    set<string> physicalNodes;

    // 每个物理节点的虚拟节点数量
    int virtualNodeCount;

    /**
     * 哈希函数 - 使用FNV-1a算法
     * @param str 输入字符串
     * @return 哈希值
     */
    int hash(const string& str) {
        const int FNV_OFFSET_BASIS = 0x811C9DC5;
        const int FNV_PRIME = 0x01000193;

        int hash = FNV_OFFSET_BASIS;
        for (char c : str) {
            hash ^= (c & 0xff);
            hash *= FNV_PRIME;
        }

        // 确保哈希值为正数
        return hash & 0x7fffffff;
    }

public:
    /**
     * 构造函数
     * @param virtualNodeCount 每个物理节点的虚拟节点数量
     */
    ConsistentHashing(int virtualNodeCount = 3) : virtualNodeCount(virtualNodeCount) {
        if (virtualNodeCount <= 0) {
            throw invalid_argument("虚拟节点数量必须大于0");
        }
    }

    /**
     * 添加物理节点
```

```
     * @param node 物理节点名称
     * @throws invalid_argument 如果节点名为空或已存在
     */
    void addNode(const string& node) {
        if (node.empty()) {
            throw invalid_argument("节点名不能为空");
        }
        if (physicalNodes.find(node) != physicalNodes.end()) {
            throw invalid_argument("节点 " + node + " 已存在");
        }

        physicalNodes.insert(node);

        // 为物理节点创建虚拟节点
        for (int i = 0; i < virtualNodeCount; i++) {
            string virtualNode = node + "#" + to_string(i);
            int nodeHash = hash(virtualNode);
            hashRing[nodeHash] = node;
        }

        cout << "添加节点: " << node << ", 虚拟节点数: " << virtualNodeCount << endl;
    }

    /**
     * 删除物理节点
     * @param node 物理节点名称
     * @throws invalid_argument 如果节点不存在
     */
    void removeNode(const string& node) {
        if (physicalNodes.find(node) == physicalNodes.end()) {
            throw invalid_argument("节点 " + node + " 不存在");
        }

        physicalNodes.erase(node);

        // 删除该物理节点的所有虚拟节点
        auto it = hashRing.begin();
        while (it != hashRing.end()) {
            if (it->second == node) {
                it = hashRing.erase(it);
            } else {
                ++it;
            }
```

```
    }

    cout << "删除节点: " << node << endl;
}

/**
 * 根据键查找对应的物理节点
 * @param key 数据键
 * @return 负责该键的物理节点
 * @throws invalid_argument 如果键为空或哈希环为空
 */
string getNode(const string& key) {
    if (key.empty()) {
        throw invalid_argument("键不能为空");
    }
    if (hashRing.empty()) {
        throw invalid_argument("哈希环为空，请先添加节点");
    }

    int keyHash = hash(key);

    // 在哈希环上顺时针查找第一个大于等于该哈希值的节点
    auto it = hashRing.lower_bound(keyHash);

    // 如果没找到，则返回环上的第一个节点（环状结构）
    if (it == hashRing.end()) {
        it = hashRing.begin();
    }

    return it->second;
}

/**
 * 获取哈希环的状态信息
 * @return 哈希环状态字符串
 */
string getStatus() const {
    string result;
    result += "一致性哈希环状态:\n";
    result += "物理节点数: " + to_string(physicalNodes.size()) + "\n";
    result += "虚拟节点数: " + to_string(hashRing.size()) + "\n";

    // 物理节点列表
```

```
    result += "物理节点列表: ";
    for (const auto& node : physicalNodes) {
        result += node + " ";
    }
    result += "\n";

    // 统计每个物理节点的虚拟节点分布
    map<string, int> nodeDistribution;
    for (const auto& entry : hashRing) {
        nodeDistribution[entry.second]++;
    }

    result += "虚拟节点分布: ";
    for (const auto& entry : nodeDistribution) {
        result += entry.first + ":" + to_string(entry.second) + " ";
    }
    result += "\n";

    return result;
}

/**
 * 负载均衡测试
 * 模拟大量数据分布，检查负载均衡性
 * @param dataCount 测试数据数量
 */
void loadBalanceTest(int dataCount) {
    if (physicalNodes.empty()) {
        cout << "请先添加节点再进行负载均衡测试" << endl;
        return;
    }

    map<string, int> distribution;

    // 初始化分布统计
    for (const auto& node : physicalNodes) {
        distribution[node] = 0;
    }

    // 模拟数据分布
    random_device rd;
    mt19937 gen(rd());
    uniform_int_distribution<> dis(0, 1000000);
```

```
        for (int i = 0; i < dataCount; i++) {
            string key = "key" + to_string(dis(gen));
            string node = getNode(key);
            distribution[node]++;
        }

        // 计算负载均衡指标
        int total = dataCount;
        double average = (double)total / physicalNodes.size();
        double variance = 0.0;

        cout << "负载均衡测试结果 (数据量: " << dataCount << "):" << endl;
        for (const auto& entry : distribution) {
            double deviation = abs(entry.second - average);
            variance += deviation * deviation;
            cout << "节点 " << entry.first << ": " << entry.second
                 << " 数据 (" << (double)entry.second / total * 100 << "%)" << endl;
        }

        double stdDev = sqrt(variance / physicalNodes.size());
        cout << "标准差: " << stdDev << ", 相对标准差: " << stdDev / average * 100 << "%" <<
endl;
    }

    /**
     * 获取物理节点数量
     * @return 物理节点数量
     */
    int getPhysicalNodeCount() const {
        return physicalNodes.size();
    }

    /**
     * 获取虚拟节点数量
     * @return 虚拟节点数量
     */
    int getVirtualNodeCount() const {
        return hashRing.size();
    }
};

/**
```

```
 * 单元测试函数
 */
void testConsistentHashing() {
    cout << "=== 一致性哈希算法单元测试 ===" << endl;

    // 测试 1: 基本功能测试
    cout << "测试 1: 基本功能测试" << endl;
    ConsistentHashing ch(3);

    // 添加节点
    ch.addNode("Node-A");
    ch.addNode("Node-B");
    ch.addNode("Node-C");

    cout << ch.getStatus() << endl;

    // 测试数据分布
    map<string, int> testDistribution;
    vector<string> testKeys = {"user1", "user2", "user3", "data1", "data2", "file1"};

    for (const auto& key : testKeys) {
        string node = ch.getNode(key);
        testDistribution[node]++;
        cout << "键 '" << key << "' 分配到节点: " << node << endl;
    }

    cout << "测试数据分布: ";
    for (const auto& entry : testDistribution) {
        cout << entry.first << ":" << entry.second << " ";
    }
    cout << endl;

    // 测试 2: 负载均衡测试
    cout << "\n测试 2: 负载均衡测试" << endl;
    ch.loadBalanceTest(1000);

    // 测试 3: 节点删除测试
    cout << "\n测试 3: 节点删除测试" << endl;
    ch.removeNode("Node-B");
    cout << ch.getStatus() << endl;

    // 重新测试数据分布
    testDistribution.clear();
```

```
    for (const auto& key : testKeys) {
        string node = ch.getNode(key);
        testDistribution[node]++;
        cout << "键 '" << key << "' 重新分配到节点: " << node << endl;
    }

    cout << "删除节点后数据分布: ";
    for (const auto& entry : testDistribution) {
        cout << entry.first << ":" << entry.second << " ";
    }
    cout << endl;

    // 测试 4: 异常处理测试
    cout << "\n 测试 4: 异常处理测试" << endl;
    try {
        ch.addNode(""); // 空节点名
        cout << "异常处理测试失败" << endl;
    } catch (const invalid_argument& e) {
        cout << "异常处理测试通过: " << e.what() << endl;
    }

    cout << "=== 单元测试完成 ===" << endl;
}

/**
 * 性能测试函数
 */
void performanceTest() {
    cout << "=== 一致性哈希算法性能测试 ===" << endl;

    ConsistentHashing ch(5);

    // 添加多个节点
    for (int i = 1; i <= 10; i++) {
        ch.addNode("Server-" + to_string(i));
    }

    // 测试查找性能
    int testCount = 100000;

    random_device rd;
    mt19937 gen(rd());
    uniform_int_distribution<> dis(0, 1000000);
```

```
    auto start = chrono::high_resolution_clock::now();

    for (int i = 0; i < testCount; i++) {
        string key = "key" + to_string(dis(gen));
        ch.getNode(key);
    }

    auto end = chrono::high_resolution_clock::now();
    auto duration = chrono::duration_cast<chrono::milliseconds>(end - start);

    cout << "性能测试结果:" << endl;
    cout << "节点数量: " << ch.getPhysicalNodeCount() << endl;
    cout << "虚拟节点数量: " << ch.getVirtualNodeCount() << endl;
    cout << "查找次数: " << testCount << endl;
    cout << "总查找时间: " << duration.count() << " 毫秒" << endl;
    cout << "平均查找时间: " << (double)duration.count() / testCount << " 毫秒/次" << endl;
    cout << "查找吞吐量: " << (double)testCount / (duration.count() / 1000.0) << " 次/秒" <<
endl;

    cout << "=== 性能测试完成 ===" << endl;
}

int main() {
    // 运行单元测试
    testConsistentHashing();

    // 运行性能测试
    performanceTest();

    // 演示示例
    cout << "=== 一致性哈希算法演示 ===" << endl;

    ConsistentHashing demo(5);

    // 添加节点
    demo.addNode("Server-1");
    demo.addNode("Server-2");
    demo.addNode("Server-3");

    cout << demo.getStatus() << endl;

    // 演示数据分布
```

```
    vector<string> demoKeys = {"user:1001", "order:2001", "product:3001", "cache:4001"};
    for (const auto& key : demoKeys) {
        cout << "数据 '" << key << "' 分配到: " << demo.getNode(key) << endl;
    }

    // 负载均衡测试
    demo.loadBalanceTest(10000);

    return 0;
}
```

==========

文件: Code14_ConsistentHashing.java

==========

```
package class111;

// 一致性哈希算法实现 (Java 版本)
// 题目来源: 分布式系统设计面试题
// 应用场景: 负载均衡、分布式缓存、分布式存储系统
// 题目描述: 实现一致性哈希算法，支持节点的动态增删和虚拟节点技术
//
// 解题思路:
// 1. 使用哈希环存储节点和虚拟节点
// 2. 使用虚拟节点技术解决数据分布不均问题
// 3. 支持节点的动态添加和删除
// 4. 实现高效的数据查找和节点定位
//
// 时间复杂度分析:
// - 添加节点: O(k)，其中 k 是虚拟节点数量
// - 删除节点: O(k)
// - 查找节点: O(log n)，其中 n 是节点总数
//
// 空间复杂度: O(n * k)，其中 n 是物理节点数，k 是每个节点的虚拟节点数
//
// 工程化考量:
// 1. 异常处理: 验证节点和数据的有效性
// 2. 性能优化: 使用 TreeMap 实现高效的区间查找
// 3. 负载均衡: 虚拟节点技术确保数据均匀分布
// 4. 容错性: 支持节点的动态增删，最小化数据迁移

import java.util.*;
```

```
public class Code14_ConsistentHashing {

    // 哈希环，存储虚拟节点到物理节点的映射
    private TreeMap<Integer, String> hashRing;

    // 物理节点列表
    private Set<String> physicalNodes;

    // 每个物理节点的虚拟节点数量
    private int virtualNodeCount;

    /**
     * 构造函数
     * @param virtualNodeCount 每个物理节点的虚拟节点数量
     */
    public Code14_ConsistentHashing(int virtualNodeCount) {
        this.hashRing = new TreeMap<>();
        this.physicalNodes = new HashSet<>();
        this.virtualNodeCount = virtualNodeCount;
    }

    /**
     * 添加物理节点
     * @param node 物理节点名称
     * @throws IllegalArgumentException 如果节点名为空或已存在
     */
    public void addNode(String node) {
        if (node == null || node.trim().isEmpty()) {
            throw new IllegalArgumentException("节点名不能为空");
        }
        if (physicalNodes.contains(node)) {
            throw new IllegalArgumentException("节点 " + node + " 已存在");
        }

        physicalNodes.add(node);

        // 为物理节点创建虚拟节点
        for (int i = 0; i < virtualNodeCount; i++) {
            String virtualNode = node + "#" + i;
            int hash = hash(virtualNode);
            hashRing.put(hash, node);
        }
```

```
        System.out.println("添加节点: " + node + ", 虚拟节点数: " + virtualNodeCount);
    }

    /**
     * 删除物理节点
     * @param node 物理节点名称
     * @throws IllegalArgumentException 如果节点不存在
     */
    public void removeNode(String node) {
        if (!physicalNodes.contains(node)) {
            throw new IllegalArgumentException("节点 " + node + " 不存在");
        }

        physicalNodes.remove(node);

        // 删除该物理节点的所有虚拟节点
        Iterator<Map.Entry<Integer, String>> iterator = hashRing.entrySet().iterator();
        while (iterator.hasNext()) {
            Map.Entry<Integer, String> entry = iterator.next();
            if (entry.getValue().equals(node)) {
                iterator.remove();
            }
        }

        System.out.println("删除节点: " + node);
    }

    /**
     * 根据键查找对应的物理节点
     * @param key 数据键
     * @return 负责该键的物理节点
     * @throws IllegalArgumentException 如果键为空或哈希环为空
     */
    public String getNode(String key) {
        if (key == null || key.trim().isEmpty()) {
            throw new IllegalArgumentException("键不能为空");
        }
        if (hashRing.isEmpty()) {
            throw new IllegalArgumentException("哈希环为空，请先添加节点");
        }

        int hash = hash(key);
```

```
        // 在哈希环上顺时针查找第一个大于等于该哈希值的节点
        Map.Entry<Integer, String> entry = hashRing.ceilingEntry(hash);

        // 如果没找到，则返回环上的第一个节点（环状结构）
        if (entry == null) {
            entry = hashRing.firstEntry();
        }

        return entry.getValue();
    }

    /**
     * 哈希函数 - 使用 FNV-1a 算法
     * @param str 输入字符串
     * @return 哈希值
     */
    private int hash(String str) {
        final int FNV_OFFSET_BASIS = 0x811C9DC5;
        final int FNV_PRIME = 0x01000193;

        int hash = FNV_OFFSET_BASIS;
        for (byte b : str.getBytes()) {
            hash ^= (b & 0xff);
            hash *= FNV_PRIME;
        }

        // 确保哈希值为正数
        return hash & 0x7fffffff;
    }

    /**
     * 获取哈希环的状态信息
     * @return 哈希环状态字符串
     */
    public String getStatus() {
        StringBuilder sb = new StringBuilder();
        sb.append("一致性哈希环状态:\n");
        sb.append("物理节点数: ").append(physicalNodes.size()).append("\n");
        sb.append("虚拟节点数: ").append(hashRing.size()).append("\n");
        sb.append("物理节点列表: ").append(physicalNodes).append("\n");

        // 统计每个物理节点的虚拟节点分布
        Map<String, Integer> nodeDistribution = new HashMap<>();
```

```
        for (String node : hashRing.values()) {
            nodeDistribution.put(node, nodeDistribution.getOrDefault(node, 0) + 1);
        }

        sb.append("虚拟节点分布: ").append(nodeDistribution).append("\n");

        return sb.toString();
    }

    /**
     * 负载均衡测试
     * 模拟大量数据分布，检查负载均衡性
     * @param dataCount 测试数据数量
     */
    public void loadBalanceTest(int dataCount) {
        if (physicalNodes.isEmpty()) {
            System.out.println("请先添加节点再进行负载均衡测试");
            return;
        }

        Map<String, Integer> distribution = new HashMap<>();

        // 初始化分布统计
        for (String node : physicalNodes) {
            distribution.put(node, 0);
        }

        // 模拟数据分布
        Random random = new Random();
        for (int i = 0; i < dataCount; i++) {
            String key = "key" + random.nextInt(1000000);
            String node = ch.getNode(key);
            distribution.put(node, distribution.get(node) + 1);
        }

        // 计算负载均衡指标
        int total = dataCount;
        double average = (double) total / physicalNodes.size();
        double variance = 0.0;

        System.out.println("负载均衡测试结果 (数据量: " + dataCount + "):");
        for (Map.Entry<String, Integer> entry : distribution.entrySet()) {
            double deviation = Math.abs(entry.getValue() - average);
```

```
            variance += deviation * deviation;
            System.out.printf("节点 %s: %d 数据 (%.2f%%)\n",
                entry.getKey(), entry.getValue(),
                (double) entry.getValue() / total * 100);
        }

        double stdDev = Math.sqrt(variance / physicalNodes.size());
        System.out.printf("标准差: %.2f, 相对标准差: %.2f%%\n",
            stdDev, stdDev / average * 100);
    }

    /**
     * 单元测试方法
     */
    public static void test() {
        System.out.println("=== 一致性哈希算法单元测试 ===");

        // 创建一致性哈希实例，每个物理节点有 3 个虚拟节点
        Code14_ConsistentHashing ch = new Code14_ConsistentHashing(3);

        // 测试 1: 添加节点
        ch.addNode("Node-A");
        ch.addNode("Node-B");
        ch.addNode("Node-C");

        System.out.println(ch.getStatus());

        // 测试 2: 数据分布测试
        Map<String, Integer> testDistribution = new HashMap<>();
        String[] testKeys = {"user1", "user2", "user3", "data1", "data2", "file1"};

        for (String key : testKeys) {
            String node = ch.getNode(key);
            testDistribution.put(node, testDistribution.getOrDefault(node, 0) + 1);
            System.out.println("键 '" + key + "' 分配到节点: " + node);
        }

        System.out.println("测试数据分布: " + testDistribution);

        // 测试 3: 负载均衡测试
        ch.loadBalanceTest(1000);

        // 测试 4: 节点删除测试
```

```
        System.out.println("\n=== 节点删除测试 ===");
        ch.removeNode("Node-B");
        System.out.println(ch.getStatus());

        // 重新测试数据分布
        testDistribution.clear();
        for (String key : testKeys) {
            String node = ch.getNode(key);
            testDistribution.put(node, testDistribution.getOrDefault(node, 0) + 1);
            System.out.println("键 '" + key + "' 重新分配到节点: " + node);
        }

        System.out.println("删除节点后数据分布: " + testDistribution);

        System.out.println("=== 单元测试完成 ===");
    }

    public static void main(String[] args) {
        if (args.length > 0 && "test".equals(args[0])) {
            test();
            return;
        }

        // 演示示例
        Code14_ConsistentHashing ch = new Code14_ConsistentHashing(5);

        // 添加节点
        ch.addNode("Server-1");
        ch.addNode("Server-2");
        ch.addNode("Server-3");

        System.out.println(ch.getStatus());

        // 演示数据分布
        String[] demoKeys = {"user:1001", "order:2001", "product:3001", "cache:4001"};
        for (String key : demoKeys) {
            System.out.println("数据 '" + key + "' 分配到: " + ch.getNode(key));
        }

        // 负载均衡测试
        ch.loadBalanceTest(10000);
    }
}
```

==========

文件: Code14_ConsistentHashing.py
==========
```
# 一致性哈希算法实现 (Python 版本)
# 题目来源: 分布式系统设计面试题
# 应用场景: 负载均衡、分布式缓存、分布式存储系统
# 题目描述: 实现一致性哈希算法，支持节点的动态增删和虚拟节点技术
#
# 解题思路:
# 1. 使用哈希环存储节点和虚拟节点
# 2. 使用虚拟节点技术解决数据分布不均问题
# 3. 支持节点的动态添加和删除
# 4. 实现高效的数据查找和节点定位
#
# 时间复杂度分析:
# - 添加节点: O(k)，其中 k 是虚拟节点数量
# - 删除节点: O(k)
# - 查找节点: O(log n)，其中 n 是节点总数
#
# 空间复杂度: O(n * k)，其中 n 是物理节点数，k 是每个节点的虚拟节点数
#
# 工程化考量:
# 1. 异常处理: 验证节点和数据的有效性
# 2. 性能优化: 使用 TreeMap 实现高效的区间查找
# 3. 负载均衡: 虚拟节点技术确保数据均匀分布
# 4. 容错性: 支持节点的动态增删，最小化数据迁移

import hashlib
import bisect
import random
from typing import List, Dict, Set

class ConsistentHashing:
    def __init__(self, virtual_node_count: int = 3):
        """
        构造函数
        :param virtual_node_count: 每个物理节点的虚拟节点数量
        :raises ValueError: 如果虚拟节点数量小于等于 0
        """
        if virtual_node_count <= 0:
            raise ValueError("虚拟节点数量必须大于 0")
```

```
        self.virtual_node_count = virtual_node_count

        # 哈希环，存储虚拟节点到物理节点的映射
        self.hash_ring: Dict[int, str] = {}

        # 物理节点集合
        self.physical_nodes: Set[str] = set()

        # 排序的哈希值列表，用于快速查找
        self.sorted_hashes: List[int] = []

    def _hash(self, key: str) -> int:
        """
        哈希函数 - 使用 MD5 算法
        :param key: 输入字符串
        :return: 哈希值
        """
        # 使用 MD5 哈希，然后取前 4 个字节作为整数
        hash_obj = hashlib.md5(key.encode('utf-8'))
        hash_bytes = hash_obj.digest()

