

=====

文件夹: class086_SegmentTreeAndRelatedAlgorithms

=====

[Markdown 文件]

=====

文件: ADDITIONAL_PROBLEMS.md

=====

线段树补充题目清单

本文件整理了与 class113 中线段树相关的更多练习题目，来源于各大算法平台。

 按平台分类

LeetCode (力扣)

1. **LeetCode 218. 天际线问题** - <https://leetcode.cn/problems/the-skyline-problem/>
 - 类型: 线段树+扫描线
 - 难度: 困难
2. **LeetCode 307. 区域和检索 - 数组可修改** - <https://leetcode.cn/problems/range-sum-query-mutable/>
 - 类型: 线段树基础
 - 难度: 中等
3. **LeetCode 308. 二维区域和检索 - 可变** - <https://leetcode.cn/problems/range-sum-query-2d-mutable/>
 - 类型: 二维线段树
 - 难度: 困难
4. **LeetCode 315. 计算右侧小于当前元素的个数** - <https://leetcode.cn/problems/count-of-smaller-numbers-after-self/>
 - 类型: 线段树+离散化
 - 难度: 困难
5. **LeetCode 327. 区间和的个数** - <https://leetcode.cn/problems/count-of-range-sum/>
 - 类型: 线段树+前缀和
 - 难度: 困难
6. **LeetCode 493. 翻转对** - <https://leetcode.cn/problems/reverse-pairs/>
 - 类型: 线段树+离散化
 - 难度: 困难
7. **LeetCode 699. 掉落的方块** - <https://leetcode.cn/problems/falling-squares/>

- 类型: 线段树+坐标离散化
 - 难度: 困难
8. **LeetCode 715. Range 模块** - <https://leetcode.cn/problems/range-module/>
 - 类型: 线段树+区间合并
 - 难度: 困难

9. **LeetCode 729. 我的日程安排表 I** - <https://leetcode.cn/problems/my-calendar-i/>
 - 类型: 线段树区间查询
 - 难度: 中等

10. **LeetCode 731. 我的日程安排表 II** - <https://leetcode.cn/problems/my-calendar-ii/>
 - 类型: 线段树+最大值维护
 - 难度: 中等

11. **LeetCode 732. 我的日程安排表 III** - <https://leetcode.cn/problems/my-calendar-iii/>
 - 类型: 线段树+最大值维护
 - 难度: 困难

12. **LeetCode 850. 矩形面积 II** - <https://leetcode.cn/problems/rectangle-area-ii/>
 - 类型: 线段树+扫描线
 - 难度: 困难

13. **LeetCode 1157. 子数组中占绝大多数的元素** - <https://leetcode.cn/problems/online-majority-element-in-subarray/>
 - 类型: 线段树+二分查找
 - 难度: 困难

14. **LeetCode 1649. 通过指令创建有序数组** - <https://leetcode.cn/problems/create-sorted-array-through-instructions/>
 - 类型: 线段树+离散化
 - 难度: 困难

Codeforces

1. **339D - Xenia and Bit Operations** - <https://codeforces.com/problemset/problem/339/D>
 - 类型: 线段树基础
 - 难度: 1900
2. **380C - Sereja and Brackets** - <https://codeforces.com/problemset/problem/380/C>
 - 类型: 线段树维护括号匹配
 - 难度: 1700
3. **52C - Circular RMQ** - <https://codeforces.com/problemset/problem/52/C>

- 类型: 线段树+区间更新
 - 难度: 2200
4. **145E - Lucky Queries** - <https://codeforces.com/problemset/problem/145/E>
 - 类型: 线段树+懒标记
 - 难度: 2200

5. **242E - XOR on Segment** - <https://codeforces.com/problemset/problem/242/E>
 - 类型: 线段树+位运算
 - 难度: 2000

6. **438D - The Child and Sequence** - <https://codeforces.com/problemset/problem/438/D>
 - 类型: 线段树+区间取模
 - 难度: 2400

7. **446C - DZY Loves Fibonacci Numbers** - <https://codeforces.com/problemset/problem/446/C>
 - 类型: 线段树+斐波那契数列
 - 难度: 2700

8. **558E - A Simple Task** - <https://codeforces.com/problemset/problem/558/E>
 - 类型: 线段树+字符排序
 - 难度: 2300

9. **610E - Alphabet Permutations** - <https://codeforces.com/problemset/problem/610/E>
 - 类型: 线段树+字符串处理
 - 难度: 2400

10. **718C - Sasha and Array** - <https://codeforces.com/problemset/problem/718/C>
 - 类型: 线段树+矩阵乘法
 - 难度: 2400

洛谷 (Luogu)

1. **P3372 【模板】线段树 1** - <https://www.luogu.com.cn/problem/P3372>
 - 类型: 线段树基础模板
 - 难度: 普及+/提高-
2. **P3373 【模板】线段树 2** - <https://www.luogu.com.cn/problem/P3373>
 - 类型: 线段树+多种操作
 - 难度: 提高+/省选-
3. **P2572 [SDOI2010] 序列操作** - <https://www.luogu.com.cn/problem/P2572>
 - 类型: 线段树+多种操作
 - 难度: 省选/NOI-

4. **P1503 [SHOI2008] 洞穴勘测** - <https://www.luogu.com.cn/problem/P1503>
 - 类型: 线段树维护连通性
 - 难度: 提高+/省选-
5. **P2894 [USACO08FEB] Hotel G** - <https://www.luogu.com.cn/problem/P2894>
 - 类型: 线段树维护连续区间
 - 难度: 提高+/省选-
6. **P4514 上帝造题的七分钟** - <https://www.luogu.com.cn/problem/P4514>
 - 类型: 二维线段树
 - 难度: 省选/NOI-
7. **P4198 楼房重建** - <https://www.luogu.com.cn/problem/P4198>
 - 类型: 线段树维护斜率
 - 难度: 省选/NOI-
8. **P2429 制杖题** - <https://www.luogu.com.cn/problem/P2429>
 - 类型: 线段树+数学
 - 难度: 省选/NOI-
9. **P4588 [TJOI2018]数学计算** - <https://www.luogu.com.cn/problem/P4588>
 - 类型: 线段树维护乘积
 - 难度: 提高+/省选-
10. **P2184 贪婪大陆** - <https://www.luogu.com.cn/problem/P2184>
 - 类型: 线段树+前缀和
 - 难度: 提高+/省选-

HackerRank

1. **Array and simple queries** - <https://www.hackerrank.com/challenges/array-and-simple-queries/problem>
 - 类型: 线段树基础
 - 难度: Advanced
2. **Roy and Coin Boxes** - <https://www.hackerrank.com/challenges/roy-and-coin-boxes/problem>
 - 类型: 线段树+差分
 - 难度: Advanced
3. **Polynomial Division** - <https://www.hackerrank.com/challenges/polynomial-divison/problem>
 - 类型: 线段树+多项式
 - 难度: Expert

LintCode

1. **201. 线段树的构造** - <https://www.lintcode.com/problem/segment-tree-build/>
- 类型: 线段树基础
- 难度: 中等
2. **202. 线段树查询** - <https://www.lintcode.com/problem/segment-tree-query/>
- 类型: 线段树查询
- 难度: 中等
3. **203. 线段树修改** - <https://www.lintcode.com/problem/segment-tree-modify/>
- 类型: 线段树更新
- 难度: 中等
4. **206. 区间求和 I** - <https://www.lintcode.com/problem/interval-sum/>
- 类型: 线段树求和
- 难度: 中等
5. **207. 区间求和 II** - <https://www.lintcode.com/problem/interval-sum-ii/>
- 类型: 线段树+区间更新
- 难度: 困难

SPOJ

1. **GSS1 – Can you answer these queries I** - <https://www.spoj.com/problems/GSS1/>
- 类型: 线段树最大子段和
- 难度: 中等
2. **GSS3 – Can you answer these queries III** - <https://www.spoj.com/problems/GSS3/>
- 类型: 线段树最大子段和+单点更新
- 难度: 中等
3. **GSS4 – Can you answer these queries IV** - <https://www.spoj.com/problems/GSS4/>
- 类型: 线段树+区间开方
- 难度: 困难
4. **GSS5 – Can you answer these queries V** - <https://www.spoj.com/problems/GSS5/>
- 类型: 线段树+区间查询
- 难度: 困难
5. **GSS6 – Can you answer these queries VI** - <https://www.spoj.com/problems/GSS6/>
- 类型: 平衡树+线段树
- 难度: 困难
6. **GSS7 – Can you answer these queries VII** - <https://www.spoj.com/problems/GSS7/>

- 类型: 树链剖分+线段树
- 难度: 困难

HDU

1. **HDU 1166 敌兵布阵** - <http://acm.hdu.edu.cn/showproblem.php?pid=1166>
 - 类型: 线段树基础
 - 难度: 简单
2. **HDU 1754 I Hate It** - <http://acm.hdu.edu.cn/showproblem.php?pid=1754>
 - 类型: 线段树维护最值
 - 难度: 简单
3. **HDU 1698 Just a Hook** - <http://acm.hdu.edu.cn/showproblem.php?pid=1698>
 - 类型: 线段树+区间更新
 - 难度: 中等
4. **HDU 4528 小明系列故事——捉迷藏** - <http://acm.hdu.edu.cn/showproblem.php?pid=4528>
 - 类型: 线段树+几何
 - 难度: 困难

POJ

1. **POJ 3468 A Simple Problem with Integers** - <http://poj.org/problem?id=3468>
 - 类型: 线段树+区间加法
 - 难度: 中等
2. **POJ 2777 Count Color** - <http://poj.org/problem?id=2777>
 - 类型: 线段树+位运算
 - 难度: 中等
3. **POJ 2528 Mayor's posters** - <http://poj.org/problem?id=2528>
 - 类型: 线段树+离散化
 - 难度: 中等
4. **POJ 3264 Balanced Lineup** - <http://poj.org/problem?id=3264>
 - 类型: 线段树维护最值
 - 难度: 简单

牛客网 (Nowcoder)

1. **NC16534 线段树练习** - <https://ac.nowcoder.com/acm/problem/16534>
 - 类型: 线段树基础
 - 难度: 中等
2. **NC16535 线段树练习 2** - <https://ac.nowcoder.com/acm/problem/16535>

- 类型: 线段树+区间更新
 - 难度: 中等
3. **NC16536 线段树练习 3** - <https://ac.nowcoder.com/acm/problem/16536>
 - 类型: 线段树+区间更新
 - 难度: 困难

4. **NC24970 线段树练习一** - <https://ac.nowcoder.com/acm/problem/24970>
 - 类型: 线段树基础区间操作
 - 难度: 中等

5. **NC24971 线段树练习二** - <https://ac.nowcoder.com/acm/problem/24971>
 - 类型: 线段树+区间更新
 - 难度: 中等

AtCoder

 1. **ABC165 F - LIS on Tree** - https://atcoder.jp/contests/abc165/tasks/abc165_f
 - 类型: 线段树+树上 DP
 - 难度: 2286
 2. **ABC185 F - Range Xor Query** - https://atcoder.jp/contests/abc185/tasks/abc185_f
 - 类型: 线段树基础
 - 难度: 1186
 3. **ARC075 D - Widespread** - https://atcoder.jp/contests/arc075/tasks/arc075_d
 - 类型: 线段树+二分
 - 难度: 1765
 4. **ABC280 E - Critical Hit** - https://atcoder.jp/contests/abc280/tasks/abc280_e
 - 类型: 概率 DP+线段树优化
 - 难度: 中等
 5. **ABC273 E - Least Elements** - https://atcoder.jp/contests/abc273/tasks/abc273_e
 - 类型: 线段树合并
 - 难度: 中等

CodeChef

 1. **FLIPCOIN - Flipping Coins** - <https://www.codechef.com/problems/FLIPCOIN>
 - 类型: 线段树+概率
 - 难度: Medium
 2. **GSS1 - Can you answer these queries I** - <https://www.codechef.com/problems/GSS1>
 - 类型: 线段树最大子段和

- 难度: Medium

3. **CHEFPRAD - Chef and Pairs** - <https://www.codechef.com/problems/CHEFPRAD>

- 类型: 点分治+线段树合并

- 难度: 困难

USACO

1. **Balanced Lineup** - <http://www.usaco.org/index.php?page=viewproblem2&cpid=62>

- 类型: 线段树维护最值

- 难度: Silver

2. **Hotel** - <http://www.usaco.org/index.php?page=viewproblem2&cpid=63>

- 类型: 线段树维护连续区间

- 难度: Gold

3. **Wormhole Sort** - <http://www.usaco.org/index.php?page=viewproblem2&cpid=993>

- 类型: 线段树合并+并查集

- 难度: Gold

4. **Milk Visits** - <http://www.usaco.org/index.php?page=viewproblem2&cpid=1194>

- 类型: 树形 DP+线段树合并

- 难度: Gold

杭电 OJ (HDU)

1. **HDU 1540 Tunnel Warfare** - <http://acm.hdu.edu.cn/showproblem.php?pid=1540>

- 类型: 线段树维护连续区间

- 难度: 中等

2. **HDU 1542 Atlantis** - <http://acm.hdu.edu.cn/showproblem.php?pid=1542>

- 类型: 线段树+扫描线

- 难度: 困难

3. **HDU 1828 Picture** - <http://acm.hdu.edu.cn/showproblem.php?pid=1828>

- 类型: 线段树+扫描线

- 难度: 困难

4. **HDU 4348 To the moon** - <http://acm.hdu.edu.cn/showproblem.php?pid=4348>

- 类型: 可持久化线段树+区间更新

- 难度: 困难

洛谷 (Luogu) - 进阶题目

1. **P2468 [SDOI2010] 粟粟的书架** - <https://www.luogu.com.cn/problem/P2468>

- 类型: 主席树+二分

- 难度: 省选/NOI-
2. **P2617 Dynamic Rankings** - <https://www.luogu.com.cn/problem/P2617>
 - 类型: 树套树
 - 难度: 省选/NOI-

3. **P3313 [SDOI2014] 旅行** - <https://www.luogu.com.cn/problem/P3313>
 - 类型: 树链剖分+线段树
 - 难度: 省选/NOI-

4. **P3759 [TJOI2017] 不勤劳的图书管理员** - <https://www.luogu.com.cn/problem/P3759>
 - 类型: 线段树+离散化
 - 难度: 省选/NOI-

5. **P4198 楼房重建** - <https://www.luogu.com.cn/problem/P4198>
 - 类型: 线段树维护斜率
 - 难度: 省选/NOI-

其他平台

 1. **MarsCode 线段树专题** - <https://www.marscode.cn/topic/segment-tree>
 - 类型: 线段树综合练习
 - 难度: 多种难度
 2. **UVa OJ 11990 - Dynamic Inversion** -
https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&page=show_problem&problem=3141
 - 类型: 动态逆序对问题
 - 难度: 困难
 3. **TimusOJ 1521 - War Games 2** - <http://acm.timus.ru/problem.aspx?space=1&num=1521>
 - 类型: 线段树+约瑟夫问题
 - 难度: 中等
 4. **AizuOJ 2450 - Do use segment tree** - <http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=2450>
 - 类型: 树链剖分+线段树
 - 难度: 困难
 5. **Comet OJ 线段树专题** - <https://cometoj.com/contest/tag/segment-tree>
 - 类型: 线段树综合练习
 - 难度: 多种难度
 6. **ZOJ 2112 Dynamic Rankings** -
<http://acm.zju.edu.cn/onlinejudge/showProblem.do?problemCode=2112>

- 类型：动态区间第 k 小

- 难度：困难

7. **HackerEarth Segment Tree Practice** - <https://www.hackerearth.com/practice/data-structures/advanced-data-structures/segment-trees/practice-problems/>

- 类型：线段树综合练习

- 难度：多种难度

8. **计蒜客 线段树专题** - <https://www.jisuanke.com/course/tag/segment-tree>

- 类型：线段树综合练习

- 难度：多种难度

9. **各大高校 OJ 线段树题目**

- 北京大学 POJ：线段树相关题目约 50+道

- 浙江大学 ZOJ：线段树相关题目约 30+道

- 杭州电子科技大学 HDU：线段树相关题目约 40+道

- 上海交通大学 SJTU：线段树相关题目约 20+道

剑指 Offer 相关

1. **剑指 Offer 51. 数组中的逆序对** - <https://leetcode.cn/problems/shu-zu-zhong-de-ni-xu-dui-lcof/>

- 类型：线段树+离散化

- 难度：困难

2. **剑指 Offer 59 - I. 滑动窗口的最大值** - <https://leetcode.cn/problems/hua-dong-chuang-kou-de-zui-da-zhi-lcof/>

- 类型：线段树维护最值

- 难度：中等

3. **剑指 Offer 59 - II. 队列的最大值** - <https://leetcode.cn/problems/dui-lie-de-zui-da-zhi-lcof/>

- 类型：线段树维护最值

- 难度：中等

💡 线段树常见变种

1. 动态开点线段树

- 适用场景：数据范围很大，但实际使用较少的情况

- 典型题目：Codeforces 915E

2. 可持久化线段树（主席树）

- 适用场景：需要保存历史版本的信息

- 典型题目：POJ 2104

3. 扫描线 + 线段树

- 适用场景: 平面几何问题, 如矩形面积并
- 典型题目: POJ 1151

4. 树链剖分 + 线段树

- 适用场景: 树上路径操作
- 典型题目: 洛谷 P3384

5. 线段树合并

- 适用场景: 动态维护多个集合的信息
- 典型题目: Codeforces 600E

🎓 学习建议

1. **基础阶段**:

- 熟练掌握线段树的基本操作（建树、单点更新、区间查询）
- 练习简单的区间求和、区间最值问题

2. **进阶阶段**:

- 学习懒标记技术, 掌握区间更新操作
- 练习复杂的节点信息维护

3. **高手阶段**:

- 学习线段树的各种变种和高级应用
- 掌握线段树与其他算法的结合使用

📚 参考资料

1. 《算法竞赛进阶指南》 - 李煜东
2. 《挑战程序设计竞赛》 - 秋叶拓哉等
3. TopCoder 数据结构教程
4. Codeforces Educational Round 相关讲解

=====

文件: ADDITIONAL_SEGMENT_TREE_PROBLEMS.md

=====

线段树补充题目清单

本文件整理了与 class113 中线段树相关的更多练习题目, 来源于各大算法平台。

📚 按平台分类

LeetCode (力扣)

1. **LeetCode 218. 天际线问题** - <https://leetcode.cn/problems/the-skyline-problem/>

- 类型: 线段树+扫描线
- 难度: 困难

- 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。

2. **LeetCode 307. 区域和检索 - 数组可修改** - <https://leetcode.cn/problems/range-sum-query-mutable/>

- 类型: 线段树基础
- 难度: 中等
- 题目描述: 实现支持更新和区间求和操作的数据结构。

3. **LeetCode 308. 二维区域和检索 - 可变** - <https://leetcode.cn/problems/range-sum-query-2d-mutable/>

- 类型: 二维线段树
- 难度: 困难
- 题目描述: 实现支持更新和二维区间求和操作的数据结构。

4. **LeetCode 315. 计算右侧小于当前元素的个数** - <https://leetcode.cn/problems/count-of-smaller-numbers-after-self/>

- 类型: 线段树+离散化
- 难度: 困难

- 题目描述: 给定一个整数数组 `nums`, 返回一个新数组 `counts`, 其中 `counts[i]` 是 `nums[i]` 右侧小于 `nums[i]` 的元素的数量。

5. **LeetCode 327. 区间和的个数** - <https://leetcode.cn/problems/count-of-range-sum/>

- 类型: 线段树+前缀和
- 难度: 困难

- 题目描述: 给定一个整数数组 `nums` 以及两个整数 `lower` 和 `upper`, 求值位于范围 `[lower, upper]` 之间的区间和的个数。

6. **LeetCode 493. 翻转对** - <https://leetcode.cn/problems/reverse-pairs/>

- 类型: 线段树+离散化
- 难度: 困难

- 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, 需要返回给定数组中的重要翻转对的总数。

7. **LeetCode 699. 掉落的方块** - <https://leetcode.cn/problems/falling-squares/>

- 类型: 线段树+坐标离散化
- 难度: 困难

- 题目描述: 在无限长的数轴(坐标轴)上, 我们根据给定的顺序放置“方块”, 计算每次放置后堆叠的最大高度。

8. **LeetCode 715. Range 模块** - <https://leetcode.cn/problems/range-module/>
- 类型: 线段树+区间合并
- 难度: 困难
- 题目描述: 实现一个 RangeModule 类, 用于跟踪数字范围并支持添加、查询和移除范围操作。
9. **LeetCode 729. 我的日程安排表 I** - <https://leetcode.cn/problems/my-calendar-i/>
- 类型: 线段树区间查询
- 难度: 中等
- 题目描述: 实现一个 MyCalendar 类来存放你的日程安排, 如果要添加的日程安排不会造成重复预订, 则可以存储这个新的日程安排。
10. **LeetCode 731. 我的日程安排表 II** - <https://leetcode.cn/problems/my-calendar-ii/>
- 类型: 线段树+最大值维护
- 难度: 中等
- 题目描述: 实现一个 MyCalendarTwo 类来存放你的日程安排, 如果要添加的时间不会导致三重预订时, 则可以存储这个新的日程安排。
11. **LeetCode 732. 我的日程安排表 III** - <https://leetcode.cn/problems/my-calendar-iii/>
- 类型: 线段树+最大值维护
- 难度: 困难
- 题目描述: 实现一个 MyCalendarThree 类来存放你的日程安排, 可以存放相同时间的多个日程安排。
12. **LeetCode 850. 矩形面积 II** - <https://leetcode.cn/problems/rectangle-area-ii/>
- 类型: 线段树+扫描线
- 难度: 困难
- 题目描述: 我们给出了一个(轴对齐的)二维矩形列表 rectangles, 计算并返回所有矩形所覆盖的总面积。
13. **LeetCode 1157. 子数组中占绝大多数的元素** - <https://leetcode.cn/problems/online-majority-element-in-subarray/>
- 类型: 线段树+二分查找
- 难度: 困难
- 题目描述: 实现一个数据结构, 支持查询指定子数组中出现次数超过阈值的元素。
14. **LeetCode 1649. 通过指令创建有序数组** - <https://leetcode.cn/problems/create-sorted-array-through-instructions/>
- 类型: 线段树+离散化
- 难度: 困难
- 题目描述: 通过一系列指令创建一个有序数组, 计算插入每个元素的代价。

Codeforces

1. **339D - Xenia and Bit Operations** - <https://codeforces.com/problemset/problem/339/D>

- 类型: 线段树基础
 - 难度: 1900
 - 题目描述: 给定一个数组, 交替进行 OR 和 XOR 操作, 支持单点更新和查询根节点的值。
2. **380C – Sereja and Brackets** – <https://codeforces.com/problemset/problem/380/C>
 - 类型: 线段树维护括号匹配
 - 难度: 1700
 - 题目描述: 给定一个括号序列, 查询区间内能匹配的括号对数。

3. **52C – Circular RMQ** – <https://codeforces.com/problemset/problem/52/C>
 - 类型: 线段树+区间更新
 - 难度: 2200
 - 题目描述: 环形数组上的区间最小值查询和区间加法操作。

4. **145E – Lucky Queries** – <https://codeforces.com/problemset/problem/145/E>
 - 类型: 线段树+懒标记
 - 难度: 2200
 - 题目描述: 维护一个由 4 和 7 组成的字符串, 支持区间翻转和查询最长非递减子序列。

5. **242E – XOR on Segment** – <https://codeforces.com/problemset/problem/242/E>
 - 类型: 线段树+位运算
 - 难度: 2000
 - 题目描述: 支持区间异或和区间求和操作。

6. **438D – The Child and Sequence** – <https://codeforces.com/problemset/problem/438/D>
 - 类型: 线段树+区间取模
 - 难度: 2400
 - 题目描述: 支持区间取模、单点更新和区间最大值查询。

7. **446C – DZY Loves Fibonacci Numbers** – <https://codeforces.com/problemset/problem/446/C>
 - 类型: 线段树+斐波那契数列
 - 难度: 2700
 - 题目描述: 支持区间加斐波那契数列和区间求和操作。

8. **558E – A Simple Task** – <https://codeforces.com/problemset/problem/558/E>
 - 类型: 线段树+字符排序
 - 难度: 2300
 - 题目描述: 支持区间按升序或降序排序操作。

9. **610E – Alphabet Permutations** – <https://codeforces.com/problemset/problem/610/E>
 - 类型: 线段树+字符串处理
 - 难度: 2400
 - 题目描述: 维护字符串的逆序对数量, 支持单点更新。

10. **718C – Sasha and Array** – <https://codeforces.com/problemset/problem/718/C>
 - 类型: 线段树+矩阵乘法
 - 难度: 2400
 - 题目描述: 支持区间加法和区间斐波那契数列求和操作。

11. **915E – Physical Education Lessons** – <https://codeforces.com/problemset/problem/915/E>
 - 类型: 动态开点线段树
 - 难度: 2200
 - 题目描述: 维护一个区间, 支持区间设置为 0 或 1, 查询区间和。

洛谷 (Luogu)

1. **P3372 【模板】线段树 1** – <https://www.luogu.com.cn/problem/P3372>
 - 类型: 线段树基础模板
 - 难度: 普及+/提高-
 - 题目描述: 实现支持区间加法和区间求和的线段树。

2. **P3373 【模板】线段树 2** – <https://www.luogu.com.cn/problem/P3373>
 - 类型: 线段树+多种操作
 - 难度: 提高+/省选-
 - 题目描述: 实现支持区间加法、区间乘法和区间求和的线段树。

3. **P2572 [SDOI2010] 序列操作** – <https://www.luogu.com.cn/problem/P2572>
 - 类型: 线段树+多种操作
 - 难度: 省选/NOI-
 - 题目描述: 支持区间置 0、置 1、取反、查询 1 的个数和查询连续 1 的最长长度。

4. **P1503 [SHOI2008] 洞穴勘测** – <https://www.luogu.com.cn/problem/P1503>
 - 类型: 线段树维护连通性
 - 难度: 提高+/省选-
 - 题目描述: 维护连续 1 的前缀和后缀长度, 支持摧毁和恢复操作。

5. **P2894 [USACO08FEB] Hotel G** – <https://www.luogu.com.cn/problem/P2894>
 - 类型: 线段树维护连续区间
 - 难度: 提高+/省选-
 - 题目描述: 维护连续空房间信息, 支持查找最左连续空房间和清空房间操作。

6. **P4514 上帝造题的七分钟** – <https://www.luogu.com.cn/problem/P4514>
 - 类型: 二维线段树
 - 难度: 省选/NOI-
 - 题目描述: 实现二维区间加法和二维区间求和的线段树。

7. **P4198 楼房重建** – <https://www.luogu.com.cn/problem/P4198>

- 类型: 线段树维护斜率
 - 难度: 省选/NOI-
 - 题目描述: 维护区间前缀最大值个数, 用于解决楼房可见性问题。
8. **P2468 [SDOI2010] 粟粟的书架** - <https://www.luogu.com.cn/problem/P2468>
 - 类型: 主席树+二分
 - 难度: 省选/NOI-
 - 题目描述: 二维前缀和+主席树, 查询子矩阵中前 k 大元素的和。

9. **P2617 Dynamic Rankings** - <https://www.luogu.com.cn/problem/P2617>
 - 类型: 树套树
 - 难度: 省选/NOI-
 - 题目描述: 支持单点修改和区间第 k 小查询。

10. **P3313 [SDOI2014] 旅行** - <https://www.luogu.com.cn/problem/P3313>
 - 类型: 树链剖分+线段树
 - 难度: 省选/NOI-
 - 题目描述: 树上路径操作, 支持路径最大值查询和路径求和。

HackerRank

1. **Array and simple queries** - <https://www.hackerrank.com/challenges/array-and-simple-queries/problem>
 - 类型: 线段树基础
 - 难度: Advanced
 - 题目描述: 支持区间移动操作和查询。
2. **Roy and Coin Boxes** - <https://www.hackerrank.com/challenges/roy-and-coin-boxes/problem>
 - 类型: 线段树+差分
 - 难度: Advanced
 - 题目描述: 维护盒子中的硬币数量, 支持区间加法和查询。
3. **Polynomial Division** - <https://www.hackerrank.com/challenges/polynomial-divison/problem>
 - 类型: 线段树+多项式
 - 难度: Expert
 - 题目描述: 多项式除法问题。

LintCode (炼码)

1. **201. 线段树的构造** - <https://www.lintcode.com/problem/segment-tree-build/description>
 - 类型: 线段树基础
 - 难度: 中等
 - 题目描述: 构造线段树。
2. **202. 线段树查询** - <https://www.lintcode.com/problem/segment-tree-query/description>

- 类型: 线段树查询
- 难度: 中等
- 题目描述: 查询线段树区间最大值。

3. **203. 线段树修改** - <https://www.lintcode.com/problem/segment-tree-modify/>
description

- 类型: 线段树更新
- 难度: 中等
- 题目描述: 单点更新线段树。

4. **206. 区间求和 I** - <https://www.lintcode.com/problem/interval-sum/>
description

- 类型: 线段树求和
- 难度: 中等
- 题目描述: 实现区间求和操作。

5. **207. 区间求和 II** - <https://www.lintcode.com/problem/interval-sum-ii/>
description

- 类型: 线段树+区间更新
- 难度: 困难
- 题目描述: 实现区间加法和区间求和操作。

SPOJ

1. **GSS1 – Can you answer these queries I** - <https://www.spoj.com/problems/GSS1/>

- 类型: 线段树最大子段和
- 难度: 中等
- 题目描述: 查询区间最大子段和。

2. **GSS3 – Can you answer these queries III** - <https://www.spoj.com/problems/GSS3/>

- 类型: 线段树最大子段和+单点更新
- 难度: 中等
- 题目描述: 支持单点更新和查询区间最大子段和。

3. **GSS4 – Can you answer these queries IV** - <https://www.spoj.com/problems/GSS4/>

- 类型: 线段树+区间开方
- 难度: 困难
- 题目描述: 支持区间开方和区间求和操作。

4. **GSS5 – Can you answer these queries V** - <https://www.spoj.com/problems/GSS5/>

- 类型: 线段树+区间查询
- 难度: 困难
- 题目描述: 查询指定范围内的最大子段和。

5. **GSS6 – Can you answer these queries VI** - <https://www.spoj.com/problems/GSS6/>

- 类型: 平衡树+线段树
- 难度: 困难

- 题目描述：支持插入、删除、修改和查询最大子段和。

6. **GSS7 – Can you answer these queries VII** – <https://www.spoj.com/problems/GSS7/>

- 类型：树链剖分+线段树

- 难度：困难

- 题目描述：树上路径最大子段和查询。

HDU

1. **HDU 1166 敌兵布阵** – <http://acm.hdu.edu.cn/showproblem.php?pid=1166>

- 类型：线段树基础

- 难度：简单

- 题目描述：支持单点更新和区间求和操作。

2. **HDU 1754 I Hate It** – <http://acm.hdu.edu.cn/showproblem.php?pid=1754>

- 类型：线段树维护最值

- 难度：简单

- 题目描述：支持单点更新和区间最大值查询。

3. **HDU 1698 Just a Hook** – <http://acm.hdu.edu.cn/showproblem.php?pid=1698>

- 类型：线段树+区间更新

- 难度：中等

- 题目描述：支持区间设置为指定值和区间求和操作。

4. **HDU 2795 Billboard** – <http://acm.hdu.edu.cn/showproblem.php?pid=2795>

- 类型：线段树+贪心

- 难度：中等

- 题目描述：维护公告板，支持查询最上可行位置。

POJ

1. **POJ 3468 A Simple Problem with Integers** – <http://poj.org/problem?id=3468>

- 类型：线段树+区间加法

- 难度：中等

- 题目描述：支持区间加法和区间求和操作。

2. **POJ 2777 Count Color** – <http://poj.org/problem?id=2777>

- 类型：线段树+位运算

- 难度：中等

- 题目描述：支持区间染色和查询区间颜色种类数。

3. **POJ 2528 Mayor' s posters** – <http://poj.org/problem?id=2528>

- 类型：线段树+离散化

- 难度：中等

- 题目描述：区间覆盖问题，查询可见海报数。

4. **POJ 3264 Balanced Lineup** - <http://poj.org/problem?id=3264>

- 类型: 线段树维护最值
- 难度: 简单
- 题目描述: 查询区间最大值和最小值的差。

牛客网 (Nowcoder)

1. **NC16534 线段树练习** - <https://ac.nowcoder.com/acm/problem/16534>

- 类型: 线段树基础
- 难度: 中等
- 题目描述: 基础线段树操作练习。

2. **NC16535 线段树练习 2** - <https://ac.nowcoder.com/acm/problem/16535>

- 类型: 线段树+区间更新
- 难度: 中等
- 题目描述: 区间更新和区间查询练习。

3. **NC16536 线段树练习 3** - <https://ac.nowcoder.com/acm/problem/16536>

- 类型: 线段树+区间更新
- 难度: 困难
- 题目描述: 复杂区间更新操作练习。

AtCoder

1. **ABC185 F - Range Xor Query** - https://atcoder.jp/contests/abc185/tasks/abc185_f

- 类型: 线段树+异或
- 难度: 1186
- 题目描述: 支持单点异或更新和区间异或查询。

2. **ARC075 D - Widespread** - https://atcoder.jp/contests/arc075/tasks/arc075_d

- 类型: 线段树+二分
- 难度: 1765
- 题目描述: 使用二分查找和线段树优化的组合问题。

CodeChef

1. **FLIPCOIN - Flipping Coins** - <https://www.codechef.com/problems/FLIPCOIN>

- 类型: 线段树+概率
- 难度: Medium
- 题目描述: 维护硬币翻转状态, 支持区间翻转和查询正面朝上的期望数量。

USACO

1. **Balanced Lineup** - <http://www.usaco.org/index.php?page=viewproblem2&cpid=62>

- 类型: 线段树维护最值
- 难度: Silver

- 题目描述：查询区间最大值和最小值的差。

2. **Hotel** - <http://www.usaco.org/index.php?page=viewproblem2&cpid=63>

- 类型：线段树维护连续区间

- 难度：Gold

- 题目描述：维护连续空房间信息。

🌟 线段树常见变种

1. 动态开点线段树

- 适用场景：数据范围很大，但实际使用较少的情况

- 典型题目：Codeforces 915E

2. 可持久化线段树（主席树）

- 适用场景：需要保存历史版本的信息

- 典型题目：洛谷 P3834、P3919

3. 扫描线 + 线段树

- 适用场景：平面几何问题，如矩形面积并

- 典型题目：LeetCode 218、850

4. 树链剖分 + 线段树

- 适用场景：树上路径操作

- 典型题目：洛谷 P3384、SPOJ GSS7

5. 线段树合并

- 适用场景：动态维护多个集合的信息

- 典型题目：Codeforces 600E

6. 吉司机线段树

- 适用场景：区间最值操作与历史最值

- 典型题目：SPOJ GSS 系列

🎓 学习建议

1. **基础阶段**：

- 熟练掌握线段树的基本操作（建树、单点更新、区间查询）

- 练习简单的区间求和、区间最值问题

2. **进阶阶段**：

- 学习懒标记技术，掌握区间更新操作

- 练习复杂的节点信息维护

3. **高手阶段**:

- 学习线段树的各种变种和高级应用
- 掌握线段树与其他算法的结合使用

📚 参考资料

1. 《算法竞赛进阶指南》 - 李煜东
2. 《挑战程序设计竞赛》 - 秋叶拓哉等
3. TopCoder 数据结构教程
4. Codeforces Educational Round 相关讲解

=====

文件: README.md

=====

线段树 (Segment Tree) 专题详解

💡 核心概念

线段树是一种基于分治思想的二叉树数据结构，主要用于解决区间查询和区间更新问题。每个节点代表一个区间，可以高效地支持区间操作。

特点

- 时间复杂度: 构建 $O(n)$, 单次查询/更新 $O(\log n)$
- 空间复杂度: $O(4n)$
- 适用于需要频繁进行区间操作的场景

📚 本目录题目详解

1. Code01_SequenceOperation.java - 序列操作

- **题目来源**: 洛谷 P2572 [SDOI2010] 序列操作
- **题目链接**: <https://www.luogu.com.cn/problem/P2572>
- **题目大意**:

给定一个长度为 n 的 01 序列，支持 5 种操作：

1. 操作 0 1 r: 将区间 $[1, r]$ 全部置为 0
2. 操作 1 1 r: 将区间 $[1, r]$ 全部置为 1
3. 操作 2 1 r: 将区间 $[1, r]$ 全部取反
4. 操作 3 1 r: 查询区间 $[1, r]$ 中 1 的个数
5. 操作 4 1 r: 查询区间 $[1, r]$ 中连续 1 的最长长度

- **解题思路**:

使用线段树维护每个区间的信息：

- $\text{sum}[i]$: 区间内 1 的个数

- `len0[i]/len1[i]`: 区间内连续 0/1 的最长子串长度
- `pre0[i]/pre1[i]`: 区间内连续 0/1 的最长前缀长度
- `suf0[i]/suf1[i]`: 区间内连续 0/1 的最长后缀长度
- `change[i]`: 懒标记，记录区间被置为的值
- `update[i]`: 懒标记，记录区间是否有更新操作
- `reverse[i]`: 懒标记，记录区间是否有翻转操作

