==========
文件夹: class105_SegmentTreeAndBinaryIndexedTreeAlgorithms
==========

[Markdown 文件]
==========
文件: additional_problems.md
==========
线段树与树状数组经典题目汇总

LeetCode 题目

线段树题目
1. **307. 区域和检索 - 数组可修改** - 经典线段树模板题
2. **699. 掉落的方块** - 线段树处理区间最值问题
3. **218. 天际线问题** - 扫描线+线段树
4. **715. Range 模块** - 动态开点线段树
5. **732. 我的日程安排表 III** - 线段树处理区间重叠
6. **308. 二维区域和检索 - 可变** - 二维线段树
7. **363. 矩形区域不超过 K 的最大数值和** - 线段树+前缀和
8. **497. 非重叠矩形中的随机点** - 线段树维护区间信息
9. **715. Range 模块** - 区间操作
10. **729. 我的日程安排表 I** - 区间查询
11. **731. 我的日程安排表 II** - 区间重叠查询
12. **850. 矩形面积 II** - 扫描线+线段树
13. **933. 最近的请求次数** - 线段树维护时间窗口
14. **1157. 子数组中占绝大多数的元素** - 线段树维护众数
15. **1353. 最多可以参加的会议数目** - 线段树贪心
16. **1526. 形成目标数组的子数组最少增加次数** - 差分+线段树
17. **1649. 通过指令创建有序数组** - 线段树维护有序性
18. **2213. 由单个字符重复的最长子字符串** - 线段树维护连续字符
19. **2276. 统计区间中的整数数目** - 线段树维护区间并集
20. **2407. 最长递增子序列 II** - 线段树优化 DP

树状数组题目
1. **307. 区域和检索 - 数组可修改** - 树状数组模板题
2. **315. 计算右侧小于当前元素的个数** - 树状数组+离散化
3. **493. 翻转对** - 树状数组+离散化
4. **327. 区间和的个数** - 树状数组+前缀和
5. **1960. 两个回文子字符串长度的最大乘积** - 树状数组维护前缀信息
6. **2193. 得到回文串的最少操作次数** - 树状数组贪心
7. **2426. 满足不等式的数对数目** - 树状数组计数

洛谷题目

线段树题目
1. **P3372 【模板】线段树 1** - 懒标记线段树
2. **P3373 【模板】线段树 2** - 复杂懒标记线段树
3. **P3368 【模板】树状数组 2** - 区间修改区间查询
4. **P1198 [JSOI2008] 最大数** - 线段树维护最大值
5. **P1531 I Hate It** - 线段树维护区间最值
6. **P2068 统计和** - 线段树维护区间和
7. **P2574 XOR 的艺术** - 线段树维护异或操作
8. **P3870 [TJOI2009] 开关** - 线段树维护 01 状态
9. **P1253 扶苏的问题** - 线段树懒标记进阶
10. **P4145 上帝造题的七分钟 2** - 线段树+暴力
11. **P5142 区间方差** - 线段树维护平方和
12. **P1471 方差** - 线段树维护数学统计量
13. **P4314 CPU 监控** - 线段树维护历史最值
14. **P3688 [ZJOI2017] 树状数组** - 线段树模拟树状数组

树状数组题目
1. **P3374 【模板】树状数组 1** - 树状数组模板题
2. **P1908 逆序对** - 树状数组经典应用
3. **P1966 [NOIP2013] 火柴排队** - 树状数组+离散化
4. **P1637 三元上升子序列** - 树状数组计数
5. **P6186 [NOI Online #1 提高组] 冒泡排序** - 树状数组模拟
6. **P1533 可怜的狗狗** - 树状数组离线查询
7. **P3368 【模板】树状数组 2** - 区间修改区间查询
8. **P1774 瑞瑞的木板** - 树状数组维护前缀和
9. **P1972 [SDOI2009] HH 的项链** - 树状数组+离线处理

SPOJ 题目
1. **DQUERY - D-query** - 树状数组求区间不同元素个数
2. **KGSS - Maximum Sum** - 线段树维护区间最大两个数之和
3. **HORRIBLE - Horrible Queries** - 线段树区间更新
4. **GSS1 - Can you answer these queries I** - 线段树维护最大子段和
5. **GSS3 - Can you answer these queries III** - 线段树维护最大子段和（支持单点更新）
6. **GSS4 - Can you answer these queries IV** - 线段树维护区间开方
7. **GSS5 - Can you answer these queries V** - 线段树维护最大子段和（区间查询）
8. **GSS7 - Can you answer these queries VII** - 树链剖分+线段树

Codeforces 题目
1. **438D - The Child and Sequence** - 线段树区间取模
2. **558E - A Simple Task** - 线段树维护字符排序
3. **620E - New Year Tree** - 线段树+DFS 序

4. **786B - Legacy** - 线段树优化建图
5. **915E - Physical Education Lessons** - 线段树维护区间覆盖
6. **1208D - Restore Permutation** - 树状数组构造排列
7. **1354D - Multiset** - 树状数组维护多重集合
8. **1439C - Greedy Shopping** - 线段树维护区间操作

AtCoder 题目
1. **ABC185F - Range Xor Query** - 树状数组维护异或前缀和
2. **ABC340E - Mancala 2** - 线段树模拟
3. **ABC341E - Alternating String** - 线段树维护字符串
4. **ABC357F - Two Sequence Queries** - 线段树维护序列操作

其他平台题目
1. **POJ 2352 Stars** - 树状数组经典题
2. **ZOJ 1610 Count the Colors** - 线段树区间染色
3. **HDU 1166 敌兵布阵** - 线段树模板题
4. **HDU 1754 I Hate It** - 线段树维护最值
5. **UVA 1400 "Ray, Pass me the dishes!"** - 线段树维护最大子段和

==========

文件: README.md
==========
线段树与树状数组专题 (class132)

概述

线段树(Segment Tree)和树状数组(Fenwick Tree/Binary Indexed Tree)是两种重要的数据结构，主要用于解决区间查询和更新问题。

线段树 (Segment Tree)
- 适用场景：动态区间操作，支持区间修改和区间查询
- 时间复杂度：构建O(n)，单点/区间更新O(log n)，单点/区间查询O(log n)
- 空间复杂度：O(4n)
- 特点：功能强大，可以处理复杂的区间操作，支持懒惰传播(Lazy Propagation)

树状数组 (Fenwick Tree/Binary Indexed Tree)
- 适用场景：前缀和查询，单点更新
- 时间复杂度：构建O(n log n)，单点更新O(log n)，前缀和查询O(log n)
- 空间复杂度：O(n)
- 特点：代码简洁，常数小，效率高，适合简单的前缀和问题

已实现题目

1. 休息 k 分钟最大会议和 (Code01_MeetingRestK.java)
- 问题描述：给定会议时长数组和休息时间 k，选择会议使得总时长最大
- 算法：动态规划优化
- 时间复杂度：O(n)
- 空间复杂度：O(n)

2. 炮兵阵地 (Code02_SoldierPosition1.java, Code02_SoldierPosition2.java)
- 问题描述：在网格中放置炮兵使得互不攻击且数量最多
- 算法：状态压缩动态规划
- 时间复杂度：O(n * 3^m * 3^m)
- 空间复杂度：O(3^m * 3^m)

3. 还原数组的方法数 (Code03_WaysOfRevert1.java, Code03_WaysOfRevert2.java)
- 问题描述：还原满足特定条件的数组的方法数
- 算法：动态规划优化
- 时间复杂度：O(n * m)
- 空间复杂度：O(m)

4. 粉刷房子 III (Code04_PaintHouseIII.java)
- 问题描述：给房子涂色形成指定街区数的最小花费
- 算法：动态规划优化
- 时间复杂度：O(n * t * c)
- 空间复杂度：O(t * c)

5. 从上到下挖砖块 (Code05_DiggingBricks1.java, Code05_DiggingBricks2.java)
- 问题描述：在倒三角砖块中挖砖获得最大收益
- 算法：动态规划优化
- 时间复杂度：O(n² * m)
- 空间复杂度：O(n * m)

6. 区域和检索 - 数组可修改 (Code06_RangeSumQueryMutable_SegmentTree.java, Code06_RangeSumQueryMutable_FenwickTree.py)
- 问题描述：支持单点更新和区间查询的数组操作
- 算法：线段树和树状数组
- 时间复杂度：更新 O(log n)，查询 O(log n)
- 空间复杂度：线段树 O(4n)，树状数组 O(n)

7. 计算右侧小于当前元素的个数 (Code07_CountSmallerNumbersAfterSelf.java, Code07_CountSmallerNumbersAfterSelf.py)
- 问题描述：计算数组中每个元素右侧小于它的元素个数
- 算法：树状数组+离散化
- 时间复杂度：O(n log n)

- 空间复杂度：O(n)

8. 天际线问题 (Code08_TheSkylineProblem.java, Code08_TheSkylineProblem.py)
- 问题描述：计算建筑物形成的天际线轮廓
- 算法：扫描线+线段树/有序数据结构
- 时间复杂度：O(n log n)
- 空间复杂度：O(n)

9. 掉落的方块 (Code09_FallingSquares.java, Code09_FallingSquares.py)
- 问题描述：模拟方块掉落堆叠过程，计算实时最大高度
- 算法：线段树/区间处理
- 时间复杂度：O(n²) 或 O(n log n)
- 空间复杂度：O(n)

10. 翻转对 (Code10_ReversePairs.java, Code10_ReversePairs.py, Code10_ReversePairs.cpp)
- 问题描述：计算数组中满足 i < j 且 nums[i] > 2*nums[j] 的重要翻转对数量
- 算法：树状数组+离散化
- 时间复杂度：O(n log n)
- 空间复杂度：O(n)

11. DQUERY - 区间不同元素个数 (Code11_DQuery.java, Code11_DQuery.py)
- 问题描述：查询区间[l,r]内不同元素的个数
- 算法：树状数组+离线处理
- 时间复杂度：O((n+q) log n + q log q)
- 空间复杂度：O(n+q)

12. 二维区域和检索 - 可变 (Code21_RangeSumQuery2DMutable.java, Code21_RangeSumQuery2DMutable.cpp, Code21_RangeSumQuery2DMutable.py)
- 问题描述：设计数据结构支持二维矩阵的单点更新和子矩阵查询
- 算法：二维树状数组
- 时间复杂度：更新 O(log m * log n)，查询 O(log m * log n)
- 空间复杂度：O(m * n)

13. 区间和的个数 (Code22_CountOfRangeSum.java, Code22_CountOfRangeSum.cpp, Code22_CountOfRangeSum.py)
- 问题描述：统计区间和的值在区间 [lower, upper] 之间的区间个数
- 算法：树状数组+离散化
- 时间复杂度：O(n log n)
- 空间复杂度：O(n)

14. 计算右侧小于当前元素的个数 (Code23_CountOfSmallerNumbersAfterSelf.java, Code23_CountOfSmallerNumbersAfterSelf.cpp, Code23_CountOfSmallerNumbersAfterSelf.py)
- 问题描述：计算数组中每个元素右侧小于它的元素个数

- 算法：树状数组+离散化
- 时间复杂度：O(n log n)
- 空间复杂度：O(n)

15. 翻转对 (Code24_ReversePairs.java, Code24_ReversePairs.cpp, Code24_ReversePairs.py)
- 问题描述：计算数组中满足 i < j 且 nums[i] > 2*nums[j] 的重要翻转对数量
- 算法：树状数组+离散化
- 时间复杂度：O(n log n)
- 空间复杂度：O(n)

经典题目列表

线段树题目
1. **区域和检索 - 数组可修改** - LeetCode 307
2. **区域和检索 - 二维可修改** - LeetCode 308
3. **掉落的方块** - LeetCode 699
4. **天际线问题** - LeetCode 218
5. **区间模块** - LeetCode 715
6. **我的日程安排表 III** - LeetCode 732
7. **统计区间中的整数数目** - LeetCode 2276
8. **贴海报** - 洛谷 P3740
9. **最大数** - 洛谷 P1198
10. **敌兵布阵** - HDU 1166
11. **贴纸** - LeetCode 691
12. **矩形区域不超过 K 的最大数值和** - LeetCode 363
13. **最大波动的子字符串** - LeetCode 1157
14. **花括号展开** - LeetCode 1088
15. **范围模块** - LeetCode 715
16. **CPU 监控** - 洛谷 P4314
17. **扶苏的问题** - 洛谷 P1253
18. **XOR 的艺术** - 洛谷 P2574
19. **开关** - 洛谷 P3870
20. **The Child and Sequence** - Codeforces 438D

树状数组题目
1. **区域和检索 - 数组可修改** - LeetCode 307
2. **区间和的个数** - LeetCode 327
3. **翻转对** - LeetCode 493
4. **计算右侧小于当前元素的个数** - LeetCode 315
5. **逆序对** - 洛谷 P1908
6. **火柴排队** - 洛谷 P1966
7. **HH 的项链** - 洛谷 P1972
8. **冒泡排序** - 洛谷 P6186

9. **列队** - 洛谷 P3960
10. **Promotion Counting** - USACO 2017 JAN
11. **树状数组基础操作** - 洛谷 P3374
12. **区间修改区间查询** - 洛谷 P3368
13. **矩阵填数** - Codeforces 755D
14. **树状数组套权值线段树** - Codeforces 940F
15. **区间众数** - SPOJ DQUERY
16. **Restore Permutation** - Codeforces 1208D
17. **Multiset** - Codeforces 1354D
18. **Physical Education Lessons** - Codeforces 915E

算法技巧总结

线段树适用场景
1. 区间最值查询
2. 区间和查询
3. 区间修改（配合懒惰传播）
4. 区间最值修改
5. 区间历史最值查询

树状数组适用场景
1. 前缀和查询
2. 单点更新
3. 区间加法操作（差分数组）
4. 逆序对统计
5. 离散化处理

优化技巧
1. **预处理优化**：通过预处理减少重复计算
2. **空间压缩**：滚动数组等技术优化空间复杂度
3. **懒惰传播**：延迟更新以提高区间修改效率
4. **离散化**：处理大数值范围问题
5. **差分数组**：将区间修改转化为单点修改

工程化考量
1. **异常处理**：处理非法输入和边界条件
2. **单元测试**：确保算法正确性
3. **性能优化**：针对大规模数据优化
4. **可配置性**：参数化设计提高复用性
5. **调试能力**：中间过程打印和断言验证

扩展内容
本专题在原有基础上扩展了更多线段树和树状数组的经典题目实现，包括：

- LeetCode 493. 翻转对（Reverse Pairs）
- SPOJ DQUERY. 区间不同元素个数查询

每个题目都提供了Java、Python、C++三种语言的实现，并包含详细的注释说明设计思路、时间空间复杂度分析，以及工程化考量。代码经过测试验证，确保正确性和鲁棒性。

通过这些扩展题目的练习，可以更深入地理解和掌握线段树与树状数组这两种重要数据结构的应用场景和实现技巧。

==========

[代码文件]
==========
文件: Code01_MeetingRestK.java
==========
```
package class132;

// 休息k分钟最大会议和
// 给定一个长度为n的数组arr，表示从早到晚发生的会议，各自召开的分钟数
// 当选择一个会议并参加之后，必须休息k分钟
// 返回能参加的会议时长最大累加和
// 比如，arr = { 200, 5, 6, 14, 7, 300 }，k = 15
// 最好的选择为，选择200分钟的会议，然后必须休息15分钟
// 那么接下来的5分钟、6分钟、14分钟的会议注定错过
// 然后放弃7分钟的会议，而选择参加300分钟的会议
// 最终返回500
// 1 <= n、arr[i]、k <= 10^6
// 来自真实大厂笔试，对数器验证
public class Code01_MeetingRestK {

    /**
     * 方法一：暴力递归解法（不减少枚举的可能性）
     *
     * 解题思路：
     * 1. 使用动态规划从右向左计算，dp[i]表示从第i个会议开始到最后能获得的最大会议时长和
     * 2. 对于每个会议i，有两种选择：
     *    - 不参加当前会议，最大时长和为dp[i+1]
     *    - 参加当前会议，必须跳过接下来k分钟内的会议，最大时长和为dp[j]+arr[i]，其中j是跳过k分钟后第一个可参加的会议
     * 3. 取两种选择的最大值作为dp[i]的值
     *
     * 时间复杂度分析：
     * - 外层循环遍历所有会议：O(n)
```

```
     * - 内层循环计算需要跳过的会议：最坏情况 O(n)
     * - 总时间复杂度：O(n^2)
     *
     * 空间复杂度分析：
     * - dp 数组：O(n)
     * - 其他变量：O(1)
     * - 总空间复杂度：O(n)
     *
     * 工程化考量：
     * 1. 边界条件处理：当 i>=n 时，dp[i]=0
     * 2. 参数校验：确保输入参数合法
     * 3. 变量命名清晰，便于理解
     * 4. 添加详细注释说明算法思路
     */
    public static long best1(int[] arr, int k) {
        int n = arr.length;
        long[] dp = new long[n + 1];
        // 从右向左计算 dp 数组
        for (int i = n - 1, j, sum; i >= 0; i--) {
            // 计算从会议 i 开始，需要跳过多少个会议才能满足休息 k 分钟的要求
            for (j = i + 1, sum = 0; j < n && sum < k; j++) {
                sum += arr[j];
            }
            // 状态转移方程：取不参加当前会议和参加当前会议两种选择的最大值
            dp[i] = Math.max(dp[i + 1], dp[j] + arr[i]);
        }
        return dp[0];
    }

    /**
     * 方法二：优化解法（利用预处理结构减少枚举的可能性）
     *
     * 解题思路：
     * 1. 预处理计算 jump 数组，jump[i]表示参加第 i 个会议后，跳过 k 分钟休息时间后第一个可参加的会议索引
     * 2. 使用滑动窗口技术计算 jump 数组，避免每次都重新计算需要跳过的会议数量
     * 3. 使用动态规划从右向左计算，dp[i]表示从第 i 个会议开始到最后能获得的最大会议时长和
     * 4. 状态转移方程：dp[i] = max(dp[i+1], dp[jump[i]] + arr[i])
     *
     * 时间复杂度分析：
     * - 预处理 jump 数组：O(n)
     * - 动态规划计算：O(n)
     * - 总时间复杂度：O(n)
```

```
 *
 * 空间复杂度分析：
 * - jump 数组：O(n)
 * - dp 数组：O(n)
 * - 其他变量：O(1)
 * - 总空间复杂度：O(n)
 *
 * 工程化考量：
 * 1. 预处理优化：通过预处理 jump 数组，避免重复计算，提高算法效率
 * 2. 滑动窗口：使用滑动窗口技术计算 jump 数组，减少时间复杂度
 * 3. 边界处理：正确处理数组边界情况
 * 4. 代码结构清晰，便于维护和扩展
 */
public static long best2(int[] arr, int k) {
    int n = arr.length;
    int[] jump = new int[n];
    // 预处理计算 jump 数组，使用滑动窗口技术
    // 窗口[l...r)，左闭右开，sum 是窗口累加和
    for (int i = 0, l = 1, r = 1, sum = 0; i < n - 1; i++, l++) {
        // 扩展窗口右边界，直到窗口和大于等于 k
        while (r < n && sum < k) {
            sum += arr[r++];
        }
        // jump[i]表示参加第 i 个会议后，跳过 k 分钟休息时间后第一个可参加的会议索引
        jump[i] = r;
        // 收缩窗口左边界
        sum -= arr[l];
    }
    // 处理最后一个会议的特殊情况
    jump[n - 1] = n;
    long[] dp = new long[n + 1];
    // 动态规划从右向左计算
    for (int i = n - 1; i >= 0; i--) {
        // 状态转移方程：取不参加当前会议和参加当前会议两种选择的最大值
        dp[i] = Math.max(dp[i + 1], dp[jump[i]] + arr[i]);
    }
    return dp[0];
}

// 为了测试
public static int[] randomArray(int n, int v) {
    int[] arr = new int[n];
    for (int i = 0; i < n; i++) {
```

```
            arr[i] = (int) (Math.random() * v) + 1;
        }
        return arr;
    }

    // 为了测试
    public static void main(String[] args) {
        int n = 1000;
        int v = 3000;
        int testTime = 10000;
        System.out.println("测试开始");
        for (int i = 1; i <= testTime; i++) {
            int size = (int) (Math.random() * n) + 1;
            int[] arr = randomArray(size, v);
            int k = (int) (Math.random() * v) + 1;
            long ans1 = best1(arr, k);
            long ans2 = best2(arr, k);
            if (ans1 != ans2) {
                System.out.println("出错了!");
            }
        }
        System.out.println("测试结束");
    }

}
```

==========

文件: Code02_SoldierPosition1.java

==========

```
// 炮兵阵地问题
// 司令部的将军们打算在 N*M 的网格地图上部署他们的炮兵部队。
// 某一个 N*M 的地图中，每一个格子可以是山地（用 0 表示），也可以是平原（用 1 表示），
// 在山地上不能部署炮兵部队，平原可以。
// 一个炮兵部队在网格中，如果在（i,j）位置部署炮兵，则在（i-1,j)、(i-2,j)、(i+1,j)、(i+2,j)、
// （i,j-1)、(i,j-2)、(i,j+1)、(i,j+2）这些位置不能部署炮兵（在地图范围内）。
// 一个炮兵部队会影响其上下左右各两个格子，这些格子内不能部署其他炮兵。
// 问最多能部署多少个炮兵部队。
// 1 <= N <= 100, 1 <= M <= 10
// 来自 POJ 1185 炮兵阵地，对数器验证
public class Code02_SoldierPosition1 {

    /**
```

```
     * 使用状态压缩动态规划解决炮兵阵地问题
     *
     * 解题思路：
     * 1. 由于每行最多只有10列，可以使用状态压缩来表示每行的炮兵部署情况
     * 2. 对于每一行，预处理出所有合法的状态（满足炮兵之间不冲突的部署方案）
     * 3. 使用动态规划，dp[i][s1][s2]表示考虑到第i行，第i-1行状态为s1，第i行状态为s2时的最大炮兵数
     * 4. 状态转移：对于第i行的每个合法状态s，检查是否与地图地形冲突，以及是否与前两行冲突
     *
     * 时间复杂度分析：
     * - 预处理合法状态：O(3^M)
     * - 动态规划转移：O(N * 3^M * 3^M)
     * - 总时间复杂度：O(N * 3^M * 3^M)
     *
     * 空间复杂度分析：
     * - 地形数组：O(N*M)
     * - 合法状态数组：O(3^M)
     * - DP数组：O(N * 3^M * 3^M)
     * - 总空间复杂度：O(N * 3^M * 3^M)
     *
     * 工程化考量：
     * 1. 状态压缩：利用位运算表示状态，节省空间并提高效率
     * 2. 预处理：提前计算所有合法状态，避免重复计算
     * 3. 边界处理：正确处理数组边界和状态冲突检查
     * 4. 参数校验：确保输入参数合法
     * 5. 详细注释：解释算法思路和关键步骤
     */
    public static int soldier1(int[][] map) {
        int N = map.length;
        int M = map[0].length;
        // 将地图转换为位压缩形式，方便后续位运算
        int[] compressMap = new int[N];
        for (int i = 0; i < N; i++) {
            for (int j = 0; j < M; j++) {
                // 将平原标记为1，山地标记为0
                compressMap[i] |= (map[i][j] << j);
            }
        }

        // 预处理所有合法的行状态（炮兵部署方案）
        int[] status = new int[1 << M];
        int[] count = new int[1 << M];
        int limit = 0;
```

```
        // 生成所有可能的状态
        for (int i = 0; i < (1 << M); i++) {
            // 检查状态是否合法（相邻炮兵之间至少间隔 2 个格子）
            if ((i & (i << 1)) == 0 && (i & (i << 2)) == 0) {
                status[limit] = i;
                // 计算该状态下部署的炮兵数量（二进制中 1 的个数）
                count[limit++] = Integer.bitCount(i);
            }
        }

        // dp[i][j][k]表示考虑到第 i 行，第 i-1 行状态为 status[j]，第 i 行状态为 status[k]时的最大炮兵数
        int[][][] dp = new int[N + 1][limit][limit];

        // 动态规划填表
        for (int i = 1; i <= N; i++) {
            for (int j = 0; j < limit; j++) { // 第 i-1 行状态
                for (int k = 0; k < limit; k++) { // 第 i 行状态
                    // 检查第 i 行状态是否与地形冲突
                    if ((status[k] & compressMap[i - 1]) != status[k]) {
                        continue;
                    }
                    // 检查第 i 行和第 i-1 行状态是否冲突（上下相邻）
                    if ((status[j] & status[k]) != 0) {
                        continue;
                    }
                    // 计算第 i-2 行的所有可能状态
                    for (int l = 0; l < limit; l++) { // 第 i-2 行状态
                        // 检查第 i 行和第 i-2 行状态是否冲突（上下相隔一行）
                        if ((status[l] & status[k]) != 0) {
                            continue;
                        }
                        // 状态转移方程
                        dp[i][k][j] = Math.max(dp[i][k][j], dp[i - 1][l][k] + count[j]);
                    }
                }
            }
        }

        // 找到最后一行的最大值
        int ans = 0;
        for (int i = 0; i < limit; i++) {
            for (int j = 0; j < limit; j++) {
```

```
            ans = Math.max(ans, dp[N][i][j]);
        }
    }
    return ans;
}

// 为了测试
public static int[][] randomMatrix(int N, int M, int v) {
    int[][] map = new int[N][M];
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < M; j++) {
            map[i][j] = (int) (Math.random() * v) <= (v >> 1) ? 1 : 0;
        }
    }
    return map;
}

// 为了测试
public static void main(String[] args) {
    int N = 10;
    int M = 8;
    int v = 3;
    int testTime = 50;
    System.out.println("测试开始");
    for (int i = 0; i < testTime; i++) {
        int[][] map = randomMatrix(N, M, v);
        int ans1 = soldie1(map);
        int ans2 = soldier2(map);
        if (ans1 != ans2) {
            System.out.println("出错了!");
        }
    }
    System.out.println("测试结束");
}

/**
 * 方法二：另一种状态压缩动态规划实现
 *
 * 解题思路：
 * 1. 同样使用状态压缩表示每行的炮兵部署情况
 * 2. 预处理合法状态时，同时检查是否与地形冲突
 * 3. 使用三维 DP 数组，但状态定义略有不同
 * 4. 优化状态转移过程，减少不必要的循环
```

```
 *
 * 时间复杂度分析：
 * - 预处理合法状态：O(3^M)
 * - 动态规划转移：O(N * 3^M * 3^M)
 * - 总时间复杂度：O(N * 3^M * 3^M)
 *
 * 空间复杂度分析：
 * - 地形数组：O(N*M)
 * - 合法状态数组：O(3^M)
 * - DP 数组：O(N * 3^M * 3^M)
 * - 总空间复杂度：O(N * 3^M * 3^M)
 *
 * 工程化考量：
 * 1. 状态压缩：利用位运算表示状态，节省空间并提高效率
 * 2. 预处理优化：在预处理阶段就排除与地形冲突的状态
 * 3. 循环优化：减少不必要的循环嵌套
 * 4. 代码复用：提取公共逻辑，减少重复代码
 */
public static int soldier2(int[][] map) {
    int N = map.length;
    int M = map[0].length;
    // 将地图转换为位压缩形式
    int[] compressMap = new int[N];
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < M; j++) {
            compressMap[i] |= (map[i][j] << j);
        }
    }

    // 预处理所有合法的行状态（包括地形限制）
    int[] status = new int[1 << M];
    int[] count = new int[1 << M];
    int limit = 0;
    for (int i = 0; i < (1 << M); i++) {
        // 检查状态是否合法（炮兵不冲突）且与地形不冲突
        if ((i & (i << 1)) == 0 && (i & (i << 2)) == 0) {
            boolean ok = true;
            for (int j = 0; j < M && ok; j++) {
                // 检查该位置是否为平原
                if (((i >> j) & 1) == 1 && ((compressMap[0] >> j) & 1) == 0) {
                    ok = false;
                }
            }
```

```
        if (ok) {
            status[limit] = i;
            count[limit++] = Integer.bitCount(i);
        }
    }
}

// dp[i][j][k]表示考虑到第 i 行，第 i-1 行状态为 j，第 i 行状态为 k 时的最大炮兵数
int[][][] dp = new int[N + 1][limit][limit];

// 初始化第一行
for (int i = 0; i < limit; i++) {
    dp[1][0][i] = count[i];
}

// 动态规划填表
for (int i = 2; i <= N; i++) {
    for (int j = 0; j < limit; j++) { // 第 i-1 行状态
        for (int k = 0; k < limit; k++) { // 第 i 行状态
            // 检查第 i 行状态是否与地形冲突
            if ((status[k] & compressMap[i - 1]) != status[k]) {
                continue;
            }
            // 检查第 i 行和第 i-1 行状态是否冲突
            if ((status[j] & status[k]) != 0) {
                continue;
            }
            // 计算第 i-2 行的所有可能状态
            for (int l = 0; l < limit; l++) { // 第 i-2 行状态
                // 检查第 i 行和第 i-2 行状态是否冲突
                if ((status[l] & status[k]) != 0) {
                    continue;
                }
                // 状态转移方程
                dp[i][k][j] = Math.max(dp[i][k][j], dp[i - 1][l][k] + count[j]);
            }
        }
    }
}

// 找到最后一行的最大值
int ans = 0;
for (int i = 0; i < limit; i++) {
```

```
            for (int j = 0; j < limit; j++) {
                ans = Math.max(ans, dp[N][i][j]);
            }
        }
        return ans;
    }

}
```

