==========
文件夹: class085_SegmentTreeAndRelatedAlgorithms
==========

[Markdown 文件]
==========
文件: ADDITIONAL_SEGMENT_TREE_PROBLEMS.md
==========
线段树补充题目大全 - Additional Segment Tree Problems

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。本文件收集了来自各大算法平台的线段树相关题目，帮助学习者全面掌握线段树的应用。

LeetCode 题目

1. LeetCode 307. Range Sum Query - Mutable (区域和检索 - 数组可修改)
- **类型**: 单点更新 + 区间求和
- **难度**: Medium
- **题目链接**: https://leetcode.com/problems/range-sum-query-mutable/
- **核心思想**: 经典线段树应用，支持单点更新和区间求和查询

2. LeetCode 308. Range Sum Query 2D - Mutable (二维区域和检索 - 可变)
- **类型**: 二维线段树 + 区间求和
- **难度**: Hard
- **题目链接**: https://leetcode.com/problems/range-sum-query-2d-mutable/
- **核心思想**: 二维线段树或树状数组维护二维区间和

3. 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
- **题目链接**: 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，结合离散化处理

4. LeetCode 327. Count of Range Sum (区间和的个数)
- **类型**: 前缀和 + 离散化 + 区间查询
- **难度**: Hard
- **题目链接**: https://leetcode.com/problems/count-of-range-sum/
- **核心思想**: 前缀和转换 + 线段树维护区间信息

5. LeetCode 493. Reverse Pairs (翻转对)

- **类型**: 离散化 + 单点更新 + 区间求和
- **难度**: Hard
- **题目链接**: https://leetcode.com/problems/reverse-pairs/
- **核心思想**: 计算满足条件的逆序对，使用线段树优化

6. LeetCode 699. Falling Squares (掉落的方块)
- **类型**: 区间最值查询 + 离散化
- **难度**: Hard
- **题目链接**: 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

7. LeetCode 218. The Skyline Problem (天际线问题)
- **类型**: 扫描线 + 线段树
- **难度**: Hard
- **题目链接**: https://leetcode.com/problems/the-skyline-problem/
- **核心思想**: 扫描线算法 + 线段树维护区间最大值

8. LeetCode 715. Range Module (Range 模块)
- **类型**: 区间合并 + 线段树
- **难度**: Hard
- **题目链接**: https://leetcode.com/problems/range-module/
- **核心思想**: 维护区间覆盖状态，支持区间添加、查询、删除

9. 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
- **题目链接**: https://leetcode.com/problems/my-calendar-iii/
- **核心思想**: 维护区间最大值，计算最大重叠次数

10. LeetCode 850. Rectangle Area II (矩形面积 II)
- **类型**: 扫描线 + 线段树
- **难度**: Hard
- **题目链接**: 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 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
- **题目链接**: https://leetcode.com/problems/online-majority-element-in-subarray/
- **核心思想**: 线段树维护区间众数信息

12. 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
- **题目链接**: 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

13. LeetCode 1649. Create Sorted Array through Instructions (通过指令创建有序数组)
- **类型**: 离散化 + 单点更新 + 区间求和
- **难度**: Hard
- **题目链接**: https://leetcode.com/problems/create-sorted-array-through-instructions/
- **核心思想**: 使用线段树维护插入代价

Codeforces 题目

1. Codeforces 339D. Xenia and Bit Operations (Xenia 和位运算)
- **类型**: 位运算 + 线段树
- **难度**: Medium
- **题目链接**: 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 (Pashmak 和 Parmida 的问题)
- **类型**: 离散化 + 线段树
- **难度**: Hard
- **题目链接**: 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 (循环 RMQ)
- **类型**: 循环数组 + 线段树
- **难度**: Hard
- **题目链接**: 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 (Valera 和查询)
- **类型**: 离线处理 + 线段树
- **难度**: Hard
- **题目链接**: https://codeforces.com/contest/369/problem/E
- **核心思想**: 离线处理 + 线段树维护

5. Codeforces 121E. Lucky Array (幸运数组)
- **类型**: 区间更新 + 区间查询
- **难度**: Hard
- **题目链接**: https://codeforces.com/contest/121/problem/E
- **核心思想**: 带懒惰传播的线段树

6. Codeforces 292E. Copying Data (复制数据)
- **类型**: 区间覆盖 + 线段树
- **难度**: Hard
- **题目链接**: https://codeforces.com/contest/292/problem/E
- **核心思想**: 线段树维护区间覆盖操作

SPOJ 题目

1. SPOJ GSS1 - Can you answer these queries I (你能回答这些问题吗 I)
- **类型**: 区间最大子段和 + 线段树
- **难度**: Hard
- **题目链接**: https://www.spoj.com/problems/GSS1/
- **核心思想**: 线段树维护区间最大子段和信息

2. SPOJ GSS3 - Can you answer these queries III (你能回答这些问题吗 III)
- **类型**: 区间最大子段和 + 单点更新
- **难度**: Hard
- **题目链接**: 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 (你能回答这些问题吗 IV)
- **类型**: 区间开方 + 线段树
- **难度**: Hard
- **题目链接**: 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 GSS5 - Can you answer these queries V (你能回答这些问题吗 V)
- **类型**: 区间最大子段和 + 复杂查询
- **难度**: Hard
- **题目链接**: https://www.spoj.com/problems/GSS5/
- **核心思想**: 复杂区间查询的线段树实现

5. SPOJ GSS6 - Can you answer these queries VI (你能回答这些问题吗 VI)
- **类型**: 区间插入删除 + 线段树
- **难度**: Hard
- **题目链接**: https://www.spoj.com/problems/GSS6/
- **核心思想**: 支持动态插入删除的线段树

6. SPOJ GSS7 - Can you answer these queries VII (你能回答这些问题吗 VII)
- **类型**: 树链剖分 + 线段树
- **难度**: Hard
- **题目链接**: https://www.spoj.com/problems/GSS7/

- **核心思想**: 树链剖分 + 线段树维护路径信息

7. SPOJ HORRIBLE - Horrible Queries (可怕的查询)
- **类型**: 区间更新 + 区间求和 + 懒惰传播
- **难度**: Hard
- **题目链接**: https://www.spoj.com/problems/HORRIBLE/
- **核心思想**: 带懒惰传播的线段树

8. SPOJ BRCKTS - Brackets (括号)
- **类型**: 括号匹配 + 线段树
- **难度**: Medium
- **题目链接**: https://www.spoj.com/problems/BRCKTS/
- **核心思想**: 线段树维护括号匹配信息

9. SPOJ FREQUENT - Frequent values (频繁出现的值)
- **类型**: 区间众数 + 线段树
- **难度**: Hard
- **题目链接**: https://www.spoj.com/problems/FREQUENT/
- **核心思想**: 线段树维护区间众数信息

10. SPOJ KGSS - Maximum Sum (最大和)
- **类型**: 区间最大值 + 线段树
- **难度**: Medium
- **题目链接**: https://www.spoj.com/problems/KGSS/
- **核心思想**: 维护区间最大值和次大值

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
- **题目链接**: https://www.luogu.com.cn/problem/P3372
- **核心思想**: 线段树模板题

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
- **题目链接**: 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

- **题目链接**: https://www.luogu.com.cn/problem/P4198
- **核心思想**: 线段树维护区间斜率信息

4. P1558 色板游戏
- **类型**: 区间颜色更新 + 位运算
- **难度**: Hard
- **题目链接**: https://www.luogu.com.cn/problem/P1558
- **核心思想**: 位运算表示颜色状态 + 线段树维护

5. P2184 贪婪大陆
- **类型**: 区间更新 + 区间查询
- **难度**: Hard
- **题目链接**: https://www.luogu.com.cn/problem/P2184
- **核心思想**: 使用两个线段树分别维护区间起始点和终止点

6. P3870 开关
- **类型**: 区间翻转 + 区间求和
- **难度**: Medium
- **题目链接**: https://www.luogu.com.cn/problem/P3870
- **核心思想**: 懒惰标记实现区间翻转操作

7. P1438 无聊的数列
- **类型**: 差分数组 + 线段树
- **难度**: Hard
- **题目链接**: https://www.luogu.com.cn/problem/P1438
- **核心思想**: 差分数组 + 线段树维护区间更新

8. P1471 方差
- **类型**: 区间更新 + 区间查询
- **难度**: Hard
- **题目链接**: https://www.luogu.com.cn/problem/P1471
- **核心思想**: 维护区间和与区间平方和计算方差

HDU 题目

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

2. HDU 1754. I Hate It
- **类型**: 单点更新 + 区间最值

- **难度**: Medium
- **题目链接**: 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
- **题目链接**: https://acm.hdu.edu.cn/showproblem.php?pid=1698
- **核心思想**: 带懒惰传播的线段树

4. HDU 4521. 小明系列故事——吃蜜糖
- **类型**: 区间更新 + 区间查询
- **难度**: Hard
- **题目链接**: https://acm.hdu.edu.cn/showproblem.php?pid=4521
- **核心思想**: 线段树维护区间信息

POJ 题目

1. POJ 3468. A Simple Problem with Integers (简单的整数问题)
- **类型**: 区间更新 + 区间求和 + 懒惰传播
- **难度**: Hard
- **题目链接**: http://poj.org/problem?id=3468
- **核心思想**: 经典的带懒惰传播线段树

2. POJ 2528. Mayor's posters (市长的海报)
- **类型**: 离散化 + 区间覆盖
- **难度**: Hard
- **题目链接**: 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 (平衡的阵容)
- **类型**: 区间最值查询
- **难度**: Medium
- **题目链接**: http://poj.org/problem?id=3264
- **核心思想**: RMQ 问题的线段树解法

4. POJ 3667. Hotel (旅馆)
- **类型**: 区间分配 + 线段树
- **难度**: Hard
- **题目链接**: http://poj.org/problem?id=3667
- **核心思想**: 线段树维护区间连续空闲长度

AtCoder 题目

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

2. AtCoder ABC185 F - Range Xor Query
- **类型**: 区间异或查询
- **难度**: Medium
- **题目链接**: https://atcoder.jp/contests/abc185/tasks/abc185_f
- **核心思想**: 线段树维护区间异或值

LintCode 题目

1. LintCode 206. Interval Sum (区间求和)
- **类型**: 区间求和
- **难度**: Easy
- **题目链接**: 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 (统计前面比自己小的数的个数)
- **类型**: 离散化 + 单点更新 + 区间求和
- **难度**: Medium
- **题目链接**: https://www.lintcode.com/problem/249/
- **核心思想**: 线段树维护值域信息

3. LintCode 439. Segment Tree Build II (线段树构造 II)
- **类型**: 线段树构造
- **难度**: Medium
- **题目链接**: https://www.lintcode.com/problem/439/
- **核心思想**: 线段树的构建过程

4. LintCode 247. Segment Tree Query II (线段树查询 II)
- **类型**: 线段树查询
- **难度**: Medium
- **题目链接**: https://www.lintcode.com/problem/247/
- **核心思想**: 线段树区间查询操作

HackerRank 题目

1. Range Minimum Query (区间最小值查询)
- **类型**: 区间最小值查询

- **难度**: Easy
- **题目链接**: https://www.hackerrank.com/challenges/range-minimum-query/problem
- **核心思想**: 线段树维护区间最小值

2. Array Manipulation (数组操作)
- **类型**: 差分数组 + 线段树
- **难度**: Hard
- **题目链接**: https://www.hackerrank.com/challenges/crush/problem
- **核心思想**: 差分思想优化区间更新

USACO 题目

1. USACO 2017 January Gold - Balanced Photo (平衡的照片)
- **类型**: 区间查询 + 线段树
- **难度**: Hard
- **题目链接**: http://www.usaco.org/index.php?page=viewproblem2&cpid=693
- **核心思想**: 线段树优化计数查询

其他平台题目

1. Timus 1846. GCD 2010
- **类型**: 区间 GCD 查询 + 线段树
- **难度**: Hard
- **题目链接**: https://acm.timus.ru/problem.aspx?space=1&num=1846
- **核心思想**: 线段树维护区间 GCD

2. UVa 12086. Potentiometers (电位器)
- **类型**: 单点更新 + 区间求和
- **难度**: Medium
- **题目链接**:
https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&page=show_problem&problem=3238
- **核心思想**: 经典线段树应用

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

6. 扫描线算法
- 处理几何问题中的区间覆盖
- 应用：矩形面积计算、天际线问题等

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

4. 树链剖分 + 线段树
- 处理树上路径查询问题
- 应用：树上区间操作、路径最值等

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
==========
线段树 (Segment Tree)

线段树是一种非常重要的数据结构，主要用于处理区间查询和区间更新操作。它能够将查询和更新操作的时间复杂度都降低到O(log n)。

本目录中的题目

1. 开关问题 (Code01_Switch.java)
- 题目来源：洛谷 P3870
- 操作类型：
  - 操作 0 l r：改变 l~r 范围上所有灯的状态（开变关，关变开）
  - 操作 1 l r：查询 l~r 范围上有多少灯是打开的
- 算法要点：使用线段树维护区间和，通过懒惰标记实现区间翻转

2. 贪婪大陆 (Code02_Bombs.java)

- 题目来源：洛谷 P2184
- 操作类型：
  - 操作 1 l r：在 l~r 范围的格子上放置一种新型地雷
  - 操作 2 l r：查询 l~r 范围的格子上一共放置过多少款不同的地雷
- 算法要点：使用两个线段树分别维护地雷范围的起始点和终止点

3. 无聊的数列 (Code03_BoringSequence.java)
- 题目来源：洛谷 P1438
- 操作类型：
  - 操作 1 l r k d：arr[l..r]范围上的数依次加上等差数列，首项 k，公差 d
  - 操作 2 p：查询 arr[p]的值
- 算法要点：使用差分数组+线段树维护区间更新和单点查询

4. 平均数和方差 (Code04_MeanVariance1.java, Code04_MeanVariance2.java)
- 题目来源：洛谷 P1471
- 操作类型：
  - 操作 1 l r：arr 数组中[l, r]范围上每个数字加上 k
  - 操作 2 l r：查询 arr 数组中[l, r]范围上所有数字的平均数
  - 操作 3 l r：查询 arr 数组中[l, r]范围上所有数字的方差
- 算法要点：维护区间和与区间平方和，通过数学公式计算平均数和方差

5. 色板游戏 (ExtraQuestion.java)
- 题目来源：洛谷 P1558
- 操作类型：
  - 操作 C A B C：A~B 范围的色板都涂上 C 颜色
  - 操作 P A B：查询 A~B 范围的色板一共有几种颜色
- 算法要点：使用位运算表示颜色状态，通过线段树维护区间颜色种类

补充题目

6. 区域和检索 - 数组可修改 (Code05_RangeSumQueryMutable.java, Code05_RangeSumQueryMutable.py)
- 题目来源：LeetCode 307
- 操作类型：
  - update(index, val)：将 nums[index]的值更新为 val
  - sumRange(left, right)：返回数组 nums 中索引 left 和索引 right 之间（包含）的 nums 元素的和
- 算法要点：标准线段树实现，支持区间求和和单点更新

7. 天际线问题 (Code06_TheSkylineProblem.java, Code06_TheSkylineProblem.py)
- 题目来源：LeetCode 218
- 问题描述：根据建筑物的位置和高度计算城市的天际线
- 算法要点：使用离散化+线段树维护区间最大值，处理坐标压缩

8. 掉落的方块 (Code07_FallingSquares.java, Code07_FallingSquares.py)

- 题目来源：LeetCode 699
- 问题描述：计算方块掉落后的堆叠最高高度
- 算法要点：使用线段树维护区间覆盖操作，处理区间最大值查询

9. 二维区域和检索 - 可变 (Code08_RangeSumQuery2D.java, Code08_RangeSumQuery2D.py)
- 题目来源：LeetCode 308
- 操作类型：
  - update(row, col, val)：更新矩阵中某个单元格的值
  - sumRegion(row1, col1, row2, col2)：计算子矩阵的总和
- 算法要点：二维线段树实现，支持区间求和和单点更新

10. 计算右侧小于当前元素的个数 (Code09_CountOfSmallerNumbersAfterSelf.java, Code09_CountOfSmallerNumbersAfterSelf.py)
- 题目来源：LeetCode 315
- 问题描述：计算数组中每个元素右侧小于该元素的元素数量
- 算法要点：动态开点线段树 + 离散化处理

11. 翻转对 (Code10_ReversePairs.java, Code10_ReversePairs.py)
- 题目来源：LeetCode 493
- 问题描述：计算满足 i < j 且 nums[i] > 2*nums[j] 的重要翻转对数量
- 算法要点：线段树维护值域信息 + 离散化处理

12. 范围模块 (Code12_RangeModule.java, Code12_RangeModule.cpp, Code12_RangeModule.py)
- 题目来源：LeetCode 715
- 问题描述：设计一个数据结构来跟踪区间范围，支持添加、删除和查询操作
- 算法要点：动态开点线段树 + 区间合并操作

13. 区间和查询 - 可变 (Code13_RangeMinimumQuery.java)
- 题目来源：LeetCode 307（扩展版本）
- 问题描述：支持区间和查询和单点更新的线段树实现
- 算法要点：标准线段树实现，支持高效区间操作

14. 区间最小值查询 (Code14_RangeMinimumQuery.cpp, Code14_RangeMinimumQuery.py)
- 题目来源：经典线段树问题
- 问题描述：支持区间最小值查询的线段树实现
- 算法要点：线段树维护区间最小值信息

15. 区间最大值查询 (Code15_RangeMaximumQuery.java)
- 题目来源：经典线段树问题
- 问题描述：支持区间最大值查询的线段树实现
- 算法要点：线段树维护区间最大值信息

16. 懒惰传播线段树 (Code16_LazyPropagation.java)

- 题目来源：经典线段树问题
- 问题描述：支持区间更新和区间查询的懒惰传播线段树
- 算法要点：懒惰标记技术实现高效区间更新

17. 线段树应用合集 (Code17_SegmentTreeApplications.java)
- 题目来源：多种线段树应用场景
- 问题描述：包含多种线段树应用的综合实现
- 算法要点：覆盖线段树的多种应用场景

18. 动态线段树 (Code18_DynamicSegmentTree.java)
- 题目来源：经典线段树问题
- 问题描述：支持动态开点的线段树实现
- 算法要点：动态开点技术处理大值域问题

19. 线段树懒惰传播模板 (Code19_SegmentTreeWithLazy.java, Code19_SegmentTreeWithLazy.cpp, Code19_SegmentTreeWithLazy.py)
- 题目来源：洛谷 P3372 【模板】线段树 1
- 问题描述：线段树懒惰传播的标准模板实现
- 算法要点：完整的懒惰传播机制，支持区间加法和区间求和

20. 线段树区间最大值查询 (Code20_SegmentTreeMaxQuery.java)
- 题目来源：洛谷 P3865 【模板】ST 表
- 问题描述：支持区间最大值查询的线段树实现
- 算法要点：线段树维护区间最大值，支持动态更新

21. 线段树 GCD 查询 (Code21_SegmentTreeGCD.java, Code21_SegmentTreeGCD.cpp, Code21_SegmentTreeGCD.py)
- 题目来源：Codeforces 914D - Bash and a Tough Math Puzzle
- 问题描述：支持区间 GCD 查询和单点更新的线段树实现
- 算法要点：线段树维护区间 GCD 信息，支持高效区间查询和单点更新

22. 线段树区间赋值 (Code22_SegmentTreeAssignment.java)
- 题目来源：Codeforces 438D - The Child and Sequence
- 问题描述：支持区间赋值、区间求和、区间取模等操作的线段树实现
- 算法要点：线段树维护区间信息，支持多种区间操作

线段树分类体系

基础线段树
- 区间求和线段树 (Code05, Code13)
- 区间最值线段树 (Code14, Code15, Code20)
- 单点更新线段树 (Code05, Code13)

高级线段树
- 懒惰传播线段树 (Code16, Code19)
- 动态开点线段树 (Code09, Code10, Code18)
- 二维线段树 (Code08)

数学运算线段树
- GCD 线段树 (Code21)
- 区间赋值线段树 (Code22)
- 数学类线段树 (Code03, Code04)

应用场景线段树
- 统计类线段树 (Code01, Code02, Code09, Code10)
- 几何类线段树 (Code06, Code07)
- 综合应用线段树 (Code17)

各大算法平台线段树题目来源

LeetCode
- 307. 区域和检索 - 数组可修改 (Code05)
- 493. 翻转对 (Code10)
- 715. Range Module (Code12)
- 218. The Skyline Problem (Code06)
- 699. Falling Squares (Code07)

Codeforces
- 914D. Bash and a Tough Math Puzzle (Code21)
- 438D. The Child and Sequence (Code22)
- 多种线段树模板题和应用题

洛谷 (Luogu)
- P3870 开关问题 (Code01)
- P2184 贪婪大陆 (Code02)
- P1438 无聊的数列 (Code03)
- P1471 平均数和方差 (Code04)
- P3372 线段树 1 (Code19)
- P3865 ST 表 (Code20)

其他平台
- HackerRank：多种线段树练习题
- AtCoder：线段树竞赛题目
- USACO：线段树算法训练题
- 各大高校 OJ：线段树经典题目

线段树核心思想

线段树是一种基于分治思想的二叉树结构，每个节点代表一个区间。它支持两种基本操作：

1. **区间更新**：对某个区间内的所有元素进行统一的操作
2. **区间查询**：查询某个区间内的统计信息（如和、最大值、最小值等）

通过引入懒惰标记（Lazy Propagation），线段树可以高效地处理区间更新操作，避免了对区间内每个元素逐一更新的开销。

时间复杂度分析

- 建树：O(n)
- 单点更新：O(log n)
- 区间更新：O(log n)
- 单点查询：O(log n)
- 区间查询：O(log n)

空间复杂度分析

线段树的空间复杂度为O(n)，通常需要4n的空间来保证足够的节点数量。

应用场景

线段树适用于以下场景：
1. 需要频繁进行区间查询和更新操作
2. 查询操作涉及区间统计信息（如和、最值等）
3. 数据规模较大，需要高效处理

线段树与其他数据结构的比较

数据结构	单点更新	区间更新	单点查询	区间查询
普通数组	O(1)	O(n)	O(1)	O(n)
前缀和数组	O(n)	O(n)	O(1)	O(1)
线段树	O(log n)	O(log n)	O(log n)	O(log n)
树状数组	O(log n)	O(log n)	O(log n)	O(log n)

线段树相比其他数据结构，在区间更新和区间查询方面具有更好的平衡性，特别适合需要频繁进行这两种操作的场景。

工程化考量

1. 异常处理
- 输入参数验证：确保区间边界合法
- 数组越界检查：防止访问无效索引
- 空数组处理：特殊情况的边界处理

2. 性能优化
- 内存预分配：避免频繁内存分配
- 常数优化：减少不必要的计算
- 缓存友好：优化内存访问模式

3. 可配置性
- 支持多种数据类型
- 可配置的合并函数
- 灵活的懒惰标记策略

调试技巧

1. 打印中间过程
```java
// 在关键节点打印调试信息
System.out.println("区间[" + start + "," + end + "]: " + tree[idx]);
```

2. 边界条件测试
- 空数组测试
- 单元素数组测试
- 全区间操作测试
- 重叠区间测试

3. 性能分析
- 大规模数据测试
- 时间复杂度验证
- 内存使用监控

三语言实现对比

Java 实现特点
- 面向对象设计
- 丰富的标准库支持
- 良好的异常处理机制

C++实现特点
- 高性能执行效率

- 模板编程支持
- 内存管理控制

Python 实现特点
- 简洁的语法
- 动态类型系统
- 丰富的第三方库

学习建议

1. **基础掌握**：先理解线段树的基本原理和实现
2. **模板练习**：熟练掌握标准线段树模板
3. **应用拓展**：学习各种线段树变体和应用场景
4. **工程实践**：在实际项目中应用线段树解决问题

参考资料

1. 《算法导论》- 线段树章节
2. LeetCode 线段树专题
3. Codeforces 线段树题目集
4. 各大 OJ 平台的线段树训练题

本目录包含了线段树算法的全面实现，涵盖了基础到高级的各种应用场景，适合算法学习和面试准备。

==========

文件: SUMMARY.md
==========
class112 线段树专题总结

项目概述

本项目对 class112 中的线段树专题进行了全面的扩展和完善，包括：

1. 创建了详细的补充题目清单
2. 为新增题目实现了 Java、Python 版本的完整代码
3. 对于 C++版本，根据项目环境限制创建了简化实现
4. 所有代码都经过编译和测试验证，确保正确性

完成的题目

原有题目（8个）
1. 开关问题 (Code01_Switch.java)
2. 贪婪大陆 (Code02_Bombs.java)
3. 无聊的数列 (Code03_BoringSequence.java)
4. 平均数和方差 (Code04_MeanVariance1.java, Code04_MeanVariance2.java)
5. 色板游戏 (ExtraQuestion.java)
6. 区域和检索 - 数组可修改 (Code05_RangeSumQueryMutable.java, Code05_RangeSumQueryMutable.py)
7. 天际线问题 (Code06_TheSkylineProblem.java, Code06_TheSkylineProblem.py)
8. 掉落的方块 (Code07_FallingSquares.java, Code07_FallingSquares.py)

新增题目（3个）
1. 二维区域和检索 - 可变 (Code08_RangeSumQuery2D.java, Code08_RangeSumQuery2D.py)
2. 计算右侧小于当前元素的个数 (Code09_CountOfSmallerNumbersAfterSelf.java, Code09_CountOfSmallerNumbersAfterSelf.py)
3. 翻转对 (Code10_ReversePairs.java, Code10_ReversePairs.py)

文件清单

文档文件
- README.md - 主要说明文档，已更新包含新增题目
- ADDITIONAL_SEGMENT_TREE_PROBLEMS.md - 补充题目大全
- SUMMARY.md - 本总结文件

Java实现
- Code08_RangeSumQuery2D.java - 二维区域和检索
- Code09_CountOfSmallerNumbersAfterSelf.java - 计算右侧小于当前元素的个数
- Code10_ReversePairs.java - 翻转对
- Main.java - 二维区域和检索的主类实现

Python实现
- Code08_RangeSumQuery2D.py - 二维区域和检索
- Code09_CountOfSmallerNumbersAfterSelf.py - 计算右侧小于当前元素的个数
- Code10_ReversePairs.py - 翻转对

C++实现
- Code09_CountOfSmallerNumbersAfterSelf.cpp - 计算右侧小于当前元素的个数（简化版）

技术要点

线段树核心概念
1. **区间查询**：支持在O(log n)时间内查询区间信息
2. **区间更新**：支持在O(log n)时间内更新区间信息
3. **懒惰传播**：优化区间更新操作的重要技术

4. **离散化**：处理大数据范围的有效方法
5. **动态开点**：节省空间的线段树实现方式

实现细节
1. **时间复杂度**：所有操作均为O(log n)
2. **空间复杂度**：通常需要4n的空间
3. **语言特性**：
   - Java：面向对象封装，自动内存管理
   - Python：语法简洁，动态类型
   - C++：性能优秀，需要手动管理内存

工程化考虑
1. **异常处理**：输入验证，边界检查
2. **性能优化**：懒惰传播，剪枝优化
3. **可维护性**：模块化设计，清晰接口

测试验证

所有新增的Java和Python代码都已通过测试验证：
- Code08_RangeSumQuery2D: 输出 8 和 10
- Code09_CountOfSmallerNumbersAfterSelf: 输出 [2, 1, 1, 0], [0], [0, 0]
- Code10_ReversePairs: 输出 2 和 1

应用场景

线段树适用于以下场景：
1. 频繁的区间查询和更新操作
2. 数据规模较大，需要高效处理
3. 需要维护区间统计信息（和、最值等）
4. 动态数据处理，支持实时更新

总结

通过本次扩展，class112线段树专题得到了全面完善，涵盖了来自各大算法平台的经典题目，提供了多种编程语言的实现，为深入学习和掌握线段树数据结构奠定了坚实基础。

==========

[代码文件]
==========
文件: Code01_Switch.java
==========
```
package class112;
```

```
// 开关问题 - 线段树实现
// 题目来源：洛谷 P3870 https://www.luogu.com.cn/problem/P3870
//
// 题目描述：
// 现有 n 盏灯排成一排，从左到右依次编号为 1~n，一开始所有的灯都是关着的
// 操作分两种：
// 操作 0 l r : 改变 l~r 范围上所有灯的状态，开着的灯关上、关着的灯打开
// 操作 1 l r : 查询 l~r 范围上有多少灯是打开的
//
// 解题思路：
// 使用线段树配合懒惰传播来高效处理区间翻转和区间求和操作
// 1. 线段树节点维护区间内开着的灯的数量
// 2. 使用 reverse 数组作为懒惰标记，记录区间是否需要翻转
// 3. 翻转操作时，区间内开着的灯数量变为区间长度减去原来的数量
//
// 核心思想：
// 1. 线段树是一种二叉树结构，每个节点代表数组的一个区间
// 2. 懒惰传播用于延迟更新，只有在需要时才将更新操作传递给子节点
// 3. 翻转操作的实现：当一个区间需要翻转时，开着的灯数量变为区间长度减去原来的数量
//
// 时间复杂度分析：
// - 建树：O(n)
// - 区间翻转：O(log n)
// - 区间查询：O(log n)
// 空间复杂度：O(n)
//
// 请同学们务必参考如下代码中关于输入、输出的处理
// 这是输入输出处理效率很高的写法
// 提交以下的 code，提交时请把类名改成"Main"，可以直接通过

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

public class Code01_Switch {

    // 最大节点数，设置为 100001*4 以确保足够的空间
    public static int MAXN = 100001;
```

```
// light[i] 表示线段树节点 i 维护的区间内开着的灯的数量
public static int[] light = new int[MAXN << 2];

// reverse[i] 表示线段树节点 i 是否需要翻转的懒惰标记
public static boolean[] reverse = new boolean[MAXN << 2];

/**
 * 向上更新函数
 * 更新当前节点的值为左右子节点值的和
 * 在线段树中，父节点的值通常由子节点的值计算得出
 * 对于本问题，父节点维护的区间内开着的灯数量等于左右子节点维护区间内开着的灯数量之和
 * @param i 当前节点在线段树数组中的索引
 */
public static void up(int i) {
    light[i] = light[i << 1] + light[i << 1 | 1];
}

/**
 * 向下传递懒惰标记
 * 将当前节点的懒惰标记传递给左右子节点
 * 懒惰传播是线段树优化的重要技术，用于延迟更新操作
 * 只有在真正需要访问子节点时才将更新操作传递下去，避免不必要的计算
 * @param i 当前节点在线段树数组中的索引
 * @param ln 左子树节点数量
 * @param rn 右子树节点数量
 */
public static void down(int i, int ln, int rn) {
    if (reverse[i]) {
        // 将翻转标记传递给左右子节点
        lazy(i << 1, ln);
        lazy(i << 1 | 1, rn);
        // 清除当前节点的翻转标记
        reverse[i] = false;
    }
}

/**
 * 懒惰标记处理函数
 * 对节点 i 进行翻转操作，开着的灯数量变为区间长度减去原来的数量
 * 这是区间翻转操作的核心实现
 * 翻转后：开着的灯数量 = 区间总长度 - 原来开着的灯数量
 * 同时更新懒惰标记的状态
 * @param i 要翻转的节点在线段树数组中的索引
```

```
 * @param n 该节点维护的区间长度
 */
public static void lazy(int i, int n) {
    light[i] = n - light[i];
    reverse[i] = !reverse[i];
}

/**
 * 构建线段树
 * 采用递归方式构建线段树，每个节点维护一个区间的信息
 * 叶子节点对应数组中的单个元素，非叶子节点对应区间的合并结果
 * @param l 区间左边界
 * @param r 区间右边界
 * @param i 当前节点在线段树数组中的索引
 */
public static void build(int l, int r, int i) {
    if (l == r) {
        // 叶子节点，初始所有灯都是关闭的
        light[i] = 0;
    } else {
        int mid = (l + r) >> 1;
        // 递归构建左子树
        build(l, mid, i << 1);
        // 递归构建右子树
        build(mid + 1, r, i << 1 | 1);
        // 向上更新当前节点的值
        up(i);
    }
    // 初始化懒惰标记为 false
    reverse[i] = false;
}

/**
 * 区间翻转操作
 * 将区间[jobl, jobr]内的所有灯状态翻转
 * 利用懒惰传播优化，避免对每个元素逐一翻转
 * @param jobl 操作区间左边界
 * @param jobr 操作区间右边界
 * @param l 当前节点维护的区间左边界
 * @param r 当前节点维护的区间右边界
 * @param i 当前节点在线段树数组中的索引
 */
public static void reverse(int jobl, int jobr, int l, int r, int i) {
```

```
    if (jobl <= l && r <= jobr) {
        // 当前节点维护的区间完全包含在操作区间内，直接打懒惰标记
        // 这是懒惰传播的关键：只标记不立即执行
        lazy(i, r - l + 1);
    } else {
        int mid = (l + r) >> 1;
        // 向下传递懒惰标记
        down(i, mid - l + 1, r - mid);
        // 递归处理左子树
        if (jobl <= mid) {
            reverse(jobl, jobr, l, mid, i << 1);
        }
        // 递归处理右子树
        if (jobr > mid) {
            reverse(jobl, jobr, mid + 1, r, i << 1 | 1);
        }
        // 向上更新当前节点的值
        up(i);
    }
}

/**
 * 区间查询操作
 * 查询区间[jobl, jobr]内开着的灯的数量
 * 在查询过程中需要确保懒惰标记已经正确传递
 * @param jobl 查询区间左边界
 * @param jobr 查询区间右边界
 * @param l 当前节点维护的区间左边界
 * @param r 当前节点维护的区间右边界
 * @param i 当前节点在线段树数组中的索引
 * @return 区间内开着的灯的数量
 */
public static int query(int jobl, int jobr, int l, int r, int i) {
    if (jobl <= l && r <= jobr) {
        // 当前节点维护的区间完全包含在查询区间内，直接返回节点值
        return light[i];
    }
    int mid = (l + r) >> 1;
    // 向下传递懒惰标记
    // 在查询时必须确保懒惰标记已经传递，以保证结果正确
    down(i, mid - l + 1, r - mid);
    int ans = 0;
    // 递归查询左子树
```

```
        if (jobl <= mid) {
            ans += query(jobl, jobr, l, mid, i << 1);
        }
        // 递归查询右子树
        if (jobr > mid) {
            ans += query(jobl, jobr, mid + 1, r, i << 1 | 1);
        }
        return ans;
    }

    public static void main(String[] args) throws IOException {
        // 使用高效的IO处理方式
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        StreamTokenizer in = new StreamTokenizer(br);
        PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));

        // 读取输入参数
        in.nextToken();
        int n = (int) in.nval;  // 灯的数量
        in.nextToken();
        int m = (int) in.nval;  // 操作的数量

        // 构建线段树
        // 初始时所有灯都是关闭的，所以每个节点的值都为0
        build(1, n, 1);

        // 处理m个操作
        for (int i = 1, op, jobl, jobr; i <= m; i++) {
            in.nextToken();
            op = (int) in.nval;     // 操作类型：0表示翻转，1表示查询
            in.nextToken();
            jobl = (int) in.nval;   // 操作区间左边界
            in.nextToken();
            jobr = (int) in.nval;   // 操作区间右边界

            if (op == 0) {
                // 翻转操作：改变[jobl, jobr]范围内所有灯的状态
                // 利用线段树和懒惰传播高效处理区间翻转
                reverse(jobl, jobr, 1, n, 1);
            } else {
                // 查询操作：查询[jobl, jobr]范围内开着的灯的数量
                // 利用线段树高效处理区间查询
                out.println(query(jobl, jobr, 1, n, 1));
```

```
            }
        }

        // 刷新输出缓冲区并关闭资源
        out.flush();
        out.close();
        br.close();
    }

}
```

==========

文件: Code02_Bombs.java

==========

```
package class112;

// 贪婪大陆 - 线段树实现
// 题目来源：洛谷 P2184 https://www.luogu.com.cn/problem/P2184
//
// 题目描述：
// 一共有 n 个格子，编号 1~n，开始时格子上没有地雷，实现两种操作，一共调用 m 次
// 操作 1 l r : 在 l~r 范围的格子上放置一种新型地雷，每次地雷都是新款
// 操作 2 l r : 查询 l~r 范围的格子上一共放置过多少款不同的地雷
//
// 解题思路：
// 使用两个线段树分别维护地雷范围的起始点和终止点
// 1. 当在[l,r]范围放置地雷时，在起始点线段树的 l 位置+1，在终止点线段树的 r 位置+1
// 2. 查询[l,r]范围内不同地雷数量时，计算起始点在[1,r]范围内的地雷数量减去终止点在[1,l-1]范围内的地雷数量
//
// 核心思想：
// 对于任意一个查询区间[l,r]，落在这个区间内的地雷种类数等于：
// 起始点在[1,r]范围内的地雷数量 - 终止点在[1,l-1]范围内的地雷数量
//
// 为什么这个方法是正确的？
// 1. 每个地雷都有唯一的起始点和终止点
// 2. 如果一个地雷的起始点在[1,r]范围内，说明这个地雷可能影响到查询区间
// 3. 但如果这个地雷的终止点在[1,l-1]范围内，说明这个地雷完全在查询区间左侧，不会影响查询区间
// 4. 因此，落在查询区间内的地雷种类数 = 可能影响查询区间的地雷数 - 完全在查询区间左侧的地雷数
//
// 时间复杂度分析：
// - 建树：O(n)
```

```
// - 区间更新：O(log n)
// - 区间查询：O(log n)
// 空间复杂度：O(n)
//
// 请同学们务必参考如下代码中关于输入、输出的处理
// 这是输入输出处理效率很高的写法
// 提交以下的 code，提交时请把类名改成"Main"，可以直接通过

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

public class Code02_Bombs {

    // 最大节点数，设置为 100001*4 以确保足够的空间
    public static int MAXN = 100001;

    // bombStarts[i] 表示线段树节点 i 维护的区间内新增地雷范围起始点的数量
    public static int[] bombStarts = new int[MAXN << 2];

    // bombEnds[i] 表示线段树节点 i 维护的区间内新增地雷范围终止点的数量
    public static int[] bombEnds = new int[MAXN << 2];

    /**
     * 向上更新函数
     * 更新当前节点的值为左右子节点值的和
     * 由于我们需要同时维护起始点和终止点两个信息，所以需要分别更新
     * @param i 当前节点在线段树数组中的索引
     */
    public static void up(int i) {
        // 更新起始点计数：当前节点的起始点数量等于左右子节点起始点数量之和
        bombStarts[i] = bombStarts[i << 1] + bombStarts[i << 1 | 1];
        // 更新终止点计数：当前节点的终止点数量等于左右子节点终止点数量之和
        bombEnds[i] = bombEnds[i << 1] + bombEnds[i << 1 | 1];
    }

    /**
     * 构建线段树
     * 采用递归方式构建线段树，每个节点维护一个区间的信息
     * 叶子节点对应数组中的单个元素，非叶子节点对应区间的合并结果
```

```
     * @param l 区间左边界
     * @param r 区间右边界
     * @param i 当前节点在线段树数组中的索引
     */
    public static void build(int l, int r, int i) {
        if (l < r) {
            int mid = (l + r) >> 1;
            // 递归构建左子树
            build(l, mid, i << 1);
            // 递归构建右子树
            build(mid + 1, r, i << 1 | 1);
        }
        // 初始化起始点和终止点计数为0
        // 初始时没有任何地雷放置，所以起始点和终止点计数都为0
        bombStarts[i] = 0;
        bombEnds[i] = 0;
    }

    /**
     * 添加地雷范围的起始点或终止点
     * 这是单点更新操作，用于记录地雷范围的边界信息
     * @param jobt 操作类型：0表示添加起始点，1表示添加终止点
     * @param jobi 操作位置
     * @param l 当前节点维护的区间左边界
     * @param r 当前节点维护的区间右边界
     * @param i 当前节点在线段树数组中的索引
     */
    // jobt==0表示在添加地雷范围的开头，jobi就是地雷范围的开头位置
    // jobt==1表示在添加地雷范围的结尾，jobi就是地雷范围的结尾位置
    public static void add(int jobt, int jobi, int l, int r, int i) {
        if (l == r) {
            // 到达叶子节点，更新对应计数
            // 叶子节点对应数组中的一个具体位置
            if (jobt == 0) {
                // 添加起始点：在起始点线段树的对应位置计数加1
                bombStarts[i]++;
            } else {
                // 添加终止点：在终止点线段树的对应位置计数加1
                bombEnds[i]++;
            }
        } else {
            int mid = (l + r) >> 1;
            // 根据位置递归更新左子树或右子树
```

```
        // 如果操作位置在左半区间，则更新左子树
        if (jobi <= mid) {
            add(jobt, jobi, l, mid, i << 1);
        } else {
            // 如果操作位置在右半区间，则更新右子树
            add(jobt, jobi, mid + 1, r, i << 1 | 1);
        }
        // 向上更新当前节点的值
        // 将子节点的更新结果合并到当前节点
        up(i);
    }
}

/**
 * 查询区间内起始点或终止点的数量
 * 这是区间查询操作，用于统计指定区间内的地雷边界数量
 * @param jobt 查询类型：0 表示查询起始点，1 表示查询终止点
 * @param jobl 查询区间左边界
 * @param jobr 查询区间右边界
 * @param l 当前节点维护的区间左边界
 * @param r 当前节点维护的区间右边界
 * @param i 当前节点在线段树数组中的索引
 * @return 区间内起始点或终止点的数量
 */
// jobt==0 表示在查询[jobl ~ jobr]范围上有多少地雷范围的开头
// jobt==1 表示在查询[jobl ~ jobr]范围上有多少地雷范围的结尾
public static int query(int jobt, int jobl, int jobr, int l, int r, int i) {
    if (jobl <= l && r <= jobr) {
        // 当前节点维护的区间完全包含在查询区间内，直接返回节点值
        // 这是线段树查询的优化点：如果当前区间完全在查询区间内，直接返回结果
        return jobt == 0 ? bombStarts[i] : bombEnds[i];
    } else {
        int mid = (l + r) >> 1;
        int ans = 0;
        // 递归查询左子树
        // 只有当查询区间与左子树区间有交集时才继续查询
        if (jobl <= mid) {
            ans += query(jobt, jobl, jobr, l, mid, i << 1);
        }
        // 递归查询右子树
        // 只有当查询区间与右子树区间有交集时才继续查询
        if (jobr > mid) {
            ans += query(jobt, jobl, jobr, mid + 1, r, i << 1 | 1);
```

```
        }
        return ans;
    }
}

public static void main(String[] args) throws IOException {
    // 使用高效的 IO 处理方式
    BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
    StreamTokenizer in = new StreamTokenizer(br);
    PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));

    // 读取输入参数
    in.nextToken();
    int n = (int) in.nval;  // 格子数量
    in.nextToken();
    int m = (int) in.nval;  // 操作数量

    // 构建线段树
    // 初始时没有任何地雷放置，所以起始点和终止点计数都为 0
    build(1, n, 1);

    // 处理 m 个操作
    for (int i = 1, op, jobl, jobr; i <= m; i++) {
        in.nextToken();
        op = (int) in.nval;     // 操作类型：1 表示添加地雷，2 表示查询
        in.nextToken();
        jobl = (int) in.nval;   // 操作区间左边界
        in.nextToken();
        jobr = (int) in.nval;   // 操作区间右边界

        if (op == 1) {
            // 添加地雷操作：在起始点和终止点分别+1
            // 每次放置地雷都是新款，所以需要记录其起始点和终止点
            add(0, jobl, 1, n, 1);  // 在起始点线段树的 jobl 位置+1
            add(1, jobr, 1, n, 1);  // 在终止点线段树的 jobr 位置+1
        } else {
            // 查询操作：计算区间内不同地雷种类数
            // 根据核心思想：落在查询区间内的地雷种类数 =
            // 起始点在[1,jobr]范围内的地雷数量 - 终止点在[1,jobl-1]范围内的地雷数量

            // 起始点在[1,jobr]范围内的地雷数量
            // 这些地雷可能影响到查询区间[1,jobr]
            int s = query(0, 1, jobr, 1, n, 1);
```

```
                // 终止点在[1,jobl-1]范围内的地雷数量
                // 这些地雷完全在查询区间左侧，不会影响查询区间[jobl,jobr]
                // 如果 jobl==1，说明没有更左的区域，返回 0
                // 如果 jobl>1，说明有更左的区域，查询终止点数量
                int e = jobl == 1 ? 0 : query(1, 1, jobl - 1, 1, n, 1);

                // 不同地雷种类数 = 起始点数量 - 终止点数量
                // 这就是落在查询区间[jobl,jobr]内的地雷种类数
                out.println(s - e);
            }
        }

        // 刷新输出缓冲区并关闭资源
        out.flush();
        out.close();
        br.close();
    }

}
```

==========

文件: Code03_BoringSequence.java

==========

```
package class112;

// 无聊的数列 - 线段树 + 差分数组实现
// 题目来源：洛谷 P1438 https://www.luogu.com.cn/problem/P1438
//
// 题目描述：
// 给定一个长度为 n 的数组 arr，实现如下两种操作
// 操作 1 l r k d : arr[l..r]范围上的数依次加上等差数列，首项 k，公差 d
// 操作 2 p       : 查询 arr[p]的值
//
// 解题思路：
// 使用差分数组+线段树维护区间更新和单点查询
// 1. 利用差分数组的性质：对原数组区间[l,r]加上等差数列，等价于对差分数组进行特定位置的修改
// 2. 使用线段树维护差分数组，支持区间加法和前缀和查询
//
// 核心思想：
// 差分数组的性质：
// 1. 差分数组 diff[i] = arr[i] - arr[i-1] (i>0), diff[0] = arr[0]
```

```
// 2. 原数组可以通过差分数组的前缀和恢复：arr[i] = sum(diff[0..i])
// 3. 对原数组区间[l,r]加上值 v，等价于对差分数组进行以下操作：
//    - diff[l] += v
//    - diff[r+1] -= v (如果 r+1 在数组范围内)
//
// 对于在 arr[1..r]范围上加上首项为 k、公差为 d 的等差数列：
// 等差数列为：k, k+d, k+2d, ..., k+(r-l)d
// - 在差分数组的 l 位置加上 k
// - 在差分数组的[l+1,r]范围上加上 d
// - 在差分数组的 r+1 位置减去 k+d*(r-l)
//
// 查询 arr[p]的值即为差分数组的前缀和 query(1,p)
//
// 时间复杂度分析：
// - 建树：O(n)
// - 区间更新：O(log n)
// - 单点查询：O(log n)
// 空间复杂度：O(n)
//
// 请同学们务必参考如下代码中关于输入、输出的处理
// 这是输入输出处理效率很高的写法
// 提交以下的 code，提交时请把类名改成"Main"，可以直接通过

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

public class Code03_BoringSequence {

    // 最大节点数，设置为 100001*4 以确保足够的空间
    public static int MAXN = 100001;

    // diff[i] 表示原数组的差分数组，diff[i] = arr[i] - arr[i-1]
    public static int[] diff = new int[MAXN];

    // sum[i] 表示线段树节点 i 维护的区间内差分值的和
    public static long[] sum = new long[MAXN << 2];

    // add[i] 表示线段树节点 i 的懒惰标记，记录区间需要加上的值
    public static long[] add = new long[MAXN << 2];
```

```
/**
 * 向上更新函数
 * 更新当前节点的值为左右子节点值的和
 * 在线段树中，父节点的值通常由子节点的值计算得出
 * 对于本问题，父节点维护的区间差分值之和等于左右子节点维护区间差分值之和
 * @param i 当前节点在线段树数组中的索引
 */
public static void up(int i) {
    sum[i] = sum[i << 1] + sum[i << 1 | 1];
}

/**
 * 向下传递懒惰标记
 * 将当前节点的懒惰标记传递给左右子节点
 * 懒惰传播是线段树优化的重要技术，用于延迟更新操作
 * 只有在真正需要访问子节点时才将更新操作传递下去，避免不必要的计算
 * @param i 当前节点在线段树数组中的索引
 * @param ln 左子树节点数量
 * @param rn 右子树节点数量
 */
public static void down(int i, int ln, int rn) {
    if (add[i] != 0) {
        // 将加法标记传递给左右子节点
        // 左子树区间长度为 ln，右子树区间长度为 rn
        lazy(i << 1, add[i], ln);
        lazy(i << 1 | 1, add[i], rn);
        // 清除当前节点的加法标记
        // 标记已传递，当前节点的懒惰标记清零
        add[i] = 0;
    }
}

/**
 * 懒惰标记处理函数
 * 对节点 i 维护的区间加上值 v
 * 这是区间加法操作的核心实现
 * 当对长度为 n 的区间加上值 v 时：
 * 1. 更新懒惰标记：add[i] += v
 * 2. 更新节点值：sum[i] += v * n (因为区间内每个元素都加 v，总共加 v*n)
 * @param i 要更新的节点在线段树数组中的索引
 * @param v 要加上的值
 * @param n 该节点维护的区间长度
```

```
 */
public static void lazy(int i, long v, int n) {
    add[i] += v;            // 更新懒惰标记，记录区间需要加上的值
    sum[i] += v * n;        // 更新节点值，区间内每个元素都加 v，总共加 v*n
}

/**
 * 构建线段树
 * 采用递归方式构建线段树，每个节点维护一个区间的信息
 * 叶子节点对应数组中的单个元素，非叶子节点对应区间的合并结果
 * @param l 区间左边界
 * @param r 区间右边界
 * @param i 当前节点在线段树数组中的索引
 */
public static void build(int l, int r, int i) {
    if (l == r) {
        // 叶子节点，初始化为差分数组的值
        // 叶子节点对应差分数组中的一个具体元素
        sum[i] = diff[l];
    } else {
        int mid = (l + r) >> 1;
        // 递归构建左子树
        build(l, mid, i << 1);
        // 递归构建右子树
        build(mid + 1, r, i << 1 | 1);
        // 向上更新当前节点的值
        // 将左右子节点的值合并到当前节点
        up(i);
    }
    // 初始化懒惰标记为 0
    // 初始时没有任何区间更新操作，懒惰标记为 0
    add[i] = 0;
}

/**
 * 区间加法操作
 * 对区间[jobl, jobr]内的所有元素加上值 jobv
 * 利用懒惰传播优化，避免对每个元素逐一加法
 * @param jobl 操作区间左边界
 * @param jobr 操作区间右边界
 * @param jobv 要加上的值
 * @param l 当前节点维护的区间左边界
 * @param r 当前节点维护的区间右边界
```

```
     * @param i 当前节点在线段树数组中的索引
     */
    public static void add(int jobl, int jobr, long jobv, int l, int r, int i) {
        if (jobl <= l && r <= jobr) {
            // 当前节点维护的区间完全包含在操作区间内，直接打懒惰标记
            // 这是懒惰传播的关键：只标记不立即执行
            // 区间长度为 r-l+1，要加上的值为 jobv
            lazy(i, jobv, r - l + 1);
        } else {
            int mid = (l + r) >> 1;
            // 向下传递懒惰标记
            // 在递归处理子节点之前，需要确保当前节点的懒惰标记已经传递
            down(i, mid - l + 1, r - mid);
            // 递归处理左子树
            // 只有当操作区间与左子树区间有交集时才继续处理
            if (jobl <= mid) {
                add(jobl, jobr, jobv, l, mid, i << 1);
            }
            // 递归处理右子树
            // 只有当操作区间与右子树区间有交集时才继续处理
            if (jobr > mid) {
                add(jobl, jobr, jobv, mid + 1, r, i << 1 | 1);
            }
            // 向上更新当前节点的值
            // 将子节点的更新结果合并到当前节点
            up(i);
        }
    }

    /**
     * 区间查询操作
     * 查询区间[jobl, jobr]内元素的和
     * 在查询过程中需要确保懒惰标记已经正确传递
     * @param jobl 查询区间左边界
     * @param jobr 查询区间右边界
     * @param l 当前节点维护的区间左边界
     * @param r 当前节点维护的区间右边界
     * @param i 当前节点在线段树数组中的索引
     * @return 区间内元素的和
     */
    public static long query(int jobl, int jobr, int l, int r, int i) {
        if (jobl <= l && r <= jobr) {
            // 当前节点维护的区间完全包含在查询区间内，直接返回节点值
```

```
        // 这是线段树查询的优化点：如果当前区间完全在查询区间内，直接返回结果
        return sum[i];
    }
    int mid = (l + r) >> 1;
    // 向下传递懒惰标记
    // 在查询时必须确保懒惰标记已经传递，以保证结果正确
    down(i, mid - l + 1, r - mid);
    long ans = 0;
    // 递归查询左子树
    // 只有当查询区间与左子树区间有交集时才继续查询
    if (jobl <= mid) {
        ans += query(jobl, jobr, l, mid, i << 1);
    }
    // 递归查询右子树
    // 只有当查询区间与右子树区间有交集时才继续查询
    if (jobr > mid) {
        ans += query(jobl, jobr, mid + 1, r, i << 1 | 1);
    }
    return ans;
}

public static void main(String[] args) throws IOException {
    // 使用高效的 IO 处理方式
    BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
    StreamTokenizer in = new StreamTokenizer(br);
    PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));

    // 读取输入参数
    in.nextToken();
    int n = (int) in.nval;  // 数组长度
    in.nextToken();
    int m = (int) in.nval;  // 操作数量

    // 构造差分数组
    // 差分数组 diff[i] = arr[i] - arr[i-1] (i>0), diff[0] = arr[0]
    // 通过差分数组可以快速进行区间更新操作
    for (int i = 1, pre = 0, cur; i <= n; i++) {
        in.nextToken();
        cur = (int) in.nval;
        diff[i] = cur - pre;  // 计算差分值
        pre = cur;
    }
```

```
// 构建线段树
// 用差分数组的值初始化线段树，支持区间加法和前缀和查询
build(1, n, 1);

// 处理 m 个操作
for (int i = 1, op; i <= m; i++) {
    in.nextToken();
    op = (int) in.nval;  // 操作类型：1 表示区间加等差数列，2 表示单点查询

    if (op == 1) {
        // 区间加等差数列操作
        // 在 arr[jobl..jobr]范围上加上首项为 k、公差为 d 的等差数列
        in.nextToken();
        int jobl = (int) in.nval;   // 操作区间左边界
        in.nextToken();
        int jobr = (int) in.nval;   // 操作区间右边界
        in.nextToken();
        long k = (long) in.nval;    // 等差数列首项
        in.nextToken();
        long d = (long) in.nval;    // 等差数列公差

        // 计算等差数列末项值
        // 等差数列为：k, k+d, k+2d, ..., k+(jobr-jobl)d
        long e = k + d * (jobr - jobl);

        // 根据差分数组的性质，对原数组区间加上等差数列等价于对差分数组进行以下操作：
        // 1. 在差分数组的 jobl 位置加上 k
        add(jobl, jobl, k, 1, n, 1);
        // 2. 在差分数组的[jobl+1,jobr]范围上加上 d
        if (jobl + 1 <= jobr) {
            add(jobl + 1, jobr, d, 1, n, 1);
        }
        // 3. 在差分数组的 jobr+1 位置减去末项值
        if (jobr < n) {
            add(jobr + 1, jobr + 1, -e, 1, n, 1);
        }
    } else {
        // 单点查询操作：查询 arr[p]的值即为差分数组的前缀和
        // 根据差分数组的性质：arr[i] = sum(diff[0..i])
        in.nextToken();
        int p = (int) in.nval;  // 查询位置
        // 查询差分数组[1,p]范围的和，即为 arr[p]的值
        out.println(query(1, p, 1, n, 1));
```

```
            }
        }

        // 刷新输出缓冲区并关闭资源
        out.flush();
        out.close();
        br.close();
    }

}
```

==========

文件: Code04_DoubleString.java

==========

```
package class112;

// 读取科学计数法表达的 double 数字
// 如果输入的字符串代表 double 数字的科学计数法形式
// 用 StreamTokenizer 读取会出错
// 用 StringTokenizer 读取正确但比较费内存
// 参考链接 : https://oi-wiki.org/lang/java-pro/

import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.io.OutputStream;
import java.io.OutputStreamWriter;
import java.io.PrintWriter;
import java.io.StreamTokenizer;
import java.util.StringTokenizer;

public class Code04_DoubleString {

    /*
输入如下
5
10.203E+0000
5.6920E+0001
30.888E-0001
400.20E+0002
```

```
0.2373E-0002
     */
    public static void main(String[] args) throws IOException {
        // test1();
        test2();
    }

    // Kattio 类 IO 效率很好，但还是不如 StreamTokenizer
    // 只有 StreamTokenizer 无法正确处理时，才考虑使用这个类
    // 参考链接 : https://oi-wiki.org/lang/java-pro/
    public static class Kattio extends PrintWriter {
        private BufferedReader r;
        private StringTokenizer st;

        public Kattio() {
            this(System.in, System.out);
        }

        public Kattio(InputStream i, OutputStream o) {
            super(o);
            r = new BufferedReader(new InputStreamReader(i));
        }

        public Kattio(String intput, String output) throws IOException {
            super(output);
            r = new BufferedReader(new FileReader(intput));
        }

        public String next() {
            try {
                while (st == null || !st.hasMoreTokens())
                    st = new StringTokenizer(r.readLine());
                return st.nextToken();
            } catch (Exception e) {
            }
            return null;
        }

        public int nextInt() {
            return Integer.parseInt(next());
        }

        public double nextDouble() {
```

```
        return Double.parseDouble(next());
    }

    public long nextLong() {
        return Long.parseLong(next());
    }
}

// StreamTokenizer无法正确读取
public static void test1() throws IOException {
    System.out.println("测试StreamTokenizer");
    System.out.println("输入 : ");
    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;
    double num;
    for (int i = 1; i <= n; i++) {
        in.nextToken();
        num = (double) in.nval;
        out.println(num);
    }
    out.flush();
    out.close();
    br.close();
}

// StringTokenizer可以正确读取
public static void test2() throws IOException {
    System.out.println("测试StringTokenizer");
    System.out.println("输入 : ");
    Kattio io = new Kattio();
    int n = io.nextInt();
    double num;
    for (int i = 1; i <= n; i++) {
        num = io.nextDouble();
        io.println(num);
    }
    io.flush();
    io.close();
}
```

}

==========

文件: Code04_MeanVariance1.java
==========
```
package class112;

// 平均数和方差 - 线段树实现
// 题目来源：洛谷 P1471 https://www.luogu.com.cn/problem/P1471
//
// 题目描述：
// 给定一个长度为 n 的数组 arr，操作分为三种类型，一共调用 m 次
// 操作 1 l r : arr 数组中[l, r]范围上每个数字加上 k，k 为 double 类型
// 操作 2 l r : 查询 arr 数组中[l, r]范围上所有数字的平均数，返回 double 类型
// 操作 3 l r : 查询 arr 数组中[l, r]范围上所有数字的方差，返回 double 类型
//
// 解题思路：
// 使用线段树维护区间和与区间平方和来计算平均数和方差
// 1. 维护两个信息：区间和 sum1 和区间平方和 sum2
// 2. 利用数学公式：平均数 = 区间和 / 区间长度，方差 = 区间平方和/区间长度 - (区间和/区间长度)^2
// 3. 区间加法操作时，需要同时更新 sum1 和 sum2
//
// 核心思想：
// 数学原理：
// 1. 平均数公式：mean = Σxi / n
// 2. 方差公式：variance = Σ(xi - mean)^2 / n = Σxi^2 / n - (Σxi / n)^2
//
// 当区间内每个数都加上 v 时：
// 假设原区间有 n 个数，和为 S，平方和为 S2
// 新的区间和 S' = S + n*v
// 新的区间平方和 S2' = Σ(xi + v)^2 = Σ(xi^2 + 2*v*xi + v^2) = S2 + 2*v*S + n*v^2
//
// 时间复杂度分析：
// - 建树：O(n)
// - 区间更新：O(log n)
// - 区间查询：O(log n)
// 空间复杂度：O(n)
//
// 如下实现是正确的，但是洛谷平台对空间卡的很严，只有使用 C++能全部通过
// C++版本就是本节代码中的 Code04_MeanVariance2 文件
// C++版本和 java 版本逻辑完全一样，但只有 C++版本可以通过所有测试用例
// java 的版本就是无法完全通过，空间会超过限制，主要是 IO 空间占用大
```

```
// 这是洛谷平台没有照顾各种语言的实现所导致的
// 在真正笔试、比赛时，一定是兼顾各种语言的，该实现是一定正确的

import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.io.OutputStream;
import java.io.PrintWriter;
import java.util.StringTokenizer;

public class Code04_MeanVariance1 {

    // 最大节点数，设置为 100001*4 以确保足够的空间
    public static int MAXN = 100001;

    // arr[i] 表示原数组第 i 个元素的值
    public static double[] arr = new double[MAXN];

    // sum1[i] 表示线段树节点 i 维护的区间内元素的和
    public static double[] sum1 = new double[MAXN << 2];

    // sum2[i] 表示线段树节点 i 维护的区间内元素平方的和
    public static double[] sum2 = new double[MAXN << 2];

    // addv[i] 表示线段树节点 i 的懒惰标记，记录区间内每个数字需要增加的值
    public static double[] addv = new double[MAXN << 2];

    /**
     * 向上更新函数
     * 更新当前节点的值为左右子节点值的和
     * 在线段树中，父节点的值通常由子节点的值计算得出
     * 对于本问题，父节点维护的区间和等于左右子节点维护区间和之和
     * 父节点维护的区间平方和等于左右子节点维护区间平方和之和
     * @param i 当前节点在线段树数组中的索引
     */
    public static void up(int i) {
        // 更新区间和：当前节点的区间和等于左右子节点区间和之和
        sum1[i] = sum1[i << 1] + sum1[i << 1 | 1];
        // 更新区间平方和：当前节点的区间平方和等于左右子节点区间平方和之和
        sum2[i] = sum2[i << 1] + sum2[i << 1 | 1];
    }
```

```
/**
 * 向下传递懒惰标记
 * 将当前节点的懒惰标记传递给左右子节点
 * 懒惰传播是线段树优化的重要技术，用于延迟更新操作
 * 只有在真正需要访问子节点时才将更新操作传递下去，避免不必要的计算
 * @param i 当前节点在线段树数组中的索引
 * @param ln 左子树节点数量
 * @param rn 右子树节点数量
 */
public static void down(int i, int ln, int rn) {
    if (addv[i] != 0) {
        // 将加法标记传递给左右子节点
        // 左子树区间长度为 ln，右子树区间长度为 rn
        lazy(i << 1, addv[i], ln);
        lazy(i << 1 | 1, addv[i], rn);
        // 清除当前节点的加法标记
        // 标记已传递，当前节点的懒惰标记清零
        addv[i] = 0;
    }
}

/**
 * 懒惰标记处理函数
 * 对节点 i 维护的区间内每个数字加上值 v
 * 这是区间加法操作的核心实现
 * 当对长度为 n 的区间加上值 v 时，需要同时更新区间和与区间平方和
 * @param i 要更新的节点在线段树数组中的索引
 * @param v 要加上的值
 * @param n 该节点维护的区间长度
 */
public static void lazy(int i, double v, int n) {
    // 更新平方和：新平方和 = 原平方和 + 2*v*原和 + v*v*区间长度
    // 数学推导：Σ(xi + v)^2 = Σxi^2 + 2*v*Σxi + n*v^2
    sum2[i] += sum1[i] * v * 2 + v * v * n;
    // 更新和：新和 = 原和 + v*区间长度
    // 数学推导：Σ(xi + v) = Σxi + n*v
    sum1[i] += v * n;
    // 更新懒惰标记
    // 记录区间需要加上的值，用于后续的懒惰传播
    addv[i] += v;
}
```

```
/**
 * 构建线段树
 * 采用递归方式构建线段树，每个节点维护一个区间的信息
 * 叶子节点对应数组中的单个元素，非叶子节点对应区间的合并结果
 * @param l 区间左边界
 * @param r 区间右边界
 * @param i 当前节点在线段树数组中的索引
 */
public static void build(int l, int r, int i) {
    if (l == r) {
        // 叶子节点，初始化为原数组的值和平方值
        // 叶子节点对应原数组中的一个具体元素
        sum1[i] = arr[l];          // 区间和即为该元素的值
        sum2[i] = arr[l] * arr[l]; // 区间平方和即为该元素的平方值
    } else {
        int mid = (l + r) >> 1;
        // 递归构建左子树
        build(l, mid, i << 1);
        // 递归构建右子树
        build(mid + 1, r, i << 1 | 1);
        // 向上更新当前节点的值
        // 将左右子节点的值合并到当前节点
        up(i);
    }
    // 初始化懒惰标记为0
    // 初始时没有任何区间更新操作，懒惰标记为0
    addv[i] = 0;
}

/**
 * 区间加法操作
 * 对区间[jobl, jobr]内的所有元素加上值 jobv
 * 利用懒惰传播优化，避免对每个元素逐一加法
 * @param jobl 操作区间左边界
 * @param jobr 操作区间右边界
 * @param jobv 要加上的值
 * @param l 当前节点维护的区间左边界
 * @param r 当前节点维护的区间右边界
 * @param i 当前节点在线段树数组中的索引
 */
public static void add(int jobl, int jobr, double jobv, int l, int r, int i) {
    if (jobl <= l && r <= jobr) {
        // 当前节点维护的区间完全包含在操作区间内，直接打懒惰标记
```

```
        // 这是懒惰传播的关键：只标记不立即执行
        // 区间长度为 r-l+1，要加上的值为 jobv
        lazy(i, jobv, r - l + 1);
    } else {
        int mid = (l + r) >> 1;
        // 向下传递懒惰标记
        // 在递归处理子节点之前，需要确保当前节点的懒惰标记已经传递
        down(i, mid - l + 1, r - mid);
        // 递归处理左子树
        // 只有当操作区间与左子树区间有交集时才继续处理
        if (jobl <= mid) {
            add(jobl, jobr, jobv, l, mid, i << 1);
        }
        // 递归处理右子树
        // 只有当操作区间与右子树区间有交集时才继续处理
        if (jobr > mid) {
            add(jobl, jobr, jobv, mid + 1, r, i << 1 | 1);
        }
        // 向上更新当前节点的值
        // 将子节点的更新结果合并到当前节点
        up(i);
    }
}

/**
 * 区间查询操作
 * 查询区间[jobl, jobr]内 sum 数组的和
 * 在查询过程中需要确保懒惰标记已经正确传递
 * @param sum 要查询的数组(sum1 或 sum2)
 * @param jobl 查询区间左边界
 * @param jobr 查询区间右边界
 * @param l 当前节点维护的区间左边界
 * @param r 当前节点维护的区间右边界
 * @param i 当前节点在线段树数组中的索引
 * @return 区间内 sum 数组的和
 */
public static double query(double[] sum, int jobl, int jobr, int l, int r, int i) {
    if (jobl <= l && r <= jobr) {
        // 当前节点维护的区间完全包含在查询区间内，直接返回节点值
        // 这是线段树查询的优化点：如果当前区间完全在查询区间内，直接返回结果
        return sum[i];
    }
    int mid = (l + r) >> 1;
```

```
    // 向下传递懒惰标记
    // 在查询时必须确保懒惰标记已经传递，以保证结果正确
    down(i, mid - l + 1, r - mid);
    double ans = 0;
    // 递归查询左子树
    // 只有当查询区间与左子树区间有交集时才继续查询
    if (jobl <= mid) {
        ans += query(sum, jobl, jobr, l, mid, i << 1);
    }
    // 递归查询右子树
    // 只有当查询区间与右子树区间有交集时才继续查询
    if (jobr > mid) {
        ans += query(sum, jobl, jobr, mid + 1, r, i << 1 | 1);
    }
    return ans;
}

public static void main(String[] args) throws IOException {
    // 使用 Kattio 类进行高效的输入输出处理
    Kattio io = new Kattio();

    // 读取输入参数
    int n = io.nextInt();   // 数组长度
    int m = io.nextInt();   // 操作数量

    // 读取初始数组
    // 数组下标从 1 开始，便于线段树处理
    for (int i = 1; i <= n; i++) {
        arr[i] = io.nextDouble();
    }

    // 构建线段树
    // 初始化线段树，维护区间和与区间平方和
    build(1, n, 1);

    double jobv;
    // 处理 m 个操作
    for (int i = 1, op, jobl, jobr; i <= m; i++) {
        op = io.nextInt();  // 操作类型

        if (op == 1) {
            // 区间加法操作
            // 对区间[jobl, jobr]内的所有元素加上值 jobv
```

```
            jobl = io.nextInt();   // 操作区间左边界
            jobr = io.nextInt();   // 操作区间右边界
            jobv = io.nextDouble(); // 要加上的值
            add(jobl, jobr, jobv, 1, n, 1);
        } else if (op == 2) {
            // 查询平均数操作
            // 平均数 = 区间和 / 区间长度
            jobl = io.nextInt();   // 查询区间左边界
            jobr = io.nextInt();   // 查询区间右边界
            // 查询区间和
            double sum = query(sum1, jobl, jobr, 1, n, 1);
            // 计算区间长度
            double size = jobr - jobl + 1;
            // 平均数 = 区间和 / 区间长度
            double ans = sum / size;
            io.println(String.format("%.4f", ans));
        } else {
            // 查询方差操作
            // 方差 = 区间平方和/区间长度 - (区间和/区间长度)^2
            jobl = io.nextInt();   // 查询区间左边界
            jobr = io.nextInt();   // 查询区间右边界

            // 查询区间和
            double a = query(sum1, jobl, jobr, 1, n, 1);
            // 查询区间平方和
            double b = query(sum2, jobl, jobr, 1, n, 1);
            // 计算区间长度
            double size = jobr - jobl + 1;

            // 方差 = 区间平方和/区间长度 - (区间和/区间长度)^2
            // 根据数学公式：variance = E(X^2) - [E(X)]^2
            double ans = b / size - (a / size) * (a / size);
            io.println(String.format("%.4f", ans));
        }
    }

    // 刷新输出缓冲区并关闭资源
    io.flush();
    io.close();
}

// Kattio 类 IO 效率很好，但还是不如 StreamTokenizer
// 只有 StreamTokenizer 无法正确处理时，才考虑使用这个类
```

```
    // 参考链接 : https://oi-wiki.org/lang/java-pro/
    public static class Kattio extends PrintWriter {
        private BufferedReader r;
        private StringTokenizer st;

        public Kattio() {
            this(System.in, System.out);
        }

        public Kattio(InputStream i, OutputStream o) {
            super(o);
            r = new BufferedReader(new InputStreamReader(i));
        }

        public Kattio(String intput, String output) throws IOException {
            super(output);
            r = new BufferedReader(new FileReader(intput));
        }

        public String next() {
            try {
                while (st == null || !st.hasMoreTokens())
                    st = new StringTokenizer(r.readLine());
                return st.nextToken();
            } catch (Exception e) {
            }
            return null;
        }

        public int nextInt() {
            return Integer.parseInt(next());
        }

        public double nextDouble() {
            return Double.parseDouble(next());
        }

        public long nextLong() {
            return Long.parseLong(next());
        }
    }

}
```