        # 将前 4 个字节转换为整数
        hash_int = int.from_bytes(hash_bytes[:4], byteorder='big')

        # 确保哈希值为正数
        return hash_int & 0x7fffffff

    def add_node(self, node: str):
        """
        添加物理节点
        :param node: 物理节点名称
        :raises ValueError: 如果节点名为空或已存在
        """
        if not node:
            raise ValueError("节点名不能为空")
        if node in self.physical_nodes:
            raise ValueError(f"节点 {node} 已存在")

        self.physical_nodes.add(node)

        # 为物理节点创建虚拟节点
        for i in range(self.virtual_node_count):
```

```
            virtual_node = f"{node}#{i}"
            node_hash = self._hash(virtual_node)

            # 添加到哈希环
            self.hash_ring[node_hash] = node

            # 维护排序的哈希值列表
            bisect.insort(self.sorted_hashes, node_hash)

        print(f"添加节点: {node}, 虚拟节点数: {self.virtual_node_count}")

    def remove_node(self, node: str):
        """
        删除物理节点
        :param node: 物理节点名称
        :raises ValueError: 如果节点不存在
        """
        if node not in self.physical_nodes:
            raise ValueError(f"节点 {node} 不存在")

        self.physical_nodes.remove(node)

        # 删除该物理节点的所有虚拟节点
        hashes_to_remove = []
        for node_hash, physical_node in self.hash_ring.items():
            if physical_node == node:
                hashes_to_remove.append(node_hash)

        for node_hash in hashes_to_remove:
            del self.hash_ring[node_hash]
            # 从排序列表中删除
            index = bisect.bisect_left(self.sorted_hashes, node_hash)
            if index < len(self.sorted_hashes) and self.sorted_hashes[index] == node_hash:
                del self.sorted_hashes[index]

        print(f"删除节点: {node}")

    def get_node(self, key: str) -> str:
        """
        根据键查找对应的物理节点
        :param key: 数据键
        :return: 负责该键的物理节点
        :raises ValueError: 如果键为空或哈希环为空
```

```
        """
        if not key:
            raise ValueError("键不能为空")
        if not self.hash_ring:
            raise ValueError("哈希环为空，请先添加节点")

        key_hash = self._hash(key)

        # 在哈希环上顺时针查找第一个大于等于该哈希值的节点
        index = bisect.bisect_left(self.sorted_hashes, key_hash)

        # 如果没找到，则返回环上的第一个节点（环状结构）
        if index == len(self.sorted_hashes):
            index = 0

        node_hash = self.sorted_hashes[index]
        return self.hash_ring[node_hash]

    def get_status(self) -> str:
        """
        获取哈希环的状态信息
        :return: 哈希环状态字符串
        """
        result = []
        result.append("一致性哈希环状态:")
        result.append(f"物理节点数: {len(self.physical_nodes)}")
        result.append(f"虚拟节点数: {len(self.hash_ring)}")

        # 物理节点列表
        result.append(f"物理节点列表: {', '.join(sorted(self.physical_nodes))}")

        # 统计每个物理节点的虚拟节点分布
        node_distribution: Dict[str, int] = {}
        for physical_node in self.hash_ring.values():
            node_distribution[physical_node] = node_distribution.get(physical_node, 0) + 1

        distribution_str = ", ".join([f"{node}:{count}" for node, count in
node_distribution.items()])
        result.append(f"虚拟节点分布: {distribution_str}")

        return "\n".join(result)

    def load_balance_test(self, data_count: int):
```

```
    """
    负载均衡测试
    模拟大量数据分布，检查负载均衡性
    :param data_count: 测试数据数量
    """
    if not self.physical_nodes:
        print("请先添加节点再进行负载均衡测试")
        return

    distribution: Dict[str, int] = {}

    # 初始化分布统计
    for node in self.physical_nodes:
        distribution[node] = 0

    # 模拟数据分布
    for i in range(data_count):
        key = f"key{random.randint(0, 1000000)}"
        node = self.get_node(key)
        distribution[node] += 1

    # 计算负载均衡指标
    total = data_count
    average = total / len(self.physical_nodes)
    variance = 0.0

    print(f"负载均衡测试结果 (数据量: {data_count}):")
    for node, count in distribution.items():
        deviation = abs(count - average)
        variance += deviation * deviation
        percentage = (count / total) * 100
        print(f"节点 {node}: {count} 数据 ({percentage:.2f}%)")

    std_dev = (variance / len(self.physical_nodes)) ** 0.5
    relative_std_dev = (std_dev / average) * 100
    print(f"标准差: {std_dev:.2f}, 相对标准差: {relative_std_dev:.2f}%")

def get_physical_node_count(self) -> int:
    """
    获取物理节点数量
    :return: 物理节点数量
    """
    return len(self.physical_nodes)
```

```
    def get_virtual_node_count(self) -> int:
        """
        获取虚拟节点数量
        :return: 虚拟节点数量
        """
        return len(self.hash_ring)

def test_consistent_hashing():
    """
    单元测试函数
    """
    print("=== 一致性哈希算法单元测试 ===")

    # 测试 1: 基本功能测试
    print("测试 1: 基本功能测试")
    ch = ConsistentHashing(3)

    # 添加节点
    ch.add_node("Node-A")
    ch.add_node("Node-B")
    ch.add_node("Node-C")

    print(ch.get_status())

    # 测试数据分布
    test_distribution: Dict[str, int] = {}
    test_keys = ["user1", "user2", "user3", "data1", "data2", "file1"]

    for key in test_keys:
        node = ch.get_node(key)
        test_distribution[node] = test_distribution.get(node, 0) + 1
        print(f"键 '{key}' 分配到节点: {node}")

    print(f"测试数据分布: {test_distribution}")

    # 测试 2: 负载均衡测试
    print("\n测试 2: 负载均衡测试")
    ch.load_balance_test(1000)

    # 测试 3: 节点删除测试
    print("\n测试 3: 节点删除测试")
    ch.remove_node("Node-B")
```

```
    print(ch.get_status())

    # 重新测试数据分布
    test_distribution.clear()
    for key in test_keys:
        node = ch.get_node(key)
        test_distribution[node] = test_distribution.get(node, 0) + 1
        print(f"键 '{key}' 重新分配到节点: {node}")

    print(f"删除节点后数据分布: {test_distribution}")

    # 测试 4: 异常处理测试
    print("\n 测试 4: 异常处理测试")
    try:
        ch.add_node("")  # 空节点名
        print("异常处理测试失败")
    except ValueError as e:
        print(f"异常处理测试通过: {e}")

    print("=== 单元测试完成 ===")

def performance_test():
    """
    性能测试函数
    """
    import time

    print("=== 一致性哈希算法性能测试 ===")

    ch = ConsistentHashing(5)

    # 添加多个节点
    for i in range(1, 11):
        ch.add_node(f"Server-{i}")

    # 测试查找性能
    test_count = 100000

    start_time = time.time()

    for i in range(test_count):
        key = f"key{random.randint(0, 1000000)}"
        ch.get_node(key)
```

```
    end_time = time.time()
    duration = (end_time - start_time) * 1000  # 转换为毫秒

    print("性能测试结果:")
    print(f"节点数量: {ch.get_physical_node_count()}")
    print(f"虚拟节点数量: {ch.get_virtual_node_count()}")
    print(f"查找次数: {test_count}")
    print(f"总查找时间: {duration:.2f} 毫秒")
    print(f"平均查找时间: {duration / test_count:.4f} 毫秒/次")
    print(f"查找吞吐量: {test_count / (duration / 1000):.2f} 次/秒")

    print("=== 性能测试完成 ===")

if __name__ == "__main__":
    # 运行单元测试
    test_consistent_hashing()

    # 运行性能测试
    performance_test()

    # 演示示例
    print("=== 一致性哈希算法演示 ===")

    demo = ConsistentHashing(5)

    # 添加节点
    demo.add_node("Server-1")
    demo.add_node("Server-2")
    demo.add_node("Server-3")

    print(demo.get_status())

    # 演示数据分布
    demo_keys = ["user:1001", "order:2001", "product:3001", "cache:4001"]
    for key in demo_keys:
        print(f"数据 '{key}' 分配到: {demo.get_node(key)}")