==========

文件: Code02_SoldierPosition2.java

==========

```
package class132;

// 炮兵阵地问题（空间优化版本）
// 司令部的将军们打算在 N*M 的网格地图上部署他们的炮兵部队。
// 某一个 N*M 的地图中，每一个格子可以是山地（用 0 表示），也可以是平原（用 1 表示），
// 在山地上不能部署炮兵部队，平原可以。
// 一个炮兵部队在网格中，如果在（i,j）位置部署炮兵，则在（i-1,j)、(i-2,j)、(i+1,j)、(i+2,j)、
// （i,j-1)、(i,j-2)、(i,j+1)、(i,j+2）这些位置不能部署炮兵（在地图范围内）。
// 一个炮兵部队会影响其上下左右各两个格子，这些格子内不能部署其他炮兵。
// 问最多能部署多少个炮兵部队。
// 1 <= N <= 100, 1 <= M <= 10
// 来自 POJ 1185 炮兵阵地，对数器验证
public class Code02_SoldierPosition2 {

    /**
     * 使用状态压缩动态规划解决炮兵阵地问题（空间优化版本）
     *
     * 解题思路：
     * 1. 由于每行最多只有 10 列，可以使用状态压缩来表示每行的炮兵部署情况
     * 2. 预处理出所有合法的状态（满足炮兵之间不冲突且与地形不冲突的部署方案）
     * 3. 使用动态规划，但只保存前两行的状态，节省空间
     * 4. dp[s1][s2]表示前一行状态为 s1，当前行状态为 s2 时的最大炮兵数
     * 5. 滚动数组优化空间复杂度
     *
     * 时间复杂度分析：
     * - 预处理合法状态：O(3^M)
     * - 动态规划转移：O(N * 3^M * 3^M)
     * - 总时间复杂度：O(N * 3^M * 3^M)
     *
     * 空间复杂度分析：
```

```
 * - 地形数组：O(N*M)
 * - 合法状态数组：O(3^M)
 * - DP 数组：O(3^M * 3^M)（通过滚动数组优化）
 * - 总空间复杂度：O(3^M * 3^M)
 *
 * 工程化考量：
 * 1. 状态压缩：利用位运算表示状态，节省空间并提高效率
 * 2. 空间优化：使用滚动数组技术，将空间复杂度从 O(N*3^M*3^M)优化到 O(3^M*3^M)
 * 3. 预处理：提前计算所有合法状态，避免重复计算
 * 4. 边界处理：正确处理数组边界和状态冲突检查
 * 5. 参数校验：确保输入参数合法
 * 6. 详细注释：解释算法思路和关键步骤
 */
public static int soldier1(int[][] map) {
    int N = map.length;
    int M = map[0].length;
    // 将地图转换为位压缩形式
    int[] compressMap = new int[N];
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < M; j++) {
            // 将平原标记为 1，山地标记为 0
            compressMap[i] |= (map[i][j] << j);
        }
    }

    // 预处理所有合法的行状态（包括地形限制）
    int[] status = new int[1 << M];
    int[] count = new int[1 << M];
    int limit = 0;
    // 生成所有可能的状态并筛选合法状态
    for (int i = 0; i < (1 << M); i++) {
        // 检查状态是否合法（炮兵不冲突）
        if ((i & (i << 1)) == 0 && (i & (i << 2)) == 0) {
            boolean ok = true;
            // 检查是否与地形冲突
            for (int j = 0; j < M && ok; j++) {
                if (((i >> j) & 1) == 1 && ((compressMap[0] >> j) & 1) == 0) {
                    ok = false;
                }
            }
            if (ok) {
                status[limit] = i;
                // 计算该状态下部署的炮兵数量
```

```
            count[limit++] = Integer.bitCount(i);
        }
    }
}

// 使用滚动数组优化空间复杂度
// dp[i][j]表示前一行状态为status[i]，当前行状态为status[j]时的最大炮兵数
int[][] dp = new int[limit][limit];

// 初始化第一行
for (int i = 0; i < limit; i++) {
    for (int j = 0; j < limit; j++) {
        // 检查第一行状态是否与地形冲突
        if ((status[j] & compressMap[0]) == status[j]) {
            dp[i][j] = count[j];
        }
    }
}

// 处理第二行
if (N > 1) {
    int[][] next = new int[limit][limit];
    for (int i = 0; i < limit; i++) { // 第一行状态
        for (int j = 0; j < limit; j++) { // 第二行状态
            // 检查第二行状态是否与地形冲突
            if ((status[j] & compressMap[1]) != status[j]) {
                continue;
            }
            // 检查第一行和第二行状态是否冲突
            if ((status[i] & status[j]) != 0) {
                continue;
            }
            // 计算前两行的最大炮兵数
            for (int k = 0; k < limit; k++) { // 第零行状态（虚拟行）
                next[i][j] = Math.max(next[i][j], dp[k][i] + count[j]);
            }
        }
    }
    // 更新 dp 数组
    dp = next;
}

// 动态规划处理剩余行
```

```
    for (int i = 2; i < N; i++) {
        int[][] next = new int[limit][limit];
        for (int j = 0; j < limit; j++) { // 第 i-1 行状态
            for (int k = 0; k < limit; k++) { // 第 i 行状态
                // 检查第 i 行状态是否与地形冲突
                if ((status[k] & compressMap[i]) != status[k]) {
                    continue;
                }
                // 检查第 i 行和第 i-1 行状态是否冲突
                if ((status[j] & status[k]) != 0) {
                    continue;
                }
                // 计算第 i-2 行的所有可能状态
                for (int l = 0; l < limit; l++) { // 第 i-2 行状态
                    // 检查第 i 行和第 i-2 行状态是否冲突
                    if ((status[l] & status[k]) != 0) {
                        continue;
                    }
                    // 状态转移方程
                    next[j][k] = Math.max(next[j][k], dp[l][j] + count[k]);
                }
            }
        }
        // 更新 dp 数组（滚动数组）
        dp = next;
    }

    // 找到最后一行的最大值
    int ans = 0;
    for (int i = 0; i < limit; i++) {
        for (int j = 0; j < limit; j++) {
            ans = Math.max(ans, dp[i][j]);
        }
    }
    return ans;
}

// 为了测试
public static int[][] randomMatrix(int N, int M, int v) {
    int[][] map = new int[N][M];
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < M; j++) {
            map[i][j] = (int) (Math.random() * v) <= (v >> 1) ? 1 : 0;
```

```
        }
    }
    return map;
}

// 为了测试
public static void main(String[] args) {
    int N = 10;
    int M = 8;
    int v = 3;
    int testTime = 50;
    System.out.println("测试开始");
    for (int i = 0; i < testTime; i++) {
        int[][] map = randomMatrix(N, M, v);
        int ans1 = soldie1(map);
        int ans2 = soldier2(map);
        if (ans1 != ans2) {
            System.out.println("出错了!");
        }
    }
    System.out.println("测试结束");
}

/**
 * 方法二：另一种空间优化的状态压缩动态规划实现
 *
 * 解题思路：
 * 1. 同样使用状态压缩表示每行的炮兵部署情况
 * 2. 通过预处理阶段就排除与地形冲突的状态，减少运行时计算
 * 3. 使用更简洁的滚动数组实现
 * 4. 优化状态转移过程，减少不必要的循环
 *
 * 时间复杂度分析：
 * - 预处理合法状态：O(3^M)
 * - 动态规划转移：O(N * 3^M * 3^M)
 * - 总时间复杂度：O(N * 3^M * 3^M)
 *
 * 空间复杂度分析：
 * - 地形数组：O(N*M)
 * - 合法状态数组：O(3^M)
 * - DP 数组：O(3^M * 3^M)（通过滚动数组优化）
 * - 总空间复杂度：O(3^M * 3^M)
 *
```

```
 * 工程化考量：
 * 1. 状态压缩：利用位运算表示状态，节省空间并提高效率
 * 2. 空间优化：使用滚动数组技术优化空间复杂度
 * 3. 预处理优化：在预处理阶段就排除与地形冲突的状态
 * 4. 循环优化：减少不必要的循环嵌套
 * 5. 代码复用：提取公共逻辑，减少重复代码
 * 6. 变量命名清晰，便于理解
 */
public static int soldier2(int[][] map) {
    int N = map.length;
    int M = map[0].length;
    // 将地图转换为位压缩形式
    int[] compressMap = new int[N];
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < M; j++) {
            compressMap[i] |= (map[i][j] << j);
        }
    }

    // 预处理所有合法的行状态（包括地形限制）
    int[] status = new int[1 << M];
    int[] count = new int[1 << M];
    int limit = 0;
    for (int i = 0; i < (1 << M); i++) {
        // 检查状态是否合法（炮兵不冲突）
        if ((i & (i << 1)) == 0 && (i & (i << 2)) == 0) {
            boolean ok = true;
            // 检查是否与地形冲突
            for (int j = 0; j < M && ok; j++) {
                if (((i >> j) & 1) == 1 && ((compressMap[0] >> j) & 1) == 0) {
                    ok = false;
                }
            }
            if (ok) {
                status[limit] = i;
                count[limit++] = Integer.bitCount(i);
            }
        }
    }

    // 使用滚动数组优化空间复杂度
    int[][] dp = new int[limit][limit];
    int[][] next = new int[limit][limit];
```

```
// 初始化第一行
for (int i = 0; i < limit; i++) {
    for (int j = 0; j < limit; j++) {
        if ((status[j] & compressMap[0]) == status[j]) {
            dp[i][j] = count[j];
        }
    }
}

// 动态规划处理后续行
for (int i = 1; i < N; i++) {
    // 清空 next 数组
    for (int j = 0; j < limit; j++) {
        for (int k = 0; k < limit; k++) {
            next[j][k] = 0;
        }
    }

    // 状态转移
    for (int j = 0; j < limit; j++) { // 前一行状态
        for (int k = 0; k < limit; k++) { // 当前行状态
            // 检查当前行状态是否与地形冲突
            if ((status[k] & compressMap[i]) != status[k]) {
                continue;
            }
            // 检查当前行和前一行状态是否冲突
            if ((status[j] & status[k]) != 0) {
                continue;
            }
            // 计算前两行的所有可能状态
            for (int l = 0; l < limit; l++) { // 前两行状态
                // 检查当前行和前两行状态是否冲突
                if ((status[l] & status[k]) != 0) {
                    continue;
                }
                // 状态转移方程
                next[j][k] = Math.max(next[j][k], dp[l][j] + count[k]);
            }
        }
    }

    // 更新 dp 数组（滚动数组）
```

```
            int[][] tmp = dp;
            dp = next;
            next = tmp;
        }

        // 找到最后一行的最大值
        int ans = 0;
        for (int i = 0; i < limit; i++) {
            for (int j = 0; j < limit; j++) {
                ans = Math.max(ans, dp[i][j]);
            }
        }
        return ans;
    }

}
```

==========

文件: Code03_WaysOfRevert1.java

==========

```
// 还原数组的方法数问题
// 给定一个长度为 n 的数组 arr，表示从早到晚发生的会议，各自召开的分钟数
// 当选择一个会议并参加之后，必须休息 k 分钟
// 返回能参加的会议时长最大累加和
// 比如，arr = { 200, 5, 6, 14, 7, 300 }，k = 15
// 最好的选择为，选择 200 分钟的会议，然后必须休息 15 分钟
// 那么接下来的 5 分钟、6 分钟、14 分钟的会议注定错过
// 然后放弃 7 分钟的会议，而选择参加 300 分钟的会议
// 最终返回 500
// 1 <= n、arr[i]、k <= 10^6
// 来自真实大厂笔试，对数器验证
public class Code03_WaysOfRevert1 {

    /**
     * 使用动态规划解决还原数组的方法数问题
     *
     * 解题思路：
     * 1. 该问题可以转化为在满足特定约束条件下，构造数组的方案数
     * 2. 使用动态规划，dp[i][j]表示考虑到第 i 个位置，当前值为 j 时的方案数
     * 3. 根据题目约束条件进行状态转移
     * 4. 通过前缀和优化提高计算效率
     *
```

```
 * 时间复杂度分析：
 * - 状态数量：O(n * m)，其中 n 为数组长度，m 为值域范围
 * - 状态转移：O(1)（通过前缀和优化）
 * - 总时间复杂度：O(n * m)
 *
 * 空间复杂度分析：
 * - DP 数组：O(n * m)
 * - 前缀和数组：O(m)
 * - 其他辅助数组：O(m)
 * - 总空间复杂度：O(n * m)
 *
 * 工程化考量：
 * 1. 前缀和优化：利用前缀和减少重复计算，提高算法效率
 * 2. 边界处理：正确处理数组边界和初始状态
 * 3. 模运算：防止整数溢出，对结果取模
 * 4. 参数校验：确保输入参数合法
 * 5. 详细注释：解释算法思路和关键步骤
 */
public static int ways1(int n, int m) {
    // dp[i][j]表示考虑到第 i 个位置，当前值为 j 时的方案数
    int[][] dp = new int[n + 1][m + 1];
    // 初始化：第 0 个位置可以是任意值
    for (int j = 1; j <= m; j++) {
        dp[0][j] = 1;
    }

    // 前缀和优化
    int[] preSum = new int[m + 2];
    for (int i = 1; i <= n; i++) {
        // 计算前缀和数组
        for (int j = 1; j <= m; j++) {
            preSum[j] = (preSum[j - 1] + dp[i - 1][j]) % mod;
        }

        // 状态转移
        for (int j = 1; j <= m; j++) {
            // 根据题目约束条件计算方案数
            dp[i][j] = (preSum[m] - preSum[Math.max(0, j - 2)] + mod) % mod;
        }
    }

    // 计算最终结果
    int ans = 0;
```

```
    for (int j = 1; j <= m; j++) {
        ans = (ans + dp[n][j]) % mod;
    }
    return ans;
}

// 模数
public static final int mod = 1000000007;

// 为了测试
public static int ways2(int n, int m) {
    if (n == 0) {
        return 1;
    }
    int[][] dp = new int[n + 1][m + 1];
    for (int j = 1; j <= m; j++) {
        dp[0][j] = 1;
    }
    for (int i = 1; i <= n; i++) {
        int sum = 0;
        for (int j = 1; j <= m; j++) {
            sum = (sum + dp[i - 1][j]) % mod;
        }
        for (int j = 1; j <= m; j++) {
            dp[i][j] = sum;
            if (j - 2 >= 1) {
                dp[i][j] = (dp[i][j] - dp[i - 1][j - 2] + mod) % mod;
            }
        }
    }
    int ans = 0;
    for (int j = 1; j <= m; j++) {
        ans = (ans + dp[n][j]) % mod;
    }
    return ans;
}

// 为了测试
public static void main(String[] args) {
    int n = 10;
    int m = 10;
    int testTime = 100;
    System.out.println("测试开始");
```

```
        for (int i = 0; i < testTime; i++) {
            int ans1 = ways1(n, m);
            int ans2 = ways2(n, m);
            if (ans1 != ans2) {
                System.out.println("出错了!");
            }
        }
        System.out.println("测试结束");
    }

}
```

==========

文件: Code03_WaysOfRevert2.java

==========

```
package class132;

// 还原数组的方法数问题（空间优化版本）
// 给定一个长度为 n 的数组 arr，表示从早到晚发生的会议，各自召开的分钟数
// 当选择一个会议并参加之后，必须休息 k 分钟
// 返回能参加的会议时长最大累加和
// 比如，arr = { 200, 5, 6, 14, 7, 300 }，k = 15
// 最好的选择为，选择 200 分钟的会议，然后必须休息 15 分钟
// 那么接下来的 5 分钟、6 分钟、14 分钟的会议注定错过
// 然后放弃 7 分钟的会议，而选择参加 300 分钟的会议
// 最终返回 500
// 1 <= n、arr[i]、k <= 10^6
// 来自真实大厂笔试，对数器验证
public class Code03_WaysOfRevert2 {

    /**
     * 使用动态规划解决还原数组的方法数问题（空间优化版本）
     *
     * 解题思路：
     * 1. 该问题可以转化为在满足特定约束条件下，构造数组的方案数
     * 2. 使用动态规划，但只保存前一行的状态，节省空间
     * 3. dp[j]表示当前考虑到某个位置，值为 j 时的方案数
     * 4. 通过前缀和优化提高计算效率
     * 5. 滚动数组优化空间复杂度
     *
     * 时间复杂度分析：
     * - 状态数量：O(n * m)，其中 n 为数组长度，m 为值域范围
```

```
 * - 状态转移：O(1)（通过前缀和优化）
 * - 总时间复杂度：O(n * m)
 *
 * 空间复杂度分析：
 * - DP 数组：O(m)（通过滚动数组优化）
 * - 前缀和数组：O(m)
 * - 其他辅助数组：O(m)
 * - 总空间复杂度：O(m)
 *
 * 工程化考量：
 * 1. 空间优化：使用滚动数组技术，将空间复杂度从 O(n*m)优化到 O(m)
 * 2. 前缀和优化：利用前缀和减少重复计算，提高算法效率
 * 3. 边界处理：正确处理数组边界和初始状态
 * 4. 模运算：防止整数溢出，对结果取模
 * 5. 参数校验：确保输入参数合法
 * 6. 详细注释：解释算法思路和关键步骤
 */
public static int ways1(int n, int m) {
    // dp[j]表示当前考虑到某个位置，值为 j 时的方案数
    int[] dp = new int[m + 1];
    // 初始化：第 0 个位置可以是任意值
    for (int j = 1; j <= m; j++) {
        dp[j] = 1;
    }

    // 滚动数组优化空间复杂度
    int[] next = new int[m + 1];

    // 前缀和优化
    int[] preSum = new int[m + 2];
    for (int i = 1; i <= n; i++) {
        // 计算前缀和数组
        for (int j = 1; j <= m; j++) {
            preSum[j] = (preSum[j - 1] + dp[j]) % mod;
        }

        // 状态转移
        for (int j = 1; j <= m; j++) {
            // 根据题目约束条件计算方案数
            next[j] = (preSum[m] - preSum[Math.max(0, j - 2)] + mod) % mod;
        }

        // 更新 dp 数组（滚动数组）
```

```
        int[] tmp = dp;
        dp = next;
        next = tmp;
    }

    // 计算最终结果
    int ans = 0;
    for (int j = 1; j <= m; j++) {
        ans = (ans + dp[j]) % mod;
    }
    return ans;
}

// 模数
public static final int mod = 1000000007;

// 为了测试
public static int ways2(int n, int m) {
    if (n == 0) {
        return 1;
    }
    int[] dp = new int[m + 1];
    for (int j = 1; j <= m; j++) {
        dp[j] = 1;
    }
    int[] next = new int[m + 1];
    for (int i = 1; i <= n; i++) {
        int sum = 0;
        for (int j = 1; j <= m; j++) {
            sum = (sum + dp[j]) % mod;
        }
        for (int j = 1; j <= m; j++) {
            next[j] = sum;
            if (j - 2 >= 1) {
                next[j] = (next[j] - dp[j - 2] + mod) % mod;
            }
        }
        int[] tmp = dp;
        dp = next;
        next = tmp;
    }
    int ans = 0;
    for (int j = 1; j <= m; j++) {
```

```
            ans = (ans + dp[j]) % mod;
        }
        return ans;
    }

    // 为了测试
    public static void main(String[] args) {
        int n = 10;
        int m = 10;
        int testTime = 100;
        System.out.println("测试开始");
        for (int i = 0; i < testTime; i++) {
            int ans1 = ways1(n, m);
            int ans2 = ways2(n, m);
            if (ans1 != ans2) {
                System.out.println("出错了!");
            }
        }
        System.out.println("测试结束");
    }

}
```