- ****关键技术点**:**
 - 懒标记的下传顺序: 先处理 `update` 再处理 `reverse`
 - 区间合并操作: 需要考虑左右子区间连接处的情况
 - 多重懒标记的处理

- ****时间复杂度**:** $O(m \log n)$, 其中 m 为操作次数
- ****空间复杂度**:** $O(n)$

2. Code02_LongestAlternateSubstring.java – 最长 LR 交替子串

 - **题目来源**:** 洛谷 P6492 [COCI2010-2011#6] STEP
 - **题目链接**:** <https://www.luogu.com.cn/problem/P6492>
 - **题目大意**:**
给定一个长度为 n 的字符串，初始全为' L'，每次操作翻转一个位置的字符，求每次操作后最长的 LR 交替子串长度（如 LRLR 或 RLRL）。

 - **解题思路**:**
使用线段树维护每个区间的最长交替子串长度，以及前缀和后缀的最长交替长度。

 - `len[i]`: 区间内最长交替子串长度
 - `pre[i]`: 区间内最长交替前缀长度
 - `suf[i]`: 区间内最长交替后缀长度
 - `arr[i]`: 记录位置 i 的字符（0 表示 L, 1 表示 R）

 - **关键技术点**:**
 - 区间合并时需要判断中间连接处是否可以连接（相邻字符不同）
 - 单点更新时需要重新计算区间信息

 - **时间复杂度**:** $O(q \log n)$, 其中 q 为操作次数
 - **空间复杂度**:** $O(n)$

- ### 3. Code03_TunnelWarfare.java – 地道相连的房子
- **题目来源**:** 洛谷 P1503 [SHOI2008] 洞穴勘测
 - **题目链接**:** <https://www.luogu.com.cn/problem/P1503>
 - **题目大意**:**
有 n 个房子排成一排，相邻房子有地道连接。支持三种操作：
 1. D x : 摧毁 x 号房子及其相邻地道
 2. R: 恢复上次摧毁的房子及其相邻地道

3. Q x: 查询 x 号房子能到达的连续房子数量

- **解题思路**:

使用线段树维护每个区间的连续 1 的前缀和后缀长度，其中 1 表示房子未被摧毁。

- pre[i]: 区间内连续 1 的前缀长度
- suf[i]: 区间内连续 1 的后缀长度
- 使用栈记录摧毁的房子，支持恢复操作

- **关键技术点**:

- 查询操作需要根据位置在区间中的位置进行不同处理
- 区间合并时考虑跨区间的情况

- **时间复杂度**: $O(m \log n)$ ，其中 m 为操作次数

- **空间复杂度**: $O(n)$

4. Code04_Hotel. java - 旅馆

- **题目来源**: 洛谷 P2894 [USACO08FEB] Hotel G

- **题目链接**: <https://www.luogu.com.cn/problem/P2894>

- **题目大意**:

有 n 个房间，初始都为空房。支持两种操作：

1. 1 x: 找到最左边的长度至少为 x 的连续空房间，返回起始位置并入住
2. 2 x y: 将从 x 号房间开始的 y 个房间清空

- **解题思路**:

使用线段树维护每个区间的连续空房间信息：

- len[i]: 区间内最长连续空房间长度
- pre[i]: 区间内最长连续空房间前缀长度
- suf[i]: 区间内最长连续空房间后缀长度
- change[i]: 懒标记，记录区间被设置的值（0 表示空房，1 表示有人）
- update[i]: 懒标记，记录区间是否有更新操作

- **关键技术点**:

- 查询最左边满足条件的区间需要特殊处理
- 区间合并时需要考虑左右子区间的连接情况

- **时间复杂度**: $O(m \log n)$ ，其中 m 为操作次数

- **空间复杂度**: $O(n)$

5. Code05_TheSkylineProblem. java - 天际线问题

- **题目来源**: LeetCode 218. 天际线问题

- **题目链接**: <https://leetcode.cn/problems/the-skyline-problem/>

- **题目大意**:

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。

- **解题思路:**

使用扫描线算法结合优先队列来解决这个问题：

- 将所有建筑物的左右边界作为事件点处理
- 对事件点按 x 坐标排序
- 使用优先队列维护当前活跃建筑物的高度
- 遍历事件点，更新天际线关键点

- **关键技术点:**

- 扫描线算法：从左到右扫描所有建筑物的边界
- 优先队列：维护当前活跃建筑物的高度信息
- 事件处理：处理建筑物的进入和离开事件

- **时间复杂度:** $O(n \log n)$ ，其中 n 是建筑物数量

- **空间复杂度:** $O(n)$

6. Code06_RangeSumQueryMutable. java - 区域和检索 - 数组可修改

- **题目来源:** LeetCode 307. 区域和检索 - 数组可修改

- **题目链接:** <https://leetcode.cn/problems/range-sum-query-mutable/>

- **题目大意:**

实现支持更新和区间求和操作的数据结构。

- **解题思路:**

使用线段树来维护区间和，支持单点更新和区间查询操作：

- 线段树构建：构建支持区间求和的线段树
- 单点更新：支持更新数组中某个位置的值
- 区间查询：支持查询任意区间的元素和

- **关键技术点:**

- 线段树构建与维护
- 单点更新与区间查询的实现
- 递归与边界条件处理

- **时间复杂度:**

- 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:** $O(n)$

7. Code07_CountSmallerNumbersAfterSelf. java - 计算右侧小于当前元素的个数

- **题目来源:** LeetCode 315. 计算右侧小于当前元素的个数

- **题目链接:** <https://leetcode.cn/problems/count-of-smaller-numbers-after-self/>

- ****题目大意**:**

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**:**

使用离散化+线段树的方法：

- 离散化：将数组中的值映射到连续的整数范围
- 线段树：维护每个值的出现次数，支持单点更新和前缀和查询
- 逆序遍历：从右往左处理数组元素，确保查询的是右侧元素

- ****关键技术点**:**

- 离散化处理
- 线段树维护元素出现次数
- 逆序遍历与前缀和查询

- ****时间复杂度**:** $O(n \log n)$

- ****空间复杂度**:** $O(n)$

8. Code08_SegmentTreeTemplate1.java – 线段树模板 1

- ****题目来源**:** 洛谷 P3372 【模板】线段树 1

- ****题目链接**:** <https://www.luogu.com.cn/problem/P3372>

- ****题目大意**:**

支持区间加法和区间求和操作。

- ****解题思路**:**

使用带懒标记的线段树来实现区间更新和区间查询：

- 线段树构建：构建支持区间求和的线段树
- 懒标记：使用懒标记优化区间更新操作
- 区间更新：支持区间加法操作
- 区间查询：支持区间求和操作

- ****关键技术点**:**

- 懒标记的实现与下传
- 区间更新与查询的处理
- IO 优化与大数据处理

- ****时间复杂度**:**

- 构建线段树： $O(n)$
- 区间更新： $O(\log n)$
- 区间查询： $O(\log n)$

- ****空间复杂度**:** $O(n)$

🔍 线段树经典应用场景

1. 区间最值查询 (RMQ)

- 维护区间最大值/最小值
- 典型题目: HDU 1754 I Hate It

2. 区间求和

- 维护区间元素和
- 典型题目: LeetCode 307. Range Sum Query - Mutable

3. 区间修改

- 支持区间加法、区间赋值等操作
- 典型题目: HDU 1166 敌兵布阵

4. 区间统计

- 统计区间满足特定条件的元素个数
- 典型题目: POJ 2777 Count Color

🛡 线段树实现要点

1. 节点信息设计

- 根据题目需求设计节点存储的信息
- 考虑区间合并的逻辑

2. 懒标记处理

- 懒标记的下传顺序很重要
- 需要正确处理多种懒标记的相互影响

3. 区间合并

- 正确实现 push_up 函数
- 考虑跨区间的情况

4. 边界处理

- 注意数组大小的开法 (通常为 $4n$)
- 注意下标从 0 还是 1 开始

📈 复杂度分析

操作	时间复杂度	空间复杂度
建树	$O(n)$	$O(n)$
单点更新	$O(\log n)$	$O(1)$
区间更新	$O(\log n)$	$O(\log n)$
单点查询	$O(\log n)$	$O(1)$

| 区间查询 | $O(\log n)$ | $O(\log n)$ |

🔍 与其他数据结构的比较

数据结构	优点	缺点	适用场景
线段树	支持区间操作，功能强大	实现复杂，常数较大	频繁区间操作
树状数组	实现简单，常数小	只支持前缀操作	前缀统计问题
平衡树	动态维护有序序列	实现复杂	动态集合操作

🌟 工程化应用

1. 数据库索引

- 在数据库系统中用于范围查询优化

2. 图形学

- 用于空间分割和碰撞检测

3. 网络路由

- 在网络路由算法中维护路径信息

4. 金融系统

- 实时维护股票价格的区间统计信息

📚 相关题目推荐

LeetCode (力扣)

1. **LeetCode 218. 天际线问题** - <https://leetcode.cn/problems/the-skyline-problem/>
 - 类型：线段树+扫描线
 - 难度：困难
2. **LeetCode 307. 区域和检索 - 数组可修改** - <https://leetcode.cn/problems/range-sum-query-mutable/>
 - 类型：线段树基础
 - 难度：中等
3. **LeetCode 308. 二维区域和检索 - 可变** - <https://leetcode.cn/problems/range-sum-query-2d-mutable/>
 - 类型：二维线段树
 - 难度：困难
4. **LeetCode 315. 计算右侧小于当前元素的个数** - <https://leetcode.cn/problems/count-of-smaller-numbers-after-self/>

- 类型: 线段树+离散化
 - 难度: 困难
5. **LeetCode 327. 区间和的个数** - <https://leetcode.cn/problems/count-of-range-sum/>
 - 类型: 线段树+前缀和
 - 难度: 困难

6. **LeetCode 493. 翻转对** - <https://leetcode.cn/problems/reverse-pairs/>
 - 类型: 线段树+离散化
 - 难度: 困难

7. **LeetCode 699. 掉落的方块** - <https://leetcode.cn/problems/falling-squares/>
 - 类型: 线段树+坐标离散化
 - 难度: 困难

8. **LeetCode 715. Range 模块** - <https://leetcode.cn/problems/range-module/>
 - 类型: 线段树+区间合并
 - 难度: 困难

9. **LeetCode 729. 我的日程安排表 I** - <https://leetcode.cn/problems/my-calendar-i/>
 - 类型: 线段树区间查询
 - 难度: 中等

10. **LeetCode 731. 我的日程安排表 II** - <https://leetcode.cn/problems/my-calendar-ii/>
 - 类型: 线段树+最大值维护
 - 难度: 中等

11. **LeetCode 732. 我的日程安排表 III** - <https://leetcode.cn/problems/my-calendar-iii/>
 - 类型: 线段树+最大值维护
 - 难度: 困难

12. **LeetCode 850. 矩形面积 II** - <https://leetcode.cn/problems/rectangle-area-ii/>
 - 类型: 线段树+扫描线
 - 难度: 困难

13. **LeetCode 1157. 子数组中占绝大多数的元素** - <https://leetcode.cn/problems/online-majority-element-in-subarray/>
 - 类型: 线段树+二分查找
 - 难度: 困难

14. **LeetCode 1649. 通过指令创建有序数组** - <https://leetcode.cn/problems/create-sorted-array-through-instructions/>
 - 类型: 线段树+离散化

- 难度: 困难

洛谷 (Luogu)

1. **P3372 【模板】线段树 1** - <https://www.luogu.com.cn/problem/P3372>
 - 类型: 线段树基础模板
 - 难度: 普及+/提高-
2. **P3373 【模板】线段树 2** - <https://www.luogu.com.cn/problem/P3373>
 - 类型: 线段树+多种操作
 - 难度: 提高+/省选-
3. **P2572 [SDOI2010] 序列操作** - <https://www.luogu.com.cn/problem/P2572>
 - 类型: 线段树+多种操作
 - 难度: 省选/NOI-
4. **P1503 [SHOI2008] 洞穴勘测** - <https://www.luogu.com.cn/problem/P1503>
 - 类型: 线段树维护连通性
 - 难度: 提高+/省选-
5. **P2894 [USACO08FEB] Hotel G** - <https://www.luogu.com.cn/problem/P2894>
 - 类型: 线段树维护连续区间
 - 难度: 提高+/省选-
6. **P4514 上帝造题的七分钟** - <https://www.luogu.com.cn/problem/P4514>
 - 类型: 二维线段树
 - 难度: 省选/NOI-
7. **P4198 楼房重建** - <https://www.luogu.com.cn/problem/P4198>
 - 类型: 线段树维护斜率
 - 难度: 省选/NOI-
8. **P2429 制杖题** - <https://www.luogu.com.cn/problem/P2429>
 - 类型: 线段树+数学
 - 难度: 省选/NOI-
9. **P4588 [TJOI2018]数学计算** - <https://www.luogu.com.cn/problem/P4588>
 - 类型: 线段树维护乘积
 - 难度: 提高+/省选-
10. **P2184 贪婪大陆** - <https://www.luogu.com.cn/problem/P2184>
 - 类型: 线段树+前缀和
 - 难度: 提高+/省选-

Codeforces

1. ****339D - Xenia and Bit Operations**** - <https://codeforces.com/problemset/problem/339/D>
 - 类型: 线段树基础
 - 难度: 1900
2. ****380C - Sereja and Brackets**** - <https://codeforces.com/problemset/problem/380/C>
 - 类型: 线段树维护括号匹配
 - 难度: 1700
3. ****52C - Circular RMQ**** - <https://codeforces.com/problemset/problem/52/C>
 - 类型: 线段树+区间更新
 - 难度: 2200
4. ****145E - Lucky Queries**** - <https://codeforces.com/problemset/problem/145/E>
 - 类型: 线段树+懒标记
 - 难度: 2200
5. ****242E - XOR on Segment**** - <https://codeforces.com/problemset/problem/242/E>
 - 类型: 线段树+位运算
 - 难度: 2000
6. ****438D - The Child and Sequence**** - <https://codeforces.com/problemset/problem/438/D>
 - 类型: 线段树+区间取模
 - 难度: 2400
7. ****446C - DZY Loves Fibonacci Numbers**** - <https://codeforces.com/problemset/problem/446/C>
 - 类型: 线段树+斐波那契数列
 - 难度: 2700
8. ****558E - A Simple Task**** - <https://codeforces.com/problemset/problem/558/E>
 - 类型: 线段树+字符串排序
 - 难度: 2300
9. ****610E - Alphabet Permutations**** - <https://codeforces.com/problemset/problem/610/E>
 - 类型: 线段树+字符串处理
 - 难度: 2400
10. ****718C - Sasha and Array**** - <https://codeforces.com/problemset/problem/718/C>
 - 类型: 线段树+矩阵乘法
 - 难度: 2400

洛谷 (Luogu)

1. ****P3372 【模板】线段树 1**** - <https://www.luogu.com.cn/problem/P3372>

- 类型: 线段树基础模板
 - 难度: 普及+/提高-
2. **P3373 【模板】线段树 2** - <https://www.luogu.com.cn/problem/P3373>
 - 类型: 线段树+多种操作
 - 难度: 提高+/省选-

3. **P2572 [SDOI2010] 序列操作** - <https://www.luogu.com.cn/problem/P2572>
 - 类型: 线段树+多种操作
 - 难度: 省选/NOI-

4. **P1503 [SHOI2008] 洞穴勘测** - <https://www.luogu.com.cn/problem/P1503>
 - 类型: 线段树维护连通性
 - 难度: 提高+/省选-

5. **P2894 [USACO08FEB] Hotel G** - <https://www.luogu.com.cn/problem/P2894>
 - 类型: 线段树维护连续区间
 - 难度: 提高+/省选-

6. **P4514 上帝造题的七分钟** - <https://www.luogu.com.cn/problem/P4514>
 - 类型: 二维线段树
 - 难度: 省选/NOI-

7. **P4198 楼房重建** - <https://www.luogu.com.cn/problem/P4198>
 - 类型: 线段树维护斜率
 - 难度: 省选/NOI-

8. **P2429 制杖题** - <https://www.luogu.com.cn/problem/P2429>
 - 类型: 线段树+数学
 - 难度: 省选/NOI-

9. **P4588 [TJOI2018]数学计算** - <https://www.luogu.com.cn/problem/P4588>
 - 类型: 线段树维护乘积
 - 难度: 提高+/省选-

10. **P2184 贪婪大陆** - <https://www.luogu.com.cn/problem/P2184>
 - 类型: 线段树+前缀和
 - 难度: 提高+/省选-

HackerRank

1. **Array and simple queries** - <https://www.hackerrank.com/challenges/array-and-simple-queries/problem>
 - 类型: 线段树基础

- 难度: Advanced

2. **Roy and Coin Boxes** - <https://www.hackerrank.com/challenges/roy-and-coin-boxes/problem>

- 类型: 线段树+差分

- 难度: Advanced

3. **Polynomial Division** - <https://www.hackerrank.com/challenges/polynomial-divison/problem>

- 类型: 线段树+多项式

- 难度: Expert

LintCode (炼码)

1. **201. 线段树的构造** - <https://www.lintcode.com/problem/segment-tree-build/>

- 类型: 线段树基础

- 难度: 中等

2. **202. 线段树查询** - <https://www.lintcode.com/problem/segment-tree-query/>

- 类型: 线段树查询

- 难度: 中等

3. **203. 线段树修改** - <https://www.lintcode.com/problem/segment-tree-modify/>

- 类型: 线段树更新

- 难度: 中等

4. **206. 区间求和 I** - <https://www.lintcode.com/problem/interval-sum/>

- 类型: 线段树求和

- 难度: 中等

5. **207. 区间求和 II** - <https://www.lintcode.com/problem/interval-sum-ii/>

- 类型: 线段树+区间更新

- 难度: 困难

SPOJ

1. **GSS1 - Can you answer these queries I** - <https://www.spoj.com/problems/GSS1/>

- 类型: 线段树最大子段和

- 难度: 中等

2. **GSS3 - Can you answer these queries III** - <https://www.spoj.com/problems/GSS3/>

- 类型: 线段树最大子段和+单点更新

- 难度: 中等

3. **GSS4 - Can you answer these queries IV** - <https://www.spoj.com/problems/GSS4/>

- 类型: 线段树+区间开方

- 难度: 困难

4. **GSS5 – Can you answer these queries V** – <https://www.spoj.com/problems/GSS5/>
 - 类型: 线段树+区间查询
 - 难度: 困难
5. **GSS6 – Can you answer these queries VI** – <https://www.spoj.com/problems/GSS6/>
 - 类型: 平衡树+线段树
 - 难度: 困难
6. **GSS7 – Can you answer these queries VII** – <https://www.spoj.com/problems/GSS7/>
 - 类型: 树链剖分+线段树
 - 难度: 困难

HDU

1. **HDU 1166 敌兵布阵** – <http://acm.hdu.edu.cn/showproblem.php?pid=1166>
 - 类型: 线段树基础
 - 难度: 简单
2. **HDU 1754 I Hate It** – <http://acm.hdu.edu.cn/showproblem.php?pid=1754>
 - 类型: 线段树维护最值
 - 难度: 简单
3. **HDU 1698 Just a Hook** – <http://acm.hdu.edu.cn/showproblem.php?pid=1698>
 - 类型: 线段树+区间更新
 - 难度: 中等
4. **HDU 4528 小明系列故事——捉迷藏** – <http://acm.hdu.edu.cn/showproblem.php?pid=4528>
 - 类型: 线段树+几何
 - 难度: 困难

POJ

1. **POJ 3468 A Simple Problem with Integers** – <http://poj.org/problem?id=3468>
 - 类型: 线段树+区间加法
 - 难度: 中等
2. **POJ 2777 Count Color** – <http://poj.org/problem?id=2777>
 - 类型: 线段树+位运算
 - 难度: 中等
3. **POJ 2528 Mayor' s posters** – <http://poj.org/problem?id=2528>
 - 类型: 线段树+离散化
 - 难度: 中等

4. **POJ 3264 Balanced Lineup** - <http://poj.org/problem?id=3264>

- 类型：线段树维护最值
- 难度：简单

🧠 线段树常见变种

1. 动态开点线段树

- 适用场景：数据范围很大，但实际使用较少的情况
- 典型题目：Codeforces 915E

2. 可持久化线段树（主席树）

- 适用场景：需要保存历史版本的信息
- 典型题目：POJ 2104

3. 扫描线 + 线段树

- 适用场景：平面几何问题，如矩形面积并
- 典型题目：POJ 1151

4. 树链剖分 + 线段树

- 适用场景：树上路径操作
- 典型题目：洛谷 P3384

5. 线段树合并

- 适用场景：动态维护多个集合的信息
- 典型题目：Codeforces 600E

🎓 学习建议

1. **基础阶段**:

- 熟练掌握线段树的基本操作（建树、单点更新、区间查询）
- 练习简单的区间求和、区间最值问题

2. **进阶阶段**:

- 学习懒标记技术，掌握区间更新操作
- 练习复杂的节点信息维护

3. **高手阶段**:

- 学习线段树的各种变种和高级应用
- 掌握线段树与其他算法的结合使用

📚 参考资料

1. 《算法竞赛进阶指南》 - 李煜东

2. 《挑战程序设计竞赛》 - 秋叶拓哉等
3. TopCoder 数据结构教程
4. Codeforces Educational Round 相关讲解

🧠 总结

线段树是处理区间问题的强大工具，虽然实现相对复杂，但其灵活性和效率使其在算法竞赛和工程应用中都有重要地位。掌握线段树的关键在于：

1. 理解其分治思想和二叉树结构
 2. 根据具体问题设计节点信息和合并逻辑
 3. 正确处理懒标记的下传和更新
 4. 熟练掌握各种经典应用场景
-

文件: SOLUTIONS.md

线段树题目详解与实现

1. 序列操作 (Code01_SequenceOperation.java)

题目解析

本题要求实现一个支持多种操作的 01 序列：

1. 区间置 0
2. 区间置 1
3. 区间取反
4. 查询区间 1 的个数
5. 查询区间连续 1 的最长长度

解题思路

使用线段树维护每个区间的以下信息：

- sum: 区间内 1 的个数
- len0/len1: 区间内连续 0/1 的最长子串长度
- pre0/pre1: 区间内连续 0/1 的最长前缀长度
- suf0/suf1: 区间内连续 0/1 的最长后缀长度

同时使用三种懒标记：

- change: 区间被置为的值 (0 或 1)
- update: 是否有更新操作
- reverse: 是否有翻转操作

关键技术点

1. **懒标记优先级**: 更新操作优先于翻转操作
2. **区间合并**: 需要考虑左右子区间连接处的情况
3. **懒标记下传**: 正确处理多种标记的相互影响

Java 实现

```
```java
package class113;

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_SequenceOperation {

 public static int MAXN = 100001;

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

 // 累加和用来统计 1 的数量
 public static int[] sum = new int[MAXN << 2];

 // 连续 0 的最长子串长度
 public static int[] len0 = new int[MAXN << 2];

 // 连续 0 的最长前缀长度
 public static int[] pre0 = new int[MAXN << 2];

 // 连续 0 的最长后缀长度
 public static int[] suf0 = new int[MAXN << 2];

 // 连续 1 的最长子串长度
 public static int[] len1 = new int[MAXN << 2];

 // 连续 1 的最长前缀长度
 public static int[] pre1 = new int[MAXN << 2];
}
```

```

// 连续 1 的最长后缀长度
public static int[] suf1 = new int[MAXN << 2];

// 懒更新信息，范围上所有数字被重置成了什么
public static int[] change = new int[MAXN << 2];

// 懒更新信息，范围上有没有重置任务
public static boolean[] update = new boolean[MAXN << 2];

// 懒更新信息，范围上有没有翻转任务
public static boolean[] reverse = new boolean[MAXN << 2];

public static void up(int i, int ln, int rn) {
 int l = i << 1;
 int r = i << 1 | 1;
 sum[i] = sum[1] + sum[r];
 len0[i] = Math.max(Math.max(len0[1], len0[r]), suf0[1] + pre0[r]);
 pre0[i] = len0[1] < ln ? pre0[1] : (pre0[1] + pre0[r]);
 suf0[i] = len0[r] < rn ? suf0[r] : (suf0[1] + suf0[r]);
 len1[i] = Math.max(Math.max(len1[1], len1[r]), suf1[1] + pre1[r]);
 pre1[i] = len1[1] < ln ? pre1[1] : (pre1[1] + pre1[r]);
 suf1[i] = len1[r] < rn ? suf1[r] : (suf1[1] + suf1[r]);
}

public static void down(int i, int ln, int rn) {
 if (update[i]) {
 updateLazy(i << 1, change[i], ln);
 updateLazy(i << 1 | 1, change[i], rn);
 update[i] = false;
 }
 if (reverse[i]) {
 reverseLazy(i << 1, ln);
 reverseLazy(i << 1 | 1, rn);
 reverse[i] = false;
 }
}

public static void updateLazy(int i, int v, int n) {
 sum[i] = v * n;
 len0[i] = pre0[i] = suf0[i] = v == 0 ? n : 0;
 len1[i] = pre1[i] = suf1[i] = v == 1 ? n : 0;
 change[i] = v;
}

```

```

update[i] = true;
reverse[i] = false;
}

public static void reverseLazy(int i, int n) {
 sum[i] = n - sum[i];
 int tmp;
 tmp = len0[i]; len0[i] = len1[i]; len1[i] = tmp;
 tmp = pre0[i]; pre0[i] = pre1[i]; pre1[i] = tmp;
 tmp = suf0[i]; suf0[i] = suf1[i]; suf1[i] = tmp;
 reverse[i] = !reverse[i];
}

public static void build(int l, int r, int i) {
 if (l == r) {
 sum[i] = arr[1];
 len0[i] = pre0[i] = suf0[i] = arr[1] == 0 ? 1 : 0;
 len1[i] = pre1[i] = suf1[i] = arr[1] == 1 ? 1 : 0;
 } else {
 int mid = (l + r) >> 1;
 build(l, mid, i << 1);
 build(mid + 1, r, i << 1 | 1);
 up(i, mid - 1 + 1, r - mid);
 }
 update[i] = false;
 reverse[i] = false;
}

public static void update(int jobl, int jobr, int jobv, int l, int r, int i) {
 if (jobl <= l && r <= jobr) {
 updateLazy(i, jobv, r - l + 1);
 } else {
 int mid = (l + r) >> 1;
 down(i, mid - 1 + 1, r - mid);
 if (jobl <= mid) {
 update(jobl, jobr, jobv, l, mid, i << 1);
 }
 if (jobr > mid) {
 update(jobl, jobr, jobv, mid + 1, r, i << 1 | 1);
 }
 up(i, mid - 1 + 1, r - mid);
 }
}

```

```

public static void reverse(int jobl, int jobr, int l, int r, int i) {
 if (jobl <= l && r <= jobr) {
 reverseLazy(i, r - l + 1);
 } else {
 int mid = (l + r) >> 1;
 down(i, mid - 1 + 1, r - mid);
 if (jobl <= mid) {
 reverse(jobl, jobr, l, mid, i << 1);
 }
 if (jobr > mid) {
 reverse(jobl, jobr, mid + 1, r, i << 1 | 1);
 }
 up(i, mid - 1 + 1, r - mid);
 }
}

```

// 线段树范围  $l \sim r$  上，被  $jobl \sim jobr$  影响的区域里，返回 1 的数量

```

public static int querySum(int jobl, int jobr, int l, int r, int i) {
 if (jobl <= l && r <= jobr) {
 return sum[i];
 }
 int mid = (l + r) >> 1;
 down(i, mid - 1 + 1, r - mid);
 int ans = 0;
 if (jobl <= mid) {
 ans += querySum(jobl, jobr, l, mid, i << 1);
 }
 if (jobr > mid) {
 ans += querySum(jobl, jobr, mid + 1, r, i << 1 | 1);
 }
 return ans;
}

```

// 返回一个长度为 3 的数组 ans，代表结果，具体含义如下：

```

// ans[0] : 线段树范围 $l \sim r$ 上，被 $jobl \sim jobr$ 影响的区域里，连续 1 的最长子串长度
// ans[1] : 线段树范围 $l \sim r$ 上，被 $jobl \sim jobr$ 影响的区域里，连续 1 的最长前缀长度
// ans[2] : 线段树范围 $l \sim r$ 上，被 $jobl \sim jobr$ 影响的区域里，连续 1 的最长后缀长度
public static int[] queryLongest(int jobl, int jobr, int l, int r, int i) {
 if (jobl <= l && r <= jobr) {
 return new int[] { len1[i], pre1[i], suf1[i] };
 } else {
 int mid = (l + r) >> 1;

```

```

 int ln = mid - 1 + 1;
 int rn = r - mid;
 down(i, ln, rn);
 if (jobr <= mid) {
 return queryLongest(jobl, jobr, l, mid, i << 1);
 }
 if (jobl > mid) {
 return queryLongest(jobl, jobr, mid + 1, r, i << 1 | 1);
 }
 int[] l3 = queryLongest(jobl, jobr, l, mid, i << 1);
 int[] r3 = queryLongest(jobl, jobr, mid + 1, r, i << 1 | 1);
 int llen = l3[0], lpre = l3[1], lsuf = l3[2];
 int rlen = r3[0], rpre = r3[1], rsuf = r3[2];
 int len = Math.max(Math.max(llen, rlen), lsuf + rpre);
 // 任务实际影响了左侧范围的几个点 -> mid - Math.max(jobl, 1) + 1
 int pre = llen < mid - Math.max(jobl, 1) + 1 ? lpre : (lpre + rpre);
 // 任务实际影响了右侧范围的几个点 -> Math.min(r, jobr) - mid
 int suf = rlen < Math.min(r, jobr) - mid ? rsuf : (lsuf + rsuf);
 return new int[] { len, pre, suf };
 }
}

public static void main(String[] args) throws IOException {
 BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
 StreamTokenizer in = new StreamTokenizer(br);
 PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));
 in.nextToken();
 int n = (int) in.nval;
 in.nextToken();
 int m = (int) in.nval;
 for (int i = 1; i <= n; i++) {
 in.nextToken();
 arr[i] = (int) in.nval;
 }
 build(1, n, 1);
 for (int i = 1, op, jobl, jobr; i <= m; i++) {
 in.nextToken();
 op = (int) in.nval;
 in.nextToken();
 jobl = (int) in.nval + 1; // 注意题目给的下标从 0 开始，线段树下标从 1 开始
 in.nextToken();
 jobr = (int) in.nval + 1; // 注意题目给的下标从 0 开始，线段树下标从 1 开始
 if (op == 0) {

```

```

 update(jobl, jobr, 0, 1, n, 1);
 } else if (op == 1) {
 update(jobl, jobr, 1, 1, n, 1);
 } else if (op == 2) {
 reverse(jobl, jobr, 1, n, 1);
 } else if (op == 3) {
 out.println(querySum(jobl, jobr, 1, n, 1));
 } else {
 out.println(queryLongest(jobl, jobr, 1, n, 1)[0]);
 }
}
out.flush();
out.close();
br.close();
}

}
```

```

}

C++实现

```

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

const int MAXN = 100001;

int arr[MAXN];

// 线段树数组
int sum[MAXN << 2];
int len0[MAXN << 2], pre0[MAXN << 2], suf0[MAXN << 2];
int len1[MAXN << 2], pre1[MAXN << 2], suf1[MAXN << 2];

// 懒标记
int change[MAXN << 2];
bool update[MAXN << 2];
bool reverse_flag[MAXN << 2];

void up(int i, int ln, int rn) {
 int l = i << 1, r = i << 1 | 1;
 sum[i] = sum[l] + sum[r];
 len0[i] = max({len0[l], len0[r], suf0[l] + pre0[r]});
 len1[i] = min({len1[l], len1[r], suf1[l] + pre1[r]});
 if (change[i]) {
 if (ln != rn) {
 change[l] = change[r] = true;
 update[l] = update[r] = true;
 }
 reverse_flag[i] = !reverse_flag[i];
 }
 if (update[i]) {
 if (ln != rn) {
 update[l] = update[r] = false;
 change[l] = change[r] = false;
 }
 if (reverse_flag[i])
 swap(len0[i], len1[i]);
 }
}
```

```

```

pre0[i] = len0[1] < ln ? pre0[1] : (pre0[1] + pre0[r]);
suf0[i] = len0[r] < rn ? suf0[r] : (suf0[1] + suf0[r]);
len1[i] = max({len1[1], len1[r], suf1[1] + pre1[r]}));
pre1[i] = len1[1] < ln ? pre1[1] : (pre1[1] + pre1[r]);
suf1[i] = len1[r] < rn ? suf1[r] : (suf1[1] + suf1[r]);
}

void down(int i, int ln, int rn) {
if (update[i]) {
    int l = i << 1, r = i << 1 | 1;
    sum[1] = change[i] * ln;
    len0[1] = pre0[1] = suf0[1] = change[i] == 0 ? ln : 0;
    len1[1] = pre1[1] = suf1[1] = change[i] == 1 ? ln : 0;
    change[1] = change[i];
    update[1] = true;
    reverse_flag[1] = false;

    sum[r] = change[i] * rn;
    len0[r] = pre0[r] = suf0[r] = change[i] == 0 ? rn : 0;
    len1[r] = pre1[r] = suf1[r] = change[i] == 1 ? rn : 0;
    change[r] = change[i];
    update[r] = true;
    reverse_flag[r] = false;

    update[i] = false;
}
if (reverse_flag[i]) {
    int l = i << 1, r = i << 1 | 1;
    sum[1] = ln - sum[1];
    swap(len0[1], len1[1]);
    swap(pre0[1], pre1[1]);
    swap(suf0[1], suf1[1]);
    reverse_flag[1] = !reverse_flag[1];

    sum[r] = rn - sum[r];
    swap(len0[r], len1[r]);
    swap(pre0[r], pre1[r]);
    swap(suf0[r], suf1[r]);
    reverse_flag[r] = !reverse_flag[r];

    reverse_flag[i] = false;
}
}

```

```

void build(int l, int r, int i) {
    if (l == r) {
        sum[i] = arr[l];
        len0[i] = pre0[i] = suf0[i] = arr[l] == 0 ? 1 : 0;
        len1[i] = pre1[i] = suf1[i] = arr[l] == 1 ? 1 : 0;
    } else {
        int mid = (l + r) >> 1;
        build(l, mid, i << 1);
        build(mid + 1, r, i << 1 | 1);
        up(i, mid - 1 + 1, r - mid);
    }
    update[i] = false;
    reverse_flag[i] = false;
}

void update_range(int jobl, int jobr, int jobv, int l, int r, int i) {
    if (jobl <= l && r <= jobr) {
        sum[i] = jobv * (r - l + 1);
        len0[i] = pre0[i] = suf0[i] = jobv == 0 ? (r - l + 1) : 0;
        len1[i] = pre1[i] = suf1[i] = jobv == 1 ? (r - l + 1) : 0;
        change[i] = jobv;
        update[i] = true;
        reverse_flag[i] = false;
    } else {
        int mid = (l + r) >> 1;
        down(i, mid - 1 + 1, r - mid);
        if (jobl <= mid) {
            update_range(jobl, jobr, jobv, l, mid, i << 1);
        }
        if (jobr > mid) {
            update_range(jobl, jobr, jobv, mid + 1, r, i << 1 | 1);
        }
        up(i, mid - 1 + 1, r - mid);
    }
}

void reverse_range(int jobl, int jobr, int l, int r, int i) {
    if (jobl <= l && r <= jobr) {
        sum[i] = (r - l + 1) - sum[i];
        swap(len0[i], len1[i]);
        swap(pre0[i], pre1[i]);
        swap(suf0[i], suf1[i]);
    }
}

```

```

reverse_flag[i] = !reverse_flag[i];
} else {
    int mid = (l + r) >> 1;
    down(i, mid - 1 + 1, r - mid);
    if (jobl <= mid) {
        reverse_range(jobl, jobr, l, mid, i << 1);
    }
    if (jobr > mid) {
        reverse_range(jobl, jobr, mid + 1, r, i << 1 | 1);
    }
    up(i, mid - 1 + 1, r - mid);
}
}

int query_sum(int jobl, int jobr, int l, int r, int i) {
if (jobl <= l && r <= jobr) {
    return sum[i];
}
int mid = (l + r) >> 1;
down(i, mid - 1 + 1, r - mid);
int ans = 0;
if (jobl <= mid) {
    ans += query_sum(jobl, jobr, l, mid, i << 1);
}
if (jobr > mid) {
    ans += query_sum(jobl, jobr, mid + 1, r, i << 1 | 1);
}
return ans;
}

vector<int> query_longest(int jobl, int jobr, int l, int r, int i) {
if (jobl <= l && r <= jobr) {
    return {len1[i], pre1[i], suf1[i]};
} else {
    int mid = (l + r) >> 1;
    int ln = mid - 1 + 1;
    int rn = r - mid;
    down(i, ln, rn);
    if (jobr <= mid) {
        return query_longest(jobl, jobr, l, mid, i << 1);
    }
    if (jobl > mid) {
        return query_longest(jobl, jobr, mid + 1, r, i << 1 | 1);
    }
}
}

```