    # 负载均衡测试
    demo.load_balance_test(10000)

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

文件: Code15_BloomFilter.cpp

```
==========================================
// 布隆过滤器实现 (C++版本)
// 题目来源: 大数据处理、缓存系统、网络爬虫去重
// 应用场景: 网页去重、垃圾邮件过滤、缓存穿透防护
// 题目描述: 实现布隆过滤器，支持元素添加和存在性检查
//
// 解题思路:
// 1. 使用多个哈希函数将元素映射到位数组的不同位置
// 2. 添加元素时，将所有哈希位置设为1
// 3. 检查元素时，如果所有哈希位置都为1，则元素可能存在
// 4. 使用误判率公式计算最优的哈希函数数量和位数组大小
//
// 时间复杂度分析:
// - 添加元素: O(k)，其中k是哈希函数数量
// - 检查元素: O(k)
//
// 空间复杂度: O(m)，其中m是位数组大小
//
// 工程化考量:
// 1. 误判率控制: 根据预期元素数量和可接受的误判率计算最优参数
// 2. 哈希函数选择: 使用不同的哈希函数减少冲突
// 3. 内存优化: 使用位数组节省空间
// 4. 线程安全: 在多线程环境下安全使用

#include <iostream>
#include <vector>
#include <bitset>
#include <functional>
#include <random>
#include <cmath>
#include <stdexcept>
#include <string>
#include <chrono>

using namespace std;

class BloomFilter {
private:
    // 位数组，用于存储元素的存在性信息
    vector<bool> bitSet;

    // 位数组大小
```

```
    size_t bitSetSize;

    // 哈希函数数量
    size_t hashFunctionCount;

    // 预期元素数量
    size_t expectedElementCount;

    // 实际添加的元素数量
    size_t actualElementCount;

    // 哈希函数种子
    vector<size_t> seeds;

    /**
     * 哈希函数 - 使用简单的字符串哈希算法
     * @param str 输入字符串
     * @param seed 哈希种子
     * @return 哈希值
     */
    size_t hash(const string& str, size_t seed) const {
        size_t hash = seed;
        for (char c : str) {
            hash = hash * 31 + c;
        }
        return hash;
    }

public:
    /**
     * 构造函数 - 根据预期元素数量和误判率自动计算参数
     * @param expectedElementCount 预期元素数量
     * @param falsePositiveRate 可接受的误判率 (0 < falsePositiveRate < 1)
     * @throws invalid_argument 如果参数无效
     */
    BloomFilter(size_t expectedElementCount, double falsePositiveRate) {
        if (expectedElementCount <= 0) {
            throw invalid_argument("预期元素数量必须大于0");
        }
        if (falsePositiveRate <= 0 || falsePositiveRate >= 1) {
            throw invalid_argument("误判率必须在0和1之间");
        }
```

```
    this->expectedElementCount = expectedElementCount;
    this->actualElementCount = 0;

    // 根据误判率公式计算最优参数
    // m = - (n * ln(p)) / (ln(2))^2
    // k = (m / n) * ln(2)
    this->bitSetSize = static_cast<size_t>(
        ceil(-expectedElementCount * log(falsePositiveRate) / (log(2) * log(2)))
    );
    this->hashFunctionCount = static_cast<size_t>(
        ceil((bitSetSize / static_cast<double>(expectedElementCount)) * log(2))
    );

    // 确保哈希函数数量至少为1
    this->hashFunctionCount = max(static_cast<size_t>(1), this->hashFunctionCount);

    // 初始化位数组
    this->bitSet.resize(bitSetSize, false);

    // 初始化哈希种子
    this->seeds.resize(hashFunctionCount);
    mt19937 gen(42); // 固定种子保证可重复性
    uniform_int_distribution<size_t> dis(1, 1000000);

    for (size_t i = 0; i < hashFunctionCount; i++) {
        seeds[i] = dis(gen);
    }

    cout << "布隆过滤器初始化: 预期元素数=" << expectedElementCount
         << ", 误判率=" << falsePositiveRate
         << ", 位数组大小=" << bitSetSize
         << ", 哈希函数数=" << hashFunctionCount << endl;
}

/**
 * 构造函数 - 手动指定参数
 * @param bitSetSize 位数组大小
 * @param hashFunctionCount 哈希函数数量
 */
BloomFilter(size_t bitSetSize, size_t hashFunctionCount) {
    if (bitSetSize <= 0) {
        throw invalid_argument("位数组大小必须大于0");
    }
```

```
    if (hashFunctionCount <= 0) {
        throw invalid_argument("哈希函数数量必须大于0");
    }

    this->bitSetSize = bitSetSize;
    this->hashFunctionCount = hashFunctionCount;
    this->expectedElementCount = 0; // 未知预期数量
    this->actualElementCount = 0;
    this->bitSet.resize(bitSetSize, false);

    // 初始化哈希种子
    this->seeds.resize(hashFunctionCount);
    mt19937 gen(42);
    uniform_int_distribution<size_t> dis(1, 1000000);

    for (size_t i = 0; i < hashFunctionCount; i++) {
        seeds[i] = dis(gen);
    }
}

/**
 * 添加元素到布隆过滤器
 * @param element 要添加的元素
 * @throws invalid_argument 如果元素为空
 */
void add(const string& element) {
    if (element.empty()) {
        throw invalid_argument("元素不能为空");
    }

    // 计算所有哈希位置并设置为1
    for (size_t i = 0; i < hashFunctionCount; i++) {
        size_t hashValue = hash(element, seeds[i]);
        size_t position = hashValue % bitSetSize;
        bitSet[position] = true;
    }

    actualElementCount++;
}

/**
 * 检查元素是否可能在布隆过滤器中
 * @param element 要检查的元素
```

```
     * @return true 如果元素可能存在，false 如果元素一定不存在
     * @throws invalid_argument 如果元素为空
     */
    bool mightContain(const string& element) const {
        if (element.empty()) {
            throw invalid_argument("元素不能为空");
        }

        // 检查所有哈希位置是否都为 1
        for (size_t i = 0; i < hashFunctionCount; i++) {
            size_t hashValue = hash(element, seeds[i]);
            size_t position = hashValue % bitSetSize;
            if (!bitSet[position]) {
                return false; // 如果有一个位置为 0，元素一定不存在
            }
        }

        return true; // 所有位置都为 1，元素可能存在
    }

    /**
     * 获取布隆过滤器的状态信息
     * @return 状态信息字符串
     */
    string getStatus() const {
        size_t setBits = 0;
        for (bool bit : bitSet) {
            if (bit) setBits++;
        }

        double fillRatio = static_cast<double>(setBits) / bitSetSize;

        // 计算当前误判率
        double currentFalsePositiveRate = pow(fillRatio, hashFunctionCount);

        string result;
        result += "布隆过滤器状态:\n";
        result += "位数组大小: " + to_string(bitSetSize) + "\n";
        result += "哈希函数数量: " + to_string(hashFunctionCount) + "\n";
        result += "预期元素数量: " + to_string(expectedElementCount) + "\n";
        result += "实际元素数量: " + to_string(actualElementCount) + "\n";
        result += "已设置位数: " + to_string(setBits) + "\n";
        result += "填充比例: " + to_string(fillRatio) + "\n";
```

```
    result += "当前误判率: " + to_string(currentFalsePositiveRate) + "\n";

    return result;
}

/**
 * 性能测试 - 测试布隆过滤器的误判率
 * @param testElementCount 测试元素数量
 */
void performanceTest(size_t testElementCount) {
    if (testElementCount <= 0) {
        cout << "测试元素数量必须大于0" << endl;
        return;
    }

    cout << "=== 布隆过滤器性能测试 ===" << endl;
    cout << "测试元素数量: " << testElementCount << endl;

    // 添加测试元素
    size_t falsePositives = 0;
    size_t trueNegatives = 0;

    // 添加真实元素
    for (size_t i = 0; i < testElementCount; i++) {
        add("real" + to_string(i));
    }

    // 测试不存在元素
    for (size_t i = 0; i < testElementCount; i++) {
        string fakeElement = "fake" + to_string(i);
        if (mightContain(fakeElement)) {
            falsePositives++; // 误判
        } else {
            trueNegatives++; // 正确判断
        }
    }

    double actualFalsePositiveRate = static_cast<double>(falsePositives) / testElementCount;

    cout << "测试结果:" << endl;
    cout << "误判数量: " << falsePositives << endl;
    cout << "正确判断数量: " << trueNegatives << endl;
    cout << "实际误判率: " << actualFalsePositiveRate << endl;
```

```
        size_t setBits = 0;
        for (bool bit : bitSet) {
            if (bit) setBits++;
        }
        double theoreticalFalsePositiveRate = pow(static_cast<double>(setBits) / bitSetSize,
hashFunctionCount);
        cout << "理论误判率: " << theoreticalFalsePositiveRate << endl;

        cout << getStatus() << endl;
    }

    /**
     * 获取位数组大小
     * @return 位数组大小
     */
    size_t getBitSetSize() const {
        return bitSetSize;
    }

    /**
     * 获取哈希函数数量
     * @return 哈希函数数量
     */
    size_t getHashFunctionCount() const {
        return hashFunctionCount;
    }

    /**
     * 获取实际元素数量
     * @return 实际元素数量
     */
    size_t getActualElementCount() const {
        return actualElementCount;
    }
};

/**
 * 单元测试函数
 */
void testBloomFilter() {
    cout << "=== 布隆过滤器单元测试 ===" << endl;
```

```
// 测试 1: 基本功能测试
cout << "测试 1: 基本功能测试" << endl;
BloomFilter bf(1000, 0.01);

// 添加元素
bf.add("apple");
bf.add("banana");
bf.add("cherry");

// 检查存在的元素
if (bf.mightContain("apple") && bf.mightContain("banana") && bf.mightContain("cherry")) {
    cout << "存在性检查测试通过" << endl;
} else {
    cout << "存在性检查测试失败" << endl;
}

// 检查不存在的元素
if (!bf.mightContain("orange") && !bf.mightContain("grape")) {
    cout << "不存在性检查测试通过" << endl;
} else {
    cout << "不存在性检查测试失败" << endl;
}

// 测试 2: 性能测试
cout << "\n 测试 2: 性能测试" << endl;
bf.performanceTest(1000);

// 测试 3: 不同参数对比
cout << "\n 测试 3: 不同参数对比" << endl;

BloomFilter bf1(1000, 0.1);  // 高误判率
BloomFilter bf2(1000, 0.01); // 低误判率
BloomFilter bf3(1000, 0.001); // 极低误判率

// 添加相同元素
for (int i = 0; i < 500; i++) {
    bf1.add("test" + to_string(i));
    bf2.add("test" + to_string(i));
    bf3.add("test" + to_string(i));
}

// 测试误判率
size_t fp1 = 0, fp2 = 0, fp3 = 0;
```

```
    for (int i = 500; i < 1000; i++) {
        if (bf1.mightContain("test" + to_string(i))) fp1++;
        if (bf2.mightContain("test" + to_string(i))) fp2++;
        if (bf3.mightContain("test" + to_string(i))) fp3++;
    }

    cout << "不同误判率配置的测试结果:" << endl;
    cout << "误判率 0.1: 实际误判率=" << static_cast<double>(fp1)/500 << endl;
    cout << "误判率 0.01: 实际误判率=" << static_cast<double>(fp2)/500 << endl;
    cout << "误判率 0.001: 实际误判率=" << static_cast<double>(fp3)/500 << endl;

    // 测试 4: 异常处理测试
    cout << "\n 测试 4: 异常处理测试" << endl;
    try {
        bf.add(""); // 空元素
        cout << "异常处理测试失败" << endl;
    } catch (const invalid_argument& e) {
        cout << "异常处理测试通过: " << e.what() << endl;
    }

    cout << "=== 单元测试完成 ===" << endl;
}

/**
 * 性能测试函数
 */
void performanceTest() {
    cout << "=== 布隆过滤器性能测试 ===" << endl;

    // 测试不同规模的布隆过滤器
    vector<pair<size_t, double>> testCases = {
        {1000, 0.01},
        {10000, 0.01},
        {100000, 0.01}
    };

    for (const auto& testCase : testCases) {
        size_t expectedCount = testCase.first;
        double falsePositiveRate = testCase.second;

        cout << "\n 测试规模: 预期元素数=" << expectedCount
             << ", 误判率=" << falsePositiveRate << endl;
```

```
        BloomFilter bf(expectedCount, falsePositiveRate);

        // 测试添加性能
        auto start = chrono::high_resolution_clock::now();

        for (size_t i = 0; i < expectedCount; i++) {
            bf.add("element" + to_string(i));
        }

        auto end = chrono::high_resolution_clock::now();
        auto addDuration = chrono::duration_cast<chrono::milliseconds>(end - start);

        // 测试查询性能
        start = chrono::high_resolution_clock::now();

        size_t found = 0;
        for (size_t i = 0; i < expectedCount; i++) {
            if (bf.mightContain("element" + to_string(i))) {
                found++;
            }
        }

        end = chrono::high_resolution_clock::now();
        auto queryDuration = chrono::duration_cast<chrono::milliseconds>(end - start);

        cout << "性能测试结果:" << endl;
        cout << "添加时间: " << addDuration.count() << " 毫秒" << endl;
        cout << "平均添加时间: " << (double)addDuration.count() / expectedCount << " 毫秒/元素"
<< endl;
        cout << "查询时间: " << queryDuration.count() << " 毫秒" << endl;
        cout << "平均查询时间: " << (double)queryDuration.count() / expectedCount << " 毫秒/元素"
<< endl;
        cout << "正确查询数量: " << found << "/" << expectedCount << endl;
    }

    cout << "=== 性能测试完成 ===" << endl;
}

int main() {
    // 运行单元测试
    testBloomFilter();

    // 运行性能测试
```

```
performanceTest();

// 演示示例
cout << "=== 布隆过滤器演示 ===" << endl;

// 创建一个预期处理 10000 个元素，误判率为 1%的布隆过滤器
BloomFilter bloomFilter(10000, 0.01);

// 添加一些 URL 到布隆过滤器（模拟网页去重）
vector<string> urls = {
    "https://example.com/page1",
    "https://example.com/page2",
    "https://example.com/page3",
    "https://example.com/page4",
    "https://example.com/page5"
};

for (const auto& url : urls) {
    bloomFilter.add(url);
    cout << "添加 URL: " << url << endl;
}

// 检查 URL 是否已存在
vector<string> testUrls = {
    "https://example.com/page1",     // 已存在
    "https://example.com/page6",     // 不存在
    "https://example.com/page3",     // 已存在
    "https://example.com/page7"      // 不存在
};

cout << "\nURL 存在性检查:" << endl;
for (const auto& url : testUrls) {
    bool exists = bloomFilter.mightContain(url);
    cout << "URL " << url << ": " << (exists ? "可能已存在" : "一定不存在") << endl;
}

// 显示布隆过滤器状态
cout << "\n" << bloomFilter.getStatus() << endl;

// 性能测试
bloomFilter.performanceTest(1000);

return 0;
```

```
}

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

文件: Code15_BloomFilter.java
==========================================
package class111;

// 布隆过滤器实现 (Java 版本)
// 题目来源: 大数据处理、缓存系统、网络爬虫去重
// 应用场景: 网页去重、垃圾邮件过滤、缓存穿透防护
// 题目描述: 实现布隆过滤器，支持元素添加和存在性检查
//
// 解题思路:
// 1. 使用多个哈希函数将元素映射到位数组的不同位置
// 2. 添加元素时，将所有哈希位置设为 1
// 3. 检查元素时，如果所有哈希位置都为 1，则元素可能存在
// 4. 使用误判率公式计算最优的哈希函数数量和位数组大小
//
// 时间复杂度分析:
// - 添加元素: O(k)，其中 k 是哈希函数数量
// - 检查元素: O(k)
//
// 空间复杂度: O(m)，其中 m 是位数组大小
//
// 工程化考量:
// 1. 误判率控制: 根据预期元素数量和可接受的误判率计算最优参数
// 2. 哈希函数选择: 使用不同的哈希函数减少冲突
// 3. 内存优化: 使用位数组节省空间
// 4. 线程安全: 在多线程环境下安全使用

import java.util.BitSet;
import java.util.Random;

public class Code15_BloomFilter {

    // 位数组，用于存储元素的存在性信息
    private BitSet bitSet;

    // 位数组大小
    private int bitSetSize;

    // 哈希函数数量
```

```
    private int hashFunctionCount;

    // 预期元素数量
    private int expectedElementCount;

    // 实际添加的元素数量
    private int actualElementCount;

    // 随机种子，用于生成不同的哈希函数
    private int[] seeds;

    /**
     * 构造函数 - 根据预期元素数量和误判率自动计算参数
     * @param expectedElementCount 预期元素数量
     * @param falsePositiveRate 可接受的误判率 (0 < falsePositiveRate < 1)
     * @throws IllegalArgumentException 如果参数无效
     */
    public Code15_BloomFilter(int expectedElementCount, double falsePositiveRate) {
        if (expectedElementCount <= 0) {
            throw new IllegalArgumentException("预期元素数量必须大于 0");
        }
        if (falsePositiveRate <= 0 || falsePositiveRate >= 1) {
            throw new IllegalArgumentException("误判率必须在 0 和 1 之间");
        }

        this.expectedElementCount = expectedElementCount;
        this.actualElementCount = 0;

        // 根据误判率公式计算最优参数
        // m = - (n * ln(p)) / (ln(2))^2
        // k = (m / n) * ln(2)
        this.bitSetSize = (int) Math.ceil(-expectedElementCount * Math.log(falsePositiveRate) /
(Math.log(2) * Math.log(2)));
        this.hashFunctionCount = (int) Math.ceil((bitSetSize / (double) expectedElementCount) *
Math.log(2));

        // 确保哈希函数数量至少为 1
        this.hashFunctionCount = Math.max(1, this.hashFunctionCount);

        this.bitSet = new BitSet(bitSetSize);

        // 初始化哈希种子
        this.seeds = new int[hashFunctionCount];
```

```
        Random random = new Random(42); // 固定种子保证可重复性
        for (int i = 0; i < hashFunctionCount; i++) {
            seeds[i] = random.nextInt();
        }

        System.out.printf("布隆过滤器初始化: 预期元素数=%d, 误判率=%.6f, 位数组大小=%d, 哈希函数数=%d%n",
                expectedElementCount, falsePositiveRate, bitSetSize, hashFunctionCount);
    }

    /**
     * 构造函数 - 手动指定参数
     * @param bitSetSize 位数组大小
     * @param hashFunctionCount 哈希函数数量
     */
    public Code15_BloomFilter(int bitSetSize, int hashFunctionCount) {
        if (bitSetSize <= 0) {
            throw new IllegalArgumentException("位数组大小必须大于 0");
        }
        if (hashFunctionCount <= 0) {
            throw new IllegalArgumentException("哈希函数数量必须大于 0");
        }

        this.bitSetSize = bitSetSize;
        this.hashFunctionCount = hashFunctionCount;
        this.expectedElementCount = 0; // 未知预期数量
        this.actualElementCount = 0;
        this.bitSet = new BitSet(bitSetSize);

        // 初始化哈希种子
        this.seeds = new int[hashFunctionCount];
        Random random = new Random(42);
        for (int i = 0; i < hashFunctionCount; i++) {
            seeds[i] = random.nextInt();
        }
    }

    /**
     * 添加元素到布隆过滤器
     * @param element 要添加的元素
     * @throws IllegalArgumentException 如果元素为空
     */
    public void add(String element) {
```

```
    if (element == null) {
        throw new IllegalArgumentException("元素不能为空");
    }

    // 计算所有哈希位置并设置为1
    for (int i = 0; i < hashFunctionCount; i++) {
        int hash = hash(element, seeds[i]);
        int position = Math.abs(hash % bitSetSize);
        bitSet.set(position);
    }

    actualElementCount++;
}

/**
 * 检查元素是否可能在布隆过滤器中
 * @param element 要检查的元素
 * @return true如果元素可能存在，false如果元素一定不存在
 * @throws IllegalArgumentException 如果元素为空
 */
public boolean mightContain(String element) {
    if (element == null) {
        throw new IllegalArgumentException("元素不能为空");
    }

    // 检查所有哈希位置是否都为1
    for (int i = 0; i < hashFunctionCount; i++) {
        int hash = hash(element, seeds[i]);
        int position = Math.abs(hash % bitSetSize);
        if (!bitSet.get(position)) {
            return false; // 如果有一个位置为0，元素一定不存在
        }
    }

    return true; // 所有位置都为1，元素可能存在
}

/**
 * 哈希函数 - 使用简单的字符串哈希算法
 * @param str 输入字符串
 * @param seed 哈希种子
 * @return 哈希值
 */
```

```
    private int hash(String str, int seed) {
        int hash = seed;
        for (char c : str.toCharArray()) {
            hash = hash * 31 + c;
        }
        return hash;
    }

    /**
     * 获取布隆过滤器的状态信息
     * @return 状态信息字符串
     */
    public String getStatus() {
        int setBits = bitSet.cardinality();
        double fillRatio = (double) setBits / bitSetSize;

        // 计算当前误判率
        double currentFalsePositiveRate = Math.pow(fillRatio, hashFunctionCount);

        StringBuilder sb = new StringBuilder();
        sb.append("布隆过滤器状态:\n");
        sb.append("位数组大小: ").append(bitSetSize).append("\n");
        sb.append("哈希函数数量: ").append(hashFunctionCount).append("\n");
        sb.append("预期元素数量: ").append(expectedElementCount).append("\n");
        sb.append("实际元素数量: ").append(actualElementCount).append("\n");
        sb.append("已设置位数: ").append(setBits).append("\n");
        sb.append("填充比例: ").append(String.format("%.6f", fillRatio)).append("\n");
        sb.append("当前误判率: ").append(String.format("%.6f",
currentFalsePositiveRate)).append("\n");

        return sb.toString();
    }

    /**
     * 性能测试 - 测试布隆过滤器的误判率
     * @param testElementCount 测试元素数量
     */
    public void performanceTest(int testElementCount) {
        if (testElementCount <= 0) {
            System.out.println("测试元素数量必须大于 0");
            return;
        }
```

```
        System.out.println("=== 布隆过滤器性能测试 ===");
        System.out.println("测试元素数量: " + testElementCount);

        // 添加测试元素
        int falsePositives = 0;
        int trueNegatives = 0;

        // 添加真实元素
        for (int i = 0; i < testElementCount; i++) {
            add("real" + i);
        }

        // 测试不存在元素
        for (int i = 0; i < testElementCount; i++) {
            String fakeElement = "fake" + i;
            if (mightContain(fakeElement)) {
                falsePositives++; // 误判
            } else {
                trueNegatives++; // 正确判断
            }
        }

        double actualFalsePositiveRate = (double) falsePositives / testElementCount;

        System.out.println("测试结果:");
        System.out.println("误判数量: " + falsePositives);
        System.out.println("正确判断数量: " + trueNegatives);
        System.out.println("实际误判率: " + String.format("%.6f", actualFalsePositiveRate));
        System.out.println("理论误判率: " + String.format("%.6f", Math.pow((double)
bitSet.cardinality() / bitSetSize, hashFunctionCount)));

        System.out.println(getStatus());
    }

    /**
     * 单元测试方法
     */
    public static void test() {
        System.out.println("=== 布隆过滤器单元测试 ===");

        // 测试 1: 基本功能测试
        System.out.println("测试 1: 基本功能测试");
        Code15_BloomFilter bf = new Code15_BloomFilter(1000, 0.01);
```

```
// 添加元素
bf.add("apple");
bf.add("banana");
bf.add("cherry");

// 检查存在的元素
assert bf.mightContain("apple") : "应该包含 apple";
assert bf.mightContain("banana") : "应该包含 banana";
assert bf.mightContain("cherry") : "应该包含 cherry";

// 检查不存在的元素
assert !bf.mightContain("orange") : "不应该包含 orange";
assert !bf.mightContain("grape") : "不应该包含 grape";

System.out.println("基本功能测试通过");

// 测试 2: 性能测试
System.out.println("\n 测试 2: 性能测试");
bf.performanceTest(1000);

// 测试 3: 不同参数对比
System.out.println("\n 测试 3: 不同参数对比");

Code15_BloomFilter bf1 = new Code15_BloomFilter(1000, 0.1);  // 高误判率
Code15_BloomFilter bf2 = new Code15_BloomFilter(1000, 0.01); // 低误判率
Code15_BloomFilter bf3 = new Code15_BloomFilter(1000, 0.001); // 极低误判率

// 添加相同元素
for (int i = 0; i < 500; i++) {
    bf1.add("test" + i);
    bf2.add("test" + i);
    bf3.add("test" + i);
}

// 测试误判率
int fp1 = 0, fp2 = 0, fp3 = 0;
for (int i = 500; i < 1000; i++) {
    if (bf1.mightContain("test" + i)) fp1++;
    if (bf2.mightContain("test" + i)) fp2++;
    if (bf3.mightContain("test" + i)) fp3++;
}
```

```
        System.out.printf("不同误判率配置的测试结果:%n");
        System.out.printf("误判率 0.1: 实际误判率=%.4f%n", (double)fp1/500);
        System.out.printf("误判率 0.01: 实际误判率=%.4f%n", (double)fp2/500);
        System.out.printf("误判率 0.001: 实际误判率=%.4f%n", (double)fp3/500);

        System.out.println("=== 单元测试完成 ===");
    }

    public static void main(String[] args) {
        if (args.length > 0 && "test".equals(args[0])) {
            test();
            return;
        }

        // 演示示例
        System.out.println("=== 布隆过滤器演示 ===");

        // 创建一个预期处理 10000 个元素，误判率为 1%的布隆过滤器
        Code15_BloomFilter bloomFilter = new Code15_BloomFilter(10000, 0.01);

        // 添加一些 URL 到布隆过滤器（模拟网页去重）
        String[] urls = {
            "https://example.com/page1",
            "https://example.com/page2",
            "https://example.com/page3",
            "https://example.com/page4",
            "https://example.com/page5"
        };

        for (String url : urls) {
            bloomFilter.add(url);
            System.out.println("添加 URL: " + url);
        }

        // 检查 URL 是否已存在
        String[] testUrls = {
            "https://example.com/page1",     // 已存在
            "https://example.com/page6",     // 不存在
            "https://example.com/page3",     // 已存在
            "https://example.com/page7"      // 不存在
        };

        System.out.println("\nURL 存在性检查:");
```

```
        for (String url : testUrls) {
            boolean exists = bloomFilter.mightContain(url);
            System.out.printf("URL %s: %s%n", url, exists ? "可能已存在" : "一定不存在");
        }

        // 显示布隆过滤器状态
        System.out.println("\n" + bloomFilter.getStatus());

        // 性能测试
        bloomFilter.performanceTest(1000);
    }
}
```

==========

文件: Code15_BloomFilter.py

==========

```
# 布隆过滤器实现 (Python 版本)
# 题目来源: 大数据处理、缓存系统、网络爬虫去重
# 应用场景: 网页去重、垃圾邮件过滤、缓存穿透防护
# 题目描述: 实现布隆过滤器，支持元素添加和存在性检查
#
# 解题思路:
# 1. 使用多个哈希函数将元素映射到位数组的不同位置
# 2. 添加元素时，将所有哈希位置设为 1
# 3. 检查元素时，如果所有哈希位置都为 1，则元素可能存在
# 4. 使用误判率公式计算最优的哈希函数数量和位数组大小
#
# 时间复杂度分析:
# - 添加元素: O(k)，其中 k 是哈希函数数量
# - 检查元素: O(k)
#
# 空间复杂度: O(m)，其中 m 是位数组大小
#
# 工程化考量:
# 1. 误判率控制: 根据预期元素数量和可接受的误判率计算最优参数
# 2. 哈希函数选择: 使用不同的哈希函数减少冲突
# 3. 内存优化: 使用位数组节省空间
# 4. 线程安全: 在多线程环境下安全使用

import math
import hashlib
import random
```

```
from typing import List

class BloomFilter:
    def __init__(self, expected_element_count: int, false_positive_rate: float):
        """
        构造函数 - 根据预期元素数量和误判率自动计算参数
        :param expected_element_count: 预期元素数量
        :param false_positive_rate: 可接受的误判率 (0 < false_positive_rate < 1)
        :raises ValueError: 如果参数无效
        """
        if expected_element_count <= 0:
            raise ValueError("预期元素数量必须大于 0")
        if false_positive_rate <= 0 or false_positive_rate >= 1:
            raise ValueError("误判率必须在 0 和 1 之间")

        self.expected_element_count = expected_element_count
        self.actual_element_count = 0

        # 根据误判率公式计算最优参数
        # m = - (n * ln(p)) / (ln(2))^2
        # k = (m / n) * ln(2)
        self.bit_set_size = math.ceil(
            -expected_element_count * math.log(false_positive_rate) / (math.log(2) ** 2)
        )
        self.hash_function_count = math.ceil(
            (self.bit_set_size / expected_element_count) * math.log(2)
        )

        # 确保哈希函数数量至少为 1
        self.hash_function_count = max(1, self.hash_function_count)

        # 初始化位数组
        self.bit_set = [False] * self.bit_set_size

        # 初始化哈希种子
        random.seed(42)  # 固定种子保证可重复性
        self.seeds = [random.randint(1, 1000000) for _ in range(self.hash_function_count)]

        print(f"布隆过滤器初始化: 预期元素数={expected_element_count}, "
              f"误判率={false_positive_rate}, "
              f"位数组大小={self.bit_set_size}, "
              f"哈希函数数={self.hash_function_count}")
```

```
def __init_manual(self, bit_set_size: int, hash_function_count: int):
    """
    构造函数 - 手动指定参数
    :param bit_set_size: 位数组大小
    :param hash_function_count: 哈希函数数量
    """
    if bit_set_size <= 0:
        raise ValueError("位数组大小必须大于 0")
    if hash_function_count <= 0:
        raise ValueError("哈希函数数量必须大于 0")

    self.bit_set_size = bit_set_size
    self.hash_function_count = hash_function_count
    self.expected_element_count = 0  # 未知预期数量
    self.actual_element_count = 0
    self.bit_set = [False] * self.bit_set_size

    # 初始化哈希种子
    random.seed(42)
    self.seeds = [random.randint(1, 1000000) for _ in range(self.hash_function_count)]

def _hash(self, element: str, seed: int) -> int:
    """
    哈希函数 - 使用简单的字符串哈希算法
    :param element: 输入字符串
    :param seed: 哈希种子
    :return: 哈希值
    """
    hash_value = seed
    for char in element:
        hash_value = (hash_value * 31 + ord(char)) % (2**32)
    return hash_value

def add(self, element: str):
    """
    添加元素到布隆过滤器
    :param element: 要添加的元素
    :raises ValueError: 如果元素为空
    """
    if not element:
        raise ValueError("元素不能为空")

    # 计算所有哈希位置并设置为 True
```

```
        for i in range(self.hash_function_count):
            hash_value = self._hash(element, self.seeds[i])
            position = hash_value % self.bit_set_size
            self.bit_set[position] = True

        self.actual_element_count += 1

    def might_contain(self, element: str) -> bool:
        """
        检查元素是否可能在布隆过滤器中
        :param element: 要检查的元素
        :return: True 如果元素可能存在，False 如果元素一定不存在
        :raises ValueError: 如果元素为空
        """
        if not element:
            raise ValueError("元素不能为空")

        # 检查所有哈希位置是否都为 True
        for i in range(self.hash_function_count):
            hash_value = self._hash(element, self.seeds[i])
            position = hash_value % self.bit_set_size
            if not self.bit_set[position]:
                return False  # 如果有一个位置为 False，元素一定不存在

        return True  # 所有位置都为 True，元素可能存在

    def get_status(self) -> str:
        """
        获取布隆过滤器的状态信息
        :return: 状态信息字符串
        """
        set_bits = sum(1 for bit in self.bit_set if bit)
        fill_ratio = set_bits / self.bit_set_size

        # 计算当前误判率
        current_false_positive_rate = fill_ratio ** self.hash_function_count

        result = []
        result.append("布隆过滤器状态:")
        result.append(f"位数组大小: {self.bit_set_size}")
        result.append(f"哈希函数数量: {self.hash_function_count}")
        result.append(f"预期元素数量: {self.expected_element_count}")
        result.append(f"实际元素数量: {self.actual_element_count}")
```

```
    result.append(f"已设置位数: {set_bits}")
    result.append(f"填充比例: {fill_ratio:.6f}")
    result.append(f"当前误判率: {current_false_positive_rate:.6f}")

    return "\n".join(result)

def performance_test(self, test_element_count: int):
    """
    性能测试 - 测试布隆过滤器的误判率
    :param test_element_count: 测试元素数量
    """
    if test_element_count <= 0:
        print("测试元素数量必须大于 0")
        return

    print("=== 布隆过滤器性能测试 ===")
    print(f"测试元素数量: {test_element_count}")

    # 添加测试元素
    false_positives = 0
    true_negatives = 0

    # 添加真实元素
    for i in range(test_element_count):
        self.add(f"real{i}")

    # 测试不存在元素
    for i in range(test_element_count):
        fake_element = f"fake{i}"
        if self.might_contain(fake_element):
            false_positives += 1  # 误判
        else:
            true_negatives += 1  # 正确判断

    actual_false_positive_rate = false_positives / test_element_count

    print("测试结果:")
    print(f"误判数量: {false_positives}")
    print(f"正确判断数量: {true_negatives}")
    print(f"实际误判率: {actual_false_positive_rate:.6f}")

    set_bits = sum(1 for bit in self.bit_set if bit)
    theoretical_false_positive_rate = (set_bits / self.bit_set_size) **
```

```
self.hash_function_count
        print(f"理论误判率: {theoretical_false_positive_rate:.6f}")

        print(self.get_status())

    def get_bit_set_size(self) -> int:
        """
        获取位数组大小
        :return: 位数组大小
        """
        return self.bit_set_size

    def get_hash_function_count(self) -> int:
        """
        获取哈希函数数量
        :return: 哈希函数数量
        """
        return self.hash_function_count

    def get_actual_element_count(self) -> int:
        """
        获取实际元素数量
        :return: 实际元素数量
        """
        return self.actual_element_count

def test_bloom_filter():
    """
    单元测试函数
    """
    print("=== 布隆过滤器单元测试 ===")

    # 测试 1: 基本功能测试
    print("测试 1: 基本功能测试")
    bf = BloomFilter(1000, 0.01)

    # 添加元素
    bf.add("apple")
    bf.add("banana")
    bf.add("cherry")

    # 检查存在的元素
    if bf.might_contain("apple") and bf.might_contain("banana") and bf.might_contain("cherry"):
```

```
    print("存在性检查测试通过")
else:
    print("存在性检查测试失败")

# 检查不存在的元素
if not bf.might_contain("orange") and not bf.might_contain("grape"):
    print("不存在性检查测试通过")
else:
    print("不存在性检查测试失败")

# 测试 2: 性能测试
print("\n 测试 2: 性能测试")
bf.performance_test(1000)

# 测试 3: 不同参数对比
print("\n 测试 3: 不同参数对比")

bf1 = BloomFilter(1000, 0.1)   # 高误判率
bf2 = BloomFilter(1000, 0.01)  # 低误判率
bf3 = BloomFilter(1000, 0.001) # 极低误判率

# 添加相同元素
for i in range(500):
    bf1.add(f"test{i}")
    bf2.add(f"test{i}")
    bf3.add(f"test{i}")

# 测试误判率
fp1, fp2, fp3 = 0, 0, 0
for i in range(500, 1000):
    if bf1.might_contain(f"test{i}"):
        fp1 += 1
    if bf2.might_contain(f"test{i}"):
        fp2 += 1
    if bf3.might_contain(f"test{i}"):
        fp3 += 1

print("不同误判率配置的测试结果:")
print(f"误判率 0.1: 实际误判率={fp1/500:.4f}")
print(f"误判率 0.01: 实际误判率={fp2/500:.4f}")
print(f"误判率 0.001: 实际误判率={fp3/500:.4f}")

# 测试 4: 异常处理测试
```

```
    print("\n测试4: 异常处理测试")
    try:
        bf.add("")  # 空元素
        print("异常处理测试失败")
    except ValueError as e:
        print(f"异常处理测试通过: {e}")

    print("=== 单元测试完成 ===")

def performance_test():
    """
    性能测试函数
    """
    import time

    print("=== 布隆过滤器性能测试 ===")

    # 测试不同规模的布隆过滤器
    test_cases = [
        (1000, 0.01),
        (10000, 0.01),
        (100000, 0.01)
    ]

    for expected_count, false_positive_rate in test_cases:
        print(f"\n测试规模: 预期元素数={expected_count}, 误判率={false_positive_rate}")

        bf = BloomFilter(expected_count, false_positive_rate)

        # 测试添加性能
        start_time = time.time()

        for i in range(expected_count):
            bf.add(f"element{i}")

        end_time = time.time()
        add_duration = (end_time - start_time) * 1000  # 转换为毫秒

        # 测试查询性能
        start_time = time.time()

        found = 0
        for i in range(expected_count):
```

```
            if bf.might_contain(f"element{i}"):
                found += 1

        end_time = time.time()
        query_duration = (end_time - start_time) * 1000  # 转换为毫秒

        print("性能测试结果:")
        print(f"添加时间: {add_duration:.2f} 毫秒")
        print(f"平均添加时间: {add_duration / expected_count:.4f} 毫秒/元素")
        print(f"查询时间: {query_duration:.2f} 毫秒")
        print(f"平均查询时间: {query_duration / expected_count:.4f} 毫秒/元素")
        print(f"正确查询数量: {found}/{expected_count}")

    print("=== 性能测试完成 ===")

if __name__ == "__main__":
    # 运行单元测试
    test_bloom_filter()

    # 运行性能测试
    performance_test()

    # 演示示例
    print("=== 布隆过滤器演示 ===")

    # 创建一个预期处理 10000 个元素，误判率为 1%的布隆过滤器
    bloom_filter = BloomFilter(10000, 0.01)

    # 添加一些 URL 到布隆过滤器（模拟网页去重）
    urls = [
        "https://example.com/page1",
        "https://example.com/page2",
        "https://example.com/page3",
        "https://example.com/page4",
        "https://example.com/page5"
    ]

    for url in urls:
        bloom_filter.add(url)
        print(f"添加 URL: {url}")

    # 检查 URL 是否已存在
    test_urls = [
```

```
        "https://example.com/page1",     # 已存在
        "https://example.com/page6",     # 不存在
        "https://example.com/page3",     # 已存在
        "https://example.com/page7"      # 不存在
    ]

    print("\nURL存在性检查:")
    for url in test_urls:
        exists = bloom_filter.might_contain(url)
        print(f"URL {url}: {'可能已存在' if exists else '一定不存在'}")

    # 显示布隆过滤器状态
    print(f"\n{bloom_filter.get_status()}")

    # 性能测试
    bloom_filter.performance_test(1000)
```