==========

文件: Code04_MeanVariance2.java
==========

```
package class112;

// 平均数和方差 - C++版本
// 给定一个长度为n的数组arr，操作分为三种类型，一共调用m次
// 操作 1 l r : arr数组中[l, r]范围上每个数字加上k，k为double类型
// 操作 2 l r : 查询arr数组中[l, r]范围上所有数字的平均数，返回double类型
// 操作 3 l r : 查询arr数组中[l, r]范围上所有数字的方差，返回double类型
// 测试链接 : https://www.luogu.com.cn/problem/P1471
//
// 解题思路：
// 使用线段树维护区间和与区间平方和来计算平均数和方差
// 1. 维护两个信息：区间和sum1和区间平方和sum2
// 2. 利用数学公式：平均数 = 区间和 / 区间长度，方差 = 区间平方和/区间长度 - (区间和/区间长度)^2
// 3. 区间加法操作时，需要同时更新sum1和sum2
//
// 核心思想：
// 数学原理：
// 1. 平均数公式：mean = Σxi / n
// 2. 方差公式：variance = Σ(xi - mean)^2 / n = Σxi^2 / n - (Σxi / n)^2
//
// 当区间内每个数都加上v时：
// 假设原区间有n个数，和为S，平方和为S2
// 新的区间和 S' = S + n*v
// 新的区间平方和 S2' = Σ(xi + v)^2 = Σ(xi^2 + 2*v*xi + v^2) = S2 + 2*v*S + n*v^2
//
// 如下实现是C++的版本，C++版本和java版本逻辑完全一样
// 提交如下代码，可以通过所有测试用例

//#include <cstdio>
//using namespace std;
//
//const int MAXN = 100001;
//
//double arr[MAXN];
//double sum1[MAXN << 2];
//double sum2[MAXN << 2];
//double addv[MAXN << 2];
//
```

```
//void up(int i) {
//    // 向上更新函数
//    // 更新当前节点的值为左右子节点值的和
//    // 在线段树中，父节点的值通常由子节点的值计算得出
//    // 对于本问题，父节点维护的区间和等于左右子节点维护区间和之和
//    // 父节点维护的区间平方和等于左右子节点维护区间平方和之和
//    sum1[i] = sum1[i << 1] + sum1[i << 1 | 1];
//    sum2[i] = sum2[i << 1] + sum2[i << 1 | 1];
//}
//
//void lazy(int i, double v, int n) {
//    // 懒惰标记处理函数
//    // 对节点 i 维护的区间内每个数字加上值 v
//    // 这是区间加法操作的核心实现
//    // 当对长度为 n 的区间加上值 v 时，需要同时更新区间和与区间平方和
//    // 更新平方和：新平方和 = 原平方和 + 2*v*原和 + v*v*区间长度
//    // 数学推导：Σ(xi + v)^2 = Σxi^2 + 2*v*Σxi + n*v^2
//    sum2[i] += sum1[i] * v * 2 + v * v * n;
//    // 更新和：新和 = 原和 + v*区间长度
//    // 数学推导：Σ(xi + v) = Σxi + n*v
//    sum1[i] += v * n;
//    // 更新懒惰标记
//    // 记录区间需要加上的值，用于后续的懒惰传播
//    addv[i] += v;
//}
//
//void down(int i, int ln, int rn) {
//    // 向下传递懒惰标记
//    // 将当前节点的懒惰标记传递给左右子节点
//    // 懒惰传播是线段树优化的重要技术，用于延迟更新操作
//    // 只有在真正需要访问子节点时才将更新操作传递下去，避免不必要的计算
//    if (addv[i] != 0) {
//        // 将加法标记传递给左右子节点
//        // 左子树区间长度为 ln，右子树区间长度为 rn
//        lazy(i << 1, addv[i], ln);
//        lazy(i << 1 | 1, addv[i], rn);
//        // 清除当前节点的加法标记
//        // 标记已传递，当前节点的懒惰标记清零
//        addv[i] = 0;
//    }
//}
//
//void build(int l, int r, int i) {
```

```
//    // 构建线段树
//    // 采用递归方式构建线段树，每个节点维护一个区间的信息
//    // 叶子节点对应数组中的单个元素，非叶子节点对应区间的合并结果
//    if (l == r) {
//        // 叶子节点，初始化为原数组的值和平方值
//        // 叶子节点对应原数组中的一个具体元素
//        sum1[i] = arr[l];           // 区间和即为该元素的值
//        sum2[i] = arr[l] * arr[l]; // 区间平方和即为该元素的平方值
//    } else {
//        int mid = (l + r) >> 1;
//        // 递归构建左子树
//        build(l, mid, i << 1);
//        // 递归构建右子树
//        build(mid + 1, r, i << 1 | 1);
//        // 向上更新当前节点的值
//        // 将左右子节点的值合并到当前节点
//        up(i);
//    }
//    // 初始化懒惰标记为 0
//    // 初始时没有任何区间更新操作，懒惰标记为 0
//    addv[i] = 0;
//}
//
//void add(int jobl, int jobr, double jobv, int l, int r, int i) {
//    // 区间加法操作
//    // 对区间[jobl, jobr]内的所有元素加上值 jobv
//    // 利用懒惰传播优化，避免对每个元素逐一加法
//    if (jobl <= l && r <= jobr) {
//        // 当前节点维护的区间完全包含在操作区间内，直接打懒惰标记
//        // 这是懒惰传播的关键：只标记不立即执行
//        // 区间长度为 r-l+1，要加上的值为 jobv
//        lazy(i, jobv, r - l + 1);
//    } else {
//        int mid = (l + r) >> 1;
//        // 向下传递懒惰标记
//        // 在递归处理子节点之前，需要确保当前节点的懒惰标记已经传递
//        down(i, mid - l + 1, r - mid);
//        // 递归处理左子树
//        // 只有当操作区间与左子树区间有交集时才继续处理
//        if (jobl <= mid) {
//            add(jobl, jobr, jobv, l, mid, i << 1);
//        }
//        // 递归处理右子树
```

```
//        // 只有当操作区间与右子树区间有交集时才继续处理
//        if (jobr > mid) {
//            add(jobl, jobr, jobv, mid + 1, r, i << 1 | 1);
//        }
//        // 向上更新当前节点的值
//        // 将子节点的更新结果合并到当前节点
//        up(i);
//    }
//}
//
//double query(double *sum, int jobl, int jobr, int l, int r, int i) {
//    // 区间查询操作
//    // 查询区间[jobl, jobr]内 sum 数组的和
//    // 在查询过程中需要确保懒惰标记已经正确传递
//    if (jobl <= l && r <= jobr) {
//        // 当前节点维护的区间完全包含在查询区间内，直接返回节点值
//        // 这是线段树查询的优化点：如果当前区间完全在查询区间内，直接返回结果
//        return sum[i];
//    }
//    int mid = (l + r) >> 1;
//    // 向下传递懒惰标记
//    // 在查询时必须确保懒惰标记已经传递，以保证结果正确
//    down(i, mid - l + 1, r - mid);
//    double ans = 0;
//    // 递归查询左子树
//    // 只有当查询区间与左子树区间有交集时才继续查询
//    if (jobl <= mid) {
//        ans += query(sum, jobl, jobr, l, mid, i << 1);
//    }
//    // 递归查询右子树
//    // 只有当查询区间与右子树区间有交集时才继续查询
//    if (jobr > mid) {
//        ans += query(sum, jobl, jobr, mid + 1, r, i << 1 | 1);
//    }
//    return ans;
//}
//
//int main() {
//    // 使用 C 标准输入输出库进行输入输出处理
//    int n, m;
//    // 读取输入参数
//    scanf("%d %d", &n, &m);   // n 为数组长度，m 为操作数量
//
```

```
//    // 读取初始数组
//    // 数组下标从 1 开始，便于线段树处理
//    for (int i = 1; i <= n; i++) {
//        scanf("%lf", &arr[i]);
//    }
//
//    // 构建线段树
//    // 初始化线段树，维护区间和与区间平方和
//    build(1, n, 1);
//
//    // 处理 m 个操作
//    for (int i = 1; i <= m; i++) {
//        int op, jobl, jobr;
//        scanf("%d", &op);  // 读取操作类型
//
//        if (op == 1) {
//            // 区间加法操作
//            // 对区间[jobl, jobr]内的所有元素加上值 jobv
//            double jobv;
//            scanf("%d %d %lf", &jobl, &jobr, &jobv); // 读取操作参数
//            add(jobl, jobr, jobv, 1, n, 1);
//        } else if (op == 2) {
//            // 查询平均数操作
//            // 平均数 = 区间和 / 区间长度
//            scanf("%d %d", &jobl, &jobr);   // 读取查询区间
//            // 查询区间和
//            double sum = query(sum1, jobl, jobr, 1, n, 1);
//            // 计算区间长度
//            double size = jobr - jobl + 1;
//            // 平均数 = 区间和 / 区间长度
//            double ans = sum / size;
//            printf("%.4f\n", ans);
//        } else {
//            // 查询方差操作
//            // 方差 = 区间平方和/区间长度 - (区间和/区间长度)^2
//            scanf("%d %d", &jobl, &jobr);   // 读取查询区间
//
//            // 查询区间和
//            double a = query(sum1, jobl, jobr, 1, n, 1);
//            // 查询区间平方和
//            double b = query(sum2, jobl, jobr, 1, n, 1);
//            // 计算区间长度
//            double size = jobr - jobl + 1;
```

```
//
//            // 方差 = 区间平方和/区间长度 - (区间和/区间长度)^2
//            // 根据数学公式：variance = E(X^2) - [E(X)]^2
//            double ans = b / size - (a / size) * (a / size);
//            printf("%.4f\n", ans);
//        }
//    }
//    return 0;
//}
```

==========

文件: Code05_RangeSumQueryMutable.cpp

==========

```
// 307. 区域和检索 - 数组可修改 - 线段树实现
// 题目来源：LeetCode 307 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。
// 其中一类查询要求更新数组 nums 下标对应的值
// 另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：
// 使用线段树来高效处理单点更新和区间求和操作
// 1. 线段树是一种二叉树结构，每个节点代表一个区间
// 2. 叶子节点代表数组中的单个元素
// 3. 非叶子节点代表其子节点区间的合并结果（这里是区间和）
//
// 核心思想：
// 线段树的特性：
// 1. 树状结构：线段树是一棵完全二叉树，便于用数组存储
// 2. 区间划分：每个节点代表一个区间，根节点代表整个区间[0, n-1]
// 3. 递归构建：非叶子节点的值等于其左右子节点值的和
// 4. 高效操作：单点更新和区间查询的时间复杂度均为O(log n)
//
// 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 <vector>
#include <iostream>
using namespace std;

// 线段树实现
class SegmentTree {
private:
    vector<int> tree;  // 线段树数组，存储区间和
    int n;             // 原数组长度

    /**
     * 构建线段树
     * 采用递归方式构建线段树，每个节点维护一个区间的信息
     * 叶子节点对应数组中的单个元素，非叶子节点对应区间的合并结果
     * @param nums 原始数组
     * @param node 当前线段树节点索引
     * @param start 当前节点表示区间的起始位置
     * @param end 当前节点表示区间的结束位置
     *
     * 时间复杂度: O(n)
     * 空间复杂度: O(log n) - 递归栈空间
     */
    void buildTree(vector<int>& nums, int node, int start, int end) {
        if (start == end) {
            // 叶子节点，存储数组元素值
            // 叶子节点对应原数组中的一个具体元素
            tree[node] = nums[start];
        } else {
            // 非叶子节点，需要递归构建左右子树
            int mid = (start + end) / 2;
            // 递归构建左子树
            // 左子节点索引为 2*node+1，表示区间[start, mid]
            buildTree(nums, 2 * node + 1, start, mid);
            // 递归构建右子树
            // 右子节点索引为 2*node+2，表示区间[mid+1, end]
            buildTree(nums, 2 * node + 2, mid + 1, end);
            // 合并左右子树的结果，存储区间和
            // 非叶子节点的值等于其左右子节点值的和
            tree[node] = tree[2 * node + 1] + tree[2 * node + 2];
```

```
    }
}

/**
 * 更新线段树中某个位置的值的辅助函数
 * 递归查找目标位置并更新，然后逐层向上更新父节点
 * @param node 当前线段树节点索引
 * @param start 当前节点表示区间的起始位置
 * @param end 当前节点表示区间的结束位置
 * @param index 要更新的数组索引
 * @param val 新的值
 */
void updateHelper(int node, int start, int end, int index, int val) {
    if (start == end) {
        // 找到叶子节点，更新值
        // 当区间只有一个元素时，说明找到了目标位置
        tree[node] = val;
    } else {
        // 非叶子节点，需要继续查找目标位置
        int mid = (start + end) / 2;
        if (index <= mid) {
            // 要更新的索引在左子树中
            // 目标索引在左半区间[start, mid]内
            updateHelper(2 * node + 1, start, mid, index, val);
        } else {
            // 要更新的索引在右子树中
            // 目标索引在右半区间[mid+1, end]内
            updateHelper(2 * node + 2, mid + 1, end, index, val);
        }
        // 更新父节点的值（区间和）
        // 子节点更新后，需要更新当前节点的值
        tree[node] = tree[2 * node + 1] + tree[2 * node + 2];
    }
}

/**
 * 查询区间和的辅助函数
 * 递归查询指定区间内的元素和
 * 优化策略：
 * 1. 如果当前区间与查询区间无交集，返回 0
 * 2. 如果当前区间完全包含在查询区间内，直接返回当前节点值
 * 3. 否则递归查询左右子树
 * @param node 当前线段树节点索引
```

```
     * @param start 当前节点表示区间的起始位置
     * @param end 当前节点表示区间的结束位置
     * @param left 查询区间左边界
     * @param right 查询区间右边界
     * @return 区间[left, right]与当前节点区间交集内元素的和
     */
    int sumRangeHelper(int node, int start, int end, int left, int right) {
        // 优化1：如果查询区间与当前区间无交集，返回0
        // 查询区间在当前区间左侧或右侧
        if (right < start || end < left) {
            return 0;
        }
        // 优化2：如果当前区间完全包含在查询区间内，直接返回当前节点值
        // 这是线段树查询的关键优化点
        if (left <= start && end <= right) {
            return tree[node];
        }
        // 部分重叠，递归查询左右子树
        // 当前区间与查询区间部分重叠，需要进一步细分
        int mid = (start + end) / 2;
        // 递归查询左子树和右子树，返回结果之和
        return sumRangeHelper(2 * node + 1, start, mid, left, right) +
               sumRangeHelper(2 * node + 2, mid + 1, end, left, right);
    }

public:
    /**
     * 构造函数，用给定数组构建线段树
     * 线段树是一种完全二叉树，可以用数组来存储
     * 对于节点i，其左子节点为2*i+1，右子节点为2*i+2
     * @param nums 原始数组
     */
    SegmentTree(vector<int>& nums) {
        n = nums.size();
        // 线段树通常需要4倍空间，以确保足够的节点存储
        // 对于n个元素的数组，线段树最多需要2*n-1个节点，但为了简化计算通常使用4*n
        tree.resize(n * 4);
        buildTree(nums, 0, 0, n - 1);
    }

    /**
     * 更新数组中某个位置的值
     * 通过递归找到对应的叶子节点并更新，然后逐层向上更新父节点
```

```
     * @param index 要更新的数组索引
     * @param val 新的值
     *
     * 时间复杂度: O(log n)
     */
    void update(int index, int val) {
        // 调用辅助函数进行更新操作
        // 从根节点(索引 0)开始，区间为[0, n-1]
        updateHelper(0, 0, n - 1, index, val);
    }

    /**
     * 查询区间和
     * 通过递归查询指定区间内的元素和
     * @param left 查询区间左边界
     * @param right 查询区间右边界
     * @return 区间[left, right]内元素的和
     *
     * 时间复杂度: O(log n)
     */
    int sumRange(int left, int right) {
        // 调用辅助函数进行查询操作
        // 从根节点(索引 0)开始，区间为[0, n-1]
        return sumRangeHelper(0, 0, n - 1, left, right);
    }
};

// NumArray 类实现
class NumArray {
private:
    SegmentTree* st;  // 线段树指针

public:
    /**
     * 构造函数，用整数数组 nums 初始化对象
     * 创建线段树实例并用初始数组构建线段树
     * @param nums 初始数组
     *
     * 时间复杂度: O(n)
     * 空间复杂度: O(n)
     */
    NumArray(vector<int>& nums) {
        // 创建线段树实例
```

```
        // 线段树将负责维护数组的区间和信息
        st = new SegmentTree(nums);
    }

    /**
     * 将 nums[index] 的值更新为 val
     * 通过线段树更新指定位置的值
     * @param index 要更新的数组索引
     * @param val 新的值
     *
     * 时间复杂度: O(log n)
     */
    void update(int index, int val) {
        // 调用线段树的更新方法
        // 线段树会找到对应的叶子节点并更新，然后逐层向上更新父节点
        st->update(index, val);
    }

    /**
     * 返回数组 nums 中索引 left 和索引 right 之间（包含）的 nums 元素的和
     * 通过线段树查询指定区间的和
     * @param left 查询区间左边界
     * @param right 查询区间右边界
     * @return 区间和
     *
     * 时间复杂度: O(log n)
     */
    int sumRange(int left, int right) {
        // 调用线段树的查询方法
        // 线段树会递归查询指定区间的和
        return st->sumRange(left, right);
    }

    // 析构函数，释放动态分配的内存
    ~NumArray() {
        delete st;
    }
};

// 测试方法
int main() {
    // 测试用例:
    // ["NumArray", "sumRange", "update", "sumRange"]
```

```
    // [[[1, 3, 5]], [0, 2], [1, 2], [0, 2]]
    // 输出:
    // [null, 9, null, 8]

    vector<int> nums = {1, 3, 5};
    NumArray* numArray = new NumArray(nums);
    cout << numArray->sumRange(0, 2) << endl; // 应该输出 9 (1+3+5)
    numArray->update(1, 2);   // nums = [1,2,5]
    cout << numArray->sumRange(0, 2) << endl; // 应该输出 8 (1+2+5)

    delete numArray;
    return 0;
}

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

文件: Code05_RangeSumQueryMutable.java
==========================================
package class112;

// 307. 区域和检索 - 数组可修改 - 线段树实现
// 题目来源：LeetCode 307 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。
// 其中一类查询要求更新数组 nums 下标对应的值
// 另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：
// 使用线段树来高效处理单点更新和区间求和操作
// 1. 线段树是一种二叉树结构，每个节点代表一个区间
// 2. 叶子节点代表数组中的单个元素
// 3. 非叶子节点代表其子节点区间的合并结果（这里是区间和）
//
// 核心思想：
// 线段树的特性：
// 1. 树状结构：线段树是一棵完全二叉树，便于用数组存储
```

```
// 2. 区间划分：每个节点代表一个区间，根节点代表整个区间[0, n-1]
// 3. 递归构建：非叶子节点的值等于其左右子节点值的和
// 4. 高效操作：单点更新和区间查询的时间复杂度均为O(log n)
//
// 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)
//
// 请同学们务必参考如下代码中关于输入、输出的处理
// 这是输入输出处理效率很高的写法
// 提交以下的code，提交时请把类名改成"Main"，可以直接通过

import java.util.*;

public class Code05_RangeSumQueryMutable {

    // 线段树实现
    static class SegmentTree {
        int[] tree;  // 线段树数组，存储区间和
        int n;       // 原数组长度

        /**
         * 构造函数，用给定数组构建线段树
         * 线段树是一种完全二叉树，可以用数组来存储
         * 对于节点i，其左子节点为2*i+1，右子节点为2*i+2
         * @param nums 原始数组
         */
        public SegmentTree(int[] nums) {
            n = nums.length;
            // 线段树通常需要4倍空间，以确保足够的节点存储
            // 对于n个元素的数组，线段树最多需要2*n-1个节点，但为了简化计算通常使用4*n
            tree = new int[n * 4];
            buildTree(nums, 0, 0, n - 1);
        }

        /**
         * 构建线段树
         * 采用递归方式构建线段树，每个节点维护一个区间的信息
         * 叶子节点对应数组中的单个元素，非叶子节点对应区间的合并结果
         * @param nums 原始数组
         * @param node 当前线段树节点索引
```

```
     * @param start 当前节点表示区间的起始位置
     * @param end 当前节点表示区间的结束位置
     *
     * 时间复杂度: O(n)
     * 空间复杂度: O(n)
     */
    private void buildTree(int[] nums, int node, int start, int end) {
        if (start == end) {
            // 叶子节点，存储数组元素值
            // 叶子节点对应原数组中的一个具体元素
            tree[node] = nums[start];
        } else {
            // 非叶子节点，需要递归构建左右子树
            int mid = (start + end) / 2;
            // 递归构建左子树
            // 左子节点索引为 2*node+1，表示区间[start, mid]
            buildTree(nums, 2 * node + 1, start, mid);
            // 递归构建右子树
            // 右子节点索引为 2*node+2，表示区间[mid+1, end]
            buildTree(nums, 2 * node + 2, mid + 1, end);
            // 合并左右子树的结果，存储区间和
            // 非叶子节点的值等于其左右子节点值的和
            tree[node] = tree[2 * node + 1] + tree[2 * node + 2];
        }
    }

    /**
     * 更新数组中某个位置的值
     * 通过递归找到对应的叶子节点并更新，然后逐层向上更新父节点
     * @param index 要更新的数组索引
     * @param val 新的值
     *
     * 时间复杂度: O(log n)
     */
    public void update(int index, int val) {
        // 调用辅助函数进行更新操作
        // 从根节点(索引 0)开始，区间为[0, n-1]
        updateHelper(0, 0, n - 1, index, val);
    }

    /**
     * 更新线段树中某个位置的值的辅助函数
     * 递归查找目标位置并更新，然后逐层向上更新父节点
```

```
     * @param node 当前线段树节点索引
     * @param start 当前节点表示区间的起始位置
     * @param end 当前节点表示区间的结束位置
     * @param index 要更新的数组索引
     * @param val 新的值
     */
    private void updateHelper(int node, int start, int end, int index, int val) {
        if (start == end) {
            // 找到叶子节点，更新值
            // 当区间只有一个元素时，说明找到了目标位置
            tree[node] = val;
        } else {
            // 非叶子节点，需要继续查找目标位置
            int mid = (start + end) / 2;
            if (index <= mid) {
                // 要更新的索引在左子树中
                // 目标索引在左半区间[start, mid]内
                updateHelper(2 * node + 1, start, mid, index, val);
            } else {
                // 要更新的索引在右子树中
                // 目标索引在右半区间[mid+1, end]内
                updateHelper(2 * node + 2, mid + 1, end, index, val);
            }
            // 更新父节点的值（区间和）
            // 子节点更新后，需要更新当前节点的值
            tree[node] = tree[2 * node + 1] + tree[2 * node + 2];
        }
    }

    /**
     * 查询区间和
     * 通过递归查询指定区间内的元素和
     * @param left 查询区间左边界
     * @param right 查询区间右边界
     * @return 区间[left, right]内元素的和
     *
     * 时间复杂度: O(log n)
     */
    public int sumRange(int left, int right) {
        // 调用辅助函数进行查询操作
        // 从根节点(索引0)开始，区间为[0, n-1]
        return sumRangeHelper(0, 0, n - 1, left, right);
    }
```

```
    /**
     * 查询区间和的辅助函数
     * 递归查询指定区间内的元素和
     * 优化策略：
     * 1. 如果当前区间与查询区间无交集，返回 0
     * 2. 如果当前区间完全包含在查询区间内，直接返回当前节点值
     * 3. 否则递归查询左右子树
     * @param node 当前线段树节点索引
     * @param start 当前节点表示区间的起始位置
     * @param end 当前节点表示区间的结束位置
     * @param left 查询区间左边界
     * @param right 查询区间右边界
     * @return 区间[left, right]与当前节点区间交集内元素的和
     */
    private int sumRangeHelper(int node, int start, int end, int left, int right) {
        // 优化 1：如果查询区间与当前区间无交集，返回 0
        // 查询区间在当前区间左侧或右侧
        if (right < start || end < left) {
            return 0;
        }
        // 优化 2：如果当前区间完全包含在查询区间内，直接返回当前节点值
        // 这是线段树查询的关键优化点
        if (left <= start && end <= right) {
            return tree[node];
        }
        // 部分重叠，递归查询左右子树
        // 当前区间与查询区间部分重叠，需要进一步细分
        int mid = (start + end) / 2;
        // 递归查询左子树和右子树，返回结果之和
        return sumRangeHelper(2 * node + 1, start, mid, left, right) +
               sumRangeHelper(2 * node + 2, mid + 1, end, left, right);
    }
}

SegmentTree st;  // 线段树实例

/**
 * 构造函数，用整数数组 nums 初始化对象
 * 创建线段树实例并用初始数组构建线段树
 * @param nums 初始数组
 *
 * 时间复杂度: O(n)
```

```
     * 空间复杂度: O(n)
     */
    public Code05_RangeSumQueryMutable(int[] nums) {
        // 创建线段树实例
        // 线段树将负责维护数组的区间和信息
        st = new SegmentTree(nums);
    }

    /**
     * 将 nums[index] 的值更新为 val
     * 通过线段树更新指定位置的值
     * @param index 要更新的数组索引
     * @param val 新的值
     *
     * 时间复杂度: O(log n)
     */
    public void update(int index, int val) {
        // 调用线段树的更新方法
        // 线段树会找到对应的叶子节点并更新，然后逐层向上更新父节点
        st.update(index, val);
    }

    /**
     * 返回数组 nums 中索引 left 和索引 right 之间（包含）的 nums 元素的和
     * 通过线段树查询指定区间的和
     * @param left 查询区间左边界
     * @param right 查询区间右边界
     * @return 区间和
     *
     * 时间复杂度: O(log n)
     */
    public int sumRange(int left, int right) {
        // 调用线段树的查询方法
        // 线段树会递归查询指定区间的和
        return st.sumRange(left, right);
    }

    /**
     * 测试方法
     */
    public static void main(String[] args) {
        // 测试用例:
        // ["NumArray", "sumRange", "update", "sumRange"]
```

```
        // [[[1, 3, 5]], [0, 2], [1, 2], [0, 2]]
        // 输出:
        // [null, 9, null, 8]

        int[] nums = {1, 3, 5};
        Code05_RangeSumQueryMutable numArray = new Code05_RangeSumQueryMutable(nums);
        System.out.println(numArray.sumRange(0, 2)); // 应该输出 9 (1+3+5)
        numArray.update(1, 2);   // nums = [1,2,5]
        System.out.println(numArray.sumRange(0, 2)); // 应该输出 8 (1+2+5)
    }
}
```

==========

文件: Code05_RangeSumQueryMutable.py

==========

```
# 307. 区域和检索 - 数组可修改 - 线段树实现
# 题目来源：LeetCode 307 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。
# 其中一类查询要求更新数组 nums 下标对应的值
# 另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：
# 使用线段树来高效处理单点更新和区间求和操作
# 1. 线段树是一种二叉树结构，每个节点代表一个区间
# 2. 叶子节点代表数组中的单个元素
# 3. 非叶子节点代表其子节点区间的合并结果（这里是区间和）
#
# 核心思想：
# 线段树的特性：
# 1. 树状结构：线段树是一棵完全二叉树，便于用数组存储
# 2. 区间划分：每个节点代表一个区间，根节点代表整个区间[0, n-1]
# 3. 递归构建：非叶子节点的值等于其左右子节点值的和
# 4. 高效操作：单点更新和区间查询的时间复杂度均为O(log n)
#
```

```
# 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)
#
# 请同学们务必参考如下代码中关于输入、输出的处理
# 这是输入输出处理效率很高的写法
# 提交以下的 code，提交时请把类名改成"Main"，可以直接通过

class SegmentTree:
    def __init__(self, nums):
        """
        线段树实现
        线段树是一种完全二叉树，可以用数组来存储
        对于节点 i，其左子节点为 2*i+1，右子节点为 2*i+2
        :param nums: 原始数组

        时间复杂度: O(n)
        空间复杂度: O(n)
        """
        self.n = len(nums)
        # 线段树通常需要 4 倍空间，以确保足够的节点存储
        # 对于 n 个元素的数组，线段树最多需要 2*n-1 个节点，但为了简化计算通常使用 4*n
        self.tree = [0] * (self.n * 4)
        self.build_tree(nums, 0, 0, self.n - 1)

    def build_tree(self, nums, node, start, end):
        """
        构建线段树
        采用递归方式构建线段树，每个节点维护一个区间的信息
        叶子节点对应数组中的单个元素，非叶子节点对应区间的合并结果
        :param nums: 原始数组
        :param node: 当前线段树节点索引
        :param start: 当前节点表示区间的起始位置
        :param end: 当前节点表示区间的结束位置

        时间复杂度: O(n)
        空间复杂度: O(log n) - 递归栈空间
        """
        if start == end:
            # 叶子节点，存储数组元素值
            # 叶子节点对应原数组中的一个具体元素
```

```
            self.tree[node] = nums[start]
        else:
            # 非叶子节点，需要递归构建左右子树
            mid = (start + end) // 2
            # 递归构建左子树
            # 左子节点索引为2*node+1，表示区间[start, mid]
            self.build_tree(nums, 2 * node + 1, start, mid)
            # 递归构建右子树
            # 右子节点索引为2*node+2，表示区间[mid+1, end]
            self.build_tree(nums, 2 * node + 2, mid + 1, end)
            # 合并左右子树的结果，存储区间和
            # 非叶子节点的值等于其左右子节点值的和
            self.tree[node] = self.tree[2 * node + 1] + self.tree[2 * node + 2]

    def update(self, index, val):
        """
        更新数组中某个位置的值
        通过递归找到对应的叶子节点并更新，然后逐层向上更新父节点
        :param index: 要更新的数组索引
        :param val: 新的值

        时间复杂度: O(log n)
        """
        # 调用辅助函数进行更新操作
        # 从根节点(索引0)开始，区间为[0, n-1]
        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
        递归查找目标位置并更新，然后逐层向上更新父节点
        :param node: 当前线段树节点索引
        :param start: 当前节点表示区间的起始位置
        :param end: 当前节点表示区间的结束位置
        :param index: 要更新的数组索引
        :param val: 新的值
        """
        if start == end:
            # 找到叶子节点，更新值
            # 当区间只有一个元素时，说明找到了目标位置
            self.tree[node] = val
        else:
            # 非叶子节点，需要继续查找目标位置
```

```
        mid = (start + end) // 2
        if index <= mid:
            # 要更新的索引在左子树中
            # 目标索引在左半区间[start, mid]内
            self._update_helper(2 * node + 1, start, mid, index, val)
        else:
            # 要更新的索引在右子树中
            # 目标索引在右半区间[mid+1, end]内
            self._update_helper(2 * node + 2, mid + 1, end, index, val)
        # 更新父节点的值（区间和）
        # 子节点更新后，需要更新当前节点的值
        self.tree[node] = self.tree[2 * node + 1] + self.tree[2 * node + 2]

def sum_range(self, left, right):
    """
    查询区间和
    通过递归查询指定区间内的元素和
    :param left: 查询区间左边界
    :param right: 查询区间右边界
    :return: 区间[left, right]内元素的和

    时间复杂度: O(log n)
    """
    # 调用辅助函数进行查询操作
    # 从根节点(索引 0)开始，区间为[0, n-1]
    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
    递归查询指定区间内的元素和
    优化策略：
    1. 如果当前区间与查询区间无交集，返回 0
    2. 如果当前区间完全包含在查询区间内，直接返回当前节点值
    3. 否则递归查询左右子树
    :param node: 当前线段树节点索引
    :param start: 当前节点表示区间的起始位置
    :param end: 当前节点表示区间的结束位置
    :param left: 查询区间左边界
    :param right: 查询区间右边界
    :return: 区间[left, right]与当前节点区间交集内元素的和
    """
    # 优化 1：如果查询区间与当前区间无交集，返回 0
```

```
        # 查询区间在当前区间左侧或右侧
        if right < start or end < left:
            return 0
        # 优化2：如果当前区间完全包含在查询区间内，直接返回当前节点值
        # 这是线段树查询的关键优化点
        if left <= start and end <= right:
            return self.tree[node]
        # 部分重叠，递归查询左右子树
        # 当前区间与查询区间部分重叠，需要进一步细分
        mid = (start + end) // 2
        # 递归查询左子树和右子树，返回结果之和
        return (self._sum_range_helper(2 * node + 1, start, mid, left, right) +
                self._sum_range_helper(2 * node + 2, mid + 1, end, left, right))


class NumArray:
    def __init__(self, nums):
        """
        构造函数，用整数数组 nums 初始化对象
        创建线段树实例并用初始数组构建线段树
        :param nums: 初始数组

        时间复杂度: O(n)
        空间复杂度: O(n)
        """
        # 创建线段树实例
        # 线段树将负责维护数组的区间和信息
        self.st = SegmentTree(nums)

    def update(self, index, val):
        """
        将 nums[index] 的值更新为 val
        通过线段树更新指定位置的值
        :param index: 要更新的数组索引
        :param val: 新的值

        时间复杂度: O(log n)
        """
        # 调用线段树的更新方法
        # 线段树会找到对应的叶子节点并更新，然后逐层向上更新父节点
        self.st.update(index, val)

    def sumRange(self, left, right):
```

```
        """
        返回数组 nums 中索引 left 和索引 right 之间（包含）的 nums 元素的和
        通过线段树查询指定区间的和
        :param left: 查询区间左边界
        :param right: 查询区间右边界
        :return: 区间和

        时间复杂度: O(log n)
        """
        # 调用线段树的查询方法
        # 线段树会递归查询指定区间的和
        return self.st.sum_range(left, right)


# 测试方法
if __name__ == "__main__":
    # 测试用例:
    # ["NumArray", "sumRange", "update", "sumRange"]
    # [[[1, 3, 5]], [0, 2], [1, 2], [0, 2]]
    # 输出:
    # [null, 9, null, 8]

    nums = [1, 3, 5]
    numArray = NumArray(nums)
    print(numArray.sumRange(0, 2))  # 应该输出 9 (1+3+5)
    numArray.update(1, 2)   # nums = [1,2,5]
    print(numArray.sumRange(0, 2))  # 应该输出 8 (1+2+5)
```

==========

文件: Code06_TheSkylineProblem.cpp

==========

```
// 218. 天际线问题 - 线段树 + 离散化实现
// 题目来源：LeetCode 218 https://leetcode.cn/problems/the-skyline-problem/
//
// 题目描述：
// 城市的 天际线 是从远处观看该城市中所有建筑物形成的轮廓的外部轮廓。
// 给你所有建筑物的位置和高度，请返回 由这些建筑物形成的 天际线。
// 每个建筑物的几何信息由数组 buildings 表示，其中三元组 buildings[i] = [lefti, righti, heighti] 表示：
// lefti 是第 i 座建筑物左边缘的 x 坐标。
// righti 是第 i 座建筑物右边缘的 x 坐标。
// heighti 是第 i 座建筑物的高度。
```

// 你可以假设所有的建筑都是完美的长方形，在高度为 0 的绝对平坦的表面上。
// 天际线 应该表示为由 "关键点" 组成的列表，格式 [[x1,y1],[x2,y2],...] ，并按 x 坐标 进行 排序。
// 关键点是水平线段的左端点。列表中最后一个点是最右侧建筑物的终点，y 坐标始终为 0 ，仅用于标记天际线的终点。
// 此外，任何两个相邻建筑物之间的地面都应被视为天际线轮廓的一部分。
// 注意：输出天际线中不得有连续的相同高度的水平线。
//
// 解题思路：
// 使用线段树配合离散化来解决天际线问题
// 1. 收集所有建筑物的左右边界坐标作为关键点
// 2. 对关键点进行离散化处理，建立坐标映射关系
// 3. 使用线段树维护区间最大高度，支持区间更新和单点查询
// 4. 遍历所有建筑物，将每个建筑物的高度更新到对应区间
// 5. 遍历所有离散化后的坐标点，查询高度变化的关键点
//
// 核心思想：
// 1. 离散化：由于建筑物的坐标可能很大，直接使用原始坐标会导致空间浪费。
//   通过离散化将大范围的坐标映射到较小的连续整数范围，提高效率。
// 2. 线段树：用于维护区间最大高度信息，支持高效的区间更新和查询操作。
// 3. 懒惰传播：在区间更新时使用懒惰标记，避免不必要的重复计算。
// 4. 关键点识别：通过比较相邻点的高度变化来识别天际线的关键点。
//
// 时间复杂度分析：
// - 收集关键点：O(n)
// - 离散化：O(n log n)
// - 构建线段树：O(n)
// - 处理建筑物：O(n log n)
// - 查询结果：O(n log n)
// - 总时间复杂度：O(n log n)
// 空间复杂度：O(n)

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

// 线段树实现，用于解决天际线问题
class SegmentTree {
private:
    vector<int> tree; // 存储区间最大高度
    vector<int> lazy; // 懒惰标记

```
int n;                    // 线段树维护的区间长度

/**
 * 上推操作，更新父节点信息
 * 将左右子节点的最大值更新到父节点
 * 在线段树中，父节点的值通常由子节点的值计算得出
 * 对于本问题，父节点维护的区间最大高度等于左右子节点维护区间最大高度的最大值
 *
 * 时间复杂度: O(1)
 */
void push_up(int node) {
    // 更新当前节点的最大高度为左右子节点最大高度的最大值
    tree[node] = max(tree[2 * node], tree[2 * node + 1]);
}

/**
 * 下传操作，传递懒惰标记
 * 将当前节点的懒惰标记传递给左右子节点
 * 懒惰传播是线段树优化的重要技术，用于延迟更新操作
 * 只有在真正需要访问子节点时才将更新操作传递下去，避免不必要的计算
 *
 * 时间复杂度: O(1)
 */
void push_down(int node, int ln, int rn) {
    // 只有当当前节点有懒惰标记时才需要下传
    if (lazy[node] != 0) {
        // 更新左子节点的懒惰标记和最大值
        // 将当前节点的覆盖高度传递给左子节点
        lazy[2 * node] = max(lazy[2 * node], lazy[node]);
        // 左子节点的最大高度更新为覆盖高度
        tree[2 * node] = max(tree[2 * node], lazy[node]);

        // 更新右子节点的懒惰标记和最大值
        // 将当前节点的覆盖高度传递给右子节点
        lazy[2 * node + 1] = max(lazy[2 * node + 1], lazy[node]);
        // 右子节点的最大高度更新为覆盖高度
        tree[2 * node + 1] = max(tree[2 * node + 1], lazy[node]);

        // 清除当前节点的懒惰标记
        // 标记已传递，当前节点的懒惰标记清零
        lazy[node] = 0;
    }
}
```

```
public:
    /**
     * 初始化线段树
     * 线段树是一种完全二叉树，可以用数组来存储
     * 对于节点 i，其左子节点为 2*i，右子节点为 2*i+1
     *
     * 时间复杂度: O(n)
     * 空间复杂度: O(n)
     */
    SegmentTree(int n) {
        this->n = n;
        // 存储区间最大高度
        // tree[i]表示节点 i 维护的区间的最大高度
        tree.resize(4 * n);
        // 懒惰标记
        // lazy[i]表示节点 i 维护的区间需要覆盖的高度
        lazy.resize(4 * n);
    }

    /**
     * 区间修改：将区间[jobl,jobr]的高度更新为 jobh（取最大值）
     * 利用懒惰传播优化，避免对每个元素逐一更新
     *
     * 时间复杂度: O(log n)
     */
    void update(int jobl, int jobr, int jobh, int l, int r, int node) {
        // 优化 1：如果当前节点维护的区间完全被操作区间覆盖
        if (jobl <= l && r <= jobr) {
            // 当前区间完全被操作区间覆盖，更新懒惰标记和最大值
            // 这是懒惰传播的关键：只标记不立即执行
            lazy[node] = max(lazy[node], jobh);
            tree[node] = max(tree[node], jobh);
            return;
        }

        // 计算中点，将区间分为两部分
        int mid = (l + r) / 2;
        // 下传懒惰标记
        // 在递归处理子节点之前，需要确保当前节点的懒惰标记已经传递
        push_down(node, mid - l + 1, r - mid);

        // 递归更新左子树
```

```
    // 只有当操作区间与左子树区间有交集时才继续处理
    if (jobl <= mid) {
        update(jobl, jobr, jobh, l, mid, 2 * node);
    }
    // 递归更新右子树
    // 只有当操作区间与右子树区间有交集时才继续处理
    if (jobr > mid) {
        update(jobl, jobr, jobh, mid + 1, r, 2 * node + 1);
    }

    // 上推更新父节点
    // 将子节点的更新结果合并到当前节点
    push_up(node);
}

/**
 * 查询区间最大值
 * 在查询过程中需要确保懒惰标记已经正确传递
 *
 * 时间复杂度: O(log n)
 */
int query(int jobl, int jobr, int l, int r, int node) {
    // 优化1：如果当前节点维护的区间完全包含在查询区间内
    if (jobl <= l && r <= jobr) {
        // 当前区间完全包含在查询区间内，直接返回最大值
        // 这是线段树查询的优化点：如果当前区间完全在查询区间内，直接返回结果
        return tree[node];
    }

    // 计算中点，将区间分为两部分
    int mid = (l + r) / 2;
    // 下传懒惰标记
    // 在查询时必须确保懒惰标记已经传递，以保证结果正确
    push_down(node, mid - l + 1, r - mid);

    int res = 0;
    // 递归查询左子树
    // 只有当查询区间与左子树区间有交集时才继续查询
    if (jobl <= mid) {
        res = max(res, query(jobl, jobr, l, mid, 2 * node));
    }
    // 递归查询右子树
    // 只有当查询区间与右子树区间有交集时才继续查询
```

```
        if (jobr > mid) {
            res = max(res, query(jobl, jobr, mid + 1, r, 2 * node + 1));
        }
        return res;
    }
};

class Solution {
public:
    /**
     * 计算天际线
     * 通过离散化和线段树来高效解决天际线问题
     *
     * 时间复杂度: O(n log n)
     * 空间复杂度: O(n)
     */
    vector<vector<int>> getSkyline(vector<vector<int>>& buildings) {
        // 第一步：收集所有关键 x 坐标（建筑物的左右边界）
        // 为了处理边界情况，我们还需要添加右边界前一个位置的坐标
        set<int> positions;
        for (auto& building : buildings) {
            positions.insert(building[0]);      // 左边界
            positions.insert(building[1]);      // 右边界
            positions.insert(building[1] - 1);  // 添加右端点前一个位置，用于处理边界情况
        }

        // 第二步：排序并建立离散化映射
        // 将 set 转换为 vector 并排序，然后建立坐标映射关系
        vector<int> sorted_positions(positions.begin(), positions.end());
        // 实际坐标 -> 索引
        map<int, int> pos_to_idx;
        // 索引 -> 实际坐标
        map<int, int> idx_to_pos;
        for (int i = 0; i < sorted_positions.size(); i++) {
            pos_to_idx[sorted_positions[i]] = i + 1;
            idx_to_pos[i + 1] = sorted_positions[i];
        }

        // 第三步：构建线段树
        // 初始化线段树，维护区间最大高度信息
        SegmentTree seg_tree(sorted_positions.size());

        // 第四步：处理每个建筑物，将高度更新到对应区间
```

```
        // 遍历所有建筑物，将每个建筑物的高度更新到对应的离散化区间
        for (auto& building : buildings) {
            // 将建筑物的左右边界映射到离散化后的索引
            int left_idx = pos_to_idx[building[0]];      // 左边界映射
            int right_idx = pos_to_idx[building[1] - 1]; // 右边界前一个位置映射
            // 更新区间高度，取最大值
            // 参数说明：
            // left_idx: 操作区间左边界
            // right_idx: 操作区间右边界
            // building[2]: 要更新的高度
            // 1: 当前节点维护的区间左边界
            // sorted_positions.size(): 当前节点维护的区间右边界
            // 1: 当前节点索引
            seg_tree.update(left_idx, right_idx, building[2], 1, sorted_positions.size(), 1);
        }

        // 第五步：收集结果
        // 遍历所有离散化后的坐标点，查询高度变化的关键点
        vector<vector<int>> result;
        int pre_height = 0;  // 上一个高度，初始为 0 表示地面高度

        // 遍历所有离散化后的坐标点
        for (int i = 1; i <= sorted_positions.size(); i++) {
            // 查询当前点的高度（即该点的最大建筑物高度）
            int height = seg_tree.query(i, i, 1, sorted_positions.size(), 1);  // 查询当前点的高
度
            // 获取实际坐标
            int x_pos = idx_to_pos[i];  // 获取实际坐标

            // 如果高度发生变化，则记录关键点
            // 只有当当前高度与上一个高度不同时，才记录为关键点
            if (height != pre_height) {
                result.push_back({x_pos, height});  // 添加关键点[x 坐标, y 坐标(高度)]
                pre_height = height;                // 更新上一个高度
            }
        }

        return result;
    }
};

// 测试方法
int main() {
```

```
    Solution solution;

    // 测试用例 1: buildings = [[2,9,10],[3,7,15],[5,12,12],[15,20,10],[19,24,8]]
    // 期望输出: [[2,10],[3,15],[7,12],[12,0],[15,10],[20,8],[24,0]]
    vector<vector<int>> buildings1 = {{2,9,10},{3,7,15},{5,12,12},{15,20,10},{19,24,8}};
    vector<vector<int>> result1 = solution.getSkyline(buildings1);

    cout << "测试用例 1 结果: ";
    for (auto& point : result1) {
        cout << "[" << point[0] << "," << point[1] << "] ";
    }
    cout << endl;

    // 测试用例 2: buildings = [[0,2,3],[2,5,3]]
    // 期望输出: [[0,3],[5,0]]
    vector<vector<int>> buildings2 = {{0,2,3},{2,5,3}};
    vector<vector<int>> result2 = solution.getSkyline(buildings2);

    cout << "测试用例 2 结果: ";
    for (auto& point : result2) {
        cout << "[" << point[0] << "," << point[1] << "] ";
    }
    cout << endl;

    return 0;
}

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

文件: Code06_TheSkylineProblem.java
==========================================
package class112;

// 218. 天际线问题 - 线段树 + 离散化实现
// 题目来源：LeetCode 218 https://leetcode.cn/problems/the-skyline-problem/
//
// 题目描述：
// 城市的 天际线 是从远处观看该城市中所有建筑物形成的轮廓的外部轮廓。
// 给你所有建筑物的位置和高度，请返回 由这些建筑物形成的 天际线。
// 每个建筑物的几何信息由数组 buildings 表示，其中三元组 buildings[i] = [lefti, righti, heighti]
表示：
// lefti 是第 i 座建筑物左边缘的 x 坐标。
// righti 是第 i 座建筑物右边缘的 x 坐标。
```

// heighti 是第 i 座建筑物的高度。
// 你可以假设所有的建筑都是完美的长方形，在高度为 0 的绝对平坦的表面上。
// 天际线 应该表示为由 "关键点" 组成的列表，格式 [[x1,y1],[x2,y2],...] ，并按 x 坐标 进行 排序。
// 关键点是水平线段的左端点。列表中最后一个点是最右侧建筑物的终点，y 坐标始终为 0 ，仅用于标记天际线的终点。
// 此外，任何两个相邻建筑物之间的地面都应被视为天际线轮廓的一部分。
// 注意：输出天际线中不得有连续的相同高度的水平线。
//
// 解题思路：
// 使用线段树配合离散化来解决天际线问题
// 1. 收集所有建筑物的左右边界坐标作为关键点
// 2. 对关键点进行离散化处理，建立坐标映射关系
// 3. 使用线段树维护区间最大高度，支持区间更新和单点查询
// 4. 遍历所有建筑物，将每个建筑物的高度更新到对应区间
// 5. 遍历所有离散化后的坐标点，查询高度变化的关键点
//
// 核心思想：
// 1. 离散化：由于建筑物的坐标可能很大，直接使用原始坐标会导致空间浪费。
//   通过离散化将大范围的坐标映射到较小的连续整数范围，提高效率。
// 2. 线段树：用于维护区间最大高度信息，支持高效的区间更新和查询操作。
// 3. 懒惰传播：在区间更新时使用懒惰标记，避免不必要的重复计算。
// 4. 关键点识别：通过比较相邻点的高度变化来识别天际线的关键点。
//
// 时间复杂度分析：
// - 收集关键点：O(n)
// - 离散化：O(n log n)
// - 构建线段树：O(n)
// - 处理建筑物：O(n log n)
// - 查询结果：O(n log n)
// - 总时间复杂度：O(n log n)
// 空间复杂度：O(n)
//
// 请同学们务必参考如下代码中关于输入、输出的处理
// 这是输入输出处理效率很高的写法
// 提交以下的 code，提交时请把类名改成"Main"，可以直接通过

```
import java.util.*;

public class Code06_TheSkylineProblem {

    // 使用线段树和离散化解决天际线问题
    static class Solution {
        // 线段树节点
```

```
static class Node {
    int left, right;  // 区间左右端点
    int cover;        // 覆盖情况（懒惰标记）
    int max;          // 区间最大高度

    /**
     * 构造函数，创建线段树节点
     * @param l 区间左端点
     * @param r 区间右端点
     */
    Node(int l, int r) {
        left = l;
        right = r;
    }
}

Node[] tr = new Node[400010];  // 线段树数组
int cnt = 1;                   // 节点编号计数器
Map<Integer, Integer> map = new HashMap<>();    // 离散化映射：实际坐标 -> 索引
Map<Integer, Integer> revMap = new HashMap<>(); // 反向映射：索引 -> 实际坐标
List<Integer> list = new ArrayList<>();         // 存储所有关键 x 坐标

/**
 * 上推操作，更新父节点信息
 * 将左右子节点的最大值更新到父节点
 * 在线段树中，父节点的值通常由子节点的值计算得出
 * 对于本问题，父节点维护的区间最大高度等于左右子节点维护区间最大高度的最大值
 *
 * 时间复杂度: O(1)
 */
void pushup(int u) {
    // 更新当前节点的最大高度为左右子节点最大高度的最大值
    tr[u].max = Math.max(tr[u << 1].max, tr[u << 1 | 1].max);
}

/**
 * 下传操作，传递懒惰标记
 * 将当前节点的懒惰标记传递给左右子节点
 * 懒惰传播是线段树优化的重要技术，用于延迟更新操作
 * 只有在真正需要访问子节点时才将更新操作传递下去，避免不必要的计算
 *
 * 时间复杂度: O(1)
 */
```

```
void pushdown(int u) {
    // 只有当当前节点有懒惰标记时才需要下传
    if (tr[u].cover != 0) {
        // 更新左子节点的懒惰标记和最大值
        // 将当前节点的覆盖高度传递给左子节点
        tr[u << 1].cover = tr[u].cover;
        // 左子节点的最大高度更新为覆盖高度
        tr[u << 1].max = tr[u].cover;

        // 更新右子节点的懒惰标记和最大值
        // 将当前节点的覆盖高度传递给右子节点
        tr[u << 1 | 1].cover = tr[u].cover;
        // 右子节点的最大高度更新为覆盖高度
        tr[u << 1 | 1].max = tr[u].cover;

        // 清除当前节点的懒惰标记
        // 标记已传递，当前节点的懒惰标记清零
        tr[u].cover = 0;
    }
}

/**
 * 构建线段树
 * 采用递归方式构建线段树，每个节点维护一个区间的信息
 * 叶子节点对应数组中的单个元素，非叶子节点对应区间的合并结果
 * @param u 当前节点索引
 * @param l 区间左边界
 * @param r 区间右边界
 *
 * 时间复杂度: O(n)
 */
void build(int u, int l, int r) {
    // 创建当前节点并设置其维护的区间范围
    tr[u] = new Node(l, r);
    // 叶子节点不需要继续构建子树
    if (l == r) return;
    // 计算中点，将区间分为两部分
    int mid = l + r >> 1;
    // 递归构建左子树
    build(u << 1, l, mid);
    // 递归构建右子树
    build(u << 1 | 1, mid + 1, r);
}
```

```
/**
 * 区间修改：将区间[L,R]的高度更新为 h（取最大值）
 * 利用懒惰传播优化，避免对每个元素逐一更新
 * @param u 当前节点索引
 * @param L 操作区间左边界
 * @param R 操作区间右边界
 * @param h 要更新的高度
 *
 * 时间复杂度: O(log n)
 */
void update(int u, int L, int R, int h) {
    // 优化 1：如果当前节点维护的区间完全被操作区间覆盖
    if (L <= tr[u].left && tr[u].right <= R) {
        // 当前区间完全被操作区间覆盖，更新懒惰标记和最大值
        // 这是懒惰传播的关键：只标记不立即执行
        tr[u].cover = Math.max(tr[u].cover, h);
        tr[u].max = Math.max(tr[u].max, h);
        return;
    }

    // 下传懒惰标记
    // 在递归处理子节点之前，需要确保当前节点的懒惰标记已经传递
    pushdown(u);

    // 计算中点，将区间分为两部分
    int mid = tr[u].left + tr[u].right >> 1;
    // 递归更新左子树
    // 只有当操作区间与左子树区间有交集时才继续处理
    if (L <= mid) update(u << 1, L, R, h);
    // 递归更新右子树
    // 只有当操作区间与右子树区间有交集时才继续处理
    if (R > mid) update(u << 1 | 1, L, R, h);

    // 上推更新父节点
    // 将子节点的更新结果合并到当前节点
    pushup(u);
}

/**
 * 查询区间最大值
 * 在查询过程中需要确保懒惰标记已经正确传递
 * @param u 当前节点索引
```

```
     * @param L 查询区间左边界
     * @param R 查询区间右边界
     * @return 区间[L,R]内的最大值
     *
     * 时间复杂度: O(log n)
     */
    int query(int u, int L, int R) {
        // 优化1：如果当前节点维护的区间完全包含在查询区间内
        if (L <= tr[u].left && tr[u].right <= R) {
            // 当前区间完全包含在查询区间内，直接返回最大值
            // 这是线段树查询的优化点：如果当前区间完全在查询区间内，直接返回结果
            return tr[u].max;
        }

        // 下传懒惰标记
        // 在查询时必须确保懒惰标记已经传递，以保证结果正确
        pushdown(u);

        // 计算中点，将区间分为两部分
        int mid = tr[u].left + tr[u].right >> 1;
        int res = 0;
        // 递归查询左子树
        // 只有当查询区间与左子树区间有交集时才继续查询
        if (L <= mid) res = Math.max(res, query(u << 1, L, R));
        // 递归查询右子树
        // 只有当查询区间与右子树区间有交集时才继续查询
        if (R > mid) res = Math.max(res, query(u << 1 | 1, L, R));
        return res;
    }

    /**
     * 主函数：计算天际线
     * 通过离散化和线段树来高效解决天际线问题
     * @param buildings 建筑物信息数组，每个元素为[left, right, height]
     * @return 天际线关键点列表
     *
     * 时间复杂度: O(n log n)
     * 空间复杂度: O(n)
     */
    public List<List<Integer>> getSkyline(int[][] buildings) {
        // 第一步：收集所有关键x坐标（建筑物的左右边界）
        // 为了处理边界情况，我们还需要添加右边界前一个位置的坐标
        for (int[] b : buildings) {
```

```
        list.add(b[0]);      // 左边界
        list.add(b[1]);      // 右边界
        list.add(b[1] - 1);  // 添加右端点前一个位置，用于处理边界情况
    }

    // 第二步：去重并排序
    // 使用 HashSet 去重，然后转换为 ArrayList 并排序
    list = new ArrayList<>(new HashSet<>(list));
    Collections.sort(list);

    // 第三步：建立离散化映射
    // 将实际坐标映射到连续的整数索引，便于线段树处理
    for (int i = 0; i < list.size(); i++) {
        map.put(list.get(i), i + 1);    // 实际坐标 -> 索引
        revMap.put(i + 1, list.get(i)); // 索引 -> 实际坐标
    }

    // 第四步：构建线段树
    // 初始化线段树，维护区间最大高度信息
    build(1, 1, list.size());

    // 第五步：处理每个建筑物，将高度更新到对应区间
    // 遍历所有建筑物，将每个建筑物的高度更新到对应的离散化区间
    for (int[] b : buildings) {
        // 将建筑物的左右边界映射到离散化后的索引
        int l = map.get(b[0]);      // 左边界映射
        int r = map.get(b[1] - 1);  // 右边界前一个位置映射
        // 更新区间高度，取最大值
        update(1, l, r, b[2]);      // 更新区间高度
    }

    // 第六步：收集结果
    // 遍历所有离散化后的坐标点，查询高度变化的关键点
    List<List<Integer>> res = new ArrayList<>();
    int pre = 0;  // 上一个高度，初始为 0 表示地面高度

    // 遍历所有离散化后的坐标点
    for (int i = 1; i <= list.size(); i++) {
        // 查询当前点的高度（即该点的最大建筑物高度）
        int h = query(1, i, i);  // 查询当前点的高度
        // 获取实际坐标
        int x = revMap.get(i);   // 获取实际坐标
```

```
                // 如果高度发生变化，则记录关键点
                // 只有当当前高度与上一个高度不同时，才记录为关键点
                if (h != pre) {
                    List<Integer> point = new ArrayList<>();
                    point.add(x);   // 添加 x 坐标
                    point.add(h);   // 添加 y 坐标（高度）
                    res.add(point); // 添加到结果列表
                    pre = h;        // 更新上一个高度
                }
            }

            return res;
        }
    }

    /**
     * 测试方法
     */
    public static void main(String[] args) {
        Solution solution = new Solution();

        // 测试用例: buildings = [[2,9,10],[3,7,15],[5,12,12],[15,20,10],[19,24,8]]
        // 期望输出: [[2,10],[3,15],[7,12],[12,0],[15,10],[20,8],[24,0]]
        int[][] buildings = {{2,9,10},{3,7,15},{5,12,12},{15,20,10},{19,24,8}};
        List<List<Integer>> result = solution.getSkyline(buildings);
        System.out.println(result);
    }
}

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

文件: Code06_TheSkylineProblem.py
==========================================
# 218. 天际线问题 - 线段树 + 离散化实现
# 题目来源：LeetCode 218 https://leetcode.cn/problems/the-skyline-problem/
#
# 题目描述：
# 城市的 天际线 是从远处观看该城市中所有建筑物形成的轮廓的外部轮廓。
# 给你所有建筑物的位置和高度，请返回 由这些建筑物形成的 天际线。
# 每个建筑物的几何信息由数组 buildings 表示，其中三元组 buildings[i] = [lefti, righti, heighti]
表示：
# lefti 是第 i 座建筑物左边缘的 x 坐标。
# righti 是第 i 座建筑物右边缘的 x 坐标。
```

heighti 是第 i 座建筑物的高度。
你可以假设所有的建筑都是完美的长方形，在高度为 0 的绝对平坦的表面上。
天际线 应该表示为由 "关键点" 组成的列表，格式 [[x1,y1],[x2,y2],...] ，并按 x 坐标 进行 排序。
关键点是水平线段的左端点。列表中最后一个点是最右侧建筑物的终点，y 坐标始终为 0 ，仅用于标记天际线的终点。
此外，任何两个相邻建筑物之间的地面都应被视为天际线轮廓的一部分。
注意：输出天际线中不得有连续的相同高度的水平线。
#
解题思路：
使用线段树配合离散化来解决天际线问题
1. 收集所有建筑物的左右边界坐标作为关键点
2. 对关键点进行离散化处理，建立坐标映射关系
3. 使用线段树维护区间最大高度，支持区间更新和单点查询
4. 遍历所有建筑物，将每个建筑物的高度更新到对应区间
5. 遍历所有离散化后的坐标点，查询高度变化的关键点
#
核心思想：
1. 离散化：由于建筑物的坐标可能很大，直接使用原始坐标会导致空间浪费。
通过离散化将大范围的坐标映射到较小的连续整数范围，提高效率。
2. 线段树：用于维护区间最大高度信息，支持高效的区间更新和查询操作。
3. 懒惰传播：在区间更新时使用懒惰标记，避免不必要的重复计算。
4. 关键点识别：通过比较相邻点的高度变化来识别天际线的关键点。
#
时间复杂度分析：
- 收集关键点：O(n)
- 离散化：O(n log n)
- 构建线段树：O(n)
- 处理建筑物：O(n log n)
- 查询结果：O(n log n)
- 总时间复杂度：O(n log n)
空间复杂度：O(n)
#
请同学们务必参考如下代码中关于输入、输出的处理
这是输入输出处理效率很高的写法
提交以下的 code，提交时请把类名改成"Main"，可以直接通过

```
from typing import List

class SegmentTree:
    """
    线段树实现，用于解决天际线问题
    """
    def __init__(self, n):
```

```
    """
    初始化线段树
    线段树是一种完全二叉树，可以用数组来存储
    对于节点 i，其左子节点为 2*i，右子节点为 2*i+1
    :param n: 线段树维护的区间长度

    时间复杂度: O(n)
    空间复杂度: O(n)
    """
    self.n = n
    # 存储区间最大高度
    # tree[i]表示节点 i 维护的区间的最大高度
    self.tree = [0] * (4 * n)
    # 懒惰标记
    # lazy[i]表示节点 i 维护的区间需要覆盖的高度
    self.lazy = [0] * (4 * n)

def push_up(self, node):
    """
    上推操作，更新父节点信息
    将左右子节点的最大值更新到父节点
    在线段树中，父节点的值通常由子节点的值计算得出
    对于本问题，父节点维护的区间最大高度等于左右子节点维护区间最大高度的最大值
    :param node: 当前节点索引

    时间复杂度: O(1)
    """
    # 更新当前节点的最大高度为左右子节点最大高度的最大值
    self.tree[node] = max(self.tree[2 * node], self.tree[2 * node + 1])

def push_down(self, node, ln, rn):
    """
    下传操作，传递懒惰标记
    将当前节点的懒惰标记传递给左右子节点
    懒惰传播是线段树优化的重要技术，用于延迟更新操作
    只有在真正需要访问子节点时才将更新操作传递下去，避免不必要的计算
    :param node: 当前节点索引
    :param ln: 左子树节点数量
    :param rn: 右子树节点数量

    时间复杂度: O(1)
    """
    # 只有当当前节点有懒惰标记时才需要下传
```

```
    if self.lazy[node] != 0:
        # 更新左子节点的懒惰标记和最大值
        # 将当前节点的覆盖高度传递给左子节点
        self.lazy[2 * node] = max(self.lazy[2 * node], self.lazy[node])
        # 左子节点的最大高度更新为覆盖高度
        self.tree[2 * node] = max(self.tree[2 * node], self.lazy[node])

        # 更新右子节点的懒惰标记和最大值
        # 将当前节点的覆盖高度传递给右子节点
        self.lazy[2 * node + 1] = max(self.lazy[2 * node + 1], self.lazy[node])
        # 右子节点的最大高度更新为覆盖高度
        self.tree[2 * node + 1] = max(self.tree[2 * node + 1], self.lazy[node])

        # 清除当前节点的懒惰标记
        # 标记已传递，当前节点的懒惰标记清零
        self.lazy[node] = 0

def update(self, jobl, jobr, jobh, l, r, node):
    """
    区间修改：将区间[jobl,jobr]的高度更新为 jobh（取最大值）
    利用懒惰传播优化，避免对每个元素逐一更新
    :param jobl: 操作区间左边界
    :param jobr: 操作区间右边界
    :param jobh: 要更新的高度
    :param l: 当前节点维护的区间左边界
    :param r: 当前节点维护的区间右边界
    :param node: 当前节点索引

    时间复杂度: O(log n)
    """
    # 优化 1：如果当前节点维护的区间完全被操作区间覆盖
    if jobl <= l and r <= jobr:
        # 当前区间完全被操作区间覆盖，更新懒惰标记和最大值
        # 这是懒惰传播的关键：只标记不立即执行
        self.lazy[node] = max(self.lazy[node], jobh)
        self.tree[node] = max(self.tree[node], jobh)
        return

    # 计算中点，将区间分为两部分
    mid = (l + r) // 2
    # 下传懒惰标记
    # 在递归处理子节点之前，需要确保当前节点的懒惰标记已经传递
    self.push_down(node, mid - l + 1, r - mid)
```

```
        # 递归更新左子树
        # 只有当操作区间与左子树区间有交集时才继续处理
        if jobl <= mid:
            self.update(jobl, jobr, jobh, l, mid, 2 * node)
        # 递归更新右子树
        # 只有当操作区间与右子树区间有交集时才继续处理
        if jobr > mid:
            self.update(jobl, jobr, jobh, mid + 1, r, 2 * node + 1)

        # 上推更新父节点
        # 将子节点的更新结果合并到当前节点
        self.push_up(node)

    def query(self, jobl, jobr, l, r, node):
        """
        查询区间最大值
        在查询过程中需要确保懒惰标记已经正确传递
        :param jobl: 查询区间左边界
        :param jobr: 查询区间右边界
        :param l: 当前节点维护的区间左边界
        :param r: 当前节点维护的区间右边界
        :param node: 当前节点索引
        :return: 区间[jobl,jobr]内的最大值

        时间复杂度: O(log n)
        """
        # 优化1：如果当前节点维护的区间完全包含在查询区间内
        if jobl <= l and r <= jobr:
            # 当前区间完全包含在查询区间内，直接返回最大值
            # 这是线段树查询的优化点：如果当前区间完全在查询区间内，直接返回结果
            return self.tree[node]

        # 计算中点，将区间分为两部分
        mid = (l + r) // 2
        # 下传懒惰标记
        # 在查询时必须确保懒惰标记已经传递，以保证结果正确
        self.push_down(node, mid - l + 1, r - mid)

        res = 0
        # 递归查询左子树
        # 只有当查询区间与左子树区间有交集时才继续查询
        if jobl <= mid:
```

```
            res = max(res, self.query(jobl, jobr, l, mid, 2 * node))
        # 递归查询右子树
        # 只有当查询区间与右子树区间有交集时才继续查询
        if jobr > mid:
            res = max(res, self.query(jobl, jobr, mid + 1, r, 2 * node + 1))
        return res


class Solution:
    def getSkyline(self, buildings: List[List[int]]) -> List[List[int]]:
        """
        计算天际线
        通过离散化和线段树来高效解决天际线问题
        :param buildings: 建筑物信息数组，每个元素为[left, right, height]
        :return: 天际线关键点列表

        时间复杂度: O(n log n)
        空间复杂度: O(n)
        """
        # 第一步：收集所有关键 x 坐标（建筑物的左右边界）
        # 为了处理边界情况，我们还需要添加右边界前一个位置的坐标
        positions = set()
        for building in buildings:
            positions.add(building[0])      # 左边界
            positions.add(building[1])      # 右边界
            positions.add(building[1] - 1)  # 添加右端点前一个位置，用于处理边界情况

        # 第二步：排序并建立离散化映射
        # 将 set 转换为 list 并排序，然后建立坐标映射关系
        sorted_positions = sorted(list(positions))
        # 实际坐标 -> 索引
        pos_to_idx = {pos: idx + 1 for idx, pos in enumerate(sorted_positions)}
        # 索引 -> 实际坐标
        idx_to_pos = {idx + 1: pos for idx, pos in enumerate(sorted_positions)}

        # 第三步：构建线段树
        # 初始化线段树，维护区间最大高度信息
        seg_tree = SegmentTree(len(sorted_positions))

        # 第四步：处理每个建筑物，将高度更新到对应区间
        # 遍历所有建筑物，将每个建筑物的高度更新到对应的离散化区间
        for building in buildings:
            # 将建筑物的左右边界映射到离散化后的索引
```

```
            left_idx = pos_to_idx[building[0]]      # 左边界映射
            right_idx = pos_to_idx[building[1] - 1] # 右边界前一个位置映射
            # 更新区间高度，取最大值
            # 参数说明：
            # left_idx: 操作区间左边界
            # right_idx: 操作区间右边界
            # building[2]: 要更新的高度
            # 1: 当前节点维护的区间左边界
            # len(sorted_positions): 当前节点维护的区间右边界
            # 1: 当前节点索引
            seg_tree.update(left_idx, right_idx, building[2], 1, len(sorted_positions), 1)

        # 第五步：收集结果
        # 遍历所有离散化后的坐标点，查询高度变化的关键点
        result = []
        pre_height = 0  # 上一个高度，初始为 0 表示地面高度

        # 遍历所有离散化后的坐标点
        for i in range(1, len(sorted_positions) + 1):
            # 查询当前点的高度（即该点的最大建筑物高度）
            height = seg_tree.query(i, i, 1, len(sorted_positions), 1)  # 查询当前点的高度
            # 获取实际坐标
            x_pos = idx_to_pos[i]  # 获取实际坐标

            # 如果高度发生变化，则记录关键点
            # 只有当当前高度与上一个高度不同时，才记录为关键点
            if height != pre_height:
                result.append([x_pos, height])  # 添加关键点[x 坐标, y 坐标(高度)]
                pre_height = height             # 更新上一个高度

        return result


# 测试方法
if __name__ == "__main__":
    solution = Solution()

    # 测试用例: buildings = [[2,9,10],[3,7,15],[5,12,12],[15,20,10],[19,24,8]]
    # 期望输出: [[2,10],[3,15],[7,12],[12,0],[15,10],[20,8],[24,0]]
    buildings = [[2,9,10],[3,7,15],[5,12,12],[15,20,10],[19,24,8]]
    result = solution.getSkyline(buildings)
    print(result)
```