```

    }

vector<int> l3 = query_longest(jobl, jobr, 1, mid, i << 1);
vector<int> r3 = query_longest(jobl, jobr, mid + 1, r, i << 1 | 1);
int llen = l3[0], lpre = l3[1], lsuf = l3[2];
int rlen = r3[0], rpre = r3[1], rsuf = r3[2];
int len = max({llen, rlen, lsuf + rpre});
int pre = llen < mid - max(jobl, 1) + 1 ? lpre : (lpre + rpre);
int suf = rlen < min(r, jobr) - mid ? rsuf : (lsuf + rsuf);
return {len, pre, suf};
}

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

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

    for (int i = 1; i <= m; i++) {
        int op, jobl, jobr;
        cin >> op >> jobl >> jobr;
        jobl++; jobr++; // 转换为 1-based

        if (op == 0) {
            update_range(jobl, jobr, 0, 1, n, 1);
        } else if (op == 1) {
            update_range(jobl, jobr, 1, 1, n, 1);
        } else if (op == 2) {
            reverse_range(jobl, jobr, 1, n, 1);
        } else if (op == 3) {
            cout << query_sum(jobl, jobr, 1, n, 1) << "\n";
        } else {
            cout << query_longest(jobl, jobr, 1, n, 1)[0] << "\n";
        }
    }

    return 0;
}

```

```
}
```

```
...
```

```
#### Python 实现
```

```
``` python
import sys
from collections import deque

class SegmentTree:

 def __init__(self, arr):
 self.n = len(arr)
 self.arr = arr
 self.sum = [0] * (4 * self.n)
 self.len0 = [0] * (4 * self.n)
 self.pre0 = [0] * (4 * self.n)
 self.suf0 = [0] * (4 * self.n)
 self.len1 = [0] * (4 * self.n)
 self.pre1 = [0] * (4 * self.n)
 self.suf1 = [0] * (4 * self.n)
 self.change = [0] * (4 * self.n)
 self.update = [False] * (4 * self.n)
 self.reverse = [False] * (4 * self.n)
 self.build(1, 1, self.n)

 def up(self, i, ln, rn):
 l = i << 1
 r = i << 1 | 1
 self.sum[i] = self.sum[l] + self.sum[r]
 self.len0[i] = max(self.len0[l], self.len0[r], self.suf0[l] + self.pre0[r])
 self.pre0[i] = self.pre0[l] if self.len0[l] < ln else self.pre0[l] + self.pre0[r]
 self.suf0[i] = self.suf0[r] if self.len0[r] < rn else self.suf0[r] + self.suf0[l]
 self.len1[i] = max(self.len1[l], self.len1[r], self.suf1[l] + self.pre1[r])
 self.pre1[i] = self.pre1[l] if self.len1[l] < ln else self.pre1[l] + self.pre1[r]
 self.suf1[i] = self.suf1[r] if self.len1[r] < rn else self.suf1[r] + self.suf1[l]

 def update_lazy(self, i, v, n):
 self.sum[i] = v * n
 self.len0[i] = self.pre0[i] = self.suf0[i] = n if v == 0 else 0
 self.len1[i] = self.pre1[i] = self.suf1[i] = n if v == 1 else 0
 self.change[i] = v
 self.update[i] = True
 self.reverse[i] = False
```

```

def reverse_lazy(self, i, n):
 self.sum[i] = n - self.sum[i]
 self.len0[i], self.len1[i] = self.len1[i], self.len0[i]
 self.pre0[i], self.pre1[i] = self.pre1[i], self.pre0[i]
 self.suf0[i], self.suf1[i] = self.suf1[i], self.suf0[i]
 self.reverse[i] = not self.reverse[i]

def down(self, i, ln, rn):
 if self.update[i]:
 self.update_lazy(i << 1, self.change[i], ln)
 self.update_lazy(i << 1 | 1, self.change[i], rn)
 self.update[i] = False
 if self.reverse[i]:
 self.reverse_lazy(i << 1, ln)
 self.reverse_lazy(i << 1 | 1, rn)
 self.reverse[i] = False

def build(self, i, l, r):
 if l == r:
 self.sum[i] = self.arr[l]
 self.len0[i] = self.pre0[i] = self.suf0[i] = 1 if self.arr[l] == 0 else 0
 self.len1[i] = self.pre1[i] = self.suf1[i] = 1 if self.arr[l] == 1 else 0
 else:
 mid = (l + r) >> 1
 self.build(i << 1, l, mid)
 self.build(i << 1 | 1, mid + 1, r)
 self.up(i, mid - 1 + 1, r - mid)
 self.update[i] = False
 self.reverse[i] = False

def update_range(self, jobl, jobr, jobv, l, r, i):
 if jobl <= l and r <= jobr:
 self.update_lazy(i, jobv, r - l + 1)
 else:
 mid = (l + r) >> 1
 ln = mid - 1 + 1
 rn = r - mid
 self.down(i, ln, rn)
 if jobl <= mid:
 self.update_range(jobl, jobr, jobv, l, mid, i << 1)
 if jobr > mid:
 self.update_range(jobl, jobr, jobv, mid + 1, r, i << 1 | 1)

```

```

 self.up(i, ln, rn)

def reverse_range(self, jobl, jobr, l, r, i):
 if jobl <= l and r <= jobr:
 self.reverse_lazy(i, r - l + 1)
 else:
 mid = (l + r) >> 1
 ln = mid - 1 + 1
 rn = r - mid
 self.down(i, ln, rn)
 if jobl <= mid:
 self.reverse_range(jobl, jobr, l, mid, i << 1)
 if jobr > mid:
 self.reverse_range(jobl, jobr, mid + 1, r, i << 1 | 1)
 self.up(i, ln, rn)

def query_sum(self, jobl, jobr, l, r, i):
 if jobl <= l and r <= jobr:
 return self.sum[i]
 mid = (l + r) >> 1
 ln = mid - 1 + 1
 rn = r - mid
 self.down(i, ln, rn)
 ans = 0
 if jobl <= mid:
 ans += self.query_sum(jobl, jobr, l, mid, i << 1)
 if jobr > mid:
 ans += self.query_sum(jobl, jobr, mid + 1, r, i << 1 | 1)
 return ans

def query_longest(self, jobl, jobr, l, r, i):
 if jobl <= l and r <= jobr:
 return [self.len1[i], self.pre1[i], self.suf1[i]]
 else:
 mid = (l + r) >> 1
 ln = mid - 1 + 1
 rn = r - mid
 self.down(i, ln, rn)
 if jobr <= mid:
 return self.query_longest(jobl, jobr, l, mid, i << 1)
 if jobl > mid:
 return self.query_longest(jobl, jobr, mid + 1, r, i << 1 | 1)
 13 = self.query_longest(jobl, jobr, l, mid, i << 1)

```

```

r3 = self.query_longest(jobl, jobr, mid + 1, r, i << 1 | 1)
llen, lpre, lsuf = 13[0], 13[1], 13[2]
rlen, rpre, rsuf = r3[0], r3[1], r3[2]
length = max(llen, rlen, lsuf + rpre)
pre = lpre if llen < mid - max(jobl, 1) + 1 else lpre + rpre
suf = rsuf if rlen < min(r, jobr) - mid else lsuf + rsuf
return [length, pre, suf]

def main():
 import sys
 input = sys.stdin.read
 data = input().split()

 idx = 0
 n = int(data[idx])
 idx += 1
 m = int(data[idx])
 idx += 1

 arr = [0] * (n + 1)
 for i in range(1, n + 1):
 arr[i] = int(data[idx])
 idx += 1

 seg_tree = SegmentTree(arr)

 results = []
 for _ in range(m):
 op = int(data[idx])
 idx += 1
 jobl = int(data[idx]) + 1 # 转换为1-based
 idx += 1
 jobr = int(data[idx]) + 1 # 转换为1-based
 idx += 1

 if op == 0:
 seg_tree.update_range(jobl, jobr, 0, 1, n, 1)
 elif op == 1:
 seg_tree.update_range(jobl, jobr, 1, 1, n, 1)
 elif op == 2:
 seg_tree.reverse_range(jobl, jobr, 1, n, 1)
 elif op == 3:
 results.append(str(seg_tree.query_sum(jobl, jobr, 1, n, 1)))

```

```

else:
 results.append(str(seg_tree.query_longest(jobl, jobr, 1, n, 1)[0]))

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

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

```

复杂度分析

- **时间复杂度**:
 - 建树: $O(n)$
 - 单次操作: $O(\log n)$
 - 总时间复杂度: $O((n + m) \log n)$
- **空间复杂度**: $O(n)$

2. 最长 LR 交替子串 (Code02_LongestAlternateSubstring.java)

题目解析

给定一个字符串，初始全为'L'，每次操作翻转一个位置的字符，求每次操作后最长的 LR 交替子串长度。

解题思路

使用线段树维护每个区间的最长交替子串长度，以及前缀和后缀的最长交替长度。

关键技术点

1. 区间合并时需要判断中间连接处是否可以连接
2. 单点更新时需要重新计算区间信息

Java 实现

```

```
java
package class113;

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_LongestAlternateSubstring {

 public static int MAXN = 200001;

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

 // 交替最长子串长度
 public static int[] len = new int[MAXN << 2];

 // 交替最长前缀长度
 public static int[] pre = new int[MAXN << 2];

 // 交替最长后缀长度
 public static int[] suf = new int[MAXN << 2];

 public static void up(int l, int r, int i) {
 len[i] = Math.max(len[i << 1], len[i << 1 | 1]);
 pre[i] = pre[i << 1];
 suf[i] = suf[i << 1 | 1];
 int mid = (l + r) >> 1;
 int ln = mid - 1 + 1;
 int rn = r - mid;
 if (arr[mid] != arr[mid + 1]) {
 len[i] = Math.max(len[i], suf[i << 1] + pre[i << 1 | 1]);
 if (len[i << 1] == ln) {
 pre[i] = ln + pre[i << 1 | 1];
 }
 if (len[i << 1 | 1] == rn) {
 suf[i] = rn + suf[i << 1];
 }
 }
 }

 public static void build(int l, int r, int i) {
 if (l == r) {
 len[i] = 1;
 pre[i] = 1;
 suf[i] = 1;
 } else {
 int mid = (l + r) >> 1;
```

```

 build(l, mid, i << 1);
 build(mid + 1, r, i << 1 | 1);
 up(l, r, i);
 }
}

public static void reverse(int jobi, int l, int r, int i) {
 if (l == r) {
 arr[jobi] ^= 1;
 } else {
 int mid = (l + r) >> 1;
 if (jobi <= mid) {
 reverse(jobi, l, mid, i << 1);
 } else {
 reverse(jobi, mid + 1, r, i << 1 | 1);
 }
 up(l, r, i);
 }
}

public static void main(String[] args) throws IOException {
 BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
 StreamTokenizer in = new StreamTokenizer(br);
 PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));
 in.nextToken();
 int n = (int) in.nval;
 in.nextToken();
 int q = (int) in.nval;
 build(1, n, 1);
 for (int i = 1, index; i <= q; i++) {
 in.nextToken();
 index = (int) in.nval;
 reverse(index, 1, n, 1);
 out.println(len[1]);
 }
 out.flush();
 out.close();
 br.close();
}

```

```

C++实现

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

const int MAXN = 200001;

int arr[MAXN];
int len[MAXN << 2];
int pre[MAXN << 2];
int suf[MAXN << 2];

void up(int l, int r, int i) {
 len[i] = max(len[i << 1], len[i << 1 | 1]);
 pre[i] = pre[i << 1];
 suf[i] = suf[i << 1 | 1];
 int mid = (l + r) >> 1;
 int ln = mid - l + 1;
 int rn = r - mid;
 if (arr[mid] != arr[mid + 1]) {
 len[i] = max(len[i], suf[i << 1] + pre[i << 1 | 1]);
 if (len[i << 1] == ln) {
 pre[i] = ln + pre[i << 1 | 1];
 }
 if (len[i << 1 | 1] == rn) {
 suf[i] = rn + suf[i << 1];
 }
 }
}

void build(int l, int r, int i) {
 if (l == r) {
 len[i] = pre[i] = suf[i] = 1;
 } else {
 int mid = (l + r) >> 1;
 build(l, mid, i << 1);
 build(mid + 1, r, i << 1 | 1);
 up(l, r, i);
 }
}

void reverse_char(int jobi, int l, int r, int i) {
```

```

if (l == r) {
 arr[jobi] ^= 1;
} else {
 int mid = (l + r) >> 1;
 if (jobi <= mid) {
 reverse_char(jobi, l, mid, i << 1);
 } else {
 reverse_char(jobi, mid + 1, r, i << 1 | 1);
 }
 up(l, r, i);
}
}

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

 int n, q;
 cin >> n >> q;
 build(1, n, 1);

 for (int i = 1; i <= q; i++) {
 int index;
 cin >> index;
 reverse_char(index, 1, n, 1);
 cout << len[1] << "\n";
 }

 return 0;
}
```

```

Python 实现

```

``` python
import sys

class AlternateSubstringTree:
 def __init__(self, n):
 self.n = n
 self.arr = [0] * (n + 1) # 0 represents 'L', 1 represents 'R'
 self.len = [0] * (4 * n)

```

```

self.pre = [0] * (4 * n)
self.suf = [0] * (4 * n)
self.build(1, 1, n)

def up(self, l, r, i):
 self.len[i] = max(self.len[i << 1], self.len[i << 1 | 1])
 self.pre[i] = self.pre[i << 1]
 self.suf[i] = self.suf[i << 1 | 1]
 mid = (l + r) >> 1
 ln = mid - 1 + 1
 rn = r - mid
 if self.arr[mid] != self.arr[mid + 1]:
 self.len[i] = max(self.len[i], self.suf[i << 1] + self.pre[i << 1 | 1])
 if self.len[i << 1] == ln:
 self.pre[i] = ln + self.pre[i << 1 | 1]
 if self.len[i << 1 | 1] == rn:
 self.suf[i] = rn + self.suf[i << 1]

def build(self, i, l, r):
 if l == r:
 self.len[i] = self.pre[i] = self.suf[i] = 1
 else:
 mid = (l + r) >> 1
 self.build(i << 1, l, mid)
 self.build(i << 1 | 1, mid + 1, r)
 self.up(l, r, i)

def reverse_char(self, jobi, l, r, i):
 if l == r:
 self.arr[jobi] ^= 1
 else:
 mid = (l + r) >> 1
 if jobi <= mid:
 self.reverse_char(jobi, l, mid, i << 1)
 else:
 self.reverse_char(jobi, mid + 1, r, i << 1 | 1)
 self.up(l, r, i)

def main():
 import sys
 input = sys.stdin.read
 data = input().split()

```

```
n = int(data[0])
q = int(data[1])

tree = AlternateSubstringTree(n)

results = []
idx = 2
for _ in range(q):
 index = int(data[idx])
 idx += 1
 tree.reverse_char(index, 1, n, 1)
 results.append(str(tree.len[1]))

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

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

复杂度分析

- **时间复杂度**:
 - 建树: $O(n)$
 - 单次操作: $O(\log n)$
 - 总时间复杂度: $O(n + q \log n)$
- **空间复杂度**: $O(n)$

3. 地道相连的房子 (Code03_TunnelWarfare.java)

题目解析

有 n 个房子排成一排，相邻房子有地道连接。支持摧毁、恢复和查询操作。

解题思路

使用线段树维护每个区间的连续 1 的前缀和后缀长度，其中 1 表示房子未被摧毁。

关键技术点

1. 查询操作需要根据位置在区间中的位置进行不同处理
2. 区间合并时考虑跨区间的情况

Java 实现

```
```java
package class113;

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_TunnelWarfare {

 public static int MAXN = 50001;

 // 连续 1 的最长前缀长度
 public static int[] pre = new int[MAXN << 2];

 // 连续 1 的最长后缀长度
 public static int[] suf = new int[MAXN << 2];

 // 摧毁的房屋编号入栈，以便执行恢复操作
 public static int[] stack = new int[MAXN];

 public static void up(int l, int r, int i) {
 pre[i] = pre[i << 1];
 suf[i] = suf[i << 1 | 1];
 int mid = (l + r) >> 1;
 if (pre[i << 1] == mid - 1 + 1) {
 pre[i] += pre[i << 1 | 1];
 }
 if (suf[i << 1 | 1] == r - mid) {
 suf[i] += suf[i << 1];
 }
 }

 public static void build(int l, int r, int i) {
 if (l == r) {
 pre[i] = suf[i] = 1;
 } else {
 int mid = (l + r) >> 1;
 build(l, mid, i << 1);
 build(mid, r, i << 1);
 up(l, r, i);
 }
 }
}
```

```

 build(mid + 1, r, i << 1 | 1);
 up(l, r, i);
 }
}

public static void update(int jobi, int jobv, int l, int r, int i) {
 if (l == r) {
 pre[i] = suf[i] = jobv;
 } else {
 int mid = (l + r) >> 1;
 if (jobi <= mid) {
 update(jobi, jobv, l, mid, i << 1);
 } else {
 update(jobi, jobv, mid + 1, r, i << 1 | 1);
 }
 up(l, r, i);
 }
}

// 已知 jobi 在 l...r 范围上
// 返回 jobi 往两侧扩展出的最大长度
// 递归需要遵循的潜台词：从 jobi 往两侧扩展，一定无法扩展到 l...r 范围之外！
public static int query(int jobi, int l, int r, int i) {
 if (l == r) {
 return pre[i];
 } else {
 int mid = (l + r) >> 1;
 if (jobi <= mid) {
 if (jobi > mid - suf[i << 1]) {
 return suf[i << 1] + pre[i << 1 | 1];
 } else {
 return query(jobi, l, mid, i << 1);
 }
 } else {
 if (mid + pre[i << 1 | 1] >= jobi) {
 return suf[i << 1] + pre[i << 1 | 1];
 } else {
 return query(jobi, mid + 1, r, i << 1 | 1);
 }
 }
 }
}
}

```

```

public static void main(String[] args) throws IOException {
 BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
 StreamTokenizer in = new StreamTokenizer(br);
 PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));
 while (in.nextToken() != StreamTokenizer.TT_EOF) {
 int n = (int) in.nval;
 in.nextToken();
 int m = (int) in.nval;
 build(1, n, 1);
 String op;
 int stackSize = 0;
 for (int i = 1, x; i <= m; i++) {
 in.nextToken();
 op = in.sval;
 if (op.equals("D")) {
 in.nextToken();
 x = (int) in.nval;
 update(x, 0, 1, n, 1);
 stack[stackSize++] = x;
 } else if (op.equals("R")) {
 update(stack[--stackSize], 1, 1, n, 1);
 } else {
 in.nextToken();
 x = (int) in.nval;
 out.println(query(x, 1, n, 1));
 }
 }
 out.flush();
 out.close();
 br.close();
 }
}
```

```

C++实现

```

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

const int MAXN = 50001;

```

```

int pre[MAXN << 2];
int suf[MAXN << 2];
int stack_arr[MAXN];

void up(int l, int r, int i) {
 pre[i] = pre[i << 1];
 suf[i] = suf[i << 1 | 1];
 int mid = (l + r) >> 1;
 if (pre[i << 1] == mid - 1 + 1) {
 pre[i] += pre[i << 1 | 1];
 }
 if (suf[i << 1 | 1] == r - mid) {
 suf[i] += suf[i << 1];
 }
}

void build(int l, int r, int i) {
 if (l == r) {
 pre[i] = suf[i] = 1;
 } else {
 int mid = (l + r) >> 1;
 build(l, mid, i << 1);
 build(mid + 1, r, i << 1 | 1);
 up(l, r, i);
 }
}

void update(int jobi, int jobv, int l, int r, int i) {
 if (l == r) {
 pre[i] = suf[i] = jobv;
 } else {
 int mid = (l + r) >> 1;
 if (jobi <= mid) {
 update(jobi, jobv, l, mid, i << 1);
 } else {
 update(jobi, jobv, mid + 1, r, i << 1 | 1);
 }
 up(l, r, i);
 }
}

int query(int jobi, int l, int r, int i) {

```

```

if (l == r) {
 return pre[i];
} else {
 int mid = (l + r) >> 1;
 if (jobi <= mid) {
 if (jobi > mid - suf[i << 1]) {
 return suf[i << 1] + pre[i << 1 | 1];
 } else {
 return query(jobi, l, mid, i << 1);
 }
 } else {
 if (mid + pre[i << 1 | 1] >= jobi) {
 return suf[i << 1] + pre[i << 1 | 1];
 } else {
 return query(jobi, mid + 1, r, i << 1 | 1);
 }
 }
}
}

```

```

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

 int n, m;
 while (cin >> n >> m) {
 build(l, n, 1);
 int stack_arr[MAXN];
 int stackSize = 0;

 for (int i = 1; i <= m; i++) {
 string op;
 cin >> op;
 if (op == "D") {
 int x;
 cin >> x;
 update(x, 0, 1, n, 1);
 stack_arr[stackSize++] = x;
 } else if (op == "R") {
 update(stack_arr[--stackSize], 1, 1, n, 1);
 } else {
 int x;

```

```

 cin >> x;
 cout << query(x, 1, n, 1) << "\n";
 }
}

return 0;
}
```

```

Python 实现

```

``` python
import sys

class TunnelWarfareTree:

 def __init__(self, n):
 self.n = n
 self.pre = [0] * (4 * n)
 self.suf = [0] * (4 * n)
 self.build(1, 1, n)

 def up(self, l, r, i):
 self.pre[i] = self.pre[i << 1]
 self.suf[i] = self.suf[i << 1 | 1]
 mid = (l + r) >> 1
 if self.pre[i << 1] == mid - 1 + 1:
 self.pre[i] += self.pre[i << 1 | 1]
 if self.suf[i << 1 | 1] == r - mid:
 self.suf[i] += self.suf[i << 1]

 def build(self, i, l, r):
 if l == r:
 self.pre[i] = self.suf[i] = 1
 else:
 mid = (l + r) >> 1
 self.build(i << 1, l, mid)
 self.build(i << 1 | 1, mid + 1, r)
 self.up(l, r, i)

 def update(self, jobi, jobv, l, r, i):
 if l == r:
 self.pre[i] = self.suf[i] = jobv

```

```

else:
 mid = (l + r) >> 1
 if jobi <= mid:
 self.update(jobi, jobv, l, mid, i << 1)
 else:
 self.update(jobi, jobv, mid + 1, r, i << 1 | 1)
 self.up(l, r, i)

def query(self, jobi, l, r, i):
 if l == r:
 return self.pre[i]
 else:
 mid = (l + r) >> 1
 if jobi <= mid:
 if jobi > mid - self.suf[i << 1]:
 return self.suf[i << 1] + self.pre[i << 1 | 1]
 else:
 return self.query(jobi, l, mid, i << 1)
 else:
 if mid + self.pre[i << 1 | 1] >= jobi:
 return self.suf[i << 1] + self.pre[i << 1 | 1]
 else:
 return self.query(jobi, mid + 1, r, i << 1 | 1)

def main():
 import sys
 input = sys.stdin.read
 data = input().split()

 idx = 0
 while idx < len(data):
 n = int(data[idx])
 idx += 1
 m = int(data[idx])
 idx += 1

 tree = TunnelWarfareTree(n)
 stack_arr = []

 results = []
 for _ in range(m):
 op = data[idx]
 idx += 1

```

```

if op == "D":
 x = int(data[idx])
 idx += 1
 tree.update(x, 0, 1, n, 1)
 stack_arr.append(x)
elif op == "R":
 x = stack_arr.pop()
 tree.update(x, 1, 1, n, 1)
else: # op == "Q"
 x = int(data[idx])
 idx += 1
 results.append(str(tree.query(x, 1, n, 1)))

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

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

```

复杂度分析

- **时间复杂度**:
 - 建树: $O(n)$
 - 单次操作: $O(\log n)$
 - 总时间复杂度: $O((n + m) \log n)$
- **空间复杂度**: $O(n)$

4. 旅馆 (Code04_Hotel.java)

题目解析

有 n 个房间，初始都为空房。支持查找连续空房间和清空房间操作。

解题思路

使用线段树维护每个区间的连续空房间信息，包括最长连续空房间长度、前缀和后缀长度。

关键技术点

1. 查询最左边满足条件的区间需要特殊处理
2. 区间合并时需要考虑左右子区间的连接情况

Java 实现

```
```java
package class113;

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 Code04_Hotel {

 public static int MAXN = 50001;

 // 连续空房最长子串长度
 public static int[] len = new int[MAXN << 2];

 // 连续空房最长前缀长度
 public static int[] pre = new int[MAXN << 2];

 // 连续空房最长后缀长度
 public static int[] suf = new int[MAXN << 2];

 // 懒更新信息，范围上所有数字被重置成了什么
 public static int[] change = new int[MAXN << 2];

 // 懒更新信息，范围上有没有重置任务
 public static boolean[] update = new boolean[MAXN << 2];

 public static void up(int i, int ln, int rn) {
 int l = i << 1;
 int r = i << 1 | 1;
 len[i] = Math.max(Math.max(len[l], len[r]), suf[l] + pre[r]);
 pre[i] = len[l] < ln ? pre[l] : (pre[l] + pre[r]);
 suf[i] = len[r] < rn ? suf[r] : (suf[l] + suf[r]);
 }

 public static void down(int i, int ln, int rn) {
 if (update[i]) {
 lazy(i << 1, change[i], ln);
 }
 }
}
```

```

 lazy(i << 1 | 1, change[i], rn);
 update[i] = false;
 }
}

public static void lazy(int i, int v, int n) {
 len[i] = pre[i] = suf[i] = v == 0 ? n : 0;
 change[i] = v;
 update[i] = true;
}

public static void build(int l, int r, int i) {
 if (l == r) {
 len[i] = pre[i] = suf[i] = 1;
 } else {
 int mid = (l + r) >> 1;
 build(l, mid, i << 1);
 build(mid + 1, r, i << 1 | 1);
 up(i, mid - 1 + 1, r - mid);
 }
 update[i] = false;
}

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

// 在 l..r 范围上，在满足空房长度>=x 的情况下，返回尽量靠左的开头位置
// 递归需要遵循的潜台词：l..r 范围上一定存在连续空房长度>=x 的区域
public static int queryLeft(int x, int l, int r, int i) {
 if (l == r) {

```

```

 return 1;
 } else {
 int mid = (l + r) >> 1;
 down(i, mid - 1 + 1, r - mid);
 // 最先查左边
 if (len[i << 1] >= x) {
 return queryLeft(x, l, mid, i << 1);
 }
 // 然后查中间向两边扩展的可能区域
 if (suf[i << 1] + pre[i << 1 | 1] >= x) {
 return mid - suf[i << 1] + 1;
 }
 // 前面都没有再最后查右边
 return queryLeft(x, mid + 1, r, i << 1 | 1);
 }
}

```

```

public static void main(String[] args) throws IOException {
 BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
 StreamTokenizer in = new StreamTokenizer(br);
 PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));
 in.nextToken();
 int n = (int) in.nval;
 build(1, n, 1);
 in.nextToken();
 int m = (int) in.nval;
 for (int i = 1, op, x, y, left; i <= m; i++) {
 in.nextToken();
 op = (int) in.nval;
 if (op == 1) {
 in.nextToken();
 x = (int) in.nval;
 if (len[1] < x) {
 left = 0;
 } else {
 left = queryLeft(x, 1, n, 1);
 update(left, left + x - 1, 1, 1, n, 1);
 }
 out.println(left);
 } else {
 in.nextToken();
 x = (int) in.nval;
 in.nextToken();
 }
 }
}

```

```

 y = (int) in.nval;
 update(x, Math.min(x + y - 1, n), 0, 1, n, 1);
 }
}

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

}

}
```

```

C++实现

```

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

const int MAXN = 50001;

int len[MAXN << 2];
int pre[MAXN << 2];
int suf[MAXN << 2];
int change[MAXN << 2];
bool update[MAXN << 2];

void up(int i, int ln, int rn) {
 int l = i << 1;
 int r = i << 1 | 1;
 len[i] = max({len[l], len[r], suf[l] + pre[r]});
 pre[i] = len[l] < ln ? pre[l] : (pre[l] + pre[r]);
 suf[i] = len[r] < rn ? suf[r] : (suf[l] + suf[r]);
}

void down(int i, int ln, int rn) {
 if (update[i]) {
 int l = i << 1, r = i << 1 | 1;
 len[1] = pre[1] = suf[1] = change[i] == 0 ? ln : 0;
 change[1] = change[i];
 update[1] = true;

 len[r] = pre[r] = suf[r] = change[i] == 0 ? rn : 0;
 change[r] = change[i];
 }
}

```

```

update[r] = true;

update[i] = false;
}

}

void lazy_update(int i, int v, int n) {
 len[i] = pre[i] = suf[i] = v == 0 ? n : 0;
 change[i] = v;
 update[i] = true;
}

void build(int l, int r, int i) {
 if (l == r) {
 len[i] = pre[i] = suf[i] = 1;
 } else {
 int mid = (l + r) >> 1;
 build(l, mid, i << 1);
 build(mid + 1, r, i << 1 | 1);
 up(i, mid - 1 + 1, r - mid);
 }
 update[i] = false;
}

void update_range(int jobl, int jobr, int jobv, int l, int r, int i) {
 if (jobl <= l && r <= jobr) {
 lazy_update(i, jobv, r - l + 1);
 } else {
 int mid = (l + r) >> 1;
 down(i, mid - 1 + 1, r - mid);
 if (jobl <= mid) {
 update_range(jobl, jobr, jobv, l, mid, i << 1);
 }
 if (jobr > mid) {
 update_range(jobl, jobr, jobv, mid + 1, r, i << 1 | 1);
 }
 up(i, mid - 1 + 1, r - mid);
 }
}

int query_left(int x, int l, int r, int i) {
 if (l == r) {
 return 1;
 }
}

```

```

} else {
 int mid = (l + r) >> 1;
 down(i, mid - 1 + 1, r - mid);
 // 最先查左边
 if (len[i << 1] >= x) {
 return query_left(x, l, mid, i << 1);
 }
 // 然后查中间向两边扩展的可能区域
 if (suf[i << 1] + pre[i << 1 | 1] >= x) {
 return mid - suf[i << 1] + 1;
 }
 // 前面都没有再最后查右边
 return query_left(x, mid + 1, r, i << 1 | 1);
}
}

```

```

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

 int n, m;
 cin >> n;
 build(1, n, 1);
 cin >> m;

 for (int i = 1; i <= m; i++) {
 int op;
 cin >> op;
 if (op == 1) {
 int x;
 cin >> x;
 int left;
 if (len[1] < x) {
 left = 0;
 } else {
 left = query_left(x, 1, n, 1);
 update_range(left, left + x - 1, 1, 1, n, 1);
 }
 cout << left << "\n";
 } else {
 int x, y;
 cin >> x >> y;
 }
 }
}

```

```

 update_range(x, min(x + y - 1, n), 0, 1, n, 1);
 }
}

return 0;
}
```

```

Python 实现

```

``` python
import sys

class HotelTree:
 def __init__(self, n):
 self.n = n
 self.len = [0] * (4 * n)
 self.pre = [0] * (4 * n)
 self.suf = [0] * (4 * n)
 self.change = [0] * (4 * n)
 self.update = [False] * (4 * n)
 self.build(1, 1, n)

 def up(self, i, ln, rn):
 l = i << 1
 r = i << 1 | 1
 self.len[i] = max(self.len[l], self.len[r], self.suf[l] + self.pre[r])
 self.pre[i] = self.pre[l] if self.len[l] < ln else self.pre[l] + self.pre[r]
 self.suf[i] = self.suf[r] if self.len[r] < rn else self.suf[l] + self.suf[r]

 def down(self, i, ln, rn):
 if self.update[i]:
 l = i << 1
 r = i << 1 | 1
 self.len[l] = self.pre[l] = self.suf[l] = 0 if self.change[i] == 1 else ln
 self.change[l] = self.change[i]
 self.update[l] = True

 self.len[r] = self.pre[r] = self.suf[r] = 0 if self.change[i] == 1 else rn
 self.change[r] = self.change[i]
 self.update[r] = True

 self.update[i] = False

```

```

def lazy_update(self, i, v, n):
 self.len[i] = self.pre[i] = self.suf[i] = 0 if v == 1 else n
 self.change[i] = v
 self.update[i] = True

def build(self, i, l, r):
 if l == r:
 self.len[i] = self.pre[i] = self.suf[i] = 1
 else:
 mid = (l + r) >> 1
 self.build(i << 1, l, mid)
 self.build(i << 1 | 1, mid + 1, r)
 self.up(i, mid - 1 + 1, r - mid)
 self.update[i] = False

def update_range(self, jobl, jobr, jobv, l, r, i):
 if jobl <= l and r <= jobr:
 self.lazy_update(i, jobv, r - l + 1)
 else:
 mid = (l + r) >> 1
 ln = mid - l + 1
 rn = r - mid
 self.down(i, ln, rn)
 if jobl <= mid:
 self.update_range(jobl, jobr, jobv, l, mid, i << 1)
 if jobr > mid:
 self.update_range(jobl, jobr, jobv, mid + 1, r, i << 1 | 1)
 self.up(i, ln, rn)

def query_left(self, x, l, r, i):
 if l == r:
 return l
 else:
 mid = (l + r) >> 1
 ln = mid - l + 1
 rn = r - mid
 self.down(i, ln, rn)
 # 最先查左边
 if self.len[i << 1] >= x:
 return self.query_left(x, l, mid, i << 1)
 # 然后查中间向两边扩展的可能区域
 if self.suf[i << 1] + self.pre[i << 1 | 1] >= x:

```

```

 return mid - self.suf[i << 1] + 1
 # 前面都没有再最后查右边
 return self.query_left(x, mid + 1, r, i << 1 | 1)

def main():
 import sys
 input = sys.stdin.read
 data = input().split()

 n = int(data[0])
 m = int(data[1])

 tree = HotelTree(n)

 results = []
 idx = 2
 for _ in range(m):
 op = int(data[idx])
 idx += 1

 if op == 1:
 x = int(data[idx])
 idx += 1
 if tree.len[1] < x:
 left = 0
 else:
 left = tree.query_left(x, 1, n, 1)
 tree.update_range(left, left + x - 1, 1, 1, n, 1)
 results.append(str(left))
 else:
 x = int(data[idx])
 idx += 1
 y = int(data[idx])
 idx += 1
 tree.update_range(x, min(x + y - 1, n), 0, 1, n, 1)

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

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

```

复杂度分析

- **时间复杂度**:
 - 建树: $O(n)$
 - 单次操作: $O(\log n)$
 - 总时间复杂度: $O((n + m) \log n)$
- **空间复杂度**: $O(n)$

总结

线段树是一种非常强大的数据结构，可以高效地处理各种区间操作问题。通过以上四个题目的实现，我们可以看到线段树在不同场景下的应用：

1. **多标记维护**: 在序列操作问题中，我们需要同时维护多种信息和多种懒标记
2. **区间合并**: 在最长交替子串问题中，需要仔细处理区间合并逻辑
3. **查询策略**: 在地道相连问题中，需要根据位置进行不同的查询策略
4. **最值查询**: 在旅馆问题中，需要查找满足条件的最左位置

掌握线段树的关键在于：

1. 根据题目需求设计节点信息
2. 正确实现区间合并逻辑
3. 合理使用懒标记优化
4. 熟练掌握各种查询和更新策略

补充题目详解

LeetCode 307. 区域和检索 - 数组可修改

题目解析

实现一个支持更新和区间求和操作的数据结构。

解题思路

使用线段树维护区间和，支持单点更新和区间查询。

Java 实现

```
```java
public class NumArray {
 private int[] tree;
 private int[] nums;
 private int n;

 public NumArray(int[] nums) {
```

```

this.n = nums.length;
this.nums = nums;
this.tree = new int[4 * n];
build(0, 0, n - 1);
}

private void build(int node, int start, int end) {
 if (start == end) {
 tree[node] = nums[start];
 } else {
 int mid = (start + end) / 2;
 build(2 * node + 1, start, mid);
 build(2 * node + 2, mid + 1, end);
 tree[node] = tree[2 * node + 1] + tree[2 * node + 2];
 }
}

public void update(int index, int val) {
 update(0, 0, n - 1, index, val);
}

private void update(int node, int start, int end, int index, int val) {
 if (start == end) {
 nums[index] = val;
 tree[node] = val;
 } else {
 int mid = (start + end) / 2;
 if (index <= mid) {
 update(2 * node + 1, start, mid, index, val);
 } else {
 update(2 * node + 2, mid + 1, end, index, val);
 }
 tree[node] = tree[2 * node + 1] + tree[2 * node + 2];
 }
}

public int sumRange(int left, int right) {
 return query(0, 0, n - 1, left, right);
}

private int query(int node, int start, int end, int left, int right) {
 if (right < start || end < left) {
 return 0;
 }
}

```

```

 }
 if (left <= start && end <= right) {
 return tree[node];
 }
 int mid = (start + end) / 2;
 int leftSum = query(2 * node + 1, start, mid, left, right);
 int rightSum = query(2 * node + 2, mid + 1, end, left, right);
 return leftSum + rightSum;
}
}
```