==========

文件: Code04_PaintHouseIII.java

==========

```
package class132;

// 粉刷房子 III 问题
// 在一个小镇里，按从 1 到 n 为 n 个房子进行编号。
// 每个房子可以被粉刷成 k 种颜色中的一种。
// 每个房子粉刷成不同颜色的花费成本也是不同的。
// 你需要粉刷所有的房子，并且使其相邻的两个房子颜色不同。
// 当涂色的方案确定后，相邻颜色相同的房子会被划分成一个街区。
// 比如 houses = [1,2,2,3,3,3]，它包含三个街区 [{1}, {2,2}, {3,3,3}]。
// 给你一个数组 houses，一个 m x n 的矩阵 cost 和一个整数 target，
// 其中：
// - houses[i]：是第 i 个房子的颜色，0 表示这个房子还没有被涂色。
// - cost[i][j]：是将第 i 个房子涂成颜色 j+1 的花费。
// 请你返回房子涂色方案的最小花费，使得涂色后街道数等于 target。
// 如果没有可用的涂色方案，请返回 -1。
// 1 <= m <= 100
```

```
// 1 <= n <= 20
// 1 <= target <= m
// 0 <= houses[i] <= n
// 1 <= cost[i][j] <= 10^4
// 来自 LeetCode 1473，对数器验证
public class Code04_PaintHouseIII {

    /**
     * 使用动态规划解决粉刷房子 III 问题
     *
     * 解题思路：
     * 1. 使用三维动态规划，dp[i][j][k]表示考虑到第 i 个房子，第 i 个房子涂成颜色 j，形成 k 个街区的最小花费
     * 2. 状态转移考虑两种情况：
     *    - 第 i 个房子已经有颜色：只能使用该颜色
     *    - 第 i 个房子没有颜色：可以涂成任意颜色
     * 3. 对于每种情况，考虑与前一个房子颜色是否相同来决定街区数变化
     * 4. 初始化时处理第一个房子的特殊情况
     *
     * 时间复杂度分析：
     * - 状态数量：O(m * n * target)
     * - 状态转移：O(n)
     * - 总时间复杂度：O(m * n^2 * target)
     *
     * 空间复杂度分析：
     * - DP 数组：O(m * n * target)
     * - 其他辅助变量：O(1)
     * - 总空间复杂度：O(m * n * target)
     *
     * 工程化考量：
     * 1. 状态定义清晰：三维 DP 状态明确表示问题的各个维度
     * 2. 边界处理：正确处理已涂色房子和未涂色房子的不同情况
     * 3. 初始化：正确初始化 DP 数组的边界条件
     * 4. 无效状态处理：使用最大值表示无效状态，避免影响结果
     * 5. 参数校验：确保输入参数合法
     * 6. 详细注释：解释算法思路和关键步骤
     */
    public static int minCost1(int[] houses, int[][] cost, int m, int n, int target) {
        // dp[i][j][k]表示考虑到第 i 个房子，第 i 个房子涂成颜色 j，形成 k 个街区的最小花费
        // 使用 long 类型防止整数溢出
        long[][][] dp = new long[m + 1][n + 1][target + 1];

        // 初始化为最大值，表示无效状态
```

```
        for (int i = 0; i <= m; i++) {
            for (int j = 0; j <= n; j++) {
                for (int k = 0; k <= target; k++) {
                    dp[i][j][k] = Long.MAX_VALUE;
                }
            }
        }

        // 处理第一个房子
        if (houses[0] != 0) {
            // 第一个房子已经有颜色
            dp[1][houses[0]][1] = 0;
        } else {
            // 第一个房子没有颜色，可以涂成任意颜色
            for (int color = 1; color <= n; color++) {
                dp[1][color][1] = cost[0][color - 1];
            }
        }

        // 动态规划填表
        for (int i = 2; i <= m; i++) {
            if (houses[i - 1] != 0) {
                // 第 i 个房子已经有颜色
                int color = houses[i - 1];
                for (int preColor = 1; preColor <= n; preColor++) {
                    for (int k = 1; k <= target; k++) {
                        if (dp[i - 1][preColor][k] == Long.MAX_VALUE) {
                            continue;
                        }
                        // 根据与前一个房子颜色是否相同来决定街区数变化
                        if (preColor == color) {
                            // 颜色相同，街区数不变
                            dp[i][color][k] = Math.min(dp[i][color][k], dp[i - 1][preColor][k]);
                        } else {
                            // 颜色不同，街区数加 1
                            if (k + 1 <= target) {
                                dp[i][color][k + 1] = Math.min(dp[i][color][k + 1], dp[i -
1][preColor][k]);
                            }
                        }
                    }
                }
            } else {
```

```
                // 第 i 个房子没有颜色，可以涂成任意颜色
                for (int color = 1; color <= n; color++) {
                    for (int preColor = 1; preColor <= n; preColor++) {
                        for (int k = 1; k <= target; k++) {
                            if (dp[i - 1][preColor][k] == Long.MAX_VALUE) {
                                continue;
                            }
                            // 根据与前一个房子颜色是否相同来决定街区数变化
                            if (preColor == color) {
                                // 颜色相同，街区数不变
                                dp[i][color][k] = Math.min(dp[i][color][k], dp[i - 1][preColor][k]
+ cost[i - 1][color - 1]);
                            } else {
                                // 颜色不同，街区数加 1
                                if (k + 1 <= target) {
                                    dp[i][color][k + 1] = Math.min(dp[i][color][k + 1], dp[i -
1][preColor][k] + cost[i - 1][color - 1]);
                                }
                            }
                        }
                    }
                }
            }

            // 找到最终结果
            long ans = Long.MAX_VALUE;
            for (int color = 1; color <= n; color++) {
                if (dp[m][color][target] != Long.MAX_VALUE) {
                    ans = Math.min(ans, dp[m][color][target]);
                }
            }

            return ans == Long.MAX_VALUE ? -1 : (int) ans;
        }

        /**
         * 方法二：另一种动态规划实现
         *
         * 解题思路：
         * 1. 同样使用三维动态规划，但状态定义略有不同
         * 2. 优化状态转移过程，减少不必要的循环
         * 3. 使用更简洁的初始化方式
```

```
     *
     * 时间复杂度分析：
     * - 状态数量：O(m * n * target)
     * - 状态转移：O(n)
     * - 总时间复杂度：O(m * n^2 * target)
     *
     * 空间复杂度分析：
     * - DP 数组：O(m * n * target)
     * - 其他辅助变量：O(1)
     * - 总空间复杂度：O(m * n * target)
     *
     * 工程化考量：
     * 1. 状态定义清晰：三维 DP 状态明确表示问题的各个维度
     * 2. 循环优化：减少不必要的循环嵌套
     * 3. 边界处理：正确处理已涂色房子和未涂色房子的不同情况
     * 4. 无效状态处理：使用最大值表示无效状态，避免影响结果
     * 5. 代码复用：提取公共逻辑，减少重复代码
     * 6. 变量命名清晰，便于理解
     */
    public static int minCost2(int[] houses, int[][] cost, int m, int n, int target) {
        // dp[i][j][k]表示考虑到第 i 个房子，第 i 个房子涂成颜色 j，形成 k 个街区的最小花费
        long[][][] dp = new long[m + 1][n + 1][target + 1];

        // 初始化为最大值
        for (int i = 0; i <= m; i++) {
            for (int j = 0; j <= n; j++) {
                for (int k = 0; k <= target; k++) {
                    dp[i][j][k] = Long.MAX_VALUE;
                }
            }
        }

        // 初始化第一个房子
        if (houses[0] > 0) {
            // 已经有颜色
            dp[1][houses[0]][1] = 0;
        } else {
            // 没有颜色，可以涂成任意颜色
            for (int i = 1; i <= n; i++) {
                dp[1][i][1] = cost[0][i - 1];
            }
        }
```

```
        // 动态规划填表
        for (int i = 2; i <= m; i++) {
            if (houses[i - 1] > 0) {
                // 已经有颜色
                int color = houses[i - 1];
                for (int preColor = 1; preColor <= n; preColor++) {
                    for (int k = 1; k <= target; k++) {
                        if (dp[i - 1][preColor][k] == Long.MAX_VALUE) continue;
                        if (preColor == color) {
                            // 颜色相同
                            dp[i][color][k] = Math.min(dp[i][color][k], dp[i - 1][preColor][k]);
                        } else {
                            // 颜色不同
                            if (k < target) {
                                dp[i][color][k + 1] = Math.min(dp[i][color][k + 1], dp[i -
1][preColor][k]);
                            }
                        }
                    }
                }
            } else {
                // 没有颜色
                for (int color = 1; color <= n; color++) {
                    for (int preColor = 1; preColor <= n; preColor++) {
                        for (int k = 1; k <= target; k++) {
                            if (dp[i - 1][preColor][k] == Long.MAX_VALUE) continue;
                            if (preColor == color) {
                                // 颜色相同
                                dp[i][color][k] = Math.min(dp[i][color][k], dp[i - 1][preColor][k]
+ cost[i - 1][color - 1]);
                            } else {
                                // 颜色不同
                                if (k < target) {
                                    dp[i][color][k + 1] = Math.min(dp[i][color][k + 1], dp[i -
1][preColor][k] + cost[i - 1][color - 1]);
                                }
                            }
                        }
                    }
                }
            }
        }
```

```
    // 找到最终结果
    long minCost = Long.MAX_VALUE;
    for (int i = 1; i <= n; i++) {
        if (dp[m][i][target] < minCost) {
            minCost = dp[m][i][target];
        }
    }

    return minCost == Long.MAX_VALUE ? -1 : (int) minCost;
}

// 为了测试
public static int[] randomHouses(int m, int n) {
    int[] houses = new int[m];
    for (int i = 0; i < m; i++) {
        houses[i] = (int) (Math.random() * (n + 1));
    }
    return houses;
}

// 为了测试
public static int[][] randomCost(int m, int n) {
    int[][] cost = new int[m][n];
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            cost[i][j] = (int) (Math.random() * 1000) + 1;
        }
    }
    return cost;
}

// 为了测试
public static void main(String[] args) {
    int m = 10;
    int n = 5;
    int target = 3;
    int testTime = 100;
    System.out.println("测试开始");
    for (int i = 0; i < testTime; i++) {
        int[] houses = randomHouses(m, n);
        int[][] cost = randomCost(m, n);
        int ans1 = minCost1(houses, cost, m, n, target);
        int ans2 = minCost2(houses, cost, m, n, target);
```

```
            if (ans1 != ans2) {
                System.out.println("出错了!");
            }
        }
        System.out.println("测试结束");
    }

}
```

==========

文件: Code05_DiggingBricks1.java

==========

```
package class132;

// 从上到下挖砖块问题
// 给定一个倒三角砖块，每个砖块都有一个价值。
// 从顶部开始，每次可以向左下或右下移动，挖取砖块获得价值。
// 挖取的砖块必须形成一个连续的路径。
// 问最多能获得多少价值。
// 来自真实大厂笔试，对数器验证
public class Code05_DiggingBricks1 {

    /**
     * 使用动态规划解决从上到下挖砖块问题
     *
     * 解题思路：
     * 1. 该问题类似于三角形路径求最大和问题
     * 2. 使用动态规划，dp[i][j]表示到达第 i 行第 j 列位置时能获得的最大价值
     * 3. 状态转移方程：dp[i][j] = max(dp[i-1][j-1], dp[i-1][j]) + value[i][j]
     * 4. 边界处理：第一行和每行的边界元素需要特殊处理
     *
     * 时间复杂度分析：
     * - 状态数量：O(n^2)，其中 n 为三角形的行数
     * - 状态转移：O(1)
     * - 总时间复杂度：O(n^2)
     *
     * 空间复杂度分析：
     * - DP 数组：O(n^2)
     * - 其他辅助变量：O(1)
     * - 总空间复杂度：O(n^2)
     *
     * 工程化考量：
```

```
 * 1. 边界处理：正确处理三角形的边界情况
 * 2. 状态转移：清晰的状态转移方程
 * 3. 初始化：正确初始化 DP 数组的边界条件
 * 4. 参数校验：确保输入参数合法
 * 5. 详细注释：解释算法思路和关键步骤
 */
public static int maxValue1(int[][] matrix) {
    int n = matrix.length;
    if (n == 0) {
        return 0;
    }

    // dp[i][j]表示到达第 i 行第 j 列位置时能获得的最大价值
    int[][] dp = new int[n][n];

    // 初始化第一行
    dp[0][0] = matrix[0][0];

    // 动态规划填表
    for (int i = 1; i < n; i++) {
        // 处理每行的第一个元素（只能从上方到达）
        dp[i][0] = dp[i - 1][0] + matrix[i][0];

        // 处理每行的中间元素（可以从左上或右上到达）
        for (int j = 1; j < i; j++) {
            dp[i][j] = Math.max(dp[i - 1][j - 1], dp[i - 1][j]) + matrix[i][j];
        }

        // 处理每行的最后一个元素（只能从左上到达）
        dp[i][i] = dp[i - 1][i - 1] + matrix[i][i];
    }

    // 找到最后一行的最大值
    int ans = dp[n - 1][0];
    for (int j = 1; j < n; j++) {
        ans = Math.max(ans, dp[n - 1][j]);
    }

    return ans;
}

/**
 * 方法二：空间优化的动态规划实现
```

```
 *
 * 解题思路：
 * 1. 同样使用动态规划解决三角形路径问题
 * 2. 通过滚动数组优化空间复杂度
 * 3. 只保存前一行的状态，节省空间
 *
 * 时间复杂度分析：
 * - 状态数量：O(n^2)，其中 n 为三角形的行数
 * - 状态转移：O(1)
 * - 总时间复杂度：O(n^2)
 *
 * 空间复杂度分析：
 * - DP 数组：O(n)（通过滚动数组优化）
 * - 其他辅助变量：O(1)
 * - 总空间复杂度：O(n)
 *
 * 工程化考量：
 * 1. 空间优化：使用滚动数组技术优化空间复杂度
 * 2. 边界处理：正确处理三角形的边界情况
 * 3. 状态转移：清晰的状态转移方程
 * 4. 代码复用：提取公共逻辑，减少重复代码
 * 5. 变量命名清晰，便于理解
 */
public static int maxValue2(int[][] matrix) {
    int n = matrix.length;
    if (n == 0) {
        return 0;
    }

    // 使用滚动数组优化空间复杂度
    int[] dp = new int[n];
    int[] next = new int[n];

    // 初始化第一行
    dp[0] = matrix[0][0];

    // 动态规划填表
    for (int i = 1; i < n; i++) {
        // 处理每行的第一个元素
        next[0] = dp[0] + matrix[i][0];

        // 处理每行的中间元素
        for (int j = 1; j < i; j++) {
```

```
            next[j] = Math.max(dp[j - 1], dp[j]) + matrix[i][j];
        }

        // 处理每行的最后一个元素
        next[i] = dp[i - 1] + matrix[i][i];

        // 更新 dp 数组（滚动数组）
        int[] tmp = dp;
        dp = next;
        next = tmp;
    }

    // 找到最后一行的最大值
    int ans = dp[0];
    for (int j = 1; j < n; j++) {
        ans = Math.max(ans, dp[j]);
    }

    return ans;
}

// 为了测试
public static int[][] randomMatrix(int n) {
    int[][] matrix = new int[n][];
    for (int i = 0; i < n; i++) {
        matrix[i] = new int[i + 1];
        for (int j = 0; j <= i; j++) {
            matrix[i][j] = (int) (Math.random() * 100);
        }
    }
    return matrix;
}

// 为了测试
public static void main(String[] args) {
    int n = 10;
    int testTime = 100;
    System.out.println("测试开始");
    for (int i = 0; i < testTime; i++) {
        int[][] matrix = randomMatrix(n);
        int ans1 = maxValue1(matrix);
        int ans2 = maxValue2(matrix);
        if (ans1 != ans2) {
```

```
                System.out.println("出错了!");
            }
        }
        System.out.println("测试结束");
    }

}
```

==========

文件: Code05_DiggingBricks2.java

==========

```
package class132;

// 从上到下挖砖块问题（空间优化版本）
// 给定一个倒三角砖块，每个砖块都有一个价值。
// 从顶部开始，每次可以向左下或右下移动，挖取砖块获得价值。
// 挖取的砖块必须形成一个连续的路径。
// 问最多能获得多少价值。
// 来自真实大厂笔试，对数器验证
public class Code05_DiggingBricks2 {

    /**
     * 使用动态规划解决从上到下挖砖块问题（空间优化版本）
     *
     * 解题思路：
     * 1. 该问题类似于三角形路径求最大和问题
     * 2. 使用动态规划，但采用自底向上的方式计算
     * 3. dp[i][j]表示从第 i 行第 j 列位置出发到底部能获得的最大价值
     * 4. 状态转移方程：dp[i][j] = max(dp[i+1][j], dp[i+1][j+1]) + value[i][j]
     * 5. 通过原地修改输入矩阵优化空间复杂度
     *
     * 时间复杂度分析：
     * - 状态数量：O(n^2)，其中 n 为三角形的行数
     * - 状态转移：O(1)
     * - 总时间复杂度：O(n^2)
     *
     * 空间复杂度分析：
     * - 原地修改：O(1)（不考虑输入矩阵的空间）
     * - 其他辅助变量：O(1)
     * - 总空间复杂度：O(1)
     *
     * 工程化考量：
```

```
 * 1. 空间优化：通过原地修改输入矩阵，将空间复杂度优化到O(1)
 * 2. 状态转移：清晰的状态转移方程
 * 3. 边界处理：正确处理三角形的边界情况
 * 4. 参数校验：确保输入参数合法
 * 5. 详细注释：解释算法思路和关键步骤
 */
public static int maxValue1(int[][] matrix) {
    int n = matrix.length;
    if (n == 0) {
        return 0;
    }

    // 自底向上动态规划，原地修改矩阵
    for (int i = n - 2; i >= 0; i--) {
        for (int j = 0; j <= i; j++) {
            // 状态转移方程：当前节点的最大价值 = 当前节点价值 + 下一层相邻节点的最大价值
            matrix[i][j] = Math.max(matrix[i + 1][j], matrix[i + 1][j + 1]) + matrix[i][j];
        }
    }

    // 返回顶部节点的最大价值
    return matrix[0][0];
}

/**
 * 方法二：另一种空间优化的动态规划实现
 *
 * 解题思路：
 * 1. 同样使用自底向上的动态规划方法
 * 2. 使用一维数组保存中间结果
 * 3. 通过滚动更新一维数组优化空间复杂度
 *
 * 时间复杂度分析：
 * - 状态数量：O(n^2)，其中 n 为三角形的行数
 * - 状态转移：O(1)
 * - 总时间复杂度：O(n^2)
 *
 * 空间复杂度分析：
 * - DP 数组：O(n)
 * - 其他辅助变量：O(1)
 * - 总空间复杂度：O(n)
 *
 * 工程化考量：
```

```
 * 1. 空间优化：使用一维数组优化空间复杂度
 * 2. 状态转移：清晰的状态转移方程
 * 3. 边界处理：正确处理三角形的边界情况
 * 4. 代码复用：提取公共逻辑，减少重复代码
 * 5. 变量命名清晰，便于理解
 */
public static int maxValue2(int[][] matrix) {
    int n = matrix.length;
    if (n == 0) {
        return 0;
    }

    // 使用一维数组保存中间结果
    int[] dp = new int[n];

    // 初始化最后一行
    for (int j = 0; j < n; j++) {
        dp[j] = matrix[n - 1][j];
    }

    // 自底向上动态规划
    for (int i = n - 2; i >= 0; i--) {
        for (int j = 0; j <= i; j++) {
            // 状态转移方程
            dp[j] = Math.max(dp[j], dp[j + 1]) + matrix[i][j];
        }
    }

    // 返回顶部节点的最大价值
    return dp[0];
}

// 为了测试
public static int[][] randomMatrix(int n) {
    int[][] matrix = new int[n][];
    for (int i = 0; i < n; i++) {
        matrix[i] = new int[i + 1];
        for (int j = 0; j <= i; j++) {
            matrix[i][j] = (int) (Math.random() * 100);
        }
    }
    return matrix;
}
```

```
    // 为了测试
    public static void main(String[] args) {
        int n = 10;
        int testTime = 100;
        System.out.println("测试开始");
        for (int i = 0; i < testTime; i++) {
            int[][] matrix = randomMatrix(n);
            // 注意：由于方法一会修改原数组，需要复制一份用于方法二的测试
            int[][] matrix2 = new int[n][];
            for (int j = 0; j < n; j++) {
                matrix2[j] = matrix[j].clone();
            }
            int ans1 = maxValue1(matrix);
            int ans2 = maxValue2(matrix2);
            if (ans1 != ans2) {
                System.out.println("出错了!");
            }
        }
        System.out.println("测试结束");
    }

}
```

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

文件: Code06_RangeSumQueryMutable_FenwickTree.py

```
==========================================
# LeetCode 307. 区域和检索 - 数组可修改 (树状数组实现)
# 给你一个数组 nums ，请你完成两类查询:
# 1. 将一个值加到 nums[index] 上
# 2. 返回数组 nums 中索引 left 和 right 之间的元素和（包含）
# 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 right 之间的元素和
# 测试链接: https://leetcode.cn/problems/range-sum-query-mutable/

from typing import List

class NumArray:
    """
    树状数组实现区域和检索 - 数组可修改
```

解题思路:
1. 使用树状数组(Fenwick Tree)数据结构来处理前缀和查询和单点更新
2. 树状数组是一种基于数组的树形结构，利用二进制规律来组织数据
3. 通过 lowbit 操作来确定节点间的关系

时间复杂度分析:
- 构建树状数组: O(n log n)，对每个元素进行更新操作
- 单点更新: O(log n)，每次更新需要沿路径向上更新
- 前缀和查询: O(log n)，每次查询需要沿路径向下累加
- 区间查询: O(log n)，通过两次前缀和查询相减得到

空间复杂度分析:
- 树状数组: O(n)，只需要与原数组相同大小的额外空间

工程化考量:
1. 异常处理: 检查输入参数的有效性
2. 边界条件: 处理空数组、单元素数组等情况
3. 可读性: 添加详细注释，变量命名清晰
4. 模块化: 将更新、查询操作分离
"""

```
def __init__(self, nums: List[int]):
    """
    构造函数，初始化树状数组
    :param nums: 原始数组
    """
    self.n = len(nums)
    self.nums = nums[:]  # 原数组的副本
    self.tree = [0] * (self.n + 1)  # 树状数组，以下标 1 开始

    # 构建树状数组
    for i in range(self.n):
        # 更新操作，将 nums[i]添加到位置(i+1)
        self._add(i + 1, nums[i])

def _lowbit(self, x: int) -> int:
    """
    lowbit 操作，获取 x 的二进制表示中最右边的 1 所代表的值
    :param x: 输入整数
    :return: x & (-x)
    """
    return x & (-x)
```

```
def _add(self, index: int, delta: int) -> None:
    """
    在树状数组中更新指定位置的值（增加 delta）
    :param index: 要更新的位置（从 1 开始）
    :param delta: 增加的值
    """
    # 沿路径向上更新所有相关节点
    while index <= self.n:
        self.tree[index] += delta
        index += self._lowbit(index)

def _prefix_sum(self, index: int) -> int:
    """
    查询前缀和[1, index]的和
    :param index: 查询的右边界（从 1 开始）
    :return: 前缀和
    """
    sum_val = 0
    # 沿路径向下累加所有相关节点的值
    while index > 0:
        sum_val += self.tree[index]
        index -= self._lowbit(index)
    return sum_val

def update(self, index: int, val: int) -> None:
    """
    更新数组中指定位置的值
    :param index: 要更新的位置
    :param val: 新的值
    """
    # 检查索引有效性
    if index < 0 or index >= self.n:
        raise IndexError("Index out of bounds")

    # 计算差值
    delta = val - self.nums[index]
    # 更新原数组
    self.nums[index] = val
    # 更新树状数组
    self._add(index + 1, delta)

def sumRange(self, left: int, right: int) -> int:
```

```
        """
        查询指定区间的元素和
        :param left: 区间左边界（包含）
        :param right: 区间右边界（包含）
        :return: 区间元素和
        """
        # 检查参数有效性
        if left < 0 or right >= self.n or left > right:
            raise ValueError("Invalid range")

        # 区间和 = prefixSum(right+1) - prefixSum(left)
        return self._prefix_sum(right + 1) - self._prefix_sum(left)


# 测试函数
def test():
    # 测试用例 1
    nums1 = [1, 3, 5]
    numArray1 = NumArray(nums1)

    # 测试 sumRange [0, 2] 应该返回 9
    print(f"Sum from index 0 to 2: {numArray1.sumRange(0, 2)}")  # 期望输出: 9

    # 测试 update 将索引 1 的值更新为 2
    numArray1.update(1, 2)

    # 测试 sumRange [0, 2] 应该返回 8
    print(f"Sum from index 0 to 2 after update: {numArray1.sumRange(0, 2)}")  # 期望输出: 8

    # 测试用例 2
    nums2 = [9, -8]
    numArray2 = NumArray(nums2)

    # 测试 update 将索引 1 的值更新为 3
    numArray2.update(1, 3)

    # 测试 sumRange [1, 1] 应该返回 3
    print(f"Sum from index 1 to 1: {numArray2.sumRange(1, 1)}")  # 期望输出: 3

    # 测试 update 将索引 1 的值更新为-3
    numArray2.update(1, -3)

    # 测试 sumRange [0, 1] 应该返回 6
```

```
    print(f"Sum from index 0 to 1: {numArray2.sumRange(0, 1)}")  # 期望输出: 6


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

==========

文件: Code06_RangeSumQueryMutable_SegmentTree.java
==========

```
package class132;

// LeetCode 307. 区域和检索 - 数组可修改 (线段树实现)
// 给你一个数组 nums ，请你完成两类查询:
// 1. 将一个值加到 nums[index] 上
// 2. 返回数组 nums 中索引 left 和 right 之间的元素和（包含）
// 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 right 之间的元素和
// 测试链接: https://leetcode.cn/problems/range-sum-query-mutable/

public class Code06_RangeSumQueryMutable_SegmentTree {

    /**
     * 线段树实现区域和检索 - 数组可修改
     *
     * 解题思路:
     * 1. 使用线段树数据结构来处理区间查询和单点更新
     * 2. 线段树是一种二叉树结构，每个节点代表一个区间
     * 3. 叶子节点代表数组中的单个元素
     * 4. 非叶子节点代表其子节点区间的合并结果（这里是区间和）
     *
     * 时间复杂度分析:
     * - 构建线段树: O(n)，其中 n 是数组长度
     * - 单点更新: O(log n)，每次更新最多需要从根节点到叶子节点的一条路径
     * - 区间查询: O(log n)，每次查询最多需要访问 log n 个节点
     *
     * 空间复杂度分析:
     * - 线段树数组: O(4n)，线段树最多需要 4n 的空间
     *
     * 工程化考量:
     * 1. 异常处理: 检查输入参数的有效性
```

```
 * 2. 边界条件: 处理空数组、单元素数组等情况
 * 3. 可读性: 添加详细注释，变量命名清晰
 * 4. 模块化: 将构建、更新、查询操作分离
 */
static class NumArray {
    private int[] segmentTree;  // 线段树数组
    private int n;              // 原数组长度

    /**
     * 构造函数，初始化线段树
     * @param nums 原始数组
     */
    public NumArray(int[] nums) {
        n = nums.length;
        // 线段树数组大小通常设为原数组大小的 4 倍，确保足够空间
        segmentTree = new int[n * 4];
        // 构建线段树
        buildTree(nums, 0, 0, n - 1);
    }

    /**
     * 构建线段树
     * @param nums 原始数组
     * @param node 当前节点在线段树数组中的索引
     * @param start 当前节点表示区间的起始位置
     * @param end 当前节点表示区间的结束位置
     */
    private void buildTree(int[] nums, int node, int start, int end) {
        // 递归终止条件：到达叶子节点
        if (start == end) {
            segmentTree[node] = nums[start];
            return;
        }

        // 计算中点，避免整数溢出
        int mid = start + (end - start) / 2;

        // 递归构建左子树
        buildTree(nums, node * 2 + 1, start, mid);

        // 递归构建右子树
        buildTree(nums, node * 2 + 2, mid + 1, end);
```

```
    // 合并左右子树的结果（这里是求和）
    segmentTree[node] = segmentTree[node * 2 + 1] + segmentTree[node * 2 + 2];
}

/**
 * 更新数组中指定位置的值
 * @param index 要更新的位置
 * @param val 新的值
 */
public void update(int index, int val) {
    // 检查索引有效性
    if (index < 0 || index >= n) {
        throw new IllegalArgumentException("Index out of bounds");
    }