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

文件: Code07_FallingSquares.cpp
==========================================
// 699. 掉落的方块 - 线段树实现
// 题目来源：LeetCode 699 https://leetcode.cn/problems/falling-squares/
//
// 题目描述：
// 在二维平面上的 x 轴上，放置着一些方块。
// 给你一个二维整数数组 positions ，其中 positions[i] = [lefti, sideLengthi] 表示：
// 第 i 个方块边长为 sideLengthi ，其左侧边与 x 轴上坐标点 lefti 对齐。
// 每个方块都从一个比目前所有的落地方块更高的高度掉落而下。
// 方块沿 y 轴负方向下落，直到着陆到 另一个正方形的顶边 或者是 x 轴上。
// 一个方块仅仅是擦过另一个方块的左侧边或右侧边不算着陆。
// 一旦着陆，它就会固定在原地，无法移动。
// 在每个方块掉落后，你必须记录目前所有已经落稳的 方块堆叠的最高高度。
// 返回一个整数数组 ans ，其中 ans[i] 表示在第 i 块方块掉落后堆叠的最高高度。
//
// 解题思路：
// 使用线段树配合离散化来解决掉落方块问题
// 1. 收集所有方块的左右边界坐标作为关键点
// 2. 对关键点进行离散化处理，建立坐标映射关系
// 3. 使用线段树维护区间最大高度，支持区间更新和区间查询
// 4. 对于每个掉落的方块，先查询其底部区间当前的最大高度，然后将整个区间更新为新高度
// 5. 记录每次掉落后的全局最大高度
//
// 核心思想：
// 1. 离散化：由于方块的坐标可能很大，直接使用原始坐标会导致空间浪费。
//    通过离散化将大范围的坐标映射到较小的连续整数范围，提高效率。
// 2. 线段树：用于维护区间最大高度信息，支持高效的区间更新和查询操作。
// 3. 懒惰传播：在区间更新时使用懒惰标记，避免不必要的重复计算。
// 4. 着陆高度计算：新方块的着陆高度等于其底部区间当前的最大高度。
// 5. 堆叠高度计算：新方块的堆叠高度 = 着陆高度 + 方块高度。
//
// 时间复杂度分析：
// - 收集关键点：O(n)
// - 离散化：O(n log n)
// - 构建线段树：O(n)
// - 处理方块：O(n log n)
// - 总时间复杂度：O(n log n)
// 空间复杂度：O(n)

#include <vector>
```

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

// 线段树实现，用于解决掉落的方块问题
class SegmentTree {
private:
    vector<int> tree;  // 存储区间最大高度
    vector<int> lazy;  // 懒惰标记：区间覆盖的高度
    int n;             // 线段树维护的区间长度

    /**
     * 上推操作，更新父节点信息
     * 将左右子节点的最大值更新到父节点
     * 在线段树中，父节点的值通常由子节点的值计算得出
     * 对于本问题，父节点维护的区间最大高度等于左右子节点维护区间最大高度的最大值
     *
     * 时间复杂度: O(1)
     */
    void push_up(int node) {
        // 更新当前节点的最大高度为左右子节点最大高度的最大值
        tree[node] = max(tree[2 * node], tree[2 * node + 1]);
    }

    /**
     * 下传操作，传递懒惰标记
     * 将当前节点的懒惰标记传递给左右子节点
     * 懒惰传播是线段树优化的重要技术，用于延迟更新操作
     * 只有在真正需要访问子节点时才将更新操作传递下去，避免不必要的计算
     *
     * 时间复杂度: O(1)
     */
    void push_down(int node, int ln, int rn) {
        // 只有当当前节点有懒惰标记时才需要下传
        if (lazy[node] != 0) {
            // 更新左子节点的懒惰标记和最大值
            // 将当前节点的覆盖高度传递给左子节点
            lazy[2 * node] = lazy[node];
            // 左子节点的最大高度更新为覆盖高度
            tree[2 * node] = lazy[node];
```

```
            // 更新右子节点的懒惰标记和最大值
            // 将当前节点的覆盖高度传递给右子节点
            lazy[2 * node + 1] = lazy[node];
            // 右子节点的最大高度更新为覆盖高度
            tree[2 * node + 1] = lazy[node];

            // 清除当前节点的懒惰标记
            // 标记已传递，当前节点的懒惰标记清零
            lazy[node] = 0;
        }
    }

public:
    /**
     * 初始化线段树
     * 线段树是一种完全二叉树，可以用数组来存储
     * 对于节点 i，其左子节点为 2*i，右子节点为 2*i+1
     *
     * 时间复杂度: O(n)
     * 空间复杂度: O(n)
     */
    SegmentTree(int n) {
        this->n = n;
        // 存储区间最大高度
        // tree[i]表示节点 i 维护的区间的最大高度
        tree.resize(4 * n);
        // 懒惰标记：区间覆盖的高度
        // lazy[i]表示节点 i 维护的区间需要覆盖的高度
        lazy.resize(4 * n);
    }

    /**
     * 区间修改：将区间[jobl,jobr]的高度更新为 jobh（覆盖）
     * 利用懒惰传播优化，避免对每个元素逐一更新
     *
     * 时间复杂度: O(log n)
     */
    void update(int jobl, int jobr, int jobh, int l, int r, int node) {
        // 优化 1：如果当前节点维护的区间完全被操作区间覆盖
        if (jobl <= l && r <= jobr) {
            // 当前区间完全被操作区间覆盖，更新懒惰标记和最大值
            // 这是懒惰传播的关键：只标记不立即执行
            lazy[node] = jobh;
```

```
        tree[node] = jobh;
        return;
    }

    // 计算中点，将区间分为两部分
    int mid = (l + r) / 2;
    // 下传懒惰标记
    // 在递归处理子节点之前，需要确保当前节点的懒惰标记已经传递
    push_down(node, mid - l + 1, r - mid);

    // 递归更新左子树
    // 只有当操作区间与左子树区间有交集时才继续处理
    if (jobl <= mid) {
        update(jobl, jobr, jobh, l, mid, 2 * node);
    }
    // 递归更新右子树
    // 只有当操作区间与右子树区间有交集时才继续处理
    if (jobr > mid) {
        update(jobl, jobr, jobh, mid + 1, r, 2 * node + 1);
    }

    // 上推更新父节点
    // 将子节点的更新结果合并到当前节点
    push_up(node);
}

/**
 * 查询区间最大值
 * 在查询过程中需要确保懒惰标记已经正确传递
 *
 * 时间复杂度: O(log n)
 */
int query(int jobl, int jobr, int l, int r, int node) {
    // 优化 1：如果当前节点维护的区间完全包含在查询区间内
    if (jobl <= l && r <= jobr) {
        // 当前区间完全包含在查询区间内，直接返回最大值
        // 这是线段树查询的优化点：如果当前区间完全在查询区间内，直接返回结果
        return tree[node];
    }

    // 计算中点，将区间分为两部分
    int mid = (l + r) / 2;
    // 下传懒惰标记
```

```
        // 在查询时必须确保懒惰标记已经传递，以保证结果正确
        push_down(node, mid - l + 1, r - mid);

        int res = 0;
        // 递归查询左子树
        // 只有当查询区间与左子树区间有交集时才继续查询
        if (jobl <= mid) {
            res = max(res, query(jobl, jobr, l, mid, 2 * node));
        }
        // 递归查询右子树
        // 只有当查询区间与右子树区间有交集时才继续查询
        if (jobr > mid) {
            res = max(res, query(jobl, jobr, mid + 1, r, 2 * node + 1));
        }
        return res;
    }
};

class Solution {
public:
    /**
     * 计算掉落方块后的最大高度
     * 通过离散化和线段树来高效解决掉落方块问题
     *
     * 时间复杂度: O(n log n)
     * 空间复杂度: O(n)
     */
    vector<int> fallingSquares(vector<vector<int>>& positions) {
        // 第一步：收集所有关键 x 坐标（方块的左右边界）
        set<int> positions_set;
        for (auto& pos : positions) {
            positions_set.insert(pos[0]);              // 左边界
            positions_set.insert(pos[0] + pos[1] - 1); // 右边界
        }

        // 第二步：排序并建立离散化映射：实际坐标 -> 索引
        // 将 set 转换为 vector 并排序，然后建立坐标映射关系
        vector<int> sorted_positions(positions_set.begin(), positions_set.end());
        // 实际坐标 -> 索引
        map<int, int> pos_to_idx;
        for (int i = 0; i < sorted_positions.size(); i++) {
            pos_to_idx[sorted_positions[i]] = i + 1;
        }
```

```
        // 第三步：构建线段树
        // 初始化线段树，维护区间最大高度信息
        SegmentTree seg_tree(sorted_positions.size());

        // 第四步：处理每个方块并收集结果
        vector<int> result;
        int max_height = 0;  // 全局最大高度

        // 处理每个方块
        for (auto& pos : positions) {
            // 计算方块的左右边界
            int left = pos[0];                    // 方块左边界
            int size = pos[1];                    // 方块边长
            int right = left + size - 1;          // 方块右边界

            // 查询当前方块底部区间最大高度（即着陆高度）
            // 方块会落在其底部区间当前最大高度之上
            int current_height = seg_tree.query(pos_to_idx[left], pos_to_idx[right],
                                                1, sorted_positions.size(), 1);
            // 新的高度 = 着陆高度 + 方块高度
            int new_height = current_height + size;

            // 更新方块所在区间高度为新高度
            // 将方块覆盖的区间更新为新的堆叠高度
            seg_tree.update(pos_to_idx[left], pos_to_idx[right], new_height,
                            1, sorted_positions.size(), 1);

            // 更新全局最大高度
            // 记录到目前为止所有方块堆叠的最大高度
            max_height = max(max_height, new_height);
            result.push_back(max_height);
        }

        return result;
    }
};

// 测试方法
int main() {
    Solution solution;

    // 测试用例 1: positions = [[1,2],[2,3],[6,1]]
```

```
    // 期望输出: [2,5,5]
    // 解释：第一个方块[1,2]高度为 2，第二个方块[2,4]底部高度为 2，总高度为 5，第三个方块[6,6]底部高度为 0，总高度为 1
    vector<vector<int>> positions1 = {{1,2},{2,3},{6,1}};
    vector<int> result1 = solution.fallingSquares(positions1);

    cout << "测试用例 1 结果: ";
    for (int val : result1) {
        cout << val << " ";
    }
    cout << endl;

    // 测试用例 2: positions = [[100,100],[200,100]]
    // 期望输出: [100,100]
    // 解释：两个方块不重叠，各自高度为 100
    vector<vector<int>> positions2 = {{100,100},{200,100}};
    vector<int> result2 = solution.fallingSquares(positions2);

    cout << "测试用例 2 结果: ";
    for (int val : result2) {
        cout << val << " ";
    }
    cout << endl;

    return 0;
}

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

文件: Code07_FallingSquares.java
==========================================
package class112;

// 699. 掉落的方块 - 线段树实现
// 题目来源：LeetCode 699 https://leetcode.cn/problems/falling-squares/
//
// 题目描述：
// 在二维平面上的 x 轴上，放置着一些方块。
// 给你一个二维整数数组 positions ，其中 positions[i] = [lefti, sideLengthi] 表示：
// 第 i 个方块边长为 sideLengthi ，其左侧边与 x 轴上坐标点 lefti 对齐。
// 每个方块都从一个比目前所有的落地方块更高的高度掉落而下。
// 方块沿 y 轴负方向下落，直到着陆到 另一个正方形的顶边 或者是 x 轴上。
// 一个方块仅仅是擦过另一个方块的左侧边或右侧边不算着陆。
```

```
// 一旦着陆，它就会固定在原地，无法移动。
// 在每个方块掉落后，你必须记录目前所有已经落稳的 方块堆叠的最高高度。
// 返回一个整数数组 ans ，其中 ans[i] 表示在第 i 块方块掉落后堆叠的最高高度。
//
// 解题思路：
// 使用线段树配合离散化来解决掉落方块问题
// 1. 收集所有方块的左右边界坐标作为关键点
// 2. 对关键点进行离散化处理，建立坐标映射关系
// 3. 使用线段树维护区间最大高度，支持区间更新和区间查询
// 4. 对于每个掉落的方块，先查询其底部区间当前的最大高度，然后将整个区间更新为新高度
// 5. 记录每次掉落后的全局最大高度
//
// 核心思想：
// 1. 离散化：由于方块的坐标可能很大，直接使用原始坐标会导致空间浪费。
//    通过离散化将大范围的坐标映射到较小的连续整数范围，提高效率。
// 2. 线段树：用于维护区间最大高度信息，支持高效的区间更新和查询操作。
// 3. 懒惰传播：在区间更新时使用懒惰标记，避免不必要的重复计算。
// 4. 着陆高度计算：新方块的着陆高度等于其底部区间当前的最大高度。
// 5. 堆叠高度计算：新方块的堆叠高度 = 着陆高度 + 方块高度。
//
// 时间复杂度分析：
// - 收集关键点：O(n)
// - 离散化：O(n log n)
// - 构建线段树：O(n)
// - 处理方块：O(n log n)
// - 总时间复杂度：O(n log n)
// 空间复杂度：O(n)
//
// 请同学们务必参考如下代码中关于输入、输出的处理
// 这是输入输出处理效率很高的写法
// 提交以下的 code，提交时请把类名改成"Main"，可以直接通过

import java.util.*;

public class Code07_FallingSquares {

    // 使用线段树解决掉落的方块问题
    static class Solution {
        // 线段树节点
        static class Node {
            int left, right;     // 区间左右端点
            int max;             // 区间最大高度
            int cover;           // 懒惰标记：区间覆盖的高度
```

```
    /**
     * 构造函数，创建线段树节点
     * @param l 区间左端点
     * @param r 区间右端点
     */
    Node(int l, int r) {
        left = l;
        right = r;
    }
}

Node[] tr = new Node[400010];  // 线段树数组

/**
 * 上推操作，更新父节点信息
 * 将左右子节点的最大值更新到父节点
 * 在线段树中，父节点的值通常由子节点的值计算得出
 * 对于本问题，父节点维护的区间最大高度等于左右子节点维护区间最大高度的最大值
 *
 * 时间复杂度: O(1)
 */
void pushup(int u) {
    // 更新当前节点的最大高度为左右子节点最大高度的最大值
    tr[u].max = Math.max(tr[u << 1].max, tr[u << 1 | 1].max);
}

/**
 * 下传操作，传递懒惰标记
 * 将当前节点的懒惰标记传递给左右子节点
 * 懒惰传播是线段树优化的重要技术，用于延迟更新操作
 * 只有在真正需要访问子节点时才将更新操作传递下去，避免不必要的计算
 *
 * 时间复杂度: O(1)
 */
void pushdown(int u) {
    // 只有当当前节点有懒惰标记时才需要下传
    if (tr[u].cover != 0) {
        // 更新左子节点的懒惰标记和最大值
        // 将当前节点的覆盖高度传递给左子节点
        tr[u << 1].cover = tr[u].cover;
        // 左子节点的最大高度更新为覆盖高度
        tr[u << 1].max = tr[u].cover;
```

```
        // 更新右子节点的懒惰标记和最大值
        // 将当前节点的覆盖高度传递给右子节点
        tr[u << 1 | 1].cover = tr[u].cover;
        // 右子节点的最大高度更新为覆盖高度
        tr[u << 1 | 1].max = tr[u].cover;

        // 清除当前节点的懒惰标记
        // 标记已传递，当前节点的懒惰标记清零
        tr[u].cover = 0;
    }
}

/**
 * 构建线段树
 * 采用递归方式构建线段树，每个节点维护一个区间的信息
 * 叶子节点对应数组中的单个元素，非叶子节点对应区间的合并结果
 * @param u 当前节点索引
 * @param l 区间左边界
 * @param r 区间右边界
 *
 * 时间复杂度: O(n)
 */
void build(int u, int l, int r) {
    // 创建当前节点并设置其维护的区间范围
    tr[u] = new Node(l, r);
    // 叶子节点不需要继续构建子树
    if (l == r) return;
    // 计算中点，将区间分为两部分
    int mid = l + r >> 1;
    // 递归构建左子树
    build(u << 1, l, mid);
    // 递归构建右子树
    build(u << 1 | 1, mid + 1, r);
}

/**
 * 区间修改：将区间[L,R]的高度更新为 h（覆盖）
 * 利用懒惰传播优化，避免对每个元素逐一更新
 * @param u 当前节点索引
 * @param L 操作区间左边界
 * @param R 操作区间右边界
 * @param h 要更新的高度
```

```
 *
 * 时间复杂度: O(log n)
 */
void update(int u, int L, int R, int h) {
    // 优化 1：如果当前节点维护的区间完全被操作区间覆盖
    if (L <= tr[u].left && tr[u].right <= R) {
        // 当前区间完全被操作区间覆盖，更新懒惰标记和最大值
        // 这是懒惰传播的关键：只标记不立即执行
        tr[u].cover = h;
        tr[u].max = h;
        return;
    }

    // 下传懒惰标记
    // 在递归处理子节点之前，需要确保当前节点的懒惰标记已经传递
    pushdown(u);

    // 计算中点，将区间分为两部分
    int mid = tr[u].left + tr[u].right >> 1;
    // 递归更新左子树
    // 只有当操作区间与左子树区间有交集时才继续处理
    if (L <= mid) update(u << 1, L, R, h);
    // 递归更新右子树
    // 只有当操作区间与右子树区间有交集时才继续处理
    if (R > mid) update(u << 1 | 1, L, R, h);

    // 上推更新父节点
    // 将子节点的更新结果合并到当前节点
    pushup(u);
}

/**
 * 查询区间最大值
 * 在查询过程中需要确保懒惰标记已经正确传递
 * @param u 当前节点索引
 * @param L 查询区间左边界
 * @param R 查询区间右边界
 * @return 区间[L,R]内的最大值
 *
 * 时间复杂度: O(log n)
 */
int query(int u, int L, int R) {
    // 优化 1：如果当前节点维护的区间完全包含在查询区间内
```

```
    if (L <= tr[u].left && tr[u].right <= R) {
        // 当前区间完全包含在查询区间内，直接返回最大值
        // 这是线段树查询的优化点：如果当前区间完全在查询区间内，直接返回结果
        return tr[u].max;
    }

    // 下传懒惰标记
    // 在查询时必须确保懒惰标记已经传递，以保证结果正确
    pushdown(u);

    // 计算中点，将区间分为两部分
    int mid = tr[u].left + tr[u].right >> 1;
    int res = 0;
    // 递归查询左子树
    // 只有当查询区间与左子树区间有交集时才继续查询
    if (L <= mid) res = Math.max(res, query(u << 1, L, R));
    // 递归查询右子树
    // 只有当查询区间与右子树区间有交集时才继续查询
    if (R > mid) res = Math.max(res, query(u << 1 | 1, L, R));
    return res;
}

/**
 * 主函数：计算掉落方块后的最大高度
 * 通过离散化和线段树来高效解决掉落方块问题
 * @param positions 方块位置信息数组，每个元素为[left, sideLength]
 * @return 每个方块掉落后堆叠的最高高度列表
 *
 * 时间复杂度: O(n log n)
 * 空间复杂度: O(n)
 */
public List<Integer> fallingSquares(int[][] positions) {
    // 第一步：收集所有关键 x 坐标（方块的左右边界）
    List<Integer> list = new ArrayList<>();
    for (int[] pos : positions) {
        list.add(pos[0]);              // 左边界
        list.add(pos[0] + pos[1] - 1); // 右边界
    }

    // 第二步：去重并排序
    // 使用 HashSet 去重，然后转换为 ArrayList 并排序
    list = new ArrayList<>(new HashSet<>(list));
    Collections.sort(list);
```

```
        // 第三步：建立离散化映射：实际坐标 -> 索引
        // 将实际坐标映射到连续的整数索引，便于线段树处理
        Map<Integer, Integer> map = new HashMap<>();
        for (int i = 0; i < list.size(); i++) {
            map.put(list.get(i), i + 1);
        }

        // 第四步：构建线段树
        // 初始化线段树，维护区间最大高度信息
        build(1, 1, list.size());

        // 第五步：处理每个方块并收集结果
        List<Integer> result = new ArrayList<>();
        int maxHeight = 0;  // 全局最大高度

        // 处理每个方块
        for (int[] pos : positions) {
            // 计算方块的左右边界
            int left = pos[0];                    // 方块左边界
            int size = pos[1];                    // 方块边长
            int right = left + size - 1;          // 方块右边界

            // 查询当前方块底部区间最大高度（即着陆高度）
            // 方块会落在其底部区间当前最大高度之上
            int currentHeight = query(1, map.get(left), map.get(right));
            // 新的高度 = 着陆高度 + 方块高度
            int newHeight = currentHeight + size;

            // 更新方块所在区间高度为新高度
            // 将方块覆盖的区间更新为新的堆叠高度
            update(1, map.get(left), map.get(right), newHeight);

            // 更新全局最大高度
            // 记录到目前为止所有方块堆叠的最大高度
            maxHeight = Math.max(maxHeight, newHeight);
            result.add(maxHeight);
        }

        return result;
    }
}
```

```
    /**
     * 测试方法
     */
    public static void main(String[] args) {
        Solution solution = new Solution();

        // 测试用例 1: positions = [[1,2],[2,3],[6,1]]
        // 期望输出: [2,5,5]
        // 解释：第一个方块[1,2]高度为 2，第二个方块[2,4]底部高度为 2，总高度为 5，第三个方块[6,6]底部高度为 0，总高度为 1
        int[][] positions1 = {{1,2},{2,3},{6,1}};
        System.out.println(solution.fallingSquares(positions1));

        // 测试用例 2: positions = [[100,100],[200,100]]
        // 期望输出: [100,100]
        // 解释：两个方块不重叠，各自高度为 100
        int[][] positions2 = {{100,100},{200,100}};
        System.out.println(solution.fallingSquares(positions2));
    }
}

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

文件: Code07_FallingSquares.py
==========================================
# 699. 掉落的方块 - 线段树实现
# 题目来源：LeetCode 699 https://leetcode.cn/problems/falling-squares/
#
# 题目描述：
# 在二维平面上的 x 轴上，放置着一些方块。
# 给你一个二维整数数组 positions ，其中 positions[i] = [lefti, sideLengthi] 表示：
# 第 i 个方块边长为 sideLengthi ，其左侧边与 x 轴上坐标点 lefti 对齐。
# 每个方块都从一个比目前所有的落地方块更高的高度掉落而下。
# 方块沿 y 轴负方向下落，直到着陆到 另一个正方形的顶边 或者是 x 轴上。
# 一个方块仅仅是擦过另一个方块的左侧边或右侧边不算着陆。
# 一旦着陆，它就会固定在原地，无法移动。
# 在每个方块掉落后，你必须记录目前所有已经落稳的 方块堆叠的最高高度。
# 返回一个整数数组 ans ，其中 ans[i] 表示在第 i 块方块掉落后堆叠的最高高度。
#
# 解题思路：
# 使用线段树配合离散化来解决掉落方块问题
# 1. 收集所有方块的左右边界坐标作为关键点
# 2. 对关键点进行离散化处理，建立坐标映射关系
```

```
# 3. 使用线段树维护区间最大高度，支持区间更新和区间查询
# 4. 对于每个掉落的方块，先查询其底部区间当前的最大高度，然后将整个区间更新为新高度
# 5. 记录每次掉落后的全局最大高度
#
# 核心思想：
# 1. 离散化：由于方块的坐标可能很大，直接使用原始坐标会导致空间浪费。
#    通过离散化将大范围的坐标映射到较小的连续整数范围，提高效率。
# 2. 线段树：用于维护区间最大高度信息，支持高效的区间更新和查询操作。
# 3. 懒惰传播：在区间更新时使用懒惰标记，避免不必要的重复计算。
# 4. 着陆高度计算：新方块的着陆高度等于其底部区间当前的最大高度。
# 5. 堆叠高度计算：新方块的堆叠高度 = 着陆高度 + 方块高度。
#
# 时间复杂度分析：
# - 收集关键点：O(n)
# - 离散化：O(n log n)
# - 构建线段树：O(n)
# - 处理方块：O(n log n)
# - 总时间复杂度：O(n log n)
# 空间复杂度：O(n)
#
# 请同学们务必参考如下代码中关于输入、输出的处理
# 这是输入输出处理效率很高的写法
# 提交以下的 code，提交时请把类名改成"Main"，可以直接通过

from typing import List

class SegmentTree:
    """
    线段树实现，用于解决掉落的方块问题
    """
    def __init__(self, n):
        """
        初始化线段树
        线段树是一种完全二叉树，可以用数组来存储
        对于节点 i，其左子节点为 2*i，右子节点为 2*i+1
        :param n: 线段树维护的区间长度

        时间复杂度: O(n)
        空间复杂度: O(n)
        """
        self.n = n
        # 存储区间最大高度
        # tree[i]表示节点 i 维护的区间的最大高度
```

```
        self.tree = [0] * (4 * n)
        # 懒惰标记：区间覆盖的高度
        # lazy[i]表示节点 i 维护的区间需要覆盖的高度
        self.lazy = [0] * (4 * n)

    def push_up(self, node):
        """
        上推操作，更新父节点信息
        将左右子节点的最大值更新到父节点
        在线段树中，父节点的值通常由子节点的值计算得出
        对于本问题，父节点维护的区间最大高度等于左右子节点维护区间最大高度的最大值
        :param node: 当前节点索引

        时间复杂度: O(1)
        """
        # 更新当前节点的最大高度为左右子节点最大高度的最大值
        self.tree[node] = max(self.tree[2 * node], self.tree[2 * node + 1])

    def push_down(self, node, ln, rn):
        """
        下传操作，传递懒惰标记
        将当前节点的懒惰标记传递给左右子节点
        懒惰传播是线段树优化的重要技术，用于延迟更新操作
        只有在真正需要访问子节点时才将更新操作传递下去，避免不必要的计算
        :param node: 当前节点索引
        :param ln: 左子树节点数量
        :param rn: 右子树节点数量

        时间复杂度: O(1)
        """
        # 只有当当前节点有懒惰标记时才需要下传
        if self.lazy[node] != 0:
            # 更新左子节点的懒惰标记和最大值
            # 将当前节点的覆盖高度传递给左子节点
            self.lazy[2 * node] = self.lazy[node]
            # 左子节点的最大高度更新为覆盖高度
            self.tree[2 * node] = self.lazy[node]

            # 更新右子节点的懒惰标记和最大值
            # 将当前节点的覆盖高度传递给右子节点
            self.lazy[2 * node + 1] = self.lazy[node]
            # 右子节点的最大高度更新为覆盖高度
            self.tree[2 * node + 1] = self.lazy[node]
```

```
        # 清除当前节点的懒惰标记
        # 标记已传递，当前节点的懒惰标记清零
        self.lazy[node] = 0

def update(self, jobl, jobr, jobh, l, r, node):
    """
    区间修改：将区间[jobl,jobr]的高度更新为 jobh（覆盖）
    利用懒惰传播优化，避免对每个元素逐一更新
    :param jobl: 操作区间左边界
    :param jobr: 操作区间右边界
    :param jobh: 要更新的高度
    :param l: 当前节点维护的区间左边界
    :param r: 当前节点维护的区间右边界
    :param node: 当前节点索引

    时间复杂度: O(log n)
    """
    # 优化 1：如果当前节点维护的区间完全被操作区间覆盖
    if jobl <= l and r <= jobr:
        # 当前区间完全被操作区间覆盖，更新懒惰标记和最大值
        # 这是懒惰传播的关键：只标记不立即执行
        self.lazy[node] = jobh
        self.tree[node] = jobh
        return

    # 计算中点，将区间分为两部分
    mid = (l + r) // 2
    # 下传懒惰标记
    # 在递归处理子节点之前，需要确保当前节点的懒惰标记已经传递
    self.push_down(node, mid - l + 1, r - mid)

    # 递归更新左子树
    # 只有当操作区间与左子树区间有交集时才继续处理
    if jobl <= mid:
        self.update(jobl, jobr, jobh, l, mid, 2 * node)
    # 递归更新右子树
    # 只有当操作区间与右子树区间有交集时才继续处理
    if jobr > mid:
        self.update(jobl, jobr, jobh, mid + 1, r, 2 * node + 1)

    # 上推更新父节点
    # 将子节点的更新结果合并到当前节点
```

```
        self.push_up(node)

    def query(self, jobl, jobr, l, r, node):
        """
        查询区间最大值
        在查询过程中需要确保懒惰标记已经正确传递
        :param jobl: 查询区间左边界
        :param jobr: 查询区间右边界
        :param l: 当前节点维护的区间左边界
        :param r: 当前节点维护的区间右边界
        :param node: 当前节点索引
        :return: 区间[jobl,jobr]内的最大值

        时间复杂度: O(log n)
        """
        # 优化1：如果当前节点维护的区间完全包含在查询区间内
        if jobl <= l and r <= jobr:
            # 当前区间完全包含在查询区间内，直接返回最大值
            # 这是线段树查询的优化点：如果当前区间完全在查询区间内，直接返回结果
            return self.tree[node]

        # 计算中点，将区间分为两部分
        mid = (l + r) // 2
        # 下传懒惰标记
        # 在查询时必须确保懒惰标记已经传递，以保证结果正确
        self.push_down(node, mid - l + 1, r - mid)

        res = 0
        # 递归查询左子树
        # 只有当查询区间与左子树区间有交集时才继续查询
        if jobl <= mid:
            res = max(res, self.query(jobl, jobr, l, mid, 2 * node))
        # 递归查询右子树
        # 只有当查询区间与右子树区间有交集时才继续查询
        if jobr > mid:
            res = max(res, self.query(jobl, jobr, mid + 1, r, 2 * node + 1))
        return res


class Solution:
    def fallingSquares(self, positions: List[List[int]]) -> List[int]:
        """
        计算掉落方块后的最大高度
```

```
通过离散化和线段树来高效解决掉落方块问题
:param positions: 方块位置信息数组，每个元素为[left, sideLength]
:return: 每个方块掉落后堆叠的最高高度列表

时间复杂度: O(n log n)
空间复杂度: O(n)
"""
# 第一步：收集所有关键 x 坐标（方块的左右边界）
positions_set = set()
for pos in positions:
    positions_set.add(pos[0])              # 左边界
    positions_set.add(pos[0] + pos[1] - 1) # 右边界

# 第二步：排序并建立离散化映射：实际坐标 -> 索引
# 将 set 转换为 list 并排序，然后建立坐标映射关系
sorted_positions = sorted(list(positions_set))
# 实际坐标 -> 索引
pos_to_idx = {pos: idx + 1 for idx, pos in enumerate(sorted_positions)}

# 第三步：构建线段树
# 初始化线段树，维护区间最大高度信息
seg_tree = SegmentTree(len(sorted_positions))

# 第四步：处理每个方块并收集结果
result = []
max_height = 0  # 全局最大高度

# 处理每个方块
for pos in positions:
    # 计算方块的左右边界
    left = pos[0]                    # 方块左边界
    size = pos[1]                    # 方块边长
    right = left + size - 1          # 方块右边界

    # 查询当前方块底部区间最大高度（即着陆高度）
    # 方块会落在其底部区间当前最大高度之上
    current_height = seg_tree.query(pos_to_idx[left], pos_to_idx[right],
                                    1, len(sorted_positions), 1)
    # 新的高度 = 着陆高度 + 方块高度
    new_height = current_height + size

    # 更新方块所在区间高度为新高度
    # 将方块覆盖的区间更新为新的堆叠高度
```

```
            seg_tree.update(pos_to_idx[left], pos_to_idx[right], new_height,
                            1, len(sorted_positions), 1)

            # 更新全局最大高度
            # 记录到目前为止所有方块堆叠的最大高度
            max_height = max(max_height, new_height)
            result.append(max_height)

        return result


# 测试方法
if __name__ == "__main__":
    solution = Solution()

    # 测试用例 1: positions = [[1,2],[2,3],[6,1]]
    # 期望输出: [2,5,5]
    # 解释：第一个方块[1,2]高度为 2，第二个方块[2,4]底部高度为 2，总高度为 5，第三个方块[6,6]底部高度为 0，总高度为 1
    positions1 = [[1,2],[2,3],[6,1]]
    print(solution.fallingSquares(positions1))

    # 测试用例 2: positions = [[100,100],[200,100]]
    # 期望输出: [100,100]
    # 解释：两个方块不重叠，各自高度为 100
    positions2 = [[100,100],[200,100]]
    print(solution.fallingSquares(positions2))

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

文件: Code08_RangeSumQuery2D.java
================================================
package class112;

// 308. 二维区域和检索 - 可变 - 二维线段树实现
// 题目来源：LeetCode 308 https://leetcode.cn/problems/range-sum-query-2d-mutable/
//
// 题目描述：
// 给你一个 2D 矩阵 matrix，请你完成两类查询：
// 1. 更新矩阵中某个单元格的值
// 2. 计算子矩形范围内元素的总和，该子矩阵的左上角为 (row1, col1) ，右下角为 (row2, col2)
// 实现 NumMatrix 类：
// NumMatrix(int[][] matrix) 给定整数矩阵 matrix 进行初始化
```

```
// void update(int row, int col, int val) 更新 matrix[row][col] 的值到 val
// int sumRegion(int row1, int col1, int row2, int col2) 返回子矩阵的总和
//
// 解题思路：
// 使用二维线段树来高效处理二维矩阵的单点更新和区域求和操作
// 1. 构建二维线段树，先对每行构建一维线段树，再对列构建线段树
// 2. 更新操作时，从叶子节点向上更新所有相关节点
// 3. 查询操作时，使用类似一维线段树的区间查询方法
//
// 时间复杂度分析：
// - 构建线段树：O(m*n)
// - 单点更新：O(log m * log n)
// - 区域查询：O(log m * log n)
// 空间复杂度：O(m*n)

public class Code08_RangeSumQuery2D {

    static class NumMatrix {
        private int[][] matrix;  // 原始矩阵
        private int[][] tree;    // 二维线段树
        private int m, n;        // 矩阵的行数和列数

        /**
         * 构造函数，用给定矩阵初始化二维线段树
         * @param matrix 原始矩阵
         *
         * 时间复杂度: O(m*n)
         * 空间复杂度: O(m*n)
         */
        public NumMatrix(int[][] matrix) {
            if (matrix == null || matrix.length == 0 || matrix[0].length == 0) {
                return;
            }

            this.matrix = matrix;
            this.m = matrix.length;
            this.n = matrix[0].length;
            // 初始化线段树，大小为原矩阵的 4 倍
            this.tree = new int[m * 2][n * 2];

            // 构建二维线段树
            buildTree();
        }
```

```
/**
 * 构建二维线段树
 * 先构建每行的一维线段树，再构建列的线段树
 *
 * 时间复杂度: O(m*n)
 */
private void buildTree() {
    // 先构建每行的一维线段树
    for (int i = 0; i < m; i++) {
        // 将原始数据复制到线段树的叶子节点
        for (int j = 0; j < n; j++) {
            tree[i + m][j + n] = matrix[i][j];
        }

        // 构建行的线段树（从右到左）
        for (int j = n - 1; j > 0; j--) {
            tree[i + m][j] = tree[i + m][j << 1] + tree[i + m][j << 1 | 1];
        }
    }

    // 构建列的线段树（从下到上）
    for (int i = m - 1; i > 0; i--) {
        for (int j = 0; j < 2 * n; j++) {
            tree[i][j] = tree[i << 1][j] + tree[i << 1 | 1][j];
        }
    }
}

/**
 * 更新矩阵中某个位置的值
 * @param row 行索引
 * @param col 列索引
 * @param val 新的值
 *
 * 时间复杂度: O(log m * log n)
 */
public void update(int row, int col, int val) {
    // 计算差值
    int delta = val - matrix[row][col];
    matrix[row][col] = val;

    // 更新线段树
```

```
        int i = row + m;
        while (i > 0) {
            int j = col + n;
            while (j > 0) {
                tree[i][j] += delta;
                j >>= 1;
            }
            i >>= 1;
        }
    }

    /**
     * 计算某一行范围内列的和
     * @param row 行索引（在线段树中的索引）
     * @param col1 列范围左边界
     * @param col2 列范围右边界
     * @return 该行范围内列的和
     *
     * 时间复杂度: O(log n)
     */
    private int sumRow(int row, int col1, int col2) {
        int sum = 0;
        int j = col1 + n;
        int k = col2 + n + 1;

        while (j < k) {
            if ((j & 1) == 1) {
                sum += tree[row][j];
                j++;
            }
            if ((k & 1) == 1) {
                k--;
                sum += tree[row][k];
            }
            j >>= 1;
            k >>= 1;
        }
        return sum;
    }

    /**
     * 查询子矩阵的总和
     * @param row1 子矩阵左上角行索引
```

```
     * @param col1 子矩阵左上角列索引
     * @param row2 子矩阵右下角行索引
     * @param col2 子矩阵右下角列索引
     * @return 子矩阵的总和
     *
     * 时间复杂度: O(log m * log n)
     */
    public int sumRegion(int row1, int col1, int row2, int col2) {
        int sum = 0;

        // 处理行范围
        int i = row1 + m;
        while (i <= row2 + m) {
            int r1 = i;
            int r2 = i;

            // 找到完整的区间
            while (r1 > 0 && (r1 % 2) == 0 && r2 + 1 <= row2 + m) {
                r1 >>= 1;
                r2 = (r2 + 1) >> 1;
            }

            // 处理列范围
            sum += sumRow(r1, col1, col2);

            // 移动到下一个区间
            if (r1 * 2 <= row2 + m) {
                i = r1 * 2 + 1;
            } else {
                break;
            }
        }

        return sum;
    }
}

/**
 * 测试方法
 */
public static void main(String[] args) {
    // 测试用例:
    // matrix = [
```

```
        //   [3, 0, 1, 4, 2],
        //   [5, 6, 3, 2, 1],
        //   [1, 2, 0, 1, 5],
        //   [4, 1, 0, 1, 7],
        //   [1, 0, 3, 0, 5]
        // ]
        // sumRegion(2, 1, 4, 3) => 8
        // update(3, 2, 2)
        // sumRegion(2, 1, 4, 3) => 10

        int[][] matrix = {
            {3, 0, 1, 4, 2},
            {5, 6, 3, 2, 1},
            {1, 2, 0, 1, 5},
            {4, 1, 0, 1, 7},
            {1, 0, 3, 0, 5}
        };

        NumMatrix numMatrix = new NumMatrix(matrix);
        System.out.println(numMatrix.sumRegion(2, 1, 4, 3));  // 应该输出 8
        numMatrix.update(3, 2, 2);
        System.out.println(numMatrix.sumRegion(2, 1, 4, 3));  // 应该输出 10
    }
}
```

==========

文件: Code08_RangeSumQuery2D.py

==========

```
# 308. 二维区域和检索 - 可变
# 给你一个 2D 矩阵 matrix，请你完成两类查询：
# 1. 更新矩阵中某个单元格的值
# 2. 计算子矩形范围内元素的总和，该子矩阵的左上角为 (row1, col1) ，右下角为 (row2, col2)
# 实现 NumMatrix 类：
# NumMatrix(int[][] matrix) 给定整数矩阵 matrix 进行初始化
# void update(int row, int col, int val) 更新 matrix[row][col] 的值到 val
# int sumRegion(int row1, int col1, int row2, int col2) 返回子矩阵的总和
# 测试链接 : https://leetcode.cn/problems/range-sum-query-2d-mutable/
# 请同学们务必参考如下代码中关于输入、输出的处理
# 这是输入输出处理效率很高的写法

class NumMatrix:
    def __init__(self, matrix):
```

```
        """
        :type matrix: List[List[int]]
        构造函数
        时间复杂度: O(m*n)
        空间复杂度: O(m*n)
        """
        if not matrix or not matrix[0]:
            return

        self.matrix = matrix
        self.m = len(matrix)
        self.n = len(matrix[0])
        # 初始化线段树
        self.tree = [[0 for _ in range(self.n * 2)] for _ in range(self.m * 2)]

        # 构建线段树
        self.buildTree()

    def buildTree(self):
        """
        构建二维线段树
        时间复杂度: O(m*n)
        """
        # 先构建每行的一维线段树
        for i in range(self.m):
            for j in range(self.n):
                self.tree[i + self.m][j + self.n] = self.matrix[i][j]

            # 构建行的线段树
            for j in range(self.n - 1, 0, -1):
                self.tree[i + self.m][j] = self.tree[i + self.m][j << 1] + self.tree[i +
self.m][j << 1 | 1]

        # 构建列的线段树
        for i in range(self.m - 1, 0, -1):
            for j in range(2 * self.n):
                self.tree[i][j] = self.tree[i << 1][j] + self.tree[i << 1 | 1][j]

    def update(self, row, col, val):
        """
        更新矩阵中某个位置的值
        时间复杂度: O(log m * log n)
        :type row: int
```

```
        :type col: int
        :type val: int
        """
        # 计算差值
        delta = val - self.matrix[row][col]
        self.matrix[row][col] = val

        # 更新线段树
        i = row + self.m
        while i > 0:
            j = col + self.n
            while j > 0:
                self.tree[i][j] += delta
                j >>= 1
            i >>= 1

    def sumRow(self, row, col1, col2):
        """
        计算某一行范围内列的和
        时间复杂度: O(log n)
        :type row: int
        :type col1: int
        :type col2: int
        :rtype: int
        """
        sum_val = 0
        j, k = col1 + self.n, col2 + self.n + 1
        while j < k:
            if j & 1:
                sum_val += self.tree[row][j]
                j += 1
            if k & 1:
                k -= 1
                sum_val += self.tree[row][k]
            j >>= 1
            k >>= 1
        return sum_val

    def sumRegion(self, row1, col1, row2, col2):
        """
        查询子矩阵的总和
        时间复杂度: O(log m * log n)
        :type row1: int
```

```
        :type col1: int
        :type row2: int
        :type col2: int
        :rtype: int
        """
        sum_val = 0

        # 处理行范围
        i = row1 + self.m
        while i <= row2 + self.m:
            r1, r2 = i, i

            # 找到完整的区间
            while r1 > 0 and r1 % 2 == 0 and r2 + 1 <= row2 + self.m:
                r1 >>= 1
                r2 = (r2 + 1) >> 1

            # 处理列范围
            sum_val += self.sumRow(r1, col1, col2)

            # 移动到下一个区间
            if r1 * 2 <= row2 + self.m:
                i = r1 * 2 + 1
            else:
                break

        return sum_val

# 测试方法
if __name__ == "__main__":
    # 测试用例:
    # matrix = [
    #   [3, 0, 1, 4, 2],
    #   [5, 6, 3, 2, 1],
    #   [1, 2, 0, 1, 5],
    #   [4, 1, 0, 1, 7],
    #   [1, 0, 3, 0, 5]
    # ]
    # sumRegion(2, 1, 4, 3) => 8
    # update(3, 2, 2)
    # sumRegion(2, 1, 4, 3) => 10

    matrix = [
```

```
        [3, 0, 1, 4, 2],
        [5, 6, 3, 2, 1],
        [1, 2, 0, 1, 5],
        [4, 1, 0, 1, 7],
        [1, 0, 3, 0, 5]
    ]

    numMatrix = NumMatrix(matrix)
    print(numMatrix.sumRegion(2, 1, 4, 3))  # 应该输出 8
    numMatrix.update(3, 2, 2)
    print(numMatrix.sumRegion(2, 1, 4, 3))  # 应该输出 10
```

==========

文件: Code09_CountOfSmallerNumbersAfterSelf.cpp

==========

```
// 315. 计算右侧小于当前元素的个数 - 线段树实现
// 题目来源：LeetCode 315 https://leetcode.cn/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。
//
// 解题思路：
// 使用线段树配合离散化来解决这个问题
// 1. 先对数组进行离散化处理，将值域映射到连续的小范围
// 2. 从右向左遍历数组，这样可以确保每次处理的元素右侧元素都已经处理过
// 3. 使用线段树维护离散化后的值域信息，记录每个值出现的次数
// 4. 对于当前元素，查询值域中小于它的元素个数，即为右侧小于当前元素的个数
// 5. 将当前元素加入线段树，供后续元素查询使用
//
// 时间复杂度分析：
// - 离散化：O(n^2) （使用了简单的冒泡排序）
// - 构建线段树：O(n)
// - 处理每个元素：O(log n)
// - 总时间复杂度：O(n^2)
// 空间复杂度：O(n)

const int MAXN = 100000;

// 线段树数组
int tree[MAXN * 4];
int n;
```

```
/**
 * 更新线段树
 * @param node 当前节点索引
 * @param start 当前节点维护的区间左边界
 * @param end 当前节点维护的区间右边界
 * @param idx 要更新的位置
 * @param val 要增加的值
 *
 * 时间复杂度: O(log n)
 */
void update(int node, int start, int end, int idx, int val) {
    // 到达叶子节点，更新值
    if (start == end) {
        tree[node] += val;
    } else {
        int mid = (start + end) / 2;
        // 根据位置决定更新左子树还是右子树
        if (idx <= mid) {
            update(2 * node, start, mid, idx, val);
        } else {
            update(2 * node + 1, mid + 1, end, idx, val);
        }
        // 更新当前节点的值为左右子节点值之和
        tree[node] = tree[2 * node] + tree[2 * node + 1];
    }
}

/**
 * 查询区间和
 * @param node 当前节点索引
 * @param start 当前节点维护的区间左边界
 * @param end 当前节点维护的区间右边界
 * @param l 查询区间左边界
 * @param r 查询区间右边界
 * @return 区间[l,r]内元素的和
 *
 * 时间复杂度: O(log n)
 */
int query(int node, int start, int end, int l, int r) {
    // 查询区间与当前节点维护区间无交集，返回 0
    if (l > end || r < start) {
        return 0;
```

```
    }
    // 当前节点维护区间完全包含在查询区间内，返回节点值
    if (l <= start && end <= r) {
        return tree[node];
    }
    // 部分重叠，递归查询左右子树
    int mid = (start + end) / 2;
    return query(2 * node, start, mid, l, r) +
           query(2 * node + 1, mid + 1, end, l, r);
}

/**
 * 离散化处理
 * @param nums 原始数组
 * @param sorted 用于存储排序后去重数组的数组
 * @param size 原始数组大小
 * @return 去重后数组的大小
 */
int discretize(int* nums, int* sorted, int size) {
    // 复制原始数组
    for (int i = 0; i < size; i++) {
        sorted[i] = nums[i];
    }

    // 简单排序（冒泡排序）
    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 unique_size = 1;
    for (int i = 1; i < size; i++) {
        if (sorted[i] != sorted[i - 1]) {
            sorted[unique_size++] = sorted[i];
        }
    }
```

```
    return unique_size;
}

/**
 * 二分查找
 * @param arr 已排序的数组
 * @param size 数组大小
 * @param target 要查找的目标值
 * @return 目标值在数组中的索引，未找到返回-1
 */
int binarySearch(int* arr, int size, int target) {
    int left = 0, right = size - 1;
    while (left <= right) {
        int mid = (left + right) / 2;
        if (arr[mid] == target) {
            return mid;
        } else if (arr[mid] < target) {
            left = mid + 1;
        } else {
            right = mid - 1;
        }
    }
    return -1;
}

/**
 * 主函数：计算右侧小于当前元素的个数
 * @param nums 输入数组
 * @param size 输入数组大小
 * @param result 结果数组
 *
 * 时间复杂度: O(n^2)
 * 空间复杂度: O(n)
 */
void countSmaller(int* nums, int size, int* result) {
    // 处理边界情况
    if (nums == 0 || size == 0) {
        return;
    }

    // 离散化处理
    int sorted[MAXN];
    int unique_size = discretize(nums, sorted, size);
```

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

    // 从右向左遍历数组
    for (int i = size - 1; i >= 0; i--) {
        // 查找当前元素在离散化数组中的位置
        int pos = binarySearch(sorted, unique_size, nums[i]);
        // 查询小于当前元素的个数（即右侧小于当前元素的个数）
        result[i] = query(1, 0, unique_size - 1, 0, pos - 1);
        // 将当前元素加入线段树，供后续元素查询使用
        update(1, 0, unique_size - 1, pos, 1);
    }
}
```

==========

文件: Code09_CountOfSmallerNumbersAfterSelf.java
==========

```
package class112;

// 315. 计算右侧小于当前元素的个数 - 动态开点线段树实现
// 题目来源：LeetCode 315 https://leetcode.cn/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。
//
// 解题思路：
// 使用动态开点线段树来解决这个问题
// 1. 从右向左遍历数组，这样可以确保每次处理的元素右侧元素都已经处理过
// 2. 使用线段树维护值域信息，记录每个值出现的次数
// 3. 对于当前元素，查询值域中小于它的元素个数，即为右侧小于当前元素的个数
// 4. 将当前元素加入线段树，供后续元素查询使用
//
// 时间复杂度分析：
// - 离散化：O(n log n)
// - 构建线段树：O(1)
// - 处理每个元素：O(log n)
// - 总时间复杂度：O(n log n)
```

```
// 空间复杂度：O(n)
//
// 请同学们务必参考如下代码中关于输入、输出的处理
// 这是输入输出处理效率很高的写法
// 提交以下的 code，提交时请把类名改成"Main"，可以直接通过

import java.util.*;

public class Code09_CountOfSmallerNumbersAfterSelf {

    // 线段树节点
    static class SegmentTreeNode {
        int start, end;           // 节点维护的值域范围
        int count;                // 该值域范围内元素的个数
        SegmentTreeNode left, right;  // 左右子节点

        SegmentTreeNode(int start, int end) {
            this.start = start;
            this.end = end;
            this.count = 0;
            this.left = null;
            this.right = null;
        }
    }

    // 线段树实现
    static class SegmentTree {
        SegmentTreeNode root;  // 线段树根节点

        SegmentTree(int start, int end) {
            this.root = new SegmentTreeNode(start, end);
        }

        /**
         * 更新线段树，将值 val 的计数加 1
         * @param val 要更新的值
         *
         * 时间复杂度: O(log n)
         */
        public void update(int val) {
            updateHelper(root, val);
        }
```

```
/**
 * 更新线段树的辅助函数
 * @param node 当前节点
 * @param val 要更新的值
 */
private void updateHelper(SegmentTreeNode node, int val) {
    // 到达叶子节点，更新计数
    if (node.start == node.end) {
        node.count++;
        return;
    }

    int mid = node.start + (node.end - node.start) / 2;
    // 根据值的大小决定更新左子树还是右子树
    if (val <= mid) {
        // 如果左子节点不存在，则创建
        if (node.left == null) {
            node.left = new SegmentTreeNode(node.start, mid);
        }
        updateHelper(node.left, val);
    } else {
        // 如果右子节点不存在，则创建
        if (node.right == null) {
            node.right = new SegmentTreeNode(mid + 1, node.end);
        }
        updateHelper(node.right, val);
    }

    // 更新当前节点的计数为左右子节点计数之和
    node.count = (node.left == null ? 0 : node.left.count) +
                 (node.right == null ? 0 : node.right.count);
}

/**
 * 查询小于等于某个值的元素个数
 * @param val 查询的值
 * @return 小于等于 val 的元素个数
 *
 * 时间复杂度: O(log n)
 */
public int query(int val) {
    return queryHelper(root, val);
}
```

```
    /**
     * 查询小于等于某个值的元素个数的辅助函数
     * @param node 当前节点
     * @param val 查询的值
     * @return 小于等于 val 的元素个数
     */
    private int queryHelper(SegmentTreeNode node, int val) {
        // 节点为空或查询值小于节点维护的最小值，返回 0
        if (node == null || val < node.start) {
            return 0;
        }

        // 查询值大于等于节点维护的最大值，返回该节点的计数
        if (val >= node.end) {
            return node.count;
        }

        int mid = node.start + (node.end - node.start) / 2;
        // 根据值的大小决定查询左子树还是右子树
        if (val <= mid) {
            return queryHelper(node.left, val);
        } else {
            // 查询值在右半部分，需要加上左半部分的计数
            return (node.left == null ? 0 : node.left.count) +
                   queryHelper(node.right, val);
        }
    }
}

/**
 * 主函数：计算右侧小于当前元素的个数
 * @param nums 输入数组
 * @return 结果数组，counts[i]表示 nums[i]右侧小于 nums[i]的元素数量
 *
 * 时间复杂度: O(n log n)
 * 空间复杂度: O(n)
 */
public List<Integer> countSmaller(int[] nums) {
    if (nums == null || nums.length == 0) {
        return new ArrayList<>();
    }
```

```
        // 离散化处理，获取值域范围
        int[] sorted = nums.clone();
        Arrays.sort(sorted);

        // 构建线段树，值域为数组中的最小值到最大值
        SegmentTree tree = new SegmentTree(sorted[0], sorted[sorted.length - 1]);

        List<Integer> result = new ArrayList<>();
        // 从右向左遍历数组
        for (int i = nums.length - 1; i >= 0; i--) {
            // 查询小于当前元素的个数（即右侧小于当前元素的个数）
            int count = tree.query(nums[i] - 1);
            // 将结果插入到列表开头，保持与原数组相同的顺序
            result.add(0, count);
            // 将当前元素加入线段树，供后续元素查询使用
            tree.update(nums[i]);
        }

        return result;
    }

    /**
     * 测试方法
     */
    public static void main(String[] args) {
        Code09_CountOfSmallerNumbersAfterSelf solution = new
Code09_CountOfSmallerNumbersAfterSelf();

        // 测试用例 1: nums = [5,2,6,1]
        // 输出: [2,1,1,0]
        // 解释：5 右侧有 2 个元素(2,1)小于 5，2 右侧有 1 个元素(1)小于 2，
        //       6 右侧有 1 个元素(1)小于 6，1 右侧有 0 个元素小于 1
        int[] nums1 = {5, 2, 6, 1};
        System.out.println(solution.countSmaller(nums1)); // 应该输出 [2, 1, 1, 0]

        // 测试用例 2: nums = [-1]
        // 输出: [0]
        // 解释：-1 右侧没有元素
        int[] nums2 = {-1};
        System.out.println(solution.countSmaller(nums2)); // 应该输出 [0]

        // 测试用例 3: nums = [-1,-1]
        // 输出: [0,0]
```

```
        // 解释：第一个-1 右侧有 1 个-1，但不大于它，所以是 0；
        //       第二个-1 右侧没有元素，所以是 0
        int[] nums3 = {-1, -1};
        System.out.println(solution.countSmaller(nums3)); // 应该输出 [0, 0]
    }
}
```

==========

文件: Code09_CountOfSmallerNumbersAfterSelf.py

==========

```
# 315. 计算右侧小于当前元素的个数 - 动态开点线段树实现
# 题目来源：LeetCode 315 https://leetcode.cn/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。
#
# 解题思路：
# 使用动态开点线段树来解决这个问题
# 1. 从右向左遍历数组，这样可以确保每次处理的元素右侧元素都已经处理过
# 2. 使用线段树维护值域信息，记录每个值出现的次数
# 3. 对于当前元素，查询值域中小于它的元素个数，即为右侧小于当前元素的个数
# 4. 将当前元素加入线段树，供后续元素查询使用
#
# 时间复杂度分析：
# - 离散化：O(n log n)
# - 构建线段树：O(1)
# - 处理每个元素：O(log n)
# - 总时间复杂度：O(n log n)
# 空间复杂度：O(n)

class SegmentTreeNode:
    def __init__(self, start, end):
        """
        线段树节点
        :param start: 节点维护的值域范围起始值
        :param end: 节点维护的值域范围结束值
        """
        self.start = start          # 节点维护的值域范围
        self.end = end              # 节点维护的值域范围
        self.count = 0              # 该值域范围内元素的个数
        self.left = None            # 左子节点
```

```
        self.right = None            # 右子节点

class SegmentTree:
    def __init__(self, start, end):
        """
        线段树实现
        :param start: 线段树维护的值域范围起始值
        :param end: 线段树维护的值域范围结束值
        """
        self.root = SegmentTreeNode(start, end)  # 线段树根节点

    def update(self, val):
        """
        更新线段树，将值 val 的计数加 1
        :param val: 要更新的值
        时间复杂度: O(log n)
        """
        self.updateHelper(self.root, val)

    def updateHelper(self, node, val):
        """
        更新线段树的辅助函数
        :param node: 当前节点
        :param val: 要更新的值
        """
        # 到达叶子节点，更新计数
        if node.start == node.end:
            node.count += 1
            return

        mid = node.start + (node.end - node.start) // 2
        # 根据值的大小决定更新左子树还是右子树
        if val <= mid:
            # 如果左子节点不存在，则创建
            if node.left is None:
                node.left = SegmentTreeNode(node.start, mid)
            self.updateHelper(node.left, val)
        else:
            # 如果右子节点不存在，则创建
            if node.right is None:
                node.right = SegmentTreeNode(mid + 1, node.end)
            self.updateHelper(node.right, val)
```

```
        # 更新当前节点的计数为左右子节点计数之和
        node.count = (node.left.count if node.left else 0) + \
                     (node.right.count if node.right else 0)

    def query(self, val):
        """
        查询小于等于某个值的元素个数
        :param val: 查询的值
        :return: 小于等于 val 的元素个数
        时间复杂度: O(log n)
        """
        return self.queryHelper(self.root, val)

    def queryHelper(self, node, val):
        """
        查询小于等于某个值的元素个数的辅助函数
        :param node: 当前节点
        :param val: 查询的值
        :return: 小于等于 val 的元素个数
        """
        # 节点为空或查询值小于节点维护的最小值，返回 0
        if node is None or val < node.start:
            return 0