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

文件: Code16_PerfectHashing.java

```
==========================================
package class111;

// 完美哈希算法实现 (Java版本)
// 题目来源: 数据库索引、编译器符号表、静态字典查找
// 应用场景: 静态数据集的高效查找、编译时符号表、数据库索引优化
// 题目描述: 实现完美哈希算法，为静态数据集构建无冲突的哈希函数
//
// 解题思路:
// 1. 使用两级哈希结构：第一级哈希将元素分配到桶中，第二级为每个桶构建完美哈希
// 2. 第一级使用通用哈希函数，第二级为每个桶构建自定义哈希函数
// 3. 通过调整哈希参数确保每个桶内无冲突
//
// 时间复杂度分析:
// - 构建时间: O(n^2) 最坏情况，但实际中通常为 O(n)
// - 查找时间: O(1) 最坏情况
//
// 空间复杂度: O(n)
//
// 工程化考量:
// 1. 静态数据集: 完美哈希适用于数据不变化的场景
// 2. 构建成本: 构建过程较复杂，但查询效率极高
// 3. 内存优化: 使用紧凑的数据结构存储哈希参数
```

```
// 4. 适用场景：适合编译时符号表、静态字典等

import java.util.*;

public class Code16_PerfectHashing {

    // 第一级哈希函数参数
    private int a1, b1, p1, m1;

    // 第二级哈希函数参数数组
    private int[] a2, b2, p2, m2;

    // 第二级哈希表数组
    private String[][] secondLevelTables;

    // 数据集大小
    private int n;

    /**
     * 构造函数 - 为给定的静态数据集构建完美哈希
     * @param keys 静态数据集（不能包含重复元素）
     * @throws IllegalArgumentException 如果数据集为空或包含重复元素
     */
    public Code16_PerfectHashing(String[] keys) {
        if (keys == null || keys.length == 0) {
            throw new IllegalArgumentException("数据集不能为空");
        }

        this.n = keys.length;

        // 检查重复元素
        Set<String> keySet = new HashSet<>(Arrays.asList(keys));
        if (keySet.size() != n) {
            throw new IllegalArgumentException("数据集包含重复元素");
        }

        // 初始化第一级哈希参数
        initializeFirstLevel();

        // 构建完美哈希结构
        buildPerfectHash(keys);
    }
```

```
/**
 * 初始化第一级哈希参数
 */
private void initializeFirstLevel() {
    Random random = new Random(42);

    // 选择足够大的质数
    p1 = nextPrime(n * n);
    m1 = n; // 第一级桶数量等于元素数量

    a1 = random.nextInt(p1 - 1) + 1; // a ∈ [1, p-1]
    b1 = random.nextInt(p1);           // b ∈ [0, p-1]
}

/**
 * 构建完美哈希结构
 * @param keys 静态数据集
 */
private void buildPerfectHash(String[] keys) {
    // 第一级哈希：将元素分配到桶中
    List<List<String>> buckets = new ArrayList<>(m1);
    for (int i = 0; i < m1; i++) {
        buckets.add(new ArrayList<>());
    }

    // 分配元素到桶中
    for (String key : keys) {
        int bucketIndex = firstLevelHash(key);
        buckets.get(bucketIndex).add(key);
    }

    // 初始化第二级结构
    a2 = new int[m1];
    b2 = new int[m1];
    p2 = new int[m1];
    m2 = new int[m1];
    secondLevelTables = new String[m1][];

    Random random = new Random(42);

    // 为每个桶构建第二级完美哈希
    for (int i = 0; i < m1; i++) {
        List<String> bucket = buckets.get(i);
```

```
int bucketSize = bucket.size();

if (bucketSize == 0) {
    // 空桶
    m2[i] = 0;
    secondLevelTables[i] = new String[0];
    continue;
}

// 计算第二级哈希表大小：桶大小的平方（确保无冲突）
m2[i] = bucketSize * bucketSize;

// 选择质数
p2[i] = nextPrime(m2[i]);

boolean collisionFree = false;
int attempts = 0;

// 尝试不同的哈希参数直到无冲突
while (!collisionFree && attempts < 100) {
    a2[i] = random.nextInt(p2[i] - 1) + 1;
    b2[i] = random.nextInt(p2[i]);

    String[] table = new String[m2[i]];
    collisionFree = true;

    // 测试当前参数是否会产生冲突
    for (String key : bucket) {
        int hash = secondLevelHash(key, i);
        if (table[hash] != null) {
            // 发生冲突，重新选择参数
            collisionFree = false;
            break;
        }
        table[hash] = key;
    }

    if (collisionFree) {
        // 无冲突，保存结果
        secondLevelTables[i] = table;
    }

    attempts++;
```

```
        }

        if (!collisionFree) {
            throw new RuntimeException("无法为桶 " + i + " 构建无冲突的完美哈希");
        }
    }

    System.out.println("完美哈希构建完成，数据集大小: " + n);
}

/**
 * 第一级哈希函数
 * @param key 键
 * @return 桶索引
 */
private int firstLevelHash(String key) {
    long hash = 0;
    for (char c : key.toCharArray()) {
        hash = (hash * 31 + c) % p1;
    }
    hash = (a1 * hash + b1) % p1;
    return (int) (hash % m1);
}

/**
 * 第二级哈希函数
 * @param key 键
 * @param bucketIndex 桶索引
 * @return 第二级哈希表中的位置
 */
private int secondLevelHash(String key, int bucketIndex) {
    long hash = 0;
    for (char c : key.toCharArray()) {
        hash = (hash * 31 + c) % p2[bucketIndex];
    }
    hash = (a2[bucketIndex] * hash + b2[bucketIndex]) % p2[bucketIndex];
    return (int) (hash % m2[bucketIndex]);
}

/**
 * 查找元素是否存在于完美哈希表中
 * @param key 要查找的键
 * @return true 如果存在，false 如果不存在
```

```
     */
    public boolean contains(String key) {
        if (key == null) {
            return false;
        }

        // 第一级哈希确定桶
        int bucketIndex = firstLevelHash(key);

        // 检查桶是否为空
        if (m2[bucketIndex] == 0) {
            return false;
        }

        // 第二级哈希定位元素
        int position = secondLevelHash(key, bucketIndex);

        // 检查该位置是否存储了目标元素
        return key.equals(secondLevelTables[bucketIndex][position]);
    }

    /**
     * 查找下一个质数
     * @param n 起始数字
     * @return 大于等于 n 的最小质数
     */
    private int nextPrime(int n) {
        if (n <= 2) return 2;
        if (n % 2 == 0) n++;

        while (!isPrime(n)) {
            n += 2;
        }
        return n;
    }

    /**
     * 判断是否为质数
     * @param n 数字
     * @return true 如果是质数
     */
    private boolean isPrime(int n) {
        if (n <= 1) return false;
```

```
        if (n <= 3) return true;
        if (n % 2 == 0 || n % 3 == 0) return false;

        for (int i = 5; i * i <= n; i += 6) {
            if (n % i == 0 || n % (i + 2) == 0) {
                return false;
            }
        }
        return true;
    }

    /**
     * 获取完美哈希表的状态信息
     * @return 状态信息字符串
     */
    public String getStatus() {
        StringBuilder sb = new StringBuilder();
        sb.append("完美哈希表状态:\n");
        sb.append("数据集大小: ").append(n).append("\n");
        sb.append("第一级桶数量: ").append(m1).append("\n");
        sb.append("第一级哈希参数: a1=").append(a1).append(", b1=").append(b1).append(",
p1=").append(p1).append("\n");

        // 统计桶分布
        int emptyBuckets = 0;
        int maxBucketSize = 0;
        long totalSecondLevelSize = 0;

        for (int i = 0; i < m1; i++) {
            if (m2[i] == 0) {
                emptyBuckets++;
            } else {
                int bucketSize = (int) Math.sqrt(m2[i]);
                maxBucketSize = Math.max(maxBucketSize, bucketSize);
                totalSecondLevelSize += m2[i];
            }
        }

        sb.append("空桶数量: ").append(emptyBuckets).append("\n");
        sb.append("最大桶大小: ").append(maxBucketSize).append("\n");
        sb.append("第二级总空间: ").append(totalSecondLevelSize).append("\n");
        sb.append("空间利用率: ").append(String.format("%.2f%%", (double) n /
totalSecondLevelSize * 100)).append("\n");
```

```
        return sb.toString();
    }

    /**
     * 性能测试 - 测试查找性能
     * @param testKeys 测试键数组
     */
    public void performanceTest(String[] testKeys) {
        if (testKeys == null || testKeys.length == 0) {
            System.out.println("测试键数组不能为空");
            return;
        }

        System.out.println("=== 完美哈希性能测试 ===");
        System.out.println("测试键数量: " + testKeys.length);

        long startTime = System.nanoTime();

        int found = 0;
        int notFound = 0;

        for (String key : testKeys) {
            if (contains(key)) {
                found++;
            } else {
                notFound++;
            }
        }

        long endTime = System.nanoTime();
        long duration = endTime - startTime;

        System.out.println("测试结果:");
        System.out.println("找到元素: " + found);
        System.out.println("未找到元素: " + notFound);
        System.out.println("总查找时间: " + duration + " 纳秒");
        System.out.println("平均查找时间: " + (double) duration / testKeys.length + " 纳秒/次");
        System.out.println("查找吞吐量: " + String.format("%.2f", (double) testKeys.length /
(duration / 1e9)) + " 次/秒");
    }

    /**
```

```
     * 单元测试方法
     */
    public static void test() {
        System.out.println("=== 完美哈希单元测试 ===");

        // 测试 1: 基本功能测试
        System.out.println("测试 1: 基本功能测试");
        String[] keys = {"apple", "banana", "cherry", "date", "elderberry", "fig", "grape"};
        Code16_PerfectHashing ph = new Code16_PerfectHashing(keys);

        // 测试存在的元素
        for (String key : keys) {
            assert ph.contains(key) : "应该包含 " + key;
        }

        // 测试不存在的元素
        assert !ph.contains("honeydew") : "不应该包含 honeydew";
        assert !ph.contains("kiwi") : "不应该包含 kiwi";

        System.out.println("基本功能测试通过");

        // 测试 2: 性能测试
        System.out.println("\n测试 2: 性能测试");
        ph.performanceTest(keys);

        // 测试 3: 大规模数据测试
        System.out.println("\n测试 3: 大规模数据测试");
        String[] largeKeys = new String[1000];
        for (int i = 0; i < 1000; i++) {
            largeKeys[i] = "key" + i;
        }

        Code16_PerfectHashing largePH = new Code16_PerfectHashing(largeKeys);
        largePH.performanceTest(largeKeys);

        System.out.println(largePH.getStatus());

        System.out.println("=== 单元测试完成 ===");
    }

    public static void main(String[] args) {
        if (args.length > 0 && "test".equals(args[0])) {
            test();
```

```
            return;
        }

        // 演示示例
        System.out.println("=== 完美哈希演示 ===");

        // 创建一个编程语言关键字的完美哈希表
        String[] programmingKeywords = {
            "if", "else", "for", "while", "do", "switch", "case", "break",
            "continue", "return", "class", "interface", "extends", "implements",
            "public", "private", "protected", "static", "final", "abstract",
            "void", "int", "long", "float", "double", "boolean", "char", "String"
        };

        Code16_PerfectHashing keywordTable = new Code16_PerfectHashing(programmingKeywords);

        System.out.println("构建编程语言关键字完美哈希表");
        System.out.println(keywordTable.getStatus());

        // 测试关键字查找
        String[] testKeywords = {"if", "for", "class", "xyz", "abc", "static"};

        System.out.println("\n关键字查找测试:");
        for (String keyword : testKeywords) {
            boolean exists = keywordTable.contains(keyword);
            System.out.printf("关键字 '%s': %s%n", keyword, exists ? "存在" : "不存在");
        }

        // 性能测试
        keywordTable.performanceTest(programmingKeywords);
    }
}
```