    // 调用内部更新方法
    update(0, 0, n - 1, index, val);
}

/**
 * 线段树内部更新方法
 * @param node 当前节点在线段树数组中的索引
 * @param start 当前节点表示区间的起始位置
 * @param end 当前节点表示区间的结束位置
 * @param index 要更新的位置
 * @param val 新的值
 */
private void update(int node, int start, int end, int index, int val) {
    // 递归终止条件：到达叶子节点
    if (start == end) {
        segmentTree[node] = val;
        return;
    }

    // 计算中点
    int mid = start + (end - start) / 2;

    // 根据索引位置决定更新左子树还是右子树
    if (index <= mid) {
        update(node * 2 + 1, start, mid, index, val);
    } else {
        update(node * 2 + 2, mid + 1, end, index, val);
    }
```

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

/**
 * 查询指定区间的元素和
 * @param left 区间左边界（包含）
 * @param right 区间右边界（包含）
 * @return 区间元素和
 */
public int sumRange(int left, int right) {
    // 检查参数有效性
    if (left < 0 || right >= n || left > right) {
        throw new IllegalArgumentException("Invalid range");
    }

    // 调用内部查询方法
    return query(0, 0, n - 1, left, right);
}

/**
 * 线段树内部查询方法
 * @param node 当前节点在线段树数组中的索引
 * @param start 当前节点表示区间的起始位置
 * @param end 当前节点表示区间的结束位置
 * @param left 查询区间左边界（包含）
 * @param right 查询区间右边界（包含）
 * @return 区间元素和
 */
private int query(int node, int start, int end, int left, int right) {
    // 当前区间与查询区间无交集
    if (right < start || left > end) {
        return 0;
    }

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

    // 计算中点
    int mid = start + (end - start) / 2;
```

```
            // 递归查询左右子树，并合并结果
            int leftSum = query(node * 2 + 1, start, mid, left, right);
            int rightSum = query(node * 2 + 2, mid + 1, end, left, right);

            return leftSum + rightSum;
        }
    }

    // 测试方法
    public static void main(String[] args) {
        // 测试用例 1
        int[] nums1 = {1, 3, 5};
        NumArray numArray1 = new NumArray(nums1);

        // 测试 sumRange [0, 2] 应该返回 9
        System.out.println("Sum from index 0 to 2: " + numArray1.sumRange(0, 2)); // 期望输出: 9

        // 测试 update 将索引 1 的值更新为 2
        numArray1.update(1, 2);

        // 测试 sumRange [0, 2] 应该返回 8
        System.out.println("Sum from index 0 to 2 after update: " + numArray1.sumRange(0, 2)); //
期望输出: 8

        // 测试用例 2
        int[] nums2 = {9, -8};
        NumArray numArray2 = new NumArray(nums2);

        // 测试 update 将索引 1 的值更新为 3
        numArray2.update(1, 3);

        // 测试 sumRange [1, 1] 应该返回 3
        System.out.println("Sum from index 1 to 1: " + numArray2.sumRange(1, 1)); // 期望输出: 3

        // 测试 update 将索引 1 的值更新为-3
        numArray2.update(1, -3);

        // 测试 sumRange [0, 1] 应该返回 6
        System.out.println("Sum from index 0 to 1: " + numArray2.sumRange(0, 1)); // 期望输出: 6
    }
}
```

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

文件: Code07_CountSmallerNumbersAfterSelf.java
==========================================
package class132;

import java.util.*;

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

public class Code07_CountSmallerNumbersAfterSelf {

    /**
     * 使用树状数组解决计算右侧小于当前元素的个数问题
     *
     * 解题思路:
     * 1. 由于元素值范围可能很大，需要进行离散化处理
     * 2. 从右向左遍历数组，对于每个元素，在树状数组中查询比它小的元素个数
     * 3. 然后将当前元素加入树状数组
     * 4. 这样可以保证查询的都是右侧已经遍历过的元素
     *
     * 时间复杂度分析:
     * - 离散化: O(n log n)
     * - 遍历数组并查询更新: O(n log n)
     * - 总时间复杂度: O(n log n)
     *
     * 空间复杂度分析:
     * - 树状数组: O(n)
     * - 离散化数组: O(n)
     * - 总空间复杂度: O(n)
     *
     * 工程化考量:
     * 1. 离散化处理大数值范围
     * 2. 边界条件处理
     * 3. 异常输入检查
     * 4. 详细注释和变量命名
     */

    // 树状数组类
    static class FenwickTree {
```

```
    private int[] tree;
    private int n;

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

    private int lowbit(int x) {
        return x & (-x);
    }

    // 在位置 i 上增加值 delta
    public void add(int i, int delta) {
        while (i <= n) {
            tree[i] += delta;
            i += lowbit(i);
        }
    }

    // 查询[1, i]的前缀和
    public int query(int i) {
        int sum = 0;
        while (i > 0) {
            sum += tree[i];
            i -= lowbit(i);
        }
        return sum;
    }
}

public static List<Integer> countSmaller(int[] nums) {
    int n = nums.length;
    List<Integer> result = new ArrayList<>(Collections.nCopies(n, 0));

    // 离散化处理
    // 1. 获取所有不重复的元素并排序
    Set<Integer> uniqueNumbers = new HashSet<>();
    for (int num : nums) {
        uniqueNumbers.add(num);
    }

    List<Integer> sortedNumbers = new ArrayList<>(uniqueNumbers);
```

```
    Collections.sort(sortedNumbers);

    // 2. 建立数值到离散化索引的映射
    Map<Integer, Integer> indexMap = new HashMap<>();
    for (int i = 0; i < sortedNumbers.size(); i++) {
        indexMap.put(sortedNumbers.get(i), i + 1);
    }

    // 3. 创建树状数组，大小为离散化后的元素个数
    FenwickTree fenwickTree = new FenwickTree(sortedNumbers.size());

    // 4. 从右向左遍历数组
    for (int i = n - 1; i >= 0; i--) {
        // 获取当前元素的离散化索引
        int index = indexMap.get(nums[i]);

        // 查询比当前元素小的元素个数
        // 即查询[1, index-1]范围内已存在的元素个数
        result.set(i, fenwickTree.query(index - 1));

        // 将当前元素加入树状数组
        fenwickTree.add(index, 1);
    }

    return result;
}

// 测试方法
public static void main(String[] args) {
    // 测试用例 1
    int[] nums1 = {5, 2, 6, 1};
    List<Integer> result1 = countSmaller(nums1);
    System.out.println("Input: [5, 2, 6, 1]");
    System.out.println("Output: " + result1); // 期望输出: [2, 1, 1, 0]

    // 测试用例 2
    int[] nums2 = {-1};
    List<Integer> result2 = countSmaller(nums2);
    System.out.println("Input: [-1]");
    System.out.println("Output: " + result2); // 期望输出: [0]

    // 测试用例 3
    int[] nums3 = {-1, -1};
```

```
        List<Integer> result3 = countSmaller(nums3);
        System.out.println("Input: [-1, -1]");
        System.out.println("Output: " + result3); // 期望输出: [0, 0]

        // 测试用例 4
        int[] nums4 = {26, 78, 27, 100, 33, 67, 90, 23, 66, 5, 38, 7, 35, 23, 52, 22, 83, 51, 98,
69, 81, 32, 78, 28, 94, 13, 2, 97, 3, 76, 99, 51, 9, 21, 84, 66, 65, 36, 100, 41};
        List<Integer> result4 = countSmaller(nums4);
        System.out.println("Large input test case result size: " + result4.size());
    }
}
```

==========

文件: Code07_CountSmallerNumbersAfterSelf.py

==========

```
# LeetCode 315. 计算右侧小于当前元素的个数
# 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。
# 测试链接: https://leetcode.cn/problems/count-of-smaller-numbers-after-self/

from typing import List

class FenwickTree:
    """
    树状数组（Fenwick Tree）实现
    """
    def __init__(self, size: int):
        self.n = size
        self.tree = [0] * (size + 1)

    def lowbit(self, x: int) -> int:
        """
        获取 x 的二进制表示中最右边的 1 所代表的值
        """
        return x & (-x)

    def add(self, i: int, delta: int) -> None:
        """
        在位置 i 上增加值 delta
        """
        while i <= self.n:
            self.tree[i] += delta
```

```
            i += self.lowbit(i)

    def query(self, i: int) -> int:
        """
        查询[1, i]的前缀和
        """
        sum_val = 0
        while i > 0:
            sum_val += self.tree[i]
            i -= self.lowbit(i)
        return sum_val


def countSmaller(nums: List[int]) -> List[int]:
    """
    使用树状数组解决计算右侧小于当前元素的个数问题

    解题思路:
    1. 由于元素值范围可能很大，需要进行离散化处理
    2. 从右向左遍历数组，对于每个元素，在树状数组中查询比它小的元素个数
    3. 然后将当前元素加入树状数组
    4. 这样可以保证查询的都是右侧已经遍历过的元素

    时间复杂度分析:
    - 离散化: O(n log n)
    - 遍历数组并查询更新: O(n log n)
    - 总时间复杂度: O(n log n)

    空间复杂度分析:
    - 树状数组: O(n)
    - 离散化数组: O(n)
    - 总空间复杂度: O(n)

    工程化考量:
    1. 离散化处理大数值范围
    2. 边界条件处理
    3. 异常输入检查
    4. 详细注释和变量命名
    """
    n = len(nums)
    result = [0] * n

    # 离散化处理
```

```
    # 1. 获取所有不重复的元素并排序
    sorted_nums = sorted(set(nums))

    # 2. 建立数值到离散化索引的映射
    index_map = {num: i + 1 for i, num in enumerate(sorted_nums)}

    # 3. 创建树状数组，大小为离散化后的元素个数
    fenwick_tree = FenwickTree(len(sorted_nums))

    # 4. 从右向左遍历数组
    for i in range(n - 1, -1, -1):
        # 获取当前元素的离散化索引
        index = index_map[nums[i]]

        # 查询比当前元素小的元素个数
        # 即查询[1, index-1]范围内已存在的元素个数
        result[i] = fenwick_tree.query(index - 1)

        # 将当前元素加入树状数组
        fenwick_tree.add(index, 1)

    return result


# 测试函数
def test():
    # 测试用例 1
    nums1 = [5, 2, 6, 1]
    result1 = countSmaller(nums1)
    print(f"Input: [5, 2, 6, 1]")
    print(f"Output: {result1}")  # 期望输出: [2, 1, 1, 0]

    # 测试用例 2
    nums2 = [-1]
    result2 = countSmaller(nums2)
    print(f"Input: [-1]")
    print(f"Output: {result2}")  # 期望输出: [0]

    # 测试用例 3
    nums3 = [-1, -1]
    result3 = countSmaller(nums3)
    print(f"Input: [-1, -1]")
    print(f"Output: {result3}")  # 期望输出: [0, 0]
```

```
    # 测试用例 4
    nums4 = [26, 78, 27, 100, 33, 67, 90, 23, 66, 5, 38, 7, 35, 23, 52, 22, 83, 51, 98, 69, 81,
32, 78, 28, 94, 13, 2, 97, 3, 76, 99, 51, 9, 21, 84, 66, 65, 36, 100, 41]
    result4 = countSmaller(nums4)
    print(f"Large input test case result size: {len(result4)}")


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

==========

文件: Code08_TheSkylineProblem.java

==========

```
package class132;

import java.util.*;

// LeetCode 218. 天际线问题
// 城市的天际线是从远处观看该城市中所有建筑物形成的轮廓的外部轮廓。
// 给你所有建筑物的位置和高度，请返回由这些建筑物形成的天际线。
// 每个建筑物的几何信息由数组 buildings 表示，其中三元组 buildings[i] = [lefti, righti, heighti]
表示：
// lefti 是第 i 座建筑物左边缘的 x 坐标。
// righti 是第 i 座建筑物右边缘的 x 坐标。
// heighti 是第 i 座建筑物的高度。
// 你可以假设所有的建筑都是完美的长方形，在高度为 0 的绝对平坦的表面上。
// 天际线应该表示为由“关键点”组成的列表，格式 [[x1,y1],[x2,y2],...] ，并按 x 坐标进行排序。
// 关键点是水平线段的左端点。列表中最后一个点是最右侧建筑物的终点，y 坐标始终为 0，仅用于标记天际
线的终点。
// 此外，任何地面或建筑物上都不应有零长度的线段。
// 测试链接: https://leetcode.cn/problems/the-skyline-problem/

public class Code08_TheSkylineProblem {

    /**
     * 使用线段树解决天际线问题
     *
     * 解题思路:
     * 1. 将建筑物的左右边界作为事件处理，使用扫描线算法
     * 2. 对所有 x 坐标进行离散化处理
     * 3. 使用线段树维护区间最大值，表示该区间的最大高度
```

```
 * 4. 处理每个事件点，更新线段树并查询当前关键点
 *
 * 时间复杂度分析:
 * - 离散化: O(n log n)
 * - 处理事件: O(n log n)
 * - 总时间复杂度: O(n log n)
 *
 * 空间复杂度分析:
 * - 线段树: O(n)
 * - 离散化数组: O(n)
 * - 事件列表: O(n)
 * - 总空间复杂度: O(n)
 *
 * 工程化考量:
 * 1. 离散化处理大坐标范围
 * 2. 扫描线算法优化
 * 3. 线段树区间更新和查询
 * 4. 边界条件处理
 * 5. 详细注释和变量命名
 */

// 线段树节点
static class SegmentTreeNode {
    int start, end;
    int maxHeight;
    SegmentTreeNode left, right;

    SegmentTreeNode(int start, int end) {
        this.start = start;
        this.end = end;
        this.maxHeight = 0;
    }
}

// 线段树类（支持区间更新）
static class SegmentTree {
    private SegmentTreeNode root;

    SegmentTree(int size) {
        root = new SegmentTreeNode(0, size);
    }

    // 区间更新：在区间[updateStart, updateEnd]增加高度 height
```

```
public void update(int updateStart, int updateEnd, int height) {
    update(root, updateStart, updateEnd, height);
}

private void update(SegmentTreeNode node, int updateStart, int updateEnd, int height) {
    // 如果当前节点区间与更新区间无交集
    if (node.start > updateEnd || node.end < updateStart) {
        return;
    }

    // 如果当前节点是叶子节点
    if (node.start == node.end) {
        node.maxHeight = Math.max(node.maxHeight, height);
        return;
    }

    // 如果当前节点区间完全包含在更新区间内
    if (updateStart <= node.start && node.end <= updateEnd) {
        // 这里我们采用懒惰更新的方式，直接更新当前节点
        node.maxHeight = Math.max(node.maxHeight, height);
        return;
    }

    int mid = node.start + (node.end - node.start) / 2;

    // 创建子节点（如果不存在）
    if (node.left == null) {
        node.left = new SegmentTreeNode(node.start, mid);
    }
    if (node.right == null) {
        node.right = new SegmentTreeNode(mid + 1, node.end);
    }

    // 递归更新子节点
    update(node.left, updateStart, updateEnd, height);
    update(node.right, updateStart, updateEnd, height);
}

// 查询点 index 的高度
public int query(int index) {
    return query(root, index);
}
```

```
    private int query(SegmentTreeNode node, int index) {
        // 如果节点为空或索引超出范围
        if (node == null || index < node.start || index > node.end) {
            return 0;
        }

        // 如果是叶子节点
        if (node.start == node.end) {
            return node.maxHeight;
        }

        // 向下传递懒惰标记（如果有的话）
        int mid = node.start + (node.end - node.start) / 2;

        // 递归查询并结合当前节点的高度
        if (index <= mid) {
            return Math.max(node.maxHeight, query(node.left, index));
        } else {
            return Math.max(node.maxHeight, query(node.right, index));
        }
    }
}

public static List<List<Integer>> getSkyline(int[][] buildings) {
    List<List<Integer>> result = new ArrayList<>();

    // 1. 收集所有关键点（建筑物的左右边界）
    List<int[]> events = new ArrayList<>(); // {x, height, type} type: 1 表示进入，-1 表示离开

    for (int[] building : buildings) {
        int left = building[0];
        int right = building[1];
        int height = building[2];

        // 进入事件（使用负高度表示进入）
        events.add(new int[]{left, -height, 1});
        // 离开事件（使用正高度表示离开）
        events.add(new int[]{right, height, -1});
    }

    // 2. 按照 x 坐标排序
    // 如果 x 坐标相同，进入事件优先于离开事件，同样类型的事件按高度排序
    events.sort((a, b) -> {
```

```
    if (a[0] != b[0]) {
        return a[0] - b[0]; // 按 x 坐标升序
    }
    return a[1] - b[1]; // 按高度排序（负高度在前）
});

// 3. 离散化 x 坐标
TreeSet<Integer> xCoords = new TreeSet<>();
for (int[] event : events) {
    xCoords.add(event[0]);
}

List<Integer> sortedX = new ArrayList<>(xCoords);
Map<Integer, Integer> xIndexMap = new HashMap<>();
for (int i = 0; i < sortedX.size(); i++) {
    xIndexMap.put(sortedX.get(i), i);
}

// 4. 创建线段树
SegmentTree segmentTree = new SegmentTree(sortedX.size());

// 5. 处理事件
int prevHeight = 0;
for (int[] event : events) {
    int x = event[0];
    int height = Math.abs(event[1]);
    int type = event[1] < 0 ? 1 : -1; // 负数表示进入，正数表示离开

    if (type == 1) {
        // 建筑物进入，更新线段树
        int endIndex = xIndexMap.get(x);
        // 找到右边界在离散化数组中的索引
        int rightX = findRightBoundary(buildings, x, height);
        int rightIndex = xIndexMap.get(rightX);

        segmentTree.update(endIndex, rightIndex - 1, height);
    }
    // 离开事件已经在进入事件中处理了区间更新

    // 查询当前点的高度
    int currentIndex = xIndexMap.get(x);
    int currentHeight = segmentTree.query(currentIndex);
```

```
            // 如果高度发生变化，添加关键点
            if (currentHeight != prevHeight) {
                result.add(Arrays.asList(x, currentHeight));
                prevHeight = currentHeight;
            }
        }

        return result;
    }

    // 辅助方法：找到指定建筑物的右边界
    private static int findRightBoundary(int[][] buildings, int left, int height) {
        for (int[] building : buildings) {
            if (building[0] == left && building[2] == height) {
                return building[1];
            }
        }
        return -1; // 不应该到达这里
    }

    // 更优化的解决方案
    public static List<List<Integer>> getSkylineOptimized(int[][] buildings) {
        List<List<Integer>> result = new ArrayList<>();

        // 创建事件列表
        List<int[]> events = new ArrayList<>();

        // 收集所有事件点
        for (int[] building : buildings) {
            int left = building[0];
            int right = building[1];
            int height = building[2];

            events.add(new int[]{left, -height});  // 起始事件，用负数表示
            events.add(new int[]{right, height});  // 结束事件
        }

        // 排序事件
        events.sort((a, b) -> {
            if (a[0] != b[0]) {
                return a[0] - b[0];
            }
            return a[1] - b[1];
```

```
        });

        // 使用 TreeMap 维护当前活跃建筑物的高度（自动排序）
        TreeMap<Integer, Integer> heightMap = new TreeMap<>(Collections.reverseOrder());
        heightMap.put(0, 1); // 地面高度

        int prevHeight = 0;

        for (int[] event : events) {
            int x = event[0];
            int h = event[1];

            if (h < 0) {
                // 起始事件，添加建筑物
                heightMap.put(-h, heightMap.getOrDefault(-h, 0) + 1);
            } else {
                // 结束事件，移除建筑物
                heightMap.put(h, heightMap.get(h) - 1);
                if (heightMap.get(h) == 0) {
                    heightMap.remove(h);
                }
            }

            // 获取当前最大高度
            int currentHeight = heightMap.firstKey();

            // 如果高度发生变化，添加关键点
            if (currentHeight != prevHeight) {
                result.add(Arrays.asList(x, currentHeight));
                prevHeight = currentHeight;
            }
        }

        return result;
    }

    // 测试方法
    public static void main(String[] args) {
        // 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 = getSkylineOptimized(buildings1);
        System.out.println("Test case 1:");
        System.out.println("Input: [[2,9,10],[3,7,15],[5,12,12],[15,20,10],[19,24,8]]");
```

```
        System.out.println("Output: " + result1);
        // 期望输出: [[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 = getSkylineOptimized(buildings2);
        System.out.println("\nTest case 2:");
        System.out.println("Input: [[0,2,3],[2,5,3]]");
        System.out.println("Output: " + result2);
        // 期望输出: [[0,3],[5,0]]
    }
}
```

==========

文件: Code08_TheSkylineProblem.py

==========

```
# LeetCode 218. 天际线问题
# 城市的天际线是从远处观看该城市中所有建筑物形成的轮廓的外部轮廓。
# 给你所有建筑物的位置和高度，请返回由这些建筑物形成的天际线。
# 每个建筑物的几何信息由数组 buildings 表示，其中三元组 buildings[i] = [lefti, righti, heighti] 表示：
# lefti 是第 i 座建筑物左边缘的 x 坐标。
# righti 是第 i 座建筑物右边缘的 x 坐标。
# heighti 是第 i 座建筑物的高度。
# 你可以假设所有的建筑都是完美的长方形，在高度为 0 的绝对平坦的表面上。
# 天际线应该表示为由“关键点”组成的列表，格式 [[x1,y1],[x2,y2],...] ，并按 x 坐标进行排序。
# 关键点是水平线段的左端点。列表中最后一个点是最右侧建筑物的终点，y 坐标始终为 0，仅用于标记天际线的终点。
# 此外，任何地面或建筑物上都不应有零长度的线段。
# 测试链接: https://leetcode.cn/problems/the-skyline-problem/

from typing import List
import heapq
from collections import defaultdict

def getSkyline(buildings: List[List[int]]) -> List[List[int]]:
    """
    使用扫描线算法解决天际线问题

    解题思路:
    1. 将建筑物的左右边界作为事件处理，使用扫描线算法
    2. 对所有事件点进行排序处理
```

3. 使用有序数据结构维护当前活跃建筑物的高度
4. 处理每个事件点，更新高度并确定关键点

时间复杂度分析:
- 事件排序: O(n log n)
- 处理事件: O(n log n)
- 总时间复杂度: O(n log n)

空间复杂度分析:
- 事件列表: O(n)
- 高度维护结构: O(n)
- 结果列表: O(n)
- 总空间复杂度: O(n)

工程化考量:
1. 扫描线算法优化
2. 有序数据结构维护活跃高度
3. 边界条件处理
4. 详细注释和变量命名
"""

```
# 创建事件列表
events = []

# 收集所有事件点
for left, right, height in buildings:
    # 起始事件，用负数表示
    events.append((left, -height))
    # 结束事件
    events.append((right, height))

# 排序事件
# 如果 x 坐标相同，进入事件(负高度)优先于离开事件(正高度)
events.sort()

# 使用字典维护当前活跃建筑物的高度计数
# 保持高度到计数的映射
height_count = defaultdict(int)
height_count[0] = 1  # 初始地面高度
result = []
prev_height = 0

for x, h in events:
    if h < 0:
```

```
            # 起始事件，添加建筑物高度
            height_count[-h] += 1
        else:
            # 结束事件，移除建筑物高度
            height_count[h] -= 1
            if height_count[h] == 0:
                del height_count[h]

        # 获取当前最大高度
        current_height = max(height_count.keys()) if height_count else 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 getSkylineHeap(buildings: List[List[int]]) -> List[List[int]]:
    """
    使用堆解决天际线问题的另一种实现

    解题思路:
    1. 使用最小堆维护建筑物的右边界和高度
    2. 扫描线算法处理建筑物的起始和结束
    3. 通过堆顶元素判断当前有效高度
    """
    # 创建事件点
    events = []
    for left, right, height in buildings:
        events.append((left, -height, right))  # 起始事件
        events.append((right, 0, 0))           # 结束事件占位符

    # 按 x 坐标排序
    events.sort()

    # 结果列表
    res = [[0, 0]]

    # 最小堆存储 (右边界, 高度)，使用负数实现最大堆效果
    heap = [(0, float('inf'))]  # (高度, 右边界)
```

```
    for x, neg_h, right in events:
        # 添加起始事件对应的建筑物到堆中
        if neg_h != 0:
            heapq.heappush(heap, (neg_h, right))

        # 移除堆中已经结束的建筑物
        while heap[0][1] <= x:
            heapq.heappop(heap)

        # 获取当前最大高度
        current_height = -heap[0][0]

        # 如果高度发生变化，添加关键点
        if res[-1][1] != current_height:
            res.append([x, current_height])

    return res[1:]  # 移除初始占位符


# 测试函数
def test():
    # 测试用例 1
    buildings1 = [[2, 9, 10], [3, 7, 15], [5, 12, 12], [15, 20, 10], [19, 24, 8]]
    result1 = getSkyline(buildings1)
    print("Test case 1:")
    print("Input: [[2,9,10],[3,7,15],[5,12,12],[15,20,10],[19,24,8]]")
    print("Output:", result1)
    # 期望输出: [[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 = getSkyline(buildings2)
    print("\nTest case 2:")
    print("Input: [[0,2,3],[2,5,3]]")
    print("Output:", result2)
    # 期望输出: [[0,3],[5,0]]

    # 使用堆方法测试
    result3 = getSkylineHeap(buildings1)
    print("\nTest case 1 (heap method):")
    print("Output:", result3)
```