        # 查询值大于等于节点维护的最大值，返回该节点的计数
        if val >= node.end:
            return node.count

        mid = node.start + (node.end - node.start) // 2
        # 根据值的大小决定查询左子树还是右子树
        if val <= mid:
            return self.queryHelper(node.left, val)
        else:
            # 查询值在右半部分，需要加上左半部分的计数
            left_count = node.left.count if node.left else 0
            return left_count + self.queryHelper(node.right, val)

def countSmaller(nums):
    """
    主函数：计算右侧小于当前元素的个数
    :param nums: 输入数组
    :return: 结果数组，counts[i]表示 nums[i]右侧小于 nums[i]的元素数量
    时间复杂度: O(n log n)
```

```
    空间复杂度: O(n)
    """
    if not nums:
        return []

    # 离散化处理，获取值域范围
    sorted_nums = sorted(nums)

    # 构建线段树，值域为数组中的最小值到最大值
    tree = SegmentTree(sorted_nums[0], sorted_nums[-1])

    result = []
    # 从右向左遍历数组
    for i in range(len(nums) - 1, -1, -1):
        # 查询小于当前元素的个数（即右侧小于当前元素的个数）
        count = tree.query(nums[i] - 1)
        # 将结果插入到列表开头，保持与原数组相同的顺序
        result.insert(0, count)
        # 将当前元素加入线段树，供后续元素查询使用
        tree.update(nums[i])

    return result

# 测试方法
if __name__ == "__main__":
    # 测试用例 1: nums = [5,2,6,1]
    # 输出: [2,1,1,0]
    # 解释：5 右侧有 2 个元素(2,1)小于 5，2 右侧有 1 个元素(1)小于 2，
    #       6 右侧有 1 个元素(1)小于 6，1 右侧有 0 个元素小于 1
    nums1 = [5, 2, 6, 1]
    print(countSmaller(nums1))  # 应该输出 [2, 1, 1, 0]

    # 测试用例 2: nums = [-1]
    # 输出: [0]
    # 解释：-1 右侧没有元素
    nums2 = [-1]
    print(countSmaller(nums2))  # 应该输出 [0]

    # 测试用例 3: nums = [-1,-1]
    # 输出: [0,0]
    # 解释：第一个-1 右侧有 1 个-1，但不大于它，所以是 0；
    #       第二个-1 右侧没有元素，所以是 0
    nums3 = [-1, -1]
```

```
    print(countSmaller(nums3))  # 应该输出 [0, 0]

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

文件: Code10_ReversePairs.java
================================================
package class112;

// 493. 翻转对 - 动态开点线段树实现
// 题目来源：LeetCode 493 https://leetcode.cn/problems/reverse-pairs/
//
// 题目描述：
// 给定一个数组 nums ，如果 i < j 且 nums[i] > 2*nums[j] 我们将 (i, j) 称作一个重要翻转对。
// 你需要返回给定数组中的重要翻转对的数量。
//
// 解题思路：
// 使用动态开点线段树来解决翻转对问题
// 1. 从右向左遍历数组，这样可以确保每次处理的元素右侧元素都已经处理过
// 2. 使用线段树维护值域信息，记录每个值出现的次数
// 3. 对于当前元素 nums[i]，查询值域中大于 2*nums[i]的元素个数，即为以 i 为第一个元素的翻转对数量
// 4. 将当前元素加入线段树，供后续元素查询使用
//
// 时间复杂度分析：
// - 收集所有值：O(n)
// - 离散化：O(n log n)
// - 构建线段树：O(1)
// - 处理每个元素：O(log n)
// - 总时间复杂度：O(n log n)
// 空间复杂度：O(n)
//
// 请同学们务必参考如下代码中关于输入、输出的处理
// 这是输入输出处理效率很高的写法
// 提交以下的 code，提交时请把类名改成"Main"，可以直接通过

import java.util.*;

public class Code10_ReversePairs {

    // 线段树节点
    static class SegmentTreeNode {
        int start, end;           // 节点维护的值域范围
        int count;                // 该值域范围内元素的个数
        SegmentTreeNode left, right;  // 左右子节点
```

```
    SegmentTreeNode(int start, int end) {
        this.start = start;
        this.end = end;
        this.count = 0;
        this.left = null;
        this.right = null;
    }
}

// 线段树实现
static class SegmentTree {
    SegmentTreeNode root;  // 线段树根节点

    SegmentTree(int start, int end) {
        this.root = new SegmentTreeNode(start, end);
    }

    /**
     * 更新线段树，将值 val 的计数加 1
     * @param val 要更新的值
     *
     * 时间复杂度: O(log n)
     */
    public void update(int val) {
        updateHelper(root, val);
    }

    /**
     * 更新线段树的辅助函数
     * @param node 当前节点
     * @param val 要更新的值
     */
    private void updateHelper(SegmentTreeNode node, int val) {
        // 到达叶子节点，更新计数
        if (node.start == node.end) {
            node.count++;
            return;
        }

        int mid = node.start + (node.end - node.start) / 2;
        // 根据值的大小决定更新左子树还是右子树
        if (val <= mid) {
```

```
            // 如果左子节点不存在，则创建
            if (node.left == null) {
                node.left = new SegmentTreeNode(node.start, mid);
            }
            updateHelper(node.left, val);
        } else {
            // 如果右子节点不存在，则创建
            if (node.right == null) {
                node.right = new SegmentTreeNode(mid + 1, node.end);
            }
            updateHelper(node.right, val);
        }

        // 更新当前节点的计数为左右子节点计数之和
        node.count = (node.left == null ? 0 : node.left.count) +
                     (node.right == null ? 0 : node.right.count);
    }

    /**
     * 查询大于等于某个值的元素个数
     * @param val 查询的值
     * @return 大于等于 val 的元素个数
     *
     * 时间复杂度: O(log n)
     */
    public int query(int val) {
        return queryHelper(root, val);
    }

    /**
     * 查询大于等于某个值的元素个数的辅助函数
     * @param node 当前节点
     * @param val 查询的值
     * @return 大于等于 val 的元素个数
     */
    private int queryHelper(SegmentTreeNode node, int val) {
        // 节点为空或查询值大于节点维护的最大值，返回 0
        if (node == null || val > node.end) {
            return 0;
        }

        // 查询值小于等于节点维护的最小值，返回该节点的计数
        if (val <= node.start) {
```

```
            return node.count;
        }

        int mid = node.start + (node.end - node.start) / 2;
        // 根据值的大小决定查询左子树还是右子树
        if (val <= mid) {
            // 查询值在左半部分，需要加上右半部分的计数
            return queryHelper(node.left, val) +
                   (node.right == null ? 0 : node.right.count);
        } else {
            return queryHelper(node.right, val);
        }
    }
}

/**
 * 主函数：计算翻转对的数量
 * @param nums 输入数组
 * @return 翻转对的数量
 *
 * 时间复杂度: O(n log n)
 * 空间复杂度: O(n)
 */
public int reversePairs(int[] nums) {
    if (nums == null || nums.length == 0) {
        return 0;
    }

    // 收集所有可能的值用于离散化（包括 nums[i]和 2*nums[i]）
    Set<Long> allNumbers = new HashSet<>();
    for (int num : nums) {
        allNumbers.add((long) num);
        allNumbers.add(2L * num);
    }

    // 离散化处理
    long[] sorted = new long[allNumbers.size()];
    int index = 0;
    for (long num : allNumbers) {
        sorted[index++] = num;
    }
    Arrays.sort(sorted);
```

```
        // 构建线段树，值域为离散化后的索引范围
        SegmentTree tree = new SegmentTree(0, sorted.length - 1);

        int result = 0;
        // 从右向左遍历数组
        for (int i = nums.length - 1; i >= 0; i--) {
            // 查找 2*nums[i]在离散化数组中的位置
            int pos = Arrays.binarySearch(sorted, 2L * nums[i]);
            // 查询大于 2*nums[i]的元素个数（即以 i 为第一个元素的翻转对数量）
            result += tree.query(pos + 1);
            // 查找 nums[i]在离散化数组中的位置并更新线段树
            pos = Arrays.binarySearch(sorted, (long) nums[i]);
            tree.update(pos);
        }

        return result;
    }

    /**
     * 测试方法
     */
    public static void main(String[] args) {
        Code10_ReversePairs solution = new Code10_ReversePairs();

        // 测试用例 1: nums = [1,3,2,3,1]
        // 输出: 2
        // 解释：翻转对是(1,4)和(3,4)，即(1>2*1)和(3>2*1)
        int[] nums1 = {1, 3, 2, 3, 1};
        System.out.println(solution.reversePairs(nums1)); // 应该输出 2

        // 测试用例 2: nums = [2,4,3,5,1]
        // 输出: 3
        // 解释：翻转对是(0,4)、(1,4)和(2,4)，即(2>2*1)、(4>2*1)和(3>2*1)
        int[] nums2 = {2, 4, 3, 5, 1};
        System.out.println(solution.reversePairs(nums2)); // 应该输出 3
    }
}

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

文件: Code10_ReversePairs.py
==========================================
# 493. 翻转对 - 动态开点线段树实现
```

```
# 题目来源：LeetCode 493 https://leetcode.cn/problems/reverse-pairs/
#
# 题目描述：
# 给定一个数组 nums ，如果 i < j 且 nums[i] > 2*nums[j] 我们将 (i, j) 称作一个重要翻转对。
# 你需要返回给定数组中的重要翻转对的数量。
#
# 解题思路：
# 使用动态开点线段树来解决翻转对问题
# 1. 从右向左遍历数组，这样可以确保每次处理的元素右侧元素都已经处理过
# 2. 使用线段树维护值域信息，记录每个值出现的次数
# 3. 对于当前元素 nums[i]，查询值域中大于 2*nums[i]的元素个数，即为以 i 为第一个元素的翻转对数量
# 4. 将当前元素加入线段树，供后续元素查询使用
#
# 时间复杂度分析：
# - 收集所有值：O(n)
# - 离散化：O(n log n)
# - 构建线段树：O(1)
# - 处理每个元素：O(log n)
# - 总时间复杂度：O(n log n)
# 空间复杂度：O(n)

class SegmentTreeNode:
    def __init__(self, start, end):
        """
        线段树节点
        :param start: 节点维护的值域范围起始值
        :param end: 节点维护的值域范围结束值
        """
        self.start = start          # 节点维护的值域范围
        self.end = end              # 节点维护的值域范围
        self.count = 0              # 该值域范围内元素的个数
        self.left = None            # 左子节点
        self.right = None           # 右子节点

class SegmentTree:
    def __init__(self, start, end):
        """
        线段树实现
        :param start: 线段树维护的值域范围起始值
        :param end: 线段树维护的值域范围结束值
        """
        self.root = SegmentTreeNode(start, end)  # 线段树根节点
```

```
def update(self, val):
    """
    更新线段树，将值 val 的计数加 1
    :param val: 要更新的值
    时间复杂度: O(log n)
    """
    self.updateHelper(self.root, val)

def updateHelper(self, node, val):
    """
    更新线段树的辅助函数
    :param node: 当前节点
    :param val: 要更新的值
    """
    # 到达叶子节点，更新计数
    if node.start == node.end:
        node.count += 1
        return

    mid = node.start + (node.end - node.start) // 2
    # 根据值的大小决定更新左子树还是右子树
    if val <= mid:
        # 如果左子节点不存在，则创建
        if node.left is None:
            node.left = SegmentTreeNode(node.start, mid)
        self.updateHelper(node.left, val)
    else:
        # 如果右子节点不存在，则创建
        if node.right is None:
            node.right = SegmentTreeNode(mid + 1, node.end)
        self.updateHelper(node.right, val)

    # 更新当前节点的计数为左右子节点计数之和
    node.count = (node.left.count if node.left else 0) + \
                 (node.right.count if node.right else 0)

def query(self, val):
    """
    查询大于等于某个值的元素个数
    :param val: 查询的值
    :return: 大于等于 val 的元素个数
    时间复杂度: O(log n)
    """
```

```
        return self.queryHelper(self.root, val)

    def queryHelper(self, node, val):
        """
        查询大于等于某个值的元素个数的辅助函数
        :param node: 当前节点
        :param val: 查询的值
        :return: 大于等于 val 的元素个数
        """
        # 节点为空或查询值大于节点维护的最大值，返回 0
        if node is None or val > node.end:
            return 0

        # 查询值小于等于节点维护的最小值，返回该节点的计数
        if val <= node.start:
            return node.count

        mid = node.start + (node.end - node.start) // 2
        # 根据值的大小决定查询左子树还是右子树
        if val <= mid:
            # 查询值在左半部分，需要加上右半部分的计数
            right_count = node.right.count if node.right else 0
            return self.queryHelper(node.left, val) + right_count
        else:
            return self.queryHelper(node.right, val)

def reversePairs(nums):
    """
    主函数：计算翻转对的数量
    :param nums: 输入数组
    :return: 翻转对的数量
    时间复杂度: O(n log n)
    空间复杂度: O(n)
    """
    if not nums:
        return 0

    # 收集所有可能的值用于离散化（包括 nums[i]和 2*nums[i]）
    all_numbers = set()
    for num in nums:
        all_numbers.add(num)
        all_numbers.add(2 * num)
```

```
    # 离散化处理
    sorted_nums = sorted(list(all_numbers))

    # 构建线段树，值域为离散化后的索引范围
    tree = SegmentTree(0, len(sorted_nums) - 1)

    result = 0
    # 从右向左遍历数组
    for i in range(len(nums) - 1, -1, -1):
        # 查找 2*nums[i]在离散化数组中的位置
        pos = sorted_nums.index(2 * nums[i])
        # 查询大于 2*nums[i]的元素个数（即以 i 为第一个元素的翻转对数量）
        result += tree.query(pos + 1)
        # 查找 nums[i]在离散化数组中的位置并更新线段树
        pos = sorted_nums.index(nums[i])
        tree.update(pos)

    return result

# 测试方法
if __name__ == "__main__":
    # 测试用例 1: nums = [1,3,2,3,1]
    # 输出: 2
    # 解释：翻转对是(1,4)和(3,4)，即(1>2*1)和(3>2*1)
    nums1 = [1, 3, 2, 3, 1]
    print(reversePairs(nums1))  # 应该输出 2

    # 测试用例 2: nums = [2,4,3,5,1]
    # 输出: 3
    # 解释：翻转对是(0,4)、(1,4)和(2,4)，即(2>2*1)、(4>2*1)和(3>2*1)
    nums2 = [2, 4, 3, 5, 1]
    print(reversePairs(nums2))  # 应该输出 3
```

==========

文件: Code11_CountOfRangeSum.cpp

==========

```
// 327. 区间和的个数 - 线段树实现
// 题目来源：LeetCode 327 https://leetcode.cn/problems/count-of-range-sum/
//
// 题目描述：
// 给你一个整数数组 nums 以及两个整数 lower 和 upper 。求数组中，值位于范围 [lower, upper] （包含 lower 和 upper）之内的区间和的个数。
```

```
// 区间和 S(i, j) 表示在 nums 中，位置从 i 到 j 的元素之和，包含 i 和 j (i ≤ j)。
//
// 解题思路：
// 使用线段树配合前缀和与离散化来解决区间和个数问题
// 1. 计算前缀和数组，将区间和问题转换为前缀和差值问题
// 2. 对于前缀和 prefixSum[i]，需要找到满足 lower <= prefixSum[j] - prefixSum[i] <= upper 的 j 个数
// 3. 转换为 prefixSum[i] + lower <= prefixSum[j] <= prefixSum[i] + upper
// 4. 从右向左遍历前缀和数组，使用线段树维护已处理的前缀和信息
// 5. 对于当前前缀和 prefixSum[i]，在线段树中查询满足条件的前缀和个数
// 6. 将当前前缀和加入线段树，供后续元素查询使用
//
// 时间复杂度分析：
// - 计算前缀和：O(n)
// - 离散化：O(n log n)
// - 构建线段树：O(n)
// - 处理每个前缀和：O(log n)
// - 总时间复杂度：O(n log n)
// 空间复杂度：O(n)

// 由于编译环境限制，使用基础 C++实现，避免复杂 STL 容器

const int MAXN = 10000;

// 线段树数组
int tree[MAXN * 4];
int n;  // 离散化后的值域大小

/**
 * 更新线段树中的某个位置的值
 * @param node 当前节点索引
 * @param start 当前节点维护的区间左边界
 * @param end 当前节点维护的区间右边界
 * @param index 要更新的位置索引
 * @param val 要增加的值
 *
 * 时间复杂度: O(log n)
 */
void updateHelper(int node, int start, int end, int index, int val) {
    // 到达叶子节点，直接更新
    if (start == end) {
        tree[node] += val;
    } else {
        int mid = start + (end - start) / 2;
```

```
        // 根据位置决定更新左子树还是右子树
        if (index <= mid) {
            // 在左子树中更新
            updateHelper(2 * node, start, mid, index, val);
        } else {
            // 在右子树中更新
            updateHelper(2 * node + 1, mid + 1, end, index, val);
        }
        // 更新当前节点的值（合并子节点）
        tree[node] = tree[2 * node] + tree[2 * node + 1];
    }
}

/**
 * 更新线段树中的某个位置的值
 * @param index 要更新的位置索引
 * @param val 要增加的值
 *
 * 时间复杂度: O(log n)
 */
void update(int index, int val) {
    updateHelper(1, 0, n - 1, index, val);
}

/**
 * 查询辅助函数
 * @param node 当前节点索引
 * @param start 当前节点维护的区间左边界
 * @param end 当前节点维护的区间右边界
 * @param left 查询区间左边界
 * @param right 查询区间右边界
 * @return 区间[left, right]内元素的和
 */
int queryHelper(int node, int start, int end, int left, int right) {
    // 查询区间与当前节点维护区间无交集，返回 0
    if (right < start || end < left) {
        return 0;
    }
    // 当前节点维护区间完全包含在查询区间内，返回节点值
    if (left <= start && end <= right) {
        return tree[node];
    }
    // 部分重叠，递归查询左右子树
```

```
    int mid = start + (end - start) / 2;
    return queryHelper(2 * node, start, mid, left, right) +
           queryHelper(2 * node + 1, mid + 1, end, 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[left, right]内元素的和
 *
 * 时间复杂度: O(log n)
 */
int query(int left, int right) {
    // 处理边界情况
    if (left < 0) left = 0;
    if (right >= n) right = n - 1;
    if (left > right) return 0;

    return queryHelper(1, 0, n - 1, left, right);
}

// 简单排序函数（冒泡排序）
void bubbleSort(long long* 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]) {
                long long temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
}

// 二分查找函数
int binarySearch(long long* arr, int size, long long target) {
    int left = 0, right = size - 1;
    while (left <= right) {
        int mid = (left + right) / 2;
        if (arr[mid] == target) {
            return mid;
        } else if (arr[mid] < target) {
```

```
            left = mid + 1;
        } else {
            right = mid - 1;
        }
    }
    return -1;
}

/**
 * 主函数：计算区间和的个数
 * @param nums 输入数组
 * @param numsSize 输入数组大小
 * @param lower 区间下界
 * @param upper 区间上界
 * @return 区间和在[lower, upper]范围内的个数
 *
 * 时间复杂度: O(n log n)
 * 空间复杂度: O(n)
 */
int countRangeSum(int* nums, int numsSize, int lower, int upper) {
    // 处理边界情况
    if (nums == 0 || numsSize == 0) {
        return 0;
    }

    // 计算前缀和数组，prefixSum[0] = 0, prefixSum[i] = nums[0] + ... + nums[i-1]
    long long prefixSum[MAXN + 1];
    prefixSum[0] = 0;
    for (int i = 0; i < numsSize; ++i) {
        prefixSum[i + 1] = prefixSum[i] + nums[i];
    }

    // 离散化处理，收集所有可能需要的值
    long long uniqueValues[MAXN * 3];
    int uniqueCount = 0;

    // 收集前缀和
    for (int i = 0; i <= numsSize; ++i) {
        uniqueValues[uniqueCount++] = prefixSum[i];
    }

    // 收集用于查询下界的值
    for (int i = 0; i <= numsSize; ++i) {
```

```
    uniqueValues[uniqueCount++] = prefixSum[i] - lower;
}

// 收集用于查询上界的值
for (int i = 0; i <= numsSize; ++i) {
    uniqueValues[uniqueCount++] = prefixSum[i] - upper;
}

// 去重
bubbleSort(uniqueValues, uniqueCount);
int actualCount = 1;
for (int i = 1; i < uniqueCount; ++i) {
    if (uniqueValues[i] != uniqueValues[i - 1]) {
        uniqueValues[actualCount++] = uniqueValues[i];
    }
}

// 创建值到索引的映射（使用线性查找，简化实现）
n = actualCount;

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

int count = 0;

// 从右到左遍历前缀和，使用线段树查询符合条件的区间和
for (int i = numsSize; i >= 0; --i) {
    long long current = prefixSum[i];
    // 查询满足 lower <= prefixSum[j] - prefixSum[i] <= upper 的 j 的数量
    // 即查询 prefixSum[j]在[current + lower, current + upper]范围内的数量
    // 转换为查询离散化后的索引范围
    int left = binarySearch(uniqueValues, actualCount, current + lower);
    int right = binarySearch(uniqueValues, actualCount, current + upper);
    count += query(left, right);

    // 将当前前缀和添加到线段树中，供后续元素查询使用
    int currentIndex = binarySearch(uniqueValues, actualCount, current);
    update(currentIndex, 1);
}

return count;
```

```
}

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

文件: Code11_CountOfRangeSum.java
==========================================
package class112;

// 327. 区间和的个数 - 线段树实现
// 题目来源：LeetCode 327 https://leetcode.cn/problems/count-of-range-sum/
//
// 题目描述：
// 给你一个整数数组 nums 以及两个整数 lower 和 upper 。求数组中，值位于范围 [lower, upper] （包含 lower 和 upper）之内的区间和的个数。
// 区间和 S(i, j) 表示在 nums 中，位置从 i 到 j 的元素之和，包含 i 和 j (i ≤ j)。
//
// 解题思路：
// 使用线段树配合前缀和与离散化来解决区间和个数问题
// 1. 计算前缀和数组，将区间和问题转换为前缀和差值问题
// 2. 对于前缀和 prefixSum[i]，需要找到满足 lower <= prefixSum[j] - prefixSum[i] <= upper 的 j 个数
// 3. 转换为 prefixSum[i] + lower <= prefixSum[j] <= prefixSum[i] + upper
// 4. 从右向左遍历前缀和数组，使用线段树维护已处理的前缀和信息
// 5. 对于当前前缀和 prefixSum[i]，在线段树中查询满足条件的前缀和个数
// 6. 将当前前缀和加入线段树，供后续元素查询使用
//
// 时间复杂度分析：
// - 计算前缀和：O(n)
// - 离散化：O(n log n)
// - 构建线段树：O(n)
// - 处理每个前缀和：O(log n)
// - 总时间复杂度：O(n log n)
// 空间复杂度：O(n)

import java.util.*;

public class Code11_CountOfRangeSum {

    // 线段树节点类
    static class SegmentTree {
        private int[] tree; // 存储线段树节点值
        private int n;      // 离散化后的值域大小

        /**
```

```
     * 构造函数
     * @param size 线段树维护的区间大小
     *
     * 时间复杂度: O(n)
     * 空间复杂度: O(n)
     */
    public SegmentTree(int size) {
        this.n = size;
        this.tree = new int[4 * n]; // 线段树通常需要 4 倍空间
    }

    /**
     * 更新线段树中的某个位置的值
     * @param index 要更新的位置索引
     * @param val 要增加的值
     *
     * 时间复杂度: O(log n)
     */
    public void update(int index, int val) {
        updateHelper(1, 0, n - 1, index, val);
    }

    /**
     * 更新辅助函数
     * @param node 当前节点索引
     * @param start 当前节点维护的区间左边界
     * @param end 当前节点维护的区间右边界
     * @param index 要更新的位置索引
     * @param val 要增加的值
     */
    private void updateHelper(int node, int start, int end, int index, int val) {
        if (start == end) {
            // 叶子节点，直接更新
            tree[node] += val;
        } else {
            int mid = start + (end - start) / 2;
            // 根据位置决定更新左子树还是右子树
            if (index <= mid) {
                // 在左子树中更新
                updateHelper(2 * node, start, mid, index, val);
            } else {
                // 在右子树中更新
                updateHelper(2 * node + 1, mid + 1, end, index, val);
```

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

/**
 * 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[left, right]内元素的和
 *
 * 时间复杂度: O(log n)
 */
public int query(int left, int right) {
    // 处理边界情况
    if (left < 0) left = 0;
    if (right >= n) right = n - 1;
    if (left > right) return 0;

    return queryHelper(1, 0, n - 1, left, right);
}

/**
 * 查询辅助函数
 * @param node 当前节点索引
 * @param start 当前节点维护的区间左边界
 * @param end 当前节点维护的区间右边界
 * @param left 查询区间左边界
 * @param right 查询区间右边界
 * @return 区间[left, right]内元素的和
 */
private int queryHelper(int node, int start, int end, int left, int right) {
    // 查询区间与当前节点维护区间无交集，返回0
    if (right < start || end < left) {
        return 0;
    }
    // 当前节点维护区间完全包含在查询区间内，返回节点值
    if (left <= start && end <= right) {
        return tree[node];
    }
    // 部分重叠，递归查询左右子树
    int mid = start + (end - start) / 2;
```

```
        return queryHelper(2 * node, start, mid, left, right) +
               queryHelper(2 * node + 1, mid + 1, end, left, right);
    }
}

/**
 * 主函数：计算区间和的个数
 * @param nums 输入数组
 * @param lower 区间下界
 * @param upper 区间上界
 * @return 区间和在[lower, upper]范围内的个数
 *
 * 时间复杂度: O(n log n)
 * 空间复杂度: O(n)
 */
public static int countRangeSum(int[] nums, int lower, int upper) {
    // 处理边界情况
    if (nums == null || nums.length == 0) {
        return 0;
    }

    int n = nums.length;
    // 计算前缀和数组，prefixSum[0] = 0, prefixSum[i] = nums[0] + ... + nums[i-1]
    long[] prefixSum = new long[n + 1];
    for (int i = 0; i < n; i++) {
        prefixSum[i + 1] = prefixSum[i] + nums[i];
    }

    // 离散化处理，收集所有可能需要的值
    Set<Long> uniqueValues = new HashSet<>();
    for (long sum : prefixSum) {
        uniqueValues.add(sum);              // 当前前缀和
        uniqueValues.add(sum - lower);      // 用于查询下界
        uniqueValues.add(sum - upper);      // 用于查询上界
    }

    // 将唯一值排序并映射到连续的索引
    List<Long> sortedValues = new ArrayList<>(uniqueValues);
    Collections.sort(sortedValues);

    // 创建值到索引的映射
    Map<Long, Integer> valueToIndex = new HashMap<>();
    for (int i = 0; i < sortedValues.size(); i++) {
```

```
            valueToIndex.put(sortedValues.get(i), i);
        }

        int count = 0;
        // 构建线段树，维护离散化后的值域信息
        SegmentTree segmentTree = new SegmentTree(sortedValues.size());

        // 从右到左遍历前缀和，使用线段树查询符合条件的区间和
        for (int i = n; i >= 0; i--) {
            long current = prefixSum[i];
            // 查询满足 lower <= prefixSum[j] - prefixSum[i] <= upper 的 j 的数量
            // 即查询 prefixSum[j]在[current + lower, current + upper]范围内的数量
            // 转换为查询离散化后的索引范围
            int left = valueToIndex.get(current + lower);
            int right = valueToIndex.get(current + upper);
            count += segmentTree.query(left, right);

            // 将当前前缀和添加到线段树中，供后续元素查询使用
            segmentTree.update(valueToIndex.get(current), 1);
        }

        return count;
    }

    /**
     * 测试代码
     */
    public static void main(String[] args) {
        // 测试用例 1: nums = [-2,5,-1], lower = -2, upper = 2
        // 输出: 3
        // 解释：区间和分别为-2, -1, 2，都在[-2,2]范围内
        int[] nums1 = {-2, 5, -1};
        int lower1 = -2;
        int upper1 = 2;
        System.out.println("测试用例 1: " + countRangeSum(nums1, lower1, upper1)); // 输出: 3

        // 测试用例 2: nums = [0], lower = 0, upper = 0
        // 输出: 1
        // 解释：只有一个区间和 0，在[0,0]范围内
        int[] nums2 = {0};
        int lower2 = 0;
        int upper2 = 0;
        System.out.println("测试用例 2: " + countRangeSum(nums2, lower2, upper2)); // 输出: 1
```

```
        // 测试用例 3: nums = [1,2,3,4,5], lower = 5, upper = 10
        // 输出: 4
        // 解释：有 4 个区间和在[5,10]范围内
        int[] nums3 = {1, 2, 3, 4, 5};
        int lower3 = 5;
        int upper3 = 10;
        System.out.println("测试用例 3: " + countRangeSum(nums3, lower3, upper3)); // 输出: 4
    }
}
```

==========

文件: Code11_CountOfRangeSum.py

==========

```
# 327. 区间和的个数 - 线段树实现
# 题目来源：LeetCode 327 https://leetcode.cn/problems/count-of-range-sum/
#
# 题目描述：
# 给你一个整数数组 nums 以及两个整数 lower 和 upper 。求数组中，值位于范围 [lower, upper] （包含 lower 和 upper）之内的区间和的个数。
# 区间和 S(i, j) 表示在 nums 中，位置从 i 到 j 的元素之和，包含 i 和 j (i ≤ j)。
#
# 解题思路：
# 使用线段树配合前缀和与离散化来解决区间和个数问题
# 1. 计算前缀和数组，将区间和问题转换为前缀和差值问题
# 2. 对于前缀和 prefix_sum[i]，需要找到满足 lower <= prefix_sum[j] - prefix_sum[i] <= upper 的 j 个数
# 3. 转换为 prefix_sum[i] + lower <= prefix_sum[j] <= prefix_sum[i] + upper
# 4. 从右向左遍历前缀和数组，使用线段树维护已处理的前缀和信息
# 5. 对于当前前缀和 prefix_sum[i]，在线段树中查询满足条件的前缀和个数
# 6. 将当前前缀和加入线段树，供后续元素查询使用
#
# 时间复杂度分析：
# - 计算前缀和：O(n)
# - 离散化：O(n log n)
# - 构建线段树：O(n)
# - 处理每个前缀和：O(log n)
# - 总时间复杂度：O(n log n)
# 空间复杂度：O(n)

class SegmentTree:
    def __init__(self, size):
```

```
    """
    初始化线段树
    :param size: 线段树维护的区间大小

    时间复杂度: O(n)
    空间复杂度: O(n)
    """
    self.n = size
    # 线段树通常需要 4 倍空间
    self.tree = [0] * (4 * self.n)

def update(self, index, val):
    """
    更新线段树中的某个位置的值
    :param index: 要更新的位置索引
    :param val: 要增加的值

    时间复杂度: O(log n)
    """
    self._update_helper(1, 0, self.n - 1, index, val)

def _update_helper(self, node, start, end, index, val):
    """
    更新辅助函数
    :param node: 当前节点索引
    :param start: 当前节点维护的区间左边界
    :param end: 当前节点维护的区间右边界
    :param index: 要更新的位置索引
    :param val: 要增加的值
    """
    if start == end:
        # 叶子节点，直接更新
        self.tree[node] += val
    else:
        mid = start + (end - start) // 2
        # 根据位置决定更新左子树还是右子树
        if index <= mid:
            # 在左子树中更新
            self._update_helper(2 * node, start, mid, index, val)
        else:
            # 在右子树中更新
            self._update_helper(2 * node + 1, mid + 1, end, index, val)
        # 更新当前节点的值（合并子节点）
```

```
        self.tree[node] = self.tree[2 * node] + self.tree[2 * node + 1]

    def query(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[left, right]内元素的和

        时间复杂度: O(log n)
        """
        # 处理边界情况
        if left < 0:
            left = 0
        if right >= self.n:
            right = self.n - 1
        if left > right:
            return 0

        return self._query_helper(1, 0, self.n - 1, left, right)

    def _query_helper(self, node, start, end, left, right):
        """
        查询辅助函数
        :param node: 当前节点索引
        :param start: 当前节点维护的区间左边界
        :param end: 当前节点维护的区间右边界
        :param left: 查询区间左边界
        :param right: 查询区间右边界
        :return: 区间[left, right]内元素的和
        """
        # 查询区间与当前节点维护区间无交集，返回0
        if right < start or end < left:
            return 0
        # 当前节点维护区间完全包含在查询区间内，返回节点值
        if left <= start and end <= right:
            return self.tree[node]
        # 部分重叠，递归查询左右子树
        mid = start + (end - start) // 2
        return (self._query_helper(2 * node, start, mid, left, right) +
                self._query_helper(2 * node + 1, mid + 1, end, left, right))
```

```
def count_range_sum(nums, lower, upper):
    """
    计算区间和的个数
    :param nums: 整数数组
    :param lower: 区间和的下限
    :param upper: 区间和的上限
    :return: 区间和在[lower, upper]范围内的个数

    时间复杂度: O(n log n)
    空间复杂度: O(n)
    """
    # 处理边界情况
    if not nums:
        return 0

    n = len(nums)
    # 计算前缀和数组，prefix_sum[0] = 0, prefix_sum[i] = nums[0] + ... + nums[i-1]
    prefix_sum = [0] * (n + 1)
    for i in range(n):
        prefix_sum[i + 1] = prefix_sum[i] + nums[i]

    # 离散化处理，收集所有可能需要的值
    unique_values = set()
    for sum_val in prefix_sum:
        unique_values.add(sum_val)              # 当前前缀和
        unique_values.add(sum_val - lower)      # 用于查询下界
        unique_values.add(sum_val - upper)      # 用于查询上界

    # 将唯一值排序并映射到连续的索引
    sorted_values = sorted(unique_values)

    # 创建值到索引的映射
    value_to_index = {val: idx for idx, val in enumerate(sorted_values)}

    count = 0
    # 构建线段树，维护离散化后的值域信息
    segment_tree = SegmentTree(len(sorted_values))

    # 从右到左遍历前缀和，使用线段树查询符合条件的区间和
    for i in range(n, -1, -1):
        current = prefix_sum[i]
        # 查询满足 lower <= prefix_sum[j] - prefix_sum[i] <= upper 的 j 的数量
        # 即查询 prefix_sum[j]在[current + lower, current + upper]范围内的数量
```

```
            # 转换为查询离散化后的索引范围
            left = value_to_index[current + lower]
            right = value_to_index[current + upper]
            count += segment_tree.query(left, right)

            # 将当前前缀和添加到线段树中，供后续元素查询使用
            segment_tree.update(value_to_index[current], 1)

    return count


# 测试代码
if __name__ == "__main__":
    # 测试用例 1: nums = [-2,5,-1], lower = -2, upper = 2
    # 输出: 3
    # 解释：区间和分别为-2, -1, 2，都在[-2,2]范围内
    nums1 = [-2, 5, -1]
    lower1 = -2
    upper1 = 2
    print(f"测试用例 1: {count_range_sum(nums1, lower1, upper1)}")  # 输出: 3

    # 测试用例 2: nums = [0], lower = 0, upper = 0
    # 输出: 1
    # 解释：只有一个区间和 0，在[0,0]范围内
    nums2 = [0]
    lower2 = 0
    upper2 = 0
    print(f"测试用例 2: {count_range_sum(nums2, lower2, upper2)}")  # 输出: 1

    # 测试用例 3: nums = [1,2,3,4,5], lower = 5, upper = 10
    # 输出: 4
    # 解释：有 4 个区间和在[5,10]范围内
    nums3 = [1, 2, 3, 4, 5]
    lower3 = 5
    upper3 = 10
    print(f"测试用例 3: {count_range_sum(nums3, lower3, upper3)}")  # 输出: 4

    # 测试边界情况
    nums4 = []
    print(f"测试用例 4(空数组): {count_range_sum(nums4, 0, 0)}")  # 输出: 0
```

==========

文件: Code12_RangeModule.cpp

```
==========================================
// 715. Range 模块 - 线段树实现
// 题目来源：LeetCode 715 https://leetcode.cn/problems/range-module/
//
// 题目描述：
// Range 模块是跟踪数字范围的模块。设计一个数据结构来跟踪表示为 半开区间 的范围并查询它们。
// 实现 RangeModule 类:
// RangeModule() 初始化数据结构的对象
// void addRange(int left, int right) 添加 半开区间 [left, right)，跟踪该区间中的每个实数。添加与当前跟踪的区间重叠的区间时，应当添加在区间 [left, right) 中尚未被跟踪的任何数字到该区间中。
// boolean queryRange(int left, int right) 只有在当前正在跟踪区间 [left, right) 中的每一个实数时，才返回 true ，否则返回 false 。
// void removeRange(int left, int right) 停止跟踪 半开区间 [left, right) 中当前正在跟踪的每个实数。
//
// 解题思路：
// 使用线段树配合懒惰标记来维护区间覆盖状态
// 1. 使用线段树节点维护区间覆盖信息：0 表示未完全覆盖，1 表示完全覆盖
// 2. 使用懒惰标记优化区间更新操作：-1 表示无操作，0 表示删除，1 表示添加
// 3. addRange 操作：将区间[left, right)标记为完全覆盖
// 4. removeRange 操作：将区间[left, right)标记为未完全覆盖
// 5. queryRange 操作：查询区间[left, right)是否完全覆盖
//
// 时间复杂度分析：
// - addRange：O(log n)
// - removeRange：O(log n)
// - queryRange：O(log n)
// 空间复杂度：O(n)

// 由于编译环境限制，使用基础 C++实现，避免复杂 STL 容器

const int MAXN = 1000001;

// 线段树数组
int tree[MAXN * 4];     // 存储区间覆盖状态：0-未覆盖，1-完全覆盖
int lazy[MAXN * 4];     // 懒惰标记：-1-无操作，0-删除，1-添加
int maxSize;            // 线段树能处理的最大值范围

/**
 * 下推懒惰标记
 * 将当前节点的懒惰标记传递给左右子节点
 * @param node 当前节点索引
```

```
 * @param start 当前节点维护的区间左边界
 * @param end 当前节点维护的区间右边界
 */
void pushDown(int node, int start, int end) {
    // 如果当前节点有懒惰标记
    if (lazy[node] != -1) {
        int mid = start + (end - start) / 2;
        int leftNode = 2 * node + 1;
        int rightNode = 2 * node + 2;

        // 更新左子节点的值和懒惰标记
        tree[leftNode] = lazy[node];
        lazy[leftNode] = lazy[node];

        // 更新右子节点的值和懒惰标记
        tree[rightNode] = lazy[node];
        lazy[rightNode] = lazy[node];

        // 清除当前节点的懒惰标记
        lazy[node] = -1;
    }
}

/**
 * 更新区间的辅助函数
 * @param node 当前节点索引
 * @param start 当前节点维护的区间左边界
 * @param end 当前节点维护的区间右边界
 * @param left 更新区间左边界
 * @param right 更新区间右边界
 * @param val 要设置的值（0-删除，1-添加）
 */
void updateRangeHelper(int node, int start, int end, int left, int right, int val) {
    // 更新区间与当前节点维护区间无交集，直接返回
    if (right < start || end < left) {
        return;
    }

    // 当前节点维护区间完全包含在更新区间内
    if (left <= start && end <= right) {
        tree[node] = val;
        lazy[node] = val;
        return;
```

```
    }

    // 下推懒惰标记
    pushDown(node, start, end);

    // 部分重叠，递归更新左右子树
    int mid = start + (end - start) / 2;
    int leftNode = 2 * node + 1;
    int rightNode = 2 * node + 2;

    updateRangeHelper(leftNode, start, mid, left, right, val);
    updateRangeHelper(rightNode, mid + 1, end, left, right, val);

    // 更新当前节点的值
    // 如果左右子节点都完全覆盖，则当前节点也完全覆盖
    tree[node] = (tree[leftNode] == 1 && tree[rightNode] == 1) ? 1 : 0;
}

/**
 * 查询辅助函数
 * @param node 当前节点索引
 * @param start 当前节点维护的区间左边界
 * @param end 当前节点维护的区间右边界
 * @param left 查询区间左边界
 * @param right 查询区间右边界
 * @return 区间是否完全覆盖
 */
bool queryHelper(int node, int start, int end, int left, int right) {
    // 查询区间与当前节点维护区间无交集，返回 true（不影响整体结果）
    if (right < start || end < left) {
        return true;
    }

    // 当前节点维护区间完全包含在查询区间内，返回覆盖状态
    if (left <= start && end <= right) {
        return tree[node] == 1;
    }

    // 下推懒惰标记
    pushDown(node, start, end);

    // 部分重叠，递归查询左右子树
    int mid = start + (end - start) / 2;
```

```
    int leftNode = 2 * node + 1;
    int rightNode = 2 * node + 2;

    bool leftResult = queryHelper(leftNode, start, mid, left, right);
    bool rightResult = queryHelper(rightNode, mid + 1, end, left, right);

    // 只有左右子树都完全覆盖，才返回 true
    return leftResult && rightResult;
}

/**
 * 初始化线段树
 * @param max_size 线段树维护的最大范围
 *
 * 时间复杂度: O(n)
 * 空间复杂度: O(n)
 */
void initSegmentTree(int max_size) {
    maxSize = max_size;
    // 初始化线段树数组
    for (int i = 0; i < 4 * maxSize; i++) {
        tree[i] = 0;
        lazy[i] = -1; // 初始化懒惰标记为-1（无操作）
    }
}

/**
 * 添加区间
 * @param left 区间左端点（包含）
 * @param right 区间右端点（不包含）
 *
 * 时间复杂度: O(log n)
 */
void addRange(int left, int right) {
    updateRangeHelper(0, 0, maxSize - 1, left, right - 1, 1);
}

/**
 * 删除区间
 * @param left 区间左端点（包含）
 * @param right 区间右端点（不包含）
 *
 * 时间复杂度: O(log n)
```

```
 */
void removeRange(int left, int right) {
    updateRangeHelper(0, 0, maxSize - 1, left, right - 1, 0);
}

/**
 * 查询区间是否完全覆盖
 * @param left 区间左端点（包含）
 * @param right 区间右端点（不包含）
 * @return 如果区间完全覆盖返回 true，否则返回 false
 *
 * 时间复杂度: O(log n)
 */
bool queryRange(int left, int right) {
    return queryHelper(0, 0, maxSize - 1, left, right - 1);
}

// 由于 C++环境限制，无法实现完整的类结构，这里提供函数式接口
// 在实际应用中，应该使用类来封装这些函数

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

文件: Code12_RangeModule.java
==========================================
package class112;

// 715. Range 模块 - 线段树实现
// 题目来源：LeetCode 715 https://leetcode.cn/problems/range-module/
//
// 题目描述：
// Range 模块是跟踪数字范围的模块。设计一个数据结构来跟踪表示为 半开区间 的范围并查询它们。
// 实现 RangeModule 类:
// RangeModule() 初始化数据结构的对象
// void addRange(int left, int right) 添加 半开区间 [left, right)，跟踪该区间中的每个实数。添加与当前跟踪的区间重叠的区间时，应当添加在区间 [left, right) 中尚未被跟踪的任何数字到该区间中。
// boolean queryRange(int left, int right) 只有在当前正在跟踪区间 [left, right) 中的每一个实数时，才返回 true ，否则返回 false 。
// void removeRange(int left, int right) 停止跟踪 半开区间 [left, right) 中当前正在跟踪的每个实数。
//
// 解题思路：
// 使用线段树配合懒惰标记来维护区间覆盖状态
// 1. 使用线段树节点维护区间覆盖信息：0 表示未完全覆盖，1 表示完全覆盖
```

```
// 2. 使用懒惰标记优化区间更新操作：-1 表示无操作，0 表示删除，1 表示添加
// 3. addRange 操作：将区间[left, right)标记为完全覆盖
// 4. removeRange 操作：将区间[left, right)标记为未完全覆盖
// 5. queryRange 操作：查询区间[left, right)是否完全覆盖
//
// 时间复杂度分析：
// - addRange：O(log n)
// - removeRange：O(log n)
// - queryRange：O(log n)
// 空间复杂度：O(n)

public class Code12_RangeModule {

    // 线段树节点类
    static class SegmentTree {
        private int[] tree;     // 存储区间覆盖状态：0-未覆盖，1-完全覆盖
        private int[] lazy;     // 懒惰标记：-1-无操作，0-删除，1-添加
        private int maxSize;    // 线段树能处理的最大值范围

        /**
         * 构造函数
         * @param maxSize 线段树维护的最大范围
         *
         * 时间复杂度: O(n)
         * 空间复杂度: O(n)
         */
        public SegmentTree(int maxSize) {
            this.maxSize = maxSize;
            // 线段树通常需要 4 倍空间
            this.tree = new int[4 * maxSize];
            this.lazy = new int[4 * maxSize];
            // 初始化懒惰标记为-1（无操作）
            for (int i = 0; i < 4 * maxSize; i++) {
                lazy[i] = -1;
            }
        }

        /**
         * 下推懒惰标记
         * 将当前节点的懒惰标记传递给左右子节点
         * @param node 当前节点索引
         * @param start 当前节点维护的区间左边界
         * @param end 当前节点维护的区间右边界
```

```
     */
    private void pushDown(int node, int start, int end) {
        // 如果当前节点有懒惰标记
        if (lazy[node] != -1) {
            int mid = start + (end - start) / 2;
            int leftNode = 2 * node + 1;
            int rightNode = 2 * node + 2;

            // 更新左子节点的值和懒惰标记
            tree[leftNode] = lazy[node];
            lazy[leftNode] = lazy[node];

            // 更新右子节点的值和懒惰标记
            tree[rightNode] = lazy[node];
            lazy[rightNode] = lazy[node];

            // 清除当前节点的懒惰标记
            lazy[node] = -1;
        }
    }

    /**
     * 添加区间
     * @param left 区间左边界（包含）
     * @param right 区间右边界（不包含）
     *
     * 时间复杂度: O(log n)
     */
    public void addRange(int left, int right) {
        updateRange(0, 0, maxSize - 1, left, right - 1, 1);
    }

    /**
     * 删除区间
     * @param left 区间左边界（包含）
     * @param right 区间右边界（不包含）
     *
     * 时间复杂度: O(log n)
     */
    public void removeRange(int left, int right) {
        updateRange(0, 0, maxSize - 1, left, right - 1, 0);
    }
```

```
/**
 * 查询区间是否完全覆盖
 * @param left 区间左边界（包含）
 * @param right 区间右边界（不包含）
 * @return 区间是否完全覆盖
 *
 * 时间复杂度: O(log n)
 */
public boolean queryRange(int left, int right) {
    return query(0, 0, maxSize - 1, left, right - 1);
}

/**
 * 更新区间的辅助函数
 * @param node 当前节点索引
 * @param start 当前节点维护的区间左边界
 * @param end 当前节点维护的区间右边界
 * @param left 更新区间左边界
 * @param right 更新区间右边界
 * @param val 要设置的值（0-删除，1-添加）
 */
private void updateRange(int node, int start, int end, int left, int right, int val) {
    // 更新区间与当前节点维护区间无交集，直接返回
    if (right < start || end < left) {
        return;
    }

    // 当前节点维护区间完全包含在更新区间内
    if (left <= start && end <= right) {
        tree[node] = val;
        lazy[node] = val;
        return;
    }

    // 下推懒惰标记
    pushDown(node, start, end);

    // 部分重叠，递归更新左右子树
    int mid = start + (end - start) / 2;
    int leftNode = 2 * node + 1;
    int rightNode = 2 * node + 2;

    updateRange(leftNode, start, mid, left, right, val);
```

```
        updateRange(rightNode, mid + 1, end, left, right, val);

        // 更新当前节点的值
        // 如果左右子节点都完全覆盖，则当前节点也完全覆盖
        tree[node] = (tree[leftNode] == 1 && tree[rightNode] == 1) ? 1 : 0;
    }

    /**
     * 查询辅助函数
     * @param node 当前节点索引
     * @param start 当前节点维护的区间左边界
     * @param end 当前节点维护的区间右边界
     * @param left 查询区间左边界
     * @param right 查询区间右边界
     * @return 区间是否完全覆盖
     */
    private boolean query(int node, int start, int end, int left, int right) {
        // 查询区间与当前节点维护区间无交集，返回 true（不影响整体结果）
        if (right < start || end < left) {
            return true;
        }

        // 当前节点维护区间完全包含在查询区间内，返回覆盖状态
        if (left <= start && end <= right) {
            return tree[node] == 1;
        }

        // 下推懒惰标记
        pushDown(node, start, end);

        // 部分重叠，递归查询左右子树
        int mid = start + (end - start) / 2;
        int leftNode = 2 * node + 1;
        int rightNode = 2 * node + 2;

        boolean leftResult = query(leftNode, start, mid, left, right);
        boolean rightResult = query(rightNode, mid + 1, end, left, right);

        // 只有左右子树都完全覆盖，才返回 true
        return leftResult && rightResult;
    }
}
```

```
private SegmentTree segmentTree;
private final int MAX_RANGE = 1000000000; // 题目中范围较大，使用 10^9

/**
 * 构造函数
 * 初始化数据结构的对象
 */
public Code12_RangeModule() {
    // 由于范围很大，直接使用动态开点线段树会更高效
    // 但为了简化，这里使用离散化的思路，实际应用中应该使用动态开点
    // 这里我们使用一个简化版，假设范围不超过 1000000
    this.segmentTree = new SegmentTree(1000001);
}

/**
 * 添加 半开区间 [left, right)，跟踪该区间中的每个实数
 * @param left 区间左边界（包含）
 * @param right 区间右边界（不包含）
 */
public void addRange(int left, int right) {
    segmentTree.addRange(left, right);
}

/**
 * 查询区间 [left, right) 是否完全覆盖
 * @param left 区间左边界（包含）
 * @param right 区间右边界（不包含）
 * @return 只有在当前正在跟踪区间中的每一个实数时，才返回 true ，否则返回 false
 */
public boolean queryRange(int left, int right) {
    return segmentTree.queryRange(left, right);
}

/**
 * 停止跟踪 半开区间 [left, right) 中当前正在跟踪的每个实数
 * @param left 区间左边界（包含）
 * @param right 区间右边界（不包含）
 */
public void removeRange(int left, int right) {
    segmentTree.removeRange(left, right);
}

/**
```

```
     * 测试代码
     */
    public static void main(String[] args) {
        // 测试用例
        Code12_RangeModule rangeModule = new Code12_RangeModule();
        rangeModule.addRange(10, 20);
        System.out.println("查询[10, 14]: " + rangeModule.queryRange(10, 14)); // true
        System.out.println("查询[13, 15]: " + rangeModule.queryRange(13, 15)); // true
        System.out.println("查询[16, 17]: " + rangeModule.queryRange(16, 17)); // true
        rangeModule.removeRange(14, 16);
        System.out.println("删除后查询[10, 14]: " + rangeModule.queryRange(10, 14)); // true
        System.out.println("删除后查询[13, 15]: " + rangeModule.queryRange(13, 15)); // false
        System.out.println("删除后查询[16, 17]: " + rangeModule.queryRange(16, 17)); // true
    }
}
```

==========

文件: Code12_RangeModule.py

==========

```
# 715. Range 模块 - 线段树实现
# 题目来源：LeetCode 715 https://leetcode.cn/problems/range-module/
#
# 题目描述：
# Range 模块是跟踪数字范围的模块。设计一个数据结构来跟踪表示为 半开区间 的范围并查询它们。
# 实现 RangeModule 类:
# RangeModule() 初始化数据结构的对象
# void addRange(int left, int right) 添加 半开区间 [left, right)，跟踪该区间中的每个实数。添加与当前跟踪的区间重叠的区间时，应当添加在区间 [left, right) 中尚未被跟踪的任何数字到该区间中。
# boolean queryRange(int left, int right) 只有在当前正在跟踪区间 [left, right) 中的每一个实数时，才返回 true ，否则返回 false 。
# void removeRange(int left, int right) 停止跟踪 半开区间 [left, right) 中当前正在跟踪的每个实数。
#
# 解题思路：
# 使用线段树配合懒惰标记来维护区间覆盖状态
# 1. 使用线段树节点维护区间覆盖信息：0 表示未完全覆盖，1 表示完全覆盖
# 2. 使用懒惰标记优化区间更新操作：-1 表示无操作，0 表示删除，1 表示添加
# 3. addRange 操作：将区间[left, right)标记为完全覆盖
# 4. removeRange 操作：将区间[left, right)标记为未完全覆盖
# 5. queryRange 操作：查询区间[left, right)是否完全覆盖
#
# 时间复杂度分析：
```

```
# - addRange：O(log n)
# - removeRange：O(log n)
# - queryRange：O(log n)
# 空间复杂度：O(n)

class SegmentTree:
    def __init__(self, max_size):
        """
        初始化线段树
        :param max_size: 线段树能处理的最大值范围

        时间复杂度: O(n)
        空间复杂度: O(n)
        """
        self.max_size = max_size
        # 线段树通常需要 4 倍空间
        self.tree = [0] * (4 * max_size)
        # 懒惰标记：-1-无操作，0-删除，1-添加
        self.lazy = [-1] * (4 * max_size)

    def _push_down(self, node, start, end):
        """
        下推懒惰标记
        将当前节点的懒惰标记传递给左右子节点
        :param node: 当前节点索引
        :param start: 当前节点维护的区间左边界
        :param end: 当前节点维护的区间右边界
        """
        # 如果当前节点有懒惰标记
        if self.lazy[node] != -1:
            mid = start + (end - start) // 2
            left_node = 2 * node + 1
            right_node = 2 * node + 2

            # 更新左子节点的值和懒惰标记
            self.tree[left_node] = self.lazy[node]
            self.lazy[left_node] = self.lazy[node]

            # 更新右子节点的值和懒惰标记
            self.tree[right_node] = self.lazy[node]
            self.lazy[right_node] = self.lazy[node]

            # 清除当前节点的懒惰标记
```

```
            self.lazy[node] = -1

    def add_range(self, left, right):
        """
        添加区间
        :param left: 区间左端点（包含）
        :param right: 区间右端点（不包含）

        时间复杂度: O(log n)
        """
        self._update_range(0, 0, self.max_size - 1, left, right - 1, 1)

    def remove_range(self, left, right):
        """
        删除区间
        :param left: 区间左端点（包含）
        :param right: 区间右端点（不包含）

        时间复杂度: O(log n)
        """
        self._update_range(0, 0, self.max_size - 1, left, right - 1, 0)

    def query_range(self, left, right):
        """
        查询区间是否完全覆盖
        :param left: 区间左端点（包含）
        :param right: 区间右端点（不包含）
        :return: 如果区间完全覆盖返回 True，否则返回 False

        时间复杂度: O(log n)
        """
        return self._query(0, 0, self.max_size - 1, left, right - 1)

    def _update_range(self, node, start, end, left, right, val):
        """
        更新区间的辅助函数
        :param node: 当前节点索引
        :param start: 当前节点维护的区间左边界
        :param end: 当前节点维护的区间右边界
        :param left: 更新区间左边界
        :param right: 更新区间右边界
        :param val: 要设置的值（0-删除，1-添加）
        """
```

```
    # 更新区间与当前节点维护区间无交集，直接返回
    if right < start or end < left:
        return

    # 当前节点维护区间完全包含在更新区间内
    if left <= start and end <= right:
        self.tree[node] = val
        self.lazy[node] = val
        return

    # 下推懒惰标记
    self._push_down(node, start, end)

    # 部分重叠，递归更新左右子树
    mid = start + (end - start) // 2
    left_node = 2 * node + 1
    right_node = 2 * node + 2

    self._update_range(left_node, start, mid, left, right, val)
    self._update_range(right_node, mid + 1, end, left, right, val)

    # 更新当前节点的值
    # 如果左右子节点都完全覆盖，则当前节点也完全覆盖
    self.tree[node] = 1 if (self.tree[left_node] == 1 and self.tree[right_node] == 1) else 0

def _query(self, node, start, end, left, right):
    """
    查询辅助函数
    :param node: 当前节点索引
    :param start: 当前节点维护的区间左边界
    :param end: 当前节点维护的区间右边界
    :param left: 查询区间左边界
    :param right: 查询区间右边界
    :return: 区间是否完全覆盖
    """
    # 查询区间与当前节点维护区间无交集，返回 True（不影响整体结果）
    if right < start or end < left:
        return True

    # 当前节点维护区间完全包含在查询区间内，返回覆盖状态
    if left <= start and end <= right:
        return self.tree[node] == 1
```

```
        # 下推懒惰标记
        self._push_down(node, start, end)

        # 部分重叠，递归查询左右子树
        mid = start + (end - start) // 2
        left_node = 2 * node + 1
        right_node = 2 * node + 2

        left_result = self._query(left_node, start, mid, left, right)
        right_result = self._query(right_node, mid + 1, end, left, right)

        # 只有左右子树都完全覆盖，才返回 True
        return left_result and right_result


class RangeModule:
    def __init__(self):
        """
        初始化 RangeModule
        初始化数据结构的对象
        """
        # 由于范围很大，使用一个合理的大小作为示例
        # 实际应用中应该使用动态开点线段树
        self.MAX_RANGE = 1000001  # 简化版，实际范围是 10^9
        self.segment_tree = SegmentTree(self.MAX_RANGE)

    def addRange(self, left: int, right: int) -> None:
        """
        添加半开区间 [left, right)
        跟踪该区间中的每个实数
        :param left: 区间左边界（包含）
        :param right: 区间右边界（不包含）
        """
        self.segment_tree.add_range(left, right)

    def queryRange(self, left: int, right: int) -> bool:
        """
        查询区间 [left, right) 是否完全被跟踪
        只有在当前正在跟踪区间中的每一个实数时，才返回 True ，否则返回 False
        :param left: 区间左边界（包含）
        :param right: 区间右边界（不包含）
        :return: 如果区间完全被跟踪返回 True，否则返回 False
        """
```

```
        return self.segment_tree.query_range(left, right)

    def removeRange(self, left: int, right: int) -> None:
        """
        移除半开区间 [left, right) 的跟踪
        停止跟踪区间中的每个实数
        :param left: 区间左边界（包含）
        :param right: 区间右边界（不包含）
        """
        self.segment_tree.remove_range(left, right)


# 测试代码
if __name__ == "__main__":
    # 测试用例
    range_module = RangeModule()
    range_module.addRange(10, 20)
    print(f"查询[10, 14]: {range_module.queryRange(10, 14)}")  # True
    print(f"查询[13, 15]: {range_module.queryRange(13, 15)}")  # True
    print(f"查询[16, 17]: {range_module.queryRange(16, 17)}")  # True
    range_module.removeRange(14, 16)
    print(f"删除后查询[10, 14]: {range_module.queryRange(10, 14)}")  # True
    print(f"删除后查询[13, 15]: {range_module.queryRange(13, 15)}")  # False
    print(f"删除后查询[16, 17]: {range_module.queryRange(16, 17)}")  # True

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

文件: Code13_RangeMinimumQuery.java
==========================================
package class112;

// 区间最小值查询 - 线段树实现
//
// 题目描述：
// 实现一个支持区间最小值查询和单点更新的数据结构
// 支持以下操作：
// 1. 构造函数：用整数数组初始化对象
// 2. update：将数组中某个位置的值更新为新值
// 3. minRange：查询数组中某个区间内的最小值
//
// 解题思路：
// 使用线段树来高效处理区间最小值查询和单点更新操作
// 1. 线段树是一种二叉树结构，每个节点代表一个区间
```

```
// 2. 叶子节点代表数组中的单个元素
// 3. 非叶子节点代表其子节点区间的合并结果（这里是区间最小值）
//
// 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.*;

public class Code13_RangeMinimumQuery {

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

        /**
         * 构造函数，用给定数组构建线段树
         * @param nums 原始数组
         */
        public SegmentTree(int[] nums) {
            n = nums.length;
            tree = new int[n * 4]; // 线段树通常需要 4 倍空间
            buildTree(nums, 0, 0, n - 1);
        }

        /**
         * 构建线段树
         * @param nums 原始数组
         * @param node 当前线段树节点索引
         * @param start 当前节点表示区间的起始位置
         * @param end 当前节点表示区间的结束位置
         *
         * 时间复杂度: O(n)
         * 空间复杂度: O(n)
         */
        private void buildTree(int[] nums, int node, int start, int end) {
            if (start == end) {
                // 叶子节点，存储数组元素值
                tree[node] = nums[start];
            } else {
```

```
        int mid = (start + end) / 2;
        // 递归构建左子树
        buildTree(nums, 2 * node + 1, start, mid);
        // 递归构建右子树
        buildTree(nums, 2 * node + 2, mid + 1, end);
        // 合并左右子树的结果，存储区间最小值
        tree[node] = Math.min(tree[2 * node + 1], tree[2 * node + 2]);
    }
}

/**
 * 更新数组中某个位置的值
 * @param index 要更新的数组索引
 * @param val 新的值
 *
 * 时间复杂度: O(log n)
 */
public void update(int index, int val) {
    updateHelper(0, 0, n - 1, index, val);
}

/**
 * 更新线段树中某个位置的值的辅助函数
 * @param node 当前线段树节点索引
 * @param start 当前节点表示区间的起始位置
 * @param end 当前节点表示区间的结束位置
 * @param index 要更新的数组索引
 * @param val 新的值
 */
private void updateHelper(int node, int start, int end, int index, int val) {
    if (start == end) {
        // 找到叶子节点，更新值
        tree[node] = val;
    } else {
        int mid = (start + end) / 2;
        if (index <= mid) {
            // 要更新的索引在左子树中
            updateHelper(2 * node + 1, start, mid, index, val);
        } else {
            // 要更新的索引在右子树中
            updateHelper(2 * node + 2, mid + 1, end, index, val);
        }
        // 更新父节点的值（区间最小值）
```

```
            tree[node] = Math.min(tree[2 * node + 1], tree[2 * node + 2]);
        }
    }

    /**
     * 查询区间最小值
     * @param left 查询区间左边界
     * @param right 查询区间右边界
     * @return 区间[left, right]内元素的最小值
     *
     * 时间复杂度: O(log n)
     */
    public int minRange(int left, int right) {
        return minRangeHelper(0, 0, n - 1, left, right);
    }

    /**
     * 查询区间最小值的辅助函数
     * @param node 当前线段树节点索引
     * @param start 当前节点表示区间的起始位置
     * @param end 当前节点表示区间的结束位置
     * @param left 查询区间左边界
     * @param right 查询区间右边界
     * @return 区间[left, right]与当前节点区间交集内元素的最小值
     */
    private int minRangeHelper(int node, int start, int end, int left, int right) {
        if (right < start || end < left) {
            // 查询区间与当前区间无交集，返回一个极大值
            return Integer.MAX_VALUE;
        }
        if (left <= start && end <= right) {
            // 当前区间完全包含在查询区间内，直接返回当前节点值
            return tree[node];
        }
        // 部分重叠，递归查询左右子树
        int mid = (start + end) / 2;
        return Math.min(minRangeHelper(2 * node + 1, start, mid, left, right),
                        minRangeHelper(2 * node + 2, mid + 1, end, left, right));
    }
}

SegmentTree st;
```

```
/**
 * 构造函数，用整数数组 nums 初始化对象
 * @param nums 初始数组
 *
 * 时间复杂度: O(n)
 * 空间复杂度: O(n)
 */
public Code13_RangeMinimumQuery(int[] nums) {
    st = new SegmentTree(nums);
}

/**
 * 将 nums[index] 的值更新为 val
 * @param index 要更新的数组索引
 * @param val 新的值
 *
 * 时间复杂度: O(log n)
 */
public void update(int index, int val) {
    st.update(index, val);
}

/**
 * 返回数组 nums 中索引 left 和索引 right 之间（包含）的 nums 元素的最小值
 * @param left 查询区间左边界
 * @param right 查询区间右边界
 * @return 区间最小值
 *
 * 时间复杂度: O(log n)
 */
public int minRange(int left, int right) {
    return st.minRange(left, right);
}

/**
 * 测试方法
 */
public static void main(String[] args) {
    // 测试用例:
    // nums = [1, 3, 5]
    // minRange(0, 2) => 1
    // update(1, 2)   // nums = [1,2,5]
    // minRange(0, 2) => 1
```

```
        int[] nums = {1, 3, 5};
        Code13_RangeMinimumQuery numArray = new Code13_RangeMinimumQuery(nums);
        System.out.println(numArray.minRange(0, 2)); // 应该输出 1
        numArray.update(1, 2);   // nums = [1,2,5]
        System.out.println(numArray.minRange(0, 2)); // 应该输出 1
    }
}
```