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

文件: Code17_RollingHash.java

```
==========================================
package class111;

// 滚动哈希算法实现 (Java 版本)
// 题目来源: 字符串匹配、子串查找、重复检测
// 应用场景: Rabin-Karp 算法、字符串相似度比较、重复子串检测
// 题目描述: 实现滚动哈希算法，支持高效计算字符串子串的哈希值
```

```
//
// 解题思路:
// 1. 使用多项式哈希函数计算字符串哈希值
// 2. 通过滚动窗口技术高效更新哈希值
// 3. 支持快速计算任意子串的哈希值
// 4. 使用双哈希技术减少哈希冲突
//
// 时间复杂度分析:
// - 初始化: O(n)，其中 n 是字符串长度
// - 计算子串哈希: O(1)
// - 滚动更新: O(1)
//
// 空间复杂度: O(n)
//
// 工程化考量:
// 1. 哈希冲突: 使用双哈希技术减少冲突概率
// 2. 数值溢出: 使用模运算防止整数溢出
// 3. 性能优化: 预计算幂次值加速滚动计算
// 4. 适用场景: 大规模字符串处理、模式匹配

import java.util.*;

public class Code17_RollingHash {

    // 原始字符串
    private String str;

    // 字符串长度
    private int n;

    // 基数（通常选择质数）
    private final int BASE1 = 131;
    private final int BASE2 = 13131;

    // 模数（选择大质数）
    private final long MOD1 = 1000000007L;
    private final long MOD2 = 1000000009L;

    // 前缀哈希数组
    private long[] prefixHash1, prefixHash2;

    // 幂次数组，用于快速计算 BASE^k
    private long[] power1, power2;
```

```
/**
 * 构造函数 - 为给定字符串构建滚动哈希
 * @param str 输入字符串
 */
public Code17_RollingHash(String str) {
    if (str == null) {
        throw new IllegalArgumentException("字符串不能为空");
    }

    this.str = str;
    this.n = str.length();

    // 初始化数组
    prefixHash1 = new long[n + 1];
    prefixHash2 = new long[n + 1];
    power1 = new long[n + 1];
    power2 = new long[n + 1];

    // 初始化幂次数组
    power1[0] = 1;
    power2[0] = 1;

    // 计算前缀哈希
    for (int i = 1; i <= n; i++) {
        char c = str.charAt(i - 1);

        // 更新幂次值
        power1[i] = (power1[i - 1] * BASE1) % MOD1;
        power2[i] = (power2[i - 1] * BASE2) % MOD2;

        // 更新前缀哈希
        prefixHash1[i] = (prefixHash1[i - 1] * BASE1 + c) % MOD1;
        prefixHash2[i] = (prefixHash2[i - 1] * BASE2 + c) % MOD2;
    }
}

/**
 * 计算子串的哈希值（双哈希）
 * @param l 子串起始位置（0-based）
 * @param r 子串结束位置（0-based，包含）
 * @return 包含两个哈希值的数组 [hash1, hash2]
 * @throws IllegalArgumentException 如果位置参数无效
```

```
     */
    public long[] getHash(int l, int r) {
        if (l < 0 || r >= n || l > r) {
            throw new IllegalArgumentException("位置参数无效: l=" + l + ", r=" + r);
        }

        // 计算第一个哈希值
        long hash1 = (prefixHash1[r + 1] - prefixHash1[l] * power1[r - l + 1] % MOD1 + MOD1) %
MOD1;

        // 计算第二个哈希值
        long hash2 = (prefixHash2[r + 1] - prefixHash2[l] * power2[r - l + 1] % MOD2 + MOD2) %
MOD2;

        return new long[]{hash1, hash2};
    }

    /**
     * 检查两个子串是否相等（使用哈希比较）
     * @param l1 第一个子串起始位置
     * @param r1 第一个子串结束位置
     * @param l2 第二个子串起始位置
     * @param r2 第二个子串结束位置
     * @return true 如果两个子串相等
     */
    public boolean equals(int l1, int r1, int l2, int r2) {
        if (r1 - l1 != r2 - l2) {
            return false; // 长度不同，肯定不相等
        }

        long[] hash1 = getHash(l1, r1);
        long[] hash2 = getHash(l2, r2);

        return hash1[0] == hash2[0] && hash1[1] == hash2[1];
    }

    /**
     * 查找最长重复子串的长度（使用二分搜索）
     * @return 最长重复子串的长度
     */
    public int longestRepeatingSubstring() {
        int left = 1, right = n - 1;
        int result = 0;
```

```
    while (left <= right) {
        int mid = left + (right - left) / 2;

        if (hasRepeatingSubstring(mid)) {
            result = mid;
            left = mid + 1;
        } else {
            right = mid - 1;
        }
    }

    return result;
}

/**
 * 检查是否存在长度为 len 的重复子串
 * @param len 子串长度
 * @return true 如果存在重复子串
 */
private boolean hasRepeatingSubstring(int len) {
    if (len <= 0 || len > n) {
        return false;
    }

    // 使用哈希集合检测重复
    Set<String> seen = new HashSet<>();

    for (int i = 0; i <= n - len; i++) {
        long[] hash = getHash(i, i + len - 1);
        String hashKey = hash[0] + "_" + hash[1];

        if (seen.contains(hashKey)) {
            // 验证是否真的相等（防止哈希冲突）
            for (String existing : seen) {
                if (existing.equals(hashKey)) {
                    // 这里可以添加实际字符串比较来确认
                    return true;
                }
            }
        }

        seen.add(hashKey);
```

```
    }

    return false;
}

/**
 * 实现 Rabin-Karp 字符串匹配算法
 * @param pattern 要匹配的模式串
 * @return 模式串在文本中出现的所有起始位置
 */
public List<Integer> rabinKarp(String pattern) {
    List<Integer> positions = new ArrayList<>();

    if (pattern == null || pattern.isEmpty() || pattern.length() > n) {
        return positions;
    }

    int m = pattern.length();

    // 计算模式串的哈希值
    RollingHash patternHash = new RollingHash(pattern);
    long[] patternHashValue = patternHash.getHash(0, m - 1);

    // 滑动窗口匹配
    for (int i = 0; i <= n - m; i++) {
        long[] textHash = getHash(i, i + m - 1);

        // 比较哈希值
        if (textHash[0] == patternHashValue[0] && textHash[1] == patternHashValue[1]) {
            // 哈希值匹配，验证实际字符串是否相等（防止哈希冲突）
            if (str.substring(i, i + m).equals(pattern)) {
                positions.add(i);
            }
        }
    }

    return positions;
}

/**
 * 获取滚动哈希的状态信息
 * @return 状态信息字符串
 */
```

```
public String getStatus() {
    StringBuilder sb = new StringBuilder();
    sb.append("滚动哈希状态:\n");
    sb.append("字符串长度: ").append(n).append("\n");
    sb.append("基数: BASE1=").append(BASE1).append(", BASE2=").append(BASE2).append("\n");
    sb.append("模数: MOD1=").append(MOD1).append(", MOD2=").append(MOD2).append("\n");

    // 显示一些示例哈希值
    if (n > 0) {
        sb.append("示例哈希值:\n");
        for (int len : new int[]{1, Math.min(5, n), Math.min(10, n)}) {
            if (len <= n) {
                long[] hash = getHash(0, len - 1);
                sb.append("前").append(len).append("个字符的哈希: [").append(hash[0])
                  .append(", ").append(hash[1]).append("]\n");
            }
        }
    }

    return sb.toString();
}

/**
 * 性能测试 - 测试各种操作的速度
 */
public void performanceTest() {
    System.out.println("=== 滚动哈希性能测试 ===");

    // 测试1: 哈希计算性能
    long startTime = System.nanoTime();

    int testCount = 10000;
    for (int i = 0; i < testCount; i++) {
        int l = i % (n - 10);
        int r = l + 9;
        if (r < n) {
            getHash(l, r);
        }
    }

    long hashTime = System.nanoTime() - startTime;
    System.out.printf("哈希计算性能: %.2f 纳秒/次%n", (double) hashTime / testCount);
```

```
        // 测试2: 字符串匹配性能
        startTime = System.nanoTime();
        List<Integer> matches = rabinKarp("test");
        long matchTime = System.nanoTime() - startTime;
        System.out.printf("Rabin-Karp匹配时间: %d 纳秒, 找到 %d 个匹配%n", matchTime,
matches.size());

        // 测试3: 重复子串检测性能
        startTime = System.nanoTime();
        int longestRepeat = longestRepeatingSubstring();
        long repeatTime = System.nanoTime() - startTime;
        System.out.printf("最长重复子串检测时间: %d 纳秒, 最长长度: %d%n", repeatTime,
longestRepeat);
    }

    /**
     * 单元测试方法
     */
    public static void test() {
        System.out.println("=== 滚动哈希单元测试 ===");

        // 测试1: 基本功能测试
        System.out.println("测试1: 基本功能测试");
        String testStr = "abcdefghij";
        Code17_RollingHash rh = new Code17_RollingHash(testStr);

        // 测试相同子串的哈希值相等
        long[] hash1 = rh.getHash(0, 2); // "abc"
        long[] hash2 = rh.getHash(0, 2); // "abc"
        assert hash1[0] == hash2[0] && hash1[1] == hash2[1] : "相同子串哈希值应该相等";

        // 测试不同子串的哈希值不同
        long[] hash3 = rh.getHash(0, 2); // "abc"
        long[] hash4 = rh.getHash(1, 3); // "bcd"
        assert !(hash3[0] == hash4[0] && hash3[1] == hash4[1]) : "不同子串哈希值应该不同";

        System.out.println("基本功能测试通过");

        // 测试2: 字符串匹配测试
        System.out.println("\n测试2: 字符串匹配测试");
        String text = "ababcabcababababd";
        String pattern = "ababd";
```

```
    Code17_RollingHash rh2 = new Code17_RollingHash(text);
    List<Integer> positions = rh2.rabinKarp(pattern);

    assert positions.size() == 1 && positions.get(0) == 10 : "模式串应该在位置 10 找到";
    System.out.println("字符串匹配测试通过，找到位置: " + positions);

    // 测试 3: 重复子串检测
    System.out.println("\n 测试 3: 重复子串检测");
    String repeatStr = "abcabcabc";
    Code17_RollingHash rh3 = new Code17_RollingHash(repeatStr);
    int longestRepeat = rh3.longestRepeatingSubstring();

    assert longestRepeat == 6 : "最长重复子串应该是 6（abcabc）";
    System.out.println("重复子串检测通过，最长长度: " + longestRepeat);

    System.out.println("=== 单元测试完成 ===");
}

public static void main(String[] args) {
    if (args.length > 0 && "test".equals(args[0])) {
        test();
        return;
    }

    // 演示示例
    System.out.println("=== 滚动哈希演示 ===");

    // 创建一个 DNA 序列的滚动哈希
    String dnaSequence = "ATCGATCGATCGATCGATCGATCG";
    Code17_RollingHash dnaHash = new Code17_RollingHash(dnaSequence);

    System.out.println("DNA 序列: " + dnaSequence);
    System.out.println(dnaHash.getStatus());

    // 演示子串哈希计算
    System.out.println("\n 子串哈希计算演示:");
    int[][] substrings = {{0, 5}, {5, 10}, {10, 15}};

    for (int[] range : substrings) {
        int l = range[0], r = range[1];
        long[] hash = dnaHash.getHash(l, r);
        String substring = dnaSequence.substring(l, r + 1);
        System.out.printf("子串 '%s' (位置%d-%d) 的哈希值: [%d, %d]%n",
```

```
                substring, l, r, hash[0], hash[1]);
        }

        // 演示字符串匹配
        System.out.println("\n字符串匹配演示:");
        String pattern = "ATCG";
        List<Integer> matches = dnaHash.rabinKarp(pattern);
        System.out.println("模式串 '" + pattern + "' 在DNA序列中出现的位置: " + matches);

        // 演示重复子串检测
        System.out.println("\n重复子串检测演示:");
        int longestRepeat = dnaHash.longestRepeatingSubstring();
        System.out.println("最长重复子串长度: " + longestRepeat);

        // 性能测试
        dnaHash.performanceTest();
    }
}
```

==========

文件: Code18_RangeSumQueryMutable.cpp

==========

```
// Range Sum Query - Mutable (区间求和 - 可变)
// 题目来源: LeetCode 307. Range Sum Query - Mutable
// 题目链接: https://leetcode.com/problems/range-sum-query-mutable/
// 题目链接: https://leetcode.cn/problems/range-sum-query-mutable
//
// 题目描述:
// 给你一个数组 nums ，请你完成两类查询:
// 1. 一类查询要求更新数组 nums 下标对应的值
// 2. 一类查询要求返回数组 nums 中索引 left 和索引 right 之间（包含）的 nums 元素的和，其中 left <= right
//
// 实现 NumArray 类:
// NumArray(int[] nums) 用整数数组 nums 初始化对象
// void update(int index, int val) 将 nums[index] 的值更新为 val
// int sumRange(int left, int right) 返回数组 nums 中索引 left 和索引 right 之间（包含）的 nums 元素的和
//
// 解题思路:
// 1. 使用线段树实现区间求和和单点更新
// 2. 线段树的每个节点存储对应区间的元素和
```

```
// 3. 更新操作时，从根节点开始，找到对应的叶子节点并更新，然后逐层向上更新父节点
// 4. 查询操作时，从根节点开始，根据查询区间与当前节点区间的关系进行递归查询
//
// 时间复杂度分析:
// - 构建线段树: O(n)
// - 单点更新: O(log n)
// - 区间查询: O(log n)
// 空间复杂度: O(n)

#include <iostream>
#include <vector>
using namespace std;

class NumArray {
private:
    vector<int> tree;
    vector<int> data;
    int n;

    // 构建线段树
    void buildTree(int treeIndex, int l, int r) {
        if (l == r) {
            tree[treeIndex] = data[l];
            return;
        }

        int mid = l + (r - l) / 2;
        int leftTreeIndex = 2 * treeIndex + 1;
        int rightTreeIndex = 2 * treeIndex + 2;

        // 构建左子树
        buildTree(leftTreeIndex, l, mid);
        // 构建右子树
        buildTree(rightTreeIndex, mid + 1, r);

        // 当前节点的值等于左右子树值的和
        tree[treeIndex] = tree[leftTreeIndex] + tree[rightTreeIndex];
    }

    // 更新线段树
    void updateTree(int treeIndex, int l, int r, int index, int val) {
        if (l == r) {
            tree[treeIndex] = val;
```

```
        return;
    }

    int mid = l + (r - l) / 2;
    int leftTreeIndex = 2 * treeIndex + 1;
    int rightTreeIndex = 2 * treeIndex + 2;

    if (index >= l && index <= mid) {
        // 要更新的索引在左子树
        updateTree(leftTreeIndex, l, mid, index, val);
    } else {
        // 要更新的索引在右子树
        updateTree(rightTreeIndex, mid + 1, r, index, val);
    }

    // 更新当前节点的值
    tree[treeIndex] = tree[leftTreeIndex] + tree[rightTreeIndex];
}

// 查询区间和
int queryTree(int treeIndex, int l, int r, int queryL, int queryR) {
    if (l == queryL && r == queryR) {
        return tree[treeIndex];
    }

    int mid = l + (r - l) / 2;
    int leftTreeIndex = 2 * treeIndex + 1;
    int rightTreeIndex = 2 * treeIndex + 2;

    if (queryR <= mid) {
        // 查询区间完全在左子树
        return queryTree(leftTreeIndex, l, mid, queryL, queryR);
    } else if (queryL > mid) {
        // 查询区间完全在右子树
        return queryTree(rightTreeIndex, mid + 1, r, queryL, queryR);
    } else {
        // 查询区间跨越左右子树
        int leftResult = queryTree(leftTreeIndex, l, mid, queryL, mid);
        int rightResult = queryTree(rightTreeIndex, mid + 1, r, mid + 1, queryR);
        return leftResult + rightResult;
    }
}
```

```
public:
    NumArray(vector<int>& nums) {
        if (nums.size() > 0) {
            n = nums.size();
            data = nums;
            tree.resize(4 * n);
            buildTree(0, 0, n - 1);
        }
    }

    void update(int index, int val) {
        if (n == 0) return;
        data[index] = val;
        updateTree(0, 0, n - 1, index, val);
    }

    int sumRange(int left, int right) {
        if (n == 0) return 0;
        return queryTree(0, 0, n - 1, left, right);
    }
};

// 测试代码
int main() {
    // 测试用例
    vector<int> nums = {1, 3, 5};
    NumArray numArray(nums);

    cout << "初始数组: [1, 3, 5]" << endl;
    cout << "sumRange(0, 2): " << numArray.sumRange(0, 2) << endl; // 应该返回 9

    numArray.update(1, 2); // nums = [1, 2, 5]
    cout << "更新索引 1 为 2 后: [1, 2, 5]" << endl;
    cout << "sumRange(0, 2): " << numArray.sumRange(0, 2) << endl; // 应该返回 8

    return 0;
}

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

文件: Code18_RangeSumQueryMutable.java
==========================================
// Range Sum Query - Mutable (区间求和 - 可变)
```

```
// 题目来源: LeetCode 307. Range Sum Query - Mutable
// 题目链接: https://leetcode.com/problems/range-sum-query-mutable/
// 题目链接: https://leetcode.cn/problems/range-sum-query-mutable
//
// 题目描述:
// 给你一个数组 nums ，请你完成两类查询:
// 1. 一类查询要求更新数组 nums 下标对应的值
// 2. 一类查询要求返回数组 nums 中索引 left 和索引 right 之间（包含）的 nums 元素的和，其中 left <= right
//
// 实现 NumArray 类：
// NumArray(int[] nums) 用整数数组 nums 初始化对象
// void update(int index, int val) 将 nums[index] 的值更新为 val
// int sumRange(int left, int right) 返回数组 nums 中索引 left 和索引 right 之间（包含）的 nums 元素的和
//
// 解题思路:
// 1. 使用线段树实现区间求和和单点更新
// 2. 线段树的每个节点存储对应区间的元素和
// 3. 更新操作时，从根节点开始，找到对应的叶子节点并更新，然后逐层向上更新父节点
// 4. 查询操作时，从根节点开始，根据查询区间与当前节点区间的关系进行递归查询
//
// 时间复杂度分析:
// - 构建线段树: O(n)
// - 单点更新: O(log n)
// - 区间查询: O(log n)
// 空间复杂度: O(n)

class NumArray {
    private int[] tree;
    private int[] data;
    private int n;

    public NumArray(int[] nums) {
        if (nums.length > 0) {
            n = nums.length;
            data = new int[n];
            for (int i = 0; i < n; i++) {
                data[i] = nums[i];
            }
            tree = new int[4 * n];
            buildTree(0, 0, n - 1);
        }
```

```
}

// 构建线段树
private void buildTree(int treeIndex, int l, int r) {
    if (l == r) {
        tree[treeIndex] = data[l];
        return;
    }

    int mid = l + (r - l) / 2;
    int leftTreeIndex = 2 * treeIndex + 1;
    int rightTreeIndex = 2 * treeIndex + 2;

    // 构建左子树
    buildTree(leftTreeIndex, l, mid);
    // 构建右子树
    buildTree(rightTreeIndex, mid + 1, r);

    // 当前节点的值等于左右子树值的和
    tree[treeIndex] = tree[leftTreeIndex] + tree[rightTreeIndex];
}

public void update(int index, int val) {
    if (n == 0) return;
    data[index] = val;
    updateTree(0, 0, n - 1, index, val);
}

// 更新线段树
private void updateTree(int treeIndex, int l, int r, int index, int val) {
    if (l == r) {
        tree[treeIndex] = val;
        return;
    }

    int mid = l + (r - l) / 2;
    int leftTreeIndex = 2 * treeIndex + 1;
    int rightTreeIndex = 2 * treeIndex + 2;

    if (index >= l && index <= mid) {
        // 要更新的索引在左子树
        updateTree(leftTreeIndex, l, mid, index, val);
    } else {
```

```
            // 要更新的索引在右子树
            updateTree(rightTreeIndex, mid + 1, r, index, val);
        }

        // 更新当前节点的值
        tree[treeIndex] = tree[leftTreeIndex] + tree[rightTreeIndex];
    }

    public int sumRange(int left, int right) {
        if (n == 0) return 0;
        return queryTree(0, 0, n - 1, left, right);
    }

    // 查询区间和
    private int queryTree(int treeIndex, int l, int r, int queryL, int queryR) {
        if (l == queryL && r == queryR) {
            return tree[treeIndex];
        }

        int mid = l + (r - l) / 2;
        int leftTreeIndex = 2 * treeIndex + 1;
        int rightTreeIndex = 2 * treeIndex + 2;

        if (queryR <= mid) {
            // 查询区间完全在左子树
            return queryTree(leftTreeIndex, l, mid, queryL, queryR);
        } else if (queryL > mid) {
            // 查询区间完全在右子树
            return queryTree(rightTreeIndex, mid + 1, r, queryL, queryR);
        } else {
            // 查询区间跨越左右子树
            int leftResult = queryTree(leftTreeIndex, l, mid, queryL, mid);
            int rightResult = queryTree(rightTreeIndex, mid + 1, r, mid + 1, queryR);
            return leftResult + rightResult;
        }
    }
}

public class Code18_RangeSumQueryMutable {
    public static void main(String[] args) {
        // 测试用例
        int[] nums = {1, 3, 5};
        NumArray numArray = new NumArray(nums);
```

```
        System.out.println("初始数组: [1, 3, 5]");
        System.out.println("sumRange(0, 2): " + numArray.sumRange(0, 2)); // 应该返回 9

        numArray.update(1, 2); // nums = [1, 2, 5]
        System.out.println("更新索引 1 为 2 后: [1, 2, 5]");
        System.out.println("sumRange(0, 2): " + numArray.sumRange(0, 2)); // 应该返回 8
    }
}
```

==========

文件: Code18_RangeSumQueryMutable.py

==========

```
# Range Sum Query - Mutable (区间求和 - 可变)
# 题目来源: LeetCode 307. Range Sum Query - Mutable
# 题目链接: https://leetcode.com/problems/range-sum-query-mutable/
# 题目链接: https://leetcode.cn/problems/range-sum-query-mutable
#
# 题目描述:
# 给你一个数组 nums ，请你完成两类查询:
# 1. 一类查询要求更新数组 nums 下标对应的值
# 2. 一类查询要求返回数组 nums 中索引 left 和索引 right 之间（包含）的 nums 元素的和，其中 left <= right
#
# 实现 NumArray 类:
# NumArray(int[] nums) 用整数数组 nums 初始化对象
# void update(int index, int val) 将 nums[index] 的值更新为 val
# int sumRange(int left, int right) 返回数组 nums 中索引 left 和索引 right 之间（包含）的 nums 元素的和
#
# 解题思路:
# 1. 使用线段树实现区间求和和单点更新
# 2. 线段树的每个节点存储对应区间的元素和
# 3. 更新操作时，从根节点开始，找到对应的叶子节点并更新，然后逐层向上更新父节点
# 4. 查询操作时，从根节点开始，根据查询区间与当前节点区间的关系进行递归查询
#
# 时间复杂度分析:
# - 构建线段树: O(n)
# - 单点更新: O(log n)
# - 区间查询: O(log n)
# 空间复杂度: O(n)
```

```
class NumArray:
    def __init__(self, nums):
        if len(nums) > 0:
            self.n = len(nums)
            self.data = nums[:]
            self.tree = [0] * (4 * self.n)
            self._build_tree(0, 0, self.n - 1)

    # 构建线段树
    def _build_tree(self, tree_index, l, r):
        if l == r:
            self.tree[tree_index] = self.data[l]
            return

        mid = l + (r - l) // 2
        left_tree_index = 2 * tree_index + 1
        right_tree_index = 2 * tree_index + 2

        # 构建左子树
        self._build_tree(left_tree_index, l, mid)
        # 构建右子树
        self._build_tree(right_tree_index, mid + 1, r)

        # 当前节点的值等于左右子树值的和
        self.tree[tree_index] = self.tree[left_tree_index] + self.tree[right_tree_index]

    def update(self, index, val):
        if self.n == 0:
            return
        self.data[index] = val
        self._update_tree(0, 0, self.n - 1, index, val)

    # 更新线段树
    def _update_tree(self, tree_index, l, r, index, val):
        if l == r:
            self.tree[tree_index] = val
            return

        mid = l + (r - l) // 2
        left_tree_index = 2 * tree_index + 1
        right_tree_index = 2 * tree_index + 2

        if index >= l and index <= mid:
```

```
            # 要更新的索引在左子树
            self._update_tree(left_tree_index, l, mid, index, val)
        else:
            # 要更新的索引在右子树
            self._update_tree(right_tree_index, mid + 1, r, index, val)

        # 更新当前节点的值
        self.tree[tree_index] = self.tree[left_tree_index] + self.tree[right_tree_index]

    def sumRange(self, left, right):
        if self.n == 0:
            return 0
        return self._query_tree(0, 0, self.n - 1, left, right)

    # 查询区间和
    def _query_tree(self, tree_index, l, r, query_l, query_r):
        if l == query_l and r == query_r:
            return self.tree[tree_index]

        mid = l + (r - l) // 2
        left_tree_index = 2 * tree_index + 1
        right_tree_index = 2 * tree_index + 2

        if query_r <= mid:
            # 查询区间完全在左子树
            return self._query_tree(left_tree_index, l, mid, query_l, query_r)
        elif query_l > mid:
            # 查询区间完全在右子树
            return self._query_tree(right_tree_index, mid + 1, r, query_l, query_r)
        else:
            # 查询区间跨越左右子树
            left_result = self._query_tree(left_tree_index, l, mid, query_l, mid)
            right_result = self._query_tree(right_tree_index, mid + 1, r, mid + 1, query_r)
            return left_result + right_result

# 测试代码
if __name__ == "__main__":
    # 测试用例
    nums = [1, 3, 5]
    numArray = NumArray(nums)

    print("初始数组:", nums)
    print("sumRange(0, 2):", numArray.sumRange(0, 2))  # 应该返回 9
```

```
    numArray.update(1, 2)  # nums = [1, 2, 5]
    print("更新索引 1 为 2 后:", [1, 2, 5])
    print("sumRange(0, 2):", numArray.sumRange(0, 2))  # 应该返回 8
```