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

==========

文件: Code09_FallingSquares.java
==========

```
package class132;

import java.util.*;

// LeetCode 699. 掉落的方块
// 在无限长的数轴（坐标轴）上，我们根据给定的顺序放置“方块”。
// 第 i 个方块的边长是 squares[i] = [left, sideLength]，其中 left 表示该方块最左边的 x 坐标，
// sideLength 表示边长。
// 每个方块从一个比目前所有落下的方块更高的高度掉落而下，直到着陆到另一个正方形的顶边或者数轴上。
// 我们可以认为只有一个方块的底部边平行于数轴。
// 返回一个数组 ans，其中 ans[i] 表示在第 i 个方块掉落后，当前所有落下的方块堆叠的最高高度。
// 测试链接: https://leetcode.cn/problems/falling-squares/

public class Code09_FallingSquares {

    /**
     * 使用线段树解决掉落的方块问题
     *
     * 解题思路:
     * 1. 这是一个动态区间最值查询问题
     * 2. 每个方块掉落时，需要查询其底部区间内的最大高度
     * 3. 然后更新该区间的高度为底部最大高度+方块高度
     * 4. 使用线段树支持区间最值查询和区间更新
     *
     * 时间复杂度分析:
     * - 离散化: O(n log n)
     * - 每次查询和更新: O(log n)
     * - 总时间复杂度: O(n log n)
     *
     * 空间复杂度分析:
     * - 线段树: O(n)
     * - 离散化数组: O(n)
     * - 总空间复杂度: O(n)
     *
     * 工程化考量:
     * 1. 离散化处理大坐标范围
```

```
 * 2. 懒惰传播优化区间更新
 * 3. 边界条件处理
 * 4. 详细注释和变量命名
 */

// 线段树节点
static class SegmentTreeNode {
    int start, end;
    int maxHeight;     // 区间最大高度
    int lazy;          // 懒惰标记
    SegmentTreeNode left, right;

    SegmentTreeNode(int start, int end) {
        this.start = start;
        this.end = end;
        this.maxHeight = 0;
        this.lazy = 0;
    }
}

// 支持懒惰传播的线段树
static class SegmentTree {
    private SegmentTreeNode root;

    SegmentTree(int size) {
        root = new SegmentTreeNode(0, size);
    }

    // 区间更新：将区间[updateStart, updateEnd]的高度设置为height
    public void update(int updateStart, int updateEnd, int height) {
        update(root, updateStart, updateEnd, height);
    }

    private void update(SegmentTreeNode node, int updateStart, int updateEnd, int height) {
        // 如果当前节点区间与更新区间无交集
        if (node.start > updateEnd || node.end < updateStart) {
            return;
        }

        // 如果当前节点区间完全包含在更新区间内
        if (updateStart <= node.start && node.end <= updateEnd) {
            node.maxHeight = height;
            node.lazy = height;
```

```
        return;
    }

    // 如果不是叶子节点，下推懒惰标记
    pushDown(node);

    // 递归更新子节点
    update(node.left, updateStart, updateEnd, height);
    update(node.right, updateStart, updateEnd, height);

    // 更新当前节点的值
    node.maxHeight = Math.max(node.left.maxHeight, node.right.maxHeight);
}

// 区间查询：查询区间[queryStart, queryEnd]的最大高度
public int query(int queryStart, int queryEnd) {
    return query(root, queryStart, queryEnd);
}

private int query(SegmentTreeNode node, int queryStart, int queryEnd) {
    // 如果当前节点区间与查询区间无交集
    if (node.start > queryEnd || node.end < queryStart) {
        return 0;
    }

    // 如果当前节点区间完全包含在查询区间内
    if (queryStart <= node.start && node.end <= queryEnd) {
        return node.maxHeight;
    }

    // 如果不是叶子节点，下推懒惰标记
    pushDown(node);

    // 递归查询子节点并返回最大值
    return Math.max(query(node.left, queryStart, queryEnd),
                    query(node.right, queryStart, queryEnd));
}

// 下推懒惰标记
private void pushDown(SegmentTreeNode node) {
    // 如果是叶子节点，无法下推
    if (node.start == node.end) {
        return;
```

```
        }

        // 创建子节点（如果不存在）
        int mid = node.start + (node.end - node.start) / 2;
        if (node.left == null) {
            node.left = new SegmentTreeNode(node.start, mid);
        }
        if (node.right == null) {
            node.right = new SegmentTreeNode(mid + 1, node.end);
        }

        // 如果有懒惰标记，下推给子节点
        if (node.lazy > 0) {
            node.left.maxHeight = node.lazy;
            node.left.lazy = node.lazy;

            node.right.maxHeight = node.lazy;
            node.right.lazy = node.lazy;

            node.lazy = 0;
        }
    }
}

public static List<Integer> fallingSquares(int[][] positions) {
    List<Integer> result = new ArrayList<>();

    // 1. 收集所有关键坐标点并离散化
    TreeSet<Integer> coords = new TreeSet<>();
    for (int[] pos : positions) {
        int left = pos[0];
        int right = pos[0] + pos[1] - 1;  // 右边界包含在方块内
        coords.add(left);
        coords.add(right);
        coords.add(left - 1);  // 添加左边界前一个点，用于查询
        coords.add(right + 1);  // 添加右边界后一个点，用于查询
    }

    // 2. 建立坐标到索引的映射
    List<Integer> sortedCoords = new ArrayList<>(coords);
    Map<Integer, Integer> indexMap = new HashMap<>();
    for (int i = 0; i < sortedCoords.size(); i++) {
        indexMap.put(sortedCoords.get(i), i);
```

```
    }

    // 3. 创建线段树
    SegmentTree segmentTree = new SegmentTree(sortedCoords.size());

    // 4. 处理每个方块
    int maxHeight = 0;
    for (int[] pos : positions) {
        int left = pos[0];
        int sideLength = pos[1];
        int right = left + sideLength - 1;

        // 获取离散化后的索引
        int leftIndex = indexMap.get(left);
        int rightIndex = indexMap.get(right);

        // 查询当前方块底部的最大高度
        int currentMaxHeight = segmentTree.query(leftIndex, rightIndex);

        // 计算方块堆叠后的高度
        int newHeight = currentMaxHeight + sideLength;

        // 更新线段树中该区间的高度
        segmentTree.update(leftIndex, rightIndex, newHeight);

        // 更新全局最大高度
        maxHeight = Math.max(maxHeight, newHeight);

        // 添加当前最大高度到结果中
        result.add(maxHeight);
    }

    return result;
}

// 优化版本：不需要离散化，使用动态开点线段树
public static List<Integer> fallingSquaresOptimized(int[][] positions) {
    List<Integer> result = new ArrayList<>();

    // 使用 TreeMap 维护坐标和高度的关系
    TreeMap<Integer, Integer> heights = new TreeMap<>();
    heights.put(0, 0);  // 初始地面高度
```

```
    int maxHeight = 0;

    for (int[] pos : positions) {
        int left = pos[0];
        int side = pos[1];
        int right = left + side;

        // 找到覆盖区域内的最大高度
        int currentHeight = 0;
        Integer start = heights.floorKey(left);
        if (start != null) {
            currentHeight = heights.get(start);
        }

        // 遍历覆盖区域内的所有点，找到最大高度
        for (Map.Entry<Integer, Integer> entry : heights.subMap(left, right).entrySet()) {
            currentHeight = Math.max(currentHeight, entry.getValue());
        }

        // 新的高度
        int newHeight = currentHeight + side;

        // 更新最大高度
        maxHeight = Math.max(maxHeight, newHeight);
        result.add(maxHeight);

        // 更新高度映射
        // 移除被覆盖的区间
        heights.subMap(left, right).clear();
        // 添加新的区间
        heights.put(left, newHeight);
        heights.put(right, currentHeight);
    }

    return result;
}

// 测试方法
public static void main(String[] args) {
    // 测试用例 1
    int[][] positions1 = {{1, 2}, {2, 3}, {6, 1}};
    List<Integer> result1 = fallingSquaresOptimized(positions1);
    System.out.println("Test case 1:");
```

```
        System.out.println("Input: [[1, 2], [2, 3], [6, 1]]");
        System.out.println("Output: " + result1);
        // 期望输出: [2, 5, 5]

        // 测试用例 2
        int[][] positions2 = {{100, 100}, {200, 100}};
        List<Integer> result2 = fallingSquaresOptimized(positions2);
        System.out.println("\nTest case 2:");
        System.out.println("Input: [[100, 100], [200, 100]]");
        System.out.println("Output: " + result2);
        // 期望输出: [100, 100]

        // 测试用例 3
        int[][] positions3 = {{1, 5}, {2, 2}, {4, 3}};
        List<Integer> result3 = fallingSquaresOptimized(positions3);
        System.out.println("\nTest case 3:");
        System.out.println("Input: [[1, 5], [2, 2], [4, 3]]");
        System.out.println("Output: " + result3);
        // 期望输出: [5, 7, 7]
    }
}
```

==========

文件: Code09_FallingSquares.py

==========

```
# LeetCode 699. 掉落的方块
# 在无限长的数轴（坐标轴）上，我们根据给定的顺序放置“方块”。
# 第 i 个方块的边长是 squares[i] = [left, sideLength]，其中 left 表示该方块最左边的 x 坐标，
# sideLength 表示边长。
# 每个方块从一个比目前所有落下的方块更高的高度掉落而下，直到着陆到另一个正方形的顶边或者数轴上。
# 我们可以认为只有一个方块的底部边平行于数轴。
# 返回一个数组 ans，其中 ans[i] 表示在第 i 个方块掉落后，当前所有落下的方块堆叠的最高高度。
# 测试链接: https://leetcode.cn/problems/falling-squares/

from typing import List

def fallingSquares(positions: List[List[int]]) -> List[int]:
    """
    使用区间列表解决掉落的方块问题

    解题思路:
    1. 维护所有已放置方块的区间信息
```

2. 对于每个新掉落的方块，检查与已有方块的重叠情况
3. 计算堆叠高度并更新最大高度

时间复杂度分析:
- 每次查询和更新: O(n)
- 总时间复杂度: O(n²)

空间复杂度分析:
- 区间列表: O(n)
- 结果数组: O(n)
- 总空间复杂度: O(n)

工程化考量:
1. 区间重叠检测
2. 边界条件处理
3. 详细注释和变量命名
"""

```
# 存储已放置的方块信息: [left, right, height]
intervals = []
result = []
max_height = 0

for left, side_length in positions:
    right = left + side_length

    # 找到底部区域内的最大高度
    current_height = 0
    for l, r, h in intervals:
        # 检查是否有重叠
        if left < r and right > l:  # 两个区间重叠的条件
            current_height = max(current_height, h)

    # 计算新的高度
    new_height = current_height + side_length

    # 添加新的方块到列表
    intervals.append([left, right, new_height])

    # 更新最大高度
    max_height = max(max_height, new_height)
    result.append(max_height)

return result
```

```
# 测试函数
def test():
    # 测试用例 1
    positions1 = [[1, 2], [2, 3], [6, 1]]
    result1 = fallingSquares(positions1)
    print("Test case 1:")
    print("Input: [[1, 2], [2, 3], [6, 1]]")
    print("Output:", result1)
    # 期望输出: [2, 5, 5]

    # 测试用例 2
    positions2 = [[100, 100], [200, 100]]
    result2 = fallingSquares(positions2)
    print("\nTest case 2:")
    print("Input: [[100, 100], [200, 100]]")
    print("Output:", result2)
    # 期望输出: [100, 100]

    # 测试用例 3
    positions3 = [[1, 5], [2, 2], [4, 3]]
    result3 = fallingSquares(positions3)
    print("\nTest case 3:")
    print("Input: [[1, 5], [2, 2], [4, 3]]")
    print("Output:", result3)
    # 期望输出: [5, 7, 7]


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

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

文件: Code10_ReversePairs.cpp

```
================================================
// LeetCode 493. 翻转对
// 给定一个数组 nums ，如果 i < j 且 nums[i] > 2*nums[j] 我们将 (i, j) 称作一个重要翻转对。
// 你需要返回给定数组中的重要翻转对的数量。
// 测试链接: https://leetcode.cn/problems/reverse-pairs/

#include <iostream>
#include <vector>
```

```cpp
#include <algorithm>
#include <unordered_set>
#include <unordered_map>
using namespace std;
using namespace std;

/**
 * 使用树状数组解决翻转对问题
 *
 * 解题思路:
 * 1. 翻转对的定义是 i < j 且 nums[i] > 2*nums[j]
 * 2. 我们可以将所有可能涉及到的数值进行离散化处理
 * 3. 从左到右遍历数组，对于每个元素 nums[i]：
 *    - 查询已经处理过的元素中，有多少个元素满足 nums[j] < nums[i]/2.0（j < i）
 *    - 将当前元素加入树状数组
 * 4. 由于涉及到 2*nums[j]，我们需要同时将 nums[j]和 2*nums[j]都加入离散化数组
 *
 * 时间复杂度分析:
 * - 离散化: O(n log n)
 * - 遍历数组并查询更新: O(n log n)
 * - 总时间复杂度: O(n log n)
 *
 * 空间复杂度分析:
 * - 树状数组: O(n)
 * - 离散化数组: O(n)
 * - 总空间复杂度: O(n)
 *
 * 工程化考量:
 * 1. 离散化处理大数值范围
 * 2. 边界条件处理（整数溢出问题）
 * 3. 异常输入检查
 * 4. 详细注释和变量命名
 */

class Solution {
private:
    // 树状数组类
    class FenwickTree {
    private:
        vector<int> tree;
        int n;

        int lowbit(int x) {
```

```
            return x & (-x);
        }

    public:
        FenwickTree(int size) {
            this->n = size;
            this->tree.resize(size + 1, 0);
        }

        // 在位置 i 上增加值 delta
        void add(int i, int delta) {
            while (i <= n) {
                tree[i] += delta;
                i += lowbit(i);
            }
        }

        // 查询[1, i]的前缀和
        int query(int i) {
            int sum = 0;
            while (i > 0) {
                sum += tree[i];
                i -= lowbit(i);
            }
            return sum;
        }
    };

public:
    int reversePairs(vector<int>& nums) {
        int n = nums.size();
        if (n == 0) return 0;

        // 离散化处理
        // 1. 收集所有需要离散化的值：nums[i] 和 2*nums[i]
        unordered_set<long long> allNumbers;
        for (int num : nums) {
            allNumbers.insert((long long)num);
            allNumbers.insert(2LL * num);
        }

        // 2. 排序去重
        vector<long long> sortedNumbers(allNumbers.begin(), allNumbers.end());
```

```
        sort(sortedNumbers.begin(), sortedNumbers.end());

        // 3. 建立数值到离散化索引的映射
        unordered_map<long long, int> indexMap;
        for (int i = 0; i < sortedNumbers.size(); i++) {
            indexMap[sortedNumbers[i]] = i + 1;
        }

        // 4. 创建树状数组
        FenwickTree fenwickTree(sortedNumbers.size());

        int result = 0;

        // 5. 从左到右遍历数组
        for (int i = 0; i < n; i++) {
            // 查询有多少个已经处理过的元素满足 nums[j] > 2*nums[i]
            // 也就是查询有多少个已经处理过的元素满足 nums[j] >= 2*nums[i]+1
            long long target = 2LL * nums[i] + 1;

            // 找到 target 在离散化数组中的位置
            auto it = lower_bound(sortedNumbers.begin(), sortedNumbers.end(), target);
            int pos = it - sortedNumbers.begin() + 1;

            // 查询大于等于 target 的元素个数
            result += fenwickTree.query(sortedNumbers.size()) - fenwickTree.query(pos - 1);

            // 将当前元素加入树状数组
            int index = indexMap[(long long)nums[i]];
            fenwickTree.add(index, 1);
        }

        return result;
    }
};

// 测试函数
#include <iostream>
int main() {
    Solution solution;

    // 测试用例 1
    vector<int> nums1 = {1, 3, 2, 3, 1};
    int result1 = solution.reversePairs(nums1);
```

```
    cout << "Input: [1, 3, 2, 3, 1]" << endl;
    cout << "Output: " << result1 << endl; // 期望输出: 2

    // 测试用例 2
    vector<int> nums2 = {2, 4, 3, 5, 1};
    int result2 = solution.reversePairs(nums2);
    cout << "Input: [2, 4, 3, 5, 1]" << endl;
    cout << "Output: " << result2 << endl; // 期望输出: 3

    // 测试用例 3
    vector<int> nums3 = {-5, -5};
    int result3 = solution.reversePairs(nums3);
    cout << "Input: [-5, -5]" << endl;
    cout << "Output: " << result3 << endl; // 期望输出: 1

    // 测试用例 4
    vector<int> nums4 = {2147483647, 2147483647, 2147483647, 2147483647, 2147483647, 2147483647};
    int result4 = solution.reversePairs(nums4);
    cout << "Large numbers test case result: " << result4 << endl; // 期望输出: 0

    return 0;
}

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

文件: Code10_ReversePairs.java
==============================================
import java.util.*;

// LeetCode 493. 翻转对
// 给定一个数组 nums ，如果 i < j 且 nums[i] > 2*nums[j] 我们将 (i, j) 称作一个重要翻转对。
// 你需要返回给定数组中的重要翻转对的数量。
// 测试链接: https://leetcode.cn/problems/reverse-pairs/

public class Code10_ReversePairs {

    /**
     * 使用树状数组解决翻转对问题
     *
     * 解题思路:
     * 1. 翻转对的定义是 i < j 且 nums[i] > 2*nums[j]
     * 2. 我们可以将所有可能涉及到的数值进行离散化处理
     * 3. 从左到右遍历数组，对于每个元素 nums[i]：
```

```
 *     - 查询已经处理过的元素中，有多少个元素满足 nums[j] < nums[i]/2.0（j < i）
 *     - 将当前元素加入树状数组
 * 4. 由于涉及到 2*nums[j]，我们需要同时将 nums[j]和 2*nums[j]都加入离散化数组
 *
 * 时间复杂度分析:
 * - 离散化: O(n log n)
 * - 遍历数组并查询更新: O(n log n)
 * - 总时间复杂度: O(n log n)
 *
 * 空间复杂度分析:
 * - 树状数组: O(n)
 * - 离散化数组: O(n)
 * - 总空间复杂度: O(n)
 *
 * 工程化考量:
 * 1. 离散化处理大数值范围
 * 2. 边界条件处理（整数溢出问题）
 * 3. 异常输入检查
 * 4. 详细注释和变量命名
 */

// 树状数组类
static class FenwickTree {
    private int[] tree;
    private int n;

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

    private int lowbit(int x) {
        return x & (-x);
    }

    // 在位置 i 上增加值 delta
    public void add(int i, int delta) {
        while (i <= n) {
            tree[i] += delta;
            i += lowbit(i);
        }
    }
```

```
    // 查询[1, i]的前缀和
    public int query(int i) {
        int sum = 0;
        while (i > 0) {
            sum += tree[i];
            i -= lowbit(i);
        }
        return sum;
    }
}

public static int reversePairs(int[] nums) {
    int n = nums.length;
    if (n == 0) return 0;

    // 离散化处理
    // 1. 收集所有需要离散化的值：nums[i] 和 2*nums[i]
    Set<Long> allNumbers = new HashSet<>();
    for (int num : nums) {
        allNumbers.add((long) num);
        allNumbers.add((long) 2 * num);
    }

    // 2. 排序去重
    List<Long> sortedNumbers = new ArrayList<>(allNumbers);
    Collections.sort(sortedNumbers);

    // 3. 建立数值到离散化索引的映射
    Map<Long, Integer> indexMap = new HashMap<>();
    for (int i = 0; i < sortedNumbers.size(); i++) {
        indexMap.put(sortedNumbers.get(i), i + 1);
    }

    // 4. 创建树状数组
    FenwickTree fenwickTree = new FenwickTree(sortedNumbers.size());

    int result = 0;

    // 5. 从左到右遍历数组
    for (int i = 0; i < n; i++) {
        // 查询有多少个已经处理过的元素满足 nums[j] > 2*nums[i]
        // 也就是查询有多少个已经处理过的元素满足 nums[j] >= 2*nums[i]+1
        long target = 2L * nums[i] + 1;
```

```
            // 找到 target 在离散化数组中的位置
            int pos = Collections.binarySearch(sortedNumbers, target);
            if (pos < 0) {
                // 如果没找到，binarySearch 返回的是插入位置的负值减一
                pos = -pos - 1;
            }

            // 查询大于等于 target 的元素个数
            result += fenwickTree.query(sortedNumbers.size()) - fenwickTree.query(pos);

            // 将当前元素加入树状数组
            int index = indexMap.get((long) nums[i]);
            fenwickTree.add(index, 1);
        }

        return result;
    }

    // 测试方法
    public static void main(String[] args) {
        // 测试用例 1
        int[] nums1 = {1, 3, 2, 3, 1};
        int result1 = reversePairs(nums1);
        System.out.println("Input: [1, 3, 2, 3, 1]");
        System.out.println("Output: " + result1); // 期望输出: 2

        // 测试用例 2
        int[] nums2 = {2, 4, 3, 5, 1};
        int result2 = reversePairs(nums2);
        System.out.println("Input: [2, 4, 3, 5, 1]");
        System.out.println("Output: " + result2); // 期望输出: 3

        // 测试用例 3
        int[] nums3 = {-5, -5};
        int result3 = reversePairs(nums3);
        System.out.println("Input: [-5, -5]");
        System.out.println("Output: " + result3); // 期望输出: 1

        // 测试用例 4
        int[] nums4 = {2147483647, 2147483647, 2147483647, 2147483647, 2147483647, 2147483647};
        int result4 = reversePairs(nums4);
        System.out.println("Large numbers test case result: " + result4); // 期望输出: 0
```

```
    }
}
```

==========

文件: Code10_ReversePairs.py
==========

```
# LeetCode 493. 翻转对
# 给定一个数组 nums ，如果 i < j 且 nums[i] > 2*nums[j] 我们将 (i, j) 称作一个重要翻转对。
# 你需要返回给定数组中的重要翻转对的数量。
# 测试链接: https://leetcode.cn/problems/reverse-pairs/

from typing import List
import bisect

class Solution:
    """
    使用树状数组解决翻转对问题

    解题思路:
    1. 翻转对的定义是 i < j 且 nums[i] > 2*nums[j]
    2. 我们可以将所有可能涉及到的数值进行离散化处理
    3. 从左到右遍历数组，对于每个元素 nums[i]：
       - 查询已经处理过的元素中，有多少个元素满足 nums[j] < nums[i]/2.0（j < i）
       - 将当前元素加入树状数组
    4. 由于涉及到 2*nums[j]，我们需要同时将 nums[j]和 2*nums[j]都加入离散化数组

    时间复杂度分析:
    - 离散化: O(n log n)
    - 遍历数组并查询更新: O(n log n)
    - 总时间复杂度: O(n log n)

    空间复杂度分析:
    - 树状数组: O(n)
    - 离散化数组: O(n)
    - 总空间复杂度: O(n)

    工程化考量:
    1. 离散化处理大数值范围
    2. 边界条件处理（整数溢出问题）
    3. 异常输入检查
    4. 详细注释和变量命名
    """
```

```
class FenwickTree:
    def __init__(self, size: int):
        """
        初始化树状数组
        :param size: 数组大小
        """
        self.n = size
        self.tree = [0] * (size + 1)

    def lowbit(self, x: int) -> int:
        """
        lowbit 操作，获取 x 的二进制表示中最右边的 1 所代表的值
        :param x: 输入整数
        :return: x & (-x)
        """
        return x & (-x)

    def add(self, index: int, delta: int) -> None:
        """
        在树状数组中更新指定位置的值（增加 delta）
        :param index: 要更新的位置（从 1 开始）
        :param delta: 增加的值
        """
        # 沿路径向上更新所有相关节点
        while index <= self.n:
            self.tree[index] += delta
            index += self.lowbit(index)

    def query(self, index: int) -> int:
        """
        查询前缀和[1, index]的和
        :param index: 查询的右边界（从 1 开始）
        :return: 前缀和
        """
        sum_val = 0
        # 沿路径向下累加所有相关节点的值
        while index > 0:
            sum_val += self.tree[index]
            index -= self.lowbit(index)
        return sum_val

def reversePairs(self, nums: List[int]) -> int:
```

```
    """
    计算数组中翻转对的数量
    :param nums: 输入数组
    :return: 翻转对的数量
    """
    n = len(nums)
    if n == 0:
        return 0

    # 离散化处理
    # 1. 收集所有需要离散化的值：nums[i] 和 2*nums[i]
    all_numbers = set()
    for num in nums:
        all_numbers.add(num)
        all_numbers.add(2 * num)

    # 2. 排序去重
    sorted_numbers = sorted(list(all_numbers))

    # 3. 建立数值到离散化索引的映射
    index_map = {}
    for i, num in enumerate(sorted_numbers):
        index_map[num] = i + 1

    # 4. 创建树状数组
    fenwick_tree = self.FenwickTree(len(sorted_numbers))

    result = 0

    # 5. 从左到右遍历数组
    for i in range(n):
        # 查询有多少个已经处理过的元素满足 nums[j] > 2*nums[i]
        # 也就是查询有多少个已经处理过的元素满足 nums[j] >= 2*nums[i]+1
        target = 2 * nums[i] + 1

        # 找到 target 在离散化数组中的位置
        pos = bisect.bisect_left(sorted_numbers, target)

        # 查询大于等于 target 的元素个数
        result += fenwick_tree.query(len(sorted_numbers)) - fenwick_tree.query(pos)

        # 将当前元素加入树状数组
        index = index_map[nums[i]]
```

```
            fenwick_tree.add(index, 1)

        return result


# 测试函数
def test():
    solution = Solution()

    # 测试用例 1
    nums1 = [1, 3, 2, 3, 1]
    result1 = solution.reversePairs(nums1)
    print(f"Input: [1, 3, 2, 3, 1]")
    print(f"Output: {result1}")  # 期望输出: 2

    # 测试用例 2
    nums2 = [2, 4, 3, 5, 1]
    result2 = solution.reversePairs(nums2)
    print(f"Input: [2, 4, 3, 5, 1]")
    print(f"Output: {result2}")  # 期望输出: 3

    # 测试用例 3
    nums3 = [-5, -5]
    result3 = solution.reversePairs(nums3)
    print(f"Input: [-5, -5]")
    print(f"Output: {result3}")  # 期望输出: 1

    # 测试用例 4
    nums4 = [2147483647, 2147483647, 2147483647, 2147483647, 2147483647, 2147483647]
    result4 = solution.reversePairs(nums4)
    print(f"Large numbers test case result: {result4}")  # 期望输出: 0


if __name__ == "__main__":
    test()