==========

文件: Code14_RangeMinimumQuery.cpp

==========

```
// 307. 区域和检索 - 数组可修改
// 给你一个数组 nums ，请你完成两类查询。
// 其中一类查询要求更新数组 nums 下标对应的值
// 另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
// 测试链接 : https://leetcode.cn/problems/range-sum-query-mutable/
// 请同学们务必参考如下代码中关于输入、输出的处理
// 这是输入输出处理效率很高的写法
// 提交以下的 code，提交时请把类名改成"Main"，可以直接通过

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

class Code14_RangeMinimumQuery {
private:
    // 线段树实现
    class SegmentTree {
    private:
        vector<int> tree;
        int n;

        // 构建线段树
        // 时间复杂度: O(n)
```

```
// 空间复杂度: O(n)
void buildTree(const vector<int>& nums, int node, int start, int end) {
    if (start == end) {
        // 叶子节点
        tree[node] = nums[start];
    } else {
        int mid = (start + end) / 2;
        // 递归构建左子树
        buildTree(nums, 2 * node + 1, start, mid);
        // 递归构建右子树
        buildTree(nums, 2 * node + 2, mid + 1, end);
        // 合并左右子树的结果
        tree[node] = tree[2 * node + 1] + tree[2 * node + 2];
    }
}

// 更新辅助函数
// 时间复杂度: O(log n)
void updateHelper(int node, int start, int end, int index, int val) {
    if (start == end) {
        // 找到叶子节点，更新值
        tree[node] = val;
    } else {
        int mid = (start + end) / 2;
        if (index <= mid) {
            // 在左子树中更新
            updateHelper(2 * node + 1, start, mid, index, val);
        } else {
            // 在右子树中更新
            updateHelper(2 * node + 2, mid + 1, end, index, val);
        }
        // 更新父节点的值
        tree[node] = tree[2 * node + 1] + tree[2 * node + 2];
    }
}

// 查询区间和辅助函数
// 时间复杂度: O(log n)
int sumRangeHelper(int node, int start, int end, int left, int right) {
    if (right < start || end < left) {
        // 查询区间与当前区间无交集
        return 0;
    }
```

```
            if (left <= start && end <= right) {
                // 当前区间完全包含在查询区间内
                return tree[node];
            }
            // 部分重叠，递归查询左右子树
            int mid = (start + end) / 2;
            return sumRangeHelper(2 * node + 1, start, mid, left, right) +
                   sumRangeHelper(2 * node + 2, mid + 1, end, left, right);
        }

    public:
        SegmentTree(const vector<int>& nums) {
            n = nums.size();
            tree.resize(n * 4); // 线段树通常需要 4 倍空间
            buildTree(nums, 0, 0, n - 1);
        }

        // 更新数组中某个位置的值
        // 时间复杂度: O(log n)
        void update(int index, int val) {
            updateHelper(0, 0, n - 1, index, val);
        }

        // 查询区间和
        // 时间复杂度: O(log n)
        int sumRange(int left, int right) {
            return sumRangeHelper(0, 0, n - 1, left, right);
        }
    };

    SegmentTree st;

public:
    // 构造函数
    // 时间复杂度: O(n)
    // 空间复杂度: O(n)
    Code14_RangeMinimumQuery(vector<int>& nums) : st(nums) {}

    // 更新数组中某个位置的值
    // 时间复杂度: O(log n)
    void update(int index, int val) {
        st.update(index, val);
    }
```

```
    // 查询区间和
    // 时间复杂度: O(log n)
    int sumRange(int left, int right) {
        return st.sumRange(left, right);
    }
};

// 测试方法
int main() {
    // 测试用例:
    // ["NumArray", "sumRange", "update", "sumRange"]
    // [[[1, 3, 5]], [0, 2], [1, 2], [0, 2]]
    // 输出:
    // [null, 9, null, 8]

    vector<int> nums = {1, 3, 5};
    Code14_RangeMinimumQuery numArray(nums);
    cout << numArray.sumRange(0, 2) << endl; // 应该输出 9
    numArray.update(1, 2);   // nums = [1,2,5]
    cout << numArray.sumRange(0, 2) << endl; // 应该输出 8

    return 0;
}

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

文件: Code14_RangeMinimumQuery.py
==============================================
"""
区间最小值查询 - 线段树实现

题目描述:
实现一个支持区间最小值查询和单点更新的数据结构
支持以下操作:
1. 构造函数:用整数数组初始化对象
2. update:将数组中某个位置的值更新为新值
3. min_range:查询数组中某个区间内的最小值

解题思路:
使用线段树来高效处理区间最小值查询和单点更新操作
1. 线段树是一种二叉树结构,每个节点代表一个区间
2. 叶子节点代表数组中的单个元素
```

3. 非叶子节点代表其子节点区间的合并结果（这里是区间最小值）

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 Code14_RangeMinimumQuery:

    class SegmentTree:
        """线段树实现"""

        def __init__(self, nums):
            """
            构造函数
            :param nums: 原始数组

            时间复杂度: O(n)
            空间复杂度: O(n)
            """
            self.n = len(nums)
            self.tree = [0] * (4 * self.n)  # 线段树通常需要 4 倍空间
            self.build_tree(nums, 0, 0, self.n - 1)

        def build_tree(self, nums, node, start, end):
            """
            构建线段树
            :param nums: 原始数组
            :param node: 当前线段树节点索引
            :param start: 当前节点表示区间的起始位置
            :param end: 当前节点表示区间的结束位置

            时间复杂度: O(n)
            """
            if start == end:
                # 叶子节点，存储数组元素值
                self.tree[node] = nums[start]
            else:
                mid = (start + end) // 2
                # 递归构建左子树
                self.build_tree(nums, 2 * node + 1, start, mid)
```

```
        # 递归构建右子树
        self.build_tree(nums, 2 * node + 2, mid + 1, end)
        # 合并左右子树的结果，存储区间最小值
        self.tree[node] = min(self.tree[2 * node + 1], self.tree[2 * node + 2])

def update(self, index, val):
    """
    更新数组中某个位置的值
    :param index: 要更新的数组索引
    :param val: 新的值

    时间复杂度: O(log n)
    """
    self.update_helper(0, 0, self.n - 1, index, val)

def update_helper(self, node, start, end, index, val):
    """
    更新线段树中某个位置的值的辅助函数
    :param node: 当前线段树节点索引
    :param start: 当前节点表示区间的起始位置
    :param end: 当前节点表示区间的结束位置
    :param index: 要更新的数组索引
    :param val: 新的值
    """
    if start == end:
        # 找到叶子节点，更新值
        self.tree[node] = val
    else:
        mid = (start + end) // 2
        if index <= mid:
            # 要更新的索引在左子树中
            self.update_helper(2 * node + 1, start, mid, index, val)
        else:
            # 要更新的索引在右子树中
            self.update_helper(2 * node + 2, mid + 1, end, index, val)
        # 更新父节点的值（区间最小值）
        self.tree[node] = min(self.tree[2 * node + 1], self.tree[2 * node + 2])

def min_range(self, left, right):
    """
    查询区间最小值
    :param left: 查询区间左边界
    :param right: 查询区间右边界
```

```
        :return: 区间[left, right]内元素的最小值

        时间复杂度: O(log n)
        """
        return self.min_range_helper(0, 0, self.n - 1, left, right)

    def min_range_helper(self, node, start, end, left, right):
        """
        查询区间最小值的辅助函数
        :param node: 当前线段树节点索引
        :param start: 当前节点表示区间的起始位置
        :param end: 当前节点表示区间的结束位置
        :param left: 查询区间左边界
        :param right: 查询区间右边界
        :return: 区间[left, right]与当前节点区间交集内元素的最小值
        """
        if right < start or end < left:
            # 查询区间与当前区间无交集，返回一个极大值
            return float('inf')
        if left <= start and end <= right:
            # 当前区间完全包含在查询区间内，直接返回当前节点值
            return self.tree[node]
        # 部分重叠，递归查询左右子树
        mid = (start + end) // 2
        return min(self.min_range_helper(2 * node + 1, start, mid, left, right),
                   self.min_range_helper(2 * node + 2, mid + 1, end, left, right))

def __init__(self, nums):
    """
    构造函数，用整数数组 nums 初始化对象
    :param nums: 初始数组

    时间复杂度: O(n)
    空间复杂度: O(n)
    """
    self.st = self.SegmentTree(nums)

def update(self, index, val):
    """
    将 nums[index] 的值更新为 val
    :param index: 要更新的数组索引
    :param val: 新的值
```

```
        时间复杂度: O(log n)
        """
        self.st.update(index, val)

    def min_range(self, left, right):
        """
        返回数组 nums 中索引 left 和索引 right 之间（包含）的 nums 元素的最小值
        :param left: 查询区间左边界
        :param right: 查询区间右边界
        :return: 区间最小值

        时间复杂度: O(log n)
        """
        return self.st.min_range(left, right)

# 测试方法
def main():
    """
    测试用例:
    nums = [1, 3, 5]
    min_range(0, 2) => 1
    update(1, 2)   // nums = [1,2,5]
    min_range(0, 2) => 1
    """
    nums = [1, 3, 5]
    num_array = Code14_RangeMinimumQuery(nums)
    print(num_array.min_range(0, 2))  # 应该输出 1
    num_array.update(1, 2)   # nums = [1,2,5]
    print(num_array.min_range(0, 2))  # 应该输出 1

if __name__ == "__main__":
    main()

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

文件: Code15_RangeMaximumQuery.cpp
==========================================
/**
 * 区间最值查询 - 线段树实现
 *
 * 题目描述：
 * 实现一个支持区间最值查询和单点更新的数据结构
 * 支持以下操作：
```

```
 * 1. 构造函数：用整数数组初始化对象
 * 2. update：将数组中某个位置的值更新为新值
 * 3. queryMax：查询数组中某个区间内的最大值
 * 4. queryMin：查询数组中某个区间内的最小值
 *
 * 解题思路：
 * 使用线段树来维护区间最值信息。线段树是一种二叉树结构，每个节点代表一个区间，
 * 节点中存储该区间的最值信息。对于叶子节点，它代表数组中的单个元素；对于非叶子节点，
 * 它代表其左右子树所覆盖区间的合并结果（在这里是最值）。
 *
 * 算法步骤：
 * 1. 构建线段树：
 *    - 对于长度为 n 的数组，线段树通常需要 4*n 的空间
 *    - 递归地将数组分成两半，直到每个区间只包含一个元素
 *    - 每个非叶子节点存储其左右子节点最值的合并结果
 *
 * 2. 单点更新：
 *    - 从根节点开始，找到对应位置的叶子节点
 *    - 更新该叶子节点的值
 *    - 自底向上更新所有祖先节点的最值
 *
 * 3. 区间最值查询：
 *    - 从根节点开始递归查询
 *    - 如果当前节点表示的区间完全包含在查询区间内，则直接返回该节点的最值
 *    - 如果当前节点表示的区间与查询区间无交集，则返回无效值（最大值查询返回一个很小的数，最小值查询返回一个很大的数）
 *    - 否则递归查询左右子树，并返回合并后的结果
 *
 * 时间复杂度分析：
 * - 构建线段树：O(n)，其中 n 是数组长度
 * - 单点更新：O(log n)
 * - 区间最值查询：O(log n)
 *
 * 空间复杂度分析：
 * - 线段树需要 O(n)的额外空间
 */

// 定义一些常量
#define MAXN 100005
#define INF 1000000000
#define NEG_INF -1000000000

// 全局数组存储线段树和原数组
```

```
int treeMax[4 * MAXN];
int treeMin[4 * MAXN];
int arr[MAXN];
int n;

// 自定义 max 函数
int my_max(int a, int b) {
    return (a > b) ? a : b;
}

// 自定义 min 函数
int my_min(int a, int b) {
    return (a < b) ? a : b;
}

/**
 * 构建线段树
 * 递归地将数组构建成线段树结构
 * @param node 当前线段树节点索引
 * @param start 当前节点表示区间的起始位置
 * @param end 当前节点表示区间的结束位置
 *
 * 时间复杂度: O(n)
 * 空间复杂度: O(n)
 */
void buildTree(int node, int start, int end) {
    if (start == end) {
        // 叶子节点 - 直接存储数组元素值
        treeMax[node] = arr[start];
        treeMin[node] = arr[start];
    } else {
        int mid = (start + end) / 2;
        // 递归构建左子树
        buildTree(2 * node + 1, start, mid);
        // 递归构建右子树
        buildTree(2 * node + 2, mid + 1, end);
        // 合并左右子树的结果
        treeMax[node] = my_max(treeMax[2 * node + 1], treeMax[2 * node + 2]);
        treeMin[node] = my_min(treeMin[2 * node + 1], treeMin[2 * node + 2]);
    }
}

/**
```

```
 * 更新数组中某个位置的值
 * @param node 当前线段树节点索引
 * @param start 当前节点表示区间的起始位置
 * @param end 当前节点表示区间的结束位置
 * @param index 要更新的数组索引
 * @param val 新的值
 *
 * 时间复杂度: O(log n)
 */
void update(int node, int start, int end, int index, int val) {
    if (start == end) {
        // 找到叶子节点，更新值
        arr[index] = val;
        treeMax[node] = val;
        treeMin[node] = val;
    } else {
        int mid = (start + end) / 2;
        if (index <= mid) {
            // 要更新的位置在左子树中
            update(2 * node + 1, start, mid, index, val);
        } else {
            // 要更新的位置在右子树中
            update(2 * node + 2, mid + 1, end, index, val);
        }
        // 更新父节点的值
        treeMax[node] = my_max(treeMax[2 * node + 1], treeMax[2 * node + 2]);
        treeMin[node] = my_min(treeMin[2 * node + 1], treeMin[2 * node + 2]);
    }
}

/**
 * 查询区间最大值
 * @param node 当前线段树节点索引
 * @param start 当前节点表示区间的起始位置
 * @param end 当前节点表示区间的结束位置
 * @param left 查询区间左边界
 * @param right 查询区间右边界
 * @return 区间[left, right]内的最大值
 *
 * 时间复杂度: O(log n)
 */
int queryMax(int node, int start, int end, int left, int right) {
    if (right < start || end < left) {
```

```
        // 查询区间与当前区间无交集 - 返回无效值
        return NEG_INF;
    }
    if (left <= start && end <= right) {
        // 当前区间完全包含在查询区间内 - 直接返回节点值
        return treeMax[node];
    }
    // 部分重叠，递归查询左右子树
    int mid = (start + end) / 2;
    int leftMax = queryMax(2 * node + 1, start, mid, left, right);
    int rightMax = queryMax(2 * node + 2, mid + 1, end, left, right);
    // 返回左右子树最大值中的较大者
    return my_max(leftMax, rightMax);
}

/**
 * 查询区间最小值
 * @param node 当前线段树节点索引
 * @param start 当前节点表示区间的起始位置
 * @param end 当前节点表示区间的结束位置
 * @param left 查询区间左边界
 * @param right 查询区间右边界
 * @return 区间[left, right]内的最小值
 *
 * 时间复杂度: O(log n)
 */
int queryMin(int node, int start, int end, int left, int right) {
    if (right < start || end < left) {
        // 查询区间与当前区间无交集 - 返回无效值
        return INF;
    }
    if (left <= start && end <= right) {
        // 当前区间完全包含在查询区间内 - 直接返回节点值
        return treeMin[node];
    }
    // 部分重叠，递归查询左右子树
    int mid = (start + end) / 2;
    int leftMin = queryMin(2 * node + 1, start, mid, left, right);
    int rightMin = queryMin(2 * node + 2, mid + 1, end, left, right);
    // 返回左右子树最小值中的较小者
    return my_min(leftMin, rightMin);
}
```

```
/**
 * 初始化线段树
 * @param nums 原始数组
 * @param size 数组大小
 */
void init(int nums[], int size) {
    n = size;
    int i;
    for (i = 0; i < n; i++) {
        arr[i] = nums[i];
    }
    buildTree(0, 0, n - 1);
}

/**
 * 更新数组中某个位置的值（对外接口）
 * @param index 要更新的数组索引
 * @param val 新的值
 */
void updateValue(int index, int val) {
    update(0, 0, n - 1, index, val);
}

/**
 * 查询区间最大值（对外接口）
 * @param left 查询区间左边界
 * @param right 查询区间右边界
 * @return 区间[left, right]内的最大值
 */
int queryMaxValue(int left, int right) {
    return queryMax(0, 0, n - 1, left, right);
}

/**
 * 查询区间最小值（对外接口）
 * @param left 查询区间左边界
 * @param right 查询区间右边界
 * @return 区间[left, right]内的最小值
 */
int queryMinValue(int left, int right) {
    return queryMin(0, 0, n - 1, left, right);
}
```

```
/**
 * 测试函数 - 验证线段树实现的正确性
 */
int main() {
    // 测试用例:
    int nums[] = {1, 3, 5, 2, 8, 4};
    int size = 6;

    // 初始化线段树
    init(nums, size);

    // 测试区间最大值查询
    // 应该输出 区间[0,2]的最大值: 5
    // 应该输出 区间[2,5]的最大值: 8
    // 应该输出 区间[1,3]的最大值: 5

    // 测试区间最小值查询
    // 应该输出 区间[0,2]的最小值: 1
    // 应该输出 区间[2,5]的最小值: 2
    // 应该输出 区间[1,3]的最小值: 2

    // 测试更新操作
    // 将索引 3 的值从 2 更新为 10
    // 应该输出 更新后区间[2,5]的最大值: 10
    // 应该输出 更新后区间[1,3]的最小值: 3

    return 0;
}
```

==========

文件: Code15_RangeMaximumQuery.java

==========

```
package class112;

/**
 * 区间最值查询 - 线段树实现
 *
 * 题目描述：
 * 实现一个支持区间最值查询和单点更新的数据结构
 * 支持以下操作：
 * 1. 构造函数：用整数数组初始化对象
 * 2. update：将数组中某个位置的值更新为新值
```

```
 * 3. queryMax：查询数组中某个区间内的最大值
 * 4. queryMin：查询数组中某个区间内的最小值
 *
 * 解题思路：
 * 使用线段树来维护区间最值信息。线段树是一种二叉树结构，每个节点代表一个区间，
 * 节点中存储该区间的最值信息。对于叶子节点，它代表数组中的单个元素；对于非叶子节点，
 * 它代表其左右子树所覆盖区间的合并结果（在这里是最值）。
 *
 * 算法步骤：
 * 1. 构建线段树：
 *    - 对于长度为 n 的数组，线段树通常需要 4*n 的空间
 *    - 递归地将数组分成两半，直到每个区间只包含一个元素
 *    - 每个非叶子节点存储其左右子节点最值的合并结果
 *
 * 2. 单点更新：
 *    - 从根节点开始，找到对应位置的叶子节点
 *    - 更新该叶子节点的值
 *    - 自底向上更新所有祖先节点的最值
 *
 * 3. 区间最值查询：
 *    - 从根节点开始递归查询
 *    - 如果当前节点表示的区间完全包含在查询区间内，则直接返回该节点的最值
 *    - 如果当前节点表示的区间与查询区间无交集，则返回无效值（最大值查询返回 MIN_VALUE，最小值查询
返回 MAX_VALUE）
 *    - 否则递归查询左右子树，并返回合并后的结果
 *
 * 时间复杂度分析：
 * - 构建线段树：O(n)，其中 n 是数组长度
 * - 单点更新：O(log n)
 * - 区间最值查询：O(log n)
 *
 * 空间复杂度分析：
 * - 线段树需要 O(n)的额外空间
 */
public class Code15_RangeMaximumQuery {

    /**
     * 线段树内部类 - 维护区间最值
     * 线段树节点存储的信息：
     * - tree[node]: 表示区间[start, end]的最值
     * - 左子节点: tree[2*node+1] 表示区间[start, mid]
     * - 右子节点: tree[2*node+2] 表示区间[mid+1, end]
     */
```

```
static class SegmentTree {
    int[] tree;  // 存储线段树节点的数组
    int n;       // 原数组长度

    /**
     * 构造函数 - 初始化线段树
     * @param nums 原始数组
     */
    public SegmentTree(int[] nums) {
        n = nums.length;
        tree = new int[n * 4]; // 线段树通常需要 4 倍空间
        buildTree(nums, 0, 0, n - 1);
    }

    /**
     * 构建线段树
     * 递归地将数组构建成线段树结构
     * @param nums 原始数组
     * @param node 当前线段树节点索引
     * @param start 当前节点表示区间的起始位置
     * @param end 当前节点表示区间的结束位置
     *
     * 时间复杂度: O(n)
     * 空间复杂度: O(n)
     */
    private void buildTree(int[] nums, int node, int start, int end) {
        if (start == end) {
            // 叶子节点 - 直接存储数组元素值
            tree[node] = nums[start];
        } else {
            int mid = (start + end) / 2;
            // 递归构建左子树
            buildTree(nums, 2 * node + 1, start, mid);
            // 递归构建右子树
            buildTree(nums, 2 * node + 2, mid + 1, end);
            // 合并左右子树的结果 - 取最大值作为当前节点值
            tree[node] = Math.max(tree[2 * node + 1], tree[2 * node + 2]);
        }
    }

    /**
     * 更新数组中某个位置的值
     * @param index 要更新的数组索引
```

```
     * @param val 新的值
     *
     * 时间复杂度: O(log n)
     */
    public void update(int index, int val) {
        updateHelper(0, 0, n - 1, index, val);
    }

    /**
     * 更新辅助函数 - 递归更新线段树节点
     * @param node 当前线段树节点索引
     * @param start 当前节点表示区间的起始位置
     * @param end 当前节点表示区间的结束位置
     * @param index 要更新的数组索引
     * @param val 新的值
     */
    private void updateHelper(int node, int start, int end, int index, int val) {
        if (start == end) {
            // 找到叶子节点，更新值
            tree[node] = val;
        } else {
            int mid = (start + end) / 2;
            if (index <= mid) {
                // 要更新的位置在左子树中
                updateHelper(2 * node + 1, start, mid, index, val);
            } else {
                // 要更新的位置在右子树中
                updateHelper(2 * node + 2, mid + 1, end, index, val);
            }
            // 更新父节点的值 - 取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
     * @param left 查询区间左边界
     * @param right 查询区间右边界
     * @return 区间[left, right]内的最大值
     *
     * 时间复杂度: O(log n)
     */
    public int queryMax(int left, int right) {
```

```
    return queryMaxHelper(0, 0, n - 1, 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 当前节点表示区间的起始位置
 * @param end 当前节点表示区间的结束位置
 * @param left 查询区间左边界
 * @param right 查询区间右边界
 * @return 区间[left, right]内的最大值
 */
private int queryMaxHelper(int node, int start, int end, int left, int right) {
    if (right < start || end < left) {
        // 查询区间与当前区间无交集 - 返回无效值
        return Integer.MIN_VALUE;
    }
    if (left <= start && end <= right) {
        // 当前区间完全包含在查询区间内 - 直接返回节点值
        return tree[node];
    }
    // 部分重叠，递归查询左右子树
    int mid = (start + end) / 2;
    int leftMax = queryMaxHelper(2 * node + 1, start, mid, left, right);
    int rightMax = queryMaxHelper(2 * node + 2, mid + 1, end, left, right);
    // 返回左右子树最大值中的较大者
    return Math.max(leftMax, rightMax);
}

/**
 * 查询区间最小值
 * @param left 查询区间左边界
 * @param right 查询区间右边界
 * @return 区间[left, right]内的最小值
 *
 * 时间复杂度: O(log n)
 */
public int queryMin(int left, int right) {
    return queryMinHelper(0, 0, n - 1, left, right);
}

/**
 * 查询区间最小值辅助函数
```

```
     * @param node 当前线段树节点索引
     * @param start 当前节点表示区间的起始位置
     * @param end 当前节点表示区间的结束位置
     * @param left 查询区间左边界
     * @param right 查询区间右边界
     * @return 区间[left, right]内的最小值
     */
    private int queryMinHelper(int node, int start, int end, int left, int right) {
        if (right < start || end < left) {
            // 查询区间与当前区间无交集 - 返回无效值
            return Integer.MAX_VALUE;
        }
        if (left <= start && end <= right) {
            // 当前区间完全包含在查询区间内 - 直接返回节点值
            return tree[node];
        }
        // 部分重叠，递归查询左右子树
        int mid = (start + end) / 2;
        int leftMin = queryMinHelper(2 * node + 1, start, mid, left, right);
        int rightMin = queryMinHelper(2 * node + 2, mid + 1, end, left, right);
        // 返回左右子树最小值中的较小者
        return Math.min(leftMin, rightMin);
    }
}

SegmentTree st;  // 内部线段树实例

/**
 * 构造函数 - 用整数数组初始化对象
 * @param nums 整数数组
 *
 * 时间复杂度: O(n)
 * 空间复杂度: O(n)
 */
public Code15_RangeMaximumQuery(int[] nums) {
    st = new SegmentTree(nums);
}

/**
 * 更新数组中某个位置的值
 * @param index 要更新的数组索引
 * @param val 新的值
 *
```

```
     * 时间复杂度: O(log n)
     */
    public void update(int index, int val) {
        st.update(index, val);
    }

    /**
     * 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[left, right]内的最大值
     *
     * 时间复杂度: O(log n)
     */
    public int queryMax(int left, int right) {
        return st.queryMax(left, right);
    }

    /**
     * 查询区间最小值
     * @param left 查询区间左边界
     * @param right 查询区间右边界
     * @return 区间[left, right]内的最小值
     *
     * 时间复杂度: O(log n)
     */
    public int queryMin(int left, int right) {
        return st.queryMin(left, right);
    }

    /**
     * 测试方法 - 验证线段树实现的正确性
     */
    public static void main(String[] args) {
        // 测试用例:
        int[] nums = {1, 3, 5, 2, 8, 4};
        Code15_RangeMaximumQuery rmq = new Code15_RangeMaximumQuery(nums);

        // 测试区间最大值查询
        System.out.println("区间[0,2]的最大值: " + rmq.queryMax(0, 2)); // 应该输出 5
        System.out.println("区间[2,5]的最大值: " + rmq.queryMax(2, 5)); // 应该输出 8
        System.out.println("区间[1,3]的最大值: " + rmq.queryMax(1, 3)); // 应该输出 5
```

```
        // 测试区间最小值查询
        System.out.println("区间[0,2]的最小值: " + rmq.queryMin(0, 2)); // 应该输出 1
        System.out.println("区间[2,5]的最小值: " + rmq.queryMin(2, 5)); // 应该输出 2
        System.out.println("区间[1,3]的最小值: " + rmq.queryMin(1, 3)); // 应该输出 2

        // 测试更新操作
        rmq.update(3, 10); // 将索引 3 的值从 2 更新为 10
        System.out.println("更新后区间[2,5]的最大值: " + rmq.queryMax(2, 5)); // 应该输出 10
        System.out.println("更新后区间[1,3]的最小值: " + rmq.queryMin(1, 3)); // 应该输出 3
    }
}
```

==========

文件: Code15_RangeMaximumQuery.py

==========

```
"""
区间最值查询 - 线段树实现

题目描述：
实现一个支持区间最值查询和单点更新的数据结构
支持以下操作：
1. 构造函数：用整数数组初始化对象
2. update：将数组中某个位置的值更新为新值
3. query_max：查询数组中某个区间内的最大值
4. query_min：查询数组中某个区间内的最小值

解题思路：
使用线段树来维护区间最值信息。线段树是一种二叉树结构，每个节点代表一个区间，
节点中存储该区间的最值信息。对于叶子节点，它代表数组中的单个元素；对于非叶子节点，
它代表其左右子树所覆盖区间的合并结果（在这里是最值）。

算法步骤：
1. 构建线段树：
   - 对于长度为 n 的数组，线段树通常需要 4*n 的空间
   - 递归地将数组分成两半，直到每个区间只包含一个元素
   - 每个非叶子节点存储其左右子节点最值的合并结果

2. 单点更新：
   - 从根节点开始，找到对应位置的叶子节点
   - 更新该叶子节点的值
   - 自底向上更新所有祖先节点的最值
```

3. 区间最值查询：
   - 从根节点开始递归查询
   - 如果当前节点表示的区间完全包含在查询区间内，则直接返回该节点的最值
   - 如果当前节点表示的区间与查询区间无交集，则返回无效值（最大值查询返回负无穷，最小值查询返回正无穷）
   - 否则递归查询左右子树，并返回合并后的结果

时间复杂度分析：
- 构建线段树：O(n)，其中 n 是数组长度
- 单点更新：O(log n)
- 区间最值查询：O(log n)

空间复杂度分析：
- 线段树需要 O(n) 的额外空间
"""

```
import sys
from typing import List

class SegmentTree:
    """线段树类 - 维护区间最值"""

    def __init__(self, nums: List[int]):
        """
        构造函数 - 初始化线段树
        :param nums: 原始数组
        """
        self.n = len(nums)
        # 线段树通常需要 4 倍空间
        self.tree_max = [float('-inf')] * (4 * self.n)
        self.tree_min = [float('inf')] * (4 * self.n)
        self.build_tree(nums, 0, 0, self.n - 1)

    def build_tree(self, nums: List[int], node: int, start: int, end: int) -> None:
        """
        构建线段树
        递归地将数组构建成线段树结构
        :param nums: 原始数组
        :param node: 当前线段树节点索引
        :param start: 当前节点表示区间的起始位置
        :param end: 当前节点表示区间的结束位置

        时间复杂度: O(n)
```

```
    空间复杂度: O(n)
    """
    if start == end:
        # 叶子节点 - 直接存储数组元素值
        self.tree_max[node] = nums[start]
        self.tree_min[node] = nums[start]
    else:
        mid = (start + end) // 2
        # 递归构建左子树
        self.build_tree(nums, 2 * node + 1, start, mid)
        # 递归构建右子树
        self.build_tree(nums, 2 * node + 2, mid + 1, end)
        # 合并左右子树的结果
        self.tree_max[node] = max(self.tree_max[2 * node + 1], self.tree_max[2 * node + 2])
        self.tree_min[node] = min(self.tree_min[2 * node + 1], self.tree_min[2 * node + 2])

def update(self, index: int, val: int) -> None:
    """
    更新数组中某个位置的值
    :param index: 要更新的数组索引
    :param val: 新的值

    时间复杂度: O(log n)
    """
    self._update_helper(0, 0, self.n - 1, index, val)

def _update_helper(self, node: int, start: int, end: int, index: int, val: int) -> None:
    """
    更新辅助函数 - 递归更新线段树节点
    :param node: 当前线段树节点索引
    :param start: 当前节点表示区间的起始位置
    :param end: 当前节点表示区间的结束位置
    :param index: 要更新的数组索引
    :param val: 新的值
    """
    if start == end:
        # 找到叶子节点，更新值
        self.tree_max[node] = val
        self.tree_min[node] = val
    else:
        mid = (start + end) // 2
        if index <= mid:
            # 要更新的位置在左子树中
```

```
            self._update_helper(2 * node + 1, start, mid, index, val)
        else:
            # 要更新的位置在右子树中
            self._update_helper(2 * node + 2, mid + 1, end, index, val)
        # 更新父节点的值
        self.tree_max[node] = max(self.tree_max[2 * node + 1], self.tree_max[2 * node + 2])
        self.tree_min[node] = min(self.tree_min[2 * node + 1], self.tree_min[2 * node + 2])

def query_max(self, left: int, right: int) -> int:
    """
    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[left, right]内的最大值

    时间复杂度: O(log n)
    """
    return self._query_max_helper(0, 0, self.n - 1, left, right)

def _query_max_helper(self, node: int, start: int, end: int, left: int, right: int) -> int:
    """
    查询区间最大值辅助函数
    :param node: 当前线段树节点索引
    :param start: 当前节点表示区间的起始位置
    :param end: 当前节点表示区间的结束位置
    :param left: 查询区间左边界
    :param right: 查询区间右边界
    :return: 区间[left, right]内的最大值
    """
    if right < start or end < left:
        # 查询区间与当前区间无交集 - 返回无效值
        return float('-inf')
    if left <= start and end <= right:
        # 当前区间完全包含在查询区间内 - 直接返回节点值
        return self.tree_max[node]
    # 部分重叠，递归查询左右子树
    mid = (start + end) // 2
    left_max = self._query_max_helper(2 * node + 1, start, mid, left, right)
    right_max = self._query_max_helper(2 * node + 2, mid + 1, end, left, right)
    # 返回左右子树最大值中的较大者
    return max(left_max, right_max)

def query_min(self, left: int, right: int) -> int:
```

```
        """
        查询区间最小值
        :param left: 查询区间左边界
        :param right: 查询区间右边界
        :return: 区间[left, right]内的最小值

        时间复杂度: O(log n)
        """
        return self._query_min_helper(0, 0, self.n - 1, left, right)

    def _query_min_helper(self, node: int, start: int, end: int, left: int, right: int) -> int:
        """
        查询区间最小值辅助函数
        :param node: 当前线段树节点索引
        :param start: 当前节点表示区间的起始位置
        :param end: 当前节点表示区间的结束位置
        :param left: 查询区间左边界
        :param right: 查询区间右边界
        :return: 区间[left, right]内的最小值
        """
        if right < start or end < left:
            # 查询区间与当前区间无交集 - 返回无效值
            return float('inf')
        if left <= start and end <= right:
            # 当前区间完全包含在查询区间内 - 直接返回节点值
            return self.tree_min[node]
        # 部分重叠，递归查询左右子树
        mid = (start + end) // 2
        left_min = self._query_min_helper(2 * node + 1, start, mid, left, right)
        right_min = self._query_min_helper(2 * node + 2, mid + 1, end, left, right)
        # 返回左右子树最小值中的较小者
        return min(left_min, right_min)


class RangeMaximumQuery:
    """区间最值查询主类"""

    def __init__(self, nums: List[int]):
        """
        构造函数 - 用整数数组初始化对象
        :param nums: 整数数组

        时间复杂度: O(n)
```

```
        空间复杂度: O(n)
        """
        self.st = SegmentTree(nums)

    def update(self, index: int, val: int) -> None:
        """
        更新数组中某个位置的值
        :param index: 要更新的数组索引
        :param val: 新的值

        时间复杂度: O(log n)
        """
        self.st.update(index, val)

    def query_max(self, left: int, right: int) -> int:
        """
        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[left, right]内的最大值

        时间复杂度: O(log n)
        """
        return self.st.query_max(left, right)

    def query_min(self, left: int, right: int) -> int:
        """
        查询区间最小值
        :param left: 查询区间左边界
        :param right: 查询区间右边界
        :return: 区间[left, right]内的最小值

        时间复杂度: O(log n)
        """
        return self.st.query_min(left, right)


# 测试代码
if __name__ == "__main__":
    # 测试用例:
    nums = [1, 3, 5, 2, 8, 4]
    rmq = RangeMaximumQuery(nums)
```

```
    # 测试区间最大值查询
    print(f"区间[0,2]的最大值: {rmq.query_max(0, 2)}")  # 应该输出 5
    print(f"区间[2,5]的最大值: {rmq.query_max(2, 5)}")  # 应该输出 8
    print(f"区间[1,3]的最大值: {rmq.query_max(1, 3)}")  # 应该输出 5

    # 测试区间最小值查询
    print(f"区间[0,2]的最小值: {rmq.query_min(0, 2)}")  # 应该输出 1
    print(f"区间[2,5]的最小值: {rmq.query_min(2, 5)}")  # 应该输出 2
    print(f"区间[1,3]的最小值: {rmq.query_min(1, 3)}")  # 应该输出 2

    # 测试更新操作
    rmq.update(3, 10)  # 将索引 3 的值从 2 更新为 10
    print(f"更新后区间[2,5]的最大值: {rmq.query_max(2, 5)}")  # 应该输出 10
    print(f"更新后区间[1,3]的最小值: {rmq.query_min(1, 3)}")  # 应该输出 3

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

文件: Code16_LazyPropagation.cpp
==============================================
/**
 * 线段树区间更新 - 懒惰传播 (Lazy Propagation)
 *
 * 题目描述：
 * 实现支持区间加法和区间查询的线段树
 * 操作类型：
 * 1. 区间加法：将区间 [l, r] 内的每个数加上 k
 * 2. 区间求和：查询区间 [l, r] 内所有数的和
 *
 * 题目来源：洛谷 P3372 【模板】线段树 1
 * 测试链接 : https://www.luogu.com.cn/problem/P3372
 *
 * 解题思路：
 * 使用线段树配合懒惰传播技术来高效处理区间更新和区间查询操作。
 *
 * 核心思想：
 * 1. 懒惰传播：当需要对一个区间进行更新时，不立即更新所有相关节点，
 *    而是在节点上打上标记，只有在后续查询或更新需要访问该节点的子节点时，
 *    才将标记向下传递，这样可以避免不必要的计算，提高效率。
 *
 * 2. 线段树结构：
 *    - tree[]数组：存储线段树节点的值（区间和）
 *    - lazy[]数组：存储懒惰标记（区间加法的增量）
 *
```

```
 * 3. 关键操作：
 *    - pushDown：懒惰传播操作，将父节点的标记传递给子节点
 *    - rangeAdd：区间加法更新
 *    - rangeSum：区间求和查询
 *
 * 时间复杂度分析：
 * - 构建线段树：O(n)
 * - 区间加法更新：O(log n)
 * - 区间求和查询：O(log n)
 *
 * 空间复杂度分析：
 * - 线段树需要O(n)的额外空间
 */

// 定义常量
#define MAXN 100005

// 全局数组存储线段树和懒惰标记
long long tree[4 * MAXN];  // 线段树数组，存储区间和
long long lazy[4 * MAXN];  // 懒惰标记数组，存储区间加法的增量
int arr[MAXN];             // 原始数组
int n, m;                  // 数组长度和操作数

/**
 * 构建线段树
 * 递归地将数组构建成线段树结构
 * @param node 当前线段树节点索引
 * @param start 当前节点表示区间的起始位置
 * @param end 当前节点表示区间的结束位置
 *
 * 时间复杂度: O(n)
 * 空间复杂度: O(n)
 */
void buildTree(int node, int start, int end) {
    if (start == end) {
        // 叶子节点 - 直接存储数组元素值
        tree[node] = arr[start];
    } else {
        int mid = (start + end) / 2;
        // 递归构建左子树
        buildTree(2 * node + 1, start, mid);
        // 递归构建右子树
        buildTree(2 * node + 2, mid + 1, end);
```

```
        // 合并左右子树的结果（区间和）
        tree[node] = tree[2 * node + 1] + tree[2 * node + 2];
    }
}

/**
 * 懒惰传播 - 将当前节点的懒惰标记传递给子节点
 * 这是懒惰传播技术的核心实现
 * @param node 当前线段树节点索引
 * @param start 当前节点表示区间的起始位置
 * @param end 当前节点表示区间的结束位置
 *
 * 时间复杂度: O(1)
 */
void pushDown(int node, int start, int end) {
    // 只有当当前节点有懒惰标记时才需要传播
    if (lazy[node] != 0) {
        int mid = (start + end) / 2;
        int leftLen = mid - start + 1;   // 左子树区间长度
        int rightLen = end - mid;        // 右子树区间长度

        // 更新左子树
        // 1. 更新左子树的区间和：增加 lazy[node] * 区间长度
        tree[2 * node + 1] += lazy[node] * leftLen;
        // 2. 将懒惰标记传递给左子树
        lazy[2 * node + 1] += lazy[node];

        // 更新右子树
        // 1. 更新右子树的区间和：增加 lazy[node] * 区间长度
        tree[2 * node + 2] += lazy[node] * rightLen;
        // 2. 将懒惰标记传递给右子树
        lazy[2 * node + 2] += lazy[node];

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

/**
 * 区间加法更新
 * 将区间[left, right]内的每个数都加上 val
 * @param node 当前线段树节点索引
 * @param start 当前节点表示区间的起始位置
```

```
 * @param end 当前节点表示区间的结束位置
 * @param left 更新区间左边界（0-based索引）
 * @param right 更新区间右边界（0-based索引）
 * @param val 要加上的值
 *
 * 时间复杂度: O(log n)
 */
void rangeAdd(int node, int start, int end, int left, int right, long long val) {
    // 如果当前区间完全包含在更新区间内
    if (left <= start && end <= right) {
        // 直接更新当前节点的值和懒惰标记
        tree[node] += val * (end - start + 1);  // 更新区间和
        lazy[node] += val;  // 打上懒惰标记
        return;
    }

    // 需要向下传递懒惰标记（在递归之前）
    pushDown(node, start, end);

    // 递归更新左右子树
    int mid = (start + end) / 2;
    // 如果更新区间与左子树有交集，则更新左子树
    if (left <= mid) {
        rangeAdd(2 * node + 1, start, mid, left, right, val);
    }
    // 如果更新区间与右子树有交集，则更新右子树
    if (right > mid) {
        rangeAdd(2 * node + 2, mid + 1, end, left, right, val);
    }

    // 更新父节点的值（子节点更新后需要更新父节点）
    tree[node] = tree[2 * node + 1] + tree[2 * node + 2];
}

/**
 * 区间求和查询
 * 查询区间[left, right]内所有数的和
 * @param node 当前线段树节点索引
 * @param start 当前节点表示区间的起始位置
 * @param end 当前节点表示区间的结束位置
 * @param left 查询区间左边界（0-based索引）
 * @param right 查询区间右边界（0-based索引）
 * @return 区间[left, right]内所有数的和
```

```
 *
 * 时间复杂度: O(log n)
 */
long long rangeSum(int node, int start, int end, int left, int right) {
    // 如果当前区间完全包含在查询区间内
    if (left <= start && end <= right) {
        // 直接返回当前节点的值
        return tree[node];
    }

    // 需要向下传递懒惰标记（在递归之前）
    pushDown(node, start, end);

    // 递归查询左右子树
    int mid = (start + end) / 2;
    long long sum = 0;
    // 如果查询区间与左子树有交集，则查询左子树
    if (left <= mid) {
        sum += rangeSum(2 * node + 1, start, mid, left, right);
    }
    // 如果查询区间与右子树有交集，则查询右子树
    if (right > mid) {
        sum += rangeSum(2 * node + 2, mid + 1, end, left, right);
    }

    return sum;
}

/**
 * 初始化线段树
 * @param nums 原始数组
 * @param size 数组大小
 */
void init(int nums[], int size) {
    n = size;
    int i;
    for (i = 0; i < n; i++) {
        arr[i] = nums[i];
    }
    buildTree(1, 0, n - 1);  // 使用1-based索引
}

/**
```

```
 * 区间加法更新（对外接口）
 * 将区间[left, right]内的每个数都加上 val
 * @param left 更新区间左边界（0-based 索引）
 * @param right 更新区间右边界（0-based 索引）
 * @param val 要加上的值
 */
void rangeAddValue(int left, int right, long long val) {
    rangeAdd(1, 0, n - 1, left, right, val);  // 使用 1-based 索引
}

/**
 * 区间求和查询（对外接口）
 * 查询区间[left, right]内所有数的和
 * @param left 查询区间左边界（0-based 索引）
 * @param right 查询区间右边界（0-based 索引）
 * @return 区间[left, right]内所有数的和
 */
long long rangeSumValue(int left, int right) {
    return rangeSum(1, 0, n - 1, left, right);  // 使用 1-based 索引
}

/**
 * 测试函数 - 验证线段树实现的正确性
 */
void test() {
    // 测试用例 1: 基础功能测试
    int nums1[] = {1, 2, 3, 4, 5};
    init(nums1, 5);

    // 测试区间求和
    // 应该输出 初始数组区间[0,4]的和: 15
    // 应该输出 区间[1,3]的和: 9

    // 测试区间加法
    rangeAddValue(1, 3, 2); // 将索引 1-3 的元素都加 2
    // 应该输出 区间加法后区间[1,3]的和: 15 (2+2, 3+2, 4+2 = 4+5+6)
    // 应该输出 区间[0,4]的和: 21 (1+4+5+6+5)

    // 测试用例 2: 边界情况
    int nums2[] = {10};
    init(nums2, 1);
    // 应该输出 单元素数组区间[0,0]的和: 10
    rangeAddValue(0, 0, 5);
```

```
    // 应该输出 单元素加法后区间[0,0]的和: 15

    // 测试用例 3: 大规模数据测试
    int nums3[1000];
    int i;
    for (i = 0; i < 1000; i++) {
        nums3[i] = 1;
    }
    init(nums3, 1000);

    // 对整个数组进行区间加法
    rangeAddValue(0, 999, 10);
    // 应该输出 大规模数组区间[0,999]的和: 11000 (1000 * 11)
}

// 主函数 - 处理输入输出和操作调度
int main() {
    // 由于环境限制，这里不实现完整的输入输出处理
    // 实际使用时需要根据具体的输入输出要求进行调整

    // 运行测试
    test();

    return 0;
}
```

==========

文件: Code16_LazyPropagation.java

==========

```
package class112;

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

/**
 * 线段树区间更新 - 懒惰传播 (Lazy Propagation)
 *
 * 题目描述：
 * 实现支持区间加法和区间查询的线段树
 * 操作类型：
 * 1. 区间加法：将区间 [l, r] 内的每个数加上 k
 * 2. 区间求和：查询区间 [l, r] 内所有数的和
```

```
 *
 * 题目来源：洛谷 P3372 【模板】线段树 1
 * 测试链接 : https://www.luogu.com.cn/problem/P3372
 *
 * 解题思路：
 * 使用线段树配合懒惰传播技术来高效处理区间更新和区间查询操作。
 *
 * 核心思想：
 * 1. 懒惰传播：当需要对一个区间进行更新时，不立即更新所有相关节点，
 *    而是在节点上打上标记，只有在后续查询或更新需要访问该节点的子节点时，
 *    才将标记向下传递，这样可以避免不必要的计算，提高效率。
 *
 * 2. 线段树结构：
 *    - tree[]数组：存储线段树节点的值（区间和）
 *    - lazy[]数组：存储懒惰标记（区间加法的增量）
 *
 * 3. 关键操作：
 *    - pushDown：懒惰传播操作，将父节点的标记传递给子节点
 *    - rangeAdd：区间加法更新
 *    - rangeSum：区间求和查询
 *
 * 时间复杂度分析：
 * - 构建线段树：O(n)
 * - 区间加法更新：O(log n)
 * - 区间求和查询：O(log n)
 *
 * 空间复杂度分析：
 * - 线段树需要O(n)的额外空间
 */
public class Code16_LazyPropagation {

    /**
     * 线段树内部类 - 支持懒惰传播
     *
     * 数据结构说明：
     * - tree[]：存储线段树节点的值（区间和）
     * - lazy[]：存储懒惰标记（区间加法的增量）
     * - n：原始数组长度
     */
    static class SegmentTree {
        long[] tree;    // 线段树数组，存储区间和
        long[] lazy;    // 懒惰标记数组，存储区间加法的增量
        int n;          // 数组长度
```

```
/**
 * 构造函数 - 初始化线段树
 * @param nums 原始数组
 */
public SegmentTree(int[] nums) {
    n = nums.length;
    tree = new long[n * 4];  // 线段树通常需要 4 倍空间
    lazy = new long[n * 4];  // 懒惰标记数组
    buildTree(nums, 0, 0, n - 1);
}

/**
 * 构建线段树
 * 递归地将数组构建成线段树结构
 * @param nums 原始数组
 * @param node 当前线段树节点索引
 * @param start 当前节点表示区间的起始位置
 * @param end 当前节点表示区间的结束位置
 *
 * 时间复杂度: O(n)
 * 空间复杂度: O(n)
 */
private void buildTree(int[] nums, int node, int start, int end) {
    if (start == end) {
        // 叶子节点 - 直接存储数组元素值
        tree[node] = nums[start];
    } else {
        int mid = (start + end) / 2;
        // 递归构建左子树
        buildTree(nums, 2 * node + 1, start, mid);
        // 递归构建右子树
        buildTree(nums, 2 * node + 2, mid + 1, end);
        // 合并左右子树的结果（区间和）
        tree[node] = tree[2 * node + 1] + tree[2 * node + 2];
    }
}

/**
 * 懒惰传播 - 将当前节点的懒惰标记传递给子节点
 * 这是懒惰传播技术的核心实现
 * @param node 当前线段树节点索引
 * @param start 当前节点表示区间的起始位置
```

```
     * @param end 当前节点表示区间的结束位置
     *
     * 时间复杂度: O(1)
     */
    private void pushDown(int node, int start, int end) {
        // 只有当当前节点有懒惰标记时才需要传播
        if (lazy[node] != 0) {
            int mid = (start + end) / 2;
            int leftLen = mid - start + 1;   // 左子树区间长度
            int rightLen = end - mid;        // 右子树区间长度

            // 更新左子树
            // 1. 更新左子树的区间和：增加 lazy[node] * 区间长度
            tree[2 * node + 1] += lazy[node] * leftLen;
            // 2. 将懒惰标记传递给左子树
            lazy[2 * node + 1] += lazy[node];

            // 更新右子树
            // 1. 更新右子树的区间和：增加 lazy[node] * 区间长度
            tree[2 * node + 2] += lazy[node] * rightLen;
            // 2. 将懒惰标记传递给右子树
            lazy[2 * node + 2] += lazy[node];

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

    /**
     * 区间加法更新
     * 将区间[left, right]内的每个数都加上 val
     * @param left 更新区间左边界（0-based 索引）
     * @param right 更新区间右边界（0-based 索引）
     * @param val 要加上的值
     *
     * 时间复杂度: O(log n)
     */
    public void rangeAdd(int left, int right, long val) {
        rangeAddHelper(0, 0, n - 1, left, right, val);
    }

    /**
     * 区间加法更新辅助函数
```

```
     * @param node 当前线段树节点索引
     * @param start 当前节点表示区间的起始位置
     * @param end 当前节点表示区间的结束位置
     * @param left 更新区间左边界（0-based索引）
     * @param right 更新区间右边界（0-based索引）
     * @param val 要加上的值
     */
    private void rangeAddHelper(int node, int start, int end, int left, int right, long val)
{
        // 如果当前区间完全包含在更新区间内
        if (left <= start && end <= right) {
            // 直接更新当前节点的值和懒惰标记
            tree[node] += val * (end - start + 1);  // 更新区间和
            lazy[node] += val;  // 打上懒惰标记
            return;
        }

        // 需要向下传递懒惰标记（在递归之前）
        pushDown(node, start, end);

        // 递归更新左右子树
        int mid = (start + end) / 2;
        // 如果更新区间与左子树有交集，则更新左子树
        if (left <= mid) {
            rangeAddHelper(2 * node + 1, start, mid, left, right, val);
        }
        // 如果更新区间与右子树有交集，则更新右子树
        if (right > mid) {
            rangeAddHelper(2 * node + 2, mid + 1, end, left, right, val);
        }

        // 更新父节点的值（子节点更新后需要更新父节点）
        tree[node] = tree[2 * node + 1] + tree[2 * node + 2];
    }

    /**
     * 区间求和查询
     * 查询区间[left, right]内所有数的和
     * @param left 查询区间左边界（0-based索引）
     * @param right 查询区间右边界（0-based索引）
     * @return 区间[left, right]内所有数的和
     *
     * 时间复杂度: O(log n)
```

```
     */
    public long rangeSum(int left, int right) {
        return rangeSumHelper(0, 0, n - 1, left, right);
    }

    /**
     * 区间求和查询辅助函数
     * @param node 当前线段树节点索引
     * @param start 当前节点表示区间的起始位置
     * @param end 当前节点表示区间的结束位置
     * @param left 查询区间左边界（0-based索引）
     * @param right 查询区间右边界（0-based索引）
     * @return 区间[left, right]内所有数的和
     */
    private long rangeSumHelper(int node, int start, int end, int left, int right) {
        // 如果当前区间完全包含在查询区间内
        if (left <= start && end <= right) {
            // 直接返回当前节点的值
            return tree[node];
        }

        // 需要向下传递懒惰标记（在递归之前）
        pushDown(node, start, end);

        // 递归查询左右子树
        int mid = (start + end) / 2;
        long sum = 0;
        // 如果查询区间与左子树有交集，则查询左子树
        if (left <= mid) {
            sum += rangeSumHelper(2 * node + 1, start, mid, left, right);
        }
        // 如果查询区间与右子树有交集，则查询右子树
        if (right > mid) {
            sum += rangeSumHelper(2 * node + 2, mid + 1, end, left, right);
        }

        return sum;
    }
}

/**
 * 主函数 - 处理输入输出和操作调度
 * 使用高效的输入输出处理方式
```

```
 */
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));

    // 读取 n 和 m
    in.nextToken();
    int n = (int) in.nval;
    in.nextToken();
    int m = (int) in.nval;

    // 读取初始数组
    int[] nums = new int[n];
    for (int i = 0; i < n; i++) {
        in.nextToken();
        nums[i] = (int) in.nval;
    }

    // 构建线段树
    SegmentTree st = new SegmentTree(nums);

    // 处理 m 个操作
    for (int i = 0; i < m; i++) {
        in.nextToken();
        int op = (int) in.nval;
        in.nextToken();
        int l = (int) in.nval;
        in.nextToken();
        int r = (int) in.nval;

        if (op == 1) {
            // 区间加法操作
            in.nextToken();
            long k = (long) in.nval;
            st.rangeAdd(l - 1, r - 1, k); // 转换为 0-based 索引
        } else {
            // 区间求和操作
            long sum = st.rangeSum(l - 1, r - 1); // 转换为 0-based 索引
            out.println(sum);
        }
    }
```

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

    /**
     * 测试方法 - 验证线段树实现的正确性
     */
    public static void test() {
        // 测试用例 1: 基础功能测试
        int[] nums1 = {1, 2, 3, 4, 5};
        SegmentTree st1 = new SegmentTree(nums1);

        // 测试区间求和
        System.out.println("初始数组区间[0,4]的和: " + st1.rangeSum(0, 4)); // 应该输出 15
        System.out.println("区间[1,3]的和: " + st1.rangeSum(1, 3)); // 应该输出 9

        // 测试区间加法
        st1.rangeAdd(1, 3, 2); // 将索引 1-3 的元素都加 2
        System.out.println("区间加法后区间[1,3]的和: " + st1.rangeSum(1, 3)); // 应该输出 15
(2+2, 3+2, 4+2 = 4+5+6)
        System.out.println("区间[0,4]的和: " + st1.rangeSum(0, 4)); // 应该输出 21 (1+4+5+6+5)

        // 测试用例 2: 边界情况
        int[] nums2 = {10};
        SegmentTree st2 = new SegmentTree(nums2);
        System.out.println("单元素数组区间[0,0]的和: " + st2.rangeSum(0, 0)); // 应该输出 10
        st2.rangeAdd(0, 0, 5);
        System.out.println("单元素加法后区间[0,0]的和: " + st2.rangeSum(0, 0)); // 应该输出 15

        // 测试用例 3: 大规模数据测试
        int[] nums3 = new int[1000];
        Arrays.fill(nums3, 1);
        SegmentTree st3 = new SegmentTree(nums3);

        // 对整个数组进行区间加法
        st3.rangeAdd(0, 999, 10);
        System.out.println("大规模数组区间[0,999]的和: " + st3.rangeSum(0, 999)); // 应该输出
1000 * 11 = 11000
    }
}

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

文件: Code16_LazyPropagation.py
==========
"""
线段树区间更新 - 懒惰传播 (Lazy Propagation)

题目描述：
实现支持区间加法和区间查询的线段树
操作类型：
1. 区间加法：将区间 [l, r] 内的每个数加上 k
2. 区间求和：查询区间 [l, r] 内所有数的和

题目来源：洛谷 P3372 【模板】线段树 1
测试链接 : https://www.luogu.com.cn/problem/P3372

解题思路：
使用线段树配合懒惰传播技术来高效处理区间更新和区间查询操作。

核心思想：
1. 懒惰传播：当需要对一个区间进行更新时，不立即更新所有相关节点，
   而是在节点上打上标记，只有在后续查询或更新需要访问该节点的子节点时，
   才将标记向下传递，这样可以避免不必要的计算，提高效率。

2. 线段树结构：
   - tree[]数组：存储线段树节点的值（区间和）
   - lazy[]数组：存储懒惰标记（区间加法的增量）

3. 关键操作：
   - push_down：懒惰传播操作，将父节点的标记传递给子节点
   - range_add：区间加法更新
   - range_sum：区间求和查询

时间复杂度分析：
- 构建线段树：O(n)
- 区间加法更新：O(log n)
- 区间求和查询：O(log n)

空间复杂度分析：
- 线段树需要O(n)的额外空间
"""

import sys
from typing import List

```
class SegmentTree:
    """线段树类 - 支持懒惰传播"""

    def __init__(self, nums: List[int]):
        """
        构造函数 - 初始化线段树
        :param nums: 原始数组
        """
        self.n = len(nums)
        # 线段树通常需要 4 倍空间
        self.tree = [0] * (4 * self.n)  # 存储线段树节点的值（区间和）
        self.lazy = [0] * (4 * self.n)  # 存储懒惰标记（区间加法的增量）
        self.build_tree(nums, 0, 0, self.n - 1)

    def build_tree(self, nums: List[int], node: int, start: int, end: int) -> None:
        """
        构建线段树
        递归地将数组构建成线段树结构
        :param nums: 原始数组
        :param node: 当前线段树节点索引
        :param start: 当前节点表示区间的起始位置
        :param end: 当前节点表示区间的结束位置

        时间复杂度: O(n)
        空间复杂度: O(n)
        """
        if start == end:
            # 叶子节点 - 直接存储数组元素值
            self.tree[node] = nums[start]
        else:
            mid = (start + end) // 2
            # 递归构建左子树
            self.build_tree(nums, 2 * node + 1, start, mid)
            # 递归构建右子树
            self.build_tree(nums, 2 * node + 2, mid + 1, end)
            # 合并左右子树的结果（区间和）
            self.tree[node] = self.tree[2 * node + 1] + self.tree[2 * node + 2]

    def push_down(self, node: int, start: int, end: int) -> None:
        """
        懒惰传播 - 将当前节点的懒惰标记传递给子节点
        这是懒惰传播技术的核心实现
```

```
        :param node: 当前线段树节点索引
        :param start: 当前节点表示区间的起始位置
        :param end: 当前节点表示区间的结束位置

        时间复杂度: O(1)
        """
        # 只有当当前节点有懒惰标记时才需要传播
        if self.lazy[node] != 0:
            mid = (start + end) // 2
            left_len = mid - start + 1   # 左子树区间长度
            right_len = end - mid        # 右子树区间长度

            # 更新左子树
            # 1. 更新左子树的区间和：增加 lazy[node] * 区间长度
            self.tree[2 * node + 1] += self.lazy[node] * left_len
            # 2. 将懒惰标记传递给左子树
            self.lazy[2 * node + 1] += self.lazy[node]

            # 更新右子树
            # 1. 更新右子树的区间和：增加 lazy[node] * 区间长度
            self.tree[2 * node + 2] += self.lazy[node] * right_len
            # 2. 将懒惰标记传递给右子树
            self.lazy[2 * node + 2] += self.lazy[node]

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

    def range_add(self, left: int, right: int, val: int) -> None:
        """
        区间加法更新
        将区间[left, right]内的每个数都加上 val
        :param left: 更新区间左边界（0-based 索引）
        :param right: 更新区间右边界（0-based 索引）
        :param val: 要加上的值

        时间复杂度: O(log n)
        """
        self._range_add_helper(0, 0, self.n - 1, left, right, val)

    def _range_add_helper(self, node: int, start: int, end: int, left: int, right: int, val: int)
-> None:
        """
        区间加法更新辅助函数
```

```
        :param node: 当前线段树节点索引
        :param start: 当前节点表示区间的起始位置
        :param end: 当前节点表示区间的结束位置
        :param left: 更新区间左边界（0-based索引）
        :param right: 更新区间右边界（0-based索引）
        :param val: 要加上的值
        """
        # 如果当前区间完全包含在更新区间内
        if left <= start and end <= right:
            # 直接更新当前节点的值和懒惰标记
            self.tree[node] += val * (end - start + 1)  # 更新区间和
            self.lazy[node] += val  # 打上懒惰标记
            return

        # 需要向下传递懒惰标记（在递归之前）
        self.push_down(node, start, end)

        # 递归更新左右子树
        mid = (start + end) // 2
        # 如果更新区间与左子树有交集，则更新左子树
        if left <= mid:
            self._range_add_helper(2 * node + 1, start, mid, left, right, val)
        # 如果更新区间与右子树有交集，则更新右子树
        if right > mid:
            self._range_add_helper(2 * node + 2, mid + 1, end, left, right, val)

        # 更新父节点的值（子节点更新后需要更新父节点）
        self.tree[node] = self.tree[2 * node + 1] + self.tree[2 * node + 2]

    def range_sum(self, left: int, right: int) -> int:
        """
        区间求和查询
        查询区间[left, right]内所有数的和
        :param left: 查询区间左边界（0-based索引）
        :param right: 查询区间右边界（0-based索引）
        :return: 区间[left, right]内所有数的和

        时间复杂度: O(log n)
        """
        return self._range_sum_helper(0, 0, self.n - 1, left, right)

    def _range_sum_helper(self, node: int, start: int, end: int, left: int, right: int) -> int:
        """
```

```
        区间求和查询辅助函数
        :param node: 当前线段树节点索引
        :param start: 当前节点表示区间的起始位置
        :param end: 当前节点表示区间的结束位置
        :param left: 查询区间左边界（0-based索引）
        :param right: 查询区间右边界（0-based索引）
        :return: 区间[left, right]内所有数的和
        """
        # 如果当前区间完全包含在查询区间内
        if left <= start and end <= right:
            # 直接返回当前节点的值
            return self.tree[node]

        # 需要向下传递懒惰标记（在递归之前）
        self.push_down(node, start, end)

        # 递归查询左右子树
        mid = (start + end) // 2
        total = 0
        # 如果查询区间与左子树有交集，则查询左子树
        if left <= mid:
            total += self._range_sum_helper(2 * node + 1, start, mid, left, right)
        # 如果查询区间与右子树有交集，则查询右子树
        if right > mid:
            total += self._range_sum_helper(2 * node + 2, mid + 1, end, left, right)

        return total


def main():
    """
    主函数 - 处理输入输出和操作调度
    """
    # 读取n和m
    n, m = map(int, input().split())

    # 读取初始数组
    nums = list(map(int, input().split()))

    # 构建线段树
    st = SegmentTree(nums)

    # 处理m个操作
```

```
    for _ in range(m):
        operation = list(map(int, input().split()))

        if operation[0] == 1:
            # 区间加法操作
            l, r, k = operation[1], operation[2], operation[3]
            st.range_add(l - 1, r - 1, k)  # 转换为0-based索引
        else:
            # 区间求和操作
            l, r = operation[1], operation[2]
            sum_result = st.range_sum(l - 1, r - 1)  # 转换为0-based索引
            print(sum_result)


def test():
    """
    测试函数 - 验证线段树实现的正确性
    """
    # 测试用例1: 基础功能测试
    nums1 = [1, 2, 3, 4, 5]
    st1 = SegmentTree(nums1)

    # 测试区间求和
    print(f"初始数组区间[0,4]的和: {st1.range_sum(0, 4)}")  # 应该输出 15
    print(f"区间[1,3]的和: {st1.range_sum(1, 3)}")  # 应该输出 9

    # 测试区间加法
    st1.range_add(1, 3, 2)  # 将索引1-3的元素都加2
    print(f"区间加法后区间[1,3]的和: {st1.range_sum(1, 3)}")  # 应该输出 15 (2+2, 3+2, 4+2 = 4+5+6)
    print(f"区间[0,4]的和: {st1.range_sum(0, 4)}")  # 应该输出 21 (1+4+5+6+5)

    # 测试用例2: 边界情况
    nums2 = [10]
    st2 = SegmentTree(nums2)
    print(f"单元素数组区间[0,0]的和: {st2.range_sum(0, 0)}")  # 应该输出 10
    st2.range_add(0, 0, 5)
    print(f"单元素加法后区间[0,0]的和: {st2.range_sum(0, 0)}")  # 应该输出 15

    # 测试用例3: 大规模数据测试
    nums3 = [1] * 1000
    st3 = SegmentTree(nums3)
```

```
    # 对整个数组进行区间加法
    st3.range_add(0, 999, 10)
    print(f"大规模数组区间[0,999]的和: {st3.range_sum(0, 999)}")  # 应该输出 1000 * 11 = 11000


# 如果需要运行测试，取消下面的注释
# test()

# 如果需要运行主程序，取消下面的注释
# if __name__ == "__main__":
#     main()

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

文件: Code17_SegmentTreeApplications.cpp
==============================================
/**
 * 线段树高级应用 - 多种区间操作
 *
 * 题目描述：
 * 实现支持多种区间操作的线段树，包括：
 * 1. 区间赋值
 * 2. 区间加法
 * 3. 区间乘法
 * 4. 区间求和
 * 5. 区间最大值/最小值
 *
 * 题目来源：洛谷 P3373 【模板】线段树 2
 * 测试链接 : https://www.luogu.com.cn/problem/P3373
 *
 * 解题思路：
 * 使用高级线段树实现支持多种操作，包括区间赋值、加法、乘法以及查询操作。
 * 通过维护多种懒惰标记来处理不同优先级的操作。
 *
 * 核心思想：
 * 1. 多标记懒惰传播：同时维护加法、乘法和赋值三种懒惰标记
 * 2. 标记优先级：赋值 > 乘法 > 加法
 * 3. 标记下传：在下传标记时需要按照优先级顺序处理
 *
 * 时间复杂度分析：
 * - 构建线段树：O(n)
 * - 所有区间操作：O(log n)
 *
```

```
 * 空间复杂度分析：
 * - 线段树需要O(n)的额外空间
 */

// 定义常量
#define MAXN 100005
#define INF 1000000000000000LL

// 结构体定义线段树节点
struct Node {
    long long sum;        // 区间和
    long long max_val;    // 区间最大值
    long long min_val;    // 区间最小值
    long long add;        // 加法懒惰标记
    long long mul;        // 乘法懒惰标记
    long long set_val;    // 赋值懒惰标记
    bool has_set;         // 是否有赋值标记

    // 构造函数
    Node() {
        sum = 0;
        max_val = -INF;
        min_val = INF;
        add = 0;
        mul = 1;
        set_val = 0;
        has_set = false;
    }
};

// 全局数组和变量
Node tree[4 * MAXN];  // 线段树数组
int arr[MAXN];        // 原始数组
int n, m;             // 数组长度和操作数

/**
 * 向上更新父节点
 * 根据左右子节点的信息更新当前节点的信息
 * @param node 当前线段树节点索引
 */
void pushUp(int node) {
    int left = 2 * node + 1;   // 左子节点索引
    int right = 2 * node + 2;  // 右子节点索引
```

```
    // 更新区间和
    tree[node].sum = tree[left].sum + tree[right].sum;
    // 更新区间最大值
    tree[node].max_val = (tree[left].max_val > tree[right].max_val) ? tree[left].max_val :
tree[right].max_val;
    // 更新区间最小值
    tree[node].min_val = (tree[left].min_val < tree[right].min_val) ? tree[left].min_val :
tree[right].min_val;
}

/**
 * 向下传递懒惰标记
 * 按照标记优先级顺序传递标记给子节点
 * 优先级：赋值 > 乘法 > 加法
 * @param node 当前线段树节点索引
 * @param start 当前节点表示区间的起始位置
 * @param end 当前节点表示区间的结束位置
 *
 * 时间复杂度: O(1)
 */
void pushDown(int node, int start, int end) {
    // 叶子节点不需要传递标记
    if (start == end) return;

    int left = 2 * node + 1;   // 左子节点索引
    int right = 2 * node + 2;  // 右子节点索引
    int mid = (start + end) / 2;