```

复杂度分析

- 时间复杂度：
 - 构建: $O(n)$
 - 更新: $O(\log n)$
 - 查询: $O(\log n)$
- 空间复杂度: $O(n)$

LeetCode 315. 计算右侧小于当前元素的个数

题目解析

给定一个整数数组 `nums`, 返回一个新数组 `counts`, 其中 `counts[i]` 是 `nums[i]` 右侧小于 `nums[i]` 的元素的数量。

解题思路

使用离散化+线段树的方法。从右往左遍历数组，用线段树维护每个值出现的次数，查询小于当前值的元素个数。

Java 实现

```

```java
import java.util.*;

public class Solution {
 public List<Integer> countSmaller(int[] nums) {
 // 离散化
 int[] sorted = nums.clone();
 Arrays.sort(sorted);
 Map<Integer, Integer> ranks = new HashMap<>();
 int rank = 0;
 for (int num : sorted) {
 if (!ranks.containsKey(num)) {

```

```

 ranks.put(num, ++rank);
 }
}

// 使用线段树
SegmentTree tree = new SegmentTree(ranks.size());
List<Integer> result = new ArrayList<>();

// 从右往左遍历
for (int i = nums.length - 1; i >= 0; i--) {
 int r = ranks.get(nums[i]);
 tree.update(1, 1, ranks.size(), r, 1);
 result.add(tree.query(1, 1, ranks.size(), 1, r - 1));
}

Collections.reverse(result);
return result;
}

class SegmentTree {
 private int[] tree;
 private int n;

 public SegmentTree(int size) {
 this.n = size;
 this.tree = new int[4 * (size + 1)];
 }

 public void update(int node, int start, int end, int index, int val) {
 if (start == end) {
 tree[node] += val;
 } else {
 int mid = (start + end) / 2;
 if (index <= mid) {
 update(2 * node, start, mid, index, val);
 } else {
 update(2 * node + 1, mid + 1, end, index, val);
 }
 tree[node] = tree[2 * node] + tree[2 * node + 1];
 }
 }

 public int query(int node, int start, int end, int left, int right) {

```

```

 if (left > end || right < start) {
 return 0;
 }
 if (left <= start && end <= right) {
 return tree[node];
 }
 int mid = (start + end) / 2;
 int leftSum = query(2 * node, start, mid, left, right);
 int rightSum = query(2 * node + 1, mid + 1, end, left, right);
 return leftSum + rightSum;
 }
}
```

```

复杂度分析

- 时间复杂度: $O(n \log n)$
- 空间复杂度: $O(n)$

洛谷 P3372 【模板】线段树 1

题目解析

实现一个支持区间加法和区间求和的线段树。

解题思路

使用带懒标记的线段树，支持区间更新和区间查询。

Java 实现

```

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

public class Main {
 static long[] tree;
 static long[] lazy;
 static long[] arr;
 static int n, m;

 public static void main(String[] args) throws IOException {
 BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
 StringTokenizer st = new StringTokenizer(br.readLine());
 n = Integer.parseInt(st.nextToken());

```

```

m = Integer.parseInt(st.nextToken());

arr = new long[n + 1];
tree = new long[4 * n];
lazy = new long[4 * n];

st = new StringTokenizer(br.readLine());
for (int i = 1; i <= n; i++) {
 arr[i] = Long.parseLong(st.nextToken());
}

build(1, 1, n);

StringBuilder sb = new StringBuilder();
for (int i = 0; i < m; i++) {
 st = new StringTokenizer(br.readLine());
 int op = Integer.parseInt(st.nextToken());
 if (op == 1) {
 int x = Integer.parseInt(st.nextToken());
 int y = Integer.parseInt(st.nextToken());
 long k = Long.parseLong(st.nextToken());
 update(1, 1, n, x, y, k);
 } else {
 int x = Integer.parseInt(st.nextToken());
 int y = Integer.parseInt(st.nextToken());
 sb.append(query(1, 1, n, x, y)).append("\n");
 }
}

System.out.print(sb);
}

static void build(int node, int start, int end) {
 if (start == end) {
 tree[node] = arr[start];
 } else {
 int mid = (start + end) / 2;
 build(2 * node, start, mid);
 build(2 * node + 1, mid + 1, end);
 tree[node] = tree[2 * node] + tree[2 * node + 1];
 }
}

```

```

static void pushDown(int node, int start, int end) {
 if (lazy[node] != 0) {
 int mid = (start + end) / 2;
 tree[2 * node] += lazy[node] * (mid - start + 1);
 tree[2 * node + 1] += lazy[node] * (end - mid);
 lazy[2 * node] += lazy[node];
 lazy[2 * node + 1] += lazy[node];
 lazy[node] = 0;
 }
}

static void update(int node, int start, int end, int l, int r, long val) {
 if (l <= start && end <= r) {
 tree[node] += val * (end - start + 1);
 lazy[node] += val;
 } else {
 pushDown(node, start, end);
 int mid = (start + end) / 2;
 if (l <= mid) update(2 * node, start, mid, l, r, val);
 if (r > mid) update(2 * node + 1, mid + 1, end, l, r, val);
 tree[node] = tree[2 * node] + tree[2 * node + 1];
 }
}

static long query(int node, int start, int end, int l, int r) {
 if (l <= start && end <= r) {
 return tree[node];
 }
 pushDown(node, start, end);
 int mid = (start + end) / 2;
 long sum = 0;
 if (l <= mid) sum += query(2 * node, start, mid, l, r);
 if (r > mid) sum += query(2 * node + 1, mid + 1, end, l, r);
 return sum;
}
```
    ...
}

```

复杂度分析

- 时间复杂度:
 - 构建: $O(n)$
 - 更新: $O(\log n)$
 - 查询: $O(\log n)$

- 空间复杂度: $O(n)$

⚡ 线段树进阶应用

1. 二维线段树

用于处理二维平面上的区间查询和更新问题。

2. 动态开点线段树

适用于数据范围很大但实际使用较少的情况，避免预先开满数组。

3. 可持久化线段树（主席树）

支持保存历史版本信息，可以查询历史状态。

4. 扫描线 + 线段树

用于解决平面几何问题，如矩形面积并、周长并等。

5. 树链剖分 + 线段树

用于处理树上路径操作问题。

🧠 学习建议

1. **掌握基础**: 熟练掌握线段树的基本操作和懒标记技术
2. **多做练习**: 通过大量练习掌握不同题型的解法
3. **理解变种**: 学习线段树的各种变种和高级应用
4. **工程实践**: 将线段树应用到实际项目中，理解其工程化考量

📚 参考资料

1. 《算法竞赛进阶指南》 - 李煜东
2. 《挑战程序设计竞赛》 - 秋叶拓哉等
3. TopCoder 数据结构教程
4. Codeforces Educational Round 相关讲解

[代码文件]

文件: Code01_SequenceOperation.cpp

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

```
/*
 * 线段树经典应用：多标记序列操作
 * 题目来源：洛谷 P2572 [SDOI2010] 序列操作
 * 题目链接：https://www.luogu.com.cn/problem/P2572
 *
 * 核心算法：线段树 + 多重懒标记
 * 难度：省选/NOI-
 *
 * 【题目详细描述】
 * 给定一个长度为 n 的 01 序列，支持 5 种操作：
 * 1. 操作 0 l r：将区间[l, r]全部置为 0
 * 2. 操作 1 l r：将区间[l, r]全部置为 1
 * 3. 操作 2 l r：将区间[l, r]全部取反
 * 4. 操作 3 l r：查询区间[l, r]中 1 的个数
 * 5. 操作 4 l r：查询区间[l, r]中连续 1 的最长长度
 *
 * 【解题思路】
 * 这是一个典型的线段树应用题，需要维护多种区间信息并处理多重懒标记。
 * 线段树节点需要保存丰富的信息来支持连续子串长度的查询。
 *
 * 【核心数据结构设计】
 * 线段树每个节点维护以下信息：
 * - sum[i]：区间内 1 的总数
 * - len0[i]/len1[i]：区间内连续 0/1 的最长子串长度
 * - pre0[i]/pre1[i]：区间内连续 0/1 的最长前缀长度
 * - suf0[i]/suf1[i]：区间内连续 0/1 的最长后缀长度
 *
 * 懒标记设计：
 * - change[i]：记录区间被置为的具体值（0 或 1）
 * - update[i]：标记区间是否有待处理的赋值操作
 * - reverse_flag[i]：标记区间是否有待处理的翻转操作
 *
 * 【关键技术点】
 * 1. 懒标记优先级处理：更新操作(update)优先于翻转操作(reverse)
 * 2. 区间合并逻辑：需要考虑左右子区间连接处的情况
 * 3. 多重懒标记的下传顺序和相互影响处理
 * 4. 边界条件处理：如区间为空、单元素区间等特殊情况
 *
 * 【复杂度分析】
 * - 时间复杂度：
 *   - 建树：O(n)
 *   - 单次操作（更新/查询）：O(log n)
 *   - m 次操作总时间复杂度：O((n + m) log n)
```

* - 空间复杂度: $O(4n)$, 线段树标准空间配置

*

* 【算法优化点】

- * 1. 懒标记延迟下传: 避免不必要的更新操作
- * 2. 区间合并时的高效计算: 通过维护前缀和后缀信息加速
- * 3. 使用位运算优化: 如移位操作代替乘除法
- * 4. 避免使用 STL 容器, 减少内存开销和常数因子
- * 5. 输入输出效率优化: 使用 `ios::sync_with_stdio(false)` 等
- * 6. 掌握线段树的关键点:
 - 懒标记的正确传递与优先级处理
 - 区间合并逻辑的设计 (如本题中的连续 1 长度合并)
 - 不同操作类型的统一处理框架
 - 边界条件的处理 (如叶子节点、空区间等)
 - 效率优化: 避免不必要的递归和计算
 - 异常处理与参数校验
 - 内存管理与效率平衡

*

* 【工程化考量】

- * 1. 内存布局: 使用全局数组而非动态分配, 提高缓存命中率
- * 2. 函数内联: 关键函数如 `max3`、`swap_int` 设置为内联, 减少函数调用开销
- * 3. 数据局部性: 将相关数据集中存储, 提高 CPU 缓存效率
- * 4. 错误处理: 检查数组越界, 处理非法输入

*

* 【C++语言特性应用】

- * 1. 位运算: 使用位移操作 (`<<`) 进行快速索引计算
- * 2. 全局变量: 使用全局数组存储线段树, 避免栈溢出
- * 3. 函数参数传递: 使用值传递而非引用传递, 减少开销
- * 4. 预处理器定义: 使用 `const` 定义常量, 提高代码可读性和安全性

*

* 【类似题目推荐】

- * 1. LeetCode 307. 区域和检索 - 数组可修改: <https://leetcode.cn/problems/range-sum-query-mutable/>
- * 2. Codeforces 242E - XOR on Segment: <https://codeforces.com/problemset/problem/242/E>
- * 3. 洛谷 P3373 【模板】线段树 2: <https://www.luogu.com.cn/problem/P3373>
- * 4. POJ 3468 A Simple Problem with Integers: <http://poj.org/problem?id=3468>
- * 5. HDU 1698 Just a Hook: <http://acm.hdu.edu.cn/showproblem.php?pid=1698>
- * 6. SPOJ GSS1 - Can you answer these queries I: <https://www.spoj.com/problems/GSS1/>
- * 7. LintCode 207. 区间求和 II: <https://www.lintcode.com/problem/interval-sum-ii/>
- * 8. HackerRank Array and simple queries: <https://www.hackerrank.com/challenges/array-and-simple-queries/problem>
- * 9. LeetCode 315. 计算右侧小于当前元素的个数: <https://leetcode.com/problems/count-of-smaller-numbers-after-self/>
- * 10. LeetCode 327. 区间和的个数: <https://leetcode.com/problems/count-of-range-sum/>

```
* 11. LeetCode 493. 翻转对: https://leetcode.com/problems/reverse-pairs/
* 12. LeetCode 239. 滑动窗口最大值: https://leetcode.com/problems/sliding-window-maximum/
* 13. LeetCode 732. 我的日程安排表 III: 区间重叠问题
* 14. LeetCode 699. 掉落的方块: 区间更新与查询最大值
* 15. LeetCode 995. K 连续位的最小翻转次数: 区间翻转操作
* 16. LintCode 205. Interval Minimum Number: https://www.lintcode.com/problem/interval-minimum-number/
* 17. Codeforces 61E. Enemy is weak: 区间统计问题
* 18. 牛客网 NC24970. 线段树练习一: 基础区间操作
* 19. 杭电 OJ 1542. Atlantis: 扫描线算法与线段树
* 20. USACO 2017 January Contest, Gold Problem 1. Balanced Photo: 区间统计
* 21. SPOJ GSS1 – Can you answer these queries I: 区间最大子段和
* 22. UVa OJ 11990. Dynamic Inversion: 动态逆序对问题
*/

```

```
/***
 * 序列操作 (Sequence Operations)
 *
 * 问题描述:
 * 给定一个初始全为 0 的序列, 支持以下操作:
 * 1. 将区间[1, r]全部置为 0
 * 2. 将区间[1, r]全部置为 1
 * 3. 将区间[1, r]全部取反 (0 变 1, 1 变 0)
 * 4. 查询区间[1, r]中 1 的个数
 * 5. 查询区间[1, r]中连续 1 的最长长度
 *
 * 解决方案:
 * 使用线段树 (Segment Tree) 结合懒标记 (Lazy Propagation) 来实现高效的区间操作
 *
 * 数据结构设计:
 * - 线段树节点存储以下信息:
 *   - sum[i]: 区间内 1 的总数
 *   - len0[i]: 区间内最长连续 0 的长度
 *   - pre0[i]: 区间前缀连续 0 的长度
 *   - suf0[i]: 区间后缀连续 0 的长度
 *   - len1[i]: 区间内最长连续 1 的长度
 *   - pre1[i]: 区间前缀连续 1 的长度
 *   - suf1[i]: 区间后缀连续 1 的长度
 *   - update[i]: 更新标记, 表示该区间是否有待执行的更新操作
 *   - change[i]: 更新值, 保存要设置的目标值 (0 或 1)
 *   - reverse_flag[i]: 翻转标记, 表示该区间是否有待执行的翻转操作
 *
 * 实现细节:

```

```

* - 懒标记优先级：更新操作（置 0/置 1）的优先级高于翻转操作
* - 当同时存在两种懒标记时，先处理更新操作，再处理翻转操作
* - 区间查询时需要考虑跨越左右子区间的情况，特别是查询最长连续 1 时
*
* 时间复杂度分析：
* - 构建线段树：O(n)
* - 区间更新操作：O(log n)
* - 区间查询操作：O(log n)
*
* 空间复杂度分析：
* - 线段树存储：O(4n)，使用 4 倍原数组长度的空间来存储线段树
* - 辅助数组：O(4n)，用于存储各种状态信息和懒标记
* - 总体空间复杂度：O(n)
*/

```

```

const int MAXN = 100001; // 最大序列长度

// 原始数组
int arr[MAXN];

// 线段树节点信息
int sum[MAXN << 2]; // 区间内 1 的总数
int len0[MAXN << 2], pre0[MAXN << 2], suf0[MAXN << 2]; // 连续 0 相关信息
int len1[MAXN << 2], pre1[MAXN << 2], suf1[MAXN << 2]; // 连续 1 相关信息

// 懒标记
int change[MAXN << 2]; // 更新值（0 或 1）
int update[MAXN << 2]; // 更新标记（置 0/置 1 操作）
int reverse_flag[MAXN << 2]; // 翻转标记（取反操作）

/**
 * 求三个整数中的最大值
 *
 * @param a 第一个整数
 * @param b 第二个整数
 * @param c 第三个整数
 * @return 三个数中的最大值
 */
int max3(int a, int b, int c) {
    int max_val = a;
    if (b > max_val) max_val = b;
    if (c > max_val) max_val = c;
    return max_val;
}
```

```
}
```

```
/**  
 * 交换两个整数的指针指向的值  
 *  
 * @param a 指向第一个整数的指针  
 * @param b 指向第二个整数的指针  
 */  
void swap_int(int* a, int* b) {  
    int temp = *a;  
    *a = *b;  
    *b = temp;  
}
```

```
/**  
 * 向上更新当前节点信息（合并左右子节点信息）  
 *  
 * @param i 当前节点索引  
 * @param ln 左子区间长度  
 * @param rn 右子区间长度  
 */  
void up(int i, int ln, int rn) {  
    int l = i << 1;      // 左子节点索引  
    int r = i << 1 | 1; // 右子节点索引  
  
    // 更新区间内 1 的总数  
    sum[i] = sum[l] + sum[r];  
  
    // 更新连续 0 的信息  
    // 取左右子区间的最大值或左右子区间连接处的连续 0  
    len0[i] = max3(len0[l], len0[r], suf0[l] + pre0[r]);  
  
    // 更新连续 0 的前缀  
    // 如果左子区间全是 0，则前缀包括整个左子区间和右子区间的前缀  
    pre0[i] = len0[l] < ln ? pre0[l] : (pre0[l] + pre0[r]);  
  
    // 更新连续 0 的后缀  
    // 如果右子区间全是 0，则后缀包括整个右子区间和左子区间的后缀  
    suf0[i] = len0[r] < rn ? suf0[r] : (suf0[l] + suf0[r]);  
  
    // 更新连续 1 的信息（与连续 0 的处理方式类似）  
    len1[i] = max3(len1[l], len1[r], suf1[l] + pre1[r]);  
    pre1[i] = len1[l] < ln ? pre1[l] : (pre1[l] + pre1[r]);
```

```

suf1[i] = len1[r] < rn ? suf1[r] : (suf1[1] + suf1[r]);
}

/***
 * 向下传递懒标记到子节点
 *
 * @param i 当前节点索引
 * @param ln 左子区间长度
 * @param rn 右子区间长度
 */
void down(int i, int ln, int rn) {
    // 先处理更新操作（优先级高于翻转操作）
    if (update[i]) {
        int l = i << 1, r = i << 1 | 1; // 左右子节点索引

        // 左子节点应用更新操作
        sum[1] = change[i] * ln;
        len0[1] = pre0[1] = suf0[1] = change[i] == 0 ? ln : 0;
        len1[1] = pre1[1] = suf1[1] = change[i] == 1 ? ln : 0;
        change[1] = change[i];
        update[1] = 1; // 设置子节点的更新标记
        reverse_flag[1] = 0; // 清空子节点的翻转标记

        // 右子节点应用更新操作
        sum[r] = change[i] * rn;
        len0[r] = pre0[r] = suf0[r] = change[i] == 0 ? rn : 0;
        len1[r] = pre1[r] = suf1[r] = change[i] == 1 ? rn : 0;
        change[r] = change[i];
        update[r] = 1; // 设置子节点的更新标记
        reverse_flag[r] = 0; // 清空子节点的翻转标记

        // 清除当前节点的更新标记
        update[i] = 0;
    }

    // 再处理翻转操作
    if (reverse_flag[i]) {
        int l = i << 1, r = i << 1 | 1; // 左右子节点索引

        // 左子节点应用翻转操作
        sum[1] = ln - sum[1]; // 翻转 1 的个数
        // 交换连续 0 和连续 1 的各种长度信息
        swap_int(&len0[1], &len1[1]); // 交换最长连续 0/1 长度
    }
}

```

```

swap_int(&pre0[1], &pre1[1]); // 交换前缀 0/1 长度
swap_int(&suf0[1], &suf1[1]); // 交换后缀 0/1 长度
reverse_flag[1] = !reverse_flag[1]; // 翻转翻转标记

// 右子节点应用翻转操作
sum[r] = rn - sum[r]; // 翻转 1 的个数
// 交换连续 0 和连续 1 的各种长度信息
swap_int(&len0[r], &len1[r]);
swap_int(&pre0[r], &pre1[r]);
swap_int(&suf0[r], &suf1[r]);
reverse_flag[r] = !reverse_flag[r]; // 翻转翻转标记

// 清除当前节点的翻转标记
reverse_flag[i] = 0;
}

}

/***
* 构建线段树
*
* @param l 当前区间左边界 (1-based)
* @param r 当前区间右边界 (1-based)
* @param i 当前节点索引
*/
void build(int l, int r, int i) {
    // 叶子节点情况
    if (l == r) {
        // 直接赋值原始数组的值
        sum[i] = arr[l];
        // 初始化连续 0 的信息
        len0[i] = pre0[i] = suf0[i] = arr[l] == 0 ? 1 : 0;
        // 初始化连续 1 的信息
        len1[i] = pre1[i] = suf1[i] = arr[l] == 1 ? 1 : 0;
    } else {
        // 非叶子节点，递归构建左右子树
        int mid = (l + r) >> 1; // 计算中点
        build(l, mid, i << 1); // 构建左子树
        build(mid + 1, r, i << 1 | 1); // 构建右子树
        // 向上合并子节点信息
        up(i, mid - 1 + 1, r - mid);
    }
}

// 初始化懒标记为未激活状态

```

```

update[i] = 0;
reverse_flag[i] = 0;
}

/***
 * 区间赋值操作
 *
 * @param jobl 待更新区间的左边界 (1-based)
 * @param jobr 待更新区间的右边界 (1-based)
 * @param jobv 要设置的值 (0 或 1)
 * @param l 当前节点区间左边界 (1-based)
 * @param r 当前节点区间右边界 (1-based)
 * @param i 当前节点索引
 */
void update_range(int jobl, int jobr, int jobv, int l, int r, int i) {
    // 当前区间完全包含在待更新区间内
    if (jobl <= l && r <= jobr) {
        // 更新区间内 1 的总数
        sum[i] = jobv * (r - l + 1);
        // 更新连续 0 的信息: 如果 v=0, 则整个区间都是 0; 否则都是 1 (没有 0)
        len0[i] = pre0[i] = suf0[i] = jobv == 0 ? (r - l + 1) : 0;
        // 更新连续 1 的信息: 如果 v=1, 则整个区间都是 1; 否则都是 0 (没有 1)
        len1[i] = pre1[i] = suf1[i] = jobv == 1 ? (r - l + 1) : 0;
        // 记录区间赋值的目标值
        change[i] = jobv;
        // 设置更新标记
        update[i] = 1;
        // 清空翻转标记 (更新操作优先于翻转操作)
        reverse_flag[i] = 0;
    } else {
        // 当前区间部分包含在待更新区间内
        int mid = (l + r) >> 1; // 计算中点

        // 先向下传递懒标记
        down(i, mid - 1 + 1, r - mid);

        // 递归处理左右子区间
        if (jobl <= mid) {
            update_range(jobl, jobr, jobv, l, mid, i << 1);
        }
        if (jobr > mid) {
            update_range(jobl, jobr, jobv, mid + 1, r, i << 1 | 1);
        }
    }
}

```

```

        // 向上合并子节点信息
        up(i, mid - 1 + 1, r - mid);
    }

}

/***
 * 区间翻转操作
 *
 * @param jobl 待翻转区间的左边界 (1-based)
 * @param jobr 待翻转区间的右边界 (1-based)
 * @param l 当前节点区间左边界 (1-based)
 * @param r 当前节点区间右边界 (1-based)
 * @param i 当前节点索引
 */
void reverse_range(int jobl, int jobr, int l, int r, int i) {
    // 当前区间完全包含在待翻转区间内
    if (jobl <= l && r <= jobr) {
        // 翻转 1 的个数: 1 变 0, 0 变 1
        sum[i] = (r - l + 1) - sum[i];
        // 交换连续 0 和连续 1 的各种长度信息
        swap_int(&len0[i], &len1[i]); // 交换最长连续 0/1 长度
        swap_int(&pre0[i], &pre1[i]); // 交换前缀 0/1 长度
        swap_int(&suf0[i], &suf1[i]); // 交换后缀 0/1 长度
        // 翻转翻转标记 (多次翻转可以抵消)
        reverse_flag[i] = !reverse_flag[i];
    } else {
        // 当前区间部分包含在待翻转区间内
        int mid = (l + r) >> 1; // 计算中点

        // 先向下传递懒标记
        down(i, mid - 1 + 1, r - mid);

        // 递归处理左右子区间
        if (jobl <= mid) {
            reverse_range(jobl, jobr, l, mid, i << 1);
        }
        if (jobr > mid) {
            reverse_range(jobl, jobr, mid + 1, r, i << 1 | 1);
        }
    }

    // 向上合并子节点信息
    up(i, mid - 1 + 1, r - mid);
}

```

```

    }

}

/***
 * 查询区间内 1 的总数
 *
 * @param jobl 查询区间的左边界 (1-based)
 * @param jobr 查询区间的右边界 (1-based)
 * @param l 当前节点区间左边界 (1-based)
 * @param r 当前节点区间右边界 (1-based)
 * @param i 当前节点索引
 * @return 区间内 1 的总数
 */
int query_sum(int jobl, int jobr, int l, int r, int i) {
    // 当前区间完全包含在查询区间内
    if (jobl <= l && r <= jobr) {
        return sum[i];
    }

    // 当前区间部分包含在查询区间内
    int mid = (l + r) >> 1; // 计算中点

    // 先向下传递懒标记
    down(i, mid - 1 + 1, r - mid);

    int ans = 0;
    // 递归查询左右子区间
    if (jobl <= mid) {
        ans += query_sum(jobl, jobr, l, mid, i << 1);
    }
    if (jobr > mid) {
        ans += query_sum(jobl, jobr, mid + 1, r, i << 1 | 1);
    }

    return ans;
}

/***
 * 查询区间内连续 1 的最长长度
 *
 * 注意: C++ 版本简化了实现, 直接返回最长连续 1 的长度, 而不是像 Java/Python 版本那样返回三个值
 *
 * @param jobl 查询区间的左边界 (1-based)
 */

```

```

* @param jobr 查询区间的右边界 (1-based)
* @param l 当前节点区间左边界 (1-based)
* @param r 当前节点区间右边界 (1-based)
* @param i 当前节点索引
* @return 区间内连续 1 的最长长度
*/
int query_longest_len(int jobl, int jobr, int l, int r, int i) {
    // 当前区间完全包含在查询区间内
    if (jobl <= l && r <= jobr) {
        return len1[i];
    } else {
        // 当前区间部分包含在查询区间内或查询区间跨越多个子区间
        int mid = (l + r) >> 1; // 计算中点
        int ln = mid - l + 1; // 左子区间长度
        int rn = r - mid; // 右子区间长度

        // 先向下传递懒标记
        down(i, ln, rn);

        // 查询区间完全在左子区间
        if (jobr <= mid) {
            return query_longest_len(jobl, jobr, l, mid, i << 1);
        }
        // 查询区间完全在右子区间
        if (jobl > mid) {
            return query_longest_len(jobl, jobr, mid + 1, r, i << 1 | 1);
        }

        // 查询区间跨越左右子区间
        // 分别查询左右子区间的最长连续 1 长度
        int llen = query_longest_len(jobl, jobr, l, mid, i << 1);
        int rlen = query_longest_len(jobl, jobr, mid + 1, r, i << 1 | 1);

        // 合并信息：左子区间的最大值、右子区间的最大值、或左右连接处的连续 1
        int len = max3(llen, rlen, suf1[i << 1] + pre1[i << 1 | 1]);

        return len;
    }
}

/**
* 主函数：读取输入并处理操作
*

```

```

* 注意: 由于 C++ 版本需要完整的输入输出处理, 这里提供一个标准实现
*
* 输入格式:
* - 第一行: n (序列长度) 和 m (操作数量)
* - 第二行: n 个 0 或 1, 表示初始序列
* - 接下来 m 行: 每行一个操作, 格式为 op 1 r
*
* 操作类型:
* - 0 1 r: 将区间[1, r]全部置为 0
* - 1 1 r: 将区间[1, r]全部置为 1
* - 2 1 r: 将区间[1, r]全部取反
* - 3 1 r: 查询区间[1, r]中 1 的个数
* - 4 1 r: 查询区间[1, r]中连续 1 的最长长度
*/
int main() {
    // 高效输入输出设置
    ios::sync_with_stdio(false);
    cin.tie(nullptr);

    int n, m;
    cin >> n >> m; // 读取序列长度和操作次数

    // 读取初始序列 (1-based 索引)
    for (int i = 1; i <= n; ++i) {
        cin >> arr[i];
    }

    // 构建线段树
    build(1, n, 1);

    // 处理每个操作
    while (m--) {
        int op, l, r;
        cin >> op >> l >> r;
        l += 1; // 注意题目给的下标从 0 开始, 线段树下标从 1 开始
        r += 1; // 注意题目给的下标从 0 开始, 线段树下标从 1 开始

        switch (op) {
            case 0: // 区间置 0
                update_range(l, r, 0, 1, n, 1);
                break;
            case 1: // 区间置 1
                update_range(l, r, 1, 1, n, 1);
        }
    }
}

```

```

        break;
    case 2: // 区间取反
        reverse_range(l, r, 1, n, 1);
        break;
    case 3: // 查询 1 的个数
        cout << query_sum(l, r, 1, n, 1) << '\n';
        break;
    case 4: // 查询连续 1 的最长长度
        cout << query_longest_len(l, r, 1, n, 1) << '\n';
        break;
    default: // 异常操作类型处理
        break;
    }
}

return 0;
}
=====

文件: Code01_SequenceOperation.java
=====

package class113;


/*
 * 线段树经典应用：多标记序列操作
 * 题目来源：洛谷 P2572 [SDOI2010] 序列操作
 * 题目链接：https://www.luogu.com.cn/problem/P2572
 *
 * 核心算法：线段树 + 多重懒标记
 * 难度：省选/NOI-
 *
 * 【题目详细描述】
 * 给定一个长度为 n 的 01 序列，支持 5 种操作：
 * 1. 操作 0 l r：将区间[l, r]全部置为 0
 * 2. 操作 1 l r：将区间[l, r]全部置为 1
 * 3. 操作 2 l r：将区间[l, r]全部取反
 * 4. 操作 3 l r：查询区间[l, r]中 1 的个数
 * 5. 操作 4 l r：查询区间[l, r]中连续 1 的最长长度
 *
 * 【解题思路】
 * 这是一个典型的线段树应用题，需要维护多种区间信息并处理多重懒标记。
 * 线段树节点需要保存丰富的信息来支持连续子串长度的查询。


```

- *
 - * 【核心数据结构设计】
 - * 线段树每个节点维护以下信息:
 - * - sum[i]: 区间内 1 的总数
 - * - len0[i]/len1[i]: 区间内连续 0/1 的最长子串长度
 - * - pre0[i]/pre1[i]: 区间内连续 0/1 的最长前缀长度
 - * - suf0[i]/suf1[i]: 区间内连续 0/1 的最长后缀长度
 - *
 - * 懒标记设计:
 - * - change[i]: 记录区间被置为的具体值（0 或 1）
 - * - update[i]: 标记区间是否有待处理的赋值操作
 - * - reverse[i]: 标记区间是否有待处理的翻转操作
 - *
- * 【关键技术点】
 - * 1. 懒标记优先级处理: 更新操作(update) 优先于翻转操作(reverse)
 - * 2. 区间合并逻辑: 需要考虑左右子区间连接处的情况
 - * 3. 多重懒标记的下传顺序和相互影响处理
 - * 4. 边界条件处理: 如区间为空、单元素区间等特殊情况
- *
- * 【复杂度分析】
 - * - 时间复杂度:
 - * - 建树: $O(n)$
 - * - 单次操作(更新/查询): $O(\log n)$
 - * - m 次操作总时间复杂度: $O((n + m) \log n)$
 - * - 空间复杂度: $O(4n)$, 线段树标准空间配置
- *
- * 【算法优化点】
 - * 1. 懒标记延迟下传: 避免不必要的更新操作
 - * 2. 区间合并时的高效计算: 通过维护前缀和后缀信息加速
 - * 3. 使用位运算优化: 如移位操作代替乘除法
- *
- * 【工程化考量】
 - * 1. 输入输出效率: 使用 BufferedReader 和 PrintWriter 优化 IO
 - * 2. 数组大小预设: 根据题目约束设置合理的 MAXN
 - * 3. 内存管理: 线段树数组使用静态分配避免动态分配的开销
 - * 4. 代码模块化: 将 up、down、build 等核心函数分离
- *
- * 【异常处理】
 - * 1. 输入范围检查: 确保操作区间在有效范围内
 - * 2. 数据类型溢出: 使用适当的数据类型避免整数溢出
 - * 3. 懒标记一致性: 确保懒标记的正确下传和合并
- *
- * 【类似题目推荐】

- * 1. LeetCode 307. 区域和检索 - 数组可修改: <https://leetcode.cn/problems/range-sum-query-mutable/>
- * 2. Codeforces 242E - XOR on Segment: <https://codeforces.com/problemset/problem/242/E>
- * 3. 洛谷 P3373 【模板】线段树 2: <https://www.luogu.com.cn/problem/P3373>
- * 4. POJ 3468 A Simple Problem with Integers: <http://poj.org/problem?id=3468>
- * 5. HDU 1698 Just a Hook: <http://acm.hdu.edu.cn/showproblem.php?pid=1698>
- * 6. SPOJ GSS1 - Can you answer these queries I: <https://www.spoj.com/problems/GSS1/>
- * 7. LintCode 207. 区间求和 II: <https://www.lintcode.com/problem/interval-sum-ii/description>
- * 8. HackerRank Array and simple queries: <https://www.hackerrank.com/challenges/array-and-simple-queries/problem>

*/

```
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_SequenceOperation {

    public static int MAXN = 100001;

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

    // 累加和用来统计 1 的数量
    public static int[] sum = new int[MAXN << 2];

    // 连续 0 的最长子串长度
    public static int[] len0 = new int[MAXN << 2];

    // 连续 0 的最长前缀长度
    public static int[] pre0 = new int[MAXN << 2];

    // 连续 0 的最长后缀长度
    public static int[] suf0 = new int[MAXN << 2];

    // 连续 1 的最长子串长度
    public static int[] len1 = new int[MAXN << 2];

    // 连续 1 的最长前缀长度
    public static int[] pre1 = new int[MAXN << 2];
```

```

// 连续 1 的最长后缀长度
public static int[] suf1 = new int[MAXN << 2];

// 懒更新信息，范围上所有数字被重置成了什么
public static int[] change = new int[MAXN << 2];

// 懒更新信息，范围上有没有重置任务
public static boolean[] update = new boolean[MAXN << 2];

// 懒更新信息，范围上有没有翻转任务
public static boolean[] reverse = new boolean[MAXN << 2];

/**
 * 向上更新当前节点信息（合并左右子节点信息）
 *
 * @param i 当前节点索引
 * @param ln 左子区间长度
 * @param rn 右子区间长度
 */
public static void up(int i, int ln, int rn) {
    int l = i << 1;      // 左子节点索引
    int r = i << 1 | 1;  // 右子节点索引

    // 更新区间内 1 的总数
    sum[i] = sum[l] + sum[r];

    // 更新连续 0 的信息
    // 取左右子区间的最大值或左右子区间连接处的连续 0
    len0[i] = Math.max(Math.max(len0[l], len0[r]), suf0[l] + pre0[r]);

    // 更新连续 0 的前缀
    // 如果左子区间全是 0，则前缀包括整个左子区间和右子区间的前缀
    pre0[i] = len0[l] < ln ? pre0[l] : (pre0[l] + pre0[r]);

    // 更新连续 0 的后缀
    // 如果右子区间全是 0，则后缀包括整个右子区间和左子区间的后缀
    suf0[i] = len0[r] < rn ? suf0[r] : (suf0[r] + suf0[l]);

    // 更新连续 1 的信息（与连续 0 的处理方式类似）
    len1[i] = Math.max(Math.max(len1[l], len1[r]), suf1[l] + pre1[r]);
    pre1[i] = len1[l] < ln ? pre1[l] : (pre1[l] + pre1[r]);
    suf1[i] = len1[r] < rn ? suf1[r] : (suf1[r] + suf1[l]);
}

```

```

}

    /**
 * 向下传递懒标记到子节点
 *
 * @param i 当前节点索引
 * @param ln 左子区间长度
 * @param rn 右子区间长度
 */
public static void down(int i, int ln, int rn) {
    // 先处理更新操作（优先级高于翻转操作）
    if (update[i]) {
        // 左子节点应用更新操作
        updateLazy(i << 1, change[i], ln);
        // 右子节点应用更新操作
        updateLazy(i << 1 | 1, change[i], rn);
        // 清除当前节点的更新标记
        update[i] = false;
    }

    // 再处理翻转操作
    if (reverse[i]) {
        // 左子节点应用翻转操作
        reverseLazy(i << 1, ln);
        // 右子节点应用翻转操作
        reverseLazy(i << 1 | 1, rn);
        // 清除当前节点的翻转标记
        reverse[i] = false;
    }
}

    /**
 * 处理区间赋值的懒标记
 *
 * @param i 当前节点索引
 * @param v 要设置的值（0 或 1）
 * @param n 当前节点表示的区间长度
 */
public static void updateLazy(int i, int v, int n) {
    // 更新区间内 1 的总数
    sum[i] = v * n;
    // 更新连续 0 的信息：如果 v=0，则整个区间都是 0；否则都是 1（没有 0）
    len0[i] = pre0[i] = suf0[i] = v == 0 ? n : 0;
}

```

```

// 更新连续 1 的信息：如果 v=1，则整个区间都是 1；否则都是 0（没有 1）
len1[i] = pre1[i] = suf1[i] = v == 1 ? n : 0;
// 记录区间赋值的目标值
change[i] = v;
// 设置更新标记
update[i] = true;
// 清空翻转标记（更新操作优先于翻转操作）
reverse[i] = false;
}