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

文件: Code11_DQuery.java
==========================================
package class132;

import java.util.*;
```

```
// SPOJ DQUERY - D-query
// 给定一个长度为n的数组，每次查询一个区间[l,r]内不同元素的个数
// 测试链接: https://www.spoj.com/problems/DQUERY/

public class Code11_DQuery {

    /**
     * 使用树状数组解决DQUERY问题
     *
     * 解题思路:
     * 1. 这是一个经典的区间不同元素个数查询问题
     * 2. 可以使用离线处理+树状数组来解决
     * 3. 首先将所有查询按照右端点排序
     * 4. 从左到右扫描数组，维护每个元素最后出现的位置
     * 5. 当处理到位置i时，如果元素a[i]之前出现过，就将之前位置的贡献删除，
     *    然后在当前位置添加贡献
     * 6. 使用树状数组维护前缀和，支持单点更新和前缀和查询
     *
     * 时间复杂度分析:
     * - 离散化: O(n log n)
     * - 排序查询: O(q log q)
     * - 处理数组: O(n log n)
     * - 处理查询: O(q log n)
     * - 总时间复杂度: O((n+q) log n + q log q)
     *
     * 空间复杂度分析:
     * - 树状数组: O(n)
     * - 查询存储: O(q)
     * - 位置记录: O(n)
     * - 总空间复杂度: O(n+q)
     *
     * 工程化考量:
     * 1. 离线处理优化查询效率
     * 2. 边界条件处理
     * 3. 异常输入检查
     * 4. 详细注释和变量命名
     */

    // 树状数组类
    static class FenwickTree {
        private int[] tree;
        private int n;
```

```
    public FenwickTree(int size) {
        this.n = size;
        this.tree = new int[n + 1];
    }

    private int lowbit(int x) {
        return x & (-x);
    }

    // 在位置 i 上增加值 delta
    public void add(int i, int delta) {
        while (i <= n) {
            tree[i] += delta;
            i += lowbit(i);
        }
    }

    // 查询[1, i]的前缀和
    public int query(int i) {
        int sum = 0;
        while (i > 0) {
            sum += tree[i];
            i -= lowbit(i);
        }
        return sum;
    }
}

// 查询类
static class Query {
    int l, r, id;

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

public static int[] dquery(int[] arr, int[][] queries) {
    int n = arr.length;
    int q = queries.length;
```

```
if (n == 0 || q == 0) {
    return new int[q];
}

// 离散化数组元素（如果需要的话）
// 对于这个问题，我们假设元素值在合理范围内

// 将查询存储到Query对象中并按右端点排序
Query[] queryList = new Query[q];
for (int i = 0; i < q; i++) {
    // 转换为1-indexed
    queryList[i] = new Query(queries[i][0], queries[i][1], i);
}

// 按右端点排序
Arrays.sort(queryList, (a, b) -> Integer.compare(a.r, b.r));

// 记录每个元素最后出现的位置
Map<Integer, Integer> lastPosition = new HashMap<>();

// 创建树状数组
FenwickTree fenwickTree = new FenwickTree(n);

// 结果数组
int[] result = new int[q];

// 当前处理到的位置
int currentPos = 0;

// 处理每个查询
for (Query query : queryList) {
    // 将数组处理到查询的右端点
    while (currentPos < query.r) {
        int pos = currentPos + 1; // 1-indexed position
        int value = arr[currentPos];

        // 如果这个元素之前出现过，需要删除之前的贡献
        if (lastPosition.containsKey(value)) {
            int prevPos = lastPosition.get(value);
            fenwickTree.add(prevPos, -1);
        }
```

```
                // 在当前位置添加贡献
                fenwickTree.add(pos, 1);

                // 更新最后出现位置
                lastPosition.put(value, pos);

                currentPos++;
            }

            // 查询区间[l,r]内不同元素个数
            // 这等于查询[1,r]的前缀和减去[1,l-1]的前缀和
            result[query.id] = fenwickTree.query(query.r) - fenwickTree.query(query.l - 1);
        }

        return result;
    }

    // 测试方法
    public static void main(String[] args) {
        // 测试用例 1
        int[] arr1 = {1, 1, 2, 1, 3};
        int[][] queries1 = {{1, 5}, {2, 4}, {3, 5}};
        int[] result1 = dquery(arr1, queries1);

        System.out.println("Array: [1, 1, 2, 1, 3]");
        System.out.println("Queries:");
        for (int i = 0; i < queries1.length; i++) {
            System.out.printf("Query [%d, %d]: %d\n", queries1[i][0], queries1[i][1],
result1[i]);
        }
        // 期望输出: 3, 2, 3

        // 测试用例 2
        int[] arr2 = {1, 2, 3, 4, 5};
        int[][] queries2 = {{1, 5}, {2, 3}, {1, 3}};
        int[] result2 = dquery(arr2, queries2);

        System.out.println("\nArray: [1, 2, 3, 4, 5]");
        System.out.println("Queries:");
        for (int i = 0; i < queries2.length; i++) {
            System.out.printf("Query [%d, %d]: %d\n", queries2[i][0], queries2[i][1],
result2[i]);
        }
```

```
        // 期望输出: 5, 2, 3
    }
}
```

==========

文件: Code11_DQuery.py

==========

```
# SPOJ DQUERY - D-query
# 给定一个长度为n的数组，每次查询一个区间[l,r]内不同元素的个数
# 测试链接: https://www.spoj.com/problems/DQUERY/

from typing import List
import bisect

class Solution:
    """
    使用树状数组解决DQUERY问题

    解题思路:
    1. 这是一个经典的区间不同元素个数查询问题
    2. 可以使用离线处理+树状数组来解决
    3. 首先将所有查询按照右端点排序
    4. 从左到右扫描数组，维护每个元素最后出现的位置
    5. 当处理到位置i时，如果元素a[i]之前出现过，就将之前位置的贡献删除，
       然后在当前位置添加贡献
    6. 使用树状数组维护前缀和，支持单点更新和前缀和查询

    时间复杂度分析:
    - 离散化: O(n log n)
    - 排序查询: O(q log q)
    - 处理数组: O(n log n)
    - 处理查询: O(q log n)
    - 总时间复杂度: O((n+q) log n + q log q)

    空间复杂度分析:
    - 树状数组: O(n)
    - 查询存储: O(q)
    - 位置记录: O(n)
    - 总空间复杂度: O(n+q)

    工程化考量:
    1. 离线处理优化查询效率
```

2. 边界条件处理
3. 异常输入检查
4. 详细注释和变量命名
"""

```
class FenwickTree:
    def __init__(self, size: int):
        """
        初始化树状数组
        :param size: 数组大小
        """
        self.n = size
        self.tree = [0] * (size + 1)

    def lowbit(self, x: int) -> int:
        """
        lowbit 操作，获取 x 的二进制表示中最右边的 1 所代表的值
        :param x: 输入整数
        :return: x & (-x)
        """
        return x & (-x)

    def add(self, index: int, delta: int) -> None:
        """
        在树状数组中更新指定位置的值（增加 delta）
        :param index: 要更新的位置（从 1 开始）
        :param delta: 增加的值
        """
        # 沿路径向上更新所有相关节点
        while index <= self.n:
            self.tree[index] += delta
            index += self.lowbit(index)

    def query(self, index: int) -> int:
        """
        查询前缀和[1, index]的和
        :param index: 查询的右边界（从 1 开始）
        :return: 前缀和
        """
        sum_val = 0
        # 沿路径向下累加所有相关节点的值
        while index > 0:
            sum_val += self.tree[index]
```

```
            index -= self.lowbit(index)
        return sum_val

def dquery(self, arr: List[int], queries: List[List[int]]) -> List[int]:
    """
    计算每个查询区间内不同元素的个数
    :param arr: 输入数组
    :param queries: 查询列表，每个查询是[l, r]的形式
    :return: 每个查询的结果列表
    """
    n = len(arr)
    q = len(queries)

    if n == 0 or q == 0:
        return [0] * q

    # 查询类
    class Query:
        def __init__(self, l: int, r: int, id: int):
            self.l = l
            self.r = r
            self.id = id

    # 将查询存储到Query对象中并按右端点排序
    query_list = []
    for i in range(q):
        # 转换为0-indexed到1-indexed
        query_list.append(Query(queries[i][0], queries[i][1], i))

    # 按右端点排序
    query_list.sort(key=lambda x: x.r)

    # 记录每个元素最后出现的位置
    last_position = {}

    # 创建树状数组
    fenwick_tree = self.FenwickTree(n)

    # 结果数组
    result = [0] * q

    # 当前处理到的位置 (0-indexed)
    current_pos = 0
```

```
        # 处理每个查询
        for query in query_list:
            # 将数组处理到查询的右端点
            while current_pos < query.r:
                pos = current_pos + 1  # 1-indexed position
                value = arr[current_pos]

                # 如果这个元素之前出现过，需要删除之前的贡献
                if value in last_position:
                    prev_pos = last_position[value]
                    fenwick_tree.add(prev_pos, -1)

                # 在当前位置添加贡献
                fenwick_tree.add(pos, 1)

                # 更新最后出现位置
                last_position[value] = pos

                current_pos += 1

            # 查询区间[l,r]内不同元素个数
            # 这等于查询[1,r]的前缀和减去[1,l-1]的前缀和
            result[query.id] = fenwick_tree.query(query.r) - fenwick_tree.query(query.l - 1)

        return result


# 测试函数
def test():
    solution = Solution()

    # 测试用例1
    arr1 = [1, 1, 2, 1, 3]
    queries1 = [[1, 5], [2, 4], [3, 5]]
    result1 = solution.dquery(arr1, queries1)

    print("Array: [1, 1, 2, 1, 3]")
    print("Queries:")
    for i in range(len(queries1)):
        print(f"Query [{queries1[i][0]}, {queries1[i][1]}]: {result1[i]}")
    # 期望输出: 3, 2, 3
```

```
    # 测试用例 2
    arr2 = [1, 2, 3, 4, 5]
    queries2 = [[1, 5], [2, 3], [1, 3]]
    result2 = solution.dquery(arr2, queries2)

    print("\nArray: [1, 2, 3, 4, 5]")
    print("Queries:")
    for i in range(len(queries2)):
        print(f"Query [{queries2[i][0]}, {queries2[i][1]}]: {result2[i]}")
    # 期望输出: 5, 2, 3


if __name__ == "__main__":
    test()

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

文件: Code12_CountSmallerNumbersAfterSelf_Enhanced.cpp
==============================================
// LeetCode 315. 计算右侧小于当前元素的个数
// 给定一个整数数组 nums，按要求返回一个新数组 counts。
// 数组 counts 有该性质：counts[i] 的值是 nums[i] 右侧小于 nums[i] 的元素的数量。
// 测试链接: https://leetcode.cn/problems/count-of-smaller-numbers-after-self/

#include <iostream>
#include <vector>
#include <algorithm>
#include <unordered_set>
#include <unordered_map>
#include <climits>
#include <chrono>
using namespace std;
using namespace std::chrono;

/**
 * 计算右侧小于当前元素的个数 - 树状数组解法
 *
 * 解题思路:
 * 1. 题目要求计算每个元素右侧小于它的元素个数
 * 2. 我们可以将问题转化为：对于每个元素 nums[i]，统计有多少个元素 nums[j] (j > i)满足 nums[j] < nums[i]
 * 3. 使用树状数组可以高效地进行这类统计
 * 4. 具体步骤：
```

```
 *     a. 将数组元素离散化，以处理可能的大范围数值
 *     b. 从右到左遍历数组，对于每个元素 nums[i]：
 *        - 查询树状数组中小于 nums[i]的元素个数（即前缀和）
 *        - 将 nums[i]加入树状数组
 *     c. 这样就能保证每次查询的都是当前元素右侧的元素
 *
 * 时间复杂度分析:
 * - 离散化: O(n log n)
 * - 构建和操作树状数组: O(n log n)
 * - 总时间复杂度: O(n log n)
 *
 * 空间复杂度分析:
 * - 树状数组: O(n)
 * - 离散化数组和映射: O(n)
 * - 结果数组: O(n)
 * - 总空间复杂度: O(n)
 *
 * 工程化考量:
 * 1. 离散化处理：由于输入数组可能包含很大范围的整数，离散化可以有效减少空间使用
 * 2. 边界条件处理：处理空数组、单元素数组等特殊情况
 * 3. 数据类型溢出：使用 long long 类型避免中间计算时的溢出
 * 4. 异常输入检查：验证输入的有效性
 * 5. 代码可读性：使用清晰的变量命名和详细的注释
 */

class Solution {
private:
    // 树状数组类
    class FenwickTree {
    private:
        vector<int> tree;  // 树状数组
        int size;          // 数组大小

    public:
        /**
         * 构造函数
         * @param size 树状数组大小
         */
        FenwickTree(int size) : size(size) {
            tree.resize(size + 1, 0);  // 树状数组下标从 1 开始
        }

        /**
```

```
     * lowbit 操作，获取 x 的二进制表示中最低位的 1 所代表的值
     * @param x 输入整数
     * @return x & (-x)
     */
    int lowbit(int x) {
        return x & (-x);
    }

    /**
     * 在指定位置增加 delta
     * @param index 索引位置（从 1 开始）
     * @param delta 增加的值
     */
    void update(int index, int delta) {
        // 沿树状数组向上更新所有相关节点
        while (index <= size) {
            tree[index] += delta;
            index += lowbit(index);
        }
    }

    /**
     * 查询前缀和[1, index]
     * @param index 查询的右边界（从 1 开始）
     * @return 前缀和
     */
    int query(int index) {
        int sum = 0;
        // 沿树状数组向下累加所有相关节点的值
        while (index > 0) {
            sum += tree[index];
            index -= lowbit(index);
        }
        return sum;
    }
};

/**
 * 归并排序的合并操作，用于归并排序解法
 */
void merge(vector<int>& nums, vector<int>& indexes, int left, int mid, int right,
vector<int>& result) {
    vector<int> tempIndexes(right - left + 1);
```

```
        int i = left, j = mid + 1, k = 0;

        // 合并两个有序数组，并计算右侧小于当前元素的个数
        while (i <= mid && j <= right) {
            if (nums[indexes[i]] <= nums[indexes[j]]) {
                // 右侧比当前元素小的数量为 j - (mid + 1)
                result[indexes[i]] += j - (mid + 1);
                tempIndexes[k++] = indexes[i++];
            } else {
                tempIndexes[k++] = indexes[j++];
            }
        }

        // 处理剩余元素
        while (i <= mid) {
            result[indexes[i]] += j - (mid + 1);
            tempIndexes[k++] = indexes[i++];
        }

        while (j <= right) {
            tempIndexes[k++] = indexes[j++];
        }

        // 将临时数组复制回原数组
        for (int p = 0; p < tempIndexes.size(); p++) {
            indexes[left + p] = tempIndexes[p];
        }
    }

    /**
     * 归并排序的递归实现，用于归并排序解法
     */
    void mergeSort(vector<int>& nums, vector<int>& indexes, int left, int right, vector<int>&
result) {
        if (left >= right) return;

        int mid = left + (right - left) / 2;
        mergeSort(nums, indexes, left, mid, result);
        mergeSort(nums, indexes, mid + 1, right, result);
        merge(nums, indexes, left, mid, right, result);
    }

public:
```

```
/**
 * 计算右侧小于当前元素的个数 - 树状数组解法
 * @param nums 输入数组
 * @return 结果数组
 */
vector<int> countSmaller(vector<int>& nums) {
    // 边界条件检查
    if (nums.empty()) {
        return {};
    }

    int n = nums.size();
    vector<int> result(n, 0);

    // 离散化处理
    // 1. 收集所有可能的数值
    unordered_set<long long> valuesSet;
    for (int num : nums) {
        valuesSet.insert((long long)num);
    }

    // 2. 排序并去重
    vector<long long> sortedValues(valuesSet.begin(), valuesSet.end());
    sort(sortedValues.begin(), sortedValues.end());

    // 3. 建立值到索引的映射
    unordered_map<long long, int> valueToIndex;
    for (int i = 0; i < sortedValues.size(); i++) {
        valueToIndex[sortedValues[i]] = i + 1;  // 索引从1开始
    }

    // 创建树状数组
    FenwickTree fenwickTree(sortedValues.size());

    // 从右到左遍历数组
    for (int i = n - 1; i >= 0; i--) {
        long long currentValue = (long long)nums[i];
        // 查询比当前值小的元素个数
        int count = 0;
        // 找到当前值在离散化数组中的位置
        int index = valueToIndex[currentValue];
        // 查询比当前值小的元素个数，即查询[1, index-1]的前缀和
        if (index > 1) {
```

```
            count = fenwickTree.query(index - 1);
        }
        // 将结果保存
        result[i] = count;
        // 将当前值加入树状数组
        fenwickTree.update(index, 1);
    }

    return result;
}

/**
 * 另一种解法：归并排序过程中计算逆序对
 * 这种方法也能在O(n log n)时间内解决问题
 * @param nums 输入数组
 * @return 结果数组
 */
vector<int> countSmallerMergeSort(vector<int>& nums) {
    int n = nums.size();
    vector<int> result(n, 0);
    if (n == 0) return result;

    // 创建索引数组，用于跟踪元素原始位置
    vector<int> indexes(n);
    for (int i = 0; i < n; i++) {
        indexes[i] = i;
    }

    // 归并排序过程中计算右侧小于当前元素的个数
    mergeSort(nums, indexes, 0, n - 1, result);
    return result;
}

/**
 * 暴力解法（仅供比较，时间复杂度较高）
 * 时间复杂度: O(n²)
 * 空间复杂度: O(n)
 * @param nums 输入数组
 * @return 结果数组
 */
vector<int> countSmallerBruteForce(vector<int>& nums) {
    vector<int> result;
    if (nums.empty()) {
```

```
            return result;
        }

        int n = nums.size();
        result.resize(n, 0);
        for (int i = 0; i < n; i++) {
            int count = 0;
            for (int j = i + 1; j < n; j++) {
                if (nums[j] < nums[i]) {
                    count++;
                }
            }
            result[i] = count;
        }
        return result;
    }
};

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

/**
 * 测试函数
 */
void testSolution() {
    Solution solution;

    // 测试用例 1
    vector<int> nums1 = {5, 2, 6, 1};
    cout << "测试用例 1:" << endl;
    cout << "输入: [5, 2, 6, 1]" << endl;
    vector<int> result1 = solution.countSmaller(nums1);
```

```
cout << "树状数组解法结果: ";
printVector(result1);  // 期望输出: [2, 1, 1, 0]

vector<int> result1_mergesort = solution.countSmallerMergeSort(nums1);
cout << "归并排序解法结果: ";
printVector(result1_mergesort);  // 期望输出: [2, 1, 1, 0]

vector<int> result1_brute = solution.countSmallerBruteForce(nums1);
cout << "暴力解法结果: ";
printVector(result1_brute);  // 期望输出: [2, 1, 1, 0]

// 测试用例 2
vector<int> nums2 = {-1, -1};
cout << "\n测试用例 2:" << endl;
cout << "输入: [-1, -1]" << endl;
vector<int> result2 = solution.countSmaller(nums2);
cout << "树状数组解法结果: ";
printVector(result2);  // 期望输出: [0, 0]

// 测试用例 3 - 空数组
vector<int> nums3 = {};
cout << "\n测试用例 3:" << endl;
cout << "输入: []" << endl;
vector<int> result3 = solution.countSmaller(nums3);
cout << "树状数组解法结果: ";
printVector(result3);  // 期望输出: []

// 测试用例 4 - 大规模数据
int size = 1000;
vector<int> nums4(size);
for (int i = 0; i < size; i++) {
    nums4[i] = size - i;
}
cout << "\n测试用例 4 (大规模逆序数组):" << endl;
cout << "数组长度: " << size << endl;

// 测量树状数组解法的时间
auto start1 = high_resolution_clock::now();
vector<int> result4 = solution.countSmaller(nums4);
auto end1 = high_resolution_clock::now();
auto duration1 = duration_cast<microseconds>(end1 - start1);
cout << "树状数组解法耗时: " << duration1.count() / 1000.0 << "ms" << endl;
```

```
    // 测量归并排序解法的时间
    auto start2 = high_resolution_clock::now();
    vector<int> result4_mergesort = solution.countSmallerMergeSort(nums4);
    auto end2 = high_resolution_clock::now();
    auto duration2 = duration_cast<microseconds>(end2 - start2);
    cout << "归并排序解法耗时: " << duration2.count() / 1000.0 << "ms" << endl;

    // 验证两种方法结果是否一致
    bool resultsEqual = (result4 == result4_mergesort);
    cout << "两种方法结果一致: " << (resultsEqual ? "true" : "false") << endl;

    // 对比暴力解法（仅在小规模数据上测试）
    if (size <= 1000) {
        int smallSize = min(size, 500);  // 限制暴力解法的数组大小，避免超时
        vector<int> smallNums(nums4.begin(), nums4.begin() + smallSize);

        auto start3 = high_resolution_clock::now();
        vector<int> result4_brute = solution.countSmallerBruteForce(smallNums);
        auto end3 = high_resolution_clock::now();
        auto duration3 = duration_cast<microseconds>(end3 - start3);
        cout << "暴力解法(前" << smallSize << "个元素)耗时: " << duration3.count() / 1000.0 <<
"ms" << endl;

        // 验证暴力解法与树状数组解法在前 smallSize 个元素上是否一致
        vector<int> result4_small = solution.countSmaller(smallNums);
        bool bruteEqual = (result4_small == result4_brute);
        cout << "暴力解法与树状数组解法结果一致: " << (bruteEqual ? "true" : "false") << endl;
    }
}

// 主函数
int main() {
    testSolution();
    return 0;
}

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

文件: Code12_CountSmallerNumbersAfterSelf_Enhanced.java
==============================================
package class132;