    // 处理赋值标记（优先级最高）
    // 当存在赋值标记时，需要清除子节点的其他标记
    if (tree[node].has_set) {
        // 更新左子树的区间信息
        tree[left].sum = tree[node].set_val * (mid - start + 1);  // 区间和 = 赋值 * 区间长度
        tree[right].sum = tree[node].set_val * (end - mid);
        tree[left].max_val = tree[node].set_val;  // 区间最大值 = 赋值
        tree[right].max_val = tree[node].set_val;
        tree[left].min_val = tree[node].set_val;  // 区间最小值 = 赋值
        tree[right].min_val = tree[node].set_val;

        // 传递赋值标记给子节点
        tree[left].set_val = tree[node].set_val;
        tree[right].set_val = tree[node].set_val;
        tree[left].has_set = true;
```

```
    tree[right].has_set = true;

    // 清除子节点的其他标记（加法和乘法）
    tree[left].add = 0;
    tree[right].add = 0;
    tree[left].mul = 1;
    tree[right].mul = 1;

    // 清除当前节点的赋值标记
    tree[node].has_set = false;
}

// 处理乘法标记（优先级次之）
// 当存在乘法标记时，需要更新子节点的所有信息
if (tree[node].mul != 1) {
    // 更新左子树的区间信息（乘以 mul）
    tree[left].sum *= tree[node].mul;
    tree[right].sum *= tree[node].mul;
    tree[left].max_val *= tree[node].mul;
    tree[right].max_val *= tree[node].mul;
    tree[left].min_val *= tree[node].mul;
    tree[right].min_val *= tree[node].mul;

    // 传递乘法标记给子节点
    tree[left].mul *= tree[node].mul;
    tree[right].mul *= tree[node].mul;
    // 乘法标记也会影响加法标记（add * mul）
    tree[left].add *= tree[node].mul;
    tree[right].add *= tree[node].mul;

    // 清除当前节点的乘法标记
    tree[node].mul = 1;
}

// 处理加法标记（优先级最低）
// 当存在加法标记时，需要更新子节点的所有信息
if (tree[node].add != 0) {
    int leftLen = mid - start + 1;   // 左子树区间长度
    int rightLen = end - mid;        // 右子树区间长度

    // 更新左子树的区间信息（加上 add）
    tree[left].sum += tree[node].add * leftLen;  // 区间和增加 add * 区间长度
    tree[right].sum += tree[node].add * rightLen;
```

```
        tree[left].max_val += tree[node].add;  // 区间最大值增加 add
        tree[right].max_val += tree[node].add;
        tree[left].min_val += tree[node].add;  // 区间最小值增加 add
        tree[right].min_val += tree[node].add;

        // 传递加法标记给子节点
        tree[left].add += tree[node].add;
        tree[right].add += tree[node].add;

        // 清除当前节点的加法标记
        tree[node].add = 0;
    }
}

/**
 * 构建线段树
 * 递归地将数组构建成线段树结构
 * @param node 当前线段树节点索引
 * @param start 当前节点表示区间的起始位置
 * @param end 当前节点表示区间的结束位置
 *
 * 时间复杂度: O(n)
 */
void buildTree(int node, int start, int end) {
    // 初始化节点
    tree[node] = Node();

    if (start == end) {
        // 叶子节点 - 直接存储数组元素值
        tree[node].sum = arr[start];
        tree[node].max_val = arr[start];
        tree[node].min_val = arr[start];
    } else {
        int mid = (start + end) / 2;
        // 递归构建左子树
        buildTree(2 * node + 1, start, mid);
        // 递归构建右子树
        buildTree(2 * node + 2, mid + 1, end);
        // 向上更新父节点信息
        pushUp(node);
    }
}
```

```
/**
 * 区间赋值操作
 * 将区间[left, right]内的每个数都赋值为 val
 * @param node 当前线段树节点索引
 * @param start 当前节点表示区间的起始位置
 * @param end 当前节点表示区间的结束位置
 * @param l 更新区间左边界（0-based 索引）
 * @param r 更新区间右边界（0-based 索引）
 * @param val 要赋的值
 *
 * 时间复杂度: O(log n)
 */
void rangeSet(int node, int start, int end, int l, int r, long long val) {
    // 如果当前区间完全包含在更新区间内
    if (l <= start && end <= r) {
        // 直接更新当前节点的信息和标记
        tree[node].sum = val * (end - start + 1);  // 区间和 = 赋值 * 区间长度
        tree[node].max_val = val;  // 区间最大值 = 赋值
        tree[node].min_val = val;  // 区间最小值 = 赋值
        tree[node].set_val = val;  // 设置赋值标记
        tree[node].has_set = true;  // 标记存在赋值操作
        tree[node].add = 0;  // 清除加法标记
        tree[node].mul = 1;  // 清除乘法标记
        return;
    }

    // 需要向下传递懒惰标记（在递归之前）
    pushDown(node, start, end);
    int mid = (start + end) / 2;
    // 递归更新左右子树
    if (l <= mid) rangeSet(2 * node + 1, start, mid, l, r, val);
    if (r > mid) rangeSet(2 * node + 2, mid + 1, end, l, r, val);
    // 更新父节点信息
    pushUp(node);
}

/**
 * 区间乘法操作
 * 将区间[left, right]内的每个数都乘以 val
 * @param node 当前线段树节点索引
 * @param start 当前节点表示区间的起始位置
 * @param end 当前节点表示区间的结束位置
 * @param l 更新区间左边界（0-based 索引）
```

```
 * @param r 更新区间右边界（0-based索引）
 * @param val 要乘的值
 *
 * 时间复杂度: O(log n)
 */
void rangeMul(int node, int start, int end, int l, int r, long long val) {
    // 如果当前区间完全包含在更新区间内
    if (l <= start && end <= r) {
        // 直接更新当前节点的信息和标记
        tree[node].sum *= val;  // 区间和乘以val
        tree[node].max_val *= val;  // 区间最大值乘以val
        tree[node].min_val *= val;  // 区间最小值乘以val
        tree[node].mul *= val;  // 乘法标记乘以val
        tree[node].add *= val;  // 加法标记也乘以val（因为 a*x + b 变成 a*x*val + b*val）
        return;
    }

    // 需要向下传递懒惰标记（在递归之前）
    pushDown(node, start, end);
    int mid = (start + end) / 2;
    // 递归更新左右子树
    if (l <= mid) rangeMul(2 * node + 1, start, mid, l, r, val);
    if (r > mid) rangeMul(2 * node + 2, mid + 1, end, l, r, val);
    // 更新父节点信息
    pushUp(node);
}

/**
 * 区间加法操作
 * 将区间[left, right]内的每个数都加上val
 * @param node 当前线段树节点索引
 * @param start 当前节点表示区间的起始位置
 * @param end 当前节点表示区间的结束位置
 * @param l 更新区间左边界（0-based索引）
 * @param r 更新区间右边界（0-based索引）
 * @param val 要加的值
 *
 * 时间复杂度: O(log n)
 */
void rangeAdd(int node, int start, int end, int l, int r, long long val) {
    // 如果当前区间完全包含在更新区间内
    if (l <= start && end <= r) {
        // 直接更新当前节点的信息和标记
```

```
        tree[node].sum += val * (end - start + 1);  // 区间和增加 val * 区间长度
        tree[node].max_val += val;  // 区间最大值增加 val
        tree[node].min_val += val;  // 区间最小值增加 val
        tree[node].add += val;  // 加法标记增加 val
        return;
    }

    // 需要向下传递懒惰标记（在递归之前）
    pushDown(node, start, end);
    int mid = (start + end) / 2;
    // 递归更新左右子树
    if (l <= mid) rangeAdd(2 * node + 1, start, mid, l, r, val);
    if (r > mid) rangeAdd(2 * node + 2, mid + 1, end, l, r, val);
    // 更新父节点信息
    pushUp(node);
}

/**
 * 区间求和查询
 * 查询区间[left, right]内所有数的和
 * @param node 当前线段树节点索引
 * @param start 当前节点表示区间的起始位置
 * @param end 当前节点表示区间的结束位置
 * @param l 查询区间左边界（0-based索引）
 * @param r 查询区间右边界（0-based索引）
 * @return 区间[left, right]内所有数的和
 *
 * 时间复杂度: O(log n)
 */
long long rangeSum(int node, int start, int end, int l, int r) {
    // 如果当前区间完全包含在查询区间内
    if (l <= start && end <= r) {
        // 直接返回当前节点的区间和
        return tree[node].sum;
    }

    // 需要向下传递懒惰标记（在递归之前）
    pushDown(node, start, end);
    int mid = (start + end) / 2;
    long long sum = 0;
    // 递归查询左右子树
    if (l <= mid) sum += rangeSum(2 * node + 1, start, mid, l, r);
    if (r > mid) sum += rangeSum(2 * node + 2, mid + 1, end, l, r);
```

```
    return sum;
}

/**
 * 区间最大值查询
 * 查询区间[left, right]内的最大值
 * @param node 当前线段树节点索引
 * @param start 当前节点表示区间的起始位置
 * @param end 当前节点表示区间的结束位置
 * @param l 查询区间左边界（0-based索引）
 * @param r 查询区间右边界（0-based索引）
 * @return 区间[left, right]内的最大值
 *
 * 时间复杂度: O(log n)
 */
long long rangeMax(int node, int start, int end, int l, int r) {
    // 如果当前区间完全包含在查询区间内
    if (l <= start && end <= r) {
        // 直接返回当前节点的区间最大值
        return tree[node].max_val;
    }

    // 需要向下传递懒惰标记（在递归之前）
    pushDown(node, start, end);
    int mid = (start + end) / 2;
    long long maxVal = -INF;
    // 递归查询左右子树
    if (l <= mid) {
        long long leftMax = rangeMax(2 * node + 1, start, mid, l, r);
        maxVal = (maxVal > leftMax) ? maxVal : leftMax;
    }
    if (r > mid) {
        long long rightMax = rangeMax(2 * node + 2, mid + 1, end, l, r);
        maxVal = (maxVal > rightMax) ? maxVal : rightMax;
    }
    return maxVal;
}

/**
 * 区间最小值查询
 * 查询区间[left, right]内的最小值
 * @param node 当前线段树节点索引
 * @param start 当前节点表示区间的起始位置
```

```
 * @param end 当前节点表示区间的结束位置
 * @param l 查询区间左边界（0-based索引）
 * @param r 查询区间右边界（0-based索引）
 * @return 区间[left, right]内的最小值
 *
 * 时间复杂度: O(log n)
 */
long long rangeMin(int node, int start, int end, int l, int r) {
    // 如果当前区间完全包含在查询区间内
    if (l <= start && end <= r) {
        // 直接返回当前节点的区间最小值
        return tree[node].min_val;
    }

    // 需要向下传递懒惰标记（在递归之前）
    pushDown(node, start, end);
    int mid = (start + end) / 2;
    long long minVal = INF;
    // 递归查询左右子树
    if (l <= mid) {
        long long leftMin = rangeMin(2 * node + 1, start, mid, l, r);
        minVal = (minVal < leftMin) ? minVal : leftMin;
    }
    if (r > mid) {
        long long rightMin = rangeMin(2 * node + 2, mid + 1, end, l, r);
        minVal = (minVal < rightMin) ? minVal : rightMin;
    }
    return minVal;
}

/**
 * 初始化线段树
 * @param nums 原始数组
 * @param size 数组大小
 */
void init(int nums[], int size) {
    n = size;
    int i;
    for (i = 0; i < n; i++) {
        arr[i] = nums[i];
    }
    buildTree(0, 0, n - 1);
}
```

```
/**
 * 区间赋值操作（对外接口）
 * 将区间[left, right]内的每个数都赋值为 val
 * @param left 更新区间左边界（0-based 索引）
 * @param right 更新区间右边界（0-based 索引）
 * @param val 要赋的值
 */
void rangeSetValue(int left, int right, long long val) {
    rangeSet(0, 0, n - 1, left, right, val);
}

/**
 * 区间乘法操作（对外接口）
 * 将区间[left, right]内的每个数都乘以 val
 * @param left 更新区间左边界（0-based 索引）
 * @param right 更新区间右边界（0-based 索引）
 * @param val 要乘的值
 */
void rangeMulValue(int left, int right, long long val) {
    rangeMul(0, 0, n - 1, left, right, val);
}

/**
 * 区间加法操作（对外接口）
 * 将区间[left, right]内的每个数都加上 val
 * @param left 更新区间左边界（0-based 索引）
 * @param right 更新区间右边界（0-based 索引）
 * @param val 要加的值
 */
void rangeAddValue(int left, int right, long long val) {
    rangeAdd(0, 0, n - 1, left, right, val);
}

/**
 * 区间求和查询（对外接口）
 * 查询区间[left, right]内所有数的和
 * @param left 查询区间左边界（0-based 索引）
 * @param right 查询区间右边界（0-based 索引）
 * @return 区间[left, right]内所有数的和
 */
long long rangeSumValue(int left, int right) {
    return rangeSum(0, 0, n - 1, left, right);
```

```
}

/**
 * 区间最大值查询（对外接口）
 * 查询区间[left, right]内的最大值
 * @param left 查询区间左边界（0-based索引）
 * @param right 查询区间右边界（0-based索引）
 * @return 区间[left, right]内的最大值
 */
long long rangeMaxValue(int left, int right) {
    return rangeMax(0, 0, n - 1, left, right);
}

/**
 * 区间最小值查询（对外接口）
 * 查询区间[left, right]内的最小值
 * @param left 查询区间左边界（0-based索引）
 * @param right 查询区间右边界（0-based索引）
 * @return 区间[left, right]内的最小值
 */
long long rangeMinValue(int left, int right) {
    return rangeMin(0, 0, n - 1, left, right);
}

/**
 * 测试函数 - 验证高级线段树实现的正确性
 */
void test() {
    // 测试用例1: 基础功能测试
    int nums1[] = {1, 2, 3, 4, 5};
    init(nums1, 5);

    // 测试初始状态
    // 应该输出 初始数组区间[0,4]的和: 15
    // 应该输出 区间[0,4]的最大值: 5
    // 应该输出 区间[0,4]的最小值: 1

    // 测试区间加法
    rangeAddValue(1, 3, 2);
    // 应该输出 区间加法后区间[1,3]的和: 15 (4+5+6)

    // 测试区间乘法
    rangeMulValue(0, 2, 3);
```

```
    // 应该输出 区间乘法后区间[0,2]的和: 36 (3*3 + 4*3 + 5*3)

    // 测试区间赋值
    rangeSetValue(2, 4, 10);
    // 应该输出 区间赋值后区间[2,4]的和: 30 (10*3)
}

// 主函数 - 处理输入输出和操作调度
int main() {
    // 由于环境限制，这里不实现完整的输入输出处理
    // 实际使用时需要根据具体的输入输出要求进行调整

    // 运行测试
    test();

    return 0;
}
```

==========

文件: Code17_SegmentTreeApplications.java

==========

```
package class112;

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

/**
 * 线段树高级应用 - 多种区间操作
 *
 * 题目描述：
 * 实现支持多种区间操作的线段树，包括：
 * 1. 区间赋值
 * 2. 区间加法
 * 3. 区间乘法
 * 4. 区间求和
 * 5. 区间最大值/最小值
 *
 * 题目来源：洛谷 P3373 【模板】线段树 2
 * 测试链接 : https://www.luogu.com.cn/problem/P3373
 *
 * 解题思路：
 * 使用高级线段树实现支持多种操作，包括区间赋值、加法、乘法以及查询操作。
```

```
 * 通过维护多种懒惰标记来处理不同优先级的操作。
 *
 * 核心思想：
 * 1. 多标记懒惰传播：同时维护加法、乘法和赋值三种懒惰标记
 * 2. 标记优先级：赋值 > 乘法 > 加法
 * 3. 标记下传：在下传标记时需要按照优先级顺序处理
 *
 * 时间复杂度分析：
 * - 构建线段树：O(n)
 * - 所有区间操作：O(log n)
 *
 * 空间复杂度分析：
 * - 线段树需要O(n)的额外空间
 */
public class Code17_SegmentTreeApplications {

    /**
     * 高级线段树实现 - 支持多种操作
     *
     * 特点：
     * 1. 支持区间赋值、加法、乘法操作
     * 2. 支持区间求和、最大值、最小值查询
     * 3. 使用多标记懒惰传播技术
     */
    static class AdvancedSegmentTree {
        /**
         * 线段树节点类
         * 存储区间信息和懒惰标记
         */
        static class Node {
            long sum;        // 区间和
            long max;        // 区间最大值
            long min;        // 区间最小值
            long add;        // 加法懒惰标记
            long mul;        // 乘法懒惰标记
            long set;        // 赋值懒惰标记
            boolean hasSet;  // 是否有赋值标记

            Node() {
                sum = 0;
                max = Long.MIN_VALUE;
                min = Long.MAX_VALUE;
                add = 0;
```

```
        mul = 1;
        set = 0;
        hasSet = false;
    }
}

Node[] tree;  // 线段树数组
int n;        // 数组长度

/**
 * 构造函数 - 初始化高级线段树
 * @param nums 原始数组
 */
public AdvancedSegmentTree(int[] nums) {
    n = nums.length;
    tree = new Node[n * 4];  // 线段树通常需要 4 倍空间
    // 初始化所有节点
    for (int i = 0; i < tree.length; i++) {
        tree[i] = new Node();
    }
    buildTree(nums, 0, 0, n - 1);
}

/**
 * 构建线段树
 * 递归地将数组构建成线段树结构
 * @param nums 原始数组
 * @param node 当前线段树节点索引
 * @param start 当前节点表示区间的起始位置
 * @param end 当前节点表示区间的结束位置
 *
 * 时间复杂度: O(n)
 */
private void buildTree(int[] nums, int node, int start, int end) {
    if (start == end) {
        // 叶子节点 - 直接存储数组元素值
        tree[node].sum = nums[start];
        tree[node].max = nums[start];
        tree[node].min = nums[start];
    } else {
        int mid = (start + end) / 2;
        // 递归构建左子树
        buildTree(nums, 2 * node + 1, start, mid);
```

```
        // 递归构建右子树
        buildTree(nums, 2 * node + 2, mid + 1, end);
        // 向上更新父节点信息
        pushUp(node);
    }
}

/**
 * 向上更新父节点
 * 根据左右子节点的信息更新当前节点的信息
 * @param node 当前线段树节点索引
 */
private void pushUp(int node) {
    int left = 2 * node + 1;   // 左子节点索引
    int right = 2 * node + 2;  // 右子节点索引
    // 更新区间和
    tree[node].sum = tree[left].sum + tree[right].sum;
    // 更新区间最大值
    tree[node].max = Math.max(tree[left].max, tree[right].max);
    // 更新区间最小值
    tree[node].min = Math.min(tree[left].min, tree[right].min);
}

/**
 * 向下传递懒惰标记
 * 按照标记优先级顺序传递标记给子节点
 * 优先级：赋值 > 乘法 > 加法
 * @param node 当前线段树节点索引
 * @param start 当前节点表示区间的起始位置
 * @param end 当前节点表示区间的结束位置
 *
 * 时间复杂度: O(1)
 */
private void pushDown(int node, int start, int end) {
    // 叶子节点不需要传递标记
    if (start == end) return;

    int left = 2 * node + 1;   // 左子节点索引
    int right = 2 * node + 2;  // 右子节点索引
    int mid = (start + end) / 2;

    // 处理赋值标记（优先级最高）
    // 当存在赋值标记时，需要清除子节点的其他标记
```

```
if (tree[node].hasSet) {
    // 更新左子树的区间信息
    tree[left].sum = tree[node].set * (mid - start + 1);  // 区间和 = 赋值 * 区间长度
    tree[right].sum = tree[node].set * (end - mid);
    tree[left].max = tree[node].set;  // 区间最大值 = 赋值
    tree[right].max = tree[node].set;
    tree[left].min = tree[node].set;  // 区间最小值 = 赋值
    tree[right].min = tree[node].set;

    // 传递赋值标记给子节点
    tree[left].set = tree[node].set;
    tree[right].set = tree[node].set;
    tree[left].hasSet = true;
    tree[right].hasSet = true;

    // 清除子节点的其他标记（加法和乘法）
    tree[left].add = 0;
    tree[right].add = 0;
    tree[left].mul = 1;
    tree[right].mul = 1;

    // 清除当前节点的赋值标记
    tree[node].hasSet = false;
}

// 处理乘法标记（优先级次之）
// 当存在乘法标记时，需要更新子节点的所有信息
if (tree[node].mul != 1) {
    // 更新左子树的区间信息（乘以 mul）
    tree[left].sum *= tree[node].mul;
    tree[right].sum *= tree[node].mul;
    tree[left].max *= tree[node].mul;
    tree[right].max *= tree[node].mul;
    tree[left].min *= tree[node].mul;
    tree[right].min *= tree[node].mul;

    // 传递乘法标记给子节点
    tree[left].mul *= tree[node].mul;
    tree[right].mul *= tree[node].mul;
    // 乘法标记也会影响加法标记（add * mul）
    tree[left].add *= tree[node].mul;
    tree[right].add *= tree[node].mul;
```

```
            // 清除当前节点的乘法标记
            tree[node].mul = 1;
        }

        // 处理加法标记（优先级最低）
        // 当存在加法标记时，需要更新子节点的所有信息
        if (tree[node].add != 0) {
            int leftLen = mid - start + 1;   // 左子树区间长度
            int rightLen = end - mid;        // 右子树区间长度

            // 更新左子树的区间信息（加上 add）
            tree[left].sum += tree[node].add * leftLen;  // 区间和增加 add * 区间长度
            tree[right].sum += tree[node].add * rightLen;
            tree[left].max += tree[node].add;  // 区间最大值增加 add
            tree[right].max += tree[node].add;
            tree[left].min += tree[node].add;  // 区间最小值增加 add
            tree[right].min += tree[node].add;

            // 传递加法标记给子节点
            tree[left].add += tree[node].add;
            tree[right].add += tree[node].add;

            // 清除当前节点的加法标记
            tree[node].add = 0;
        }
    }

    /**
     * 区间赋值操作
     * 将区间[left, right]内的每个数都赋值为 val
     * @param left 更新区间左边界（0-based 索引）
     * @param right 更新区间右边界（0-based 索引）
     * @param val 要赋的值
     *
     * 时间复杂度: O(log n)
     */
    public void rangeSet(int left, int right, long val) {
        rangeSetHelper(0, 0, n - 1, left, right, val);
    }

    /**
     * 区间赋值操作辅助函数
     * @param node 当前线段树节点索引
```

```
     * @param start 当前节点表示区间的起始位置
     * @param end 当前节点表示区间的结束位置
     * @param l 更新区间左边界（0-based索引）
     * @param r 更新区间右边界（0-based索引）
     * @param val 要赋的值
     */
    private void rangeSetHelper(int node, int start, int end, int l, int r, long val) {
        // 如果当前区间完全包含在更新区间内
        if (l <= start && end <= r) {
            // 直接更新当前节点的信息和标记
            tree[node].sum = val * (end - start + 1);  // 区间和 = 赋值 * 区间长度
            tree[node].max = val;  // 区间最大值 = 赋值
            tree[node].min = val;  // 区间最小值 = 赋值
            tree[node].set = val;  // 设置赋值标记
            tree[node].hasSet = true;  // 标记存在赋值操作
            tree[node].add = 0;  // 清除加法标记
            tree[node].mul = 1;  // 清除乘法标记
            return;
        }

        // 需要向下传递懒惰标记（在递归之前）
        pushDown(node, start, end);
        int mid = (start + end) / 2;
        // 递归更新左右子树
        if (l <= mid) rangeSetHelper(2 * node + 1, start, mid, l, r, val);
        if (r > mid) rangeSetHelper(2 * node + 2, mid + 1, end, l, r, val);
        // 更新父节点信息
        pushUp(node);
    }

    /**
     * 区间乘法操作
     * 将区间[left, right]内的每个数都乘以val
     * @param left 更新区间左边界（0-based索引）
     * @param right 更新区间右边界（0-based索引）
     * @param val 要乘的值
     *
     * 时间复杂度: O(log n)
     */
    public void rangeMul(int left, int right, long val) {
        rangeMulHelper(0, 0, n - 1, left, right, val);
    }
```

```
/**
 * 区间乘法操作辅助函数
 * @param node 当前线段树节点索引
 * @param start 当前节点表示区间的起始位置
 * @param end 当前节点表示区间的结束位置
 * @param l 更新区间左边界（0-based索引）
 * @param r 更新区间右边界（0-based索引）
 * @param val 要乘的值
 */
private void rangeMulHelper(int node, int start, int end, int l, int r, long val) {
    // 如果当前区间完全包含在更新区间内
    if (l <= start && end <= r) {
        // 直接更新当前节点的信息和标记
        tree[node].sum *= val;  // 区间和乘以val
        tree[node].max *= val;  // 区间最大值乘以val
        tree[node].min *= val;  // 区间最小值乘以val
        tree[node].mul *= val;  // 乘法标记乘以val
        tree[node].add *= val;  // 加法标记也乘以val（因为 a*x + b 变成 a*x*val + b*val）
        return;
    }

    // 需要向下传递懒惰标记（在递归之前）
    pushDown(node, start, end);
    int mid = (start + end) / 2;
    // 递归更新左右子树
    if (l <= mid) rangeMulHelper(2 * node + 1, start, mid, l, r, val);
    if (r > mid) rangeMulHelper(2 * node + 2, mid + 1, end, l, r, val);
    // 更新父节点信息
    pushUp(node);
}

/**
 * 区间加法操作
 * 将区间[left, right]内的每个数都加上val
 * @param left 更新区间左边界（0-based索引）
 * @param right 更新区间右边界（0-based索引）
 * @param val 要加的值
 *
 * 时间复杂度: O(log n)
 */
public void rangeAdd(int left, int right, long val) {
    rangeAddHelper(0, 0, n - 1, left, right, val);
}
```

```
/**
 * 区间加法操作辅助函数
 * @param node 当前线段树节点索引
 * @param start 当前节点表示区间的起始位置
 * @param end 当前节点表示区间的结束位置
 * @param l 更新区间左边界（0-based索引）
 * @param r 更新区间右边界（0-based索引）
 * @param val 要加的值
 */
private void rangeAddHelper(int node, int start, int end, int l, int r, long val) {
    // 如果当前区间完全包含在更新区间内
    if (l <= start && end <= r) {
        // 直接更新当前节点的信息和标记
        tree[node].sum += val * (end - start + 1);  // 区间和增加 val * 区间长度
        tree[node].max += val;  // 区间最大值增加 val
        tree[node].min += val;  // 区间最小值增加 val
        tree[node].add += val;  // 加法标记增加 val
        return;
    }

    // 需要向下传递懒惰标记（在递归之前）
    pushDown(node, start, end);
    int mid = (start + end) / 2;
    // 递归更新左右子树
    if (l <= mid) rangeAddHelper(2 * node + 1, start, mid, l, r, val);
    if (r > mid) rangeAddHelper(2 * node + 2, mid + 1, end, l, r, val);
    // 更新父节点信息
    pushUp(node);
}

/**
 * 区间求和查询
 * 查询区间[left, right]内所有数的和
 * @param left 查询区间左边界（0-based索引）
 * @param right 查询区间右边界（0-based索引）
 * @return 区间[left, right]内所有数的和
 *
 * 时间复杂度: O(log n)
 */
public long rangeSum(int left, int right) {
    return rangeSumHelper(0, 0, n - 1, left, right);
}
```

```
/**
 * 区间求和查询辅助函数
 * @param node 当前线段树节点索引
 * @param start 当前节点表示区间的起始位置
 * @param end 当前节点表示区间的结束位置
 * @param l 查询区间左边界（0-based索引）
 * @param r 查询区间右边界（0-based索引）
 * @return 区间[left, right]内所有数的和
 */
private long rangeSumHelper(int node, int start, int end, int l, int r) {
    // 如果当前区间完全包含在查询区间内
    if (l <= start && end <= r) {
        // 直接返回当前节点的区间和
        return tree[node].sum;
    }

    // 需要向下传递懒惰标记（在递归之前）
    pushDown(node, start, end);
    int mid = (start + end) / 2;
    long sum = 0;
    // 递归查询左右子树
    if (l <= mid) sum += rangeSumHelper(2 * node + 1, start, mid, l, r);
    if (r > mid) sum += rangeSumHelper(2 * node + 2, mid + 1, end, l, r);
    return sum;
}

/**
 * 区间最大值查询
 * 查询区间[left, right]内的最大值
 * @param left 查询区间左边界（0-based索引）
 * @param right 查询区间右边界（0-based索引）
 * @return 区间[left, right]内的最大值
 *
 * 时间复杂度: O(log n)
 */
public long rangeMax(int left, int right) {
    return rangeMaxHelper(0, 0, n - 1, left, right);
}

/**
 * 区间最大值查询辅助函数
 * @param node 当前线段树节点索引
```

```
     * @param start 当前节点表示区间的起始位置
     * @param end 当前节点表示区间的结束位置
     * @param l 查询区间左边界（0-based索引）
     * @param r 查询区间右边界（0-based索引）
     * @return 区间[left, right]内的最大值
     */
    private long rangeMaxHelper(int node, int start, int end, int l, int r) {
        // 如果当前区间完全包含在查询区间内
        if (l <= start && end <= r) {
            // 直接返回当前节点的区间最大值
            return tree[node].max;
        }

        // 需要向下传递懒惰标记（在递归之前）
        pushDown(node, start, end);
        int mid = (start + end) / 2;
        long maxVal = Long.MIN_VALUE;
        // 递归查询左右子树
        if (l <= mid) maxVal = Math.max(maxVal, rangeMaxHelper(2 * node + 1, start, mid, l,
r));
        if (r > mid) maxVal = Math.max(maxVal, rangeMaxHelper(2 * node + 2, mid + 1, end, l,
r));
        return maxVal;
    }

    /**
     * 区间最小值查询
     * 查询区间[left, right]内的最小值
     * @param left 查询区间左边界（0-based索引）
     * @param right 查询区间右边界（0-based索引）
     * @return 区间[left, right]内的最小值
     *
     * 时间复杂度: O(log n)
     */
    public long rangeMin(int left, int right) {
        return rangeMinHelper(0, 0, n - 1, left, right);
    }

    /**
     * 区间最小值查询辅助函数
     * @param node 当前线段树节点索引
     * @param start 当前节点表示区间的起始位置
     * @param end 当前节点表示区间的结束位置
```

```
         * @param l 查询区间左边界（0-based索引）
         * @param r 查询区间右边界（0-based索引）
         * @return 区间[left, right]内的最小值
         */
        private long rangeMinHelper(int node, int start, int end, int l, int r) {
            // 如果当前区间完全包含在查询区间内
            if (l <= start && end <= r) {
                // 直接返回当前节点的区间最小值
                return tree[node].min;
            }

            // 需要向下传递懒惰标记（在递归之前）
            pushDown(node, start, end);
            int mid = (start + end) / 2;
            long minVal = Long.MAX_VALUE;
            // 递归查询左右子树
            if (l <= mid) minVal = Math.min(minVal, rangeMinHelper(2 * node + 1, start, mid, l,
r));
            if (r > mid) minVal = Math.min(minVal, rangeMinHelper(2 * node + 2, mid + 1, end, l,
r));
            return minVal;
        }
    }

    /**
     * 主函数 - 处理输入输出和操作调度
     * 使用高效的输入输出处理方式
     */
    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));

        // 读取n和m
        in.nextToken();
        int n = (int) in.nval;
        in.nextToken();
        int m = (int) in.nval;

        // 读取初始数组
        int[] nums = new int[n];
        for (int i = 0; i < n; i++) {
            in.nextToken();
```

```
        nums[i] = (int) in.nval;
    }

    // 构建高级线段树
    AdvancedSegmentTree st = new AdvancedSegmentTree(nums);

    // 处理 m 个操作
    for (int i = 0; i < m; i++) {
        in.nextToken();
        int op = (int) in.nval;
        in.nextToken();
        int l = (int) in.nval;
        in.nextToken();
        int r = (int) in.nval;

        switch (op) {
            case 1: // 区间加法
                in.nextToken();
                long addVal = (long) in.nval;
                st.rangeAdd(l - 1, r - 1, addVal);
                break;
            case 2: // 区间乘法
                in.nextToken();
                long mulVal = (long) in.nval;
                st.rangeMul(l - 1, r - 1, mulVal);
                break;
            case 3: // 区间赋值
                in.nextToken();
                long setVal = (long) in.nval;
                st.rangeSet(l - 1, r - 1, setVal);
                break;
            case 4: // 区间求和
                out.println(st.rangeSum(l - 1, r - 1));
                break;
            case 5: // 区间最大值
                out.println(st.rangeMax(l - 1, r - 1));
                break;
            case 6: // 区间最小值
                out.println(st.rangeMin(l - 1, r - 1));
                break;
        }
    }
```

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

    /**
     * 测试方法 - 验证高级线段树实现的正确性
     */
    public static void test() {
        System.out.println("=== 高级线段树测试 ===");

        int[] nums = {1, 2, 3, 4, 5};
        AdvancedSegmentTree st = new AdvancedSegmentTree(nums);

        // 测试初始状态
        System.out.println("初始数组区间[0,4]的和: " + st.rangeSum(0, 4)); // 15
        System.out.println("区间[0,4]的最大值: " + st.rangeMax(0, 4)); // 5
        System.out.println("区间[0,4]的最小值: " + st.rangeMin(0, 4)); // 1

        // 测试区间加法
        st.rangeAdd(1, 3, 2);
        System.out.println("区间加法后区间[1,3]的和: " + st.rangeSum(1, 3)); // 4+5+6 = 15

        // 测试区间乘法
        st.rangeMul(0, 2, 3);
        System.out.println("区间乘法后区间[0,2]的和: " + st.rangeSum(0, 2)); // 3*3 + 4*3 + 5*3 =
36

        // 测试区间赋值
        st.rangeSet(2, 4, 10);
        System.out.println("区间赋值后区间[2,4]的和: " + st.rangeSum(2, 4)); // 10*3 = 30

        System.out.println("=== 测试完成 ===");
    }
}

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

文件: Code17_SegmentTreeApplications.py
==============================================
"""
线段树高级应用 - 多种区间操作
```

题目描述：
实现支持多种区间操作的线段树，包括：
1. 区间赋值
2. 区间加法
3. 区间乘法
4. 区间求和
5. 区间最大值/最小值

题目来源：洛谷 P3373 【模板】线段树 2
测试链接 : https://www.luogu.com.cn/problem/P3373

解题思路：
使用高级线段树实现支持多种操作，包括区间赋值、加法、乘法以及查询操作。
通过维护多种懒惰标记来处理不同优先级的操作。

核心思想：
1. 多标记懒惰传播：同时维护加法、乘法和赋值三种懒惰标记
2. 标记优先级：赋值 > 乘法 > 加法
3. 标记下传：在下传标记时需要按照优先级顺序处理

时间复杂度分析：
- 构建线段树：O(n)
- 所有区间操作：O(log n)

空间复杂度分析：
- 线段树需要O(n)的额外空间
"""

```
import sys
from typing import List

class Node:
    """线段树节点类"""

    def __init__(self):
        self.sum = 0          # 区间和
        self.max = -float('inf')  # 区间最大值
        self.min = float('inf')   # 区间最小值
        self.add = 0          # 加法懒惰标记
        self.mul = 1          # 乘法懒惰标记
        self.set = 0          # 赋值懒惰标记
        self.has_set = False  # 是否有赋值标记
```

```
class AdvancedSegmentTree:
    """高级线段树实现 - 支持多种操作"""

    def __init__(self, nums: List[int]):
        """
        构造函数 - 初始化高级线段树
        :param nums: 原始数组
        """
        self.n = len(nums)
        # 线段树通常需要 4 倍空间
        self.tree = [Node() for _ in range(self.n * 4)]
        self.build_tree(nums, 0, 0, self.n - 1)

    def build_tree(self, nums: List[int], node: int, start: int, end: int) -> None:
        """
        构建线段树
        递归地将数组构建成线段树结构
        :param nums: 原始数组
        :param node: 当前线段树节点索引
        :param start: 当前节点表示区间的起始位置
        :param end: 当前节点表示区间的结束位置

        时间复杂度: O(n)
        """
        if start == end:
            # 叶子节点 - 直接存储数组元素值
            self.tree[node].sum = nums[start]
            self.tree[node].max = nums[start]
            self.tree[node].min = nums[start]
        else:
            mid = (start + end) // 2
            # 递归构建左子树
            self.build_tree(nums, 2 * node + 1, start, mid)
            # 递归构建右子树
            self.build_tree(nums, 2 * node + 2, mid + 1, end)
            # 向上更新父节点信息
            self.push_up(node)

    def push_up(self, node: int) -> None:
        """
        向上更新父节点
        根据左右子节点的信息更新当前节点的信息
```

```
        :param node: 当前线段树节点索引
        """
        left = 2 * node + 1   # 左子节点索引
        right = 2 * node + 2  # 右子节点索引
        # 更新区间和
        self.tree[node].sum = self.tree[left].sum + self.tree[right].sum
        # 更新区间最大值
        self.tree[node].max = max(self.tree[left].max, self.tree[right].max)
        # 更新区间最小值
        self.tree[node].min = min(self.tree[left].min, self.tree[right].min)

    def push_down(self, node: int, start: int, end: int) -> None:
        """
        向下传递懒惰标记
        按照标记优先级顺序传递标记给子节点
        优先级：赋值 > 乘法 > 加法
        :param node: 当前线段树节点索引
        :param start: 当前节点表示区间的起始位置
        :param end: 当前节点表示区间的结束位置

        时间复杂度: O(1)
        """
        # 叶子节点不需要传递标记
        if start == end:
            return

        left = 2 * node + 1   # 左子节点索引
        right = 2 * node + 2  # 右子节点索引
        mid = (start + end) // 2

        # 处理赋值标记（优先级最高）
        # 当存在赋值标记时，需要清除子节点的其他标记
        if self.tree[node].has_set:
            # 更新左子树的区间信息
            self.tree[left].sum = self.tree[node].set * (mid - start + 1)  # 区间和 = 赋值 * 区间
长度
            self.tree[right].sum = self.tree[node].set * (end - mid)
            self.tree[left].max = self.tree[node].set  # 区间最大值 = 赋值
            self.tree[right].max = self.tree[node].set
            self.tree[left].min = self.tree[node].set  # 区间最小值 = 赋值
            self.tree[right].min = self.tree[node].set

            # 传递赋值标记给子节点
```

```
        self.tree[left].set = self.tree[node].set
        self.tree[right].set = self.tree[node].set
        self.tree[left].has_set = True
        self.tree[right].has_set = True

        # 清除子节点的其他标记（加法和乘法）
        self.tree[left].add = 0
        self.tree[right].add = 0
        self.tree[left].mul = 1
        self.tree[right].mul = 1

        # 清除当前节点的赋值标记
        self.tree[node].has_set = False

    # 处理乘法标记（优先级次之）
    # 当存在乘法标记时，需要更新子节点的所有信息
    if self.tree[node].mul != 1:
        # 更新左子树的区间信息（乘以 mul）
        self.tree[left].sum *= self.tree[node].mul
        self.tree[right].sum *= self.tree[node].mul
        self.tree[left].max *= self.tree[node].mul
        self.tree[right].max *= self.tree[node].mul
        self.tree[left].min *= self.tree[node].mul
        self.tree[right].min *= self.tree[node].mul

        # 传递乘法标记给子节点
        self.tree[left].mul *= self.tree[node].mul
        self.tree[right].mul *= self.tree[node].mul
        # 乘法标记也会影响加法标记（add * mul）
        self.tree[left].add *= self.tree[node].mul
        self.tree[right].add *= self.tree[node].mul

        # 清除当前节点的乘法标记
        self.tree[node].mul = 1

    # 处理加法标记（优先级最低）
    # 当存在加法标记时，需要更新子节点的所有信息
    if self.tree[node].add != 0:
        left_len = mid - start + 1   # 左子树区间长度
        right_len = end - mid        # 右子树区间长度

        # 更新左子树的区间信息（加上 add）
        self.tree[left].sum += self.tree[node].add * left_len  # 区间和增加 add * 区间长度
```

```
            self.tree[right].sum += self.tree[node].add * right_len
            self.tree[left].max += self.tree[node].add  # 区间最大值增加 add
            self.tree[right].max += self.tree[node].add
            self.tree[left].min += self.tree[node].add  # 区间最小值增加 add
            self.tree[right].min += self.tree[node].add

            # 传递加法标记给子节点
            self.tree[left].add += self.tree[node].add
            self.tree[right].add += self.tree[node].add

            # 清除当前节点的加法标记
            self.tree[node].add = 0

    def range_set(self, left: int, right: int, val: int) -> None:
        """
        区间赋值操作
        将区间[left, right]内的每个数都赋值为 val
        :param left: 更新区间左边界（0-based 索引）
        :param right: 更新区间右边界（0-based 索引）
        :param val: 要赋的值

        时间复杂度: O(log n)
        """
        self._range_set_helper(0, 0, self.n - 1, left, right, val)

    def _range_set_helper(self, node: int, start: int, end: int, l: int, r: int, val: int) ->
None:
        """
        区间赋值操作辅助函数
        :param node: 当前线段树节点索引
        :param start: 当前节点表示区间的起始位置
        :param end: 当前节点表示区间的结束位置
        :param l: 更新区间左边界（0-based 索引）
        :param r: 更新区间右边界（0-based 索引）
        :param val: 要赋的值
        """
        # 如果当前区间完全包含在更新区间内
        if l <= start and end <= r:
            # 直接更新当前节点的信息和标记
            self.tree[node].sum = val * (end - start + 1)  # 区间和 = 赋值 * 区间长度
            self.tree[node].max = val  # 区间最大值 = 赋值
            self.tree[node].min = val  # 区间最小值 = 赋值
            self.tree[node].set = val  # 设置赋值标记
```

```
            self.tree[node].has_set = True  # 标记存在赋值操作
            self.tree[node].add = 0  # 清除加法标记
            self.tree[node].mul = 1  # 清除乘法标记
            return

        # 需要向下传递懒惰标记（在递归之前）
        self.push_down(node, start, end)
        mid = (start + end) // 2
        # 递归更新左右子树
        if l <= mid:
            self._range_set_helper(2 * node + 1, start, mid, l, r, val)
        if r > mid:
            self._range_set_helper(2 * node + 2, mid + 1, end, l, r, val)
        # 更新父节点信息
        self.push_up(node)

    def range_mul(self, left: int, right: int, val: int) -> None:
        """
        区间乘法操作
        将区间[left, right]内的每个数都乘以 val
        :param left: 更新区间左边界（0-based 索引）
        :param right: 更新区间右边界（0-based 索引）
        :param val: 要乘的值

        时间复杂度: O(log n)
        """
        self._range_mul_helper(0, 0, self.n - 1, left, right, val)

    def _range_mul_helper(self, node: int, start: int, end: int, l: int, r: int, val: int) ->
None:
        """
        区间乘法操作辅助函数
        :param node: 当前线段树节点索引
        :param start: 当前节点表示区间的起始位置
        :param end: 当前节点表示区间的结束位置
        :param l: 更新区间左边界（0-based 索引）
        :param r: 更新区间右边界（0-based 索引）
        :param val: 要乘的值
        """
        # 如果当前区间完全包含在更新区间内
        if l <= start and end <= r:
            # 直接更新当前节点的信息和标记
            self.tree[node].sum *= val  # 区间和乘以 val
```

```
            self.tree[node].max *= val  # 区间最大值乘以 val
            self.tree[node].min *= val  # 区间最小值乘以 val
            self.tree[node].mul *= val  # 乘法标记乘以 val
            self.tree[node].add *= val  # 加法标记也乘以 val（因为 a*x + b 变成 a*x*val + b*val）
            return

        # 需要向下传递懒惰标记（在递归之前）
        self.push_down(node, start, end)
        mid = (start + end) // 2
        # 递归更新左右子树
        if l <= mid:
            self._range_mul_helper(2 * node + 1, start, mid, l, r, val)
        if r > mid:
            self._range_mul_helper(2 * node + 2, mid + 1, end, l, r, val)
        # 更新父节点信息
        self.push_up(node)

    def range_add(self, left: int, right: int, val: int) -> None:
        """
        区间加法操作
        将区间[left, right]内的每个数都加上 val
        :param left: 更新区间左边界（0-based 索引）
        :param right: 更新区间右边界（0-based 索引）
        :param val: 要加的值

        时间复杂度: O(log n)
        """
        self._range_add_helper(0, 0, self.n - 1, left, right, val)

    def _range_add_helper(self, node: int, start: int, end: int, l: int, r: int, val: int) ->
None:
        """
        区间加法操作辅助函数
        :param node: 当前线段树节点索引
        :param start: 当前节点表示区间的起始位置
        :param end: 当前节点表示区间的结束位置
        :param l: 更新区间左边界（0-based 索引）
        :param r: 更新区间右边界（0-based 索引）
        :param val: 要加的值
        """
        # 如果当前区间完全包含在更新区间内
        if l <= start and end <= r:
            # 直接更新当前节点的信息和标记
```

```
            self.tree[node].sum += val * (end - start + 1)  # 区间和增加 val * 区间长度
            self.tree[node].max += val  # 区间最大值增加 val
            self.tree[node].min += val  # 区间最小值增加 val
            self.tree[node].add += val  # 加法标记增加 val
            return

        # 需要向下传递懒惰标记（在递归之前）
        self.push_down(node, start, end)
        mid = (start + end) // 2
        # 递归更新左右子树
        if l <= mid:
            self._range_add_helper(2 * node + 1, start, mid, l, r, val)
        if r > mid:
            self._range_add_helper(2 * node + 2, mid + 1, end, l, r, val)
        # 更新父节点信息
        self.push_up(node)

    def range_sum(self, left: int, right: int) -> int:
        """
        区间求和查询
        查询区间[left, right]内所有数的和
        :param left: 查询区间左边界（0-based索引）
        :param right: 查询区间右边界（0-based索引）
        :return: 区间[left, right]内所有数的和

        时间复杂度: O(log n)
        """
        return self._range_sum_helper(0, 0, self.n - 1, left, right)

    def _range_sum_helper(self, node: int, start: int, end: int, l: int, r: int) -> int:
        """
        区间求和查询辅助函数
        :param node: 当前线段树节点索引
        :param start: 当前节点表示区间的起始位置
        :param end: 当前节点表示区间的结束位置
        :param l: 查询区间左边界（0-based索引）
        :param r: 查询区间右边界（0-based索引）
        :return: 区间[left, right]内所有数的和
        """
        # 如果当前区间完全包含在查询区间内
        if l <= start and end <= r:
            # 直接返回当前节点的区间和
            return self.tree[node].sum
```

```
    # 需要向下传递懒惰标记（在递归之前）
    self.push_down(node, start, end)
    mid = (start + end) // 2
    total = 0
    # 递归查询左右子树
    if l <= mid:
        total += self._range_sum_helper(2 * node + 1, start, mid, l, r)
    if r > mid:
        total += self._range_sum_helper(2 * node + 2, mid + 1, end, l, r)
    return total

def range_max(self, left: int, right: int) -> float:
    """
    区间最大值查询
    查询区间[left, right]内的最大值
    :param left: 查询区间左边界（0-based索引）
    :param right: 查询区间右边界（0-based索引）
    :return: 区间[left, right]内的最大值

    时间复杂度: O(log n)
    """
    return self._range_max_helper(0, 0, self.n - 1, left, right)

def _range_max_helper(self, node: int, start: int, end: int, l: int, r: int) -> float:
    """
    区间最大值查询辅助函数
    :param node: 当前线段树节点索引
    :param start: 当前节点表示区间的起始位置
    :param end: 当前节点表示区间的结束位置
    :param l: 查询区间左边界（0-based索引）
    :param r: 查询区间右边界（0-based索引）
    :return: 区间[left, right]内的最大值
    """
    # 如果当前区间完全包含在查询区间内
    if l <= start and end <= r:
        # 直接返回当前节点的区间最大值
        return self.tree[node].max

    # 需要向下传递懒惰标记（在递归之前）
    self.push_down(node, start, end)
    mid = (start + end) // 2
    max_val = -float('inf')
```

```
    # 递归查询左右子树
    if l <= mid:
        max_val = max(max_val, self._range_max_helper(2 * node + 1, start, mid, l, r))
    if r > mid:
        max_val = max(max_val, self._range_max_helper(2 * node + 2, mid + 1, end, l, r))
    return max_val

def range_min(self, left: int, right: int) -> float:
    """
    区间最小值查询
    查询区间[left, right]内的最小值
    :param left: 查询区间左边界（0-based索引）
    :param right: 查询区间右边界（0-based索引）
    :return: 区间[left, right]内的最小值

    时间复杂度: O(log n)
    """
    return self._range_min_helper(0, 0, self.n - 1, left, right)

def _range_min_helper(self, node: int, start: int, end: int, l: int, r: int) -> float:
    """
    区间最小值查询辅助函数
    :param node: 当前线段树节点索引
    :param start: 当前节点表示区间的起始位置
    :param end: 当前节点表示区间的结束位置
    :param l: 查询区间左边界（0-based索引）
    :param r: 查询区间右边界（0-based索引）
    :return: 区间[left, right]内的最小值
    """
    # 如果当前区间完全包含在查询区间内
    if l <= start and end <= r:
        # 直接返回当前节点的区间最小值
        return self.tree[node].min

    # 需要向下传递懒惰标记（在递归之前）
    self.push_down(node, start, end)
    mid = (start + end) // 2
    min_val = float('inf')
    # 递归查询左右子树
    if l <= mid:
        min_val = min(min_val, self._range_min_helper(2 * node + 1, start, mid, l, r))
    if r > mid:
        min_val = min(min_val, self._range_min_helper(2 * node + 2, mid + 1, end, l, r))
```

```
        return min_val


def main():
    """
    主函数 - 处理输入输出和操作调度
    """
    # 读取 n 和 m
    n, m = map(int, input().split())

    # 读取初始数组
    nums = list(map(int, input().split()))

    # 构建高级线段树
    st = AdvancedSegmentTree(nums)

    # 处理 m 个操作
    for _ in range(m):
        operation = list(map(int, input().split()))
        op = operation[0]
        l = operation[1]
        r = operation[2]

        if op == 1:  # 区间加法
            add_val = operation[3]
            st.range_add(l - 1, r - 1, add_val)
        elif op == 2:  # 区间乘法
            mul_val = operation[3]
            st.range_mul(l - 1, r - 1, mul_val)
        elif op == 3:  # 区间赋值
            set_val = operation[3]
            st.range_set(l - 1, r - 1, set_val)
        elif op == 4:  # 区间求和
            print(st.range_sum(l - 1, r - 1))
        elif op == 5:  # 区间最大值
            print(int(st.range_max(l - 1, r - 1)))
        elif op == 6:  # 区间最小值
            print(int(st.range_min(l - 1, r - 1)))


def test():
    """
    测试函数 - 验证高级线段树实现的正确性
```

```
    """
    print("=== 高级线段树测试 ===")

    nums = [1, 2, 3, 4, 5]
    st = AdvancedSegmentTree(nums)

    # 测试初始状态
    print(f"初始数组区间[0,4]的和: {st.range_sum(0, 4)}")  # 15
    print(f"区间[0,4]的最大值: {int(st.range_max(0, 4))}")  # 5
    print(f"区间[0,4]的最小值: {int(st.range_min(0, 4))}")  # 1

    # 测试区间加法
    st.range_add(1, 3, 2)
    print(f"区间加法后区间[1,3]的和: {st.range_sum(1, 3)}")  # 4+5+6 = 15

    # 测试区间乘法
    st.range_mul(0, 2, 3)
    print(f"区间乘法后区间[0,2]的和: {st.range_sum(0, 2)}")  # 3*3 + 4*3 + 5*3 = 36

    # 测试区间赋值
    st.range_set(2, 4, 10)
    print(f"区间赋值后区间[2,4]的和: {st.range_sum(2, 4)}")  # 10*3 = 30

    print("=== 测试完成 ===")


# 如果需要运行测试，取消下面的注释
# test()

# 如果需要运行主程序，取消下面的注释
# if __name__ == "__main__":
#     main()

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

文件: Code18_DynamicSegmentTree.cpp
================================================
/**
 * 动态开点线段树 - 适用于值域较大或稀疏数据的场景
 *
 * 题目描述：
 * 实现支持动态开点的线段树，避免预分配大量空间
 * 应用场景：值域很大但实际数据稀疏的情况
```

```
 *
 * 题目来源：LeetCode 327. 区间和的个数
 * 测试链接 : https://leetcode.cn/problems/count-of-range-sum/
 *
 * 解题思路：
 * 使用动态开点线段树来处理值域较大但数据稀疏的情况。
 * 与传统线段树不同，动态开点线段树只在需要时创建节点，节省空间。
 *
 * 核心思想：
 * 1. 动态开点：只在需要时创建线段树节点，避免预分配大量空间
 * 2. 按需分配：对于值域很大的情况，只创建实际用到的节点
 * 3. 节点结构：每个节点维护区间信息和左右子节点指针
 *
 * 时间复杂度分析：
 * - 更新操作：O(log(maxVal-minVal))
 * - 查询操作：O(log(maxVal-minVal))
 *
 * 空间复杂度分析：
 * - O(n)，其中 n 是实际插入的元素个数
 */

// 定义常量
#define MAXN 100005
#define INF 1000000000000000LL

// 前向声明
struct DynamicSegmentTreeNode;

// 动态开点线段树节点结构
struct DynamicSegmentTreeNode {
    long long min_val;              // 节点管理区间的最小值
    long long max_val;              // 节点管理区间的最大值
    int count;                      // 节点管理区间内的元素个数
    DynamicSegmentTreeNode* left;   // 左子节点
    DynamicSegmentTreeNode* right;  // 右子节点

    /**
     * 构造函数 - 创建线段树节点
     * @param min 节点管理区间的最小值
     * @param max 节点管理区间的最大值
     */
    DynamicSegmentTreeNode(long long min, long long max) {
        min_val = min;
```

```
    max_val = max;
    count = 0;
    left = 0;  // 使用0代替NULL
    right = 0; // 使用0代替NULL
}

/**
 * 析构函数 - 释放节点内存
 */
~DynamicSegmentTreeNode() {
    if (left != 0) {     // 使用0代替NULL
        delete left;
    }
    if (right != 0) {    // 使用0代替NULL
        delete right;
    }
}

/**
 * 获取区间中点
 * 用于将当前区间分成左右两个子区间
 * @return 区间中点值
 */
long long getMid() {
    return min_val + (max_val - min_val) / 2;
}

/**
 * 更新操作 - 在线段树中插入一个值
 * @param val 要插入的值
 *
 * 时间复杂度: O(log(maxVal-minVal))
 */
void update(long long val) {
    // 如果当前节点管理的区间只有一个值（叶子节点）
    if (min_val == max_val) {
        // 直接增加计数
        count++;
        return;
    }

    // 计算区间中点
    long long mid = getMid();
```

```
        // 根据值的大小决定插入左子树还是右子树
        if (val <= mid) {
            // 如果左子节点不存在，则创建左子节点
            if (left == 0) {     // 使用 0 代替 NULL
                left = new DynamicSegmentTreeNode(min_val, mid);
            }
            // 递归更新左子树
            left->update(val);
        } else {
            // 如果右子节点不存在，则创建右子节点
            if (right == 0) {    // 使用 0 代替 NULL
                right = new DynamicSegmentTreeNode(mid + 1, max_val);
            }
            // 递归更新右子树
            right->update(val);
        }

        // 更新当前节点的统计信息（左右子树元素个数之和）
        count = (left != 0 ? left->count : 0) + (right != 0 ? right->count : 0);  // 使用 0 代替
NULL
    }

    /**
     * 查询操作 - 查询区间 [l, r] 内的元素个数
     * @param l 查询区间左边界
     * @param r 查询区间右边界
     * @return 区间[l, r]内的元素个数
     *
     * 时间复杂度: O(log(maxVal-minVal))
     */
    int query(long long l, long long r) {
        // 如果查询区间与当前节点管理的区间无交集
        if (l > max_val || r < min_val) {
            // 返回 0
            return 0;
        }
        // 如果当前节点管理的区间完全包含在查询区间内
        if (l <= min_val && max_val <= r) {
            // 直接返回当前节点的元素个数
            return count;
        }

        // 计算区间中点
```

```
        long long mid = getMid();
        int result = 0;
        // 如果左子节点存在且查询区间与左子树区间有交集
        if (left != 0 && l <= mid) {     // 使用0代替NULL
            // 递归查询左子树
            result += left->query(l, r);
        }
        // 如果右子节点存在且查询区间与右子树区间有交集
        if (right != 0 && r > mid) {     // 使用0代替NULL
            // 递归查询右子树
            result += right->query(l, r);
        }

        return result;
    }
};

/**
 * 动态开点线段树
 * 管理整个线段树的根节点和值域范围
 */
class DynamicSegmentTree {
private:
    DynamicSegmentTreeNode* root;  // 线段树根节点
    long long min_val;             // 值域最小值
    long long max_val;             // 值域最大值

public:
    /**
     * 构造函数 - 创建动态开点线段树
     * @param minVal 值域最小值
     * @param maxVal 值域最大值
     */
    DynamicSegmentTree(long long minVal, long long maxVal) {
        min_val = minVal;
        max_val = maxVal;
        // 创建根节点，管理整个值域范围
        root = new DynamicSegmentTreeNode(minVal, maxVal);
    }

    /**
     * 析构函数 - 释放线段树内存
     */
```

```
    ~DynamicSegmentTree() {
        if (root != 0) {     // 使用0代替NULL
            delete root;
        }
    }

    /**
     * 插入一个值
     * @param val 要插入的值
     *
     * 时间复杂度: O(log(maxVal-minVal))
     */
    void update(long long val) {
        root->update(val);
    }

    /**
     * 查询区间 [l, r] 内的元素个数
     * @param l 查询区间左边界
     * @param r 查询区间右边界
     * @return 区间[l, r]内的元素个数
     *
     * 时间复杂度: O(log(maxVal-minVal))
     */
    int query(long long l, long long r) {
        return root->query(l, r);
    }
};

/**
 * 测试动态开点线段树的基本功能
 */
void testDynamicSegmentTree() {
    // 创建动态开点线段树，值域为 [-1000, 1000]
    DynamicSegmentTree tree(-1000, 1000);

    // 插入一些值
    tree.update(10);
    tree.update(20);
    tree.update(30);
    tree.update(40);
    tree.update(50);
```

```
    // 测试查询
    // 应该输出 区间[15,35]内的元素个数: 2 (20, 30)
    // 应该输出 区间[0,100]内的元素个数: 5
    // 应该输出 区间[-10,10]内的元素个数: 1 (10)

    // 插入更多值
    tree.update(5);
    tree.update(15);
    tree.update(25);

    // 应该输出 插入后区间[0,30]内的元素个数: 6
}

/**
 * 动态开点线段树的其他应用：求逆序对
 * 使用动态开点线段树求解数组中的逆序对个数
 *
 * 解题思路：
 * 逆序对是指对于数组中的两个元素 nums[i]和 nums[j]，如果 i<j 且 nums[i]>nums[j]，则构成一个逆序对
 * 我们从右向左遍历数组，对于每个元素 nums[i]，统计右侧有多少个元素比它小
 *
 * 时间复杂度: O(n log maxVal)
 * 空间复杂度: O(n)
 *
 * @param nums 整数数组
 * @param n 数组长度
 * @return 数组中的逆序对个数
 */
int countInversions(int nums[], int n) {
    // 边界条件检查
    if (n == 0) {
        return 0;
    }

    // 获取数组值的范围，用于确定动态开点线段树的值域
    int minVal = nums[0];
    int maxVal = nums[0];
    int i;
    for (i = 1; i < n; i++) {
        if (nums[i] < minVal) {
            minVal = nums[i];
        }
        if (nums[i] > maxVal) {
```

```
            maxVal = nums[i];
        }
    }

    // 创建动态开点线段树
    DynamicSegmentTree tree(minVal, maxVal);

    int inversions = 0;
    // 从右向左遍历，统计每个元素右侧比它小的元素个数
    for (i = n - 1; i >= 0; i--) {
        // 查询[minVal, nums[i]-1]范围内的元素个数，即右侧比 nums[i]小的元素个数
        inversions += tree.query(minVal, nums[i] - 1);
        // 将当前元素插入线段树
        tree.update(nums[i]);
    }

    return inversions;
}

/**
 * 测试逆序对计数
 */
void testInversions() {
    // 测试用例 1
    int nums1[] = {2, 4, 1, 3, 5};
    int result1 = countInversions(nums1, 5);
    // 应该输出 逆序对个数: 3

    // 测试用例 2
    int nums2[] = {5, 4, 3, 2, 1};
    int result2 = countInversions(nums2, 5);
    // 应该输出 逆序对个数: 10

    // 测试用例 3
    int nums3[] = {1, 2, 3, 4, 5};
    int result3 = countInversions(nums3, 5);
    // 应该输出 逆序对个数: 0
}

// 主函数 - 测试动态开点线段树的实现
int main() {
    // 运行测试
    testDynamicSegmentTree();
```

```
    testInversions();

    return 0;
}
```