==========

文件: Code19_GSS1.cpp

==========

```
// Can you answer these queries I (GSS1)
// 题目来源: SPOJ GSS1 - Can you answer these queries I
// 题目链接: https://www.spoj.com/problems/GSS1/
//
// 题目描述:
// 给定一个长度为 N 的整数序列 A，需要处理 M 个查询。
// 每个查询给定两个整数 X 和 Y，要求找出从第 X 个数到第 Y 个数之间的子段的最大和。
// 子段是指连续的一段数，空子段的和为 0。
//
// 解题思路:
// 1. 使用线段树维护区间最大子段和
// 2. 每个节点需要维护以下信息：
//    - lSum: 以左端点为起点的最大子段和
//    - rSum: 以右端点为终点的最大子段和
//    - sum: 区间和
//    - maxSum: 区间最大子段和
// 3. 合并左右子树时，父节点的信息由左右子树的信息计算得出
//
// 时间复杂度分析:
// - 构建线段树: O(n)
// - 查询: O(log n)
// 空间复杂度: O(n)

#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

struct SegmentTreeNode {
    int lSum;   // 以左端点为起点的最大子段和
    int rSum;   // 以右端点为终点的最大子段和
    int sum;    // 区间和
    int maxSum; // 区间最大子段和
```

```
    SegmentTreeNode() : lSum(0), rSum(0), sum(0), maxSum(0) {}

    SegmentTreeNode(int lSum, int rSum, int sum, int maxSum)
        : lSum(lSum), rSum(rSum), sum(sum), maxSum(maxSum) {}
};

class SegmentTree {
private:
    vector<SegmentTreeNode> tree;
    vector<int> data;
    int n;

    // 构建线段树
    void buildTree(int treeIndex, int l, int r) {
        if (l == r) {
            tree[treeIndex] = SegmentTreeNode(
                data[l],
                data[l],
                data[l],
                data[l]
            );
            return;
        }

        int mid = l + (r - l) / 2;
        int leftTreeIndex = 2 * treeIndex + 1;
        int rightTreeIndex = 2 * treeIndex + 2;

        // 构建左子树
        buildTree(leftTreeIndex, l, mid);
        // 构建右子树
        buildTree(rightTreeIndex, mid + 1, r);

        // 合并左右子树的信息
        tree[treeIndex] = merge(tree[leftTreeIndex], tree[rightTreeIndex]);
    }

    // 合并两个节点的信息
    SegmentTreeNode merge(SegmentTreeNode left, SegmentTreeNode right) {
        int sum = left.sum + right.sum;
        int lSum = max(left.lSum, left.sum + right.lSum);
        int rSum = max(right.rSum, right.sum + left.rSum);
        int maxSum = max(max(left.maxSum, right.maxSum), left.rSum + right.lSum);
```

```
        return SegmentTreeNode(lSum, rSum, sum, maxSum);
    }

    SegmentTreeNode queryTree(int treeIndex, int l, int r, int queryL, int queryR) {
        if (l == queryL && r == queryR) {
            return tree[treeIndex];
        }

        int mid = l + (r - l) / 2;
        int leftTreeIndex = 2 * treeIndex + 1;
        int rightTreeIndex = 2 * treeIndex + 2;

        if (queryR <= mid) {
            // 查询区间完全在左子树
            return queryTree(leftTreeIndex, l, mid, queryL, queryR);
        } else if (queryL > mid) {
            // 查询区间完全在右子树
            return queryTree(rightTreeIndex, mid + 1, r, queryL, queryR);
        } else {
            // 查询区间跨越左右子树
            SegmentTreeNode leftResult = queryTree(leftTreeIndex, l, mid, queryL, mid);
            SegmentTreeNode rightResult = queryTree(rightTreeIndex, mid + 1, r, mid + 1, queryR);
            return merge(leftResult, rightResult);
        }
    }

public:
    SegmentTree(vector<int>& nums) {
        n = nums.size();
        data = nums;
        tree.resize(4 * n);
        buildTree(0, 0, n - 1);
    }

    // 查询区间最大子段和
    int query(int queryL, int queryR) {
        if (n == 0) return 0;
        return queryTree(0, 0, n - 1, queryL, queryR).maxSum;
    }
};

// 测试代码
```

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

    // 读取序列长度
    int n;
    cin >> n;
    vector<int> nums(n);

    // 读取序列
    for (int i = 0; i < n; i++) {
        cin >> nums[i];
    }

    // 构建线段树
    SegmentTree segmentTree(nums);

    // 读取查询数量
    int m;
    cin >> m;

    // 处理查询
    for (int i = 0; i < m; i++) {
        int x, y;
        cin >> x >> y;
        // 转换为0索引
        x--;
        y--;
        cout << segmentTree.query(x, y) << "\n";
    }

    return 0;
}
```

==========

文件: Code19_GSS1.java

==========

```
// Can you answer these queries I (GSS1)
// 题目来源: SPOJ GSS1 - Can you answer these queries I
// 题目链接: https://www.spoj.com/problems/GSS1/
//
// 题目描述:
```

```
// 给定一个长度为 N 的整数序列 A，需要处理 M 个查询。
// 每个查询给定两个整数 X 和 Y，要求找出从第 X 个数到第 Y 个数之间的子段的最大和。
// 子段是指连续的一段数，空子段的和为 0。
//
// 解题思路:
// 1. 使用线段树维护区间最大子段和
// 2. 每个节点需要维护以下信息：
//    - lSum: 以左端点为起点的最大子段和
//    - rSum: 以右端点为终点的最大子段和
//    - sum: 区间和
//    - maxSum: 区间最大子段和
// 3. 合并左右子树时，父节点的信息由左右子树的信息计算得出
//
// 时间复杂度分析:
// - 构建线段树: O(n)
// - 查询: O(log n)
// 空间复杂度: O(n)

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

class SegmentTreeNode {
    int lSum;   // 以左端点为起点的最大子段和
    int rSum;   // 以右端点为终点的最大子段和
    int sum;    // 区间和
    int maxSum; // 区间最大子段和

    public SegmentTreeNode() {
        lSum = rSum = sum = maxSum = 0;
    }

    public SegmentTreeNode(int lSum, int rSum, int sum, int maxSum) {
        this.lSum = lSum;
        this.rSum = rSum;
        this.sum = sum;
        this.maxSum = maxSum;
    }
}

class SegmentTree {
    private SegmentTreeNode[] tree;
    private int[] data;
    private int n;
```

```
public SegmentTree(int[] nums) {
    n = nums.length;
    data = new int[n];
    for (int i = 0; i < n; i++) {
        data[i] = nums[i];
    }
    tree = new SegmentTreeNode[4 * n];
    for (int i = 0; i < 4 * n; i++) {
        tree[i] = new SegmentTreeNode();
    }
    buildTree(0, 0, n - 1);
}

// 构建线段树
private void buildTree(int treeIndex, int l, int r) {
    if (l == r) {
        tree[treeIndex] = new SegmentTreeNode(
            data[l],
            data[l],
            data[l],
            data[l]
        );
        return;
    }

    int mid = l + (r - l) / 2;
    int leftTreeIndex = 2 * treeIndex + 1;
    int rightTreeIndex = 2 * treeIndex + 2;

    // 构建左子树
    buildTree(leftTreeIndex, l, mid);
    // 构建右子树
    buildTree(rightTreeIndex, mid + 1, r);

    // 合并左右子树的信息
    tree[treeIndex] = merge(tree[leftTreeIndex], tree[rightTreeIndex]);
}

// 合并两个节点的信息
private SegmentTreeNode merge(SegmentTreeNode left, SegmentTreeNode right) {
    int sum = left.sum + right.sum;
    int lSum = Math.max(left.lSum, left.sum + right.lSum);
```

```
        int rSum = Math.max(right.rSum, right.sum + left.rSum);
        int maxSum = Math.max(Math.max(left.maxSum, right.maxSum), left.rSum + right.lSum);

        return new SegmentTreeNode(lSum, rSum, sum, maxSum);
    }

    // 查询区间最大子段和
    public int query(int queryL, int queryR) {
        if (n == 0) return 0;
        return queryTree(0, 0, n - 1, queryL, queryR).maxSum;
    }

    private SegmentTreeNode queryTree(int treeIndex, int l, int r, int queryL, int queryR) {
        if (l == queryL && r == queryR) {
            return tree[treeIndex];
        }

        int mid = l + (r - l) / 2;
        int leftTreeIndex = 2 * treeIndex + 1;
        int rightTreeIndex = 2 * treeIndex + 2;

        if (queryR <= mid) {
            // 查询区间完全在左子树
            return queryTree(leftTreeIndex, l, mid, queryL, queryR);
        } else if (queryL > mid) {
            // 查询区间完全在右子树
            return queryTree(rightTreeIndex, mid + 1, r, queryL, queryR);
        } else {
            // 查询区间跨越左右子树
            SegmentTreeNode leftResult = queryTree(leftTreeIndex, l, mid, queryL, mid);
            SegmentTreeNode rightResult = queryTree(rightTreeIndex, mid + 1, r, mid + 1, queryR);
            return merge(leftResult, rightResult);
        }
    }
}

public class Code19_GSS1 {
    public static void main(String[] args) throws IOException {
        BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));

        // 读取序列长度
        int n = Integer.parseInt(reader.readLine());
        int[] nums = new int[n];
```

```
        // 读取序列
        String[] parts = reader.readLine().split(" ");
        for (int i = 0; i < n; i++) {
            nums[i] = Integer.parseInt(parts[i]);
        }

        // 构建线段树
        SegmentTree segmentTree = new SegmentTree(nums);

        // 读取查询数量
        int m = Integer.parseInt(reader.readLine());

        // 处理查询
        for (int i = 0; i < m; i++) {
            parts = reader.readLine().split(" ");
            int x = Integer.parseInt(parts[0]) - 1; // 转换为0索引
            int y = Integer.parseInt(parts[1]) - 1; // 转换为0索引
            System.out.println(segmentTree.query(x, y));
        }
    }
}
```

==========

文件: Code19_GSS1.py

==========

```
# Can you answer these queries I (GSS1)
# 题目来源: SPOJ GSS1 - Can you answer these queries I
# 题目链接: https://www.spoj.com/problems/GSS1/
#
# 题目描述:
# 给定一个长度为N的整数序列A，需要处理M个查询。
# 每个查询给定两个整数X和Y，要求找出从第X个数到第Y个数之间的子段的最大和。
# 子段是指连续的一段数，空子段的和为0。
#
# 解题思路:
# 1. 使用线段树维护区间最大子段和
# 2. 每个节点需要维护以下信息：
#    - lSum: 以左端点为起点的最大子段和
#    - rSum: 以右端点为终点的最大子段和
#    - sum: 区间和
#    - maxSum: 区间最大子段和
```

```
# 3. 合并左右子树时，父节点的信息由左右子树的信息计算得出
#
# 时间复杂度分析:
# - 构建线段树: O(n)
# - 查询: O(log n)
# 空间复杂度: O(n)

class SegmentTreeNode:
    def __init__(self, lSum=0, rSum=0, sum=0, maxSum=0):
        self.lSum = lSum    # 以左端点为起点的最大子段和
        self.rSum = rSum    # 以右端点为终点的最大子段和
        self.sum = sum      # 区间和
        self.maxSum = maxSum # 区间最大子段和

class SegmentTree:
    def __init__(self, nums):
        self.n = len(nums)
        self.data = nums[:]
        self.tree = [SegmentTreeNode() for _ in range(4 * self.n)]
        self._build_tree(0, 0, self.n - 1)

    # 构建线段树
    def _build_tree(self, tree_index, l, r):
        if l == r:
            self.tree[tree_index] = SegmentTreeNode(
                self.data[l],
                self.data[l],
                self.data[l],
                self.data[l]
            )
            return

        mid = l + (r - l) // 2
        left_tree_index = 2 * tree_index + 1
        right_tree_index = 2 * tree_index + 2

        # 构建左子树
        self._build_tree(left_tree_index, l, mid)
        # 构建右子树
        self._build_tree(right_tree_index, mid + 1, r)

        # 合并左右子树的信息
        self.tree[tree_index] = self._merge(self.tree[left_tree_index],
```

```
self.tree[right_tree_index])

    # 合并两个节点的信息
    def _merge(self, left, right):
        sum_val = left.sum + right.sum
        lSum = max(left.lSum, left.sum + right.lSum)
        rSum = max(right.rSum, right.sum + left.rSum)
        maxSum = max(max(left.maxSum, right.maxSum), left.rSum + right.lSum)

        return SegmentTreeNode(lSum, rSum, sum_val, maxSum)

    # 查询区间最大子段和
    def query(self, query_l, query_r):
        if self.n == 0:
            return 0
        return self._query_tree(0, 0, self.n - 1, query_l, query_r).maxSum

    def _query_tree(self, tree_index, l, r, query_l, query_r):
        if l == query_l and r == query_r:
            return self.tree[tree_index]

        mid = l + (r - l) // 2
        left_tree_index = 2 * tree_index + 1
        right_tree_index = 2 * tree_index + 2

        if query_r <= mid:
            # 查询区间完全在左子树
            return self._query_tree(left_tree_index, l, mid, query_l, query_r)
        elif query_l > mid:
            # 查询区间完全在右子树
            return self._query_tree(right_tree_index, mid + 1, r, query_l, query_r)
        else:
            # 查询区间跨越左右子树
            left_result = self._query_tree(left_tree_index, l, mid, query_l, mid)
            right_result = self._query_tree(right_tree_index, mid + 1, r, mid + 1, query_r)
            return self._merge(left_result, right_result)

# 测试代码
if __name__ == "__main__":
    # 读取序列长度
    n = int(input())
    # 读取序列
    nums = list(map(int, input().split()))
```

```
    # 构建线段树
    segment_tree = SegmentTree(nums)

    # 读取查询数量
    m = int(input())

    # 处理查询
    for _ in range(m):
        x, y = map(int, input().split())
        # 转换为0索引
        x -= 1
        y -= 1
        print(segment_tree.query(x, y))
```

==========

文件: Code20_HDU1166.cpp

==========

```
// 敌兵布阵 (HDU 1166)
// 题目来源: HDU 1166. 敌兵布阵
// 题目链接: https://acm.hdu.edu.cn/showproblem.php?pid=1166
//
// 题目描述:
// C国在海岸线沿直线布置了N个工兵营地，Derek和Tidy的任务就是要监视这些工兵营地的活动情况。
// 每个工兵营地的人数C国都时刻掌握着。现在Tidy要向Derek汇报某一段连续的工兵营地一共有多少人，
// 例如Derek问："Tidy,马上汇报第3个营地到第10个营地共有多少人！"Tidy就要马上开始计算这一段的总人数并汇报。
// 但敌兵营地的人数经常变动，而Derek每次询问的段都不一样，所以Tidy要编写一个程序，支持以下两种操作：
// 1. Add i j: i和j为正整数，表示第i个营地增加j个人（j不超过30）
// 2. Query i j: i和j为正整数，表示询问第i个营地到第j个营地的总人数
//
// 解题思路:
// 1. 使用线段树实现单点更新和区间查询
// 2. 线段树的每个节点存储对应区间的元素和
// 3. 更新操作时，从根节点开始，找到对应的叶子节点并更新，然后逐层向上更新父节点
// 4. 查询操作时，从根节点开始，根据查询区间与当前节点区间的关系进行递归查询
//
// 时间复杂度分析:
// - 构建线段树: O(n)
// - 单点更新: O(log n)
// - 区间查询: O(log n)
```

```
// 空间复杂度: O(n)

#include <iostream>
#include <vector>
#include <string>
using namespace std;

class SegmentTree {
private:
    vector<int> tree;
    vector<int> data;
    int n;

    // 构建线段树
    void buildTree(int treeIndex, int l, int r) {
        if (l == r) {
            tree[treeIndex] = data[l];
            return;
        }

        int mid = l + (r - l) / 2;
        int leftTreeIndex = 2 * treeIndex + 1;
        int rightTreeIndex = 2 * treeIndex + 2;

        // 构建左子树
        buildTree(leftTreeIndex, l, mid);
        // 构建右子树
        buildTree(rightTreeIndex, mid + 1, r);

        // 当前节点的值等于左右子树值的和
        tree[treeIndex] = tree[leftTreeIndex] + tree[rightTreeIndex];
    }

    // 更新线段树
    void updateTree(int treeIndex, int l, int r, int index, int val) {
        if (l == r) {
            tree[treeIndex] = val;
            return;
        }

        int mid = l + (r - l) / 2;
        int leftTreeIndex = 2 * treeIndex + 1;
        int rightTreeIndex = 2 * treeIndex + 2;
```

```
        if (index >= l && index <= mid) {
            // 要更新的索引在左子树
            updateTree(leftTreeIndex, l, mid, index, val);
        } else {
            // 要更新的索引在右子树
            updateTree(rightTreeIndex, mid + 1, r, index, val);
        }

        // 更新当前节点的值
        tree[treeIndex] = tree[leftTreeIndex] + tree[rightTreeIndex];
    }

    // 查询区间和
    int queryTree(int treeIndex, int l, int r, int queryL, int queryR) {
        if (l == queryL && r == queryR) {
            return tree[treeIndex];
        }

        int mid = l + (r - l) / 2;
        int leftTreeIndex = 2 * treeIndex + 1;
        int rightTreeIndex = 2 * treeIndex + 2;

        if (queryR <= mid) {
            // 查询区间完全在左子树
            return queryTree(leftTreeIndex, l, mid, queryL, queryR);
        } else if (queryL > mid) {
            // 查询区间完全在右子树
            return queryTree(rightTreeIndex, mid + 1, r, queryL, queryR);
        } else {
            // 查询区间跨越左右子树
            int leftResult = queryTree(leftTreeIndex, l, mid, queryL, mid);
            int rightResult = queryTree(rightTreeIndex, mid + 1, r, mid + 1, queryR);
            return leftResult + rightResult;
        }
    }

public:
    SegmentTree(vector<int>& nums) {
        n = nums.size();
        data = nums;
        tree.resize(4 * n);
        buildTree(0, 0, n - 1);
```

```
    }

    // 单点更新
    void add(int index, int val) {
        data[index] += val;
        updateTree(0, 0, n - 1, index, data[index]);
    }

    // 查询区间和
    int query(int queryL, int queryR) {
        if (n == 0) return 0;
        return queryTree(0, 0, n - 1, queryL, queryR);
    }
};

// 测试代码
int main() {
    ios::sync_with_stdio(false);
    cin.tie(0);

    int T; // 测试用例数量
    cin >> T;

    for (int caseNum = 1; caseNum <= T; caseNum++) {
        cout << "Case " << caseNum << ":\n";

        int n; // 营地数量
        cin >> n;
        vector<int> nums(n);

        // 读取每个营地的初始人数
        for (int i = 0; i < n; i++) {
            cin >> nums[i];
        }

        // 构建线段树
        SegmentTree segmentTree(nums);

        // 处理操作
        string operation;
        while (cin >> operation && operation != "End") {
            if (operation == "Add") {
                int i, j;
```

```
                cin >> i >> j;
                segmentTree.add(i - 1, j); // 转换为0索引
            } else if (operation == "Sub") {
                int i, j;
                cin >> i >> j;
                segmentTree.add(i - 1, -j); // 转换为0索引
            } else if (operation == "Query") {
                int i, j;
                cin >> i >> j;
                cout << segmentTree.query(i - 1, j - 1) << "\n"; // 转换为0索引
            }
        }
    }

    return 0;
}
```

==========

文件: Code20_HDU1166.java

==========

```
// 敌兵布阵 (HDU 1166)
// 题目来源: HDU 1166. 敌兵布阵
// 题目链接: https://acm.hdu.edu.cn/showproblem.php?pid=1166
//
// 题目描述:
// C国在海岸线沿直线布置了N个工兵营地，Derek和Tidy的任务就是要监视这些工兵营地的活动情况。
// 每个工兵营地的人数C国都时刻掌握着。现在Tidy要向Derek汇报某一段连续的工兵营地一共有多少人，
// 例如Derek问："Tidy,马上汇报第3个营地到第10个营地共有多少人！"Tidy就要马上开始计算这一段的总人数并汇报。
// 但敌兵营地的人数经常变动，而Derek每次询问的段都不一样，所以Tidy要编写一个程序，支持以下两种操作：
// 1. Add i j: i和j为正整数，表示第i个营地增加j个人（j不超过30）
// 2. Query i j: i和j为正整数，表示询问第i个营地到第j个营地的总人数
//
// 解题思路:
// 1. 使用线段树实现单点更新和区间查询
// 2. 线段树的每个节点存储对应区间的元素和
// 3. 更新操作时，从根节点开始，找到对应的叶子节点并更新，然后逐层向上更新父节点
// 4. 查询操作时，从根节点开始，根据查询区间与当前节点区间的关系进行递归查询
//
// 时间复杂度分析:
// - 构建线段树: O(n)
```

```
// - 单点更新: O(log n)
// - 区间查询: O(log n)
// 空间复杂度: O(n)

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

class SegmentTree {
    private int[] tree;
    private int[] data;
    private int n;

    public SegmentTree(int[] nums) {
        n = nums.length;
        data = new int[n];
        for (int i = 0; i < n; i++) {
            data[i] = nums[i];
        }
        tree = new int[4 * n];
        buildTree(0, 0, n - 1);
    }

    // 构建线段树
    private void buildTree(int treeIndex, int l, int r) {
        if (l == r) {
            tree[treeIndex] = data[l];
            return;
        }

        int mid = l + (r - l) / 2;
        int leftTreeIndex = 2 * treeIndex + 1;
        int rightTreeIndex = 2 * treeIndex + 2;

        // 构建左子树
        buildTree(leftTreeIndex, l, mid);
        // 构建右子树
        buildTree(rightTreeIndex, mid + 1, r);