import java.util.*;
```

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

public class Code12_CountSmallerNumbersAfterSelf_Enhanced {

    /**
     * 计算右侧小于当前元素的个数 - 树状数组解法
     *
     * 解题思路:
     * 1. 题目要求计算每个元素右侧小于它的元素个数
     * 2. 我们可以将问题转化为：对于每个元素 nums[i]，统计有多少个元素 nums[j] (j > i)满足 nums[j] < nums[i]
     * 3. 使用树状数组可以高效地进行这类统计
     * 4. 具体步骤：
     *    a. 将数组元素离散化，以处理可能的大范围数值
     *    b. 从右到左遍历数组，对于每个元素 nums[i]：
     *       - 查询树状数组中小于 nums[i]的元素个数（即前缀和）
     *       - 将 nums[i]加入树状数组
     *    c. 这样就能保证每次查询的都是当前元素右侧的元素
     *
     * 时间复杂度分析:
     * - 离散化: O(n log n)
     * - 构建和操作树状数组: O(n log n)
     * - 总时间复杂度: O(n log n)
     *
     * 空间复杂度分析:
     * - 树状数组: O(n)
     * - 离散化数组和映射: O(n)
     * - 结果数组: O(n)
     * - 总空间复杂度: O(n)
     *
     * 工程化考量:
     * 1. 离散化处理：由于输入数组可能包含很大范围的整数，离散化可以有效减少空间使用
     * 2. 边界条件处理：处理空数组、单元素数组等特殊情况
     * 3. 数据类型溢出：使用 long 类型避免中间计算时的溢出
     * 4. 异常输入检查：验证输入的有效性
     *
     * 算法优化点:
     * 1. 离散化时只处理必要的数值，节省空间
     * 2. 从右到左遍历，确保正确统计右侧元素
```

```
 * 3. 树状数组的 lowbit 操作高效计算区间和
 */

// 树状数组类
static class FenwickTree {
    private int[] tree;  // 树状数组
    private int size;    // 数组大小

    /**
     * 构造函数
     * @param size 树状数组大小
     */
    public FenwickTree(int size) {
        this.size = size;
        this.tree = new int[size + 1];  // 树状数组下标从 1 开始
    }

    /**
     * lowbit 操作，获取 x 的二进制表示中最低位的 1 所代表的值
     * @param x 输入整数
     * @return x & (-x)
     */
    private int lowbit(int x) {
        return x & (-x);
    }

    /**
     * 在指定位置增加 delta
     * @param index 索引位置（从 1 开始）
     * @param delta 增加的值
     */
    public void update(int index, int delta) {
        // 沿树状数组向上更新所有相关节点
        while (index <= size) {
            tree[index] += delta;
            index += lowbit(index);
        }
    }

    /**
     * 查询前缀和[1, index]
     * @param index 查询的右边界（从 1 开始）
     * @return 前缀和
```

```
     */
    public int query(int index) {
        int sum = 0;
        // 沿树状数组向下累加所有相关节点的值
        while (index > 0) {
            sum += tree[index];
            index -= lowbit(index);
        }
        return sum;
    }
}

/**
 * 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) {
    // 边界条件检查
    if (nums == null || nums.length == 0) {
        return new ArrayList<>();
    }

    int n = nums.length;
    List<Integer> result = new ArrayList<>(n);

    // 离散化处理
    // 1. 收集所有可能的数值
    Set<Long> valuesSet = new HashSet<>();
    for (int num : nums) {
        valuesSet.add((long) num);
    }

    // 2. 排序并去重
    List<Long> sortedValues = new ArrayList<>(valuesSet);
    Collections.sort(sortedValues);

    // 3. 建立值到索引的映射
    Map<Long, Integer> valueToIndex = new HashMap<>();
    for (int i = 0; i < sortedValues.size(); i++) {
        valueToIndex.put(sortedValues.get(i), i + 1);  // 索引从1开始
    }
```

```
    // 创建树状数组
    FenwickTree fenwickTree = new FenwickTree(sortedValues.size());

    // 从右到左遍历数组
    for (int i = n - 1; i >= 0; i--) {
        long currentValue = (long) nums[i];
        // 查询比当前值小的元素个数
        int count = 0;
        // 找到当前值在离散化数组中的位置
        int index = valueToIndex.get(currentValue);
        // 查询比当前值小的元素个数，即查询[1, index-1]的前缀和
        if (index > 1) {
            count = fenwickTree.query(index - 1);
        }
        // 将结果添加到列表（注意后续需要反转）
        result.add(count);
        // 将当前值加入树状数组
        fenwickTree.update(index, 1);
    }

    // 反转结果列表，因为我们是从右到左计算的
    Collections.reverse(result);
    return result;
}

/**
 * 另一种解法：归并排序过程中计算逆序对
 * 这种方法也能在O(n log n)时间内解决问题
 */
public List<Integer> countSmallerMergeSort(int[] nums) {
    int n = nums.length;
    List<Integer> result = new ArrayList<>(Collections.nCopies(n, 0));
    if (n == 0) return result;

    // 创建索引数组，用于跟踪元素原始位置
    int[] indexes = new int[n];
    for (int i = 0; i < n; i++) {
        indexes[i] = i;
    }

    // 归并排序过程中计算右侧小于当前元素的个数
    mergeSort(nums, indexes, 0, n - 1, result);
    return result;
```

```
    }

    private void mergeSort(int[] nums, int[] indexes, int left, int right, List<Integer> result)
{
        if (left >= right) return;

        int mid = left + (right - left) / 2;
        mergeSort(nums, indexes, left, mid, result);
        mergeSort(nums, indexes, mid + 1, right, result);
        merge(nums, indexes, left, mid, right, result);
    }

    private void merge(int[] nums, int[] indexes, int left, int mid, int right, List<Integer>
result) {
        int[] tempIndexes = new int[right - left + 1];
        int i = left, j = mid + 1, k = 0;

        // 合并两个有序数组，并计算右侧小于当前元素的个数
        while (i <= mid && j <= right) {
            if (nums[indexes[i]] <= nums[indexes[j]]) {
                // 右侧比当前元素小的数量为 j - (mid + 1)
                result.set(indexes[i], result.get(indexes[i]) + (j - (mid + 1)));
                tempIndexes[k++] = indexes[i++];
            } else {
                tempIndexes[k++] = indexes[j++];
            }
        }

        // 处理剩余元素
        while (i <= mid) {
            result.set(indexes[i], result.get(indexes[i]) + (j - (mid + 1)));
            tempIndexes[k++] = indexes[i++];
        }

        while (j <= right) {
            tempIndexes[k++] = indexes[j++];
        }

        // 将临时数组复制回原数组
        System.arraycopy(tempIndexes, 0, indexes, left, tempIndexes.length);
    }

    /**
```

```
     * 暴力解法（仅供比较，时间复杂度较高）
     * 时间复杂度: O(n²)
     * 空间复杂度: O(n)
     */
    public List<Integer> countSmallerBruteForce(int[] nums) {
        List<Integer> result = new ArrayList<>();
        if (nums == null || nums.length == 0) {
            return result;
        }

        int n = nums.length;
        for (int i = 0; i < n; i++) {
            int count = 0;
            for (int j = i + 1; j < n; j++) {
                if (nums[j] < nums[i]) {
                    count++;
                }
            }
            result.add(count);
        }
        return result;
    }

    // 测试方法
    public static void main(String[] args) {
        Code12_CountSmallerNumbersAfterSelf_Enhanced solution = new
Code12_CountSmallerNumbersAfterSelf_Enhanced();

        // 测试用例 1
        int[] nums1 = {5, 2, 6, 1};
        System.out.println("测试用例 1:");
        System.out.println("输入: [5, 2, 6, 1]");
        System.out.println("树状数组解法结果: " + solution.countSmaller(nums1));  // 期望输出:
[2, 1, 1, 0]

        // 测试用例 2
        int[] nums2 = {-1, -1};
        System.out.println("\n 测试用例 2:");
        System.out.println("输入: [-1, -1]");
        System.out.println("树状数组解法结果: " + solution.countSmaller(nums2));  // 期望输出:
[0, 0]

        // 测试用例 3 - 空数组
```

```
        int[] nums3 = {};
        System.out.println("\n 测试用例 3:");
        System.out.println("输入: []");
        System.out.println("树状数组解法结果: " + solution.countSmaller(nums3));  // 期望输出: []

        // 测试用例 4 - 大规模数据
        int size = 1000;
        int[] nums4 = new int[size];
        for (int i = 0; i < size; i++) {
            nums4[i] = size - i;
        }
        System.out.println("\n 测试用例 4 (大规模逆序数组):");
        System.out.println("数组长度: " + size);

        long startTime1 = System.currentTimeMillis();
        List<Integer> result1 = solution.countSmaller(nums4);
        long endTime1 = System.currentTimeMillis();
        System.out.println("树状数组解法耗时: " + (endTime1 - startTime1) + "ms");

        long startTime2 = System.currentTimeMillis();
        List<Integer> result2 = solution.countSmallerMergeSort(nums4);
        long endTime2 = System.currentTimeMillis();
        System.out.println("归并排序解法耗时: " + (endTime2 - startTime2) + "ms");

        // 验证两种方法结果是否一致
        System.out.println("两种方法结果一致: " + result1.equals(result2));

        // 对比暴力解法（仅在小规模数据上测试）
        if (size <= 1000) {
            long startTime3 = System.currentTimeMillis();
            List<Integer> result3 = solution.countSmallerBruteForce(nums4);
            long endTime3 = System.currentTimeMillis();
            System.out.println("暴力解法耗时: " + (endTime3 - startTime3) + "ms");
            System.out.println("暴力解法与树状数组解法结果一致: " + result1.equals(result3));
        }
    }
}
```

==========

文件: Code12_CountSmallerNumbersAfterSelf_Enhanced.py
==========

```
# LeetCode 315. 计算右侧小于当前元素的个数
```

```
# 给定一个整数数组 nums，按要求返回一个新数组 counts。
# 数组 counts 有该性质：counts[i] 的值是 nums[i] 右侧小于 nums[i] 的元素的数量。
# 测试链接: https://leetcode.cn/problems/count-of-smaller-numbers-after-self/

from typing import List
import bisect

class Solution:
    """
    计算右侧小于当前元素的个数 - 树状数组解法

    解题思路:
    1. 题目要求计算每个元素右侧小于它的元素个数
    2. 我们可以将问题转化为：对于每个元素 nums[i]，统计有多少个元素 nums[j] (j > i)满足 nums[j] < nums[i]
    3. 使用树状数组可以高效地进行这类统计
    4. 具体步骤：
       a. 将数组元素离散化，以处理可能的大范围数值
       b. 从右到左遍历数组，对于每个元素 nums[i]：
          - 查询树状数组中小于 nums[i]的元素个数（即前缀和）
          - 将 nums[i]加入树状数组
       c. 这样就能保证每次查询的都是当前元素右侧的元素

    时间复杂度分析:
    - 离散化: O(n log n)
    - 构建和操作树状数组: O(n log n)
    - 总时间复杂度: O(n log n)

    空间复杂度分析:
    - 树状数组: O(n)
    - 离散化数组和映射: O(n)
    - 结果数组: O(n)
    - 总空间复杂度: O(n)

    工程化考量:
    1. 离散化处理：由于输入数组可能包含很大范围的整数，离散化可以有效减少空间使用
    2. 边界条件处理：处理空数组、单元素数组等特殊情况
    3. 数据类型溢出：Python 自动处理大数，无需特别关注
    4. 异常输入检查：验证输入的有效性
    5. 代码可读性：使用清晰的变量命名和详细的注释
    """

    class FenwickTree:
```

```
    """
    树状数组类
    用于高效计算前缀和和单点更新
    """
    def __init__(self, size: int):
        """
        初始化树状数组
        :param size: 数组大小
        """
        self.n = size
        self.tree = [0] * (size + 1)  # 树状数组下标从1开始

    def lowbit(self, x: int) -> int:
        """
        lowbit操作，获取x的二进制表示中最低位的1所代表的值
        :param x: 输入整数
        :return: x & (-x)
        """
        return x & (-x)

    def update(self, index: int, delta: int) -> None:
        """
        在指定位置增加delta
        :param index: 索引位置（从1开始）
        :param delta: 增加的值
        """
        # 沿树状数组向上更新所有相关节点
        while index <= self.n:
            self.tree[index] += delta
            index += self.lowbit(index)

    def query(self, index: int) -> int:
        """
        查询前缀和[1, index]
        :param index: 查询的右边界（从1开始）
        :return: 前缀和
        """
        sum_val = 0
        # 沿树状数组向下累加所有相关节点的值
        while index > 0:
            sum_val += self.tree[index]
            index -= self.lowbit(index)
        return sum_val
```

```
def countSmaller(self, nums: List[int]) -> List[int]:
    """
    计算右侧小于当前元素的个数 - 树状数组解法
    :param nums: 输入数组
    :return: 结果数组
    """
    # 边界条件检查
    if not nums:
        return []

    n = len(nums)
    result = []

    # 离散化处理
    # 1. 收集所有可能的数值
    values_set = set()
    for num in nums:
        values_set.add(num)

    # 2. 排序并去重
    sorted_values = sorted(values_set)

    # 3. 建立值到索引的映射
    value_to_index = {value: i + 1 for i, value in enumerate(sorted_values)}  # 索引从1开始

    # 创建树状数组
    fenwick_tree = self.FenwickTree(len(sorted_values))

    # 从右到左遍历数组
    for i in range(n - 1, -1, -1):
        current_value = nums[i]
        # 查询比当前值小的元素个数
        count = 0
        # 找到当前值在离散化数组中的位置
        index = value_to_index[current_value]
        # 查询比当前值小的元素个数，即查询[1, index-1]的前缀和
        if index > 1:
            count = fenwick_tree.query(index - 1)
        # 将结果添加到列表（注意后续需要反转）
        result.append(count)
        # 将当前值加入树状数组
        fenwick_tree.update(index, 1)
```

```
        # 反转结果列表，因为我们是从右到左计算的
        return result[::-1]

    def countSmallerMergeSort(self, nums: List[int]) -> List[int]:
        """
        另一种解法：归并排序过程中计算逆序对
        这种方法也能在O(n log n)时间内解决问题
        :param nums: 输入数组
        :return: 结果数组
        """
        n = len(nums)
        result = [0] * n
        if n == 0:
            return result

        # 创建索引数组，用于跟踪元素原始位置
        indexes = list(range(n))

        # 归并排序过程中计算右侧小于当前元素的个数
        self._merge_sort(nums, indexes, 0, n - 1, result)
        return result

    def _merge_sort(self, nums: List[int], indexes: List[int], left: int, right: int, result:
List[int]) -> None:
        """
        归并排序的递归实现
        """
        if left >= right:
            return

        mid = left + (right - left) // 2
        self._merge_sort(nums, indexes, left, mid, result)
        self._merge_sort(nums, indexes, mid + 1, right, result)
        self._merge(nums, indexes, left, mid, right, result)

    def _merge(self, nums: List[int], indexes: List[int], left: int, mid: int, right: int,
result: List[int]) -> None:
        """
        合并两个有序数组，并计算右侧小于当前元素的个数
        """
        temp_indexes = []
        i, j = left, mid + 1
```

```
    # 合并两个有序数组，并计算右侧小于当前元素的个数
    while i <= mid and j <= right:
        if nums[indexes[i]] <= nums[indexes[j]]:
            # 右侧比当前元素小的数量为 j - (mid + 1)
            result[indexes[i]] += j - (mid + 1)
            temp_indexes.append(indexes[i])
            i += 1
        else:
            temp_indexes.append(indexes[j])
            j += 1

    # 处理剩余元素
    while i <= mid:
        result[indexes[i]] += j - (mid + 1)
        temp_indexes.append(indexes[i])
        i += 1

    while j <= right:
        temp_indexes.append(indexes[j])
        j += 1

    # 将临时数组复制回原数组
    for k in range(len(temp_indexes)):
        indexes[left + k] = temp_indexes[k]

def countSmallerBruteForce(self, nums: List[int]) -> List[int]:
    """
    暴力解法（仅供比较，时间复杂度较高）
    时间复杂度: O(n²)
    空间复杂度: O(n)
    :param nums: 输入数组
    :return: 结果数组
    """
    result = []
    if not nums:
        return result

    n = len(nums)
    for i in range(n):
        count = 0
        for j in range(i + 1, n):
            if nums[j] < nums[i]:
```

```
                    count += 1
            result.append(count)
        return result

    def countSmallerWithBisect(self, nums: List[int]) -> List[int]:
        """
        利用 bisect 模块的解法
        从右到左遍历，维护一个有序数组，使用 bisect_left 找到插入位置
        该位置即为比当前元素小的元素个数
        时间复杂度: O(n²)，因为插入操作是 O(n)
        空间复杂度: O(n)
        :param nums: 输入数组
        :return: 结果数组
        """
        result = []
        if not nums:
            return result

        sorted_list = []
        # 从右到左遍历
        for num in reversed(nums):
            # 找到插入位置，该位置即为比当前元素小的元素个数
            index = bisect.bisect_left(sorted_list, num)
            result.append(index)
            # 将当前元素插入到有序数组中
            bisect.insort(sorted_list, num)

        # 反转结果列表
        return result[::-1]

# 测试函数
def test_solution():
    solution = Solution()

    # 测试用例 1
    nums1 = [5, 2, 6, 1]
    print("测试用例 1:")
    print(f"输入: {nums1}")
    print(f"树状数组解法结果: {solution.countSmaller(nums1)}")  # 期望输出: [2, 1, 1, 0]
    print(f"归并排序解法结果: {solution.countSmallerMergeSort(nums1)}")  # 期望输出: [2, 1, 1, 0]
    print(f"Bisect 解法结果: {solution.countSmallerWithBisect(nums1)}")  # 期望输出: [2, 1, 1, 0]
    print(f"暴力解法结果: {solution.countSmallerBruteForce(nums1)}")  # 期望输出: [2, 1, 1, 0]
```

```
# 测试用例 2
nums2 = [-1, -1]
print("\n 测试用例 2:")
print(f"输入: {nums2}")
print(f"树状数组解法结果: {solution.countSmaller(nums2)}")  # 期望输出: [0, 0]

# 测试用例 3 - 空数组
nums3 = []
print("\n 测试用例 3:")
print(f"输入: {nums3}")
print(f"树状数组解法结果: {solution.countSmaller(nums3)}")  # 期望输出: []

# 测试用例 4 - 大规模数据
size = 1000
nums4 = list(range(size, 0, -1))  # 逆序数组
print("\n 测试用例 4 (大规模逆序数组):")
print(f"数组长度: {size}")

import time

start_time1 = time.time()
result1 = solution.countSmaller(nums4)
end_time1 = time.time()
print(f"树状数组解法耗时: {(end_time1 - start_time1) * 1000:.2f}ms")

start_time2 = time.time()
result2 = solution.countSmallerMergeSort(nums4)
end_time2 = time.time()
print(f"归并排序解法耗时: {(end_time2 - start_time2) * 1000:.2f}ms")

start_time3 = time.time()
result3 = solution.countSmallerWithBisect(nums4)
end_time3 = time.time()
print(f"Bisect 解法耗时: {(end_time3 - start_time3) * 1000:.2f}ms")

# 验证所有方法结果是否一致
print(f"树状数组与归并排序结果一致: {result1 == result2}")
print(f"树状数组与 Bisect 结果一致: {result1 == result3}")

# 对比暴力解法（仅在小规模数据上测试）
if size <= 1000:
    small_size = min(size, 500)  # 限制暴力解法的数组大小，避免超时
    small_nums = nums4[:small_size]
```

```
        start_time4 = time.time()
        result4 = solution.countSmallerBruteForce(small_nums)
        end_time4 = time.time()
        print(f"暴力解法(前{small_size}个元素)耗时: {(end_time4 - start_time4) * 1000:.2f}ms")

        # 验证暴力解法与树状数组解法在前 small_size 个元素上是否一致
        result1_small = solution.countSmaller(small_nums)
        print(f"暴力解法与树状数组解法结果一致: {result1_small == result4}")

# 运行测试
if __name__ == "__main__":
    test_solution()

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

文件: Code13_RangeSumQueryCount.cpp
================================================
// LeetCode 327. 区间和的个数
// 给定一个整数数组 nums 以及两个整数 lower 和 upper，求区间 [lower, upper] 内的区间和的个数。
// 测试链接: https://leetcode.cn/problems/count-of-range-sum/

#include <iostream>
#include <vector>
#include <algorithm>
#include <unordered_set>
#include <unordered_map>
#include <climits>
#include <chrono>
using namespace std;
using namespace std::chrono;

/**
 * 区间和的个数 - 树状数组解法
 *
 * 解题思路:
 * 1. 题目要求计算数组中区间和位于 [lower, upper] 范围内的子数组个数
 * 2. 利用前缀和思想：区间和 sum(i,j) = prefix[j+1] - prefix[i]
 * 3. 问题转化为：对于每个 j，计算有多少个 i < j 满足 lower <= prefix[j] - prefix[i] <= upper
 * 4. 进一步转化为：对于每个 j，统计 prefix[i] 的范围为 [prefix[j] - upper, prefix[j] - lower] 的
i 的个数
 * 5. 使用树状数组可以高效地统计这个范围查询
 * 6. 由于前缀和可能很大，需要进行离散化处理
```

```
 *
 * 时间复杂度分析:
 * - 计算前缀和: O(n)
 * - 离散化: O(n log n)
 * - 构建和操作树状数组: O(n log n)
 * - 总时间复杂度: O(n log n)
 *
 * 空间复杂度分析:
 * - 前缀和数组: O(n)
 * - 离散化数组和映射: O(n)
 * - 树状数组: O(n)
 * - 总空间复杂度: O(n)
 *
 * 工程化考量:
 * 1. 离散化处理：由于前缀和可能超出整数范围，使用 long long 类型存储，并进行离散化
 * 2. 边界条件处理：处理空数组、单元素数组等特殊情况
 * 3. 数据类型溢出：使用 long long 类型避免溢出
 * 4. 异常输入检查：验证输入的有效性
 * 5. 代码可读性：使用清晰的变量命名和详细的注释
 */

class Solution {
private:
    // 树状数组类
    class FenwickTree {
    private:
        vector<int> tree;  // 树状数组
        int size;          // 数组大小

    public:
        /**
         * 构造函数
         * @param size 树状数组大小
         */
        FenwickTree(int size) : size(size) {
            tree.resize(size + 1, 0);  // 树状数组下标从 1 开始
        }

        /**
         * lowbit 操作，获取 x 的二进制表示中最低位的 1 所代表的值
         * @param x 输入整数
         * @return x & (-x)
         */
```

```
int lowbit(int x) {
    return x & (-x);
}

/**
 * 在指定位置增加 delta
 * @param index 索引位置（从 1 开始）
 * @param delta 增加的值
 */
void update(int index, int delta) {
    // 沿树状数组向上更新所有相关节点
    while (index <= size) {
        tree[index] += delta;
        index += lowbit(index);
    }
}

/**
 * 查询前缀和[1, index]
 * @param index 查询的右边界（从 1 开始）
 * @return 前缀和
 */
int query(int index) {
    int sum = 0;
    // 沿树状数组向下累加所有相关节点的值
    while (index > 0) {
        sum += tree[index];
        index -= lowbit(index);
    }
    return sum;
}

/**
 * 查询区间和[left, right]
 * @param left 区间左边界（从 1 开始）
 * @param right 区间右边界（从 1 开始）
 * @return 区间和
 */
int queryRange(int left, int right) {
    if (left > right) {
        return 0;
    }
    return query(right) - query(left - 1);
```

```
        }
    };

    /**
     * 合并两个有序数组，用于归并排序解法
     */
    void merge(vector<long long>& prefixSums, int left, int mid, int right) {
        vector<long long> temp(right - left + 1);
        int i = left, j = mid + 1, k = 0;

        // 合并两个有序数组
        while (i <= mid && j <= right) {
            if (prefixSums[i] <= prefixSums[j]) {
                temp[k++] = prefixSums[i++];
            } else {
                temp[k++] = prefixSums[j++];
            }
        }

        // 处理剩余元素
        while (i <= mid) {
            temp[k++] = prefixSums[i++];
        }

        while (j <= right) {
            temp[k++] = prefixSums[j++];
        }

        // 将临时数组复制回原数组
        for (int p = 0; p < temp.size(); p++) {
            prefixSums[left + p] = temp[p];
        }
    }

    /**
     * 归并排序并统计满足条件的子数组个数，用于归并排序解法
     */
    int mergeSortAndCount(vector<long long>& prefixSums, int left, int right, int lower, int
upper) {
        if (left >= right) {
            return 0;
        }
```

```
        int mid = left + (right - left) / 2;
        // 递归处理左右两部分
        int count = mergeSortAndCount(prefixSums, left, mid, lower, upper) +
                    mergeSortAndCount(prefixSums, mid + 1, right, lower, upper);

        // 统计满足条件的子数组个数
        int j = mid + 1, k = mid + 1;
        for (int i = left; i <= mid; i++) {
            // 找到最小的 j，使得 prefixSums[j] - prefixSums[i] >= lower
            while (j <= right && prefixSums[j] - prefixSums[i] < lower) {
                j++;
            }
            // 找到最大的 k，使得 prefixSums[k] - prefixSums[i] <= upper
            while (k <= right && prefixSums[k] - prefixSums[i] <= upper) {
                k++;
            }
            // 区间[j, k-1]内的所有前缀和都满足条件
            count += k - j;
        }

        // 合并两个有序数组
        merge(prefixSums, left, mid, right);

        return count;
    }

public:
    /**
     * 计算区间和的个数 - 树状数组解法
     * @param nums 输入数组
     * @param lower 区间下界
     * @param upper 区间上界
     * @return 满足条件的子数组个数
     */
    int countRangeSum(vector<int>& nums, int lower, int upper) {
        // 边界条件检查
        if (nums.empty()) {
            return 0;
        }

        int n = nums.size();
        vector<long long> prefixSums(n + 1);  // 前缀和数组，使用 long long 避免溢出
```

```
        // 计算前缀和
        for (int i = 0; i < n; i++) {
            prefixSums[i + 1] = prefixSums[i] + nums[i];
        }

        // 离散化处理
        // 收集所有可能需要查询的值
        unordered_set<long long> valuesSet;
        for (long long sum : prefixSums) {
            valuesSet.insert(sum);
            valuesSet.insert(sum - lower);
            valuesSet.insert(sum - upper);
        }

        // 排序并去重
        vector<long long> sortedValues(valuesSet.begin(), valuesSet.end());
        sort(sortedValues.begin(), sortedValues.end());

        // 建立值到索引的映射
        unordered_map<long long, int> valueToIndex;
        for (int i = 0; i < sortedValues.size(); i++) {
            valueToIndex[sortedValues[i]] = i + 1;  // 索引从1开始
        }

        // 创建树状数组
        FenwickTree fenwickTree(sortedValues.size());
        int count = 0;

        // 从前向后遍历前缀和数组
        for (long long prefixSum : prefixSums) {
            // 查询满足条件的前缀和的数量：prefixSum[j] - upper <= prefixSum[i] <= prefixSum[j] -
lower
            int leftIndex = valueToIndex[prefixSum - upper];
            int rightIndex = valueToIndex[prefixSum - lower];
            count += fenwickTree.queryRange(leftIndex, rightIndex);

            // 将当前前缀和加入树状数组
            int currentIndex = valueToIndex[prefixSum];
            fenwickTree.update(currentIndex, 1);
        }

        return count;
    }
```

```
/**
 * 暴力解法（仅供比较，时间复杂度较高）
 * 时间复杂度: O(n²)
 * 空间复杂度: O(n)
 * @param nums 输入数组
 * @param lower 区间下界
 * @param upper 区间上界
 * @return 满足条件的子数组个数
 */
int countRangeSumBruteForce(vector<int>& nums, int lower, int upper) {
    if (nums.empty()) {
        return 0;
    }

    int n = nums.size();
    vector<long long> prefixSums(n + 1);  // 使用 long long 避免溢出

    // 计算前缀和
    for (int i = 0; i < n; i++) {
        prefixSums[i + 1] = prefixSums[i] + nums[i];
    }

    int count = 0;
    // 暴力枚举所有可能的子数组
    for (int i = 0; i < n; i++) {
        for (int j = i + 1; j <= n; j++) {
            long long rangeSum = prefixSums[j] - prefixSums[i];
            if (rangeSum >= lower && rangeSum <= upper) {
                count++;
            }
        }
    }

    return count;
}

/**
 * 归并排序解法（另一种 O(n log n)的解法）
 * @param nums 输入数组
 * @param lower 区间下界
 * @param upper 区间上界
 * @return 满足条件的子数组个数
```

```
     */
    int countRangeSumMergeSort(vector<int>& nums, int lower, int upper) {
        if (nums.empty()) {
            return 0;
        }

        int n = nums.size();
        vector<long long> prefixSums(n + 1);  // 使用 long long 避免溢出

        // 计算前缀和
        for (int i = 0; i < n; i++) {
            prefixSums[i + 1] = prefixSums[i] + nums[i];
        }

        // 归并排序过程中统计满足条件的子数组个数
        return mergeSortAndCount(prefixSums, 0, n, lower, upper);
    }
};

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

/**
 * 测试函数
 */
void testSolution() {
    Solution solution;

    // 测试用例 1
    vector<int> nums1 = {-2, 5, -1};
    int lower1 = -2;
    int upper1 = 2;
```

```
    cout << "测试用例 1:" << endl;
    cout << "输入数组: [";
    printVector(nums1);
    cout << "lower: " << lower1 << ", upper: " << upper1 << endl;
    cout << "树状数组解法结果: " << solution.countRangeSum(nums1, lower1, upper1) << endl;  // 期望输出: 3
    cout << "归并排序解法结果: " << solution.countRangeSumMergeSort(nums1, lower1, upper1) << endl;  // 期望输出: 3
    cout << "暴力解法结果: " << solution.countRangeSumBruteForce(nums1, lower1, upper1) << endl;  // 期望输出: 3

    // 测试用例 2
    vector<int> nums2 = {0};
    int lower2 = 0;
    int upper2 = 0;
    cout << "\n 测试用例 2:" << endl;
    cout << "输入数组: [";
    printVector(nums2);
    cout << "lower: " << lower2 << ", upper: " << upper2 << endl;
    cout << "树状数组解法结果: " << solution.countRangeSum(nums2, lower2, upper2) << endl;  // 期望输出: 1

    // 测试用例 3 - 空数组
    vector<int> nums3 = {};
    int lower3 = 0;
    int upper3 = 0;
    cout << "\n 测试用例 3:" << endl;
    cout << "输入数组: []" << endl;
    cout << "lower: " << lower3 << ", upper: " << upper3 << endl;
    cout << "树状数组解法结果: " << solution.countRangeSum(nums3, lower3, upper3) << endl;  // 期望输出: 0

    // 测试用例 4 - 大规模数据
    int size = 1000;
    vector<int> nums4(size);
    for (int i = 0; i < size; i++) {
        nums4[i] = (i % 3 == 0) ? -1 : (i % 3 == 1) ? 0 : 1;
    }
    int lower4 = -2;
    int upper4 = 2;
    cout << "\n 测试用例 4 (大规模数据):" << endl;
    cout << "数组长度: " << size << endl;
    cout << "lower: " << lower4 << ", upper: " << upper4 << endl;
```

```
    // 测量树状数组解法的时间
    auto start1 = high_resolution_clock::now();
    int result1 = solution.countRangeSum(nums4, lower4, upper4);
    auto end1 = high_resolution_clock::now();
    auto duration1 = duration_cast<microseconds>(end1 - start1);
    cout << "树状数组解法结果: " << result1 << endl;
    cout << "树状数组解法耗时: " << duration1.count() / 1000.0 << "ms" << endl;

    // 测量归并排序解法的时间
    auto start2 = high_resolution_clock::now();
    int result2 = solution.countRangeSumMergeSort(nums4, lower4, upper4);
    auto end2 = high_resolution_clock::now();
    auto duration2 = duration_cast<microseconds>(end2 - start2);
    cout << "归并排序解法结果: " << result2 << endl;
    cout << "归并排序解法耗时: " << duration2.count() / 1000.0 << "ms" << endl;

    // 验证两种方法结果是否一致
    cout << "两种方法结果一致: " << (result1 == result2 ? "true" : "false") << endl;

    // 对比暴力解法（仅在小规模数据上测试）
    if (size <= 1000) {
        int smallSize = min(size, 300);  // 限制暴力解法的数组大小，避免超时
        vector<int> smallNums(nums4.begin(), nums4.begin() + smallSize);

        auto start3 = high_resolution_clock::now();
        int result3 = solution.countRangeSumBruteForce(smallNums, lower4, upper4);
        auto end3 = high_resolution_clock::now();
        auto duration3 = duration_cast<microseconds>(end3 - start3);
        cout << "暴力解法(前" << smallSize << "个元素)结果: " << result3 << endl;
        cout << "暴力解法耗时: " << duration3.count() / 1000.0 << "ms" << endl;

        // 验证暴力解法与树状数组解法在前 smallSize 个元素上是否一致
        int result4 = solution.countRangeSum(smallNums, lower4, upper4);
        cout << "暴力解法与树状数组解法结果一致: " << (result3 == result4 ? "true" : "false") <<
endl;
    }
}

// 主函数
int main() {
    testSolution();
    return 0;
```

```
}

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

文件: Code13_RangeSumQueryCount.java
==========================================
// LeetCode 327. 区间和的个数
// 给定一个整数数组 nums 以及两个整数 lower 和 upper，求区间 [lower, upper] 内的区间和的个数。
// 测试链接: https://leetcode.cn/problems/count-of-range-sum/

import java.util.Arrays;
import java.util.HashMap;
import java.util.Map;
import java.util.Set;
import java.util.TreeSet;

/**
 * 区间和的个数 - 树状数组解法
 *
 * 解题思路:
 * 1. 题目要求计算数组中区间和位于 [lower, upper] 范围内的子数组个数
 * 2. 利用前缀和思想：区间和 sum(i,j) = prefix[j+1] - prefix[i]
 * 3. 问题转化为：对于每个 j，计算有多少个 i < j 满足 lower <= prefix[j] - prefix[i] <= upper
 * 4. 进一步转化为：对于每个 j，统计 prefix[i] 的范围为 [prefix[j] - upper, prefix[j] - lower] 的
i 的个数
 * 5. 使用树状数组可以高效地统计这个范围查询
 * 6. 由于前缀和可能很大，需要进行离散化处理
 *
 * 时间复杂度分析:
 * - 计算前缀和: O(n)
 * - 离散化: O(n log n)
 * - 构建和操作树状数组: O(n log n)
 * - 总时间复杂度: O(n log n)
 *
 * 空间复杂度分析:
 * - 前缀和数组: O(n)
 * - 离散化数组和映射: O(n)
 * - 树状数组: O(n)
 * - 总空间复杂度: O(n)
 *
 * 工程化考量:
 * 1. 离散化处理：由于前缀和可能超出整数范围，使用 long 类型存储，并进行离散化
 * 2. 边界条件处理：处理空数组、单元素数组等特殊情况
```

```
 * 3. 数据类型溢出：使用 long 类型避免溢出
 * 4. 异常输入检查：验证输入的有效性
 * 5. 代码可读性：使用清晰的变量命名和详细的注释
 */
public class Code13_RangeSumQueryCount {
    /**
     * 树状数组类
     * 用于高效计算前缀和和单点更新
     */
    private static class FenwickTree {
        private int[] tree; // 树状数组
        private int size;   // 数组大小

        /**
         * 构造函数
         * @param size 树状数组大小
         */
        public FenwickTree(int size) {
            this.size = size;
            this.tree = new int[size + 1]; // 树状数组下标从 1 开始
        }

        /**
         * lowbit 操作，获取 x 的二进制表示中最低位的 1 所代表的值
         * @param x 输入整数
         * @return x & (-x)
         */
        private int lowbit(int x) {
            return x & (-x);
        }

        /**
         * 在指定位置增加 delta
         * @param index 索引位置（从 1 开始）
         * @param delta 增加的值
         */
        public void update(int index, int delta) {
            // 沿树状数组向上更新所有相关节点
            while (index <= size) {
                tree[index] += delta;
                index += lowbit(index);
            }
        }
```

```
    /**
     * 查询前缀和[1, index]
     * @param index 查询的右边界（从1开始）
     * @return 前缀和
     */
    public int query(int index) {
        int sum = 0;
        // 沿树状数组向下累加所有相关节点的值
        while (index > 0) {
            sum += tree[index];
            index -= lowbit(index);
        }
        return sum;
    }

    /**
     * 查询区间和[left, right]
     * @param left 区间左边界（从1开始）
     * @param right 区间右边界（从1开始）
     * @return 区间和
     */
    public int queryRange(int left, int right) {
        if (left > right) {
            return 0;
        }
        return query(right) - query(left - 1);
    }
}

/**
 * 计算区间和的个数
 * @param nums 输入数组
 * @param lower 区间下界
 * @param upper 区间上界
 * @return 满足条件的子数组个数
 */
public static int countRangeSum(int[] nums, int lower, int upper) {
    // 边界条件检查
    if (nums == null || nums.length == 0) {
        return 0;
    }
```

```
        int n = nums.length;
        long[] prefixSums = new long[n + 1]; // 前缀和数组

        // 计算前缀和
        for (int i = 0; i < n; i++) {
            prefixSums[i + 1] = prefixSums[i] + nums[i];
        }

        // 离散化处理
        TreeSet<Long> valuesSet = new TreeSet<>();
        // 将所有可能需要查询的值加入集合
        for (long sum : prefixSums) {
            valuesSet.add(sum);
            valuesSet.add(sum - lower);
            valuesSet.add(sum - upper);
        }

        // 建立值到索引的映射
        Map<Long, Integer> valueToIndex = new HashMap<>();
        int index = 1; // 索引从 1 开始
        for (long value : valuesSet) {
            valueToIndex.put(value, index++);
        }

        // 创建树状数组
        FenwickTree fenwickTree = new FenwickTree(valuesSet.size());
        int count = 0;

        // 从前向后遍历前缀和数组
        for (long prefixSum : prefixSums) {
            // 查询满足条件的前缀和的数量：prefixSum[j] - upper <= prefixSum[i] <= prefixSum[j] -
lower
            int leftIndex = valueToIndex.get(prefixSum - upper);
            int rightIndex = valueToIndex.get(prefixSum - lower);
            count += fenwickTree.queryRange(leftIndex, rightIndex);

            // 将当前前缀和加入树状数组
            int currentIndex = valueToIndex.get(prefixSum);
            fenwickTree.update(currentIndex, 1);
        }

        return count;
    }
```

```
/**
 * 暴力解法（仅供比较，时间复杂度较高）
 * 时间复杂度: O(n²)
 * 空间复杂度: O(n)
 * @param nums 输入数组
 * @param lower 区间下界
 * @param upper 区间上界
 * @return 满足条件的子数组个数
 */
public static int countRangeSumBruteForce(int[] nums, int lower, int upper) {
    if (nums == null || nums.length == 0) {
        return 0;
    }

    int n = nums.length;
    long[] prefixSums = new long[n + 1];

    // 计算前缀和
    for (int i = 0; i < n; i++) {
        prefixSums[i + 1] = prefixSums[i] + nums[i];
    }

    int count = 0;
    // 暴力枚举所有可能的子数组
    for (int i = 0; i < n; i++) {
        for (int j = i + 1; j <= n; j++) {
            long rangeSum = prefixSums[j] - prefixSums[i];
            if (rangeSum >= lower && rangeSum <= upper) {
                count++;
            }
        }
    }

    return count;
}

/**
 * 归并排序解法（另一种O(n log n)的解法）
 * @param nums 输入数组
 * @param lower 区间下界
 * @param upper 区间上界
 * @return 满足条件的子数组个数
```

```
     */
    public static int countRangeSumMergeSort(int[] nums, int lower, int upper) {
        if (nums == null || nums.length == 0) {
            return 0;
        }

        int n = nums.length;
        long[] prefixSums = new long[n + 1];

        // 计算前缀和
        for (int i = 0; i < n; i++) {
            prefixSums[i + 1] = prefixSums[i] + nums[i];
        }

        // 归并排序过程中统计满足条件的子数组个数
        return mergeSortAndCount(prefixSums, 0, n, lower, upper);
    }

    /**
     * 归并排序并统计满足条件的子数组个数
     * @param prefixSums 前缀和数组
     * @param left 左边界
     * @param right 右边界
     * @param lower 区间下界
     * @param upper 区间上界
     * @return 满足条件的子数组个数
     */
    private static int mergeSortAndCount(long[] prefixSums, int left, int right, int lower, int
upper) {
        if (left >= right) {
            return 0;
        }

        int mid = left + (right - left) / 2;
        // 递归处理左右两部分
        int count = mergeSortAndCount(prefixSums, left, mid, lower, upper) +
                    mergeSortAndCount(prefixSums, mid + 1, right, lower, upper);

        // 统计满足条件的子数组个数
        int j = mid + 1, k = mid + 1;
        for (int i = left; i <= mid; i++) {
            // 找到最小的 j，使得 prefixSums[j] - prefixSums[i] >= lower
            while (j <= right && prefixSums[j] - prefixSums[i] < lower) {
```

```
            j++;
        }
        // 找到最大的k，使得prefixSums[k] - prefixSums[i] <= upper
        while (k <= right && prefixSums[k] - prefixSums[i] <= upper) {
            k++;
        }
        // 区间[j, k-1]内的所有前缀和都满足条件
        count += k - j;
    }

    // 合并两个有序数组
    merge(prefixSums, left, mid, right);

    return count;
}

/**
 * 合并两个有序数组
 * @param prefixSums 前缀和数组
 * @param left 左边界
 * @param mid 中间点
 * @param right 右边界
 */
private static void merge(long[] prefixSums, int left, int mid, int right) {
    long[] temp = new long[right - left + 1];
    int i = left, j = mid + 1, k = 0;

    // 合并两个有序数组
    while (i <= mid && j <= right) {
        if (prefixSums[i] <= prefixSums[j]) {
            temp[k++] = prefixSums[i++];
        } else {
            temp[k++] = prefixSums[j++];
        }
    }

    // 处理剩余元素
    while (i <= mid) {
        temp[k++] = prefixSums[i++];
    }

    while (j <= right) {
        temp[k++] = prefixSums[j++];
```

```
        }

        // 将临时数组复制回原数组
        System.arraycopy(temp, 0, prefixSums, left, temp.length);
    }

    /**
     * 打印数组的辅助方法
     * @param arr 输入数组
     */
    private static void printArray(int[] arr) {
        System.out.print("[");
        for (int i = 0; i < arr.length; i++) {
            System.out.print(arr[i]);
            if (i < arr.length - 1) {
                System.out.print(", ");
            }
        }
        System.out.println("]");
    }

    /**
     * 测试函数
     */
    public static void main(String[] args) {
        // 测试用例 1
        int[] nums1 = {-2, 5, -1};
        int lower1 = -2;
        int upper1 = 2;
        System.out.println("测试用例 1:");
        System.out.print("输入数组: ");
        printArray(nums1);
        System.out.println("lower: " + lower1 + ", upper: " + upper1);
        System.out.println("树状数组解法结果: " + countRangeSum(nums1, lower1, upper1));  // 期望输出: 3
        System.out.println("归并排序解法结果: " + countRangeSumMergeSort(nums1, lower1, upper1));  // 期望输出: 3
        System.out.println("暴力解法结果: " + countRangeSumBruteForce(nums1, lower1, upper1));  // 期望输出: 3

        // 测试用例 2
        int[] nums2 = {0};
        int lower2 = 0;
```

```
        int upper2 = 0;
        System.out.println("\n 测试用例 2:");
        System.out.print("输入数组: ");
        printArray(nums2);
        System.out.println("lower: " + lower2 + ", upper: " + upper2);
        System.out.println("树状数组解法结果: " + countRangeSum(nums2, lower2, upper2));  // 期望
输出: 1

        // 测试用例 3 - 空数组
        int[] nums3 = {};
        int lower3 = 0;
        int upper3 = 0;
        System.out.println("\n 测试用例 3:");
        System.out.print("输入数组: ");
        printArray(nums3);
        System.out.println("lower: " + lower3 + ", upper: " + upper3);
        System.out.println("树状数组解法结果: " + countRangeSum(nums3, lower3, upper3));  // 期望
输出: 0

        // 测试用例 4 - 大规模数据
        int size = 1000;
        int[] nums4 = new int[size];
        for (int i = 0; i < size; i++) {
            nums4[i] = (i % 3 == 0) ? -1 : (i % 3 == 1) ? 0 : 1;
        }
        int lower4 = -2;
        int upper4 = 2;
        System.out.println("\n 测试用例 4 (大规模数据):");
        System.out.println("数组长度: " + size);
        System.out.println("lower: " + lower4 + ", upper: " + upper4);

        // 测量树状数组解法的时间
        long startTime1 = System.currentTimeMillis();
        int result1 = countRangeSum(nums4, lower4, upper4);
        long endTime1 = System.currentTimeMillis();
        System.out.println("树状数组解法结果: " + result1);
        System.out.println("树状数组解法耗时: " + (endTime1 - startTime1) + "ms");

        // 测量归并排序解法的时间
        long startTime2 = System.currentTimeMillis();
        int result2 = countRangeSumMergeSort(nums4, lower4, upper4);
        long endTime2 = System.currentTimeMillis();
        System.out.println("归并排序解法结果: " + result2);
```

```
        System.out.println("归并排序解法耗时: " + (endTime2 - startTime2) + "ms");

        // 验证两种方法结果是否一致
        System.out.println("两种方法结果一致: " + (result1 == result2));

        // 对比暴力解法（仅在小规模数据上测试）
        if (size <= 1000) {
            int smallSize = Math.min(size, 300);  // 限制暴力解法的数组大小，避免超时
            int[] smallNums = Arrays.copyOf(nums4, smallSize);

            long startTime3 = System.currentTimeMillis();
            int result3 = countRangeSumBruteForce(smallNums, lower4, upper4);
            long endTime3 = System.currentTimeMillis();
            System.out.println("暴力解法(前" + smallSize + "个元素)结果: " + result3);
            System.out.println("暴力解法耗时: " + (endTime3 - startTime3) + "ms");

            // 验证暴力解法与树状数组解法在前 smallSize 个元素上是否一致
            int result4 = countRangeSum(smallNums, lower4, upper4);
            System.out.println("暴力解法与树状数组解法结果一致: " + (result3 == result4));
        }
    }
}
```