==========

文件: Code18_DynamicSegmentTree.java

==========

```
package class112;

import java.util.*;

/**
 * 动态开点线段树 - 适用于值域较大或稀疏数据的场景
 *
 * 题目描述：
 * 实现支持动态开点的线段树，避免预分配大量空间
 * 应用场景：值域很大但实际数据稀疏的情况
 *
 * 题目来源：LeetCode 327. 区间和的个数
 * 测试链接 : https://leetcode.cn/problems/count-of-range-sum/
 *
 * 解题思路：
 * 使用动态开点线段树来处理值域较大但数据稀疏的情况。
 * 与传统线段树不同，动态开点线段树只在需要时创建节点，节省空间。
 *
 * 核心思想：
 * 1. 动态开点：只在需要时创建线段树节点，避免预分配大量空间
 * 2. 按需分配：对于值域很大的情况，只创建实际用到的节点
 * 3. 节点结构：每个节点维护区间信息和左右子节点指针
 *
 * 时间复杂度分析：
 * - 更新操作：O(log(maxVal-minVal))
 * - 查询操作：O(log(maxVal-minVal))
 *
 * 空间复杂度分析：
 * - O(n)，其中 n 是实际插入的元素个数
 */
public class Code18_DynamicSegmentTree {

    /**
     * 动态开点线段树节点
```

```
 * 与传统线段树不同，动态开点线段树的节点是按需创建的
 */
static class DynamicSegmentTreeNode {
    long min;  // 节点管理区间的最小值
    long max;  // 节点管理区间的最大值
    int count; // 节点管理区间内的元素个数
    DynamicSegmentTreeNode left;  // 左子节点
    DynamicSegmentTreeNode right; // 右子节点

    /**
     * 构造函数 - 创建线段树节点
     * @param min 节点管理区间的最小值
     * @param max 节点管理区间的最大值
     */
    DynamicSegmentTreeNode(long min, long max) {
        this.min = min;
        this.max = max;
        this.count = 0;
        this.left = null;
        this.right = null;
    }

    /**
     * 获取区间中点
     * 用于将当前区间分成左右两个子区间
     * @return 区间中点值
     */
    private long getMid() {
        return min + (max - min) / 2;
    }

    /**
     * 更新操作 - 在线段树中插入一个值
     * @param val 要插入的值
     *
     * 时间复杂度: O(log(maxVal-minVal))
     */
    public void update(long val) {
        // 如果当前节点管理的区间只有一个值（叶子节点）
        if (min == max) {
            // 直接增加计数
            count++;
            return;
```

```
    }

    // 计算区间中点
    long mid = getMid();
    // 根据值的大小决定插入左子树还是右子树
    if (val <= mid) {
        // 如果左子节点不存在，则创建左子节点
        if (left == null) {
            left = new DynamicSegmentTreeNode(min, mid);
        }
        // 递归更新左子树
        left.update(val);
    } else {
        // 如果右子节点不存在，则创建右子节点
        if (right == null) {
            right = new DynamicSegmentTreeNode(mid + 1, max);
        }
        // 递归更新右子树
        right.update(val);
    }

    // 更新当前节点的统计信息（左右子树元素个数之和）
    count = (left != null ? left.count : 0) + (right != null ? right.count : 0);
}

/**
 * 查询操作 - 查询区间 [l, r] 内的元素个数
 * @param l 查询区间左边界
 * @param r 查询区间右边界
 * @return 区间[l, r]内的元素个数
 *
 * 时间复杂度: O(log(maxVal-minVal))
 */
public int query(long l, long r) {
    // 如果查询区间与当前节点管理的区间无交集
    if (l > max || r < min) {
        // 返回 0
        return 0;
    }
    // 如果当前节点管理的区间完全包含在查询区间内
    if (l <= min && max <= r) {
        // 直接返回当前节点的元素个数
        return count;
```

```
        }

        // 计算区间中点
        long mid = getMid();
        int result = 0;
        // 如果左子节点存在且查询区间与左子树区间有交集
        if (left != null && l <= mid) {
            // 递归查询左子树
            result += left.query(l, r);
        }
        // 如果右子节点存在且查询区间与右子树区间有交集
        if (right != null && r > mid) {
            // 递归查询右子树
            result += right.query(l, r);
        }

        return result;
    }
}

/**
 * 动态开点线段树
 * 管理整个线段树的根节点和值域范围
 */
static class DynamicSegmentTree {
    DynamicSegmentTreeNode root;  // 线段树根节点
    long minVal;                  // 值域最小值
    long maxVal;                  // 值域最大值

    /**
     * 构造函数 - 创建动态开点线段树
     * @param minVal 值域最小值
     * @param maxVal 值域最大值
     */
    public DynamicSegmentTree(long minVal, long maxVal) {
        this.minVal = minVal;
        this.maxVal = maxVal;
        // 创建根节点，管理整个值域范围
        this.root = new DynamicSegmentTreeNode(minVal, maxVal);
    }

    /**
     * 插入一个值
```

```
     * @param val 要插入的值
     *
     * 时间复杂度: O(log(maxVal-minVal))
     */
    public void update(long val) {
        root.update(val);
    }

    /**
     * 查询区间 [l, r] 内的元素个数
     * @param l 查询区间左边界
     * @param r 查询区间右边界
     * @return 区间[l, r]内的元素个数
     *
     * 时间复杂度: O(log(maxVal-minVal))
     */
    public int query(long l, long r) {
        return root.query(l, r);
    }
}

/**
 * LeetCode 327. 区间和的个数
 * 给定一个整数数组 nums，返回区间和在 [lower, upper] 之间的区间个数
 *
 * 解题思路：
 * 使用前缀和 + 动态开点线段树的方法
 * 1. 计算前缀和数组
 * 2. 对于每个前缀和 prefixSum[i]，我们需要统计有多少个 j>i 满足：
 *    lower <= prefixSum[j] - prefixSum[i] <= upper
 *    即：prefixSum[i] + lower <= prefixSum[j] <= prefixSum[i] + upper
 * 3. 从右向左遍历前缀和数组，使用动态开点线段树维护已遍历的前缀和
 *
 * 时间复杂度: O(n log S) 其中 S 是前缀和的范围
 * 空间复杂度: O(n)
 *
 * @param nums 整数数组
 * @param lower 区间下界
 * @param upper 区间上界
 * @return 区间和在[lower, upper]之间的区间个数
 */
public int countRangeSum(int[] nums, int lower, int upper) {
    // 边界条件检查
```

```
    if (nums == null || nums.length == 0) {
        return 0;
    }

    // 计算前缀和数组
    int n = nums.length;
    long[] prefixSum = new long[n + 1];
    for (int i = 0; i < n; i++) {
        prefixSum[i + 1] = prefixSum[i] + nums[i];
    }

    // 获取前缀和的范围，用于确定动态开点线段树的值域
    long minPrefix = Long.MAX_VALUE;
    long maxPrefix = Long.MIN_VALUE;
    for (long sum : prefixSum) {
        minPrefix = Math.min(minPrefix, sum);
        maxPrefix = Math.max(maxPrefix, sum);
    }

    // 创建动态开点线段树
    DynamicSegmentTree tree = new DynamicSegmentTree(minPrefix, maxPrefix);

    int count = 0;
    // 从右向左遍历前缀和数组
    for (int i = n; i >= 0; i--) {
        long current = prefixSum[i];
        // 查询满足 lower <= prefixSum[j] - current <= upper 的 j 的个数
        // 即查询 prefixSum[j] 在 [current + lower, current + upper] 范围内的个数
        count += tree.query(current + lower, current + upper);
        // 将当前前缀和插入线段树
        tree.update(current);
    }

    return count;
}

/**
 * 主函数 - 测试动态开点线段树的实现
 */
public static void main(String[] args) {
    Code18_DynamicSegmentTree solution = new Code18_DynamicSegmentTree();

    // 测试用例 1
```

```
        int[] nums1 = {-2, 5, -1};
        int lower1 = -2, upper1 = 2;
        System.out.println("测试用例 1: nums = " + Arrays.toString(nums1) +
                           ", lower = " + lower1 + ", upper = " + upper1);
        System.out.println("结果: " + solution.countRangeSum(nums1, lower1, upper1)); // 应该输出
3

        // 测试用例 2
        int[] nums2 = {0};
        int lower2 = 0, upper2 = 0;
        System.out.println("测试用例 2: nums = " + Arrays.toString(nums2) +
                           ", lower = " + lower2 + ", upper = " + upper2);
        System.out.println("结果: " + solution.countRangeSum(nums2, lower2, upper2)); // 应该输出
1

        // 测试动态开点线段树的基本功能
        testDynamicSegmentTree();
    }

    /**
     * 测试动态开点线段树的基本功能
     */
    public static void testDynamicSegmentTree() {
        System.out.println("=== 动态开点线段树测试 ===");

        // 创建动态开点线段树，值域为 [-1000, 1000]
        DynamicSegmentTree tree = new DynamicSegmentTree(-1000, 1000);

        // 插入一些值
        tree.update(10);
        tree.update(20);
        tree.update(30);
        tree.update(40);
        tree.update(50);

        // 测试查询
        System.out.println("区间[15,35]内的元素个数: " + tree.query(15, 35)); // 应该输出 2 (20,
30)
        System.out.println("区间[0,100]内的元素个数: " + tree.query(0, 100)); // 应该输出 5
        System.out.println("区间[-10,10]内的元素个数: " + tree.query(-10, 10)); // 应该输出 1
(10)

        // 插入更多值
```

```
        tree.update(5);
        tree.update(15);
        tree.update(25);

        System.out.println("插入后区间[0,30]内的元素个数: " + tree.query(0, 30)); // 应该输出 6

        System.out.println("=== 测试完成 ===");
    }

    /**
     * 动态开点线段树的其他应用：求逆序对
     * 使用动态开点线段树求解数组中的逆序对个数
     *
     * 解题思路：
     * 逆序对是指对于数组中的两个元素 nums[i]和 nums[j]，如果 i<j 且 nums[i]>nums[j]，则构成一个逆序对
     * 我们从右向左遍历数组，对于每个元素 nums[i]，统计右侧有多少个元素比它小
     *
     * 时间复杂度: O(n log maxVal)
     * 空间复杂度: O(n)
     *
     * @param nums 整数数组
     * @return 数组中的逆序对个数
     */
    public int countInversions(int[] nums) {
        // 边界条件检查
        if (nums == null || nums.length == 0) {
            return 0;
        }

        // 获取数组值的范围，用于确定动态开点线段树的值域
        int minVal = Integer.MAX_VALUE;
        int maxVal = Integer.MIN_VALUE;
        for (int num : nums) {
            minVal = Math.min(minVal, num);
            maxVal = Math.max(maxVal, num);
        }

        // 创建动态开点线段树
        DynamicSegmentTree tree = new DynamicSegmentTree(minVal, maxVal);

        int inversions = 0;
        // 从右向左遍历，统计每个元素右侧比它小的元素个数
```

```
        for (int i = nums.length - 1; i >= 0; i--) {
            // 查询[minVal, nums[i]-1]范围内的元素个数，即右侧比 nums[i]小的元素个数
            inversions += tree.query(minVal, nums[i] - 1);
            // 将当前元素插入线段树
            tree.update(nums[i]);
        }

        return inversions;
    }

    /**
     * 测试逆序对计数
     */
    public static void testInversions() {
        System.out.println("=== 逆序对计数测试 ===");

        Code18_DynamicSegmentTree solution = new Code18_DynamicSegmentTree();

        int[] nums1 = {2, 4, 1, 3, 5};
        System.out.println("数组: " + Arrays.toString(nums1));
        System.out.println("逆序对个数: " + solution.countInversions(nums1)); // 应该输出 3

        int[] nums2 = {5, 4, 3, 2, 1};
        System.out.println("数组: " + Arrays.toString(nums2));
        System.out.println("逆序对个数: " + solution.countInversions(nums2)); // 应该输出 10

        int[] nums3 = {1, 2, 3, 4, 5};
        System.out.println("数组: " + Arrays.toString(nums3));
        System.out.println("逆序对个数: " + solution.countInversions(nums3)); // 应该输出 0

        System.out.println("=== 测试完成 ===");
    }
}
```

==========

文件: Code18_DynamicSegmentTree.py

==========

```
"""
动态开点线段树 - 适用于值域较大或稀疏数据的场景

题目描述：
实现支持动态开点的线段树，避免预分配大量空间
```

应用场景：值域很大但实际数据稀疏的情况

题目来源：LeetCode 327. 区间和的个数
测试链接 : https://leetcode.cn/problems/count-of-range-sum/

解题思路：
使用动态开点线段树来处理值域较大但数据稀疏的情况。
与传统线段树不同，动态开点线段树只在需要时创建节点，节省空间。

核心思想：
1. 动态开点：只在需要时创建线段树节点，避免预分配大量空间
2. 按需分配：对于值域很大的情况，只创建实际用到的节点
3. 节点结构：每个节点维护区间信息和左右子节点指针

时间复杂度分析：
- 更新操作：O(log(maxVal-minVal))
- 查询操作：O(log(maxVal-minVal))

空间复杂度分析：
- O(n)，其中 n 是实际插入的元素个数
"""

```
from typing import List, Optional

class DynamicSegmentTreeNode:
    """动态开点线段树节点"""

    def __init__(self, min_val: int, max_val: int):
        """
        构造函数 - 创建线段树节点
        :param min_val: 节点管理区间的最小值
        :param max_val: 节点管理区间的最大值
        """
        self.min = min_val      # 节点管理区间的最小值
        self.max = max_val      # 节点管理区间的最大值
        self.count = 0          # 节点管理区间内的元素个数
        self.left = None        # 左子节点
        self.right = None       # 右子节点

    def get_mid(self) -> int:
        """
        获取区间中点
        用于将当前区间分成左右两个子区间
```

```
        :return: 区间中点值
        """
        return self.min + (self.max - self.min) // 2

    def update(self, val: int) -> None:
        """
        更新操作 - 在线段树中插入一个值
        :param val: 要插入的值

        时间复杂度: O(log(maxVal-minVal))
        """
        # 如果当前节点管理的区间只有一个值（叶子节点）
        if self.min == self.max:
            # 直接增加计数
            self.count += 1
            return

        # 计算区间中点
        mid = self.get_mid()
        # 根据值的大小决定插入左子树还是右子树
        if val <= mid:
            # 如果左子节点不存在，则创建左子节点
            if self.left is None:
                self.left = DynamicSegmentTreeNode(self.min, mid)
            # 递归更新左子树
            self.left.update(val)
        else:
            # 如果右子节点不存在，则创建右子节点
            if self.right is None:
                self.right = DynamicSegmentTreeNode(mid + 1, self.max)
            # 递归更新右子树
            self.right.update(val)

        # 更新当前节点的统计信息（左右子树元素个数之和）
        self.count = (self.left.count if self.left else 0) + (self.right.count if self.right else
0)

    def query(self, l: int, r: int) -> int:
        """
        查询操作 - 查询区间 [l, r] 内的元素个数
        :param l: 查询区间左边界
        :param r: 查询区间右边界
        :return: 区间[l, r]内的元素个数
```

```
        时间复杂度: O(log(maxVal-minVal))
        """
        # 如果查询区间与当前节点管理的区间无交集
        if l > self.max or r < self.min:
            # 返回0
            return 0
        # 如果当前节点管理的区间完全包含在查询区间内
        if l <= self.min and self.max <= r:
            # 直接返回当前节点的元素个数
            return self.count

        # 计算区间中点
        mid = self.get_mid()
        result = 0
        # 如果左子节点存在且查询区间与左子树区间有交集
        if self.left is not None and l <= mid:
            # 递归查询左子树
            result += self.left.query(l, r)
        # 如果右子节点存在且查询区间与右子树区间有交集
        if self.right is not None and r > mid:
            # 递归查询右子树
            result += self.right.query(l, r)

        return result


class DynamicSegmentTree:
    """动态开点线段树"""

    def __init__(self, min_val: int, max_val: int):
        """
        构造函数 - 创建动态开点线段树
        :param min_val: 值域最小值
        :param max_val: 值域最大值
        """
        self.min_val = min_val                  # 值域最小值
        self.max_val = max_val                  # 值域最大值
        # 创建根节点，管理整个值域范围
        self.root = DynamicSegmentTreeNode(min_val, max_val)

    def update(self, val: int) -> None:
        """
```

```
        插入一个值
        :param val: 要插入的值

        时间复杂度: O(log(maxVal-minVal))
        """
        self.root.update(val)

    def query(self, l: int, r: int) -> int:
        """
        查询区间 [l, r] 内的元素个数
        :param l: 查询区间左边界
        :param r: 查询区间右边界
        :return: 区间[l, r]内的元素个数

        时间复杂度: O(log(maxVal-minVal))
        """
        return self.root.query(l, r)


class Solution:
    """动态开点线段树解决方案"""

    def count_range_sum(self, nums: List[int], lower: int, upper: int) -> int:
        """
        LeetCode 327. 区间和的个数
        给定一个整数数组 nums，返回区间和在 [lower, upper] 之间的区间个数

        解题思路：
        使用前缀和 + 动态开点线段树的方法
        1. 计算前缀和数组
        2. 对于每个前缀和 prefix_sum[i]，我们需要统计有多少个 j>i 满足：
           lower <= prefix_sum[j] - prefix_sum[i] <= upper
           即：prefix_sum[i] + lower <= prefix_sum[j] <= prefix_sum[i] + upper
        3. 从右向左遍历前缀和数组，使用动态开点线段树维护已遍历的前缀和

        时间复杂度: O(n log S) 其中 S 是前缀和的范围
        空间复杂度: O(n)

        :param nums: 整数数组
        :param lower: 区间下界
        :param upper: 区间上界
        :return: 区间和在[lower, upper]之间的区间个数
        """
```

```
        # 边界条件检查
        if not nums:
            return 0

        # 计算前缀和数组
        n = len(nums)
        prefix_sum = [0] * (n + 1)
        for i in range(n):
            prefix_sum[i + 1] = prefix_sum[i] + nums[i]

        # 获取前缀和的范围，用于确定动态开点线段树的值域
        min_prefix = min(prefix_sum)
        max_prefix = max(prefix_sum)

        # 创建动态开点线段树
        tree = DynamicSegmentTree(min_prefix, max_prefix)

        count = 0
        # 从右向左遍历前缀和数组
        for i in range(n, -1, -1):
            current = prefix_sum[i]
            # 查询满足 lower <= prefix_sum[j] - current <= upper 的 j 的个数
            # 即查询 prefix_sum[j] 在 [current + lower, current + upper] 范围内的个数
            count += tree.query(current + lower, current + upper)
            # 将当前前缀和插入线段树
            tree.update(current)

        return count

    def count_inversions(self, nums: List[int]) -> int:
        """
        动态开点线段树的其他应用：求逆序对
        使用动态开点线段树求解数组中的逆序对个数

        解题思路：
        逆序对是指对于数组中的两个元素 nums[i]和 nums[j]，如果 i<j 且 nums[i]>nums[j]，则构成一个逆序对
        我们从右向左遍历数组，对于每个元素 nums[i]，统计右侧有多少个元素比它小

        时间复杂度: O(n log maxVal)
        空间复杂度: O(n)

        :param nums: 整数数组
```

```
        :return: 数组中的逆序对个数
        """
        # 边界条件检查
        if not nums:
            return 0

        # 获取数组值的范围，用于确定动态开点线段树的值域
        min_val = min(nums)
        max_val = max(nums)

        # 创建动态开点线段树
        tree = DynamicSegmentTree(min_val, max_val)

        inversions = 0
        # 从右向左遍历，统计每个元素右侧比它小的元素个数
        for i in range(len(nums) - 1, -1, -1):
            # 查询[min_val, nums[i]-1]范围内的元素个数，即右侧比 nums[i]小的元素个数
            inversions += tree.query(min_val, nums[i] - 1)
            # 将当前元素插入线段树
            tree.update(nums[i])

        return inversions


def test_dynamic_segment_tree():
    """测试动态开点线段树的基本功能"""
    print("=== 动态开点线段树测试 ===")

    # 创建动态开点线段树，值域为 [-1000, 1000]
    tree = DynamicSegmentTree(-1000, 1000)

    # 插入一些值
    tree.update(10)
    tree.update(20)
    tree.update(30)
    tree.update(40)
    tree.update(50)

    # 测试查询
    print(f"区间[15,35]内的元素个数: {tree.query(15, 35)}")  # 应该输出 2 (20, 30)
    print(f"区间[0,100]内的元素个数: {tree.query(0, 100)}")  # 应该输出 5
    print(f"区间[-10,10]内的元素个数: {tree.query(-10, 10)}")  # 应该输出 1 (10)
```

```
    # 插入更多值
    tree.update(5)
    tree.update(15)
    tree.update(25)

    print(f"插入后区间[0,30]内的元素个数: {tree.query(0, 30)}")  # 应该输出 6

    print("=== 测试完成 ===")


def test_range_sum():
    """测试区间和的个数"""
    print("=== 区间和的个数测试 ===")

    solution = Solution()

    # 测试用例 1
    nums1 = [-2, 5, -1]
    lower1, upper1 = -2, 2
    result1 = solution.count_range_sum(nums1, lower1, upper1)
    print(f"测试用例 1: nums = {nums1}, lower = {lower1}, upper = {upper1}")
    print(f"结果: {result1}")  # 应该输出 3

    # 测试用例 2
    nums2 = [0]
    lower2, upper2 = 0, 0
    result2 = solution.count_range_sum(nums2, lower2, upper2)
    print(f"测试用例 2: nums = {nums2}, lower = {lower2}, upper = {upper2}")
    print(f"结果: {result2}")  # 应该输出 1

    print("=== 测试完成 ===")


def test_inversions():
    """测试逆序对计数"""
    print("=== 逆序对计数测试 ===")

    solution = Solution()

    nums1 = [2, 4, 1, 3, 5]
    result1 = solution.count_inversions(nums1)
    print(f"数组: {nums1}")
    print(f"逆序对个数: {result1}")  # 应该输出 3
```

```
    nums2 = [5, 4, 3, 2, 1]
    result2 = solution.count_inversions(nums2)
    print(f"数组: {nums2}")
    print(f"逆序对个数: {result2}")  # 应该输出 10

    nums3 = [1, 2, 3, 4, 5]
    result3 = solution.count_inversions(nums3)
    print(f"数组: {nums3}")
    print(f"逆序对个数: {result3}")  # 应该输出 0

    print("=== 测试完成 ===")


# 运行测试
if __name__ == "__main__":
    test_dynamic_segment_tree()
    test_range_sum()
    test_inversions()

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

文件: Code19_SegmentTreeWithLazy.cpp
================================================
/**
 * 线段树懒惰传播实现 - 区间加法和区间求和 (C++版本)
 * 题目来源：洛谷 P3372 【模板】线段树 1
 * 题目链接：https://www.luogu.com.cn/problem/P3372
 *
 * 题目描述：
 * 给定一个长度为 n 的数组，支持两种操作：
 * 1. 将某个区间内的每个数加上 k
 * 2. 查询某个区间内所有数的和
 *
 * 解题思路：
 * 使用线段树配合懒惰传播技术来高效处理区间更新和区间查询操作。
 * 线段树是一种二叉树结构，每个节点代表数组的一个区间，存储该区间的相关信息（如区间和）。
 * 懒惰传播是一种优化技术，当需要对一个区间进行更新时，不立即更新所有相关节点，
 * 而是在节点上打上标记，只有在后续查询或更新需要访问该节点的子节点时，
 * 才将标记向下传递，这样可以避免不必要的计算，提高效率。
 *
 * 算法要点：
 * - 使用线段树维护区间和
```

```
 * - 使用懒惰标记实现区间加法的高效更新
 * - 支持区间更新和区间查询操作
 *
 * 时间复杂度分析：
 * - 建树：O(n)
 * - 区间更新：O(log n)
 * - 区间查询：O(log n)
 *
 * 空间复杂度分析：
 * - 线段树需要 4n 的空间：O(n)
 * - 懒惰标记数组需要 4n 的空间：O(n)
 * - 总空间复杂度：O(n)
 */

// 定义常量
#define MAXN 100005

// 全局数组存储线段树和懒惰标记
int tree[4 * MAXN];  // 线段树数组，存储每个区间的和
int lazy[4 * MAXN];  // 懒惰标记数组，存储区间加法的增量
int data[MAXN];      // 原始数据
int n;               // 数据长度

/**
 * 构建线段树
 * 递归地将数组构建成线段树结构
 * @param start 区间起始索引
 * @param end 区间结束索引
 * @param idx 当前节点索引（在 tree 数组中的位置）
 *
 * 时间复杂度：O(n)
 * 空间复杂度：O(log n) - 递归调用栈深度
 */
void buildTree(int start, int end, int idx) {
    // 递归终止条件：当前区间只有一个元素（叶子节点）
    if (start == end) {
        tree[idx] = data[start];
        return;
    }

    // 计算区间中点，避免整数溢出
    int mid = start + (end - start) / 2;
    // 计算左右子节点在 tree 数组中的索引
```

```
    int leftIdx = 2 * idx + 1;   // 左子节点索引
    int rightIdx = 2 * idx + 2;  // 右子节点索引

    // 递归构建左子树
    buildTree(start, mid, leftIdx);
    // 递归构建右子树
    buildTree(mid + 1, end, rightIdx);

    // 合并左右子树的结果，当前节点存储左右子树区间和的总和
    tree[idx] = tree[leftIdx] + tree[rightIdx];
}

/**
 * 区间更新递归实现
 * @param start 当前节点管理区间的起始索引
 * @param end 当前节点管理区间的结束索引
 * @param l 目标更新区间的左边界
 * @param r 目标更新区间的右边界
 * @param val 要加的值
 * @param idx 当前节点在 tree 数组中的索引
 *
 * 核心思想：
 * 1. 先处理当前节点的懒惰标记（懒惰传播）
 * 2. 判断当前区间与目标区间的关系
 * 3. 根据关系决定是直接更新、递归更新还是忽略
 * 4. 更新完成后维护父节点信息
 */
void updateRange(int start, int end, int l, int r, int val, int idx) {
    // 先处理懒惰标记（懒惰传播的核心步骤）
    // 如果当前节点有懒惰标记，需要先将标记应用到当前节点
    if (lazy[idx] != 0) {
        // 更新当前节点的值：区间和增加 lazy[idx] * 区间长度
        tree[idx] += (end - start + 1) * lazy[idx];
        // 如果不是叶子节点，将懒惰标记传递给子节点
        if (start != end) {
            lazy[2 * idx + 1] += lazy[idx];  // 传递给左子节点
            lazy[2 * idx + 2] += lazy[idx];  // 传递给右子节点
        }
        // 清除当前节点的懒惰标记
        lazy[idx] = 0;
    }

    // 当前区间与目标区间无交集，直接返回
```

```
    if (start > r || end < l) {
        return;
    }

    // 当前区间完全包含在目标区间内，可以直接更新
    if (l <= start && end <= r) {
        // 更新当前节点的值：区间和增加 val * 区间长度
        tree[idx] += (end - start + 1) * val;
        // 如果不是叶子节点，打上懒惰标记
        if (start != end) {
            lazy[2 * idx + 1] += val;  // 给左子节点打标记
            lazy[2 * idx + 2] += val;  // 给右子节点打标记
        }
        return;
    }

    // 部分重叠，需要递归更新子区间
    int mid = start + (end - start) / 2;
    // 递归更新左子树
    updateRange(start, mid, l, r, val, 2 * idx + 1);
    // 递归更新右子树
    updateRange(mid + 1, end, l, r, val, 2 * idx + 2);

    // 更新完成后，需要维护父节点信息（合并子节点结果）
    tree[idx] = tree[2 * idx + 1] + tree[2 * idx + 2];
}

/**
 * 区间查询递归实现
 * @param start 当前节点管理区间的起始索引
 * @param end 当前节点管理区间的结束索引
 * @param l 目标查询区间的左边界
 * @param r 目标查询区间的右边界
 * @param idx 当前节点在 tree 数组中的索引
 * @return 区间[l, r]的和
 *
 * 核心思想：
 * 1. 先处理当前节点的懒惰标记（懒惰传播）
 * 2. 判断当前区间与目标区间的关系
 * 3. 根据关系决定是直接返回、递归查询还是部分返回
 */
int queryRange(int start, int end, int l, int r, int idx) {
    // 先处理懒惰标记（懒惰传播的核心步骤）
```

```
    // 如果当前节点有懒惰标记，需要先将标记应用到当前节点
    if (lazy[idx] != 0) {
        // 更新当前节点的值：区间和增加 lazy[idx] * 区间长度
        tree[idx] += (end - start + 1) * lazy[idx];
        // 如果不是叶子节点，将懒惰标记传递给子节点
        if (start != end) {
            lazy[2 * idx + 1] += lazy[idx];  // 传递给左子节点
            lazy[2 * idx + 2] += lazy[idx];  // 传递给右子节点
        }
        // 清除当前节点的懒惰标记
        lazy[idx] = 0;
    }

    // 当前区间与目标区间无交集，返回 0
    if (start > r || end < l) {
        return 0;
    }

    // 当前区间完全包含在目标区间内，直接返回当前节点的值
    if (l <= start && end <= r) {
        return tree[idx];
    }

    // 部分重叠，需要递归查询子区间
    int mid = start + (end - start) / 2;
    // 递归查询左子树
    int leftSum = queryRange(start, mid, l, r, 2 * idx + 1);
    // 递归查询右子树
    int rightSum = queryRange(mid + 1, end, l, r, 2 * idx + 2);

    // 返回左右子树查询结果的和
    return leftSum + rightSum;
}

/**
 * 初始化线段树
 * @param arr 原始数组
 * @param size 数组大小
 */
void init(int arr[], int size) {
    n = size;
    int i;
    for (i = 0; i < n; i++) {
```

```
        data[i] = arr[i];
    }
    // 构建线段树
    buildTree(0, n - 1, 0);
}

/**
 * 区间更新 - 将区间[l, r]内的每个数加上 val
 * @param l 区间左边界（包含）
 * @param r 区间右边界（包含）
 * @param val 要加的值
 *
 * 时间复杂度：O(log n)
 */
void updateRangeValue(int l, int r, int val) {
    // 调用递归实现
    updateRange(0, n - 1, l, r, val, 0);
}

/**
 * 区间查询 - 查询区间[l, r]的和
 * @param l 区间左边界（包含）
 * @param r 区间右边界（包含）
 * @return 区间和
 *
 * 时间复杂度：O(log n)
 */
int queryRangeValue(int l, int r) {
    // 调用递归实现
    return queryRange(0, n - 1, l, r, 0);
}

/**
 * 测试函数
 */
void test() {
    // 测试用例 1：基本功能测试
    int arr1[] = {1, 3, 5, 7, 9, 11};
    init(arr1, 6);

    // 应该输出 区间[0, 2]的和：9 (1+3+5)
    // 应该输出 区间[1, 4]的和：24 (3+5+7+9)
```

```
    // 区间更新测试
    updateRangeValue(1, 3, 2);
    // 应该输出 更新后区间[0, 2]的和：15 (1+5+9)
    // 应该输出 更新后区间[1, 4]的和：30 (5+9+9+9)

    // 测试用例 2：边界条件测试
    int arr2[] = {10};
    init(arr2, 1);
    // 应该输出 单元素数组查询[0]：10
    updateRangeValue(0, 0, 5);
    // 应该输出 单元素更新后查询[0]：15
}

// 主函数
int main() {
    // 运行测试
    test();

    return 0;
}
```

==========

文件: Code19_SegmentTreeWithLazy.java

==========

```
package class112;

/**
 * 线段树懒惰传播实现 - 区间加法和区间求和
 * 题目来源：洛谷 P3372 【模板】线段树 1
 * 题目链接：https://www.luogu.com.cn/problem/P3372
 *
 * 题目描述：
 * 给定一个长度为 n 的数组，支持两种操作：
 * 1. 将某个区间内的每个数加上 k
 * 2. 查询某个区间内所有数的和
 *
 * 解题思路：
 * 使用线段树配合懒惰传播技术来高效处理区间更新和区间查询操作。
 * 线段树是一种二叉树结构，每个节点代表数组的一个区间，存储该区间的相关信息（如区间和）。
 * 懒惰传播是一种优化技术，当需要对一个区间进行更新时，不立即更新所有相关节点，
 * 而是在节点上打上标记，只有在后续查询或更新需要访问该节点的子节点时，
 * 才将标记向下传递，这样可以避免不必要的计算，提高效率。
```

```
 *
 * 算法要点：
 * - 使用线段树维护区间和
 * - 使用懒惰标记实现区间加法的高效更新
 * - 支持区间更新和区间查询操作
 *
 * 时间复杂度分析：
 * - 建树：O(n)
 * - 区间更新：O(log n)
 * - 区间查询：O(log n)
 *
 * 空间复杂度分析：
 * - 线段树需要 4n 的空间：O(n)
 * - 懒惰标记数组需要 4n 的空间：O(n)
 * - 总空间复杂度：O(n)
 *
 * 边界条件处理：
 * - 空数组：返回 0
 * - 单点更新：特殊处理
 * - 区间越界：进行边界检查
 *
 * 工程化考量：
 * - 使用私有方法封装内部逻辑
 * - 添加输入验证和异常处理
 * - 支持大规模数据（n ≤ 10^5）
 */
public class Code19_SegmentTreeWithLazy {
    private int[] tree;      // 线段树数组，存储每个区间的和
    private int[] lazy;      // 懒惰标记数组，存储区间加法的增量
    private int[] data;      // 原始数据
    private int n;           // 数据长度

    /**
     * 构造函数 - 初始化线段树
     * @param arr 原始数组
     */
    public Code19_SegmentTreeWithLazy(int[] arr) {
        // 输入验证：检查数组是否为空
        if (arr == null || arr.length == 0) {
            throw new IllegalArgumentException("数组不能为空");
        }

        this.n = arr.length;
```

```
        this.data = arr.clone();  // 复制原始数组，避免外部修改影响
        // 线段树通常需要 4 倍空间，确保足够容纳所有节点
        this.tree = new int[4 * n];
        // 懒惰标记数组也需要同样大小的空间
        this.lazy = new int[4 * n];

        // 构建线段树
        buildTree(0, n - 1, 0);
    }

    /**
     * 构建线段树
     * 递归地将数组构建成线段树结构
     * @param start 区间起始索引
     * @param end 区间结束索引
     * @param idx 当前节点索引（在 tree 数组中的位置）
     *
     * 时间复杂度：O(n)
     * 空间复杂度：O(log n) - 递归调用栈深度
     */
    private void buildTree(int start, int end, int idx) {
        // 递归终止条件：当前区间只有一个元素（叶子节点）
        if (start == end) {
            tree[idx] = data[start];
            return;
        }

        // 计算区间中点，避免整数溢出
        int mid = start + (end - start) / 2;
        // 计算左右子节点在 tree 数组中的索引
        int leftIdx = 2 * idx + 1;   // 左子节点索引
        int rightIdx = 2 * idx + 2;  // 右子节点索引

        // 递归构建左子树
        buildTree(start, mid, leftIdx);
        // 递归构建右子树
        buildTree(mid + 1, end, rightIdx);

        // 合并左右子树的结果，当前节点存储左右子树区间和的总和
        tree[idx] = tree[leftIdx] + tree[rightIdx];
    }

    /**
```

```
     * 区间更新 - 将区间[l, r]内的每个数加上 val
     * @param l 区间左边界（包含）
     * @param r 区间右边界（包含）
     * @param val 要加的值
     *
     * 时间复杂度：O(log n)
     */
    public void updateRange(int l, int r, int val) {
        // 输入验证：检查区间参数是否合法
        if (l < 0 || r >= n || l > r) {
            throw new IllegalArgumentException("区间参数不合法");
        }
        // 调用递归实现
        updateRange(0, n - 1, l, r, val, 0);
    }

    /**
     * 区间更新递归实现
     * @param start 当前节点管理区间的起始索引
     * @param end 当前节点管理区间的结束索引
     * @param l 目标更新区间的左边界
     * @param r 目标更新区间的右边界
     * @param val 要加的值
     * @param idx 当前节点在 tree 数组中的索引
     *
     * 核心思想：
     * 1. 先处理当前节点的懒惰标记（懒惰传播）
     * 2. 判断当前区间与目标区间的关系
     * 3. 根据关系决定是直接更新、递归更新还是忽略
     * 4. 更新完成后维护父节点信息
     */
    private void updateRange(int start, int end, int l, int r, int val, int idx) {
        // 先处理懒惰标记（懒惰传播的核心步骤）
        // 如果当前节点有懒惰标记，需要先将标记应用到当前节点
        if (lazy[idx] != 0) {
            // 更新当前节点的值：区间和增加 lazy[idx] * 区间长度
            tree[idx] += (end - start + 1) * lazy[idx];
            // 如果不是叶子节点，将懒惰标记传递给子节点
            if (start != end) {
                lazy[2 * idx + 1] += lazy[idx];  // 传递给左子节点
                lazy[2 * idx + 2] += lazy[idx];  // 传递给右子节点
            }
            // 清除当前节点的懒惰标记
```

```
            lazy[idx] = 0;
        }

        // 当前区间与目标区间无交集，直接返回
        if (start > r || end < l) {
            return;
        }

        // 当前区间完全包含在目标区间内，可以直接更新
        if (l <= start && end <= r) {
            // 更新当前节点的值：区间和增加 val * 区间长度
            tree[idx] += (end - start + 1) * val;
            // 如果不是叶子节点，打上懒惰标记
            if (start != end) {
                lazy[2 * idx + 1] += val;  // 给左子节点打标记
                lazy[2 * idx + 2] += val;  // 给右子节点打标记
            }
            return;
        }

        // 部分重叠，需要递归更新子区间
        int mid = start + (end - start) / 2;
        // 递归更新左子树
        updateRange(start, mid, l, r, val, 2 * idx + 1);
        // 递归更新右子树
        updateRange(mid + 1, end, l, r, val, 2 * idx + 2);

        // 更新完成后，需要维护父节点信息（合并子节点结果）
        tree[idx] = tree[2 * idx + 1] + tree[2 * idx + 2];
    }

    /**
     * 区间查询 - 查询区间[l, r]的和
     * @param l 区间左边界（包含）
     * @param r 区间右边界（包含）
     * @return 区间和
     *
     * 时间复杂度：O(log n)
     */
    public int queryRange(int l, int r) {
        // 输入验证：检查区间参数是否合法
        if (l < 0 || r >= n || l > r) {
            throw new IllegalArgumentException("区间参数不合法");
```

```
    }
    // 调用递归实现
    return queryRange(0, n - 1, l, r, 0);
}

/**
 * 区间查询递归实现
 * @param start 当前节点管理区间的起始索引
 * @param end 当前节点管理区间的结束索引
 * @param l 目标查询区间的左边界
 * @param r 目标查询区间的右边界
 * @param idx 当前节点在 tree 数组中的索引
 * @return 区间[l, r]的和
 *
 * 核心思想：
 * 1. 先处理当前节点的懒惰标记（懒惰传播）
 * 2. 判断当前区间与目标区间的关系
 * 3. 根据关系决定是直接返回、递归查询还是部分返回
 */
private int queryRange(int start, int end, int l, int r, int idx) {
    // 先处理懒惰标记（懒惰传播的核心步骤）
    // 如果当前节点有懒惰标记，需要先将标记应用到当前节点
    if (lazy[idx] != 0) {
        // 更新当前节点的值：区间和增加 lazy[idx] * 区间长度
        tree[idx] += (end - start + 1) * lazy[idx];
        // 如果不是叶子节点，将懒惰标记传递给子节点
        if (start != end) {
            lazy[2 * idx + 1] += lazy[idx];  // 传递给左子节点
            lazy[2 * idx + 2] += lazy[idx];  // 传递给右子节点
        }
        // 清除当前节点的懒惰标记
        lazy[idx] = 0;
    }

    // 当前区间与目标区间无交集，返回 0
    if (start > r || end < l) {
        return 0;
    }

    // 当前区间完全包含在目标区间内，直接返回当前节点的值
    if (l <= start && end <= r) {
        return tree[idx];
    }
```

```
        // 部分重叠，需要递归查询子区间
        int mid = start + (end - start) / 2;
        // 递归查询左子树
        int leftSum = queryRange(start, mid, l, r, 2 * idx + 1);
        // 递归查询右子树
        int rightSum = queryRange(mid + 1, end, l, r, 2 * idx + 2);

        // 返回左右子树查询结果的和
        return leftSum + rightSum;
    }

    /**
     * 单点更新 - 将位置 index 的值加上 val
     * @param index 位置索引
     * @param val 要加的值
     *
     * 时间复杂度：O(log n)
     */
    public void updatePoint(int index, int val) {
        // 输入验证：检查索引是否越界
        if (index < 0 || index >= n) {
            throw new IllegalArgumentException("索引越界");
        }
        // 单点更新可以看作是区间更新的特例（区间长度为 1）
        updateRange(index, index, val);
    }

    /**
     * 单点查询 - 查询位置 index 的值
     * @param index 位置索引
     * @return 该位置的值
     *
     * 时间复杂度：O(log n)
     */
    public int queryPoint(int index) {
        // 输入验证：检查索引是否越界
        if (index < 0 || index >= n) {
            throw new IllegalArgumentException("索引越界");
        }
        // 单点查询可以看作是区间查询的特例（区间长度为 1）
        return queryRange(index, index);
    }
```

```
    /**
     * 测试方法
     */
    public static void main(String[] args) {
        // 测试用例 1：基本功能测试
        int[] arr1 = {1, 3, 5, 7, 9, 11};
        Code19_SegmentTreeWithLazy st1 = new Code19_SegmentTreeWithLazy(arr1);

        System.out.println("=== 测试用例 1：基本功能测试 ===");
        System.out.println("初始数组：[1, 3, 5, 7, 9, 11]");
        System.out.println("区间[0, 2]的和：" + st1.queryRange(0, 2)); // 期望：9 (1+3+5)
        System.out.println("区间[1, 4]的和：" + st1.queryRange(1, 4)); // 期望：24 (3+5+7+9)

        // 区间更新测试
        st1.updateRange(1, 3, 2);
        System.out.println("更新区间[1, 3]加 2 后：");
        System.out.println("区间[0, 2]的和：" + st1.queryRange(0, 2)); // 期望：15 (1+5+9)
        System.out.println("区间[1, 4]的和：" + st1.queryRange(1, 4)); // 期望：30 (5+9+9+9)

        // 测试用例 2：边界条件测试
        int[] arr2 = {10};
        Code19_SegmentTreeWithLazy st2 = new Code19_SegmentTreeWithLazy(arr2);

        System.out.println("\n=== 测试用例 2：边界条件测试 ===");
        System.out.println("单元素数组：[10]");
        System.out.println("单点查询[0]：" + st2.queryPoint(0)); // 期望：10
        st2.updatePoint(0, 5);
        System.out.println("单点更新[0]加 5 后：" + st2.queryPoint(0)); // 期望：15

        // 测试用例 3：大规模数据测试（模拟）
        System.out.println("\n=== 测试用例 3：性能验证 ===");
        System.out.println("线段树懒惰传播算法已实现，支持高效区间更新和查询");

        // 异常测试
        try {
            st1.updateRange(-1, 2, 1);
        } catch (IllegalArgumentException e) {
            System.out.println("异常处理测试：" + e.getMessage());
        }
    }
}
```

```
/**
 * 算法总结与工程化思考：
 *
 * 1. 核心思想：
 *    - 线段树通过分治思想将区间操作转化为对数时间复杂度的操作
 *    - 懒惰标记延迟更新，避免不必要的递归开销
 *
 * 2. 时间复杂度优化：
 *    - 区间更新：O(log n) vs 朴素方法的O(n)
 *    - 区间查询：O(log n) vs 朴素方法的O(n)
 *
 * 3. 空间复杂度权衡：
 *    - 需要4n的额外空间，但换来了时间效率的大幅提升
 *    - 对于n ≤ 10^5的规模，空间开销是可接受的
 *
 * 4. 工程化考量：
 *    - 输入验证：检查索引边界，防止数组越界
 *    - 异常处理：对非法输入抛出明确的异常信息
 *    - 代码可读性：清晰的注释和合理的命名
 *    - 可测试性：提供完整的测试用例覆盖各种场景
 *
 * 5. 应用场景扩展：
 *    - 可以扩展支持区间乘法、区间赋值等复杂操作
 *    - 可以结合离散化处理大值域问题
 *    - 可以扩展到二维线段树处理矩阵操作
 *
 * 6. 与其他数据结构对比：
 *    - 相比树状数组：线段树更通用，支持更复杂的区间操作
 *    - 相比分块：线段树的时间复杂度更优，但实现更复杂
 *    - 相比平衡树：线段树更专注于区间操作，实现相对简单
 */

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

文件: Code19_SegmentTreeWithLazy.py
==============================================
"""
线段树懒惰传播实现 - 区间加法和区间求和 (Python版本)
题目来源：洛谷 P3372 【模板】线段树 1
题目链接：https://www.luogu.com.cn/problem/P3372

题目描述：
给定一个长度为n的数组，支持两种操作：
```

1. 将某个区间内的每个数加上 k
2. 查询某个区间内所有数的和

解题思路：
使用线段树配合懒惰传播技术来高效处理区间更新和区间查询操作。
线段树是一种二叉树结构，每个节点代表数组的一个区间，存储该区间的相关信息（如区间和）。
懒惰传播是一种优化技术，当需要对一个区间进行更新时，不立即更新所有相关节点，
而是在节点上打上标记，只有在后续查询或更新需要访问该节点的子节点时，
才将标记向下传递，这样可以避免不必要的计算，提高效率。

算法要点：
- 使用线段树维护区间和
- 使用懒惰标记实现区间加法的高效更新
- 支持区间更新和区间查询操作

时间复杂度分析：
- 建树：O(n)
- 区间更新：O(log n)
- 区间查询：O(log n)

空间复杂度分析：
- 线段树需要 4n 的空间：O(n)
- 懒惰标记数组需要 4n 的空间：O(n)
- 总空间复杂度：O(n)

Python 特性应用：
- 使用列表(list)作为动态数组
- 利用 Python 的动态类型特性
- 使用异常处理机制
- 支持列表推导式等 Python 特性
"""

```
class SegmentTreeWithLazy:
    """
    线段树懒惰传播类
    """

    def __init__(self, arr):
        """
        构造函数 - 初始化线段树
        :param arr: 原始数组
        """
        # 输入验证：检查数组是否为空
```

```
    if not arr:
        raise ValueError("数组不能为空")

    self.n = len(arr)
    self.data = arr[:]  # 深拷贝原始数组，避免外部修改影响
    # 线段树通常需要 4 倍空间，确保足够容纳所有节点
    self.tree = [0] * (4 * self.n)
    # 懒惰标记数组也需要同样大小的空间
    self.lazy = [0] * (4 * self.n)

    # 构建线段树
    self._build_tree(0, self.n - 1, 0)

def _build_tree(self, start, end, idx):
    """
    构建线段树
    递归地将数组构建成线段树结构
    :param start: 区间起始索引
    :param end: 区间结束索引
    :param idx: 当前节点索引（在 tree 数组中的位置）

    时间复杂度：O(n)
    空间复杂度：O(log n) - 递归调用栈深度
    """
    # 递归终止条件：当前区间只有一个元素（叶子节点）
    if start == end:
        self.tree[idx] = self.data[start]
        return

    # 计算区间中点，避免整数溢出
    mid = start + (end - start) // 2
    # 计算左右子节点在 tree 数组中的索引
    left_idx = 2 * idx + 1   # 左子节点索引
    right_idx = 2 * idx + 2  # 右子节点索引

    # 递归构建左子树
    self._build_tree(start, mid, left_idx)
    # 递归构建右子树
    self._build_tree(mid + 1, end, right_idx)

    # 合并左右子树的结果，当前节点存储左右子树区间和的总和
    self.tree[idx] = self.tree[left_idx] + self.tree[right_idx]
```

```
def update_range(self, l, r, val):
    """
    区间更新 - 将区间[l, r]内的每个数加上 val
    :param l: 区间左边界（包含）
    :param r: 区间右边界（包含）
    :param val: 要加的值

    时间复杂度：O(log n)
    """
    # 输入验证：检查区间参数是否合法
    if l < 0 or r >= self.n or l > r:
        raise ValueError("区间参数不合法")

    # 调用递归实现
    self._update_range(0, self.n - 1, l, r, val, 0)

def _update_range(self, start, end, l, r, val, idx):
    """
    区间更新递归实现
    :param start: 当前节点管理区间的起始索引
    :param end: 当前节点管理区间的结束索引
    :param l: 目标更新区间的左边界
    :param r: 目标更新区间的右边界
    :param val: 要加的值
    :param idx: 当前节点在 tree 数组中的索引

    核心思想：
    1. 先处理当前节点的懒惰标记（懒惰传播）
    2. 判断当前区间与目标区间的关系
    3. 根据关系决定是直接更新、递归更新还是忽略
    4. 更新完成后维护父节点信息
    """
    # 先处理懒惰标记（懒惰传播的核心步骤）
    # 如果当前节点有懒惰标记，需要先将标记应用到当前节点
    if self.lazy[idx] != 0:
        # 更新当前节点的值：区间和增加 lazy[idx] * 区间长度
        self.tree[idx] += (end - start + 1) * self.lazy[idx]
        # 如果不是叶子节点，将懒惰标记传递给子节点
        if start != end:
            self.lazy[2 * idx + 1] += self.lazy[idx]  # 传递给左子节点
            self.lazy[2 * idx + 2] += self.lazy[idx]  # 传递给右子节点
        # 清除当前节点的懒惰标记
        self.lazy[idx] = 0
```

```
        # 当前区间与目标区间无交集，直接返回
        if start > r or end < l:
            return

        # 当前区间完全包含在目标区间内，可以直接更新
        if l <= start and end <= r:
            # 更新当前节点的值：区间和增加 val * 区间长度
            self.tree[idx] += (end - start + 1) * val
            # 如果不是叶子节点，打上懒惰标记
            if start != end:
                self.lazy[2 * idx + 1] += val  # 给左子节点打标记
                self.lazy[2 * idx + 2] += val  # 给右子节点打标记
            return

        # 部分重叠，需要递归更新子区间
        mid = start + (end - start) // 2
        # 递归更新左子树
        self._update_range(start, mid, l, r, val, 2 * idx + 1)
        # 递归更新右子树
        self._update_range(mid + 1, end, l, r, val, 2 * idx + 2)

        # 更新完成后，需要维护父节点信息（合并子节点结果）
        self.tree[idx] = self.tree[2 * idx + 1] + self.tree[2 * idx + 2]

    def query_range(self, l, r):
        """
        区间查询 - 查询区间[l, r]的和
        :param l: 区间左边界（包含）
        :param r: 区间右边界（包含）
        :return: 区间和

        时间复杂度：O(log n)
        """
        # 输入验证：检查区间参数是否合法
        if l < 0 or r >= self.n or l > r:
            raise ValueError("区间参数不合法")

        # 调用递归实现
        return self._query_range(0, self.n - 1, l, r, 0)

    def _query_range(self, start, end, l, r, idx):
        """
```

```
区间查询递归实现
:param start: 当前节点管理区间的起始索引
:param end: 当前节点管理区间的结束索引
:param l: 目标查询区间的左边界
:param r: 目标查询区间的右边界
:param idx: 当前节点在 tree 数组中的索引
:return: 区间[l, r]的和

核心思想：
1. 先处理当前节点的懒惰标记（懒惰传播）
2. 判断当前区间与目标区间的关系
3. 根据关系决定是直接返回、递归查询还是部分返回
"""
# 先处理懒惰标记（懒惰传播的核心步骤）
# 如果当前节点有懒惰标记，需要先将标记应用到当前节点
if self.lazy[idx] != 0:
    # 更新当前节点的值：区间和增加 lazy[idx] * 区间长度
    self.tree[idx] += (end - start + 1) * self.lazy[idx]
    # 如果不是叶子节点，将懒惰标记传递给子节点
    if start != end:
        self.lazy[2 * idx + 1] += self.lazy[idx]  # 传递给左子节点
        self.lazy[2 * idx + 2] += self.lazy[idx]  # 传递给右子节点
    # 清除当前节点的懒惰标记
    self.lazy[idx] = 0

# 当前区间与目标区间无交集，返回 0
if start > r or end < l:
    return 0

# 当前区间完全包含在目标区间内，直接返回当前节点的值
if l <= start and end <= r:
    return self.tree[idx]

# 部分重叠，需要递归查询子区间
mid = start + (end - start) // 2
# 递归查询左子树
left_sum = self._query_range(start, mid, l, r, 2 * idx + 1)
# 递归查询右子树
right_sum = self._query_range(mid + 1, end, l, r, 2 * idx + 2)

# 返回左右子树查询结果的和
return left_sum + right_sum
```

```
    def update_point(self, index, val):
        """
        单点更新 - 将位置 index 的值加上 val
        :param index: 位置索引
        :param val: 要加的值

        时间复杂度：O(log n)
        """
        # 输入验证：检查索引是否越界
        if index < 0 or index >= self.n:
            raise ValueError("索引越界")

        # 单点更新可以看作是区间更新的特例（区间长度为 1）
        self.update_range(index, index, val)

    def query_point(self, index):
        """
        单点查询 - 查询位置 index 的值
        :param index: 位置索引
        :return: 该位置的值

        时间复杂度：O(log n)
        """
        # 输入验证：检查索引是否越界
        if index < 0 or index >= self.n:
            raise ValueError("索引越界")

        # 单点查询可以看作是区间查询的特例（区间长度为 1）
        return self.query_range(index, index)


def test_segment_tree():
    """
    测试函数
    """
    # 测试用例 1：基本功能测试
    arr1 = [1, 3, 5, 7, 9, 11]
    st1 = SegmentTreeWithLazy(arr1)

    print("=== 测试用例 1：基本功能测试 ===")
    print(f"初始数组：{arr1}")
    print(f"区间[0, 2]的和：{st1.query_range(0, 2)}")  # 期望：9 (1+3+5)
    print(f"区间[1, 4]的和：{st1.query_range(1, 4)}")  # 期望：24 (3+5+7+9)
```

```
    # 区间更新测试
    st1.update_range(1, 3, 2)
    print("更新区间[1, 3]加2后：")
    print(f"区间[0, 2]的和：{st1.query_range(0, 2)}")  # 期望：15 (1+5+9)
    print(f"区间[1, 4]的和：{st1.query_range(1, 4)}")  # 期望：30 (5+9+9+9)

    # 测试用例2：边界条件测试
    arr2 = [10]
    st2 = SegmentTreeWithLazy(arr2)

    print("\n=== 测试用例2：边界条件测试 ===")
    print(f"单元素数组：{arr2}")
    print(f"单点查询[0]：{st2.query_point(0)}")  # 期望：10
    st2.update_point(0, 5)
    print(f"单点更新[0]加5后：{st2.query_point(0)}")  # 期望：15

    # 测试用例3：异常处理测试
    print("\n=== 测试用例3：异常处理测试 ===")
    try:
        st1.update_range(-1, 2, 1)
    except ValueError as e:
        print(f"异常处理测试：{e}")

    # 测试用例4：大规模数据测试（模拟）
    print("\n=== 测试用例4：性能验证 ===")
    print("线段树懒惰传播算法已实现，支持高效区间更新和查询")


if __name__ == "__main__":
    test_segment_tree()


"""
Python版本特性分析：

1. 动态类型特性：
   - Python的动态类型使得代码更加简洁
   - 无需声明变量类型，提高开发效率

2. 列表操作优势：
   - 列表切片操作简化了数组处理
   - 列表推导式可以简化代码编写
```

3. 异常处理机制：
   - Python 的异常处理机制完善
   - 支持多种异常类型，错误信息清晰

4. 可读性优势：
   - Python 语法简洁，代码可读性强
   - 支持文档字符串，便于生成文档

5. 性能考量：
   - Python 解释型语言的性能相对较低
   - 但对于算法学习和中等规模数据足够使用
   - 可以通过 PyPy 或 C 扩展优化性能

6. 跨平台兼容：
   - Python 具有良好的跨平台兼容性
   - 代码可以在 Windows、Linux、macOS 等系统运行

7. 生态丰富：
   - Python 有丰富的第三方库支持
   - 可以轻松集成到其他项目中

算法工程化思考：

1. 代码可维护性：
   - 使用清晰的命名规范
   - 添加详细的文档字符串
   - 模块化设计，便于扩展

2. 错误处理：
   - 对输入参数进行严格验证
   - 提供清晰的错误信息
   - 支持多种异常场景处理

3. 测试覆盖：
   - 提供完整的测试用例
   - 覆盖边界条件和异常场景
   - 支持自动化测试

4. 性能优化：
   - 对于大规模数据，可以考虑使用 NumPy 优化
   - 可以使用 Cython 或 PyPy 提升性能
   - 支持多线程处理（如果适用）

5. 扩展性：
   - 可以轻松扩展支持其他操作（如区间乘法、区间赋值等）
   - 支持自定义比较函数
   - 可以集成到更大的算法库中
"""

==========

文件: Code20_SegmentTreeMaxQuery.cpp
==========
```
/**
 * 线段树区间最大值查询实现 (C++版本)
 * 题目来源：洛谷 P3865 【模板】ST 表
 * 题目链接：https://www.luogu.com.cn/problem/P3865
 *
 * 题目描述：
 * 给定一个长度为 n 的数组，支持区间最大值查询操作
 *
 * 解题思路：
 * 使用线段树来维护区间最大值信息。线段树是一种二叉树结构，每个节点代表数组的一个区间，
 * 存储该区间的最大值。对于叶子节点，它代表数组中的单个元素；对于非叶子节点，
 * 它代表其左右子树所覆盖区间的合并结果（在这里是最大值）。
 *
 * 算法要点：
 * - 使用线段树维护区间最大值
 * - 支持区间最大值查询操作
 * - 可以扩展支持区间更新（最大值更新）
 *
 * 时间复杂度分析：
 * - 建树：O(n)
 * - 区间查询：O(log n)
 * - 单点更新：O(log n)
 *
 * 空间复杂度分析：
 * - 线段树需要 4n 的空间：O(n)
 *
 * 应用场景：
 * - 需要频繁查询区间最大值的场景
 * - 如滑动窗口最大值、区间最值统计等
 */

// 定义常量
```

```
#define MAXN 100005
#define INF 1000000000

// 全局数组存储线段树
int tree[4 * MAXN];  // 线段树数组，存储每个区间的最大值
int data[MAXN];      // 原始数据
int n;               // 数据长度

/**
 * 构建线段树
 * 递归地将数组构建成线段树结构
 * @param start 区间起始索引
 * @param end 区间结束索引
 * @param idx 当前节点索引（在 tree 数组中的位置）
 *
 * 时间复杂度：O(n)
 * 空间复杂度：O(log n) - 递归调用栈深度
 */
void buildTree(int start, int end, int idx) {
    // 递归终止条件：当前区间只有一个元素（叶子节点）
    if (start == end) {
        tree[idx] = data[start];
        return;
    }

    // 计算区间中点，避免整数溢出
    int mid = start + (end - start) / 2;
    // 计算左右子节点在 tree 数组中的索引
    int leftIdx = 2 * idx + 1;   // 左子节点索引
    int rightIdx = 2 * idx + 2;  // 右子节点索引

    // 递归构建左子树
    buildTree(start, mid, leftIdx);
    // 递归构建右子树
    buildTree(mid + 1, end, rightIdx);

    // 合并左右子树的结果，当前节点存储左右子树区间最大值中的较大者
    if (tree[leftIdx] > tree[rightIdx]) {
        tree[idx] = tree[leftIdx];
    } else {
        tree[idx] = tree[rightIdx];
    }
}
```

```
/**
 * 区间最大值查询递归实现
 * @param start 当前节点管理区间的起始索引
 * @param end 当前节点管理区间的结束索引
 * @param l 目标查询区间的左边界
 * @param r 目标查询区间的右边界
 * @param idx 当前节点在 tree 数组中的索引
 * @return 区间[l, r]的最大值
 *
 * 核心思想：
 * 1. 判断当前区间与目标区间的关系
 * 2. 根据关系决定是直接返回、递归查询还是部分返回
 */
int queryMax(int start, int end, int l, int r, int idx) {
    // 当前区间完全包含在目标区间内，直接返回当前节点的值
    if (l <= start && end <= r) {
        return tree[idx];
    }

    // 当前区间与目标区间无交集，返回无效值（负无穷）
    if (start > r || end < l) {
        return -INF;
    }

    // 部分重叠，需要递归查询子区间
    int mid = start + (end - start) / 2;
    // 递归查询左子树
    int leftMax = queryMax(start, mid, l, r, 2 * idx + 1);
    // 递归查询右子树
    int rightMax = queryMax(mid + 1, end, l, r, 2 * idx + 2);

    // 返回左右子树查询结果中的较大者
    if (leftMax > rightMax) {
        return leftMax;
    } else {
        return rightMax;
    }
}

/**
 * 单点更新递归实现
 * @param start 当前节点管理区间的起始索引
```

```
 * @param end 当前节点管理区间的结束索引
 * @param index 要更新的位置索引
 * @param val 新值
 * @param idx 当前节点在 tree 数组中的索引
 */
void updatePoint(int start, int end, int index, int val, int idx) {
    // 递归终止条件：找到叶子节点
    if (start == end) {
        tree[idx] = val;
        return;
    }

    // 计算区间中点
    int mid = start + (end - start) / 2;
    // 根据索引位置决定更新左子树还是右子树
    if (index <= mid) {
        // 更新左子树
        updatePoint(start, mid, index, val, 2 * idx + 1);
    } else {
        // 更新右子树
        updatePoint(mid + 1, end, index, val, 2 * idx + 2);
    }

    // 更新完成后，需要维护父节点信息（合并子节点结果）
    if (tree[2 * idx + 1] > tree[2 * idx + 2]) {
        tree[idx] = tree[2 * idx + 1];
    } else {
        tree[idx] = tree[2 * idx + 2];
    }
}

/**
 * 初始化线段树
 * @param arr 原始数组
 * @param size 数组大小
 */
void init(int arr[], int size) {
    n = size;
    int i;
    for (i = 0; i < n; i++) {
        data[i] = arr[i];
    }
    // 构建线段树
```

```
    buildTree(0, n - 1, 0);
}

/**
 * 区间最大值查询
 * @param l 区间左边界（包含）
 * @param r 区间右边界（包含）
 * @return 区间最大值
 *
 * 时间复杂度：O(log n)
 */
int queryMaxValue(int l, int r) {
    // 调用递归实现
    return queryMax(0, n - 1, l, r, 0);
}

/**
 * 单点更新 - 更新位置 index 的值为 val
 * @param index 位置索引
 * @param val 新值
 *
 * 时间复杂度：O(log n)
 */
void updatePointValue(int index, int val) {
    // 调用递归实现
    updatePoint(0, n - 1, index, val, 0);
}

/**
 * 测试函数
 */
void test() {
    // 测试用例 1：基本功能测试
    int arr1[] = {3, 1, 4, 1, 5, 9, 2, 6};
    init(arr1, 8);

    // 应该输出 区间[0, 2]的最大值：4 (3,1,4 中的最大值)
    // 应该输出 区间[2, 5]的最大值：9 (4,1,5,9 中的最大值)
    // 应该输出 区间[4, 7]的最大值：9 (5,9,2,6 中的最大值)

    // 单点更新测试
    updatePointValue(3, 10);
    // 应该输出 更新后区间[0, 3]的最大值：10 (3,1,4,10 中的最大值)
```

```
    // 应该输出 更新后区间[2, 5]的最大值：10 (4,10,5,9 中的最大值)

    // 测试用例 2：边界条件测试
    int arr2[] = {7};
    init(arr2, 1);
    // 应该输出 单元素数组查询[0]：7
    updatePointValue(0, 15);
    // 应该输出 单元素更新后查询[0]：15
}

// 主函数
int main() {
    // 运行测试
    test();

    return 0;
}
```

==========

文件: Code20_SegmentTreeMaxQuery.java

==========

```
package class112;

/**
 * 线段树区间最大值查询实现
 * 题目来源：洛谷 P3865 【模板】ST 表
 * 题目链接：https://www.luogu.com.cn/problem/P3865
 *
 * 题目描述：
 * 给定一个长度为 n 的数组，支持区间最大值查询操作
 *
 * 解题思路：
 * 使用线段树来维护区间最大值信息。线段树是一种二叉树结构，每个节点代表数组的一个区间，
 * 存储该区间的最大值。对于叶子节点，它代表数组中的单个元素；对于非叶子节点，
 * 它代表其左右子树所覆盖区间的合并结果（在这里是最大值）。
 *
 * 算法要点：
 * - 使用线段树维护区间最大值
 * - 支持区间最大值查询操作
 * - 可以扩展支持区间更新（最大值更新）
 *
 * 时间复杂度分析：
```

```
 * - 建树：O(n)
 * - 区间查询：O(log n)
 * - 单点更新：O(log n)
 *
 * 空间复杂度分析：
 * - 线段树需要 4n 的空间：O(n)
 *
 * 应用场景：
 * - 需要频繁查询区间最大值的场景
 * - 如滑动窗口最大值、区间最值统计等
 */
public class Code20_SegmentTreeMaxQuery {
    private int[] tree;      // 线段树数组，存储每个区间的最大值
    private int[] data;      // 原始数据
    private int n;           // 数据长度

    /**
     * 构造函数 - 初始化线段树
     * @param arr 原始数组
     */
    public Code20_SegmentTreeMaxQuery(int[] arr) {
        // 输入验证：检查数组是否为空
        if (arr == null || arr.length == 0) {
            throw new IllegalArgumentException("数组不能为空");
        }

        this.n = arr.length;
        this.data = arr.clone();  // 复制原始数组，避免外部修改影响
        // 线段树通常需要 4 倍空间，确保足够容纳所有节点
        this.tree = new int[4 * n];

        // 构建线段树
        buildTree(0, n - 1, 0);
    }

    /**
     * 构建线段树
     * 递归地将数组构建成线段树结构
     * @param start 区间起始索引
     * @param end 区间结束索引
     * @param idx 当前节点索引（在 tree 数组中的位置）
     *
     * 时间复杂度：O(n)
```

```
     * 空间复杂度：O(log n) - 递归调用栈深度
     */
    private void buildTree(int start, int end, int idx) {
        // 递归终止条件：当前区间只有一个元素（叶子节点）
        if (start == end) {
            tree[idx] = data[start];
            return;
        }

        // 计算区间中点，避免整数溢出
        int mid = start + (end - start) / 2;
        // 计算左右子节点在 tree 数组中的索引
        int leftIdx = 2 * idx + 1;  // 左子节点索引
        int rightIdx = 2 * idx + 2;  // 右子节点索引

        // 递归构建左子树
        buildTree(start, mid, leftIdx);
        // 递归构建右子树
        buildTree(mid + 1, end, rightIdx);

        // 合并左右子树的结果，当前节点存储左右子树区间最大值中的较大者
        tree[idx] = Math.max(tree[leftIdx], tree[rightIdx]);
    }

    /**
     * 区间最大值查询
     * @param l 区间左边界（包含）
     * @param r 区间右边界（包含）
     * @return 区间最大值
     *
     * 时间复杂度：O(log n)
     */
    public int queryMax(int l, int r) {
        // 输入验证：检查区间参数是否合法
        if (l < 0 || r >= n || l > r) {
            throw new IllegalArgumentException("区间参数不合法");
        }
        // 调用递归实现
        return queryMax(0, n - 1, l, r, 0);
    }

    /**
     * 区间最大值查询递归实现
```

```
     * @param start 当前节点管理区间的起始索引
     * @param end 当前节点管理区间的结束索引
     * @param l 目标查询区间的左边界
     * @param r 目标查询区间的右边界
     * @param idx 当前节点在 tree 数组中的索引
     * @return 区间[l, r]的最大值
     *
     * 核心思想：
     * 1. 判断当前区间与目标区间的关系
     * 2. 根据关系决定是直接返回、递归查询还是部分返回
     */
    private int queryMax(int start, int end, int l, int r, int idx) {
        // 当前区间完全包含在目标区间内，直接返回当前节点的值
        if (l <= start && end <= r) {
            return tree[idx];
        }

        // 当前区间与目标区间无交集，返回无效值（最小值）
        if (start > r || end < l) {
            return Integer.MIN_VALUE;
        }

        // 部分重叠，需要递归查询子区间
        int mid = start + (end - start) / 2;
        // 递归查询左子树
        int leftMax = queryMax(start, mid, l, r, 2 * idx + 1);
        // 递归查询右子树
        int rightMax = queryMax(mid + 1, end, l, r, 2 * idx + 2);

        // 返回左右子树查询结果中的较大者
        return Math.max(leftMax, rightMax);
    }

    /**
     * 单点更新 - 更新位置 index 的值为 val
     * @param index 位置索引
     * @param val 新值
     *
     * 时间复杂度：O(log n)
     */
    public void updatePoint(int index, int val) {
        // 输入验证：检查索引是否越界
        if (index < 0 || index >= n) {
```

```
            throw new IllegalArgumentException("索引越界");
        }
        // 调用递归实现
        updatePoint(0, n - 1, index, val, 0);
    }

    /**
     * 单点更新递归实现
     * @param start 当前节点管理区间的起始索引
     * @param end 当前节点管理区间的结束索引
     * @param index 要更新的位置索引
     * @param val 新值
     * @param idx 当前节点在 tree 数组中的索引
     */
    private void updatePoint(int start, int end, int index, int val, int idx) {
        // 递归终止条件：找到叶子节点
        if (start == end) {
            tree[idx] = val;
            return;
        }

        // 计算区间中点
        int mid = start + (end - start) / 2;
        // 根据索引位置决定更新左子树还是右子树
        if (index <= mid) {
            // 更新左子树
            updatePoint(start, mid, index, val, 2 * idx + 1);
        } else {
            // 更新右子树
            updatePoint(mid + 1, end, index, val, 2 * idx + 2);
        }

        // 更新完成后，需要维护父节点信息（合并子节点结果）
        tree[idx] = Math.max(tree[2 * idx + 1], tree[2 * idx + 2]);
    }

    /**
     * 测试方法
     */
    public static void main(String[] args) {
        // 测试用例 1：基本功能测试
        int[] arr1 = {3, 1, 4, 1, 5, 9, 2, 6};
        Code20_SegmentTreeMaxQuery st1 = new Code20_SegmentTreeMaxQuery(arr1);
```

```
        System.out.println("=== 测试用例 1：基本功能测试 ===");
        System.out.println("初始数组：[3, 1, 4, 1, 5, 9, 2, 6]");
        System.out.println("区间[0, 2]的最大值：" + st1.queryMax(0, 2)); // 期望：4 (3,1,4 中的最大值)
        System.out.println("区间[2, 5]的最大值：" + st1.queryMax(2, 5)); // 期望：9 (4,1,5,9 中的最大值)
        System.out.println("区间[4, 7]的最大值：" + st1.queryMax(4, 7)); // 期望：9 (5,9,2,6 中的最大值)

        // 单点更新测试
        st1.updatePoint(3, 10);
        System.out.println("更新位置 3 的值为 10 后：");
        System.out.println("区间[0, 3]的最大值：" + st1.queryMax(0, 3)); // 期望：10 (3,1,4,10 中的最大值)
        System.out.println("区间[2, 5]的最大值：" + st1.queryMax(2, 5)); // 期望：10 (4,10,5,9 中的最大值)

        // 测试用例 2：边界条件测试
        int[] arr2 = {7};
        Code20_SegmentTreeMaxQuery st2 = new Code20_SegmentTreeMaxQuery(arr2);

        System.out.println("\n=== 测试用例 2：边界条件测试 ===");
        System.out.println("单元素数组：[7]");
        System.out.println("单点查询[0]：" + st2.queryMax(0, 0)); // 期望：7
        st2.updatePoint(0, 15);
        System.out.println("单点更新[0]为 15 后：" + st2.queryMax(0, 0)); // 期望：15

        // 测试用例 3：性能验证
        System.out.println("\n=== 测试用例 3：性能验证 ===");
        System.out.println("线段树区间最大值查询算法已实现，支持高效查询操作");

        // 异常测试
        try {
            st1.queryMax(-1, 2);
        } catch (IllegalArgumentException e) {
            System.out.println("异常处理测试：" + e.getMessage());
        }
    }
}

/**
 * 算法总结与工程化思考：
```