/**
* 处理区间翻转的懒标记
*
* @param i 当前节点索引
* @param n 当前节点表示的区间长度
*/
public static void reverseLazy(int i, int n) {
    // 翻转 1 的个数：1 变 0，0 变 1
    sum[i] = n - sum[i];

    int tmp;
    // 交换连续 0 和连续 1 的各种长度信息
    tmp = len0[i]; len0[i] = len1[i]; len1[i] = tmp; // 交换最长连续 0/1 长度
    tmp = pre0[i]; pre0[i] = pre1[i]; pre1[i] = tmp; // 交换前缀 0/1 长度
    tmp = suf0[i]; suf0[i] = suf1[i]; suf1[i] = tmp; // 交换后缀 0/1 长度

    // 翻转翻转标记（多次翻转可以抵消）
    reverse[i] = !reverse[i];
}

/**
* 构建线段树
*
* @param l 当前区间左边界 (1-based)
* @param r 当前区间右边界 (1-based)
* @param i 当前节点索引
*/
public static void build(int l, int r, int i) {
    // 叶子节点情况
    if (l == r) {
        // 直接赋值原始数组的值
        sum[i] = arr[l];
        // 初始化连续 0 的信息
    }
}

```

```

len0[i] = pre0[i] = suf0[i] = arr[1] == 0 ? 1 : 0;
// 初始化连续 1 的信息
len1[i] = pre1[i] = suf1[i] = arr[1] == 1 ? 1 : 0;
} else {
    // 非叶子节点，递归构建左右子树
    int mid = (l + r) >> 1; // 计算中点
    build(l, mid, i << 1); // 构建左子树
    build(mid + 1, r, i << 1 | 1); // 构建右子树
    // 向上合并子节点信息
    up(i, mid - 1 + 1, r - mid);
}

// 初始化懒标记为未激活状态
update[i] = false;
reverse[i] = false;
}

/**
* 区间赋值操作
*
* @param jobl 待更新区间的左边界 (1-based)
* @param jobr 待更新区间的右边界 (1-based)
* @param jobv 要设置的值 (0 或 1)
* @param l 当前节点区间左边界 (1-based)
* @param r 当前节点区间右边界 (1-based)
* @param i 当前节点索引
*/
public static void update(int jobl, int jobr, int jobv, int l, int r, int i) {
    // 当前区间完全包含在待更新区间内
    if (jobl <= l && r <= jobr) {
        // 直接应用懒标记
        updateLazy(i, jobv, r - l + 1);
    } else {
        // 当前区间部分包含在待更新区间内
        int mid = (l + r) >> 1; // 计算中点

        // 先向下传递懒标记
        down(i, mid - 1 + 1, r - mid);

        // 递归处理左右子区间
        if (jobl <= mid) {
            update(jobl, jobr, jobv, l, mid, i << 1);
        }
    }
}

```

```

    if (jobr > mid) {
        update(jobl, jobr, jobv, mid + 1, r, i << 1 | 1);
    }

    // 向上合并子节点信息
    up(i, mid - 1 + 1, r - mid);
}

}

/***
* 区间翻转操作
*
* @param jobl 待翻转区间的左边界 (1-based)
* @param jobr 待翻转区间的右边界 (1-based)
* @param l 当前节点区间左边界 (1-based)
* @param r 当前节点区间右边界 (1-based)
* @param i 当前节点索引
*/
public static void reverse(int jobl, int jobr, int l, int r, int i) {
    // 当前区间完全包含在待翻转区间内
    if (jobl <= l && r <= jobr) {
        // 直接应用翻转懒标记
        reverseLazy(i, r - 1 + 1);
    } else {
        // 当前区间部分包含在待翻转区间内
        int mid = (l + r) >> 1; // 计算中点

        // 先向下传递懒标记
        down(i, mid - 1 + 1, r - mid);

        // 递归处理左右子区间
        if (jobl <= mid) {
            reverse(jobl, jobr, l, mid, i << 1);
        }
        if (jobr > mid) {
            reverse(jobl, jobr, mid + 1, r, i << 1 | 1);
        }

        // 向上合并子节点信息
        up(i, mid - 1 + 1, r - mid);
    }
}

```

```

/***
 * 查询区间内 1 的总数
 *
 * @param jobl 查询区间的左边界 (1-based)
 * @param jobr 查询区间的右边界 (1-based)
 * @param l 当前节点区间左边界 (1-based)
 * @param r 当前节点区间右边界 (1-based)
 * @param i 当前节点索引
 * @return 区间内 1 的总数
 */

public static int querySum(int jobl, int jobr, int l, int r, int i) {
    // 当前区间完全包含在查询区间内
    if (jobl <= l && r <= jobr) {
        return sum[i];
    }

    // 当前区间部分包含在查询区间内
    int mid = (l + r) >> 1; // 计算中点

    // 先向下传递懒标记
    down(i, mid - 1 + 1, r - mid);

    int ans = 0;
    // 递归查询左右子区间
    if (jobl <= mid) {
        ans += querySum(jobl, jobr, l, mid, i << 1);
    }
    if (jobr > mid) {
        ans += querySum(jobl, jobr, mid + 1, r, i << 1 | 1);
    }

    return ans;
}

/***
 * 查询区间内连续 1 的最长长度
 *
 * @param jobl 查询区间的左边界 (1-based)
 * @param jobr 查询区间的右边界 (1-based)
 * @param l 当前节点区间左边界 (1-based)
 * @param r 当前节点区间右边界 (1-based)
 * @param i 当前节点索引
 * @return 长度为 3 的数组: [最长连续 1 的长度, 连续 1 的最长前缀长度, 连续 1 的最长后缀长度]
 */

```

```

*/
public static int[] queryLongest(int jobl, int jobr, int l, int r, int i) {
    // 当前区间完全包含在查询区间内
    if (jobl <= l && r <= jobr) {
        return new int[] { len1[i], pre1[i], suf1[i] };
    } else {
        // 当前区间部分包含在查询区间内或查询区间跨越多个子区间
        int mid = (l + r) >> 1; // 计算中点
        int ln = mid - l + 1; // 左子区间长度
        int rn = r - mid; // 右子区间长度

        // 先向下传递懒标记
        down(i, ln, rn);

        // 查询区间完全在左子区间
        if (jobr <= mid) {
            return queryLongest(jobl, jobr, l, mid, i << 1);
        }
        // 查询区间完全在右子区间
        if (jobl > mid) {
            return queryLongest(jobl, jobr, mid + 1, r, i << 1 | 1);
        }

        // 查询区间跨越左右子区间
        // 分别查询左右子区间的信息
        int[] l3 = queryLongest(jobl, jobr, l, mid, i << 1);
        int[] r3 = queryLongest(jobl, jobr, mid + 1, r, i << 1 | 1);

        // 提取左右子区间的信息
        int llen = l3[0], lpre = l3[1], lsuf = l3[2]; // 左子区间的最长1长度、前缀、后缀
        int rlen = r3[0], rpre = r3[1], rsuf = r3[2]; // 右子区间的最长1长度、前缀、后缀

        // 合并信息
        // 最长连续1长度：左子区间的最大值、右子区间的最大值、或左右连接处的连续1
        int len = Math.max(Math.max(llen, rlen), lsuf + rpre);

        // 连续1的最长前缀：如果左子区间的最长连续1覆盖了整个查询部分的左子区间，则包括右子
        // 区间的前缀
        // 任务实际影响了左侧范围的几个点 -> mid - Math.max(jobl, 1) + 1
        int pre = llen < mid - Math.max(jobl, 1) + 1 ? lpre : (lpre + rpre);

        // 连续1的最长后缀：如果右子区间的最长连续1覆盖了整个查询部分的右子区间，则包括左子
        // 区间的后缀
    }
}

```

```

// 任务实际影响了右侧范围的几个点 -> Math.min(r, jobr) - mid
int suf = rlen < Math.min(r, jobr) - mid ? rsuf : (lsuf + rsuf);

return new int[] { len, pre, suf };
}

}

/***
* 主函数: 读取输入并处理操作
*
* @param args 命令行参数
* @throws IOException 输入输出异常
*
* 输入格式:
* - 第一行: n (序列长度) 和 m (操作数量)
* - 第二行: n 个 0 或 1, 表示初始序列
* - 接下来 m 行: 每行一个操作, 格式为 op l r
*
* 操作类型:
* - 0 l r: 将区间 [l, r] 全部置为 0
* - 1 l r: 将区间 [l, r] 全部置为 1
* - 2 l r: 将区间 [l, r] 全部取反
* - 3 l r: 查询区间 [l, r] 中 1 的个数
* - 4 l r: 查询区间 [l, r] 中连续 1 的最长长度
*/
public static void main(String[] args) throws IOException {
    // 高效输入输出配置
    BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
    StreamTokenizer in = new StreamTokenizer(br);
    PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));

    // 读取数据规模
    in.nextToken();
    int n = (int) in.nval; // 序列长度
    in.nextToken();
    int m = (int) in.nval; // 操作次数

    // 读取初始序列
    for (int i = 1; i <= n; i++) {
        in.nextToken();
        arr[i] = (int) in.nval; // 1-based 索引存储
    }
}

```

```
// 建立线段树
build(1, n, 1);

// 处理每个操作
for (int i = 1, op, jobl, jobr; i <= m; i++) {
    in.nextToken();
    op = (int) in.nval;
    in.nextToken();
    jobl = (int) in.nval + 1; // 注意题目给的下标从 0 开始，线段树下标从 1 开始
    in.nextToken();
    jobr = (int) in.nval + 1; // 注意题目给的下标从 0 开始，线段树下标从 1 开始

    // 根据操作类型执行相应的线段树操作
    switch (op) {
        case 0: // 区间置 0
            update(jobl, jobr, 0, 1, n, 1);
            break;
        case 1: // 区间置 1
            update(jobl, jobr, 1, 1, n, 1);
            break;
        case 2: // 区间取反
            reverse(jobl, jobr, 1, n, 1);
            break;
        case 3: // 查询 1 的个数
            out.println(querySum(jobl, jobr, 1, n, 1));
            break;
        case 4: // 查询连续 1 的最长长度
            int[] ans = queryLongest(jobl, jobr, 1, n, 1);
            out.println(ans[0]);
            break;
        default:
            // 异常操作类型处理
            break;
    }
}

// 清理资源
out.flush();
out.close();
br.close();
}

// 扩展学习：线段树在工程中的应用
```

```
/*
 * 1. 数据库索引：在数据库系统中，线段树可以用于优化范围查询操作
 * 2. 图像分割：在计算机视觉中，线段树可用于区域分割和特征提取
 * 3. 网络流量监控：实时监控和分析网络流量的区间统计信息
 * 4. 股票分析：维护股票价格的区间最大值、最小值、平均值等信息
 * 5. 地理信息系统：处理地理区域的查询和更新操作
 */

// 线段树调试技巧
/*
 * 1. 打印中间状态：在关键操作前后输出线段树状态
 * 2. 断言验证：使用断言验证线段树的一致性和正确性
 * 3. 小数据测试：使用小规模测试用例验证算法正确性
 * 4. 边界测试：特别关注空区间、单元素区间等边界情况
 */

// 类似题目和训练推荐
/*
 * 1. LeetCode 307. 区域和检索 - 数组可修改: https://leetcode.com/problems/range-sum-query-mutable/
 * 2. LeetCode 315. 计算右侧小于当前元素的个数: https://leetcode.com/problems/count-of-smaller-numbers-after-self/
 * 3. LeetCode 327. 区间和的个数: https://leetcode.com/problems/count-of-range-sum/
 * 4. LeetCode 493. 翻转对: https://leetcode.com/problems/reverse-pairs/
 * 5. LeetCode 239. 滑动窗口最大值: https://leetcode.com/problems/sliding-window-maximum/
 * 6. LeetCode 732. 我的日程安排表 III: 区间重叠问题
 * 7. LeetCode 699. 掉落的方块: 区间更新与查询最大值
 * 8. LeetCode 995. K 连续位的最小翻转次数: 区间翻转操作
 * 9. LintCode 205. Interval Minimum Number: https://www.lintcode.com/problem/interval-minimum-number/
 * 10. LintCode 207. 区间求和 II: https://www.lintcode.com/problem/interval-sum-ii/
 * 11. LintCode 439. 线段树的查询 II: 区间和查询
 * 12. LintCode 440. 线段树的修改: 单点更新
 * 13. Codeforces 61E. Enemy is weak: 区间统计问题
 * 14. Codeforces 459D. Pashmak and Parmida's problem: 区间逆序对统计
 * 15. 牛客网 NC24970. 线段树练习一: 基础区间操作
 * 16. 杭电 OJ 1542. Atlantis: 扫描线算法与线段树
 * 17. USACO 2017 January Contest, Gold Problem 1. Balanced Photo: 区间统计
 * 18. AtCoder ARC 008 B. 投票: 区间操作与统计
 * 19. SPOJ GSS1 - Can you answer these queries I: 区间最大子段和
 * 20. UVa OJ 11990. Dynamic Inversion: 动态逆序对问题
 * 21. 洛谷 P3373. 【模板】线段树 2: 区间乘加操作
 * 22. 洛谷 P3805. 【模板】manacher: 最长回文子串
```

```
* 23. 计蒜客 线段树专题：各种线段树应用场景
* 24. HackerRank Array and simple queries: https://www.hackerrank.com/challenges/array-and-simple-queries/problem
*/
// 掌握线段树的关键点
/*
 * 1. 懒标记的正确传递与优先级处理
 * 2. 区间合并逻辑的设计（如本题中的连续 1 长度合并）
 * 3. 不同操作类型的统一处理框架
 * 4. 边界条件的处理（如叶子节点、空区间等）
 * 5. 效率优化：避免不必要的递归和计算
 * 6. Java 中的数组实现技巧：使用静态数组存储线段树
 * 7. 数据规模评估与内存分配
 * 8. 异常处理与参数校验
 * 9. 多线程环境下的线程安全考虑
 * 10. 性能测试与基准比较
*/
}

=====
```

文件: Code01_SequenceOperation.py

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

线段树经典应用：多标记序列操作

题目来源：洛谷 P2572 [SDOI2010] 序列操作

题目链接：<https://www.luogu.com.cn/problem/P2572>

核心算法：线段树 + 多重懒标记

难度：省选/NOI-

【题目详细描述】

给定一个长度为 n 的 01 序列，支持 5 种操作：

1. 操作 0 1 r: 将区间[1, r]全部置为 0
2. 操作 1 1 r: 将区间[1, r]全部置为 1
3. 操作 2 1 r: 将区间[1, r]全部取反
4. 操作 3 1 r: 查询区间[1, r]中 1 的个数
5. 操作 4 1 r: 查询区间[1, r]中连续 1 的最长长度

【解题思路】

这是一个典型的线段树应用题，需要维护多种区间信息并处理多重懒标记。

线段树节点需要保存丰富的信息来支持连续子串长度的查询。

【核心数据结构设计】

线段树每个节点维护以下信息：

- sum: 区间内 1 的总数
- len0/len1: 区间内连续 0/1 的最长子串长度
- pre0/pre1: 区间内连续 0/1 的最长前缀长度
- suf0/suf1: 区间内连续 0/1 的最长后缀长度
- size: 当前区间的长度

懒标记设计：

- update: 记录区间被置为的具体值（0 或 1），如果是 None 表示没有赋值操作
- reverse_flag: 标记区间是否有待处理的翻转操作

【关键技术点】

1. 懒标记优先级处理：更新操作 (update) 优先于翻转操作 (reverse)
2. 区间合并逻辑：需要考虑左右子区间连接处的情况
3. 多重懒标记的下传顺序和相互影响处理
4. 边界条件处理：如区间为空、单元素区间等特殊情况

【复杂度分析】

- 时间复杂度：
 - 建树: $O(n)$
 - 单次操作（更新/查询）: $O(\log n)$
 - m 次操作总时间复杂度: $O((n + m) \log n)$
- 空间复杂度: $O(4n)$, 线段树标准空间配置

【算法优化点】

1. 懒标记延迟下传：避免不必要的更新操作
2. 区间合并时的高效计算：通过维护前缀和后缀信息加速
3. 使用类封装：将线段树封装为一个类，提高代码复用性和可维护性
4. 避免递归过深：Python 中递归深度有默认限制，可以考虑非递归实现

【工程化考量】

1. 内存布局：使用对象存储线段树节点，方便管理
2. 错误处理：添加参数校验，处理非法输入
3. 性能优化：Python 中递归实现的线段树性能可能不如 C++，对于大规模数据可以考虑其他优化方法
4. 可读性：合理使用类和函数组织代码，添加详细注释

【Python 语言特性应用】

1. 类封装：使用类封装线段树，提高代码复用性

2. 列表索引：使用列表存储节点，避免动态内存分配
3. None 值处理：使用 None 表示没有懒标记
4. 递归实现：递归编写线段树操作，代码简洁明了

【类似题目推荐】

1. LeetCode 307. 区域和检索 - 数组可修改：<https://leetcode.cn/problems/range-sum-query-mutable/>
2. Codeforces 242E - XOR on Segment：<https://codeforces.com/problemset/problem/242/E>
3. 洛谷 P3373 【模板】线段树 2：<https://www.luogu.com.cn/problem/P3373>
4. POJ 3468 A Simple Problem with Integers：<http://poj.org/problem?id=3468>
5. HDU 1698 Just a Hook：<http://acm.hdu.edu.cn/showproblem.php?pid=1698>
6. SPOJ GSS1 - Can you answer these queries I：<https://www.spoj.com/problems/GSS1/>
7. LintCode 207. 区间求和 II：<https://www.lintcode.com/problem/interval-sum-ii/>
8. HackerRank Array and simple queries：<https://www.hackerrank.com/challenges/array-and-simple-queries/problem>
9. LeetCode 732. 我的日程安排表 III：区间重叠问题
10. LeetCode 699. 掉落的方块：区间更新与查询最大值
11. LeetCode 551. 学生出勤记录 I：连续状态查询
12. LeetCode 567. 字符串的排列：滑动窗口与字符频率统计
13. LeetCode 995. K 连续位的最小翻转次数：区间翻转操作
14. LintCode 439. 线段树的查询 II：区间和查询
15. LintCode 440. 线段树的修改：单点更新
16. LintCode 441. 线段树的构造 II：构建区间和线段树
17. Codeforces 61E. Enemy is weak：区间统计问题
18. Codeforces 459D. Pashmak and Parmida's problem：区间逆序对统计
19. HackerEarth Binary Operations：区间位运算操作
20. 牛客网 NC24970. 线段树练习一：基础区间操作
21. 杭电 OJ 1542. Atlantis：扫描线算法与线段树
22. USACO 2017 January Contest, Gold Problem 1. Balanced Photo：区间统计
23. AtCoder ARC 008 B. 投票：区间操作与统计
24. SPOJ GSS1 - Can you answer these queries I：区间最大子段和
25. UVa OJ 11990. Dynamic Inversion：动态逆序对问题
26. 洛谷 P3373. 【模板】线段树 2：区间乘加操作
27. 洛谷 P3805. 【模板】manacher：最长回文子串
28. 计蒜客 线段树专题：各种线段树应用场景

【掌握线段树的关键点】

1. 懒标记的正确传递与优先级处理
 2. 区间合并逻辑的设计（如本题中的连续 1 长度合并）
 3. 不同操作类型的统一处理框架
 4. 边界条件的处理（如叶子节点、空区间等）
 5. 效率优化：避免不必要的递归和计算
- ,,,

```

class SegmentTree:
    def __init__(self, arr):
        """
        初始化线段树

        Args:
            arr: 原始数组
        """
        self.n = len(arr)
        self.arr = arr
        # 初始化线段树所需的各种信息数组
        self.sum = [0] * (4 * self.n)          # 区间内 1 的总数
        self.len0 = [0] * (4 * self.n)         # 连续 0 的最长长度
        self.pre0 = [0] * (4 * self.n)         # 连续 0 的最长前缀长度
        self.suf0 = [0] * (4 * self.n)         # 连续 0 的最长后缀长度
        self.len1 = [0] * (4 * self.n)         # 连续 1 的最长长度
        self.pre1 = [0] * (4 * self.n)         # 连续 1 的最长前缀长度
        self.suf1 = [0] * (4 * self.n)         # 连续 1 的最长后缀长度
        self.change = [0] * (4 * self.n)       # 区间赋值的目标值
        self.update = [False] * (4 * self.n)    # 区间赋值标记
        self.reverse = [False] * (4 * self.n)   # 区间翻转标记
        # 构建线段树
        self.build(1, 1, self.n)

    def up(self, i, ln, rn):
        """
        向上更新当前节点信息（合并左右子节点信息）

        Args:
            i: 当前节点索引
            ln: 左子区间长度
            rn: 右子区间长度
        """
        l = i << 1      # 左子节点索引
        r = i << 1 | 1  # 右子节点索引

        # 更新区间内 1 的总数
        self.sum[i] = self.sum[l] + self.sum[r]

        # 更新连续 0 的信息
        # 取左右子区间的最大值或左右子区间连接处的连续 0
        self.len0[i] = max(self.len0[l], self.len0[r], self.suf0[l] + self.pre0[r])

```

```

# 更新连续 0 的前缀
# 如果左子区间全是 0，则前缀包括整个左子区间和右子区间的前缀
self.pre0[i] = self.pre0[l] if self.len0[l] < ln else self.pre0[l] + self.pre0[r]

# 更新连续 0 的后缀
# 如果右子区间全是 0，则后缀包括整个右子区间和左子区间的后缀
self.suf0[i] = self.suf0[r] if self.len0[r] < rn else self.suf0[r] + self.suf0[l]

# 更新连续 1 的信息（与连续 0 的处理方式类似）
self.len1[i] = max(self.len1[l], self.len1[r], self.suf1[r] + self.pre1[l])
self.pre1[i] = self.pre1[l] if self.len1[l] < ln else self.pre1[l] + self.pre1[r]
self.suf1[i] = self.suf1[r] if self.len1[r] < rn else self.suf1[r] + self.suf1[l]

```

```
def update_lazy(self, i, v, n):
```

```
    """

```

处理区间赋值的懒标记

Args:

- i: 当前节点索引
- v: 要设置的值（0 或 1）
- n: 当前节点表示的区间长度

```
    """

```

更新区间内 1 的总数

```
self.sum[i] = v * n
```

更新连续 0 的信息：如果 v=0，则整个区间都是 0；否则都是 1（没有 0）

```
self.len0[i] = self.pre0[i] = self.suf0[i] = n if v == 0 else 0
```

更新连续 1 的信息：如果 v=1，则整个区间都是 1；否则都是 0（没有 1）

```
self.len1[i] = self.pre1[i] = self.suf1[i] = n if v == 1 else 0
```

记录区间赋值的目标值

```
self.change[i] = v
```

设置更新标记

```
self.update[i] = True
```

清空翻转标记（更新操作优先于翻转操作）

```
self.reverse[i] = False
```

```
def reverse_lazy(self, i, n):
```

```
    """

```

处理区间翻转的懒标记

Args:

- i: 当前节点索引
- n: 当前节点表示的区间长度

```
    """

```

```
# 翻转 1 的个数: 1 变 0, 0 变 1
self.sum[i] = n - self.sum[i]
# 交换连续 0 和连续 1 的各种长度信息
self.len0[i], self.len1[i] = self.len1[i], self.len0[i] # 交换最长连续 0/1 长度
self.pre0[i], self.pre1[i] = self.pre1[i], self.pre0[i] # 交换前缀 0/1 长度
self.suf0[i], self.suf1[i] = self.suf1[i], self.suf0[i] # 交换后缀 0/1 长度
# 翻转翻转标记 (多次翻转可以抵消)
self.reverse[i] = not self.reverse[i]
```

```
def down(self, i, ln, rn):
```

```
    """

```

```
    向下传递懒标记到子节点

```

```
Args:
```

```
    i: 当前节点索引
```

```
    ln: 左子区间长度
```

```
    rn: 右子区间长度
"""

```

```
# 先处理更新操作 (优先级高于翻转操作)
```

```
if self.update[i]:
```

```
    # 左子节点应用更新操作
```

```
    self.update_lazy(i << 1, self.change[i], ln)
```

```
    # 右子节点应用更新操作
```

```
    self.update_lazy(i << 1 | 1, self.change[i], rn)
```

```
    # 清除当前节点的更新标记
```

```
    self.update[i] = False
```

```
# 再处理翻转操作
```

```
if self.reverse[i]:
```

```
    # 左子节点应用翻转操作
```

```
    self.reverse_lazy(i << 1, ln)
```

```
    # 右子节点应用翻转操作
```

```
    self.reverse_lazy(i << 1 | 1, rn)
```

```
    # 清除当前节点的翻转标记
```

```
    self.reverse[i] = False
```

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

```
    """

```

```
    构建线段树

```

```
Args:
```

```
    i: 当前节点索引
```

```
    l: 当前区间左边界 (1-based)
```

```

r: 当前区间右边界 (1-based)
"""

# 叶子节点情况
if l == r:
    # 直接赋值原始数组的值
    self.sum[i] = self.arr[1]
    # 初始化连续 0 的信息
    self.len0[i] = self.pre0[i] = self.suf0[i] = 1 if self.arr[1] == 0 else 0
    # 初始化连续 1 的信息
    self.len1[i] = self.pre1[i] = self.suf1[i] = 1 if self.arr[1] == 1 else 0
else:
    # 非叶子节点, 递归构建左右子树
    mid = (l + r) >> 1 # 计算中点
    self.build(i << 1, l, mid) # 构建左子树
    self.build(i << 1 | 1, mid + 1, r) # 构建右子树
    # 向上合并子节点信息
    self.up(i, mid - 1 + 1, r - mid)

# 初始化懒标记为未激活状态
self.update[i] = False
self.reverse[i] = False

def update_range(self, jobl, jobr, jobv, l, r, i):
"""
区间赋值操作
"""

Args:
    jobl: 待更新区间的左边界 (1-based)
    jobr: 待更新区间的右边界 (1-based)
    jobv: 要设置的值 (0 或 1)
    l: 当前节点区间左边界 (1-based)
    r: 当前节点区间右边界 (1-based)
    i: 当前节点索引
"""

# 当前区间完全包含在待更新区间内
if jobl <= l and r <= jobr:
    # 直接应用懒标记
    self.update_lazy(i, jobv, r - l + 1)
else:
    # 当前区间部分包含在待更新区间内
    mid = (l + r) >> 1 # 计算中点
    ln = mid - l + 1 # 左子区间长度
    rn = r - mid # 右子区间长度

```

Args:

- jobl: 待更新区间的左边界 (1-based)
- jobr: 待更新区间的右边界 (1-based)
- jobv: 要设置的值 (0 或 1)
- l: 当前节点区间左边界 (1-based)
- r: 当前节点区间右边界 (1-based)
- i: 当前节点索引

"""

当前区间完全包含在待更新区间内

if jobl <= l and r <= jobr:

- # 直接应用懒标记
- self.update_lazy(i, jobv, r - l + 1)

else:

- # 当前区间部分包含在待更新区间内
- mid = (l + r) >> 1 # 计算中点
- ln = mid - l + 1 # 左子区间长度
- rn = r - mid # 右子区间长度

```

# 先向下传递懒标记
self.down(i, ln, rn)

# 递归处理左右子区间
if jobl <= mid:
    self.update_range(jobl, jobr, jobv, l, mid, i << 1)
if jobr > mid:
    self.update_range(jobl, jobr, jobv, mid + 1, r, i << 1 | 1)

# 向上合并子节点信息
self.up(i, ln, rn)

def reverse_range(self, jobl, jobr, l, r, i):
    """
    区间翻转操作
    """

    Args:
        jobl: 待翻转区间的左边界 (1-based)
        jobr: 待翻转区间的右边界 (1-based)
        l: 当前节点区间左边界 (1-based)
        r: 当前节点区间右边界 (1-based)
        i: 当前节点索引
    """

    # 当前区间完全包含在待翻转区间内
    if jobl <= l and r <= jobr:
        # 直接应用翻转懒标记
        self.reverse_lazy(i, r - l + 1)
    else:
        # 当前区间部分包含在待翻转区间内
        mid = (l + r) >> 1 # 计算中点
        ln = mid - l + 1 # 左子区间长度
        rn = r - mid # 右子区间长度

        # 先向下传递懒标记
        self.down(i, ln, rn)

        # 递归处理左右子区间
        if jobl <= mid:
            self.reverse_range(jobl, jobr, l, mid, i << 1)
        if jobr > mid:
            self.reverse_range(jobl, jobr, mid + 1, r, i << 1 | 1)

```

```
# 向上合并子节点信息
self.up(i, ln, rn)
```

```
def query_sum(self, jobl, jobr, l, r, i):
    """
    查询区间内 1 的总数
```

Args:

```
jobl: 查询区间的左边界 (1-based)
jobr: 查询区间的右边界 (1-based)
l: 当前节点区间左边界 (1-based)
r: 当前节点区间右边界 (1-based)
i: 当前节点索引
```

Returns:

```
int: 区间内 1 的总数
"""
```

```
# 当前区间完全包含在查询区间内
```

```
if jobl <= l and r <= jobr:
    return self.sum[i]
```

```
# 当前区间部分包含在查询区间内
```

```
mid = (l + r) >> 1 # 计算中点
ln = mid - l + 1 # 左子区间长度
rn = r - mid # 右子区间长度
```

```
# 先向下传递懒标记
```

```
self.down(i, ln, rn)
```

```
ans = 0
```

```
# 递归查询左右子区间
```

```
if jobl <= mid:
    ans += self.query_sum(jobl, jobr, l, mid, i << 1)
if jobr > mid:
    ans += self.query_sum(jobl, jobr, mid + 1, r, i << 1 | 1)
```

```
return ans
```

```
def query_longest(self, jobl, jobr, l, r, i):
    """
```

```
查询区间内连续 1 的最长长度
```

Args:

jobl: 查询区间的左边界 (1-based)
jobr: 查询区间的右边界 (1-based)
l: 当前节点区间左边界 (1-based)
r: 当前节点区间右边界 (1-based)
i: 当前节点索引

Returns:

list: [最长连续 1 的长度, 连续 1 的最长前缀长度, 连续 1 的最长后缀长度]

"""

当前区间完全包含在查询区间内

if jobl <= l and r <= jobr:

 return [self.len1[i], self.pre1[i], self.suf1[i]]

else:

当前区间部分包含在查询区间内或查询区间跨越多个子区间

 mid = (l + r) >> 1 # 计算中点

 ln = mid - l + 1 # 左子区间长度

 rn = r - mid # 右子区间长度

先向下传递懒标记

 self.down(i, ln, rn)

查询区间完全在左子区间

if jobr <= mid:

 return self.query_longest(jobl, jobr, l, mid, i << 1)

查询区间完全在右子区间

if jobl > mid:

 return self.query_longest(jobl, jobr, mid + 1, r, i << 1 | 1)

查询区间跨越左右子区间

分别查询左右子区间的信息

l3 = self.query_longest(jobl, jobr, l, mid, i << 1)

r3 = self.query_longest(jobl, jobr, mid + 1, r, i << 1 | 1)

提取左右子区间的信息

llen, lpre, lsuf = l3[0], l3[1], l3[2] # 左子区间的最长 1 长度、前缀、后缀

rlen, rpre, rsuf = r3[0], r3[1], r3[2] # 右子区间的最长 1 长度、前缀、后缀

合并信息

最长连续 1 长度: 左子区间的最大值、右子区间的最大值、或左右连接处的连续 1

length = max(llen, rlen, lsuf + rpre)

连续 1 的最长前缀: 如果左子区间的最长连续 1 覆盖了整个查询部分的左子区间, 则包括右子区间的前缀

```

    pre = lpre if llen < mid - max(jobl, 1) + 1 else lpre + rpre

    # 连续 1 的最长后缀: 如果右子区间的最长连续 1 覆盖了整个查询部分的右子区间, 则包括左子
    区间的后缀
    suf = rsuf if rlen < min(r, jobr) - mid else lsuf + rsuf

    return [length, pre, suf]

```

```

def main():
    """
    主函数: 读取输入并处理操作

```

输入格式:

- 第一行: n (序列长度) 和 m (操作数量)
- 第二行: n 个 0 或 1, 表示初始序列
- 接下来 m 行: 每行一个操作, 格式为 op l r

操作类型:

- 0 l r: 将区间[1, r]全部置为 0
- 1 l r: 将区间[1, r]全部置为 1
- 2 l r: 将区间[1, r]全部取反
- 3 l r: 查询区间[1, r]中 1 的个数
- 4 l r: 查询区间[1, r]中连续 1 的最长长度

"""

优化输入处理速度

```

import sys
input = sys.stdin.read
data = input().split()

```

读取初始参数

```

idx = 0
n = int(data[idx])
idx += 1
m = int(data[idx])
idx += 1

```

初始化数组 (1-based 索引)

```

arr = [0] * (n + 1)
for i in range(1, n + 1):
    arr[i] = int(data[idx])
    idx += 1

```

创建线段树

```

seg_tree = SegmentTree(arr)

# 处理操作并收集结果
results = []
for _ in range(m):
    # 读取操作类型和参数
    op = int(data[idx])
    idx += 1
    jobl = int(data[idx]) + 1 # 转换为 1-based 索引
    idx += 1
    jobr = int(data[idx]) + 1 # 转换为 1-based 索引
    idx += 1

    # 根据操作类型执行相应操作
    if op == 0: # 将区间置为 0
        seg_tree.update_range(jobl, jobr, 0, 1, n, 1)
    elif op == 1: # 将区间置为 1
        seg_tree.update_range(jobl, jobr, 1, 1, n, 1)
    elif op == 2: # 区间取反
        seg_tree.reverse_range(jobl, jobr, 1, n, 1)
    elif op == 3: # 查询 1 的个数
        results.append(str(seg_tree.query_sum(jobl, jobr, 1, n, 1)))
    elif op == 4: # 查询最长连续 1
        results.append(str(seg_tree.query_longest(jobl, jobr, 1, n, 1)[0]))

# 批量输出结果，提高输出效率
print('\n'.join(results))

```

```

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

文件: Code02_LongestAlternateSubstring.cpp

```

=====
/*
```

题目解析:

给定一个字符串，初始全为' L'，每次操作翻转一个位置的字符，求每次操作后最长的 LR 交替子串长度。

解题思路:

使用线段树维护每个区间的最长交替子串长度，以及前缀和后缀的最长交替长度。

关键技术点:

1. 区间合并时需要判断中间连接处是否可以连接
2. 单点更新时需要重新计算区间信息

复杂度分析:

- 时间复杂度:
 - 建树: $O(n)$
 - 单次操作: $O(\log n)$
 - 总时间复杂度: $O(n + q \log n)$
- 空间复杂度: $O(n)$

线段树常见变种:

1. 动态开点线段树: 适用场景: 数据范围很大, 但实际使用较少的情况
2. 可持久化线段树(主席树): 适用场景: 需要保存历史版本的信息
3. 扫描线 + 线段树: 适用场景: 平面几何问题, 如矩形面积并
4. 树链剖分 + 线段树: 适用场景: 树上路径操作
5. 线段树合并: 适用场景: 动态维护多个集合的信息

补充题目:

1. LeetCode 315. 计算右侧小于当前元素的个数 - <https://leetcode.cn/problems/count-of-smaller-numbers-after-self/>
 2. LeetCode 327. 区间和的个数 - <https://leetcode.cn/problems/count-of-range-sum/>
 3. LeetCode 493. 翻转对 - <https://leetcode.cn/problems/reverse-pairs/>
 4. Codeforces 339D - Xenia and Bit Operations - <https://codeforces.com/problemset/problem/339/D>
 5. Codeforces 380C - Sereja and Brackets - <https://codeforces.com/problemset/problem/380/C>
- */

```
// 使用基础的 C++ 实现, 避免使用复杂的 STL 容器
const int MAXN = 200001;
```

```
// 原始数组
int arr[MAXN];
```

```
// 交替最长子串长度
int len[MAXN << 2];
```

```
// 交替最长前缀长度
int pre[MAXN << 2];
```

```
// 交替最长后缀长度
int suf[MAXN << 2];
```

```
void up(int l, int r, int i) {
    len[i] = len[i << 1] > len[i << 1 | 1] ? len[i << 1] : len[i << 1 | 1];
```

```

pre[i] = pre[i << 1];
suf[i] = suf[i << 1 | 1];
int mid = (l + r) >> 1;
int ln = mid - 1 + 1;
int rn = r - mid;
if (arr[mid] != arr[mid + 1]) {
    int combined = suf[i << 1] + pre[i << 1 | 1];
    if (combined > len[i]) len[i] = combined;
    if (len[i << 1] == ln) {
        pre[i] = ln + pre[i << 1 | 1];
    }
    if (len[i << 1 | 1] == rn) {
        suf[i] = rn + suf[i << 1];
    }
}
}

```

```

void build(int l, int r, int i) {
    if (l == r) {
        len[i] = 1;
        pre[i] = 1;
        suf[i] = 1;
    } else {
        int mid = (l + r) >> 1;
        build(l, mid, i << 1);
        build(mid + 1, r, i << 1 | 1);
        up(l, r, i);
    }
}

```

```

void reverse_char(int jobi, int l, int r, int i) {
    if (l == r) {
        arr[jobi] ^= 1;
    } else {
        int mid = (l + r) >> 1;
        if (jobi <= mid) {
            reverse_char(jobi, l, mid, i << 1);
        } else {
            reverse_char(jobi, mid + 1, r, i << 1 | 1);
        }
        up(l, r, i);
    }
}

```

```

// 由于编译环境限制，此处省略 main 函数
// 在实际使用中，需要根据具体编译环境实现输入输出

/*
// 以下是一个示例 main 函数，用于测试
#include <iostream>
using namespace std;

int main() {
    int n, q;
    cin >> n >> q;
    build(1, n, 1);
    for (int i = 0; i < q; i++) {
        int index;
        cin >> index;
        reverse_char(index, 1, n, 1);
        cout << len[1] << endl;
    }
    return 0;
}
*/

```

文件: Code02_LongestAlternateSubstring.java

```

=====
package class113;

/**
 * 最长 LR 交替子串
 * 题目来源: 洛谷 P6492
 * 题目链接: https://www.luogu.com.cn/problem/P6492
 *
 * 核心算法: 线段树 + 区间信息维护
 * 难度: 中等
 *
 * 【题目详细描述】
 * 给定一个长度为 n 的字符串，一开始字符串中全是' L' 字符
 * 有 q 次修改，每次指定一个位置 i
 * 如果 i 位置是' L' 字符那么改成' R' 字符
 * 如果 i 位置是' R' 字符那么改成' L' 字符
 * 如果一个子串是两种字符不停交替出现的样子，也就是 LRLR... 或者 RLRL...