        // 当前节点的值等于左右子树值的和
        tree[treeIndex] = tree[leftTreeIndex] + tree[rightTreeIndex];
    }

    // 单点更新
```

```
public void add(int index, int val) {
    data[index] += val;
    updateTree(0, 0, n - 1, index, data[index]);
}

// 更新线段树
private void updateTree(int treeIndex, int l, int r, int index, int val) {
    if (l == r) {
        tree[treeIndex] = val;
        return;
    }

    int mid = l + (r - l) / 2;
    int leftTreeIndex = 2 * treeIndex + 1;
    int rightTreeIndex = 2 * treeIndex + 2;

    if (index >= l && index <= mid) {
        // 要更新的索引在左子树
        updateTree(leftTreeIndex, l, mid, index, val);
    } else {
        // 要更新的索引在右子树
        updateTree(rightTreeIndex, mid + 1, r, index, val);
    }

    // 更新当前节点的值
    tree[treeIndex] = tree[leftTreeIndex] + tree[rightTreeIndex];
}

// 查询区间和
public int query(int queryL, int queryR) {
    if (n == 0) return 0;
    return queryTree(0, 0, n - 1, queryL, queryR);
}

private int queryTree(int treeIndex, int l, int r, int queryL, int queryR) {
    if (l == queryL && r == queryR) {
        return tree[treeIndex];
    }

    int mid = l + (r - l) / 2;
    int leftTreeIndex = 2 * treeIndex + 1;
    int rightTreeIndex = 2 * treeIndex + 2;
```

```
        if (queryR <= mid) {
            // 查询区间完全在左子树
            return queryTree(leftTreeIndex, l, mid, queryL, queryR);
        } else if (queryL > mid) {
            // 查询区间完全在右子树
            return queryTree(rightTreeIndex, mid + 1, r, queryL, queryR);
        } else {
            // 查询区间跨越左右子树
            int leftResult = queryTree(leftTreeIndex, l, mid, queryL, mid);
            int rightResult = queryTree(rightTreeIndex, mid + 1, r, mid + 1, queryR);
            return leftResult + rightResult;
        }
    }
}

public class Code20_HDU1166 {
    public static void main(String[] args) throws IOException {
        BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));
        BufferedWriter writer = new BufferedWriter(new OutputStreamWriter(System.out));

        int T = Integer.parseInt(reader.readLine()); // 测试用例数量

        for (int caseNum = 1; caseNum <= T; caseNum++) {
            writer.write("Case " + caseNum + ":\n");

            int n = Integer.parseInt(reader.readLine()); // 营地数量
            int[] nums = new int[n];

            // 读取每个营地的初始人数
            String[] parts = reader.readLine().split(" ");
            for (int i = 0; i < n; i++) {
                nums[i] = Integer.parseInt(parts[i]);
            }

            // 构建线段树
            SegmentTree segmentTree = new SegmentTree(nums);

            // 处理操作
            String line;
            while (!(line = reader.readLine()).equals("End")) {
                parts = line.split(" ");
                String operation = parts[0];
```

```
                if (operation.equals("Add")) {
                    int i = Integer.parseInt(parts[1]) - 1; // 转换为0索引
                    int j = Integer.parseInt(parts[2]);
                    segmentTree.add(i, j);
                } else if (operation.equals("Sub")) {
                    int i = Integer.parseInt(parts[1]) - 1; // 转换为0索引
                    int j = Integer.parseInt(parts[2]);
                    segmentTree.add(i, -j);
                } else if (operation.equals("Query")) {
                    int i = Integer.parseInt(parts[1]) - 1; // 转换为0索引
                    int j = Integer.parseInt(parts[2]) - 1; // 转换为0索引
                    writer.write(segmentTree.query(i, j) + "\n");
                }
            }
        }

        writer.flush();
    }
}
```

==========

文件: Code20_HDU1166.py

==========

```
# 敌兵布阵 (HDU 1166)
# 题目来源: HDU 1166. 敌兵布阵
# 题目链接: https://acm.hdu.edu.cn/showproblem.php?pid=1166
#
# 题目描述:
# C国在海岸线沿直线布置了N个工兵营地，Derek和Tidy的任务就是要监视这些工兵营地的活动情况。
# 每个工兵营地的人数C国都时刻掌握着。现在Tidy要向Derek汇报某一段连续的工兵营地一共有多少人，
# 例如Derek问："Tidy,马上汇报第3个营地到第10个营地共有多少人！"Tidy就要马上开始计算这一段的总人数并汇报。
# 但敌兵营地的人数经常变动，而Derek每次询问的段都不一样，所以Tidy要编写一个程序，支持以下两种操作：
# 1. Add i j: i和j为正整数，表示第i个营地增加j个人（j不超过30）
# 2. Query i j: i和j为正整数，表示询问第i个营地到第j个营地的总人数
#
# 解题思路:
# 1. 使用线段树实现单点更新和区间查询
# 2. 线段树的每个节点存储对应区间的元素和
# 3. 更新操作时，从根节点开始，找到对应的叶子节点并更新，然后逐层向上更新父节点
# 4. 查询操作时，从根节点开始，根据查询区间与当前节点区间的关系进行递归查询
```

```
#
# 时间复杂度分析:
# - 构建线段树: O(n)
# - 单点更新: O(log n)
# - 区间查询: O(log n)
# 空间复杂度: O(n)

class SegmentTree:
    def __init__(self, nums):
        self.n = len(nums)
        self.data = nums[:]
        self.tree = [0] * (4 * self.n)
        self._build_tree(0, 0, self.n - 1)

    # 构建线段树
    def _build_tree(self, tree_index, l, r):
        if l == r:
            self.tree[tree_index] = self.data[l]
            return

        mid = l + (r - l) // 2
        left_tree_index = 2 * tree_index + 1
        right_tree_index = 2 * tree_index + 2

        # 构建左子树
        self._build_tree(left_tree_index, l, mid)
        # 构建右子树
        self._build_tree(right_tree_index, mid + 1, r)

        # 当前节点的值等于左右子树值的和
        self.tree[tree_index] = self.tree[left_tree_index] + self.tree[right_tree_index]

    # 单点更新
    def add(self, index, val):
        self.data[index] += val
        self._update_tree(0, 0, self.n - 1, index, self.data[index])

    # 更新线段树
    def _update_tree(self, tree_index, l, r, index, val):
        if l == r:
            self.tree[tree_index] = val
            return
```

```
        mid = l + (r - l) // 2
        left_tree_index = 2 * tree_index + 1
        right_tree_index = 2 * tree_index + 2

        if index >= l and index <= mid:
            # 要更新的索引在左子树
            self._update_tree(left_tree_index, l, mid, index, val)
        else:
            # 要更新的索引在右子树
            self._update_tree(right_tree_index, mid + 1, r, index, val)

        # 更新当前节点的值
        self.tree[tree_index] = self.tree[left_tree_index] + self.tree[right_tree_index]

    # 查询区间和
    def query(self, query_l, query_r):
        if self.n == 0:
            return 0
        return self._query_tree(0, 0, self.n - 1, query_l, query_r)

    def _query_tree(self, tree_index, l, r, query_l, query_r):
        if l == query_l and r == query_r:
            return self.tree[tree_index]

        mid = l + (r - l) // 2
        left_tree_index = 2 * tree_index + 1
        right_tree_index = 2 * tree_index + 2

        if query_r <= mid:
            # 查询区间完全在左子树
            return self._query_tree(left_tree_index, l, mid, query_l, query_r)
        elif query_l > mid:
            # 查询区间完全在右子树
            return self._query_tree(right_tree_index, mid + 1, r, query_l, query_r)
        else:
            # 查询区间跨越左右子树
            left_result = self._query_tree(left_tree_index, l, mid, query_l, mid)
            right_result = self._query_tree(right_tree_index, mid + 1, r, mid + 1, query_r)
            return left_result + right_result

# 测试代码
if __name__ == "__main__":
    import sys
```

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

idx = 0
T = int(data[idx])  # 测试用例数量
idx += 1

for case_num in range(1, T + 1):
    print(f"Case {case_num}:")

    n = int(data[idx])  # 营地数量
    idx += 1

    # 读取每个营地的初始人数
    nums = [int(data[idx + i]) for i in range(n)]
    idx += n

    # 构建线段树
    segment_tree = SegmentTree(nums)

    # 处理操作
    while True:
        operation = data[idx]
        idx += 1

        if operation == "End":
            break
        elif operation == "Add":
            i = int(data[idx]) - 1  # 转换为0索引
            j = int(data[idx + 1])
            idx += 2
            segment_tree.add(i, j)
        elif operation == "Sub":
            i = int(data[idx]) - 1  # 转换为0索引
            j = int(data[idx + 1])
            idx += 2
            segment_tree.add(i, -j)
        elif operation == "Query":
            i = int(data[idx]) - 1  # 转换为0索引
            j = int(data[idx + 1]) - 1  # 转换为0索引
            idx += 2
            print(segment_tree.query(i, j))
```

==========

文件: Code21_POJ3468.cpp
==========

```
// A Simple Problem with Integers (POJ 3468)
// 题目来源: POJ 3468. A Simple Problem with Integers
// 题目链接: http://poj.org/problem?id=3468
//
// 题目描述:
// 你有 N 个整数 A1, A2, ..., AN。你需要处理两种类型的操作:
// 1. C a b c: 将区间[a, b]中的每个数都加上 c
// 2. Q a b: 查询区间[a, b]中所有数的和
//
// 解题思路:
// 1. 使用带懒惰传播的线段树实现区间更新和区间查询
// 2. 懒惰传播用于延迟更新，避免不必要的计算
// 3. 区间更新时，只在必要时才将更新操作传递给子节点
// 4. 查询时确保所有相关的懒惰标记都被处理
//
// 时间复杂度分析:
// - 区间更新: O(log n)
// - 区间查询: O(log n)
// 空间复杂度: O(n)

#include <iostream>
#include <vector>
using namespace std;

class SegmentTree {
private:
    vector<long long> tree;
    vector<long long> lazy;
    vector<int> data;
    int n;

    // 构建线段树
    void buildTree(int treeIndex, int l, int r) {
        if (l == r) {
            tree[treeIndex] = data[l];
            return;
        }

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

```
    int leftTreeIndex = 2 * treeIndex + 1;
    int rightTreeIndex = 2 * treeIndex + 2;

    // 构建左子树
    buildTree(leftTreeIndex, l, mid);
    // 构建右子树
    buildTree(rightTreeIndex, mid + 1, r);

    // 当前节点的值等于左右子树值的和
    tree[treeIndex] = tree[leftTreeIndex] + tree[rightTreeIndex];
}

// 下推懒惰标记
void pushDown(int treeIndex, int l, int r) {
    if (lazy[treeIndex] != 0) {
        // 将懒惰标记应用到当前节点
        tree[treeIndex] += lazy[treeIndex] * (r - l + 1);

        // 如果不是叶子节点，将懒惰标记传递给子节点
        if (l != r) {
            lazy[2 * treeIndex + 1] += lazy[treeIndex];
            lazy[2 * treeIndex + 2] += lazy[treeIndex];
        }

        // 清除当前节点的懒惰标记
        lazy[treeIndex] = 0;
    }
}

// 区间加法更新辅助函数
void updateTree(int treeIndex, int l, int r, int queryL, int queryR, long long val) {
    // 1. 先处理懒惰标记
    pushDown(treeIndex, l, r);

    // 2. 当前区间与更新区间无交集
    if (l > queryR || r < queryL) {
        return;
    }

    // 3. 当前区间完全包含在更新区间内
    if (l >= queryL && r <= queryR) {
        // 更新当前节点的值
        tree[treeIndex] += val * (r - l + 1);
```

```
            // 如果不是叶子节点，设置懒惰标记
            if (l != r) {
                lazy[2 * treeIndex + 1] += val;
                lazy[2 * treeIndex + 2] += val;
            }
            return;
        }

        // 4. 当前区间与更新区间有部分交集，递归处理左右子树
        int mid = l + (r - l) / 2;
        updateTree(2 * treeIndex + 1, l, mid, queryL, queryR, val);
        updateTree(2 * treeIndex + 2, mid + 1, r, queryL, queryR, val);

        // 更新当前节点的值
        tree[treeIndex] = tree[2 * treeIndex + 1] + tree[2 * treeIndex + 2];
    }

    // 查询区间和辅助函数
    long long queryTree(int treeIndex, int l, int r, int queryL, int queryR) {
        // 1. 先处理懒惰标记
        pushDown(treeIndex, l, r);

        // 2. 当前区间与查询区间无交集
        if (l > queryR || r < queryL) {
            return 0;
        }

        // 3. 当前区间完全包含在查询区间内
        if (l >= queryL && r <= queryR) {
            return tree[treeIndex];
        }

        // 4. 当前区间与查询区间有部分交集，递归查询左右子树
        int mid = l + (r - l) / 2;
        long long leftSum = queryTree(2 * treeIndex + 1, l, mid, queryL, queryR);
        long long rightSum = queryTree(2 * treeIndex + 2, mid + 1, r, queryL, queryR);
        return leftSum + rightSum;
    }

public:
    SegmentTree(vector<int>& nums) {
        n = nums.size();
        data = nums;
```

```
        tree.resize(4 * n);
        lazy.resize(4 * n);
        buildTree(0, 0, n - 1);
    }

    // 区间加法更新 [queryL, queryR] 区间内每个元素加上 val
    void update(int queryL, int queryR, long long val) {
        updateTree(0, 0, n - 1, queryL, queryR, val);
    }

    // 查询区间和
    long long query(int queryL, int queryR) {
        return queryTree(0, 0, n - 1, queryL, queryR);
    }
};

// 测试代码
int main() {
    ios::sync_with_stdio(false);
    cin.tie(0);

    int n, q; // 数组长度和操作数量
    cin >> n >> q;

    vector<int> nums(n);
    for (int i = 0; i < n; i++) {
        cin >> nums[i];
    }

    // 构建线段树
    SegmentTree segmentTree(nums);

    // 处理操作
    for (int i = 0; i < q; i++) {
        char operation;
        cin >> operation;

        if (operation == 'C') {
            int a, b;
            long long c;
            cin >> a >> b >> c;
            segmentTree.update(a - 1, b - 1, c); // 转换为0索引
        } else if (operation == 'Q') {
```

```
            int a, b;
            cin >> a >> b;
            cout << segmentTree.query(a - 1, b - 1) << "\n"; // 转换为0索引
        }
    }

    return 0;
}
```

==========

文件: Code21_POJ3468.java

==========

```
// A Simple Problem with Integers (POJ 3468)
// 题目来源: POJ 3468. A Simple Problem with Integers
// 题目链接: http://poj.org/problem?id=3468
//
// 题目描述:
// 你有N个整数A1, A2, ..., AN。你需要处理两种类型的操作:
// 1. C a b c: 将区间[a, b]中的每个数都加上c
// 2. Q a b: 查询区间[a, b]中所有数的和
//
// 解题思路:
// 1. 使用带懒惰传播的线段树实现区间更新和区间查询
// 2. 懒惰传播用于延迟更新，避免不必要的计算
// 3. 区间更新时，只在必要时才将更新操作传递给子节点
// 4. 查询时确保所有相关的懒惰标记都被处理
//
// 时间复杂度分析:
// - 区间更新: O(log n)
// - 区间查询: O(log n)
// 空间复杂度: O(n)

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

class SegmentTree {
    private long[] tree;
    private long[] lazy;
    private int[] data;
    private int n;

    public SegmentTree(int[] nums) {
```

```
        n = nums.length;
        data = new int[n];
        for (int i = 0; i < n; i++) {
            data[i] = nums[i];
        }
        tree = new long[4 * n];
        lazy = new long[4 * n];
        buildTree(0, 0, n - 1);
    }

    // 构建线段树
    private void buildTree(int treeIndex, int l, int r) {
        if (l == r) {
            tree[treeIndex] = data[l];
            return;
        }

        int mid = l + (r - l) / 2;
        int leftTreeIndex = 2 * treeIndex + 1;
        int rightTreeIndex = 2 * treeIndex + 2;

        // 构建左子树
        buildTree(leftTreeIndex, l, mid);
        // 构建右子树
        buildTree(rightTreeIndex, mid + 1, r);

        // 当前节点的值等于左右子树值的和
        tree[treeIndex] = tree[leftTreeIndex] + tree[rightTreeIndex];
    }

    // 下推懒惰标记
    private void pushDown(int treeIndex, int l, int r) {
        if (lazy[treeIndex] != 0) {
            // 将懒惰标记应用到当前节点
            tree[treeIndex] += lazy[treeIndex] * (r - l + 1);

            // 如果不是叶子节点，将懒惰标记传递给子节点
            if (l != r) {
                lazy[2 * treeIndex + 1] += lazy[treeIndex];
                lazy[2 * treeIndex + 2] += lazy[treeIndex];
            }

            // 清除当前节点的懒惰标记
```

```
            lazy[treeIndex] = 0;
        }
    }

    // 区间加法更新 [queryL, queryR] 区间内每个元素加上 val
    public void update(int queryL, int queryR, int val) {
        updateTree(0, 0, n - 1, queryL, queryR, val);
    }

    // 区间加法更新辅助函数
    private void updateTree(int treeIndex, int l, int r, int queryL, int queryR, int val) {
        // 1. 先处理懒惰标记
        pushDown(treeIndex, l, r);

        // 2. 当前区间与更新区间无交集
        if (l > queryR || r < queryL) {
            return;
        }

        // 3. 当前区间完全包含在更新区间内
        if (l >= queryL && r <= queryR) {
            // 更新当前节点的值
            tree[treeIndex] += (long) val * (r - l + 1);
            // 如果不是叶子节点，设置懒惰标记
            if (l != r) {
                lazy[2 * treeIndex + 1] += val;
                lazy[2 * treeIndex + 2] += val;
            }
            return;
        }

        // 4. 当前区间与更新区间有部分交集，递归处理左右子树
        int mid = l + (r - l) / 2;
        updateTree(2 * treeIndex + 1, l, mid, queryL, queryR, val);
        updateTree(2 * treeIndex + 2, mid + 1, r, queryL, queryR, val);

        // 更新当前节点的值
        tree[treeIndex] = tree[2 * treeIndex + 1] + tree[2 * treeIndex + 2];
    }

    // 查询区间和
    public long query(int queryL, int queryR) {
        return queryTree(0, 0, n - 1, queryL, queryR);
```

```
    }

    // 查询区间和辅助函数
    private long queryTree(int treeIndex, int l, int r, int queryL, int queryR) {
        // 1. 先处理懒惰标记
        pushDown(treeIndex, l, r);

        // 2. 当前区间与查询区间无交集
        if (l > queryR || r < queryL) {
            return 0;
        }

        // 3. 当前区间完全包含在查询区间内
        if (l >= queryL && r <= queryR) {
            return tree[treeIndex];
        }

        // 4. 当前区间与查询区间有部分交集，递归查询左右子树
        int mid = l + (r - l) / 2;
        long leftSum = queryTree(2 * treeIndex + 1, l, mid, queryL, queryR);
        long rightSum = queryTree(2 * treeIndex + 2, mid + 1, r, queryL, queryR);
        return leftSum + rightSum;
    }
}

public class Code21_POJ3468 {
    public static void main(String[] args) throws IOException {
        BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));
        BufferedWriter writer = new BufferedWriter(new OutputStreamWriter(System.out));

        String[] parts = reader.readLine().split(" ");
        int n = Integer.parseInt(parts[0]); // 数组长度
        int q = Integer.parseInt(parts[1]); // 操作数量

        int[] nums = new int[n];
        parts = reader.readLine().split(" ");
        for (int i = 0; i < n; i++) {
            nums[i] = Integer.parseInt(parts[i]);
        }

        // 构建线段树
        SegmentTree segmentTree = new SegmentTree(nums);
```

```
        // 处理操作
        for (int i = 0; i < q; i++) {
            parts = reader.readLine().split(" ");
            String operation = parts[0];

            if (operation.equals("C")) {
                int a = Integer.parseInt(parts[1]) - 1; // 转换为0索引
                int b = Integer.parseInt(parts[2]) - 1; // 转换为0索引
                int c = Integer.parseInt(parts[3]);
                segmentTree.update(a, b, c);
            } else if (operation.equals("Q")) {
                int a = Integer.parseInt(parts[1]) - 1; // 转换为0索引
                int b = Integer.parseInt(parts[2]) - 1; // 转换为0索引
                writer.write(segmentTree.query(a, b) + "\n");
            }
        }

        writer.flush();
    }
}
```

==========

文件: Code21_POJ3468.py

==========

```
# A Simple Problem with Integers (POJ 3468)
# 题目来源: POJ 3468. A Simple Problem with Integers
# 题目链接: http://poj.org/problem?id=3468
#
# 题目描述:
# 你有N个整数A1, A2, ..., AN。你需要处理两种类型的操作:
# 1. C a b c: 将区间[a, b]中的每个数都加上c
# 2. Q a b: 查询区间[a, b]中所有数的和
#
# 解题思路:
# 1. 使用带懒惰传播的线段树实现区间更新和区间查询
# 2. 懒惰传播用于延迟更新，避免不必要的计算
# 3. 区间更新时，只在必要时才将更新操作传递给子节点
# 4. 查询时确保所有相关的懒惰标记都被处理
#
# 时间复杂度分析:
# - 区间更新: O(log n)
# - 区间查询: O(log n)
```