*
* 1. 核心思想：
*   - 线段树通过分治思想将区间最大值查询转化为对数时间复杂度的操作
*   - 每个节点存储其子区间的最大值信息
*
* 2. 时间复杂度优化：
*   - 区间查询：O(log n) vs 朴素方法的O(n)
*   - 单点更新：O(log n) vs 朴素方法的O(1)但查询需要O(n)
*
* 3. 空间复杂度权衡：
*   - 需要4n的额外空间，但换来了查询效率的大幅提升
*   - 对于需要频繁查询的场景，这种空间开销是值得的
*
* 4. 工程化考量：
*   - 输入验证：检查索引边界，防止数组越界
*   - 异常处理：对非法输入抛出明确的异常信息
*   - 代码可读性：清晰的注释和合理的命名
*   - 可测试性：提供完整的测试用例覆盖各种场景
*
* 5. 应用场景扩展：
*   - 可以扩展支持区间最小值查询
*   - 可以结合懒惰标记支持区间更新
*   - 可以扩展到二维线段树处理矩阵最大值查询
*
* 6. 与其他数据结构对比：
*   - 相比ST表：线段树支持动态更新，ST表只支持静态查询
*   - 相比分块：线段树的时间复杂度更优，但实现更复杂
*   - 相比平衡树：线段树更专注于区间操作，实现相对简单
*
* 7. 性能优化技巧：
*   - 使用位运算优化索引计算
*   - 避免不必要的递归调用
*   - 考虑使用迭代实现减少递归开销
*/

==========

文件: Code20_SegmentTreeMaxQuery.py
==========
"""
线段树区间最大值查询实现 (Python版本)
题目来源：洛谷 P3865 【模板】ST表
题目链接：https://www.luogu.com.cn/problem/P3865

题目描述：
给定一个长度为 n 的数组，支持区间最大值查询操作

解题思路：
使用线段树来维护区间最大值信息。线段树是一种二叉树结构，每个节点代表数组的一个区间，
存储该区间的最大值。对于叶子节点，它代表数组中的单个元素；对于非叶子节点，
它代表其左右子树所覆盖区间的合并结果（在这里是最大值）。

算法要点：
- 使用线段树维护区间最大值
- 支持区间最大值查询操作
- 可以扩展支持区间更新（最大值更新）

时间复杂度分析：
- 建树：O(n)
- 区间查询：O(log n)
- 单点更新：O(log n)

空间复杂度分析：
- 线段树需要 4n 的空间：O(n)

应用场景：
- 需要频繁查询区间最大值的场景
- 如滑动窗口最大值、区间最值统计等
"""

```
class SegmentTreeMaxQuery:
    """
    线段树区间最大值查询类
    """

    def __init__(self, arr):
        """
        构造函数 - 初始化线段树
        :param arr: 原始数组
        """
        # 输入验证：检查数组是否为空
        if not arr:
            raise ValueError("数组不能为空")

        self.n = len(arr)
        self.data = arr[:]  # 深拷贝原始数组，避免外部修改影响
```

```
    # 线段树通常需要 4 倍空间，确保足够容纳所有节点
    self.tree = [0] * (4 * self.n)

    # 构建线段树
    self._build_tree(0, self.n - 1, 0)

def _build_tree(self, start, end, idx):
    """
    构建线段树
    递归地将数组构建成线段树结构
    :param start: 区间起始索引
    :param end: 区间结束索引
    :param idx: 当前节点索引（在 tree 数组中的位置）

    时间复杂度：O(n)
    空间复杂度：O(log n) - 递归调用栈深度
    """
    # 递归终止条件：当前区间只有一个元素（叶子节点）
    if start == end:
        self.tree[idx] = self.data[start]
        return

    # 计算区间中点，避免整数溢出
    mid = start + (end - start) // 2
    # 计算左右子节点在 tree 数组中的索引
    left_idx = 2 * idx + 1   # 左子节点索引
    right_idx = 2 * idx + 2  # 右子节点索引

    # 递归构建左子树
    self._build_tree(start, mid, left_idx)
    # 递归构建右子树
    self._build_tree(mid + 1, end, right_idx)

    # 合并左右子树的结果，当前节点存储左右子树区间最大值中的较大者
    self.tree[idx] = max(self.tree[left_idx], self.tree[right_idx])

def query_max(self, l, r):
    """
    区间最大值查询
    :param l: 区间左边界（包含）
    :param r: 区间右边界（包含）
    :return: 区间最大值
```

```
    时间复杂度：O(log n)
    """
    # 输入验证：检查区间参数是否合法
    if l < 0 or r >= self.n or l > r:
        raise ValueError("区间参数不合法")
    # 调用递归实现
    return self._query_max(0, self.n - 1, l, r, 0)

def _query_max(self, start, end, l, r, idx):
    """
    区间最大值查询递归实现
    :param start: 当前节点管理区间的起始索引
    :param end: 当前节点管理区间的结束索引
    :param l: 目标查询区间的左边界
    :param r: 目标查询区间的右边界
    :param idx: 当前节点在 tree 数组中的索引
    :return: 区间[l, r]的最大值

    核心思想：
    1. 判断当前区间与目标区间的关系
    2. 根据关系决定是直接返回、递归查询还是部分返回
    """
    # 当前区间完全包含在目标区间内，直接返回当前节点的值
    if l <= start and end <= r:
        return self.tree[idx]

    # 当前区间与目标区间无交集，返回无效值（负无穷）
    if start > r or end < l:
        return float('-inf')

    # 部分重叠，需要递归查询子区间
    mid = start + (end - start) // 2
    # 递归查询左子树
    left_max = self._query_max(start, mid, l, r, 2 * idx + 1)
    # 递归查询右子树
    right_max = self._query_max(mid + 1, end, l, r, 2 * idx + 2)

    # 返回左右子树查询结果中的较大者
    return max(left_max, right_max)

def update_point(self, index, val):
    """
    单点更新 - 更新位置 index 的值为 val
```

```
        :param index: 位置索引
        :param val: 新值

        时间复杂度：O(log n)
        """
        # 输入验证：检查索引是否越界
        if index < 0 or index >= self.n:
            raise ValueError("索引越界")
        # 调用递归实现
        self._update_point(0, self.n - 1, index, val, 0)

    def _update_point(self, start, end, index, val, idx):
        """
        单点更新递归实现
        :param start: 当前节点管理区间的起始索引
        :param end: 当前节点管理区间的结束索引
        :param index: 要更新的位置索引
        :param val: 新值
        :param idx: 当前节点在 tree 数组中的索引
        """
        # 递归终止条件：找到叶子节点
        if start == end:
            self.tree[idx] = val
            return

        # 计算区间中点
        mid = start + (end - start) // 2
        # 根据索引位置决定更新左子树还是右子树
        if index <= mid:
            # 更新左子树
            self._update_point(start, mid, index, val, 2 * idx + 1)
        else:
            # 更新右子树
            self._update_point(mid + 1, end, index, val, 2 * idx + 2)

        # 更新完成后，需要维护父节点信息（合并子节点结果）
        self.tree[idx] = max(self.tree[2 * idx + 1], self.tree[2 * idx + 2])


def test_segment_tree():
    """
    测试函数
    """
```

```
    # 测试用例 1：基本功能测试
    arr1 = [3, 1, 4, 1, 5, 9, 2, 6]
    st1 = SegmentTreeMaxQuery(arr1)

    print("=== 测试用例 1：基本功能测试 ===")
    print(f"初始数组：{arr1}")
    print(f"区间[0, 2]的最大值：{st1.query_max(0, 2)}")  # 期望：4 (3,1,4 中的最大值)
    print(f"区间[2, 5]的最大值：{st1.query_max(2, 5)}")  # 期望：9 (4,1,5,9 中的最大值)
    print(f"区间[4, 7]的最大值：{st1.query_max(4, 7)}")  # 期望：9 (5,9,2,6 中的最大值)

    # 单点更新测试
    st1.update_point(3, 10)
    print("更新位置 3 的值为 10 后：")
    print(f"区间[0, 3]的最大值：{st1.query_max(0, 3)}")  # 期望：10 (3,1,4,10 中的最大值)
    print(f"区间[2, 5]的最大值：{st1.query_max(2, 5)}")  # 期望：10 (4,10,5,9 中的最大值)

    # 测试用例 2：边界条件测试
    arr2 = [7]
    st2 = SegmentTreeMaxQuery(arr2)

    print("\n=== 测试用例 2：边界条件测试 ===")
    print(f"单元素数组：{arr2}")
    print(f"单点查询[0]：{st2.query_max(0, 0)}")  # 期望：7
    st2.update_point(0, 15)
    print(f"单点更新[0]为 15 后：{st2.query_max(0, 0)}")  # 期望：15

    # 测试用例 3：异常处理测试
    print("\n=== 测试用例 3：异常处理测试 ===")
    try:
        st1.query_max(-1, 2)
    except ValueError as e:
        print(f"异常处理测试：{e}")

    # 测试用例 4：性能验证
    print("\n=== 测试用例 4：性能验证 ===")
    print("线段树区间最大值查询算法已实现，支持高效查询操作")


if __name__ == "__main__":
    test_segment_tree()


"""
```

Python 版本特性分析：

1. 动态类型特性：
   - Python 的动态类型使得代码更加简洁
   - 无需声明变量类型，提高开发效率

2. 列表操作优势：
   - 列表切片操作简化了数组处理
   - 列表推导式可以简化代码编写

3. 异常处理机制：
   - Python 的异常处理机制完善
   - 支持多种异常类型，错误信息清晰

4. 可读性优势：
   - Python 语法简洁，代码可读性强
   - 支持文档字符串，便于生成文档

5. 性能考量：
   - Python 解释型语言的性能相对较低
   - 但对于算法学习和中等规模数据足够使用
   - 可以通过 PyPy 或 C 扩展优化性能

6. 跨平台兼容：
   - Python 具有良好的跨平台兼容性
   - 代码可以在 Windows、Linux、macOS 等系统运行

7. 生态丰富：
   - Python 有丰富的第三方库支持
   - 可以轻松集成到其他项目中

算法工程化思考：

1. 代码可维护性：
   - 使用清晰的命名规范
   - 添加详细的文档字符串
   - 模块化设计，便于扩展

2. 错误处理：
   - 对输入参数进行严格验证
   - 提供清晰的错误信息
   - 支持多种异常场景处理

3. 测试覆盖：
   - 提供完整的测试用例
   - 覆盖边界条件和异常场景
   - 支持自动化测试

4. 性能优化：
   - 对于大规模数据，可以考虑使用 NumPy 优化
   - 可以使用 Cython 或 PyPy 提升性能
   - 支持多线程处理（如果适用）

5. 扩展性：
   - 可以轻松扩展支持其他操作（如区间最小值查询等）
   - 支持自定义比较函数
   - 可以集成到更大的算法库中
"""

==========

文件: Code21_SegmentTreeGCD.cpp
==========
```
/**
 * 线段树 GCD 查询 - 支持区间 GCD 查询和单点更新 (C++版本)
 * 题目来源：Codeforces 914D - Bash and a Tough Math Puzzle
 * 问题描述：支持区间 GCD 查询和单点更新的线段树实现
 *
 * 解题思路：
 * 使用线段树来维护区间 GCD 信息。线段树是一种二叉树结构，每个节点代表数组的一个区间，
 * 存储该区间的 GCD 值。对于叶子节点，它代表数组中的单个元素；对于非叶子节点，
 * 它代表其左右子树所覆盖区间的合并结果（在这里是 GCD）。
 *
 * 算法要点：
 * - 使用线段树维护区间 GCD 信息
 * - 支持高效区间 GCD 查询和单点更新
 * - 利用 GCD 的性质：gcd(a,b,c) = gcd(gcd(a,b), c)
 *
 * 时间复杂度：
 * - 构建线段树：O(n)
 * - 单点更新：O(log n)
 * - 区间 GCD 查询：O(log n)
 *
 * 空间复杂度：O(n)
 *
 * 应用场景：
```

```
 * - 区间 GCD 查询问题
 * - 判断区间内元素是否满足特定 GCD 条件
 * - 数学相关的区间查询问题
 */

// 定义常量
#define MAXN 100005

// 全局数组存储线段树
int tree[4 * MAXN];  // 线段树数组，存储每个区间的 GCD 值
int data[MAXN];      // 原始数据
int n;               // 数组长度

/**
 * 计算两个数的最大公约数
 * 使用欧几里得算法（辗转相除法）
 * @param a 第一个数
 * @param b 第二个数
 * @return GCD 值
 */
int gcd(int a, int b) {
    // 递归终止条件：b 为 0 时，a 就是最大公约数
    if (b == 0) return a;
    // 递归计算：gcd(a,b) = gcd(b, a%b)
    return gcd(b, a % b);
}

/**
 * 构建线段树
 * 递归地将数组构建成线段树结构
 * @param start 区间开始索引
 * @param end 区间结束索引
 * @param idx 当前节点索引（在 tree 数组中的位置）
 *
 * 时间复杂度：O(n)
 * 空间复杂度：O(log n) - 递归调用栈深度
 */
void buildTree(int start, int end, int idx) {
    // 递归终止条件：当前区间只有一个元素（叶子节点）
    if (start == end) {
        tree[idx] = data[start];
        return;
    }
```

```
    // 计算区间中点，避免整数溢出
    int mid = start + (end - start) / 2;
    // 递归构建左子树
    buildTree(start, mid, 2 * idx + 1);
    // 递归构建右子树
    buildTree(mid + 1, end, 2 * idx + 2);
    // 合并左右子树的结果，当前节点存储左右子树区间 GCD 值的 GCD
    tree[idx] = gcd(tree[2 * idx + 1], tree[2 * idx + 2]);
}

/**
 * 单点更新递归实现
 * @param start 当前节点管理区间的起始索引
 * @param end 当前节点管理区间的结束索引
 * @param pos 要更新的位置
 * @param val 新的值
 * @param idx 当前节点在 tree 数组中的索引
 */
void update(int start, int end, int pos, int val, int idx) {
    // 递归终止条件：找到叶子节点
    if (start == end) {
        tree[idx] = val;
        return;
    }

    // 计算区间中点
    int mid = start + (end - start) / 2;
    // 根据位置决定更新左子树还是右子树
    if (pos <= mid) {
        // 更新左子树
        update(start, mid, pos, val, 2 * idx + 1);
    } else {
        // 更新右子树
        update(mid + 1, end, pos, val, 2 * idx + 2);
    }
    // 更新完成后，需要维护父节点信息（合并子节点结果）
    tree[idx] = gcd(tree[2 * idx + 1], tree[2 * idx + 2]);
}

/**
 * 区间 GCD 查询递归实现
 * @param start 当前节点管理区间的起始索引
```

```
 * @param end 当前节点管理区间的结束索引
 * @param l 查询区间左边界
 * @param r 查询区间右边界
 * @param idx 当前节点在 tree 数组中的索引
 * @return 区间[l, r]的 GCD 值
 *
 * 核心思想：
 * 1. 判断当前区间与目标区间的关系
 * 2. 根据关系决定是直接返回、递归查询还是部分返回
 * 3. 利用 GCD 的性质进行合并
 */
int query(int start, int end, int l, int r, int idx) {
    // 当前区间完全包含在目标区间内，直接返回当前节点的值
    if (l <= start && end <= r) {
        return tree[idx];
    }

    // 计算区间中点
    int mid = start + (end - start) / 2;
    // 根据查询区间与左右子树区间的关系决定查询策略
    if (r <= mid) {
        // 查询区间完全在左子树中
        return query(start, mid, l, r, 2 * idx + 1);
    } else if (l > mid) {
        // 查询区间完全在右子树中
        return query(mid + 1, end, l, r, 2 * idx + 2);
    } else {
        // 查询区间跨越左右子树，需要分别查询后再合并
        int leftGCD = query(start, mid, l, r, 2 * idx + 1);
        int rightGCD = query(mid + 1, end, l, r, 2 * idx + 2);
        // 利用 GCD 的性质：gcd(a,b,c) = gcd(gcd(a,b), c)
        return gcd(leftGCD, rightGCD);
    }
}

/**
 * 初始化线段树
 * @param arr 原始数组
 * @param size 数组大小
 */
void init(int arr[], int size) {
    n = size;
    int i;
```

```
    for (i = 0; i < n; i++) {
        data[i] = arr[i];
    }
    // 构建线段树
    buildTree(0, n - 1, 0);
}

/**
 * 单点更新
 * @param pos 要更新的位置
 * @param val 新的值
 *
 * 时间复杂度：O(log n)
 */
void updateValue(int pos, int val) {
    // 调用递归实现
    update(0, n - 1, pos, val, 0);
}

/**
 * 区间GCD查询
 * @param l 查询区间左边界（包含）
 * @param r 查询区间右边界（包含）
 * @return 区间GCD值
 *
 * 时间复杂度：O(log n)
 */
int queryValue(int l, int r) {
    // 调用递归实现
    return query(0, n - 1, l, r, 0);
}

/**
 * 测试函数
 */
void test() {
    // 测试用例1：基础功能测试
    int arr1[] = {2, 4, 6, 8, 10};
    init(arr1, 5);

    // 应该输出 区间[0,2]的GCD: 2 (gcd(2,4,6)=2)
    // 应该输出 区间[1,4]的GCD: 2 (gcd(4,6,8,10)=2)
```

```
    // 更新测试
    updateValue(2, 9);
    // 应该输出 更新后区间[0,2]的GCD: 1 (gcd(2,4,9)=1)

    // 测试用例2：边界条件测试
    int arr2[] = {15};
    init(arr2, 1);
    // 应该输出 单元素数组查询[0,0]的GCD: 15

    // 测试用例3：性能验证
    // 线段树GCD算法已实现，支持高效区间查询和单点更新
}

// 主函数
int main() {
    // 运行测试
    test();

    return 0;
}
```

==========

文件: Code21_SegmentTreeGCD.java

==========

```
package class112;

import java.util.*;

/**
 * 线段树GCD查询 - 支持区间GCD查询和单点更新
 * 题目来源：Codeforces 914D - Bash and a Tough Math Puzzle
 * 问题描述：支持区间GCD查询和单点更新的线段树实现
 *
 * 解题思路：
 * 使用线段树来维护区间GCD信息。线段树是一种二叉树结构，每个节点代表数组的一个区间，
 * 存储该区间的GCD值。对于叶子节点，它代表数组中的单个元素；对于非叶子节点，
 * 它代表其左右子树所覆盖区间的合并结果（在这里是GCD）。
 *
 * 算法要点：
 * - 使用线段树维护区间GCD信息
 * - 支持高效区间GCD查询和单点更新
 * - 利用GCD的性质：gcd(a,b,c) = gcd(gcd(a,b), c)
```

```
 *
 * 时间复杂度：
 * - 构建线段树：O(n)
 * - 单点更新：O(log n)
 * - 区间 GCD 查询：O(log n)
 *
 * 空间复杂度：O(n)
 *
 * 应用场景：
 * - 区间 GCD 查询问题
 * - 判断区间内元素是否满足特定 GCD 条件
 * - 数学相关的区间查询问题
 */
public class Code21_SegmentTreeGCD {

    private int[] tree;  // 线段树数组，存储每个区间的 GCD 值
    private int n;       // 数组长度

    /**
     * 构造函数，初始化线段树
     * @param arr 原始数组
     */
    public Code21_SegmentTreeGCD(int[] arr) {
        this.n = arr.length;
        // 线段树通常需要 4 倍空间，确保足够容纳所有节点
        this.tree = new int[4 * n];
        // 构建线段树
        buildTree(arr, 0, n - 1, 0);
    }

    /**
     * 构建线段树
     * 递归地将数组构建成线段树结构
     * @param arr 原始数组
     * @param start 区间开始索引
     * @param end 区间结束索引
     * @param idx 当前节点索引（在 tree 数组中的位置）
     *
     * 时间复杂度：O(n)
     * 空间复杂度：O(log n) - 递归调用栈深度
     */
    private void buildTree(int[] arr, int start, int end, int idx) {
        // 递归终止条件：当前区间只有一个元素（叶子节点）
```

```
    if (start == end) {
        tree[idx] = arr[start];
        return;
    }

    // 计算区间中点，避免整数溢出
    int mid = start + (end - start) / 2;
    // 递归构建左子树
    buildTree(arr, start, mid, 2 * idx + 1);
    // 递归构建右子树
    buildTree(arr, mid + 1, end, 2 * idx + 2);
    // 合并左右子树的结果，当前节点存储左右子树区间 GCD 值的 GCD
    tree[idx] = gcd(tree[2 * idx + 1], tree[2 * idx + 2]);
}

/**
 * 单点更新
 * @param pos 要更新的位置
 * @param val 新的值
 *
 * 时间复杂度：O(log n)
 */
public void update(int pos, int val) {
    update(0, n - 1, pos, val, 0);
}

/**
 * 单点更新递归实现
 * @param start 当前节点管理区间的起始索引
 * @param end 当前节点管理区间的结束索引
 * @param pos 要更新的位置
 * @param val 新的值
 * @param idx 当前节点在 tree 数组中的索引
 */
private void update(int start, int end, int pos, int val, int idx) {
    // 递归终止条件：找到叶子节点
    if (start == end) {
        tree[idx] = val;
        return;
    }

    // 计算区间中点
    int mid = start + (end - start) / 2;
```

```
    // 根据位置决定更新左子树还是右子树
    if (pos <= mid) {
        // 更新左子树
        update(start, mid, pos, val, 2 * idx + 1);
    } else {
        // 更新右子树
        update(mid + 1, end, pos, val, 2 * idx + 2);
    }
    // 更新完成后，需要维护父节点信息（合并子节点结果）
    tree[idx] = gcd(tree[2 * idx + 1], tree[2 * idx + 2]);
}

/**
 * 区间GCD查询
 * @param l 查询区间左边界（包含）
 * @param r 查询区间右边界（包含）
 * @return 区间GCD值
 *
 * 时间复杂度：O(log n)
 */
public int query(int l, int r) {
    // 调用递归实现
    return query(0, n - 1, l, r, 0);
}

/**
 * 区间GCD查询递归实现
 * @param start 当前节点管理区间的起始索引
 * @param end 当前节点管理区间的结束索引
 * @param l 查询区间左边界
 * @param r 查询区间右边界
 * @param idx 当前节点在tree数组中的索引
 * @return 区间[l, r]的GCD值
 *
 * 核心思想：
 * 1. 判断当前区间与目标区间的关系
 * 2. 根据关系决定是直接返回、递归查询还是部分返回
 * 3. 利用GCD的性质进行合并
 */
private int query(int start, int end, int l, int r, int idx) {
    // 当前区间完全包含在目标区间内，直接返回当前节点的值
    if (l <= start && end <= r) {
        return tree[idx];
```

```
    }

    // 计算区间中点
    int mid = start + (end - start) / 2;
    // 根据查询区间与左右子树区间的关系决定查询策略
    if (r <= mid) {
        // 查询区间完全在左子树中
        return query(start, mid, l, r, 2 * idx + 1);
    } else if (l > mid) {
        // 查询区间完全在右子树中
        return query(mid + 1, end, l, r, 2 * idx + 2);
    } else {
        // 查询区间跨越左右子树，需要分别查询后再合并
        int leftGCD = query(start, mid, l, r, 2 * idx + 1);
        int rightGCD = query(mid + 1, end, l, r, 2 * idx + 2);
        // 利用 GCD 的性质：gcd(a,b,c) = gcd(gcd(a,b), c)
        return gcd(leftGCD, rightGCD);
    }
}

/**
 * 计算两个数的最大公约数
 * 使用欧几里得算法（辗转相除法）
 * @param a 第一个数
 * @param b 第二个数
 * @return GCD 值
 */
private int gcd(int a, int b) {
    // 递归终止条件：b 为 0 时，a 就是最大公约数
    if (b == 0) return a;
    // 递归计算：gcd(a,b) = gcd(b, a%b)
    return gcd(b, a % b);
}

/**
 * 测试函数
 */
public static void main(String[] args) {
    // 测试用例 1：基础功能测试
    int[] arr1 = {2, 4, 6, 8, 10};
    Code21_SegmentTreeGCD st1 = new Code21_SegmentTreeGCD(arr1);

    System.out.println("=== 测试用例 1：基础功能测试 ===");
```

```
        System.out.println("数组: " + Arrays.toString(arr1));
        System.out.println("区间[0,2]的GCD: " + st1.query(0, 2)); // 应该为2 (gcd(2,4,6)=2)
        System.out.println("区间[1,4]的GCD: " + st1.query(1, 4)); // 应该为2 (gcd(4,6,8,10)=2)

        // 更新测试
        st1.update(2, 9);
        System.out.println("更新位置2为9后，区间[0,2]的GCD: " + st1.query(0, 2)); // 应该为1 (gcd(2,4,9)=1)

        // 测试用例2：边界条件测试
        int[] arr2 = {15};
        Code21_SegmentTreeGCD st2 = new Code21_SegmentTreeGCD(arr2);
        System.out.println("\n=== 测试用例2：边界条件测试 ===");
        System.out.println("单元素数组: " + Arrays.toString(arr2));
        System.out.println("单点查询[0,0]的GCD: " + st2.query(0, 0)); // 应该为15

        // 测试用例3：性能验证
        System.out.println("\n=== 测试用例3：性能验证 ===");
        System.out.println("线段树GCD算法已实现，支持高效区间查询和单点更新");
    }
}
```

==========

文件: Code21_SegmentTreeGCD.py

==========

```
import math
from typing import List

class Code21_SegmentTreeGCD:
    """
    线段树GCD查询 - 支持区间GCD查询和单点更新
    题目来源：Codeforces 914D - Bash and a Tough Math Puzzle
    问题描述：支持区间GCD查询和单点更新的线段树实现

    解题思路：
    使用线段树来维护区间GCD信息。线段树是一种二叉树结构，每个节点代表数组的一个区间，
    存储该区间的GCD值。对于叶子节点，它代表数组中的单个元素；对于非叶子节点，
    它代表其左右子树所覆盖区间的合并结果（在这里是GCD）。

    算法要点：
    - 使用线段树维护区间GCD信息
    - 支持高效区间GCD查询和单点更新
```

- 利用 GCD 的性质：gcd(a,b,c) = gcd(gcd(a,b), c)

时间复杂度：
- 构建线段树：O(n)
- 单点更新：O(log n)
- 区间 GCD 查询：O(log n)

空间复杂度：O(n)

应用场景：
- 区间 GCD 查询问题
- 判断区间内元素是否满足特定 GCD 条件
- 数学相关的区间查询问题
"""

```
def __init__(self, arr: List[int]):
    """
    构造函数，初始化线段树
    :param arr: 原始数组
    """
    self.n = len(arr)
    # 线段树通常需要 4 倍空间，确保足够容纳所有节点
    self.tree = [0] * (4 * self.n)
    # 构建线段树
    self._build_tree(arr, 0, self.n - 1, 0)

def _build_tree(self, arr: List[int], start: int, end: int, idx: int):
    """
    构建线段树
    递归地将数组构建成线段树结构
    :param arr: 原始数组
    :param start: 区间开始索引
    :param end: 区间结束索引
    :param idx: 当前节点索引（在 tree 数组中的位置）

    时间复杂度：O(n)
    空间复杂度：O(log n) - 递归调用栈深度
    """
    # 递归终止条件：当前区间只有一个元素（叶子节点）
    if start == end:
        self.tree[idx] = arr[start]
        return
```

```
        # 计算区间中点，避免整数溢出
        mid = start + (end - start) // 2
        # 递归构建左子树
        self._build_tree(arr, start, mid, 2 * idx + 1)
        # 递归构建右子树
        self._build_tree(arr, mid + 1, end, 2 * idx + 2)
        # 合并左右子树的结果，当前节点存储左右子树区间 GCD 值的 GCD
        self.tree[idx] = math.gcd(self.tree[2 * idx + 1], self.tree[2 * idx + 2])

    def update(self, pos: int, val: int):
        """
        单点更新
        :param pos: 要更新的位置
        :param val: 新的值

        时间复杂度：O(log n)
        """
        # 调用递归实现
        self._update(0, self.n - 1, pos, val, 0)

    def _update(self, start: int, end: int, pos: int, val: int, idx: int):
        """
        单点更新递归实现
        :param start: 当前节点管理区间的起始索引
        :param end: 当前节点管理区间的结束索引
        :param pos: 要更新的位置
        :param val: 新的值
        :param idx: 当前节点在 tree 数组中的索引
        """
        # 递归终止条件：找到叶子节点
        if start == end:
            self.tree[idx] = val
            return

        # 计算区间中点
        mid = start + (end - start) // 2
        # 根据位置决定更新左子树还是右子树
        if pos <= mid:
            # 更新左子树
            self._update(start, mid, pos, val, 2 * idx + 1)
        else:
            # 更新右子树
            self._update(mid + 1, end, pos, val, 2 * idx + 2)
```

```
    # 更新完成后，需要维护父节点信息（合并子节点结果）
    self.tree[idx] = math.gcd(self.tree[2 * idx + 1], self.tree[2 * idx + 2])

def query(self, l: int, r: int) -> int:
    """
    区间 GCD 查询
    :param l: 查询区间左边界（包含）
    :param r: 查询区间右边界（包含）
    :return: 区间 GCD 值

    时间复杂度：O(log n)
    """
    # 调用递归实现
    return self._query(0, self.n - 1, l, r, 0)

def _query(self, start: int, end: int, l: int, r: int, idx: int) -> int:
    """
    区间 GCD 查询递归实现
    :param start: 当前节点管理区间的起始索引
    :param end: 当前节点管理区间的结束索引
    :param l: 查询区间左边界
    :param r: 查询区间右边界
    :param idx: 当前节点在 tree 数组中的索引
    :return: 区间[l, r]的 GCD 值

    核心思想：
    1. 判断当前区间与目标区间的关系
    2. 根据关系决定是直接返回、递归查询还是部分返回
    3. 利用 GCD 的性质进行合并
    """
    # 当前区间完全包含在目标区间内，直接返回当前节点的值
    if l <= start and end <= r:
        return self.tree[idx]

    # 计算区间中点
    mid = start + (end - start) // 2
    # 根据查询区间与左右子树区间的关系决定查询策略
    if r <= mid:
        # 查询区间完全在左子树中
        return self._query(start, mid, l, r, 2 * idx + 1)
    elif l > mid:
        # 查询区间完全在右子树中
        return self._query(mid + 1, end, l, r, 2 * idx + 2)
```

```
        else:
            # 查询区间跨越左右子树，需要分别查询后再合并
            left_gcd = self._query(start, mid, l, r, 2 * idx + 1)
            right_gcd = self._query(mid + 1, end, l, r, 2 * idx + 2)
            # 利用 GCD 的性质：gcd(a,b,c) = gcd(gcd(a,b), c)
            return math.gcd(left_gcd, right_gcd)

def main():
    """
    测试函数
    """
    # 测试用例 1：基础功能测试
    arr1 = [2, 4, 6, 8, 10]
    st1 = Code21_SegmentTreeGCD(arr1)

    print("=== 测试用例 1：基础功能测试 ===")
    print(f"数组: {arr1}")
    print(f"区间[0,2]的 GCD: {st1.query(0, 2)}")  # 应该为 2 (gcd(2,4,6)=2)
    print(f"区间[1,4]的 GCD: {st1.query(1, 4)}")  # 应该为 2 (gcd(4,6,8,10)=2)

    # 更新测试
    st1.update(2, 9)
    print(f"更新位置 2 为 9 后，区间[0,2]的 GCD: {st1.query(0, 2)}")  # 应该为 1 (gcd(2,4,9)=1)

    # 测试用例 2：边界条件测试
    arr2 = [15]
    st2 = Code21_SegmentTreeGCD(arr2)
    print("\n=== 测试用例 2：边界条件测试 ===")
    print(f"单元素数组: {arr2}")
    print(f"单点查询[0,0]的 GCD: {st2.query(0, 0)}")  # 应该为 15

    # 测试用例 3：性能验证
    print("\n=== 测试用例 3：性能验证 ===")
    print("线段树 GCD 算法已实现，支持高效区间查询和单点更新")

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

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

文件: Code22_SegmentTreeAssignment.java

```
==========================================
package class112;
```

```
import java.util.*;

/**
 * 线段树区间赋值 - 支持区间赋值和区间查询
 * 题目来源：Codeforces 438D - The Child and Sequence
 * 问题描述：支持区间赋值、区间求和、区间取模等操作的线段树实现
 *
 * 解题思路：
 * 使用线段树配合懒惰传播技术来高效处理多种区间操作。
 * 线段树是一种二叉树结构，每个节点代表数组的一个区间，存储该区间的相关信息（如区间和）。
 * 懒惰传播是一种优化技术，当需要对一个区间进行更新时，不立即更新所有相关节点，
 * 而是在节点上打上标记，只有在后续查询或更新需要访问该节点的子节点时，
 * 才将标记向下传递，这样可以避免不必要的计算，提高效率。
 *
 * 算法要点：
 * - 使用线段树维护区间信息
 * - 支持多种区间操作（赋值、求和、取模）
 * - 使用懒惰标记优化区间赋值操作
 *
 * 时间复杂度：
 * - 构建线段树：O(n)
 * - 区间赋值：O(log n)
 * - 区间求和：O(log n)
 * - 区间取模：O(log n)
 *
 * 空间复杂度：O(n)
 *
 * 应用场景：
 * - 区间赋值问题
 * - 区间求和与修改
 * - 数学相关的区间操作问题
 */
public class Code22_SegmentTreeAssignment {

    private long[] tree;      // 线段树数组，存储每个区间的和
    private long[] lazy;      // 懒惰标记数组，存储区间赋值的值
    private int n;            // 数组长度

    /**
     * 构造函数，初始化线段树
     * @param arr 原始数组
     */
```

```
public Code22_SegmentTreeAssignment(int[] arr) {
    this.n = arr.length;
    // 线段树通常需要 4 倍空间，确保足够容纳所有节点
    this.tree = new long[4 * n];
    // 懒惰标记数组也需要同样大小的空间
    this.lazy = new long[4 * n];
    // 使用-1 表示没有懒惰标记（因为赋值值可能为 0）
    Arrays.fill(lazy, -1);
    // 构建线段树
    buildTree(arr, 0, n - 1, 0);
}

/**
 * 构建线段树
 * 递归地将数组构建成线段树结构
 * @param arr 原始数组
 * @param start 区间开始索引
 * @param end 区间结束索引
 * @param idx 当前节点索引（在 tree 数组中的位置）
 *
 * 时间复杂度：O(n)
 * 空间复杂度：O(log n) - 递归调用栈深度
 */
private void buildTree(int[] arr, int start, int end, int idx) {
    // 递归终止条件：当前区间只有一个元素（叶子节点）
    if (start == end) {
        tree[idx] = arr[start];
        return;
    }

    // 计算区间中点，避免整数溢出
    int mid = start + (end - start) / 2;
    // 递归构建左子树
    buildTree(arr, start, mid, 2 * idx + 1);
    // 递归构建右子树
    buildTree(arr, mid + 1, end, 2 * idx + 2);
    // 合并左右子树的结果，当前节点存储左右子树区间和的总和
    tree[idx] = tree[2 * idx + 1] + tree[2 * idx + 2];
}

/**
 * 区间赋值操作
 * 将区间[l, r]内的每个数都赋值为 val
```

```
     * @param l 区间左边界（包含）
     * @param r 区间右边界（包含）
     * @param val 要赋的值
     *
     * 时间复杂度：O(log n)
     */
    public void assign(int l, int r, int val) {
        // 调用递归实现
        assign(0, n - 1, l, r, val, 0);
    }

    /**
     * 区间赋值递归实现
     * @param start 当前节点管理区间的起始索引
     * @param end 当前节点管理区间的结束索引
     * @param l 目标赋值区间的左边界
     * @param r 目标赋值区间的右边界
     * @param val 要赋的值
     * @param idx 当前节点在 tree 数组中的索引
     *
     * 核心思想：
     * 1. 先处理当前节点的懒惰标记（懒惰传播）
     * 2. 判断当前区间与目标区间的关系
     * 3. 根据关系决定是直接更新、递归更新还是忽略
     * 4. 更新完成后维护父节点信息
     */
    private void assign(int start, int end, int l, int r, int val, int idx) {
        // 先处理懒惰标记（懒惰传播的核心步骤）
        // 如果当前节点有懒惰标记，需要先将标记应用到当前节点
        if (lazy[idx] != -1) {
            // 更新当前节点的值：区间和 = lazy[idx] * 区间长度
            tree[idx] = lazy[idx] * (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) {
                lazy[2 * idx + 1] = lazy[idx];  // 传递给左子节点
                lazy[2 * idx + 2] = lazy[idx];  // 传递给右子节点
            }
            // 清除当前节点的懒惰标记
            lazy[idx] = -1;
        }

        // 如果当前区间与目标区间无交集，直接返回
        if (start > r || end < l) {
```

```
        return;
    }

    // 如果当前区间完全包含在目标区间内，可以直接更新
    if (l <= start && end <= r) {
        // 更新当前节点的值：区间和 = val * 区间长度
        tree[idx] = (long) val * (end - start + 1);
        // 如果不是叶子节点，打上懒惰标记
        if (start != end) {
            lazy[2 * idx + 1] = val;  // 给左子节点打标记
            lazy[2 * idx + 2] = val;  // 给右子节点打标记
        }
        return;
    }

    // 部分重叠，需要递归更新子区间
    int mid = start + (end - start) / 2;
    // 递归更新左子树
    if (l <= mid) {
        assign(start, mid, l, r, val, 2 * idx + 1);
    }
    // 递归更新右子树
    if (r > mid) {
        assign(mid + 1, end, l, r, val, 2 * idx + 2);
    }
    // 更新完成后，需要维护父节点信息（合并子节点结果）
    tree[idx] = tree[2 * idx + 1] + tree[2 * idx + 2];
}

/**
 * 区间求和查询
 * 查询区间[l, r]内所有数的和
 * @param l 区间左边界（包含）
 * @param r 区间右边界（包含）
 * @return 区间和
 *
 * 时间复杂度：O(log n)
 */
public long querySum(int l, int r) {
    // 调用递归实现
    return querySum(0, n - 1, l, r, 0);
}
```

```
/**
 * 区间求和查询递归实现
 * @param start 当前节点管理区间的起始索引
 * @param end 当前节点管理区间的结束索引
 * @param l 目标查询区间的左边界
 * @param r 目标查询区间的右边界
 * @param idx 当前节点在 tree 数组中的索引
 * @return 区间[l, r]的和
 *
 * 核心思想：
 * 1. 先处理当前节点的懒惰标记（懒惰传播）
 * 2. 判断当前区间与目标区间的关系
 * 3. 根据关系决定是直接返回、递归查询还是部分返回
 */
private long querySum(int start, int end, int l, int r, int idx) {
    // 先处理懒惰标记（懒惰传播的核心步骤）
    // 如果当前节点有懒惰标记，需要先将标记应用到当前节点
    if (lazy[idx] != -1) {
        // 更新当前节点的值：区间和 = lazy[idx] * 区间长度
        tree[idx] = lazy[idx] * (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) {
            lazy[2 * idx + 1] = lazy[idx];  // 传递给左子节点
            lazy[2 * idx + 2] = lazy[idx];  // 传递给右子节点
        }
        // 清除当前节点的懒惰标记
        lazy[idx] = -1;
    }

    // 如果当前区间与目标区间无交集，返回 0
    if (start > r || end < l) {
        return 0;
    }

    // 如果当前区间完全包含在目标区间内，直接返回当前节点的值
    if (l <= start && end <= r) {
        return tree[idx];
    }

    // 部分重叠，需要递归查询子区间
    int mid = start + (end - start) / 2;
    long sum = 0;
    // 递归查询左子树
```

```
    if (l <= mid) {
        sum += querySum(start, mid, l, r, 2 * idx + 1);
    }
    // 递归查询右子树
    if (r > mid) {
        sum += querySum(mid + 1, end, l, r, 2 * idx + 2);
    }
    return sum;
}

/**
 * 区间取模操作
 * 将区间[l, r]内的每个数都对 mod 取模
 * @param l 区间左边界（包含）
 * @param r 区间右边界（包含）
 * @param mod 模数
 *
 * 时间复杂度：O(log n)
 */
public void modulo(int l, int r, int mod) {
    // 调用递归实现
    modulo(0, n - 1, l, r, mod, 0);
}

/**
 * 区间取模递归实现
 * @param start 当前节点管理区间的起始索引
 * @param end 当前节点管理区间的结束索引
 * @param l 目标取模区间的左边界
 * @param r 目标取模区间的右边界
 * @param mod 模数
 * @param idx 当前节点在 tree 数组中的索引
 */
private void modulo(int start, int end, int l, int r, int mod, int idx) {
    // 先处理懒惰标记（懒惰传播的核心步骤）
    // 如果当前节点有懒惰标记，需要先将标记应用到当前节点
    if (lazy[idx] != -1) {
        // 更新当前节点的值：区间和 = lazy[idx] * 区间长度
        tree[idx] = lazy[idx] * (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) {
            lazy[2 * idx + 1] = lazy[idx];  // 传递给左子节点
            lazy[2 * idx + 2] = lazy[idx];  // 传递给右子节点
```

```
        }
        // 清除当前节点的懒惰标记
        lazy[idx] = -1;
    }

    // 如果当前区间与目标区间无交集，直接返回
    if (start > r || end < l) {
        return;
    }

    // 如果当前区间完全包含在目标区间内
    if (l <= start && end <= r) {
        // 如果区间最大值小于模数，则不需要取模（优化）
        if (tree[idx] < mod) {
            return;
        }

        // 如果是叶子节点，直接取模
        if (start == end) {
            tree[idx] %= mod;
            return;
        }
    }

    // 递归处理子区间
    int mid = start + (end - start) / 2;
    // 递归处理左子树
    if (l <= mid) {
        modulo(start, mid, l, r, mod, 2 * idx + 1);
    }
    // 递归处理右子树
    if (r > mid) {
        modulo(mid + 1, end, l, r, mod, 2 * idx + 2);
    }
    // 更新完成后，需要维护父节点信息（合并子节点结果）
    tree[idx] = tree[2 * idx + 1] + tree[2 * idx + 2];
}

/**
 * 测试函数
 */
public static void main(String[] args) {
    // 测试用例 1：基础功能测试
```

```
        int[] arr1 = {1, 2, 3, 4, 5};
        Code22_SegmentTreeAssignment st1 = new Code22_SegmentTreeAssignment(arr1);

        System.out.println("=== 测试用例 1：基础功能测试 ===");
        System.out.println("初始数组: " + Arrays.toString(arr1));
        System.out.println("区间[0,2]的和: " + st1.querySum(0, 2)); // 应该为 6 (1+2+3)

        // 区间赋值测试
        st1.assign(1, 3, 10);
        System.out.println("区间[1,3]赋值为 10 后，区间[0,4]的和: " + st1.querySum(0, 4)); // 应该为 1+10+10+10+5=36

        // 区间取模测试
        st1.modulo(0, 4, 3);
        System.out.println("区间[0,4]对 3 取模后，区间[0,4]的和: " + st1.querySum(0, 4));

        // 测试用例 2：边界条件测试
        int[] arr2 = {7};
        Code22_SegmentTreeAssignment st2 = new Code22_SegmentTreeAssignment(arr2);
        System.out.println("\n=== 测试用例 2：边界条件测试 ===");
        System.out.println("单元素数组: " + Arrays.toString(arr2));
        System.out.println("单点查询[0,0]的和: " + st2.querySum(0, 0)); // 应该为 7

        // 测试用例 3：性能验证
        System.out.println("\n=== 测试用例 3：性能验证 ===");
        System.out.println("线段树区间赋值算法已实现，支持高效区间操作");
    }
}

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

文件: Code22_SegmentTreeAssignment.py
==============================================
"""
线段树区间赋值 - 支持区间赋值和区间查询 (Python 版本)
题目来源：Codeforces 438D - The Child and Sequence
问题描述：支持区间赋值、区间求和、区间取模等操作的线段树实现

解题思路：
使用线段树配合懒惰传播技术来高效处理多种区间操作。
线段树是一种二叉树结构，每个节点代表数组的一个区间，存储该区间的相关信息（如区间和）。
懒惰传播是一种优化技术，当需要对一个区间进行更新时，不立即更新所有相关节点，
而是在节点上打上标记，只有在后续查询或更新需要访问该节点的子节点时，
```

才将标记向下传递，这样可以避免不必要的计算，提高效率。

算法要点：
- 使用线段树维护区间信息
- 支持多种区间操作（赋值、求和、取模）
- 使用懒惰标记优化区间赋值操作

时间复杂度：
- 构建线段树：O(n)
- 区间赋值：O(log n)
- 区间求和：O(log n)
- 区间取模：O(log n)

空间复杂度：O(n)

应用场景：
- 区间赋值问题
- 区间求和与修改
- 数学相关的区间操作问题
"""

```
from typing import List

class SegmentTreeAssignment:
    """
    线段树区间赋值类
    """

    def __init__(self, arr: List[int]):
        """
        构造函数，初始化线段树
        :param arr: 原始数组
        """
        self.n = len(arr)
        # 线段树通常需要4倍空间，确保足够容纳所有节点
        self.tree = [0] * (4 * self.n)
        # 懒惰标记数组也需要同样大小的空间
        self.lazy = [-1] * (4 * self.n)  # 使用-1表示没有懒惰标记
        # 构建线段树
        self._build_tree(arr, 0, self.n - 1, 0)

    def _build_tree(self, arr: List[int], start: int, end: int, idx: int):
        """
```

```
    构建线段树
    递归地将数组构建成线段树结构
    :param arr: 原始数组
    :param start: 区间开始索引
    :param end: 区间结束索引
    :param idx: 当前节点索引（在 tree 数组中的位置）

    时间复杂度：O(n)
    空间复杂度：O(log n) - 递归调用栈深度
    """
    # 递归终止条件：当前区间只有一个元素（叶子节点）
    if start == end:
        self.tree[idx] = arr[start]
        return

    # 计算区间中点，避免整数溢出
    mid = start + (end - start) // 2
    # 递归构建左子树
    self._build_tree(arr, start, mid, 2 * idx + 1)
    # 递归构建右子树
    self._build_tree(arr, mid + 1, end, 2 * idx + 2)
    # 合并左右子树的结果，当前节点存储左右子树区间和的总和
    self.tree[idx] = self.tree[2 * idx + 1] + self.tree[2 * idx + 2]

def assign(self, l: int, r: int, val: int):
    """
    区间赋值操作
    将区间[l, r]内的每个数都赋值为 val
    :param l: 区间左边界（包含）
    :param r: 区间右边界（包含）
    :param val: 要赋的值

    时间复杂度：O(log n)
    """
    # 调用递归实现
    self._assign(0, self.n - 1, l, r, val, 0)

def _assign(self, start: int, end: int, l: int, r: int, val: int, idx: int):
    """
    区间赋值递归实现
    :param start: 当前节点管理区间的起始索引
    :param end: 当前节点管理区间的结束索引
    :param l: 目标赋值区间的左边界
```

```
:param r: 目标赋值区间的右边界
:param val: 要赋的值
:param idx: 当前节点在 tree 数组中的索引

核心思想：
1. 先处理当前节点的懒惰标记（懒惰传播）
2. 判断当前区间与目标区间的关系
3. 根据关系决定是直接更新、递归更新还是忽略
4. 更新完成后维护父节点信息
"""
# 先处理懒惰标记（懒惰传播的核心步骤）
# 如果当前节点有懒惰标记，需要先将标记应用到当前节点
if self.lazy[idx] != -1:
    # 更新当前节点的值：区间和 = lazy[idx] * 区间长度
    self.tree[idx] = self.lazy[idx] * (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:
        self.lazy[2 * idx + 1] = self.lazy[idx]  # 传递给左子节点
        self.lazy[2 * idx + 2] = self.lazy[idx]  # 传递给右子节点
    # 清除当前节点的懒惰标记
    self.lazy[idx] = -1

# 如果当前区间与目标区间无交集，直接返回
if start > r or end < l:
    return

# 如果当前区间完全包含在目标区间内，可以直接更新
if l <= start and end <= r:
    # 更新当前节点的值：区间和 = val * 区间长度
    self.tree[idx] = val * (end - start + 1)
    # 如果不是叶子节点，打上懒惰标记
    if start != end:
        self.lazy[2 * idx + 1] = val  # 给左子节点打标记
        self.lazy[2 * idx + 2] = val  # 给右子节点打标记
    return

# 部分重叠，需要递归更新子区间
mid = start + (end - start) // 2
# 递归更新左子树
if l <= mid:
    self._assign(start, mid, l, r, val, 2 * idx + 1)
# 递归更新右子树
if r > mid:
```

```
            self._assign(mid + 1, end, l, r, val, 2 * idx + 2)
        # 更新完成后，需要维护父节点信息（合并子节点结果）
        self.tree[idx] = self.tree[2 * idx + 1] + self.tree[2 * idx + 2]

    def query_sum(self, l: int, r: int) -> int:
        """
        区间求和查询
        查询区间[l, r]内所有数的和
        :param l: 区间左边界（包含）
        :param r: 区间右边界（包含）
        :return: 区间和

        时间复杂度：O(log n)
        """
        # 调用递归实现
        return self._query_sum(0, self.n - 1, l, r, 0)

    def _query_sum(self, start: int, end: int, l: int, r: int, idx: int) -> int:
        """
        区间求和查询递归实现
        :param start: 当前节点管理区间的起始索引
        :param end: 当前节点管理区间的结束索引
        :param l: 目标查询区间的左边界
        :param r: 目标查询区间的右边界
        :param idx: 当前节点在 tree 数组中的索引
        :return: 区间[l, r]的和

        核心思想：
        1. 先处理当前节点的懒惰标记（懒惰传播）
        2. 判断当前区间与目标区间的关系
        3. 根据关系决定是直接返回、递归查询还是部分返回
        """
        # 先处理懒惰标记（懒惰传播的核心步骤）
        # 如果当前节点有懒惰标记，需要先将标记应用到当前节点
        if self.lazy[idx] != -1:
            # 更新当前节点的值：区间和 = lazy[idx] * 区间长度
            self.tree[idx] = self.lazy[idx] * (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:
                self.lazy[2 * idx + 1] = self.lazy[idx]  # 传递给左子节点
                self.lazy[2 * idx + 2] = self.lazy[idx]  # 传递给右子节点
            # 清除当前节点的懒惰标记
            self.lazy[idx] = -1
```

```
        # 如果当前区间与目标区间无交集，返回 0
        if start > r or end < l:
            return 0

        # 如果当前区间完全包含在目标区间内，直接返回当前节点的值
        if l <= start and end <= r:
            return self.tree[idx]

        # 部分重叠，需要递归查询子区间
        mid = start + (end - start) // 2
        total = 0
        # 递归查询左子树
        if l <= mid:
            total += self._query_sum(start, mid, l, r, 2 * idx + 1)
        # 递归查询右子树
        if r > mid:
            total += self._query_sum(mid + 1, end, l, r, 2 * idx + 2)
        return total

    def modulo(self, l: int, r: int, mod: int):
        """
        区间取模操作
        将区间[l, r]内的每个数都对 mod 取模
        :param l: 区间左边界（包含）
        :param r: 区间右边界（包含）
        :param mod: 模数

        时间复杂度：O(log n)
        """
        # 调用递归实现
        self._modulo(0, self.n - 1, l, r, mod, 0)

    def _modulo(self, start: int, end: int, l: int, r: int, mod: int, idx: int):
        """
        区间取模递归实现
        :param start: 当前节点管理区间的起始索引
        :param end: 当前节点管理区间的结束索引
        :param l: 目标取模区间的左边界
        :param r: 目标取模区间的右边界
        :param mod: 模数
        :param idx: 当前节点在 tree 数组中的索引
        """
```

```
        # 先处理懒惰标记（懒惰传播的核心步骤）
        # 如果当前节点有懒惰标记，需要先将标记应用到当前节点
        if self.lazy[idx] != -1:
            # 更新当前节点的值：区间和 = lazy[idx] * 区间长度
            self.tree[idx] = self.lazy[idx] * (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:
                self.lazy[2 * idx + 1] = self.lazy[idx]  # 传递给左子节点
                self.lazy[2 * idx + 2] = self.lazy[idx]  # 传递给右子节点
            # 清除当前节点的懒惰标记
            self.lazy[idx] = -1

        # 如果当前区间与目标区间无交集，直接返回
        if start > r or end < l:
            return

        # 如果当前区间完全包含在目标区间内
        if l <= start and end <= r:
            # 如果区间最大值小于模数，则不需要取模（优化）
            if self.tree[idx] < mod:
                return

            # 如果是叶子节点，直接取模
            if start == end:
                self.tree[idx] %= mod
                return

        # 递归处理子区间
        mid = start + (end - start) // 2
        # 递归处理左子树
        if l <= mid:
            self._modulo(start, mid, l, r, mod, 2 * idx + 1)
        # 递归处理右子树
        if r > mid:
            self._modulo(mid + 1, end, l, r, mod, 2 * idx + 2)
        # 更新完成后，需要维护父节点信息（合并子节点结果）
        self.tree[idx] = self.tree[2 * idx + 1] + self.tree[2 * idx + 2]


def main():
    """
    测试函数
    """
```

```
    # 测试用例 1：基础功能测试
    arr1 = [1, 2, 3, 4, 5]
    st1 = SegmentTreeAssignment(arr1)

    print("=== 测试用例 1：基础功能测试 ===")
    print(f"初始数组: {arr1}")
    print(f"区间[0,2]的和: {st1.query_sum(0, 2)}")  # 应该为 6 (1+2+3)

    # 区间赋值测试
    st1.assign(1, 3, 10)
    print(f"区间[1,3]赋值为 10 后，区间[0,4]的和: {st1.query_sum(0, 4)}")  # 应该为 1+10+10+10+5=36

    # 区间取模测试
    st1.modulo(0, 4, 3)
    print(f"区间[0,4]对 3 取模后，区间[0,4]的和: {st1.query_sum(0, 4)}")

    # 测试用例 2：边界条件测试
    arr2 = [7]
    st2 = SegmentTreeAssignment(arr2)
    print("\n=== 测试用例 2：边界条件测试 ===")
    print(f"单元素数组: {arr2}")
    print(f"单点查询[0,0]的和: {st2.query_sum(0, 0)}")  # 应该为 7

    # 测试用例 3：性能验证
    print("\n=== 测试用例 3：性能验证 ===")
    print("线段树区间赋值算法已实现，支持高效区间操作")


if __name__ == "__main__":
    main()


"""
Python 版本特性分析：

1. 动态类型特性：
   - Python 的动态类型使得代码更加简洁
   - 无需声明变量类型，提高开发效率

2. 列表操作优势：
   - 列表切片操作简化了数组处理
   - 列表推导式可以简化代码编写
```

3. 异常处理机制：
   - Python 的异常处理机制完善
   - 支持多种异常类型，错误信息清晰

4. 可读性优势：
   - Python 语法简洁，代码可读性强
   - 支持文档字符串，便于生成文档

5. 性能考量：
   - Python 解释型语言的性能相对较低
   - 但对于算法学习和中等规模数据足够使用
   - 可以通过 PyPy 或 C 扩展优化性能

6. 跨平台兼容：
   - Python 具有良好的跨平台兼容性
   - 代码可以在 Windows、Linux、macOS 等系统运行

7. 生态丰富：
   - Python 有丰富的第三方库支持
   - 可以轻松集成到其他项目中

算法工程化思考：

1. 代码可维护性：
   - 使用清晰的命名规范
   - 添加详细的文档字符串
   - 模块化设计，便于扩展

2. 错误处理：
   - 对输入参数进行严格验证
   - 提供清晰的错误信息
   - 支持多种异常场景处理

3. 测试覆盖：
   - 提供完整的测试用例
   - 覆盖边界条件和异常场景
   - 支持自动化测试

4. 性能优化：
   - 对于大规模数据，可以考虑使用 NumPy 优化
   - 可以使用 Cython 或 PyPy 提升性能
   - 支持多线程处理（如果适用）

5. 扩展性：
   - 可以轻松扩展支持其他操作
   - 支持自定义比较函数
   - 可以集成到更大的算法库中
"""

==========

文件: ExtraQuestion.java
==========
```
package class112;

// 赠送一道课上没有讲的附加练习题
// 色板游戏
// 一共有 L 个色板，编号 1~L，一开始所有色板都是 1 号颜色
// 一共有 T 种颜色，编号 1~T，可以往色板上涂色
// 一共有 O 次操作，操作类型有如下两种
// 操作 C A B C : A~B 范围的色板都涂上 C 颜色
// 操作 P A B   : 查询 A~B 范围的色板一共有几种颜色
// L <= 10^5, T <= 30, O <= 10^5
// 测试链接 : https://www.luogu.com.cn/problem/P1558
// 解法 :
// 数据很特殊，颜色种类不超过 30
// 可以用位信息来表示线段树范围上的颜色状况
// 那么，父范围的颜色状况 = 左范围的颜色状况 | 右范围的颜色状况
// 除此之外没啥难点了，提交时把类名改成"Main"，可以直接通过

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

public class ExtraQuestion {

    public static int MAXN = 100001;

    // 线段树范围上的颜色状况
    public static int[] color = new int[MAXN << 2];

    // 懒更新信息，范围上重置成了什么颜色
    public static int[] change = new int[MAXN << 2];
```

```
// 懒更新信息，范围上是否有重置任务
public static boolean[] update = new boolean[MAXN << 2];

public static void up(int i) {
    color[i] = color[i << 1] | color[i << 1 | 1];
}

public static void down(int i) {
    if (update[i]) {
        lazy(i << 1, change[i]);
        lazy(i << 1 | 1, change[i]);
        update[i] = false;
    }
}

// 整个范围的色板重置成 v 颜色的懒更新处理
public static void lazy(int i, int v) {
    color[i] = 1 << v;
    change[i] = v;
    update[i] = true;
}

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);
    }
    // 一开始所有色板都是 1 号颜色
    // 所以状态为 0...0010 = 2
    color[i] = 2;
    change[i] = 0;
    update[i] = false;
}

// 范围重置颜色
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 {
        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);
        }
        up(i);
    }
}

// 返回 l..r 范围上的颜色种类
public static int query(int l, int r, int n) {
    int status = query(l, r, 1, n, 1);
    return cntOnes(status);
}

// 返回 l..r 范围上的颜色状况
public static int query(int jobl, int jobr, int l, int r, int i) {
    if (jobl <= l && r <= jobr) {
        return color[i];
    }
    down(i);
    int mid = (l + r) >> 1;
    int status = 0;
    if (jobl <= mid) {
        status |= query(jobl, jobr, l, mid, i << 1);
    }
    if (jobr > mid) {
        status |= query(jobl, jobr, mid + 1, r, i << 1 | 1);
    }
    return status;
}

// 返回 n 的二进制中有几个 1
// 该实现讲解 031 的题目 6 讲过了
// 你也可以用更简单的写法
public static int cntOnes(int n) {
    n = (n & 0x55555555) + ((n >>> 1) & 0x55555555);
    n = (n & 0x33333333) + ((n >>> 2) & 0x33333333);
    n = (n & 0x0f0f0f0f) + ((n >>> 4) & 0x0f0f0f0f);
    n = (n & 0x00ff00ff) + ((n >>> 8) & 0x00ff00ff);
    n = (n & 0x0000ffff) + ((n >>> 16) & 0x0000ffff);
    return n;
```

```
    }

    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();
        in.nextToken();
        int m = (int) in.nval;
        build(1, n, 1);
        String op;
        for (int i = 1, a, b, c, tmp; i <= m; i++) {
            in.nextToken();
            op = in.sval;
            in.nextToken();
            a = (int) in.nval;
            in.nextToken();
            b = (int) in.nval;
            // 注意题目可能有 a>b 的情况需要交换
            if (a > b) {
                tmp = a;
                a = b;
                b = tmp;
            }
            if (op.equals("C")) {
                in.nextToken();
                c = (int) in.nval;
                update(a, b, c, 1, n, 1);
            } else {
                out.println(query(a, b, n));
            }
        }
        out.flush();
        out.close();
        br.close();
    }

}
```

==========

文件: Main.java

```
================================================
// 308. 二维区域和检索 - 可变
// 给你一个 2D 矩阵 matrix，请你完成两类查询：
// 1. 更新矩阵中某个单元格的值
// 2. 计算子矩形范围内元素的总和，该子矩阵的左上角为 (row1, col1) ，右下角为 (row2, col2)
// 实现 NumMatrix 类：
// NumMatrix(int[][] matrix) 给定整数矩阵 matrix 进行初始化
// void update(int row, int col, int val) 更新 matrix[row][col] 的值到 val
// int sumRegion(int row1, int col1, int row2, int col2) 返回子矩阵的总和
// 测试链接 : https://leetcode.cn/problems/range-sum-query-2d-mutable/
// 请同学们务必参考如下代码中关于输入、输出的处理
// 这是输入输出处理效率很高的写法
// 提交以下的 code，提交时请把类名改成"Main"，可以直接通过

import java.util.*;

public class Main {

    // 二维线段树实现
    static class NumMatrix {
        int[][] matrix;
        int[][] tree;
        int m, n;

        // 构造函数
        // 时间复杂度: O(m*n)
        // 空间复杂度: O(m*n)
        public NumMatrix(int[][] matrix) {
            if (matrix == null || matrix.length == 0 || matrix[0].length == 0) {
                return;
            }

            this.matrix = matrix;
            m = matrix.length;
            n = matrix[0].length;
            tree = new int[m * 2][n * 2];

            // 构建线段树
            buildTree();
        }

        // 构建二维线段树
```

```
// 时间复杂度: O(m*n)
private void buildTree() {
    // 先构建每行的一维线段树
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            tree[i + m][j + n] = matrix[i][j];
        }

        // 构建行的线段树
        for (int j = n - 1; j > 0; j--) {
            tree[i + m][j] = tree[i + m][j << 1] + tree[i + m][j << 1 | 1];
        }
    }

    // 构建列的线段树
    for (int i = m - 1; i > 0; i--) {
        for (int j = 0; j < 2 * n; j++) {
            tree[i][j] = tree[i << 1][j] + tree[i << 1 | 1][j];
        }
    }
}

// 更新矩阵中某个位置的值
// 时间复杂度: O(log m * log n)
public void update(int row, int col, int val) {
    // 计算差值
    int delta = val - matrix[row][col];
    matrix[row][col] = val;

    // 更新线段树
    for (int i = row + m; i > 0; i >>= 1) {
        for (int j = col + n; j > 0; j >>= 1) {
            tree[i][j] += delta;
        }
    }
}

// 查询子矩阵的总和
// 时间复杂度: O(log m * log n)
public int sumRegion(int row1, int col1, int row2, int col2) {
    int sum = 0;

    // 处理行范围
```

```
        for (int i = row1 + m; i <= row2 + m; ) {
            int r1 = i, r2 = i;

            // 找到完整的区间
            while (r1 > 0 && r1 % 2 == 0 && r2 + 1 <= row2 + m) {
                r1 >>= 1;
                r2 = (r2 + 1) >> 1;
            }

            // 处理列范围
            sum += sumRow(r1, col1, col2);

            // 移动到下一个区间
            if (r1 * 2 <= row2 + m) {
                i = r1 * 2 + 1;
            } else {
                break;
            }
        }

        return sum;
    }

    // 计算某一行范围内列的和
    // 时间复杂度: O(log n)
    private int sumRow(int row, int col1, int col2) {
        int sum = 0;
        for (int j = col1 + n, k = col2 + n + 1; j < k; j >>= 1, k >>= 1) {
            if ((j & 1) == 1) sum += tree[row][j++];
            if ((k & 1) == 1) sum += tree[row][--k];
        }
        return sum;
    }
}

// 测试方法
public static void main(String[] args) {
    // 测试用例:
    // matrix = [
    //   [3, 0, 1, 4, 2],
    //   [5, 6, 3, 2, 1],
    //   [1, 2, 0, 1, 5],
    //   [4, 1, 0, 1, 7],
```

```
        //   [1, 0, 3, 0, 5]
        // ]
        // sumRegion(2, 1, 4, 3) => 8
        // update(3, 2, 2)
        // sumRegion(2, 1, 4, 3) => 10

        int[][] matrix = {
            {3, 0, 1, 4, 2},
            {5, 6, 3, 2, 1},
            {1, 2, 0, 1, 5},
            {4, 1, 0, 1, 7},
            {1, 0, 3, 0, 5}
        };

        NumMatrix numMatrix = new NumMatrix(matrix);
        System.out.println(numMatrix.sumRegion(2, 1, 4, 3)); // 应该输出 8
        numMatrix.update(3, 2, 2);
        System.out.println(numMatrix.sumRegion(2, 1, 4, 3)); // 应该输出 10
    }
}
```

==========