```

- * 那么说这个子串是有效子串
- * 每次修改后，都打印当前整个字符串中最长交替子串的长度
- *
- * 【解题思路】
- * 使用线段树维护每个区间的最长交替子串长度，以及前缀和后缀的最长交替长度。
- *
- * 【核心算法】
- * 线段树的每个节点维护三个关键信息：
 - * 1. $len[i]$: 区间内最长交替子串的长度
 - * 2. $pre[i]$: 区间左端开始的最长交替前缀长度
 - * 3. $suf[i]$: 区间右端结束的最长交替后缀长度
- *
- * 合并两个相邻区间时，需要考虑左子区间的后缀和右子区间的前缀是否可以拼接成更长的交替子串。
- * 关键在于判断中间两个字符是否不同 ($arr[mid] \neq arr[mid + 1]$)。
- *
- * 【复杂度分析】
- * - 时间复杂度：
 - * - 建树: $O(n)$
 - * - 单次操作: $O(\log n)$
 - * - 总时间复杂度: $O(n + q \log n)$
- * - 空间复杂度: $O(n)$, 线段树所需空间为 $4n$
- *
- * 【算法优化点】
- * 1. 位运算优化：使用异或操作 ($arr[job_i] \hat{=} 1$) 高效实现字符翻转
- * 2. 懒惰标记优化：虽然本题没有使用，但对于需要区间更新的类似问题可以考虑
- * 3. 空间优化：使用数组而非对象存储线段树信息，减少内存开销
- *
- * 【工程化考量】
- * 1. 输入输出效率：使用 `BufferedReader`、`StreamTokenizer` 和 `PrintWriter` 实现高效输入输出
- * 2. 边界条件处理：初始化全为' L' 的情况，单元素区间的处理
- * 3. 数据类型选择：使用 `int` 数组存储字符 (0 表示' L'，1 表示' R')，提高效率
- * 4. 代码可读性：函数命名清晰，逻辑模块化
- * 5. 内存管理：使用 `static` 数组预分配空间，避免动态分配
- *
- * 【使用注意】
- * 提交时请把类名改成"Main"，可以直接通过

*/

/*

题目解析：

给定一个字符串，初始全为' L'，每次操作翻转一个位置的字符，求每次操作后最长的 LR 交替子串长度。

解题思路：

使用线段树维护每个区间的最长交替子串长度，以及前缀和后缀的最长交替长度。

关键技术点：

1. 区间合并时需要判断中间连接处是否可以连接
2. 单点更新时需要重新计算区间信息

复杂度分析：

- 时间复杂度：
 - 建树： $O(n)$
 - 单次操作： $O(\log n)$
 - 总时间复杂度： $O(n + q \log n)$
- 空间复杂度： $O(n)$

线段树常见变种：

1. 动态开点线段树：适用场景：数据范围很大，但实际使用较少的情况
2. 可持久化线段树（主席树）：适用场景：需要保存历史版本的信息
3. 扫描线 + 线段树：适用场景：平面几何问题，如矩形面积并
4. 树链剖分 + 线段树：适用场景：树上路径操作
5. 线段树合并：适用场景：动态维护多个集合的信息

补充题目：

1. LeetCode 315. 计算右侧小于当前元素的个数 - <https://leetcode.cn/problems/count-of-smaller-numbers-after-self/>
2. LeetCode 327. 区间和的个数 - <https://leetcode.cn/problems/count-of-range-sum/>
3. LeetCode 493. 翻转对 - <https://leetcode.cn/problems/reverse-pairs/>
4. Codeforces 339D - Xenia and Bit Operations - <https://codeforces.com/problemset/problem/339/D>
5. Codeforces 380C - Sereja and Brackets - <https://codeforces.com/problemset/problem/380/C>

```
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_LongestAlternateSubstring {

    public static int MAXN = 200001;

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

```

// 交替最长子串长度
public static int[] len = new int[MAXN << 2];

// 交替最长前缀长度
public static int[] pre = new int[MAXN << 2];

// 交替最长后缀长度
public static int[] suf = new int[MAXN << 2];

public static void up(int l, int r, int i) {
    len[i] = Math.max(len[i << 1], len[i << 1 | 1]);
    pre[i] = pre[i << 1];
    suf[i] = suf[i << 1 | 1];
    int mid = (l + r) >> 1;
    int ln = mid - l + 1;
    int rn = r - mid;
    if (arr[mid] != arr[mid + 1]) {
        len[i] = Math.max(len[i], suf[i << 1] + pre[i << 1 | 1]);
        if (len[i << 1] == ln) {
            pre[i] = ln + pre[i << 1 | 1];
        }
        if (len[i << 1 | 1] == rn) {
            suf[i] = rn + suf[i << 1];
        }
    }
}

public static void build(int l, int r, int i) {
    if (l == r) {
        len[i] = 1;
        pre[i] = 1;
        suf[i] = 1;
    } else {
        int mid = (l + r) >> 1;
        build(l, mid, i << 1);
        build(mid + 1, r, i << 1 | 1);
        up(l, r, i);
    }
}

public static void reverse(int jobi, int l, int r, int i) {
    if (l == r) {
        arr[jobi] ^= 1;
    }
}

```

```

    } else {
        int mid = (l + r) >> 1;
        if (jobi <= mid) {
            reverse(jobi, l, mid, i << 1);
        } else {
            reverse(jobi, mid + 1, r, i << 1 | 1);
        }
        up(l, r, i);
    }
}

public static void main(String[] args) throws IOException {
    BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
    StreamTokenizer in = new StreamTokenizer(br);
    PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));
    in.nextToken();
    int n = (int) in.nval;
    in.nextToken();
    int q = (int) in.nval;
    build(1, n, 1);
    for (int i = 1, index; i <= q; i++) {
        in.nextToken();
        index = (int) in.nval;
        reverse(index, 1, n, 1);
        out.println(len[1]);
    }
    out.flush();
    out.close();
    br.close();
}
}

```

}

=====

文件: Code02_LongestAlternateSubstring.py

=====

"""

题目解析:

给定一个字符串，初始全为'L'，每次操作翻转一个位置的字符，求每次操作后最长的LR交替子串长度。

解题思路:

使用线段树维护每个区间的最长交替子串长度，以及前缀和后缀的最长交替长度。

关键技术点：

1. 区间合并时需要判断中间连接处是否可以连接
2. 单点更新时需要重新计算区间信息

复杂度分析：

- 时间复杂度：

- 建树: $O(n)$
- 单次操作: $O(\log n)$
- 总时间复杂度: $O(n + q \log n)$
- 空间复杂度: $O(n)$

线段树常见变种：

1. 动态开点线段树：适用场景：数据范围很大，但实际使用较少的情况
2. 可持久化线段树（主席树）：适用场景：需要保存历史版本的信息
3. 扫描线 + 线段树：适用场景：平面几何问题，如矩形面积并
4. 树链剖分 + 线段树：适用场景：树上路径操作
5. 线段树合并：适用场景：动态维护多个集合的信息

补充题目：

1. LeetCode 315. 计算右侧小于当前元素的个数 - <https://leetcode.cn/problems/count-of-smaller-numbers-after-self/>
 2. LeetCode 327. 区间和的个数 - <https://leetcode.cn/problems/count-of-range-sum/>
 3. LeetCode 493. 翻转对 - <https://leetcode.cn/problems/reverse-pairs/>
 4. Codeforces 339D - Xenia and Bit Operations - <https://codeforces.com/problemset/problem/339/D>
 5. Codeforces 380C - Sereja and Brackets - <https://codeforces.com/problemset/problem/380/C>
- """

```
class AlternateSubstringTree:  
    def __init__(self, n):  
        self.n = n  
        self.arr = [0] * (n + 1) # 0 represents 'L', 1 represents 'R'  
        self.len = [0] * (4 * n)  
        self.pre = [0] * (4 * n)  
        self.suf = [0] * (4 * n)  
        self.build(1, 1, n)  
  
    def up(self, l, r, i):  
        self.len[i] = max(self.len[i << 1], self.len[i << 1 | 1])  
        self.pre[i] = self.pre[i << 1]  
        self.suf[i] = self.suf[i << 1 | 1]  
        mid = (l + r) >> 1  
        ln = mid - 1 + 1
```

```

rn = r - mid
if self.arr[mid] != self.arr[mid + 1]:
    self.len[i] = max(self.len[i], self.suf[i << 1] + self.pre[i << 1 | 1])
    if self.len[i << 1] == ln:
        self.pre[i] = ln + self.pre[i << 1 | 1]
    if self.len[i << 1 | 1] == rn:
        self.suf[i] = rn + self.suf[i << 1]

def build(self, i, l, r):
    if l == r:
        self.len[i] = self.pre[i] = self.suf[i] = 1
    else:
        mid = (l + r) >> 1
        self.build(i << 1, l, mid)
        self.build(i << 1 | 1, mid + 1, r)
        self.up(l, r, i)

def reverse_char(self, jobi, l, r, i):
    if l == r:
        self.arr[jobi] ^= 1
    else:
        mid = (l + r) >> 1
        if jobi <= mid:
            self.reverse_char(jobi, l, mid, i << 1)
        else:
            self.reverse_char(jobi, mid + 1, r, i << 1 | 1)
        self.up(l, r, i)

def main():
    import sys
    input = sys.stdin.read
    data = input().split()

    n = int(data[0])
    q = int(data[1])

    tree = AlternateSubstringTree(n)

    results = []
    idx = 2
    for _ in range(q):
        index = int(data[idx])
        idx += 1

```

```
tree.reverse_char(index, 1, n, 1)
results.append(str(tree.len[1]))

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

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

=====

文件: Code03_TunnelWarfare.cpp

=====

/*

题目解析:

有 n 个房子排成一排，相邻房子有地道连接。支持摧毁、恢复和查询操作。

解题思路:

使用线段树维护每个区间的连续 1 的前缀和后缀长度，其中 1 表示房子未被摧毁。

关键技术点:

1. 查询操作需要根据位置在区间中的位置进行不同处理
2. 区间合并时考虑跨区间的情况

复杂度分析:

- 时间复杂度:

- 建树: $O(n)$
- 单次操作: $O(\log n)$
- 总时间复杂度: $O((n + m) \log n)$
- 空间复杂度: $O(n)$

补充题目:

1. LeetCode 715. Range 模块 - <https://leetcode.cn/problems/range-module/>
 2. LeetCode 729. 我的日程安排表 I - <https://leetcode.cn/problems/my-calendar-i/>
 3. LeetCode 731. 我的日程安排表 II - <https://leetcode.cn/problems/my-calendar-ii/>
 4. LeetCode 732. 我的日程安排表 III - <https://leetcode.cn/problems/my-calendar-iii/>
 5. Codeforces 52C - Circular RMQ - <https://codeforces.com/problemset/problem/52/C>
 6. Codeforces 242E - XOR on Segment - <https://codeforces.com/problemset/problem/242/E>
 7. 洛谷 P2894 [USACO08FEB] Hotel G - <https://www.luogu.com.cn/problem/P2894>
- */

// 使用基础的 C++ 实现，避免使用复杂的 STL 容器
const int MAXN = 50001;

```

// 连续 1 的最长前缀长度
int pre[MAXN << 2];

// 连续 1 的最长后缀长度
int suf[MAXN << 2];

// 摧毁的房屋编号入栈，以便执行恢复操作
int stack_arr[MAXN];

void up(int l, int r, int i) {
    pre[i] = pre[i << 1];
    suf[i] = suf[i << 1 | 1];
    int mid = (l + r) >> 1;
    if (pre[i << 1] == mid - 1 + 1) {
        pre[i] += pre[i << 1 | 1];
    }
    if (suf[i << 1 | 1] == r - mid) {
        suf[i] += suf[i << 1];
    }
}

void build(int l, int r, int i) {
    if (l == r) {
        pre[i] = suf[i] = 1;
    } else {
        int mid = (l + r) >> 1;
        build(l, mid, i << 1);
        build(mid + 1, r, i << 1 | 1);
        up(l, r, i);
    }
}

void update(int jobi, int jobv, int l, int r, int i) {
    if (l == r) {
        pre[i] = suf[i] = jobv;
    } else {
        int mid = (l + r) >> 1;
        if (jobi <= mid) {
            update(jobi, jobv, l, mid, i << 1);
        } else {
            update(jobi, jobv, mid + 1, r, i << 1 | 1);
        }
        up(l, r, i);
    }
}

```

```

    }

}

// 已知 jobi 在 l...r 范围上
// 返回 jobi 往两侧扩展出的最大长度
// 递归需要遵循的潜台词：从 jobi 往两侧扩展，一定无法扩展到 l...r 范围之外！
int query(int jobi, int l, int r, int i) {
    if (l == r) {
        return pre[i];
    } else {
        int mid = (l + r) >> 1;
        if (jobi <= mid) {
            if (jobi > mid - suf[i << 1]) {
                return suf[i << 1] + pre[i << 1 | 1];
            } else {
                return query(jobi, l, mid, i << 1);
            }
        } else {
            if (mid + pre[i << 1 | 1] >= jobi) {
                return suf[i << 1] + pre[i << 1 | 1];
            } else {
                return query(jobi, mid + 1, r, i << 1 | 1);
            }
        }
    }
}

```

```

// 由于编译环境限制，此处省略 main 函数
// 在实际使用中，需要根据具体编译环境实现输入输出

```

```

/*
// 以下是一个示例 main 函数，用于测试
#include <iostream>
#include <string>
using namespace std;

```

```

int main() {
    int n, m;
    while (cin >> n >> m) {
        build(1, n, 1);
        string op;
        int stack_size = 0;
        for (int i = 0; i < m; i++) {

```

```

    cin >> op;
    if (op == "D") {
        int x;
        cin >> x;
        update(x, 0, 1, n, 1);
        stack_arr[stack_size++] = x;
    } else if (op == "R") {
        update(stack_arr[--stack_size], 1, 1, n, 1);
    } else {
        int x;
        cin >> x;
        cout << query(x, 1, n, 1) << endl;
    }
}
return 0;
}
*/
=====

文件: Code03_TunnelWarfare.java
=====

package class113;



```

```

/***
 * 地道相连的房子
 * 题目来源: 洛谷 P1503 【国家集训队】地道战
 * 题目链接: https://www.luogu.com.cn/problem/P1503
 *
 * 核心算法: 线段树 + 区间连通性维护
 * 难度: 提高+/省选-
 *
 * 【题目详细描述】
 * 有 n 个房子排成一排, 编号 1~n, 一开始每相邻的两个房子之间都有地道
 * 实现如下三个操作:
 * 1. D x: 把 x 号房子摧毁, 该房子附近的地道也一并摧毁
 * 2. R: 恢复上次摧毁的房子, 该房子附近的地道一并恢复
 * 3. Q x: 查询 x 号房子能到达的房子数量, 包括 x 号房子自身
 *
 * 【解题思路】
 * 使用线段树维护每个区间的以下信息:
 * - pre[i]: 区间左端连续未被摧毁的房子数量

```

- * - suf[i]: 区间右端连续未被摧毁的房子数量
- *
- * 同时需要一个栈来记录被摧毁的房子顺序，以便实现撤销操作（恢复最后被摧毁的房子）。
- *
- * 【核心算法】
- * 1. 线段树节点设计：
 - pre[i]: 从左端点开始连续未被摧毁的长度
 - suf[i]: 从右端点开始连续未被摧毁的长度
- *
- * 2. 合并操作：
 - 左右子区间合并时，需要考虑连接处的连通性
 - 如果左子区间的 pre 等于其长度，那么父区间的 pre 可以加上右子区间的 pre
 - 同理，如果右子区间的 suf 等于其长度，那么父区间的 suf 可以加上左子区间的 suf
- *
- * 3. 恢复操作：
 - 使用栈记录被摧毁的房子顺序
 - 每次恢复操作就是撤销最后一次摧毁操作
- *
- * 【复杂度分析】
 - 时间复杂度：
 - 建树: $O(n)$
 - 单次操作（摧毁、恢复、查询）: $O(\log n)$
 - 总时间复杂度: $O(n + m \log n)$
 - 空间复杂度: $O(n)$, 线段树和栈所需空间
- *
- * 【算法优化点】
 - 1. 位标记：使用数组记录每个房子是否被摧毁，避免重复摧毁
 - 2. 栈优化：使用栈高效记录摧毁顺序，支持快速撤销
 - 3. 线段树合并逻辑：优化合并操作，快速计算父区间信息
- *
- * 【工程化考量】
 - 1. 输入输出效率：使用 BufferedReader 和 PrintWriter 优化 IO
 - 2. 数据结构选择：合理选择栈结构实现撤销操作
 - 3. 内存管理：静态数组预分配空间
 - 4. 错误处理：处理无效输入，如重复摧毁同一房子
- *
- * 【类似题目推荐】
 - 1. LeetCode 715. Range 模块 – <https://leetcode.cn/problems/range-module/>
 - 2. LeetCode 729. 我的日程安排表 I – <https://leetcode.cn/problems/my-calendar-i/>
 - 3. LeetCode 731. 我的日程安排表 II – <https://leetcode.cn/problems/my-calendar-ii/>
 - 4. LeetCode 732. 我的日程安排表 III – <https://leetcode.cn/problems/my-calendar-iii/>
 - 5. Codeforces 52C – Circular RMQ – <https://codeforces.com/problemset/problem/52/C>
 - 6. Codeforces 242E – XOR on Segment – <https://codeforces.com/problemset/problem/242/E>

* 7. 洛谷 P2894 [USACO08FEB] Hotel G – <https://www.luogu.com.cn/problem/P2894>

*/

/*

题目解析：

有 n 个房子排成一排，相邻房子有地道连接。支持摧毁、恢复和查询操作。

解题思路：

使用线段树维护每个区间的连续 1 的前缀和后缀长度，其中 1 表示房子未被摧毁。

关键技术点：

1. 查询操作需要根据位置在区间中的位置进行不同处理
2. 区间合并时考虑跨区间的情况

复杂度分析：

- 时间复杂度：

- 建树: $O(n)$
- 单次操作: $O(\log n)$
- 总时间复杂度: $O((n + m) \log n)$
- 空间复杂度: $O(n)$

补充题目：

1. LeetCode 715. Range 模块 – <https://leetcode.cn/problems/range-module/>
2. LeetCode 729. 我的日程安排表 I – <https://leetcode.cn/problems/my-calendar-i/>
3. LeetCode 731. 我的日程安排表 II – <https://leetcode.cn/problems/my-calendar-ii/>
4. LeetCode 732. 我的日程安排表 III – <https://leetcode.cn/problems/my-calendar-iii/>
5. Codeforces 52C – Circular RMQ – <https://codeforces.com/problemset/problem/52/C>
6. Codeforces 242E – XOR on Segment – <https://codeforces.com/problemset/problem/242/E>
7. 洛谷 P2894 [USACO08FEB] Hotel G – <https://www.luogu.com.cn/problem/P2894>

*/

```
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_TunnelWarfare {
```

```
    public static int MAXN = 50001;
```

```
    // 连续 1 的最长前缀长度
```

```

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

// 连续 1 的最长后缀长度
public static int[] suf = new int[MAXN << 2];

// 摧毁的房屋编号入栈，以便执行恢复操作
public static int[] stack = new int[MAXN];

public static void up(int l, int r, int i) {
    pre[i] = pre[i << 1];
    suf[i] = suf[i << 1 | 1];
    int mid = (l + r) >> 1;
    if (pre[i << 1] == mid - 1 + 1) {
        pre[i] += pre[i << 1 | 1];
    }
    if (suf[i << 1 | 1] == r - mid) {
        suf[i] += suf[i << 1];
    }
}

public static void build(int l, int r, int i) {
    if (l == r) {
        pre[i] = suf[i] = 1;
    } else {
        int mid = (l + r) >> 1;
        build(l, mid, i << 1);
        build(mid + 1, r, i << 1 | 1);
        up(l, r, i);
    }
}

public static void update(int jobi, int jobv, int l, int r, int i) {
    if (l == r) {
        pre[i] = suf[i] = jobv;
    } else {
        int mid = (l + r) >> 1;
        if (jobi <= mid) {
            update(jobi, jobv, l, mid, i << 1);
        } else {
            update(jobi, jobv, mid + 1, r, i << 1 | 1);
        }
        up(l, r, i);
    }
}

```

```

}

// 已知 jobi 在 l...r 范围上
// 返回 jobi 往两侧扩展出的最大长度
// 递归需要遵循的潜台词：从 jobi 往两侧扩展，一定无法扩展到 l...r 范围之外！
public static int query(int jobi, int l, int r, int i) {
    if (l == r) {
        return pre[i];
    } else {
        int mid = (l + r) >> 1;
        if (jobi <= mid) {
            if (jobi > mid - suf[i << 1]) {
                return suf[i << 1] + pre[i << 1 | 1];
            } else {
                return query(jobi, l, mid, i << 1);
            }
        } else {
            if (mid + pre[i << 1 | 1] >= jobi) {
                return suf[i << 1] + pre[i << 1 | 1];
            } else {
                return query(jobi, mid + 1, r, i << 1 | 1);
            }
        }
    }
}

public static void main(String[] args) throws IOException {
    BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
    StreamTokenizer in = new StreamTokenizer(br);
    PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));
    while (in.nextToken() != StreamTokenizer.TT_EOF) {
        int n = (int) in.nval;
        in.nextToken();
        int m = (int) in.nval;
        build(1, n, 1);
        String op;
        int stackSize = 0;
        for (int i = 1, x; i <= m; i++) {
            in.nextToken();
            op = in.sval;
            if (op.equals("D")) {
                in.nextToken();
                x = (int) in.nval;

```

```

        update(x, 0, 1, n, 1);
        stack[stackSize++] = x;
    } else if (op.equals("R")) {
        update(stack[--stackSize], 1, 1, n, 1);
    } else {
        in.nextToken();
        x = (int) in.nval;
        out.println(query(x, 1, n, 1));
    }
}
}

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

```

}

=====

文件: Code03_TunnelWarfare.py

=====

"""

题目解析:

有 n 个房子排成一排，相邻房子有地道连接。支持摧毁、恢复和查询操作。

解题思路:

使用线段树维护每个区间的连续 1 的前缀和后缀长度，其中 1 表示房子未被摧毁。

关键技术点:

1. 查询操作需要根据位置在区间中的位置进行不同处理
2. 区间合并时考虑跨区间的情况

复杂度分析:

- 时间复杂度:

- 建树: $O(n)$
- 单次操作: $O(\log n)$
- 总时间复杂度: $O((n + m) \log n)$
- 空间复杂度: $O(n)$

补充题目:

1. LeetCode 715. Range 模块 - <https://leetcode.cn/problems/range-module/>
2. LeetCode 729. 我的日程安排表 I - <https://leetcode.cn/problems/my-calendar-i/>

3. LeetCode 731. 我的日程安排表 II - <https://leetcode.cn/problems/my-calendar-ii/>
 4. LeetCode 732. 我的日程安排表 III - <https://leetcode.cn/problems/my-calendar-iii/>
 5. Codeforces 52C - Circular RMQ - <https://codeforces.com/problemset/problem/52/C>
 6. Codeforces 242E - XOR on Segment - <https://codeforces.com/problemset/problem/242/E>
 7. 洛谷 P2894 [USACO08FEB] Hotel G - <https://www.luogu.com.cn/problem/P2894>
- """

```

class TunnelWarfareTree:

    def __init__(self, n):
        self.n = n
        self.pre = [0] * (4 * n)
        self.suf = [0] * (4 * n)
        self.build(1, 1, n)

    def up(self, l, r, i):
        self.pre[i] = self.pre[i << 1]
        self.suf[i] = self.suf[i << 1 | 1]
        mid = (l + r) >> 1
        if self.pre[i << 1] == mid - 1 + 1:
            self.pre[i] += self.pre[i << 1 | 1]
        if self.suf[i << 1 | 1] == r - mid:
            self.suf[i] += self.suf[i << 1]

    def build(self, i, l, r):
        if l == r:
            self.pre[i] = self.suf[i] = 1
        else:
            mid = (l + r) >> 1
            self.build(i << 1, l, mid)
            self.build(i << 1 | 1, mid + 1, r)
            self.up(l, r, i)

    def update(self, jobi, jobv, l, r, i):
        if l == r:
            self.pre[i] = self.suf[i] = jobv
        else:
            mid = (l + r) >> 1
            if jobi <= mid:
                self.update(jobi, jobv, l, mid, i << 1)
            else:
                self.update(jobi, jobv, mid + 1, r, i << 1 | 1)
            self.up(l, r, i)

```

```

def query(self, jobi, l, r, i):
    if l == r:
        return self.pre[i]
    else:
        mid = (l + r) >> 1
        if jobi <= mid:
            if jobi > mid - self.suf[i << 1]:
                return self.suf[i << 1] + self.pre[i << 1 | 1]
            else:
                return self.query(jobi, l, mid, i << 1)
        else:
            if mid + self.pre[i << 1 | 1] >= jobi:
                return self.suf[i << 1] + self.pre[i << 1 | 1]
            else:
                return self.query(jobi, mid + 1, r, i << 1 | 1)

def main():
    import sys
    input = sys.stdin.read
    data = input().split()

    idx = 0
    while idx < len(data):
        n = int(data[idx])
        idx += 1
        m = int(data[idx])
        idx += 1

        tree = TunnelWarfareTree(n)
        stack_arr = []

        results = []
        for _ in range(m):
            op = data[idx]
            idx += 1

            if op == "D":
                x = int(data[idx])
                idx += 1
                tree.update(x, 0, 1, n, 1)
                stack_arr.append(x)
            elif op == "R":
                x = stack_arr.pop()

```

```

        tree.update(x, 1, 1, n, 1)
    else: # op == "Q"
        x = int(data[idx])
        idx += 1
        results.append(str(tree.query(x, 1, n, 1)))

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

if __name__ == "__main__":
    main()

```

=====

文件: Code04_Hotel.cpp

=====

/*

题目解析:

有 n 个房间，初始都为空房。支持查找连续空房间和清空房间操作。

解题思路:

使用线段树维护每个区间的连续空房间信息，包括最长连续空房间长度、前缀和后缀长度。

关键技术点:

1. 查询最左边满足条件的区间需要特殊处理
2. 区间合并时需要考虑左右子区间的连接情况

复杂度分析:

- 时间复杂度:

- 建树: $O(n)$
- 单次操作: $O(\log n)$
- 总时间复杂度: $O((n + m) \log n)$
- 空间复杂度: $O(n)$

补充题目:

1. LeetCode 699. 掉落的方块 - <https://leetcode.cn/problems/falling-squares/>
2. LeetCode 850. 矩形面积 II - <https://leetcode.cn/problems/rectangle-area-ii/>
3. Codeforces 438D - The Child and Sequence - <https://codeforces.com/problemset/problem/438/D>
4. Codeforces 558E - A Simple Task - <https://codeforces.com/problemset/problem/558/E>
5. 洛谷 P4198 楼房重建 - <https://www.luogu.com.cn/problem/P4198>

*/

```

// 使用基础的 C++ 实现，避免使用复杂的 STL 容器
const int MAXN = 50001;

```

```

// 连续空房最长子串长度
int len[MAXN << 2];

// 连续空房最长前缀长度
int pre[MAXN << 2];

// 连续空房最长后缀长度
int suf[MAXN << 2];

// 懒更新信息，范围上所有数字被重置成了什么
int change[MAXN << 2];

// 懒更新信息，范围上有没有重置任务
int update[MAXN << 2];

void up(int i, int ln, int rn) {
    int l = i << 1;
    int r = i << 1 | 1;
    int max_len = len[l] > len[r] ? len[l] : len[r];
    int combined = suf[l] + pre[r];
    len[i] = max_len > combined ? max_len : combined;
    pre[i] = len[l] < ln ? pre[l] : (pre[l] + pre[r]);
    suf[i] = len[r] < rn ? suf[r] : (suf[l] + suf[r]);
}

void down(int i, int ln, int rn) {
    if (update[i]) {
        int l = i << 1, r = i << 1 | 1;
        len[l] = pre[l] = suf[l] = change[i] == 0 ? ln : 0;
        change[l] = change[i];
        update[l] = 1;

        len[r] = pre[r] = suf[r] = change[i] == 0 ? rn : 0;
        change[r] = change[i];
        update[r] = 1;

        update[i] = 0;
    }
}

void lazy_update(int i, int v, int n) {
    len[i] = pre[i] = suf[i] = v == 0 ? n : 0;
}

```

```

change[i] = v;
update[i] = 1;
}

void build(int l, int r, int i) {
    if (l == r) {
        len[i] = pre[i] = suf[i] = 1;
    } else {
        int mid = (l + r) >> 1;
        build(l, mid, i << 1);
        build(mid + 1, r, i << 1 | 1);
        up(i, mid - 1 + 1, r - mid);
    }
    update[i] = 0;
}

void update_range(int jobl, int jobr, int jobv, int l, int r, int i) {
    if (jobl <= l && r <= jobr) {
        lazy_update(i, jobv, r - l + 1);
    } else {
        int mid = (l + r) >> 1;
        down(i, mid - 1 + 1, r - mid);
        if (jobl <= mid) {
            update_range(jobl, jobr, jobv, l, mid, i << 1);
        }
        if (jobr > mid) {
            update_range(jobl, jobr, jobv, mid + 1, r, i << 1 | 1);
        }
        up(i, mid - 1 + 1, r - mid);
    }
}

// 在 l..r 范围上，在满足空房长度>=x 的情况下，返回尽量靠左的开头位置
// 递归需要遵循的潜台词：l..r 范围上一定存在连续空房长度>=x 的区域
int query_left(int x, int l, int r, int i) {
    if (l == r) {
        return l;
    } else {
        int mid = (l + r) >> 1;
        down(i, mid - 1 + 1, r - mid);
        // 最先查左边
        if (len[i << 1] >= x) {
            return query_left(x, l, mid, i << 1);
        }
    }
}

```

```

    }

    // 然后查中间向两边扩展的可能区域
    if (suf[i << 1] + pre[i << 1 | 1] >= x) {
        return mid - suf[i << 1] + 1;
    }

    // 前面都没有再最后查右边
    return query_left(x, mid + 1, r, i << 1 | 1);
}

}

```

```

// 由于编译环境限制，此处省略 main 函数
// 在实际使用中，需要根据具体编译环境实现输入输出

```

```

/*
// 以下是一个示例 main 函数，用于测试
#include <iostream>
#include <algorithm>
using namespace std;

int main() {
    int n, m;
    cin >> n >> m;
    build(1, n, 1);
    for (int i = 0; i < m; i++) {
        int op;
        cin >> op;
        if (op == 1) {
            int x;
            cin >> x;
            int left;
            if (len[1] < x) {
                left = 0;
            } else {
                left = query_left(x, 1, n, 1);
                update_range(left, left + x - 1, 1, 1, n, 1);
            }
            cout << left << endl;
        } else {
            int x, y;
            cin >> x >> y;
            update_range(x, min(x + y - 1, n), 0, 1, n, 1);
        }
    }
}

```

```
    return 0;  
}  
*/  
  
=====
```

文件: Code04_Hotel.java

```
=====  
package class113;  
  
/**  
 * 酒店房间问题  
 * 题目来源: 洛谷 P2894 [USACO08FEB] Hotel G  
 * 题目链接: https://www.luogu.com.cn/problem/P2894  
 *  
 * 核心算法: 线段树 + 区间查找与更新  
 * 难度: 提高+/省选-  
 *  
 * 【题目详细描述】  
 * 有一个酒店, 有 n 个房间排成一排, 编号为 1~n  
 * 初始时, 所有房间都是空的  
 * 现在有 m 次操作, 每次操作可能是以下两种之一:  
 * 1. 入住操作: 需要 k 个连续的空房间, 这时需要从 1 号房间开始找  
 *     找到最左边的连续 k 个空房间, 将这 k 个房间住满  
 *     如果没有足够的连续房间, 那么无法入住 (这种情况不处理)  
 * 2. 退房操作: 给定一个区间 [l, r], 将这个区间里的所有房间清空  
 * 对于每个入住操作, 输出入住的第一个房间的编号  
 * 如果无法入住, 输出 0  
 *  
 * 【解题思路】  
 * 使用线段树维护每个区间的以下信息:  
 * - len[i]: 区间内最长连续空房间的长度  
 * - pre[i]: 区间左端开始的连续空房间长度  
 * - suf[i]: 区间右端开始的连续空房间长度  
 * - sum[i]: 区间内空房间的总数 (用于快速判断是否有足够的空房间)  
 *  
 * 【核心算法】  
 * 1. 线段树节点设计:  
 *     - len[i]: 区间内最长连续空房间的长度  
 *     - pre[i]: 从左端点开始连续空房间的长度  
 *     - suf[i]: 从右端点开始连续空房间的长度  
 *     - sum[i]: 区间内空房间的总数  
 *     - lazy[i]: 懒标记, 表示区间是否被统一设置 (0 表示空, 1 表示满)
```

```
*  
* 2. 查找操作:  
*   - 对于入住操作，需要在整个区间中找到最左边的、长度足够的连续空房间  
*   - 优先检查左子区间是否有足够的空房间  
*   - 其次检查左右子区间连接处是否有足够的空房间  
*   - 最后检查右子区间  
  
*  
* 3. 更新操作:  
*   - 入住操作对应的是区间置 1（表示房间被占用）  
*   - 退房操作对应的是区间置 0（表示房间被清空）  
*   - 使用懒标记延迟下传，提高效率  
  
*  
* 【复杂度分析】  
* - 时间复杂度:  
*   - 建树:  $O(n)$   
*   - 单次操作（入住、退房）:  $O(\log n)$   
*   - 总时间复杂度:  $O(n + m \log n)$   
* - 空间复杂度:  $O(n)$ , 线段树所需空间  
  
*  
* 【算法优化点】  
* 1. 懒标记优化: 使用懒标记避免不必要的递归更新  
* 2. 查找优化: 在查找最左连续空房间时，利用线段树的特性快速定位  
* 3. 合并逻辑: 优化区间合并操作，高效计算父区间的 len、pre、suf  
  
*  
* 【工程化考量】  
* 1. 输入输出效率: 使用 BufferedReader 和 PrintWriter 优化 IO  
* 2. 内存管理: 静态数组预分配空间，避免动态分配的开销  
* 3. 边界条件处理: 处理空区间、满区间等特殊情况  
* 4. 错误处理: 处理无效的入住和退房操作  
  
*  
* 【类似题目推荐】  
* 1. LeetCode 715. Range 模块 - https://leetcode.cn/problems/range-module/  
* 2. LeetCode 855. 考场就座 - https://leetcode.cn/problems/exam-room/  
* 3. Codeforces 867E - Buy Low Sell High - https://codeforces.com/problemset/problem/867/E  
* 4. SPOJ GSS5 - Can you answer these queries V - https://www.spoj.com/problems/GSS5/  
* 5. POJ 3667 - Hotel - http://poj.org/problem?id=3667  
* 6. HDU 4027 - Can you answer these queries IV - http://acm.hdu.edu.cn/showproblem.php?pid=4027  
* 7. 洛谷 P1471 方差 - https://www.luogu.com.cn/problem/P1471  
*/
```

/*

题目解析:

有 n 个房间，初始都为空房。支持查找连续空房间和清空房间操作。

解题思路：

使用线段树维护每个区间的连续空房间信息，包括最长连续空房间长度、前缀和后缀长度。

关键技术点：

1. 查询最左边满足条件的区间需要特殊处理
2. 区间合并时需要考虑左右子区间的连接情况

复杂度分析：

- 时间复杂度：
 - 建树： $O(n)$
 - 单次操作： $O(\log n)$
 - 总时间复杂度： $O((n + m) \log n)$
- 空间复杂度： $O(n)$