```
# 空间复杂度: O(n)

class SegmentTree:
    def __init__(self, nums):
        self.n = len(nums)
        self.data = nums[:]
        self.tree = [0] * (4 * self.n)
        self.lazy = [0] * (4 * self.n)
        self._build_tree(0, 0, self.n - 1)

    # 构建线段树
    def _build_tree(self, tree_index, l, r):
        if l == r:
            self.tree[tree_index] = self.data[l]
            return

        mid = l + (r - l) // 2
        left_tree_index = 2 * tree_index + 1
        right_tree_index = 2 * tree_index + 2

        # 构建左子树
        self._build_tree(left_tree_index, l, mid)
        # 构建右子树
        self._build_tree(right_tree_index, mid + 1, r)

        # 当前节点的值等于左右子树值的和
        self.tree[tree_index] = self.tree[left_tree_index] + self.tree[right_tree_index]

    # 下推懒惰标记
    def _push_down(self, tree_index, l, r):
        if self.lazy[tree_index] != 0:
            # 将懒惰标记应用到当前节点
            self.tree[tree_index] += self.lazy[tree_index] * (r - l + 1)

            # 如果不是叶子节点，将懒惰标记传递给子节点
            if l != r:
                self.lazy[2 * tree_index + 1] += self.lazy[tree_index]
                self.lazy[2 * tree_index + 2] += self.lazy[tree_index]

            # 清除当前节点的懒惰标记
            self.lazy[tree_index] = 0

    # 区间加法更新 [query_l, query_r] 区间内每个元素加上 val
```

```
def update(self, query_l, query_r, val):
    self._update_tree(0, 0, self.n - 1, query_l, query_r, val)

# 区间加法更新辅助函数
def _update_tree(self, tree_index, l, r, query_l, query_r, val):
    # 1. 先处理懒惰标记
    self._push_down(tree_index, l, r)

    # 2. 当前区间与更新区间无交集
    if l > query_r or r < query_l:
        return

    # 3. 当前区间完全包含在更新区间内
    if l >= query_l and r <= query_r:
        # 更新当前节点的值
        self.tree[tree_index] += val * (r - l + 1)
        # 如果不是叶子节点，设置懒惰标记
        if l != r:
            self.lazy[2 * tree_index + 1] += val
            self.lazy[2 * tree_index + 2] += val
        return

    # 4. 当前区间与更新区间有部分交集，递归处理左右子树
    mid = l + (r - l) // 2
    self._update_tree(2 * tree_index + 1, l, mid, query_l, query_r, val)
    self._update_tree(2 * tree_index + 2, mid + 1, r, query_l, query_r, val)

    # 更新当前节点的值
    self.tree[tree_index] = self.tree[2 * tree_index + 1] + self.tree[2 * tree_index + 2]

# 查询区间和
def query(self, query_l, query_r):
    return self._query_tree(0, 0, self.n - 1, query_l, query_r)

# 查询区间和辅助函数
def _query_tree(self, tree_index, l, r, query_l, query_r):
    # 1. 先处理懒惰标记
    self._push_down(tree_index, l, r)

    # 2. 当前区间与查询区间无交集
    if l > query_r or r < query_l:
        return 0
```

```
        # 3. 当前区间完全包含在查询区间内
        if l >= query_l and r <= query_r:
            return self.tree[tree_index]

        # 4. 当前区间与查询区间有部分交集，递归查询左右子树
        mid = l + (r - l) // 2
        left_sum = self._query_tree(2 * tree_index + 1, l, mid, query_l, query_r)
        right_sum = self._query_tree(2 * tree_index + 2, mid + 1, r, query_l, query_r)
        return left_sum + right_sum

# 测试代码
if __name__ == "__main__":
    import sys
    input = sys.stdin.read
    data = input().split()

    idx = 0
    n = int(data[idx])  # 数组长度
    idx += 1
    q = int(data[idx])  # 操作数量
    idx += 1

    nums = [int(data[idx + i]) for i in range(n)]
    idx += n

    # 构建线段树
    segment_tree = SegmentTree(nums)

    # 处理操作
    for _ in range(q):
        operation = data[idx]
        idx += 1

        if operation == "C":
            a = int(data[idx]) - 1  # 转换为0索引
            b = int(data[idx + 1]) - 1  # 转换为0索引
            c = int(data[idx + 2])
            idx += 3
            segment_tree.update(a, b, c)
        elif operation == "Q":
            a = int(data[idx]) - 1  # 转换为0索引
            b = int(data[idx + 1]) - 1  # 转换为0索引
            idx += 2
```

```
            print(segment_tree.query(a, b))

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

文件: Code22_LuoguP3372.cpp
==========================================
// 【模板】线段树 1 (Luogu P3372)
// 题目来源: Luogu P3372 【模板】线段树 1
// 题目链接: https://www.luogu.com.cn/problem/P3372
//
// 题目描述:
// 如题，已知一个数列，你需要进行下面两种操作：
// 1. 将某区间每一个数加上 x
// 2. 求出某区间每一个数的和
//
// 解题思路:
// 1. 使用带懒惰传播的线段树实现区间更新和区间查询
// 2. 懒惰传播用于延迟更新，避免不必要的计算
// 3. 区间更新时，只在必要时才将更新操作传递给子节点
// 4. 查询时确保所有相关的懒惰标记都被处理
//
// 时间复杂度分析:
// - 区间更新: O(log n)
// - 区间查询: O(log n)
// 空间复杂度: O(n)

#include <iostream>
#include <vector>
using namespace std;

class SegmentTree {
private:
    vector<long long> tree;
    vector<long long> lazy;
    vector<int> data;
    int n;

    // 构建线段树
    void buildTree(int treeIndex, int l, int r) {
        if (l == r) {
            tree[treeIndex] = data[l];
            return;
        }
```

```
    int mid = l + (r - l) / 2;
    int leftTreeIndex = 2 * treeIndex + 1;
    int rightTreeIndex = 2 * treeIndex + 2;

    // 构建左子树
    buildTree(leftTreeIndex, l, mid);
    // 构建右子树
    buildTree(rightTreeIndex, mid + 1, r);

    // 当前节点的值等于左右子树值的和
    tree[treeIndex] = tree[leftTreeIndex] + tree[rightTreeIndex];
}

// 下推懒惰标记
void pushDown(int treeIndex, int l, int r) {
    if (lazy[treeIndex] != 0) {
        // 将懒惰标记应用到当前节点
        tree[treeIndex] += lazy[treeIndex] * (r - l + 1);

        // 如果不是叶子节点，将懒惰标记传递给子节点
        if (l != r) {
            lazy[2 * treeIndex + 1] += lazy[treeIndex];
            lazy[2 * treeIndex + 2] += lazy[treeIndex];
        }

        // 清除当前节点的懒惰标记
        lazy[treeIndex] = 0;
    }
}

// 区间加法更新辅助函数
void updateTree(int treeIndex, int l, int r, int queryL, int queryR, long long val) {
    // 1. 先处理懒惰标记
    pushDown(treeIndex, l, r);

    // 2. 当前区间与更新区间无交集
    if (l > queryR || r < queryL) {
        return;
    }

    // 3. 当前区间完全包含在更新区间内
    if (l >= queryL && r <= queryR) {
```

```
            // 更新当前节点的值
            tree[treeIndex] += val * (r - l + 1);
            // 如果不是叶子节点，设置懒惰标记
            if (l != r) {
                lazy[2 * treeIndex + 1] += val;
                lazy[2 * treeIndex + 2] += val;
            }
            return;
        }

        // 4. 当前区间与更新区间有部分交集，递归处理左右子树
        int mid = l + (r - l) / 2;
        updateTree(2 * treeIndex + 1, l, mid, queryL, queryR, val);
        updateTree(2 * treeIndex + 2, mid + 1, r, queryL, queryR, val);

        // 更新当前节点的值
        tree[treeIndex] = tree[2 * treeIndex + 1] + tree[2 * treeIndex + 2];
    }

    // 查询区间和辅助函数
    long long queryTree(int treeIndex, int l, int r, int queryL, int queryR) {
        // 1. 先处理懒惰标记
        pushDown(treeIndex, l, r);

        // 2. 当前区间与查询区间无交集
        if (l > queryR || r < queryL) {
            return 0;
        }

        // 3. 当前区间完全包含在查询区间内
        if (l >= queryL && r <= queryR) {
            return tree[treeIndex];
        }

        // 4. 当前区间与查询区间有部分交集，递归查询左右子树
        int mid = l + (r - l) / 2;
        long long leftSum = queryTree(2 * treeIndex + 1, l, mid, queryL, queryR);
        long long rightSum = queryTree(2 * treeIndex + 2, mid + 1, r, queryL, queryR);
        return leftSum + rightSum;
    }

public:
    SegmentTree(vector<int>& nums) {
```

```
        n = nums.size();
        data = nums;
        tree.resize(4 * n);
        lazy.resize(4 * n);
        buildTree(0, 0, n - 1);
    }

    // 区间加法更新 [queryL, queryR] 区间内每个元素加上 val
    void update(int queryL, int queryR, long long val) {
        updateTree(0, 0, n - 1, queryL, queryR, val);
    }

    // 查询区间和
    long long query(int queryL, int queryR) {
        return queryTree(0, 0, n - 1, queryL, queryR);
    }
};

// 测试代码
int main() {
    ios::sync_with_stdio(false);
    cin.tie(0);

    int n, m; // 数列长度和操作数量
    cin >> n >> m;

    vector<int> nums(n);
    for (int i = 0; i < n; i++) {
        cin >> nums[i];
    }

    // 构建线段树
    SegmentTree segmentTree(nums);

    // 处理操作
    for (int i = 0; i < m; i++) {
        int operation;
        cin >> operation;

        if (operation == 1) {
            int x, y;
            long long k;
            cin >> x >> y >> k;
```

```
            segmentTree.update(x - 1, y - 1, k); // 转换为0索引
        } else if (operation == 2) {
            int x, y;
            cin >> x >> y;
            cout << segmentTree.query(x - 1, y - 1) << "\n"; // 转换为0索引
        }
    }

    return 0;
}

==========

文件: Code22_LuoguP3372.java
==========
// 【模板】线段树 1 (Luogu P3372)
// 题目来源: Luogu P3372 【模板】线段树 1
// 题目链接: https://www.luogu.com.cn/problem/P3372
//
// 题目描述:
// 如题，已知一个数列，你需要进行下面两种操作：
// 1. 将某区间每一个数加上 x
// 2. 求出某区间每一个数的和
//
// 解题思路:
// 1. 使用带懒惰传播的线段树实现区间更新和区间查询
// 2. 懒惰传播用于延迟更新，避免不必要的计算
// 3. 区间更新时，只在必要时才将更新操作传递给子节点
// 4. 查询时确保所有相关的懒惰标记都被处理
//
// 时间复杂度分析:
// - 区间更新: O(log n)
// - 区间查询: O(log n)
// 空间复杂度: O(n)

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

class SegmentTree {
    private long[] tree;
    private long[] lazy;
    private int[] data;
    private int n;
```

```
public SegmentTree(int[] nums) {
    n = nums.length;
    data = new int[n];
    for (int i = 0; i < n; i++) {
        data[i] = nums[i];
    }
    tree = new long[4 * n];
    lazy = new long[4 * n];
    buildTree(0, 0, n - 1);
}

// 构建线段树
private void buildTree(int treeIndex, int l, int r) {
    if (l == r) {
        tree[treeIndex] = data[l];
        return;
    }

    int mid = l + (r - l) / 2;
    int leftTreeIndex = 2 * treeIndex + 1;
    int rightTreeIndex = 2 * treeIndex + 2;

    // 构建左子树
    buildTree(leftTreeIndex, l, mid);
    // 构建右子树
    buildTree(rightTreeIndex, mid + 1, r);

    // 当前节点的值等于左右子树值的和
    tree[treeIndex] = tree[leftTreeIndex] + tree[rightTreeIndex];
}

// 下推懒惰标记
private void pushDown(int treeIndex, int l, int r) {
    if (lazy[treeIndex] != 0) {
        // 将懒惰标记应用到当前节点
        tree[treeIndex] += lazy[treeIndex] * (r - l + 1);

        // 如果不是叶子节点，将懒惰标记传递给子节点
        if (l != r) {
            lazy[2 * treeIndex + 1] += lazy[treeIndex];
            lazy[2 * treeIndex + 2] += lazy[treeIndex];
        }
```

```
        // 清除当前节点的懒惰标记
        lazy[treeIndex] = 0;
    }
}

// 区间加法更新 [queryL, queryR] 区间内每个元素加上 val
public void update(int queryL, int queryR, int val) {
    updateTree(0, 0, n - 1, queryL, queryR, val);
}

// 区间加法更新辅助函数
private void updateTree(int treeIndex, int l, int r, int queryL, int queryR, int val) {
    // 1. 先处理懒惰标记
    pushDown(treeIndex, l, r);

    // 2. 当前区间与更新区间无交集
    if (l > queryR || r < queryL) {
        return;
    }

    // 3. 当前区间完全包含在更新区间内
    if (l >= queryL && r <= queryR) {
        // 更新当前节点的值
        tree[treeIndex] += (long) val * (r - l + 1);
        // 如果不是叶子节点，设置懒惰标记
        if (l != r) {
            lazy[2 * treeIndex + 1] += val;
            lazy[2 * treeIndex + 2] += val;
        }
        return;
    }

    // 4. 当前区间与更新区间有部分交集，递归处理左右子树
    int mid = l + (r - l) / 2;
    updateTree(2 * treeIndex + 1, l, mid, queryL, queryR, val);
    updateTree(2 * treeIndex + 2, mid + 1, r, queryL, queryR, val);

    // 更新当前节点的值
    tree[treeIndex] = tree[2 * treeIndex + 1] + tree[2 * treeIndex + 2];
}

// 查询区间和
```

```
    public long query(int queryL, int queryR) {
        return queryTree(0, 0, n - 1, queryL, queryR);
    }

    // 查询区间和辅助函数
    private long queryTree(int treeIndex, int l, int r, int queryL, int queryR) {
        // 1. 先处理懒惰标记
        pushDown(treeIndex, l, r);

        // 2. 当前区间与查询区间无交集
        if (l > queryR || r < queryL) {
            return 0;
        }

        // 3. 当前区间完全包含在查询区间内
        if (l >= queryL && r <= queryR) {
            return tree[treeIndex];
        }

        // 4. 当前区间与查询区间有部分交集，递归查询左右子树
        int mid = l + (r - l) / 2;
        long leftSum = queryTree(2 * treeIndex + 1, l, mid, queryL, queryR);
        long rightSum = queryTree(2 * treeIndex + 2, mid + 1, r, queryL, queryR);
        return leftSum + rightSum;
    }
}

public class Code22_LuoguP3372 {
    public static void main(String[] args) throws IOException {
        BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));
        BufferedWriter writer = new BufferedWriter(new OutputStreamWriter(System.out));

        String[] parts = reader.readLine().split(" ");
        int n = Integer.parseInt(parts[0]); // 数列长度
        int m = Integer.parseInt(parts[1]); // 操作数量

        int[] nums = new int[n];
        parts = reader.readLine().split(" ");
        for (int i = 0; i < n; i++) {
            nums[i] = Integer.parseInt(parts[i]);
        }

        // 构建线段树
```

```
        SegmentTree segmentTree = new SegmentTree(nums);

        // 处理操作
        for (int i = 0; i < m; i++) {
            parts = reader.readLine().split(" ");
            int operation = Integer.parseInt(parts[0]);

            if (operation == 1) {
                int x = Integer.parseInt(parts[1]) - 1; // 转换为0索引
                int y = Integer.parseInt(parts[2]) - 1; // 转换为0索引
                int k = Integer.parseInt(parts[3]);
                segmentTree.update(x, y, k);
            } else if (operation == 2) {
                int x = Integer.parseInt(parts[1]) - 1; // 转换为0索引
                int y = Integer.parseInt(parts[2]) - 1; // 转换为0索引
                writer.write(segmentTree.query(x, y) + "\n");
            }
        }

        writer.flush();
    }
}
```

==========

文件: Code22_LuoguP3372.py

==========

```
# 【模板】线段树 1 (Luogu P3372)
# 题目来源: Luogu P3372 【模板】线段树 1
# 题目链接: https://www.luogu.com.cn/problem/P3372
#
# 题目描述:
# 如题，已知一个数列，你需要进行下面两种操作：
# 1. 将某区间每一个数加上 x
# 2. 求出某区间每一个数的和
#
# 解题思路:
# 1. 使用带懒惰传播的线段树实现区间更新和区间查询
# 2. 懒惰传播用于延迟更新，避免不必要的计算
# 3. 区间更新时，只在必要时才将更新操作传递给子节点
# 4. 查询时确保所有相关的懒惰标记都被处理
#
# 时间复杂度分析:
```

```
# - 区间更新: O(log n)
# - 区间查询: O(log n)
# 空间复杂度: O(n)

class SegmentTree:
    def __init__(self, nums):
        self.n = len(nums)
        self.data = nums[:]
        self.tree = [0] * (4 * self.n)
        self.lazy = [0] * (4 * self.n)
        self._build_tree(0, 0, self.n - 1)

    # 构建线段树
    def _build_tree(self, tree_index, l, r):
        if l == r:
            self.tree[tree_index] = self.data[l]
            return

        mid = l + (r - l) // 2
        left_tree_index = 2 * tree_index + 1
        right_tree_index = 2 * tree_index + 2

        # 构建左子树
        self._build_tree(left_tree_index, l, mid)
        # 构建右子树
        self._build_tree(right_tree_index, mid + 1, r)

        # 当前节点的值等于左右子树值的和
        self.tree[tree_index] = self.tree[left_tree_index] + self.tree[right_tree_index]

    # 下推懒惰标记
    def _push_down(self, tree_index, l, r):
        if self.lazy[tree_index] != 0:
            # 将懒惰标记应用到当前节点
            self.tree[tree_index] += self.lazy[tree_index] * (r - l + 1)

            # 如果不是叶子节点，将懒惰标记传递给子节点
            if l != r:
                self.lazy[2 * tree_index + 1] += self.lazy[tree_index]
                self.lazy[2 * tree_index + 2] += self.lazy[tree_index]

            # 清除当前节点的懒惰标记
            self.lazy[tree_index] = 0
```

```
# 区间加法更新 [query_l, query_r] 区间内每个元素加上 val
def update(self, query_l, query_r, val):
    self._update_tree(0, 0, self.n - 1, query_l, query_r, val)

# 区间加法更新辅助函数
def _update_tree(self, tree_index, l, r, query_l, query_r, val):
    # 1. 先处理懒惰标记
    self._push_down(tree_index, l, r)

    # 2. 当前区间与更新区间无交集
    if l > query_r or r < query_l:
        return

    # 3. 当前区间完全包含在更新区间内
    if l >= query_l and r <= query_r:
        # 更新当前节点的值
        self.tree[tree_index] += val * (r - l + 1)
        # 如果不是叶子节点，设置懒惰标记
        if l != r:
            self.lazy[2 * tree_index + 1] += val
            self.lazy[2 * tree_index + 2] += val
        return

    # 4. 当前区间与更新区间有部分交集，递归处理左右子树
    mid = l + (r - l) // 2
    self._update_tree(2 * tree_index + 1, l, mid, query_l, query_r, val)
    self._update_tree(2 * tree_index + 2, mid + 1, r, query_l, query_r, val)

    # 更新当前节点的值
    self.tree[tree_index] = self.tree[2 * tree_index + 1] + self.tree[2 * tree_index + 2]

# 查询区间和
def query(self, query_l, query_r):
    return self._query_tree(0, 0, self.n - 1, query_l, query_r)

# 查询区间和辅助函数
def _query_tree(self, tree_index, l, r, query_l, query_r):
    # 1. 先处理懒惰标记
    self._push_down(tree_index, l, r)

    # 2. 当前区间与查询区间无交集
    if l > query_r or r < query_l:
```

```
            return 0

        # 3. 当前区间完全包含在查询区间内
        if l >= query_l and r <= query_r:
            return self.tree[tree_index]

        # 4. 当前区间与查询区间有部分交集，递归查询左右子树
        mid = l + (r - l) // 2
        left_sum = self._query_tree(2 * tree_index + 1, l, mid, query_l, query_r)
        right_sum = self._query_tree(2 * tree_index + 2, mid + 1, r, query_l, query_r)
        return left_sum + right_sum

# 测试代码
if __name__ == "__main__":
    import sys
    input = sys.stdin.read
    data = input().split()

    idx = 0
    n = int(data[idx])  # 数列长度
    idx += 1
    m = int(data[idx])  # 操作数量
    idx += 1

    nums = [int(data[idx + i]) for i in range(n)]
    idx += n

    # 构建线段树
    segment_tree = SegmentTree(nums)

    # 处理操作
    for _ in range(m):
        operation = int(data[idx])
        idx += 1

        if operation == 1:
            x = int(data[idx]) - 1  # 转换为0索引
            y = int(data[idx + 1]) - 1  # 转换为0索引
            k = int(data[idx + 2])
            idx += 3
            segment_tree.update(x, y, k)
        elif operation == 2:
            x = int(data[idx]) - 1  # 转换为0索引
```

```
            y = int(data[idx + 1]) - 1  # 转换为0索引
            idx += 2
            print(segment_tree.query(x, y))

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