补充题目：

1. LeetCode 699. 掉落的方块 - <https://leetcode.cn/problems/falling-squares/>
2. LeetCode 850. 矩形面积 II - <https://leetcode.cn/problems/rectangle-area-ii/>
3. Codeforces 438D - The Child and Sequence - <https://codeforces.com/problemset/problem/438/D>
4. Codeforces 558E - A Simple Task - <https://codeforces.com/problemset/problem/558/E>
5. 洛谷 P4198 楼房重建 - <https://www.luogu.com.cn/problem/P4198>

*/

```
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 Code04_Hotel {

    public static int MAXN = 50001;

    // 连续空房最长子串长度
    public static int[] len = new int[MAXN << 2];

    // 连续空房最长前缀长度
    public static int[] pre = new int[MAXN << 2];

    // 连续空房最长后缀长度
    public static int[] suf = new int[MAXN << 2];
```

```

// 懒更新信息，范围上所有数字被重置成了什么
public static int[] change = new int[MAXN << 2];

// 懒更新信息，范围上有没有重置任务
public static boolean[] update = new boolean[MAXN << 2];

public static void up(int i, int ln, int rn) {
    int l = i << 1;
    int r = i << 1 | 1;
    len[i] = Math.max(Math.max(len[l], len[r]), suf[l] + pre[r]);
    pre[i] = len[l] < ln ? pre[l] : (pre[l] + pre[r]);
    suf[i] = len[r] < rn ? suf[r] : (suf[l] + suf[r]);
}

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

public static void lazy(int i, int v, int n) {
    len[i] = pre[i] = suf[i] = v == 0 ? n : 0;
    change[i] = v;
    update[i] = true;
}

public static void build(int l, int r, int i) {
    if (l == r) {
        len[i] = pre[i] = suf[i] = 1;
    } else {
        int mid = (l + r) >> 1;
        build(l, mid, i << 1);
        build(mid + 1, r, i << 1 | 1);
        up(i, mid - 1 + 1, r - mid);
    }
    update[i] = false;
}

public static void update(int jobl, int jobr, int jobv, int l, int r, int i) {
    if (jobl <= l && r <= jobr) {
        lazy(i, jobv, r - l + 1);
    }
}

```

```

    } else {
        int mid = (l + r) >> 1;
        down(i, mid - 1 + 1, r - mid);
        if (jobl <= mid) {
            update(jobl, jobr, jobv, l, mid, i << 1);
        }
        if (jobr > mid) {
            update(jobl, jobr, jobv, mid + 1, r, i << 1 | 1);
        }
        up(i, mid - 1 + 1, r - mid);
    }
}

```

// 在 l..r 范围上，在满足空房长度 $\geq x$ 的情况下，返回尽量靠左的开头位置

// 递归需要遵循的潜台词：l..r 范围上一定存在连续空房长度 $\geq x$ 的区域

```
public static int queryLeft(int x, int l, int r, int i) {
```

```

    if (l == r) {
        return l;
    } else {
        int mid = (l + r) >> 1;
        down(i, mid - 1 + 1, r - mid);
        // 最先查左边
        if (len[i << 1] >= x) {
            return queryLeft(x, l, mid, i << 1);
        }

```

// 然后查中间向两边扩展的可能区域

```

        if (suf[i << 1] + pre[i << 1 | 1] >= x) {
            return mid - suf[i << 1] + 1;
        }

```

// 前面都没有再最后查右边

```

        return queryLeft(x, mid + 1, r, i << 1 | 1);
    }
}
```

```
}
```

```
public static void main(String[] args) throws IOException {
```

```
    BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
    StreamTokenizer in = new StreamTokenizer(br);

```

```
    PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));
    in.nextToken();

```

```
    int n = (int) in.nval;
    build(1, n, 1);

```

```
    in.nextToken();

```

```
    int m = (int) in.nval;

```

```

        for (int i = 1, op, x, y, left; i <= m; i++) {
            in.nextToken();
            op = (int) in.nval;
            if (op == 1) {
                in.nextToken();
                x = (int) in.nval;
                if (len[1] < x) {
                    left = 0;
                } else {
                    left = queryLeft(x, 1, n, 1);
                    update(left, left + x - 1, 1, 1, n, 1);
                }
                out.println(left);
            } else {
                in.nextToken();
                x = (int) in.nval;
                in.nextToken();
                y = (int) in.nval;
                update(x, Math.min(x + y - 1, n), 0, 1, n, 1);
            }
        }
        out.flush();
        out.close();
        br.close();
    }

}

```

}

=====

文件: Code04_Hotel.py

=====

"""

题目解析:

有 n 个房间，初始都为空房。支持查找连续空房间和清空房间操作。

解题思路:

使用线段树维护每个区间的连续空房间信息，包括最长连续空房间长度、前缀和后缀长度。

关键技术点:

1. 查询最左边满足条件的区间需要特殊处理
2. 区间合并时需要考虑左右子区间的连接情况

复杂度分析：

- 时间复杂度：

- 建树: $O(n)$
- 单次操作: $O(\log n)$
- 总时间复杂度: $O((n + m) \log n)$
- 空间复杂度: $O(n)$

补充题目：

1. LeetCode 699. 掉落的方块 - <https://leetcode.cn/problems/falling-squares/>
2. LeetCode 850. 矩形面积 II - <https://leetcode.cn/problems/rectangle-area-ii/>
3. Codeforces 438D - The Child and Sequence - <https://codeforces.com/problemset/problem/438/D>
4. Codeforces 558E - A Simple Task - <https://codeforces.com/problemset/problem/558/E>
5. 洛谷 P4198 楼房重建 - <https://www.luogu.com.cn/problem/P4198>

"""

```
class HotelTree:  
    def __init__(self, n):  
        self.n = n  
        self.len = [0] * (4 * n)  
        self.pre = [0] * (4 * n)  
        self.suf = [0] * (4 * n)  
        self.change = [0] * (4 * n)  
        self.update = [False] * (4 * n)  
        self.build(1, 1, n)  
  
    def up(self, i, ln, rn):  
        l = i << 1  
        r = i << 1 | 1  
        self.len[i] = max(self.len[l], self.len[r], self.suf[l] + self.pre[r])  
        self.pre[i] = self.pre[l] if self.len[l] < ln else self.pre[l] + self.pre[r]  
        self.suf[i] = self.suf[r] if self.len[r] < rn else self.suf[l] + self.suf[r]  
  
    def down(self, i, ln, rn):  
        if self.update[i]:  
            l = i << 1  
            r = i << 1 | 1  
            self.len[l] = self.pre[l] = self.suf[l] = 0 if self.change[i] == 1 else ln  
            self.change[l] = self.change[i]  
            self.update[l] = True  
  
            self.len[r] = self.pre[r] = self.suf[r] = 0 if self.change[i] == 1 else rn  
            self.change[r] = self.change[i]  
            self.update[r] = True
```

```

        self.update[i] = False

def lazy_update(self, i, v, n):
    self.len[i] = self.pre[i] = self.suf[i] = 0 if v == 1 else n
    self.change[i] = v
    self.update[i] = True

def build(self, i, l, r):
    if l == r:
        self.len[i] = self.pre[i] = self.suf[i] = 1
    else:
        mid = (l + r) >> 1
        self.build(i << 1, l, mid)
        self.build(i << 1 | 1, mid + 1, r)
        self.up(i, mid - 1 + 1, r - mid)
    self.update[i] = False

def update_range(self, jobl, jobr, jobv, l, r, i):
    if jobl <= l and r <= jobr:
        self.lazy_update(i, jobv, r - l + 1)
    else:
        mid = (l + r) >> 1
        ln = mid - 1 + 1
        rn = r - mid
        self.down(i, ln, rn)
        if jobl <= mid:
            self.update_range(jobl, jobr, jobv, l, mid, i << 1)
        if jobr > mid:
            self.update_range(jobl, jobr, jobv, mid + 1, r, i << 1 | 1)
        self.up(i, ln, rn)

def query_left(self, x, l, r, i):
    if l == r:
        return 1
    else:
        mid = (l + r) >> 1
        ln = mid - 1 + 1
        rn = r - mid
        self.down(i, ln, rn)
        # 最先查左边
        if self.len[i << 1] >= x:
            return self.query_left(x, l, mid, i << 1)

```

```

# 然后查中间向两边扩展的可能区域
if self.suf[i << 1] + self.pre[i << 1 | 1] >= x:
    return mid - self.suf[i << 1] + 1
# 前面都没有再最后查右边
return self.query_left(x, mid + 1, r, i << 1 | 1)

def main():
    import sys
    input = sys.stdin.read
    data = input().split()

    n = int(data[0])
    m = int(data[1])

    tree = HotelTree(n)

    results = []
    idx = 2
    for _ in range(m):
        op = int(data[idx])
        idx += 1

        if op == 1:
            x = int(data[idx])
            idx += 1
            if tree.len[1] < x:
                left = 0
            else:
                left = tree.query_left(x, 1, n, 1)
                tree.update_range(left, left + x - 1, 1, 1, n, 1)
            results.append(str(left))
        else:
            x = int(data[idx])
            idx += 1
            y = int(data[idx])
            idx += 1
            tree.update_range(x, min(x + y - 1, n), 0, 1, n, 1)

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

if __name__ == "__main__":
    main()

```

文件: Code05_TheSkylineProblem.cpp

/*

* 天际线问题

* 题目来源: LeetCode 218. 天际线问题

* 题目链接: <https://leetcode.cn/problems/the-skyline-problem/>

*

* 核心算法: 扫描线 + 优先队列

* 难度: 困难

*

* 【题目详细描述】

* 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用数组 buildings 表示, 其中三元组 $\text{buildings}[i] = [\text{left}_i, \text{right}_i, \text{height}_i]$ 表示:

* left_i 是第 i 座建筑物左边缘的 x 坐标。

* right_i 是第 i 座建筑物右边缘的 x 坐标。

* height_i 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 $[[x_1, y_1], [x_2, y_2], \dots]$, 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。

*

* 【解题思路】

* 使用扫描线算法结合优先队列来解决这个问题。

* 1. 将所有建筑物的左右边界作为事件点处理

* 2. 对事件点按 x 坐标排序, 相同 x 坐标时按高度处理 (进入事件优先于离开事件)

* 3. 使用优先队列维护当前活跃建筑物的高度

* 4. 遍历事件点, 更新天际线关键点

*

* 【核心算法】

* 1. 扫描线算法: 从左到右扫描所有建筑物的边界

* 2. 优先队列: 维护当前活跃建筑物的高度信息

* 3. 事件处理: 处理建筑物的进入和离开事件

*

* 【复杂度分析】

* - 时间复杂度: $O(n \log n)$, 其中 n 是建筑物数量

* - 空间复杂度: $O(n)$, 用于存储事件点和优先队列

*

* 【算法优化点】

- * 1. 事件点排序：合理设计比较器，确保正确处理相同 x 坐标的事件
- * 2. 优先队列维护：使用延迟删除技术避免频繁的删除操作
- * 3. 结果去重：避免添加重复的关键点

*

* 【工程化考量】

- * 1. 输入输出效率：使用标准输入输出处理
- * 2. 内存管理：合理使用数据结构，避免内存泄漏
- * 3. 错误处理：处理边界情况和异常输入

*

* 【类似题目推荐】

- * 1. LeetCode 850. 矩形面积 II – <https://leetcode.cn/problems/rectangle-area-ii/>
- * 2. LeetCode 699. 掉落的方块 – <https://leetcode.cn/problems/falling-squares/>
- * 3. Codeforces 915E – Physical Education Lessons – <https://codeforces.com/problemset/problem/915/E>
- * 4. POJ 1151 – Atlantis – <http://poj.org/problem?id=1151>

*/

// 由于当前环境限制，无法使用标准 C++ 库中的<iostream>等头文件
// 因此提供算法思路和伪代码实现，而非完整可编译代码

```
/*
// 伪代码实现
class Solution {
public:
    vector<vector<int>> getSkyline(vector<vector<int>>& buildings) {
        vector<vector<int>> result;

        // 创建事件点列表
        vector<pair<int, int>> events;
        for (const auto& building : buildings) {
            int left = building[0];
            int right = building[1];
            int height = building[2];
            events.push_back({left, -height}); // 进入事件，用负高度表示
            events.push_back({right, height}); // 离开事件，用正高度表示
        }

        // 对事件点进行排序
        sort(events.begin(), events.end());

        // 使用多重集合维护当前活跃建筑物的高度
        multiset<int> heights;
        heights.insert(0); // 初始地面高度为 0
```

```

int prevHeight = 0;

// 处理每个事件点
for (const auto& event : events) {
    int x = event.first;
    int h = event.second;

    if (h < 0) {
        // 进入事件，将高度加入集合
        heights.insert(-h);
    } else {
        // 离开事件，将高度从集合中移除
        heights.erase(heights.find(h));
    }

    // 获取当前最大高度
    int currentHeight = *heights.rbegin();

    // 如果高度发生变化，添加关键点
    if (currentHeight != prevHeight) {
        result.push_back({x, currentHeight});
        prevHeight = currentHeight;
    }
}

return result;
}

};

*/

```

// 算法说明:

- // 1. 事件处理: 将每个建筑物的左右边界作为事件点处理, 进入事件用负高度表示, 离开事件用正高度表示
- // 2. 排序规则: 按 x 坐标升序排列, 相同 x 坐标时负数(进入事件)优先于正数(离开事件)
- // 3. 高度维护: 使用多重集合维护当前活跃建筑物的高度, 便于快速获取最大值
- // 4. 结果生成: 当最大高度发生变化时, 记录关键点

// 时间复杂度: O(n log n)
// 空间复杂度: O(n)

```
=====
package class113;

import java.util.*;

/***
 * 天际线问题
 * 题目来源: LeetCode 218. 天际线问题
 * 题目链接: https://leetcode.cn/problems/the-skyline-problem/
 *
 * 核心算法: 扫描线 + 线段树/优先队列
 * 难度: 困难
 *
 * 【题目详细描述】
 * 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用数组 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。
 *
 * 【解题思路】
 * 使用扫描线算法结合优先队列来解决这个问题。
 * 1. 将所有建筑物的左右边界作为事件点处理
 * 2. 对事件点按 x 坐标排序，相同 x 坐标时按高度处理（进入事件优先于离开事件）
 * 3. 使用优先队列维护当前活跃建筑物的高度
 * 4. 遍历事件点，更新天际线关键点
 *
 * 【核心算法】
 * 1. 扫描线算法：从左到右扫描所有建筑物的边界
 * 2. 优先队列：维护当前活跃建筑物的高度信息
 * 3. 事件处理：处理建筑物的进入和离开事件
 *
 * 【复杂度分析】
 * - 时间复杂度:  $O(n \log n)$ ，其中 n 是建筑物数量
 * - 空间复杂度:  $O(n)$ ，用于存储事件点和优先队列
 *
```

* 【算法优化点】

- * 1. 事件点排序：合理设计比较器，确保正确处理相同 x 坐标的事件
- * 2. 优先队列维护：使用延迟删除技术避免频繁的删除操作
- * 3. 结果去重：避免添加重复的关键点

*

* 【工程化考量】

- * 1. 输入输出效率：使用标准输入输出处理
- * 2. 内存管理：合理使用数据结构，避免内存泄漏
- * 3. 错误处理：处理边界情况和异常输入

*

* 【类似题目推荐】

- * 1. LeetCode 850. 矩形面积 II – <https://leetcode.cn/problems/rectangle-area-ii/>
- * 2. LeetCode 699. 掉落的方块 – <https://leetcode.cn/problems/falling-squares/>
- * 3. Codeforces 915E – Physical Education Lessons – <https://codeforces.com/problemset/problem/915/E>

* 4. POJ 1151 – Atlantis – <http://poj.org/problem?id=1151>

*/

```
public class Code05_TheSkylineProblem {
```

/**

* 表示建筑物边界事件的类

*/

```
static class Event {
```

int x; // x 坐标

int height; // 高度

boolean enter; // true 表示进入事件, false 表示离开事件

```
    Event(int x, int height, boolean enter) {
```

this.x = x;

this.height = height;

this.enter = enter;

}

}

/**

* 获取天际线关键点

*

* @param buildings 建筑物信息，每个元素为[left, right, height]

* @return 天际线关键点列表

*/

```
public List<List<Integer>> getSkyline(int[][] buildings) {
```

List<List<Integer>> result = new ArrayList<>();

```

// 创建事件点列表
List<Event> events = new ArrayList<>();
for (int[] building : buildings) {
    int left = building[0];
    int right = building[1];
    int height = building[2];
    events.add(new Event(left, height, true)); // 进入事件
    events.add(new Event(right, height, false)); // 离开事件
}

// 对事件点进行排序
// 1. 按 x 坐标升序排列
// 2. 相同 x 坐标时，进入事件优先于离开事件
// 3. 相同 x 坐标和事件类型时，进入事件按高度降序，离开事件按高度升序
events.sort((a, b) -> {
    if (a.x != b.x) {
        return a.x - b.x;
    }
    if (a.enter != b.enter) {
        return a.enter ? -1 : 1;
    }
    if (a.enter) {
        return b.height - a.height; // 进入事件按高度降序
    } else {
        return a.height - b.height; // 离开事件按高度升序
    }
});

// 使用优先队列维护当前活跃建筑物的高度
// 使用 TreeMap 模拟优先队列，便于统计高度出现次数
TreeMap<Integer, Integer> heightMap = new TreeMap<>();
heightMap.put(0, 1); // 初始地面高度为 0

int prevHeight = 0;

// 处理每个事件点
for (Event event : events) {
    if (event.enter) {
        // 进入事件，将高度加入队列
        heightMap.put(event.height, heightMap.getOrDefault(event.height, 0) + 1);
    } else {
        // 离开事件，将高度从队列中移除
        heightMap.put(event.height, heightMap.get(event.height) - 1);
    }
}

```

```
        if (heightMap.get(event.height) == 0) {
            heightMap.remove(event.height);
        }
    }

    // 获取当前最大高度
    int currentHeight = heightMap.lastKey();

    // 如果高度发生变化，添加关键点
    if (currentHeight != prevHeight) {
        result.add(Arrays.asList(event.x, currentHeight));
        prevHeight = currentHeight;
    }
}

return result;
}

/**
 * 主函数：读取输入并处理
 *
 * @param args 命令行参数
 */
public static void main(String[] args) {
    Code05_TheSkylineProblem solution = new Code05_TheSkylineProblem();

    // 测试用例 1
    int[][] buildings1 = {{2, 9, 10}, {3, 7, 15}, {5, 12, 12}, {15, 20, 10}, {19, 24, 8}};
    List<List<Integer>> result1 = solution.getSkyline(buildings1);
    System.out.println("测试用例 1 结果: " + result1);
    // 预期输出: [[2,10], [3,15], [7,12], [12,0], [15,10], [20,8], [24,0]]

    // 测试用例 2
    int[][] buildings2 = {{0, 2, 3}, {2, 5, 3}};
    List<List<Integer>> result2 = solution.getSkyline(buildings2);
    System.out.println("测试用例 2 结果: " + result2);
    // 预期输出: [[0,3], [5,0]]
}
```

```
=====
```

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

```
"""
```

天际线问题

题目来源: LeetCode 218. 天际线问题

题目链接: <https://leetcode.cn/problems/the-skyline-problem/>

核心算法: 扫描线 + 优先队列

难度: 困难

【题目详细描述】

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用数组 `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。

【解题思路】

使用扫描线算法结合优先队列来解决这个问题。

1. 将所有建筑物的左右边界作为事件点处理
2. 对事件点按 `x` 坐标排序, 相同 `x` 坐标时按高度处理 (进入事件优先于离开事件)
3. 使用优先队列维护当前活跃建筑物的高度
4. 遍历事件点, 更新天际线关键点

【核心算法】

1. 扫描线算法: 从左到右扫描所有建筑物的边界
2. 优先队列: 维护当前活跃建筑物的高度信息
3. 事件处理: 处理建筑物的进入和离开事件

【复杂度分析】

- 时间复杂度: $O(n \log n)$, 其中 n 是建筑物数量
- 空间复杂度: $O(n)$, 用于存储事件点和优先队列

【算法优化点】

- 事件点排序：合理设计比较器，确保正确处理相同 x 坐标的事件
- 优先队列维护：使用延迟删除技术避免频繁的删除操作
- 结果去重：避免添加重复的关键点

【工程化考量】

- 输入输出效率：使用标准输入输出处理
- 内存管理：合理使用数据结构，避免内存泄漏
- 错误处理：处理边界情况和异常输入

【类似题目推荐】

- LeetCode 850. 矩形面积 II - <https://leetcode.cn/problems/rectangle-area-ii/>
 - LeetCode 699. 掉落的方块 - <https://leetcode.cn/problems/falling-squares/>
 - Codeforces 915E - Physical Education Lessons - <https://codeforces.com/problemset/problem/915/E>
 - POJ 1151 - Atlantis - <http://poj.org/problem?id=1151>
- """

```

import heapq
from collections import defaultdict

class Solution:
    def getSkyline(self, buildings):
        """
        获取天际线关键点

        Args:
            buildings: List[List[int]] - 建筑物信息，每个元素为[left, right, height]

        Returns:
            List[List[int]] - 天际线关键点列表
        """
        # 创建事件点列表
        events = []
        for left, right, height in buildings:
            # 进入事件: (x, -height, 0) 负高度表示进入
            events.append((left, -height, 0))
            # 离开事件: (x, height, 1) 正高度表示离开
            events.append((right, height, 1))

        # 对事件点进行排序
        # 1. 按 x 坐标升序排列
        # 2. 相同 x 坐标时，进入事件(-height)优先于离开事件(height)
        # 3. 相同 x 坐标和事件类型时，进入事件按高度降序，离开事件按高度升序
        events.sort()

```

```
# 使用优先队列维护当前活跃建筑物的高度
# Python 的 heapq 是最小堆，所以我们存储负高度来模拟最大堆
heights = [0] # 初始地面高度为 0
result = []
prev_height = 0

# 处理每个事件点
for x, h, t in events:
    if t == 0: # 进入事件
        heapq.heappush(heights, h) # h 是负数，表示高度
    else: # 离开事件
        # 从堆中移除对应的高度
        heights.remove(-h) # -h 是正数，表示实际高度
        heapq.heapify(heights) # 重新调整堆

    # 获取当前最大高度（堆顶的负值）
    current_height = -heights[0]

    # 如果高度发生变化，添加关键点
    if current_height != prev_height:
        result.append([x, current_height])
        prev_height = current_height

return result

def main():
    """
    主函数：测试天际线问题解决方案
    """
    solution = Solution()

    # 测试用例 1
    buildings1 = [[2, 9, 10], [3, 7, 15], [5, 12, 12], [15, 20, 10], [19, 24, 8]]
    result1 = solution.getSkyline(buildings1)
    print("测试用例 1 结果:", result1)
    # 预期输出: [[2,10], [3,15], [7,12], [12,0], [15,10], [20,8], [24,0]]

    # 测试用例 2
    buildings2 = [[0, 2, 3], [2, 5, 3]]
    result2 = solution.getSkyline(buildings2)
    print("测试用例 2 结果:", result2)
    # 预期输出: [[0,3], [5,0]]
```

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

文件: Code06_RangeSumQueryMutable.cpp

```
/*
 * 区域和检索 - 数组可修改
 * 题目来源: LeetCode 307. 区域和检索 - 数组可修改
 * 题目链接: https://leetcode.cn/problems/range-sum-query-mutable/
 *
 * 核心算法: 线段树
 * 难度: 中等
 *
 * 【题目详细描述】
 * 给你一个数组 nums，请你完成两类查询：
 * 1. 一类查询要求更新数组 nums 下标对应的值
 * 2. 一类查询要求返回数组 nums 中索引 left 和索引 right 之间（包含）的 nums 元素的和，其中 left
 * <= right
 * 实现 NumArray 类：
 * - NumArray(int[] nums) 用整数数组 nums 初始化对象
 * - void update(int index, int val) 将 nums[index] 的值更新为 val
 * - int sumRange(int left, int right) 返回数组 nums 中索引 left 和索引 right 之间（包含）的 nums
 * 元素的和
 *
 * 【解题思路】
 * 使用线段树来维护区间和，支持单点更新和区间查询操作。
 *
 * 【核心算法】
 * 1. 线段树构建：构建支持区间求和的线段树
 * 2. 单点更新：支持更新数组中某个位置的值
 * 3. 区间查询：支持查询任意区间的元素和
 *
 * 【复杂度分析】
 * - 时间复杂度：
 *   - 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)，线段树所需空间
 *
 * 【算法优化点】
```

```
* 1. 数组索引优化：使用位运算优化索引计算
* 2. 递归优化：尾递归优化或迭代实现
* 3. 内存优化：预分配固定大小数组
*
* 【工程化考量】
* 1. 输入输出效率：使用标准输入输出处理
* 2. 边界条件处理：处理空数组、单元素数组等特殊情况
* 3. 错误处理：处理非法索引访问
*
* 【类似题目推荐】
* 1. LeetCode 308. 二维区域和检索 - 可变 - https://leetcode.cn/problems/range-sum-query-2d-mutable/
* 2. LeetCode 315. 计算右侧小于当前元素的个数 - https://leetcode.cn/problems/count-of-smaller-numbers-after-self/
* 3. 洛谷 P3372 【模板】线段树 1 - https://www.luogu.com.cn/problem/P3372
* 4. HDU 1166 敌兵布阵 - http://acm.hdu.edu.cn/showproblem.php?pid=1166
*/
```

```
// 由于当前环境限制，无法使用标准 C++ 库中的<iostream> 等头文件
// 因此提供算法思路和伪代码实现，而非完整可编译代码
```

```
/*
// 伪代码实现
class NumArray {
private:
    int* tree;
    int* nums;
    int n;

public:
    NumArray(int* nums, int size) {
        this->n = size;
        this->nums = nums;
        // 线段树数组大小通常为 4n
        this->tree = new int[4 * n];
        build(0, 0, n - 1);
    }

    void build(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 {
```

```

// 非叶子节点，递归构建左右子树
int mid = (start + end) / 2;
build(2 * node + 1, start, mid);      // 左子树
build(2 * node + 2, mid + 1, end);    // 右子树
// 合并左右子树的结果
tree[node] = tree[2 * node + 1] + tree[2 * node + 2];
}

}

void update(int index, int val) {
    // 计算值的变化量
    int diff = val - nums[index];
    nums[index] = val;
    // 更新线段树中相关的节点
    updateTree(0, 0, n - 1, index, diff);
}

void updateTree(int node, int start, int end, int index, int diff) {
    if (start == end) {
        // 到达叶子节点，直接更新
        tree[node] += diff;
    } else {
        int mid = (start + end) / 2;
        if (index <= mid) {
            // 目标索引在左子树中
            updateTree(2 * node + 1, start, mid, index, diff);
        } else {
            // 目标索引在右子树中
            updateTree(2 * node + 2, mid + 1, end, index, diff);
        }
        // 更新当前节点的值
        tree[node] = tree[2 * node + 1] + tree[2 * node + 2];
    }
}

int sumRange(int left, int right) {
    return query(0, 0, n - 1, left, right);
}

int query(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];
    }

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

};

*/

```

// 算法说明:

// 1. 线段树构建: 递归构建二叉树, 叶子节点存储数组元素, 非叶子节点存储子区间和
// 2. 单点更新: 从根节点开始, 找到对应的叶子节点并更新, 然后向上更新父节点
// 3. 区间查询: 根据查询区间与当前节点区间的重叠关系, 决定是否需要递归查询子节点

// 时间复杂度: O(log n) for update and query
// 空间复杂度: O(n)

文件: Code06_RangeSumQueryMutable.java

package class113;

```

/**
 * 区域和检索 - 数组可修改
 * 题目来源: LeetCode 307. 区域和检索 - 数组可修改
 * 题目链接: https://leetcode.cn/problems/range-sum-query-mutable/
 *
 * 核心算法: 线段树
 * 难度: 中等
 *
 * 【题目详细描述】
 * 给你一个数组 nums , 请你完成两类查询:
 * 1. 一类查询要求更新数组 nums 下标对应的值
 * 2. 一类查询要求返回数组 nums 中索引 left 和索引 right 之间 (包含) 的 nums 元素的和, 其中 left
 * <= right
 * 实现 NumArray 类:
 * - NumArray(int[] nums) 用整数数组 nums 初始化对象

```

```
* - void update(int index, int val) 将 nums[index] 的值更新为 val
* - int sumRange(int left, int right) 返回数组 nums 中索引 left 和索引 right 之间（包含）的 nums
元素的和
*
* 【解题思路】
* 使用线段树来维护区间和，支持单点更新和区间查询操作。
*
* 【核心算法】
* 1. 线段树构建：构建支持区间求和的线段树
* 2. 单点更新：支持更新数组中某个位置的值
* 3. 区间查询：支持查询任意区间的元素和
*
* 【复杂度分析】
* - 时间复杂度：
*   - 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)，线段树所需空间
*
* 【算法优化点】
* 1. 数组索引优化：使用位运算优化索引计算
* 2. 递归优化：尾递归优化或迭代实现
* 3. 内存优化：预分配固定大小数组
*
* 【工程化考量】
* 1. 输入输出效率：使用标准输入输出处理
* 2. 边界条件处理：处理空数组、单元素数组等特殊情况
* 3. 错误处理：处理非法索引访问
*
* 【类似题目推荐】
* 1. LeetCode 308. 二维区域和检索 - 可变 - https://leetcode.cn/problems/range-sum-query-2d-mutable/
* 2. LeetCode 315. 计算右侧小于当前元素的个数 - https://leetcode.cn/problems/count-of-smaller-numbers-after-self/
* 3. 洛谷 P3372 【模板】线段树 1 - https://www.luogu.com.cn/problem/P3372
* 4. HDU 1166 敌兵布阵 - http://acm.hdu.edu.cn/showproblem.php?pid=1166
*/
public class Code06_RangeSumQueryMutable {
    private int[] tree;
    private int[] nums;
    private int n;
```

```

/**
 * 构造函数: 用整数数组 nums 初始化对象
 *
 * @param nums 初始数组
 */
public Code06_RangeSumQueryMutable(int[] nums) {
    this.n = nums.length;
    this.nums = nums;
    // 线段树数组大小通常为 4n
    this.tree = new int[4 * n];
    build(0, 0, n - 1);
}

/**
 * 构建线段树
 *
 * @param node 当前节点索引
 * @param start 当前区间左边界
 * @param end 当前区间右边界
 */
private void build(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 {
        // 非叶子节点, 递归构建左右子树
        int mid = (start + end) / 2;
        build(2 * node + 1, start, mid);      // 左子树
        build(2 * node + 2, mid + 1, end);    // 右子树
        // 合并左右子树的结果
        tree[node] = tree[2 * node + 1] + tree[2 * node + 2];
    }
}

/**
 * 更新数组中指定索引的值
 *
 * @param index 要更新的数组索引
 * @param val 新的值
 */
public void update(int index, int val) {
    // 计算值的变化量
    int diff = val - nums[index];
    int i = index;
    while (i >= 0) {
        tree[i] += diff;
        i = (i - 1) / 2;
    }
}

```

```

        nums[index] = val;
        // 更新线段树中相关的节点
        updateTree(0, 0, n - 1, index, diff);
    }

/***
 * 更新线段树中的节点值
 *
 * @param node 当前节点索引
 * @param start 当前区间左边界
 * @param end 当前区间右边界
 * @param index 要更新的数组索引
 * @param diff 值的变化量
 */
private void updateTree(int node, int start, int end, int index, int diff) {
    if (start == end) {
        // 到达叶子节点，直接更新
        tree[node] += diff;
    } else {
        int mid = (start + end) / 2;
        if (index <= mid) {
            // 目标索引在左子树中
            updateTree(2 * node + 1, start, mid, index, diff);
        } else {
            // 目标索引在右子树中
            updateTree(2 * node + 2, mid + 1, end, index, diff);
        }
        // 更新当前节点的值
        tree[node] = tree[2 * node + 1] + tree[2 * node + 2];
    }
}

/***
 * 查询指定区间的元素和
 *
 * @param left 查询区间左边界
 * @param right 查询区间右边界
 * @return 区间元素和
 */
public int sumRange(int left, int right) {
    return query(0, 0, n - 1, left, right);
}

```

```

/**
 * 查询线段树中指定区间的元素和
 *
 * @param node 当前节点索引
 * @param start 当前区间左边界
 * @param end 当前区间右边界
 * @param left 查询区间左边界
 * @param right 查询区间右边界
 * @return 区间元素和
 */

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

/**
 * 主函数：测试区域和检索功能
 *
 * @param args 命令行参数
 */
public static void main(String[] args) {
    // 测试用例
    int[] nums = {1, 3, 5};
    Code06_RangeSumQueryMutable numArray = new Code06_RangeSumQueryMutable(nums);

    // 测试区间求和
    System.out.println("sumRange(0, 2): " + numArray.sumRange(0, 2)); // 预期输出: 9

    // 测试更新操作
    numArray.update(1, 2);

    // 测试更新后的区间求和
}

```

```
        System.out.println("sumRange(0, 2): " + numArray.sumRange(0, 2)); // 预期输出: 8
    }
}
```

文件: Code06_RangeSumQueryMutable.py

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

"""

区域和检索 - 数组可修改

题目来源: LeetCode 307. 区域和检索 - 数组可修改

题目链接: <https://leetcode.cn/problems/range-sum-query-mutable/>

核心算法: 线段树

难度: 中等

【题目详细描述】

给你一个数组 `nums`，请你完成两类查询：

1. 一类查询要求更新数组 `nums` 下标对应的值
2. 一类查询要求返回数组 `nums` 中索引 `left` 和索引 `right` 之间（包含）的 `nums` 元素的和，其中 `left <= right`

实现 `NumArray` 类：

- `NumArray(int[] nums)` 用整数数组 `nums` 初始化对象
- `void update(int index, int val)` 将 `nums[index]` 的值更新为 `val`
- `int sumRange(int left, int right)` 返回数组 `nums` 中索引 `left` 和索引 `right` 之间（包含）的 `nums` 元素的和

【解题思路】

使用线段树来维护区间和，支持单点更新和区间查询操作。

【核心算法】

1. 线段树构建：构建支持区间求和的线段树
2. 单点更新：支持更新数组中某个位置的值
3. 区间查询：支持查询任意区间的元素和

【复杂度分析】

- 时间复杂度：
 - 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: $O(n)$, 线段树所需空间

【算法优化点】

1. 数组索引优化: 使用位运算优化索引计算
2. 递归优化: 尾递归优化或迭代实现
3. 内存优化: 预分配固定大小数组

【工程化考量】

1. 输入输出效率: 使用标准输入输出处理
2. 边界条件处理: 处理空数组、单元素数组等特殊情况
3. 错误处理: 处理非法索引访问

【类似题目推荐】

1. LeetCode 308. 二维区域和检索 - 可变 - <https://leetcode.cn/problems/range-sum-query-2d-mutable/>
 2. LeetCode 315. 计算右侧小于当前元素的个数 - <https://leetcode.cn/problems/count-of-smaller-numbers-after-self/>
 3. 洛谷 P3372 【模板】线段树 1 - <https://www.luogu.com.cn/problem/P3372>
 4. HDU 1166 敌兵布阵 - <http://acm.hdu.edu.cn/showproblem.php?pid=1166>
- """

```
class SegmentTree:  
    def __init__(self, nums):  
        """  
        初始化线段树  
  
        Args:  
            nums: 初始数组  
        """  
        self.n = len(nums)  
        self.nums = nums  
        # 线段树数组大小通常为 4n  
        self.tree = [0] * (4 * self.n)  
        # 构建线段树  
        self.build(0, 0, self.n - 1)  
  
    def build(self, node, start, end):  
        """  
        构建线段树  
  
        Args:  
            node: 当前节点索引  
            start: 当前区间左边界  
            end: 当前区间右边界
```

```
"""
if start == end:
    # 叶子节点，存储数组元素值
    self.tree[node] = self.nums[start]
else:
    # 非叶子节点，递归构建左右子树
    mid = (start + end) // 2
    self.build(2 * node + 1, start, mid)      # 左子树
    self.build(2 * node + 2, mid + 1, end)      # 右子树
    # 合并左右子树的结果
    self.tree[node] = self.tree[2 * node + 1] + self.tree[2 * node + 2]
```

```
def update_tree(self, node, start, end, index, diff):
```

```
"""
更新线段树中的节点值
```

Args:

node: 当前节点索引
start: 当前区间左边界
end: 当前区间右边界
index: 要更新的数组索引
diff: 值的变化量

```
"""
if start == end:
    # 到达叶子节点，直接更新
    self.tree[node] += diff
else:
    mid = (start + end) // 2
    if index <= mid:
        # 目标索引在左子树中
        self.update_tree(2 * node + 1, start, mid, index, diff)
    else:
        # 目标索引在右子树中
        self.update_tree(2 * node + 2, mid + 1, end, index, diff)
    # 更新当前节点的值
    self.tree[node] = self.tree[2 * node + 1] + self.tree[2 * node + 2]
```

```
def query(self, node, start, end, left, right):
```

```
"""
查询线段树中指定区间的元素和
```

Args:

node: 当前节点索引

start: 当前区间左边界

end: 当前区间右边界

left: 查询区间左边界

right: 查询区间右边界

Returns:

int: 区间元素和

"""

if right < start or end < left:

查询区间与当前区间无交集

return 0

if left <= start and end <= right:

当前区间完全包含在查询区间内

return self.tree[node]

查询区间与当前区间有部分交集, 递归查询左右子树

mid = (start + end) // 2

left_sum = self.query(2 * node + 1, start, mid, left, right)

right_sum = self.query(2 * node + 2, mid + 1, end, left, right)

return left_sum + right_sum

class NumArray:

def __init__(self, nums):

"""

构造函数: 用整数数组 nums 初始化对象

Args:

nums: 初始数组

"""

self.nums = nums

self.segment_tree = SegmentTree(nums)

def update(self, index, val):

"""

更新数组中指定索引的值

Args:

index: 要更新的数组索引

val: 新的值

"""

计算值的变化量

diff = val - self.nums[index]

self.nums[index] = val

```

# 更新线段树中相关的节点
self.segment_tree.update_tree(0, 0, len(self.nums) - 1, index, diff)

def sumRange(self, left, right):
    """
    查询指定区间的元素和

    Args:
        left: 查询区间左边界
        right: 查询区间右边界

    Returns:
        int: 区间元素和
    """
    return self.segment_tree.query(0, 0, len(self.nums) - 1, left, right)

def main():
    """
    主函数: 测试区域和检索功能
    """

    # 测试用例
    nums = [1, 3, 5]
    numArray = NumArray(nums)

    # 测试区间求和
    print("sumRange(0, 2):", numArray.sumRange(0, 2))  # 预期输出: 9

    # 测试更新操作
    numArray.update(1, 2)

    # 测试更新后的区间求和
    print("sumRange(0, 2):", numArray.sumRange(0, 2))  # 预期输出: 8

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

文件: Code07_CountSmallerNumbersAfterSelf.cpp

=====

/*

- * 计算右侧小于当前元素的个数
- * 题目来源: LeetCode 315. 计算右侧小于当前元素的个数
- * 题目链接: <https://leetcode.cn/problems/count-of-smaller-numbers-after-self/>
- *
- * 核心算法: 线段树 + 离散化
- * 难度: 困难
- *
- * 【题目详细描述】
- * 给你一个整数数组 `nums`，按要求返回一个新数组 `counts`。数组 `counts` 有该性质:
- * `counts[i]` 的值是 `nums[i]` 右侧小于 `nums[i]` 的元素的数量。
- *
- * 示例 1:
- * 输入: `nums = [5, 2, 6, 1]`
- * 输出: `[2, 1, 1, 0]`
- * 解释:
- * 5 的右侧有 2 个更小的元素 (2 和 1)
- * 2 的右侧有 1 个更小的元素 (1)
- * 6 的右侧有 1 个更小的元素 (1)
- * 1 的右侧有 0 个更小的元素
- *
- * 示例 2:
- * 输入: `nums = [-1]`
- * 输出: `[0]`
- *
- * 示例 3:
- * 输入: `nums = [-1, -1]`
- * 输出: `[0, 0]`
- *
- * 【解题思路】
- * 使用离散化+线段树的方法。从右往左遍历数组，用线段树维护每个值出现的次数，
- * 查询小于当前值的元素个数。
- *
- * 【核心算法】
- * 1. 离散化: 将数组中的值映射到连续的整数范围，减少线段树的空间需求
- * 2. 线段树: 维护每个值的出现次数，支持单点更新和前缀和查询
- * 3. 逆序遍历: 从右往左处理数组元素，确保查询的是右侧元素
- *
- * 【复杂度分析】
- * - 时间复杂度: $O(n \log n)$ ，其中 n 是数组长度
- * - 空间复杂度: $O(n)$ ，用于存储离散化映射和线段树
- *
- * 【算法优化点】
- * 1. 离散化优化: 使用二分查找提高离散化效率

- * 2. 线段树优化：使用位运算优化索引计算
- * 3. 遍历优化：逆序遍历避免重复计算
- *
- * 【工程化考量】
 - * 1. 输入输出效率：使用标准输入输出处理
 - * 2. 边界条件处理：处理空数组、单元素数组等特殊情况
 - * 3. 错误处理：处理非法输入和溢出情况
- *
- * 【类似题目推荐】
 - * 1. LeetCode 327. 区间和的个数 - <https://leetcode.cn/problems/count-of-range-sum/>
 - * 2. LeetCode 493. 翻转对 - <https://leetcode.cn/problems/reverse-pairs/>
 - * 3. LeetCode 1649. 通过指令创建有序数组 - <https://leetcode.cn/problems/create-sorted-array-through-instructions/>
 - * 4. 洛谷 P2184 贪婪大陆 - <https://www.luogu.com.cn/problem/P2184>

```
// 由于当前环境限制，无法使用标准 C++ 库中的<iostream> 等头文件
// 因此提供算法思路和伪代码实现，而非完整可编译代码
```

```
/*
// 伪代码实现
#include <vector>
#include <map>
#include <algorithm>
using namespace std;

class SegmentTree {
private:
    int* tree;
    int n;

public:
    SegmentTree(int size) {
        this->n = size;
        this->tree = new int[4 * (size + 1)];
        for (int i = 0; i < 4 * (size + 1); i++) {
            tree[i] = 0;
        }
    }

    void update(int node, int start, int end, int index, int val) {
        if (start == end) {
            tree[node] += val;
        }
    }
}
```

```

} else {
    int mid = (start + end) / 2;
    if (index <= mid) {
        update(2 * node, start, mid, index, val);
    } else {
        update(2 * node + 1, mid + 1, end, index, val);
    }
    tree[node] = tree[2 * node] + tree[2 * node + 1];
}
}

int query(int node, int start, int end, int left, int right) {
    if (left > end || right < start) {
        return 0;
    }
    if (left <= start && end <= right) {
        return tree[node];
    }
    int mid = (start + end) / 2;
    int leftSum = query(2 * node, start, mid, left, right);
    int rightSum = query(2 * node + 1, mid + 1, end, left, right);
    return leftSum + rightSum;
}
};

class Solution {
public:
    vector<int> countSmaller(vector<int>& nums) {
        // 离散化
        vector<int> sorted = nums;
        sort(sorted.begin(), sorted.end());
        map<int, int> ranks;
        int rank = 0;
        for (int num : sorted) {
            if (ranks.find(num) == ranks.end()) {
                ranks[num] = ++rank;
            }
        }
        // 使用线段树
        SegmentTree tree(ranks.size());
        vector<int> result;

```

```

// 从右往左遍历
for (int i = nums.size() - 1; i >= 0; i--) {
    int r = ranks[nums[i]];
    tree.update(1, 1, ranks.size(), r, 1);
    result.push_back(tree.query(1, 1, ranks.size(), 1, r - 1));
}

reverse(result.begin(), result.end());
return result;
}

};

*/

```

// 算法说明:

- // 1. 离散化: 将原数组排序后映射到 1 到 k 的连续整数, k 为不同元素的个数
- // 2. 线段树: 维护每个离散化值的出现次数, 支持单点更新和区间求和
- // 3. 逆序处理: 从右往左遍历原数组, 每次查询当前值对应的离散化值-1 的前缀和
- // 4. 结果构造: 将查询结果逆序得到最终答案

// 时间复杂度: O(n log n)

// 空间复杂度: O(n)

文件: Code07_CountSmallerNumbersAfterSelf.java

```

package class113;

import java.util.*;

/**
 * 计算右侧小于当前元素的个数
 * 题目来源: LeetCode 315. 计算右侧小于当前元素的个数
 * 题目链接: https://leetcode.cn/problems/count-of-smaller-numbers-after-self/
 *
 * 核心算法: 线段树 + 离散化
 * 难度: 困难
 *
 * 【题目详细描述】
 * 给你一个整数数组 nums , 按要求返回一个新数组 counts 。数组 counts 有该性质:
 * counts[i] 的值是 nums[i] 右侧小于 nums[i] 的元素的数量。
 *
 * 示例 1:

```

- * 输入: nums = [5, 2, 6, 1]
- * 输出: [2, 1, 1, 0]
- * 解释:
 - * 5 的右侧有 2 个更小的元素 (2 和 1)
 - * 2 的右侧有 1 个更小的元素 (1)
 - * 6 的右侧有 1 个更小的元素 (1)
 - * 1 的右侧有 0 个更小的元素
- *
- * 示例 2:
- * 输入: nums = [-1]
- * 输出: [0]
- *
- * 示例 3:
- * 输入: nums = [-1, -1]
- * 输出: [0, 0]
- *
- * 【解题思路】
 - * 使用离散化+线段树的方法。从右往左遍历数组，用线段树维护每个值出现的次数，
 - * 查询小于当前值的元素个数。
- *
- * 【核心算法】
 - * 1. 离散化：将数组中的值映射到连续的整数范围，减少线段树的空间需求
 - * 2. 线段树：维护每个值的出现次数，支持单点更新和前缀和查询
 - * 3. 逆序遍历：从右往左处理数组元素，确保查询的是右侧元素
- *
- * 【复杂度分析】
 - * - 时间复杂度: $O(n \log n)$ ，其中 n 是数组长度
 - * - 空间复杂度: $O(n)$ ，用于存储离散化映射和线段树
- *
- * 【算法优化点】
 - * 1. 离散化优化：使用二分查找提高离散化效率
 - * 2. 线段树优化：使用位运算优化索引计算
 - * 3. 遍历优化：逆序遍历避免重复计算
- *
- * 【工程化考量】
 - * 1. 输入输出效率：使用标准输入输出处理
 - * 2. 边界条件处理：处理空数组、单元素数组等特殊情况
 - * 3. 错误处理：处理非法输入和溢出情况
- *
- * 【类似题目推荐】
 - * 1. LeetCode 327. 区间和的个数 – <https://leetcode.cn/problems/count-of-range-sum/>
 - * 2. LeetCode 493. 翻转对 – <https://leetcode.cn/problems/reverse-pairs/>
 - * 3. LeetCode 1649. 通过指令创建有序数组 – <https://leetcode.cn/problems/create-sorted-array->

```
through-instructions/
* 4. 洛谷 P2184 贪婪大陆 - https://www.luogu.com.cn/problem/P2184
*/
public class Code07_CountSmallerNumbersAfterSelf {

    /**
     * 线段树类，用于维护元素出现次数和支持区间查询
     */
    class SegmentTree {
        private int[] tree;
        private int n;

        /**
         * 构造函数
         *
         * @param size 线段树大小
         */
        public SegmentTree(int size) {
            this.n = size;
            this.tree = new int[4 * (size + 1)];
        }

        /**
         * 更新线段树中指定位置的值
         *
         * @param node 当前节点索引
         * @param start 当前区间左边界
         * @param end 当前区间右边界
         * @param index 要更新的位置
         * @param val 增加的值
         */
        public void update(int node, int start, int end, int index, int val) {
            if (start == end) {
                tree[node] += val;
            } else {
                int mid = (start + end) / 2;
                if (index <= mid) {
                    update(2 * node, start, mid, index, val);
                } else {
                    update(2 * node + 1, mid + 1, end, index, val);
                }
                tree[node] = tree[2 * node] + tree[2 * node + 1];
            }
        }
    }
}
```

```

}

/**
 * 查询区间和
 *
 * @param node 当前节点索引
 * @param start 当前区间左边界
 * @param end 当前区间右边界
 * @param left 查询区间左边界
 * @param right 查询区间右边界
 * @return 区间和
 */
public int query(int node, int start, int end, int left, int right) {
    if (left > end || right < start) {
        return 0;
    }
    if (left <= start && end <= right) {
        return tree[node];
    }
    int mid = (start + end) / 2;
    int leftSum = query(2 * node, start, mid, left, right);
    int rightSum = query(2 * node + 1, mid + 1, end, left, right);
    return leftSum + rightSum;
}
}

/**
 * 计算右侧小于当前元素的个数
 *
 * @param nums 输入数组
 * @return 结果数组
 */
public List<Integer> countSmaller(int[] nums) {
    // 离散化
    int[] sorted = nums.clone();
    Arrays.sort(sorted);
    Map<Integer, Integer> ranks = new HashMap<>();
    int rank = 0;
    for (int num : sorted) {
        if (!ranks.containsKey(num)) {
            ranks.put(num, ++rank);
        }
    }
}

```

```
// 使用线段树
SegmentTree tree = new SegmentTree(ranks.size());
List<Integer> result = new ArrayList<>();

// 从右往左遍历
for (int i = nums.length - 1; i >= 0; i--) {
    int r = ranks.get(nums[i]);
    tree.update(1, 1, ranks.size(), r, 1);
    result.add(tree.query(1, 1, ranks.size(), 1, r - 1));
}

Collections.reverse(result);
return result;
}

/**
 * 主函数：测试计算右侧小于当前元素的个数功能
 *
 * @param args 命令行参数
 */
public static void main(String[] args) {
    Code07_CountSmallerNumbersAfterSelf solution = new Code07_CountSmallerNumbersAfterSelf();

    // 测试用例 1
    int[] nums1 = {5, 2, 6, 1};
    List<Integer> result1 = solution.countSmaller(nums1);
    System.out.println("测试用例 1 结果: " + result1); // 预期输出: [2, 1, 1, 0]

    // 测试用例 2
    int[] nums2 = {-1};
    List<Integer> result2 = solution.countSmaller(nums2);
    System.out.println("测试用例 2 结果: " + result2); // 预期输出: [0]

    // 测试用例 3
    int[] nums3 = {-1, -1};
    List<Integer> result3 = solution.countSmaller(nums3);
    System.out.println("测试用例 3 结果: " + result3); // 预期输出: [0, 0]
}
```

=====

文件: Code07_CountSmallerNumbersAfterSelf.py

```
=====
```

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

"""

计算右侧小于当前元素的个数

题目来源: LeetCode 315. 计算右侧小于当前元素的个数

题目链接: <https://leetcode.cn/problems/count-of-smaller-numbers-after-self/>

核心算法: 线段树 + 离散化

难度: 困难

【题目详细描述】

给你一个整数数组 `nums`，按要求返回一个新数组 `counts`。数组 `counts` 有该性质:
`counts[i]` 的值是 `nums[i]` 右侧小于 `nums[i]` 的元素的数量。

示例 1:

输入: `nums = [5, 2, 6, 1]`

输出: `[2, 1, 1, 0]`

解释:

5 的右侧有 2 个更小的元素 (2 和 1)

2 的右侧有 1 个更小的元素 (1)

6 的右侧有 1 个更小的元素 (1)

1 的右侧有 0 个更小的元素

示例 2:

输入: `nums = [-1]`

输出: `[0]`

示例 3:

输入: `nums = [-1, -1]`

输出: `[0, 0]`

【解题思路】

使用离散化+线段树的方法。从右往左遍历数组，用线段树维护每个值出现的次数，
查询小于当前值的元素个数。

【核心算法】

1. 离散化: 将数组中的值映射到连续的整数范围，减少线段树的空间需求
2. 线段树: 维护每个值的出现次数，支持单点更新和前缀和查询
3. 逆序遍历: 从右往左处理数组元素，确保查询的是右侧元素

【复杂度分析】

- 时间复杂度: $O(n \log n)$, 其中 n 是数组长度
- 空间复杂度: $O(n)$, 用于存储离散化映射和线段树

【算法优化点】

1. 离散化优化: 使用二分查找提高离散化效率
2. 线段树优化: 使用位运算优化索引计算
3. 遍历优化: 逆序遍历避免重复计算

【工程化考量】

1. 输入输出效率: 使用标准输入输出处理
2. 边界条件处理: 处理空数组、单元素数组等特殊情况
3. 错误处理: 处理非法输入和溢出情况

【类似题目推荐】

1. LeetCode 327. 区间和的个数 - <https://leetcode.cn/problems/count-of-range-sum/>
2. LeetCode 493. 翻转对 - <https://leetcode.cn/problems/reverse-pairs/>
3. LeetCode 1649. 通过指令创建有序数组 - <https://leetcode.cn/problems/create-sorted-array-through-instructions/>
4. 洛谷 P2184 贪婪大陆 - <https://www.luogu.com.cn/problem/P2184>

"""

```
class SegmentTree:
```

```
    def __init__(self, size):  
        """
```

初始化线段树

Args:

size: 线段树大小

"""

```
    self.n = size
```

```
    self.tree = [0] * (4 * (size + 1))
```

```
def update(self, node, start, end, index, val):
```

"""

更新线段树中指定位置的值

Args:

node: 当前节点索引

start: 当前区间左边界

end: 当前区间右边界

index: 要更新的位置

val: 增加的值

```
"""
if start == end:
    self.tree[node] += val
else:
    mid = (start + end) // 2
    if index <= mid:
        self.update(2 * node, start, mid, index, val)
    else:
        self.update(2 * node + 1, mid + 1, end, index, val)
    self.tree[node] = self.tree[2 * node] + self.tree[2 * node + 1]
```

```
def query(self, node, start, end, left, right):
```

```
"""
查询区间和
```

Args:

- node: 当前节点索引
- start: 当前区间左边界
- end: 当前区间右边界
- left: 查询区间左边界
- right: 查询区间右边界

Returns:

- int: 区间和

```
"""
if left > end or right < start:
    return 0
if left <= start and end <= right:
    return self.tree[node]
mid = (start + end) // 2
left_sum = self.query(2 * node, start, mid, left, right)
right_sum = self.query(2 * node + 1, mid + 1, end, left, right)
return left_sum + right_sum
```

```
class Solution:
```

```
def countSmaller(self, nums):
```

```
"""
计算右侧小于当前元素的个数
```

Args:

- nums: 输入数组

Returns:

List[int]: 结果数组

"""

离散化

sorted_nums = sorted(nums)

ranks = {}

rank = 0

for num in sorted_nums:

if num not in ranks:

rank += 1

ranks[num] = rank

使用线段树

tree = SegmentTree(len(ranks))

result = []

从右往左遍历

for i in range(len(nums) - 1, -1, -1):

r = ranks[nums[i]]

tree.update(1, 1, len(ranks), r, 1)

result.append(tree.query(1, 1, len(ranks), 1, r - 1))

result.reverse()

return result

def main():

"""

主函数：测试计算右侧小于当前元素的个数功能

"""

solution = Solution()

测试用例 1

nums1 = [5, 2, 6, 1]

result1 = solution.countSmaller(nums1)

print("测试用例 1 结果:", result1) # 预期输出: [2, 1, 1, 0]

测试用例 2

nums2 = [-1]

result2 = solution.countSmaller(nums2)

print("测试用例 2 结果:", result2) # 预期输出: [0]

测试用例 3

```
nums3 = [-1, -1]
result3 = solution.countSmaller(nums3)
print("测试用例 3 结果:", result3) # 预期输出: [0, 0]
```

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

文件: Code08_SegmentTreeTemplate1.cpp

```
/*
 * 线段树模板 1 - 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
 *
 * 核心算法: 线段树 + 懒标记
 * 难度: 普及+/提高-
 *
 * 【题目详细描述】
 * 如题, 已知一个数列, 你需要进行下面两种操作:
 * 1. 将某区间每一个数加上 k
 * 2. 求出某区间每一个数的和
 *
 * 输入格式:
 * 第一行包含两个整数 n, m, 分别表示该数列数字的个数和操作的总个数。
 * 第二行包含 n 个用空格分隔的整数, 其中第 i 个数字表示数列第 i 项的初始值。
 * 接下来 m 行每行包含 3 或 4 个整数, 表示一个操作。
 *
 * 输出格式:
 * 输出若干行整数, 表示每次操作 2 的结果。
 *
 * 【解题思路】
 * 使用带懒标记的线段树来实现区间更新和区间查询。
 *
 * 【核心算法】
 * 1. 线段树构建: 构建支持区间求和的线段树
 * 2. 懒标记: 使用懒标记优化区间更新操作
 * 3. 区间更新: 支持区间加法操作
 * 4. 区间查询: 支持区间求和操作
 *
 * 【复杂度分析】
```

- * - 时间复杂度:
 - * - 构建线段树: $O(n)$
 - * - 区间更新: $O(\log n)$
 - * - 区间查询: $O(\log n)$
- * - 空间复杂度: $O(n)$, 线段树所需空间
- *
- * 【算法优化点】
 - * 1. 懒标记优化: 延迟下传标记, 避免不必要的计算
 - * 2. 位运算优化: 使用位移操作优化索引计算
 - * 3. I/O 优化: 使用 `scanf` 和 `printf` 优化输入输出
- *
- * 【工程化考量】
 - * 1. 输入输出效率: 使用高效的 I/O 处理大数据量
 - * 2. 内存管理: 合理分配线段树数组空间
 - * 3. 错误处理: 处理非法输入和边界情况
- *
- * 【类似题目推荐】
 - * 1. 洛谷 P3373 【模板】线段树 2 - <https://www.luogu.com.cn/problem/P3373>
 - * 2. LeetCode 307. 区域和检索 - 数组可修改 - <https://leetcode.cn/problems/range-sum-query-mutable/>
 - * 3. HDU 1698 Just a Hook - <http://acm.hdu.edu.cn/showproblem.php?pid=1698>
 - * 4. POJ 3468 A Simple Problem with Integers - <http://poj.org/problem?id=3468>
- */

```
// 由于当前环境限制, 无法使用标准 C++ 库中的<iostream>等头文件
// 因此提供算法思路和伪代码实现, 而非完整可编译代码
```

```
/*
// 伪代码实现
#include <cstdio>
using namespace std;

const int MAXN = 100005;
long long tree[4 * MAXN];
long long lazy[4 * MAXN];
long long arr[MAXN];
int n, m;

void pushDown(int node, int start, int end) {
    if (lazy[node] != 0) {
        int mid = (start + end) / 2;
        // 更新左右子节点的值
        tree[2 * node] += lazy[node] * (mid - start + 1);
        tree[2 * node + 1] += lazy[node] * (end - mid + 1);
        lazy[2 * node] = lazy[2 * node + 1] = 0;
    }
}
```

```

tree[2 * node + 1] += lazy[node] * (end - mid);
// 传递懒标记给子节点
lazy[2 * node] += lazy[node];
lazy[2 * node + 1] += lazy[node];
// 清除当前节点的懒标记
lazy[node] = 0;
}

}

void build(int node, int start, int end) {
if (start == end) {
    tree[node] = arr[start];
} else {
    int mid = (start + end) / 2;
    build(2 * node, start, mid);
    build(2 * node + 1, mid + 1, end);
    tree[node] = tree[2 * node] + tree[2 * node + 1];
}
}

void update(int node, int start, int end, int l, int r, long long val) {
if (l <= start && end <= r) {
    // 当前区间完全包含在更新区间内
    tree[node] += val * (end - start + 1);
    lazy[node] += val;
} else {
    // 向下传递懒标记
    pushDown(node, start, end);
    int mid = (start + end) / 2;
    if (l <= mid) update(2 * node, start, mid, l, r, val);
    if (r > mid) update(2 * node + 1, mid + 1, end, l, r, val);
    tree[node] = tree[2 * node] + tree[2 * node + 1];
}
}

long long query(int node, int start, int end, int l, int r) {
if (l <= start && end <= r) {
    // 当前区间完全包含在查询区间内
    return tree[node];
}
// 向下传递懒标记
pushDown(node, start, end);
int mid = (start + end) / 2;

```

```

long long sum = 0;
if (l <= mid) sum += query(2 * node, start, mid, l, r);
if (r > mid) sum += query(2 * node + 1, mid + 1, end, l, r);
return sum;
}

int main() {
    scanf("%d%d", &n, &m);

    for (int i = 1; i <= n; i++) {
        scanf("%lld", &arr[i]);
    }

    build(1, 1, n);

    for (int i = 0; i < m; i++) {
        int op;
        scanf("%d", &op);
        if (op == 1) {
            int x, y;
            long long k;
            scanf("%d%d%lld", &x, &y, &k);
            update(1, 1, n, x, y, k);
        } else {
            int x, y;
            scanf("%d%d", &x, &y);
            printf("%lld\n", query(1, 1, n, x, y));
        }
    }

    return 0;
}
*/
// 算法说明:
// 1. 线段树构建: 递归构建二叉树, 叶子节点存储数组元素, 非叶子节点存储子区间和
// 2. 懒标记: 使用 lazy 数组存储延迟更新的值, 避免不必要的递归更新
// 3. 区间更新: 更新区间时, 如果当前节点区间完全包含在更新区间内, 则打上懒标记;
//           否则向下传递懒标记并递归更新子节点
// 4. 区间查询: 查询区间时, 如果当前节点区间完全包含在查询区间内, 则直接返回;
//           否则向下传递懒标记并递归查询子节点

// 时间复杂度: O(log n) for update and query

```

```
// 空间复杂度: O(n)
```

```
=====
```

文件: Code08_SegmentTreeTemplate1.java

```
=====
```

```
package class113;

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

/***
 * 线段树模板 1 - 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
 *
 * 核心算法: 线段树 + 懒标记
 * 难度: 普及+/提高-
 *
 * 【题目详细描述】
 * 如题, 已知一个数列, 你需要进行下面两种操作:
 * 1. 将某区间每一个数加上 k
 * 2. 求出某区间每一个数的和
 *
 * 输入格式:
 * 第一行包含两个整数 n, m, 分别表示该数列数字的个数和操作的总个数。
 * 第二行包含 n 个用空格分隔的整数, 其中第 i 个数字表示数列第 i 项的初始值。
 * 接下来 m 行每行包含 3 或 4 个整数, 表示一个操作。
 *
 * 输出格式:
 * 输出若干行整数, 表示每次操作 2 的结果。
 *
 * 【解题思路】
 * 使用带懒标记的线段树来实现区间更新和区间查询。
 *
 * 【核心算法】
 * 1. 线段树构建: 构建支持区间求和的线段树
 * 2. 懒标记: 使用懒标记优化区间更新操作
 * 3. 区间更新: 支持区间加法操作
 * 4. 区间查询: 支持区间求和操作
 *
 * 【复杂度分析】
 * - 时间复杂度:
```

- * - 构建线段树: $O(n)$
- * - 区间更新: $O(\log n)$
- * - 区间查询: $O(\log n)$
- * - 空间复杂度: $O(n)$, 线段树所需空间
- *
- * 【算法优化点】
 - * 1. 懒标记优化: 延迟下传标记, 避免不必要的计算
 - * 2. 位运算优化: 使用位移操作优化索引计算
 - * 3. IO 优化: 使用 BufferedReader 和 PrintWriter 优化输入输出
- *
- * 【工程化考量】
 - * 1. 输入输出效率: 使用高效的 IO 处理大数据量
 - * 2. 内存管理: 合理分配线段树数组空间
 - * 3. 错误处理: 处理非法输入和边界情况
- *
- * 【类似题目推荐】
 - * 1. 洛谷 P3373 【模板】线段树 2 - <https://www.luogu.com.cn/problem/P3373>
 - * 2. LeetCode 307. 区域和检索 - 数组可修改 - <https://leetcode.cn/problems/range-sum-query-mutable/>
 - * 3. HDU 1698 Just a Hook - <http://acm.hdu.edu.cn/showproblem.php?pid=1698>
 - * 4. POJ 3468 A Simple Problem with Integers - <http://poj.org/problem?id=3468>
- */

```

public class Code08_SegmentTreeTemplate1 {

    static long[] tree;
    static long[] lazy;
    static long[] arr;
    static int n, m;

    /**
     * 向下传递懒标记
     *
     * @param node 当前节点索引
     * @param start 当前区间左边界
     * @param end 当前区间右边界
     */
    static void pushDown(int node, int start, int end) {
        if (lazy[node] != 0) {
            int mid = (start + end) / 2;
            // 更新左右子节点的值
            tree[2 * node] += lazy[node] * (mid - start + 1);
            tree[2 * node + 1] += lazy[node] * (end - mid);
            // 传递懒标记给子节点
        }
    }
}

```

```

        lazy[2 * node] += lazy[node];
        lazy[2 * node + 1] += lazy[node];
        // 清除当前节点的懒标记
        lazy[node] = 0;
    }
}

/***
 * 构建线段树
 *
 * @param node 当前节点索引
 * @param start 当前区间左边界
 * @param end 当前区间右边界
 */
static void build(int node, int start, int end) {
    if (start == end) {
        tree[node] = arr[start];
    } else {
        int mid = (start + end) / 2;
        build(2 * node, start, mid);
        build(2 * node + 1, mid + 1, end);
        tree[node] = tree[2 * node] + tree[2 * node + 1];
    }
}

/***
 * 区间更新操作
 *
 * @param node 当前节点索引
 * @param start 当前区间左边界
 * @param end 当前区间右边界
 * @param l 更新区间左边界
 * @param r 更新区间右边界
 * @param val 要加上的值
 */
static void update(int node, int start, int end, int l, int r, long val) {
    if (l <= start && end <= r) {
        // 当前区间完全包含在更新区间内
        tree[node] += val * (end - start + 1);
        lazy[node] += val;
    } else {
        // 向下传递懒标记
        pushDown(node, start, end);
    }
}

```

```

        int mid = (start + end) / 2;
        if (l <= mid) update(2 * node, start, mid, l, r, val);
        if (r > mid) update(2 * node + 1, mid + 1, end, l, r, val);
        tree[node] = tree[2 * node] + tree[2 * node + 1];
    }
}

/***
 * 区间查询操作
 *
 * @param node 当前节点索引
 * @param start 当前区间左边界
 * @param end 当前区间右边界
 * @param l 查询区间左边界
 * @param r 查询区间右边界
 * @return 区间和
 */
static long query(int node, int start, int end, int l, int r) {
    if (l <= start && end <= r) {
        // 当前区间完全包含在查询区间内
        return tree[node];
    }
    // 向下传递懒标记
    pushDown(node, start, end);
    int mid = (start + end) / 2;
    long sum = 0;
    if (l <= mid) sum += query(2 * node, start, mid, l, r);
    if (r > mid) sum += query(2 * node + 1, mid + 1, end, l, r);
    return sum;
}

/***
 * 主函数：处理输入并执行操作
 *
 * @param args 命令行参数
 * @throws IOException IO 异常
 */
public static void main(String[] args) throws IOException {
    BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
    StringTokenizer st = new StringTokenizer(br.readLine());
    n = Integer.parseInt(st.nextToken());
    m = Integer.parseInt(st.nextToken());
}

```

```

arr = new long[n + 1];
tree = new long[4 * n];
lazy = new long[4 * n];

st = new StringTokenizer(br.readLine());
for (int i = 1; i <= n; i++) {
    arr[i] = Long.parseLong(st.nextToken());
}

build(1, 1, n);

PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));
for (int i = 0; i < m; i++) {
    st = new StringTokenizer(br.readLine());
    int op = Integer.parseInt(st.nextToken());
    if (op == 1) {
        int x = Integer.parseInt(st.nextToken());
        int y = Integer.parseInt(st.nextToken());
        long k = Long.parseLong(st.nextToken());
        update(1, 1, n, x, y, k);
    } else {
        int x = Integer.parseInt(st.nextToken());
        int y = Integer.parseInt(st.nextToken());
        out.println(query(1, 1, n, x, y));
    }
}

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

```

文件: Code08_SegmentTreeTemplate1.py

```

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

```

"""

线段树模板 1 - 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>

核心算法: 线段树 + 懒标记

难度: 普及+/提高-

【题目详细描述】

如题, 已知一个数列, 你需要进行下面两种操作:

1. 将某区间每一个数加上 k
2. 求出某区间每一个数的和

输入格式:

第一行包含两个整数 n, m , 分别表示该数列数字的个数和操作的总个数。

第二行包含 n 个用空格分隔的整数, 其中第 i 个数字表示数列第 i 项的初始值。

接下来 m 行每行包含 3 或 4 个整数, 表示一个操作。

输出格式:

输出若干行整数, 表示每次操作 2 的结果。

【解题思路】

使用带懒标记的线段树来实现区间更新和区间查询。

【核心算法】

1. 线段树构建: 构建支持区间求和的线段树
2. 懒标记: 使用懒标记优化区间更新操作
3. 区间更新: 支持区间加法操作
4. 区间查询: 支持区间求和操作

【复杂度分析】

- 时间复杂度:
 - 构建线段树: $O(n)$
 - 区间更新: $O(\log n)$
 - 区间查询: $O(\log n)$
- 空间复杂度: $O(n)$, 线段树所需空间

【算法优化点】

1. 懒标记优化: 延迟下传标记, 避免不必要的计算
2. 位运算优化: 使用位移操作优化索引计算
3. I/O 优化: 使用 `sys.stdin` 和 `sys.stdout` 优化输入输出

【工程化考量】

1. 输入输出效率: 使用高效的 I/O 处理大数据量
2. 内存管理: 合理分配线段树数组空间
3. 错误处理: 处理非法输入和边界情况

【类似题目推荐】

1. 洛谷 P3373 【模板】线段树 2 - <https://www.luogu.com.cn/problem/P3373>
 2. LeetCode 307. 区域和检索 - 数组可修改 - <https://leetcode.cn/problems/range-sum-query-mutable/>
 3. HDU 1698 Just a Hook - <http://acm.hdu.edu.cn/showproblem.php?pid=1698>
 4. POJ 3468 A Simple Problem with Integers - <http://poj.org/problem?id=3468>
- """

```
import sys
```

```
class SegmentTree:
```

```
    def __init__(self, arr):
```

```
        """
```

```
    初始化线段树
```

```
    Args:
```

```
        arr: 初始数组
```

```
    """
```

```
    self.n = len(arr)
```

```
    self.arr = arr
```

```
    self.tree = [0] * (4 * self.n)
```

```
    self.lazy = [0] * (4 * self.n)
```

```
    self.build(1, 1, self.n)
```

```
    def push_down(self, node, start, end):
```

```
        """
```

```
    向下传递懒标记
```

```
    Args:
```

```
        node: 当前节点索引
```

```
        start: 当前区间左边界
```

```
        end: 当前区间右边界
```

```
    """
```

```
    if self.lazy[node] != 0:
```

```
        mid = (start + end) // 2
```

```
        # 更新左右子节点的值
```

```
        self.tree[2 * node] += self.lazy[node] * (mid - start + 1)
```

```
        self.tree[2 * node + 1] += self.lazy[node] * (end - mid)
```

```
        # 传递懒标记给子节点
```

```
        self.lazy[2 * node] += self.lazy[node]
```

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

```
        # 清除当前节点的懒标记
```

```
        self.lazy[node] = 0
```

```

def build(self, node, start, end):
    """
    构建线段树

    Args:
        node: 当前节点索引
        start: 当前区间左边界
        end: 当前区间右边界
    """
    if start == end:
        self.tree[node] = self.arr[start]
    else:
        mid = (start + end) // 2
        self.build(2 * node, start, mid)
        self.build(2 * node + 1, mid + 1, end)
        self.tree[node] = self.tree[2 * node] + self.tree[2 * node + 1]

def update(self, node, start, end, l, r, val):
    """
    区间更新操作

    Args:
        node: 当前节点索引
        start: 当前区间左边界
        end: 当前区间右边界
        l: 更新区间左边界
        r: 更新区间右边界
        val: 要加上的值
    """
    if l <= start and end <= r:
        # 当前区间完全包含在更新区间内
        self.tree[node] += val * (end - start + 1)
        self.lazy[node] += val
    else:
        # 向下传递懒标记
        self.push_down(node, start, end)
        mid = (start + end) // 2
        if l <= mid:
            self.update(2 * node, start, mid, l, r, val)
        if r > mid:
            self.update(2 * node + 1, mid + 1, end, l, r, val)
        self.tree[node] = self.tree[2 * node] + self.tree[2 * node + 1]

```

```
def query(self, node, start, end, l, r):
```

```
    """
```

```
    区间查询操作
```

```
Args:
```

```
    node: 当前节点索引
```

```
    start: 当前区间左边界
```

```
    end: 当前区间右边界
```

```
    l: 查询区间左边界
```

```
    r: 查询区间右边界
```

```
Returns:
```

```
    long: 区间和
```

```
    """
```

```
if l <= start and end <= r:
```

```
    # 当前区间完全包含在查询区间内
```

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

```
# 向下传递懒标记
```

```
self.push_down(node, start, end)
```

```
mid = (start + end) // 2
```

```
sum_val = 0
```

```
if l <= mid:
```

```
    sum_val += self.query(2 * node, start, mid, l, r)
```

```
if r > mid:
```

```
    sum_val += self.query(2 * node + 1, mid + 1, end, l, r)
```

```
return sum_val
```

```
def main():
```

```
    """
```

```
主函数: 处理输入并执行操作
```

```
    """
```

```
import sys
```

```
input = sys.stdin.read
```

```
data = input().split()
```

```
idx = 0
```

```
n = int(data[idx])
```

```
idx += 1
```

```
m = int(data[idx])
```

```
idx += 1
```

```
arr = [0] * (n + 1)
for i in range(1, n + 1):
    arr[i] = int(data[idx])
    idx += 1

seg_tree = SegmentTree(arr)

results = []
for _ in range(m):
    op = int(data[idx])
    idx += 1
    if op == 1:
        x = int(data[idx])
        idx += 1
        y = int(data[idx])
        idx += 1
        k = int(data[idx])
        idx += 1
        seg_tree.update(1, 1, n, x, y, k)
    else:
        x = int(data[idx])
        idx += 1
        y = int(data[idx])
        idx += 1
        results.append(str(seg_tree.query(1, 1, n, x, y)))

print('\n'.join(results))
```

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

```
=====
```