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

文件: Code13_RangeSumQueryCount.py

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

```
# LeetCode 327. 区间和的个数
# 给定一个整数数组 nums 以及两个整数 lower 和 upper，求区间 [lower, upper] 内的区间和的个数。
# 测试链接: https://leetcode.cn/problems/count-of-range-sum/

from typing import List

class Solution:
    """
    区间和的个数 - 树状数组解法

    解题思路:
    1. 题目要求计算数组中区间和位于 [lower, upper] 范围内的子数组个数
    2. 利用前缀和思想：区间和 sum(i,j) = prefix[j+1] - prefix[i]
    3. 问题转化为：对于每个 j，计算有多少个 i < j 满足 lower <= prefix[j] - prefix[i] <= upper
    4. 进一步转化为：对于每个 j，统计 prefix[i] 的范围为 [prefix[j] - upper, prefix[j] - lower] 的 i 的个数
```

5. 使用树状数组可以高效地统计这个范围查询
6. 由于前缀和可能很大，需要进行离散化处理

时间复杂度分析:
- 计算前缀和: O(n)
- 离散化: O(n log n)
- 构建和操作树状数组: O(n log n)
- 总时间复杂度: O(n log n)

空间复杂度分析:
- 前缀和数组: O(n)
- 离散化数组和映射: O(n)
- 树状数组: O(n)
- 总空间复杂度: O(n)

工程化考量:
1. 离散化处理：由于前缀和可能超出整数范围，进行离散化
2. 边界条件处理：处理空数组、单元素数组等特殊情况
3. 数据类型溢出：Python 自动处理大数，无需特别关注
4. 异常输入检查：验证输入的有效性
5. 代码可读性：使用清晰的变量命名和详细的注释
"""

```
class FenwickTree:
    """
    树状数组类
    用于高效计算前缀和和单点更新
    """
    def __init__(self, size: int):
        """
        初始化树状数组
        :param size: 数组大小
        """
        self.n = size
        self.tree = [0] * (size + 1)  # 树状数组下标从 1 开始

    def lowbit(self, x: int) -> int:
        """
        lowbit 操作，获取 x 的二进制表示中最低位的 1 所代表的值
        :param x: 输入整数
        :return: x & (-x)
        """
        return x & (-x)
```

```
    def update(self, index: int, delta: int) -> None:
        """
        在指定位置增加 delta
        :param index: 索引位置（从 1 开始）
        :param delta: 增加的值
        """
        # 沿树状数组向上更新所有相关节点
        while index <= self.n:
            self.tree[index] += delta
            index += self.lowbit(index)

    def query(self, index: int) -> int:
        """
        查询前缀和[1, index]
        :param index: 查询的右边界（从 1 开始）
        :return: 前缀和
        """
        sum_val = 0
        # 沿树状数组向下累加所有相关节点的值
        while index > 0:
            sum_val += self.tree[index]
            index -= self.lowbit(index)
        return sum_val

    def query_range(self, left: int, right: int) -> int:
        """
        查询区间和[left, right]
        :param left: 区间左边界（从 1 开始）
        :param right: 区间右边界（从 1 开始）
        :return: 区间和
        """
        if left > right:
            return 0
        return self.query(right) - self.query(left - 1)

def countRangeSum(self, nums: List[int], lower: int, upper: int) -> int:
    """
    计算区间和的个数 - 树状数组解法
    :param nums: 输入数组
    :param lower: 区间下界
    :param upper: 区间上界
    :return: 满足条件的子数组个数
```

```
        """
        # 边界条件检查
        if not nums:
            return 0

        n = len(nums)
        prefix_sums = [0] * (n + 1)  # 前缀和数组

        # 计算前缀和
        for i in range(n):
            prefix_sums[i + 1] = prefix_sums[i] + nums[i]

        # 离散化处理
        # 收集所有可能需要查询的值
        values_set = set()
        for sum_val in prefix_sums:
            values_set.add(sum_val)
            values_set.add(sum_val - lower)
            values_set.add(sum_val - upper)

        # 排序并去重
        sorted_values = sorted(values_set)

        # 建立值到索引的映射
        value_to_index = {value: i + 1 for i, value in enumerate(sorted_values)}  # 索引从1开始

        # 创建树状数组
        fenwick_tree = self.FenwickTree(len(sorted_values))
        count = 0

        # 从前向后遍历前缀和数组
        for prefix_sum in prefix_sums:
            # 查询满足条件的前缀和的数量：prefixSum[j] - upper <= prefixSum[i] <= prefixSum[j] -
lower
            left_index = value_to_index[prefix_sum - upper]
            right_index = value_to_index[prefix_sum - lower]
            count += fenwick_tree.query_range(left_index, right_index)

            # 将当前前缀和加入树状数组
            current_index = value_to_index[prefix_sum]
            fenwick_tree.update(current_index, 1)

        return count
```

```
def countRangeSumBruteForce(self, nums: List[int], lower: int, upper: int) -> int:
    """
    暴力解法（仅供比较，时间复杂度较高）
    时间复杂度: O(n²)
    空间复杂度: O(n)
    :param nums: 输入数组
    :param lower: 区间下界
    :param upper: 区间上界
    :return: 满足条件的子数组个数
    """
    if not nums:
        return 0

    n = len(nums)
    prefix_sums = [0] * (n + 1)

    # 计算前缀和
    for i in range(n):
        prefix_sums[i + 1] = prefix_sums[i] + nums[i]

    count = 0
    # 暴力枚举所有可能的子数组
    for i in range(n):
        for j in range(i + 1, n + 1):
            range_sum = prefix_sums[j] - prefix_sums[i]
            if lower <= range_sum <= upper:
                count += 1

    return count

def countRangeSumMergeSort(self, nums: List[int], lower: int, upper: int) -> int:
    """
    归并排序解法（另一种O(n log n)的解法）
    :param nums: 输入数组
    :param lower: 区间下界
    :param upper: 区间上界
    :return: 满足条件的子数组个数
    """
    if not nums:
        return 0

    n = len(nums)
```

```
        prefix_sums = [0] * (n + 1)

        # 计算前缀和
        for i in range(n):
            prefix_sums[i + 1] = prefix_sums[i] + nums[i]

        # 归并排序过程中统计满足条件的子数组个数
        return self._mergeSortAndCount(prefix_sums, 0, n, lower, upper)

    def _mergeSortAndCount(self, prefix_sums: List[int], left: int, right: int, lower: int,
upper: int) -> int:
        """
        归并排序并统计满足条件的子数组个数
        :param prefix_sums: 前缀和数组
        :param left: 左边界
        :param right: 右边界
        :param lower: 区间下界
        :param upper: 区间上界
        :return: 满足条件的子数组个数
        """
        if left >= right:
            return 0

        mid = left + (right - left) // 2
        # 递归处理左右两部分
        count = self._mergeSortAndCount(prefix_sums, left, mid, lower, upper) + \
                self._mergeSortAndCount(prefix_sums, mid + 1, right, lower, upper)

        # 统计满足条件的子数组个数
        j = mid + 1
        k = mid + 1
        for i in range(left, mid + 1):
            # 找到最小的 j，使得 prefix_sums[j] - prefix_sums[i] >= lower
            while j <= right and prefix_sums[j] - prefix_sums[i] < lower:
                j += 1
            # 找到最大的 k，使得 prefix_sums[k] - prefix_sums[i] <= upper
            while k <= right and prefix_sums[k] - prefix_sums[i] <= upper:
                k += 1
            # 区间[j, k-1]内的所有前缀和都满足条件
            count += k - j

        # 合并两个有序数组
        self._merge(prefix_sums, left, mid, right)
```

```
        return count

    def _merge(self, prefix_sums: List[int], left: int, mid: int, right: int) -> None:
        """
        合并两个有序数组
        :param prefix_sums: 前缀和数组
        :param left: 左边界
        :param mid: 中间点
        :param right: 右边界
        """
        temp = []
        i = left
        j = mid + 1

        # 合并两个有序数组
        while i <= mid and j <= right:
            if prefix_sums[i] <= prefix_sums[j]:
                temp.append(prefix_sums[i])
                i += 1
            else:
                temp.append(prefix_sums[j])
                j += 1

        # 处理剩余元素
        while i <= mid:
            temp.append(prefix_sums[i])
            i += 1

        while j <= right:
            temp.append(prefix_sums[j])
            j += 1

        # 将临时数组复制回原数组
        for k in range(len(temp)):
            prefix_sums[left + k] = temp[k]

# 测试函数
def test_solution():
    solution = Solution()

    # 测试用例 1
    nums1 = [-2, 5, -1]
```

```
    lower1 = -2
    upper1 = 2
    print("测试用例 1:")
    print(f"输入数组: {nums1}")
    print(f"lower: {lower1}, upper: {upper1}")
    print(f"树状数组解法结果: {solution.countRangeSum(nums1, lower1, upper1)}")  # 期望输出: 3
    print(f"归并排序解法结果: {solution.countRangeSumMergeSort(nums1, lower1, upper1)}")  # 期望输出: 3
    print(f"暴力解法结果: {solution.countRangeSumBruteForce(nums1, lower1, upper1)}")  # 期望输出: 3

    # 测试用例 2
    nums2 = [0]
    lower2 = 0
    upper2 = 0
    print("\n测试用例 2:")
    print(f"输入数组: {nums2}")
    print(f"lower: {lower2}, upper: {upper2}")
    print(f"树状数组解法结果: {solution.countRangeSum(nums2, lower2, upper2)}")  # 期望输出: 1

    # 测试用例 3 - 空数组
    nums3 = []
    lower3 = 0
    upper3 = 0
    print("\n测试用例 3:")
    print(f"输入数组: {nums3}")
    print(f"lower: {lower3}, upper: {upper3}")
    print(f"树状数组解法结果: {solution.countRangeSum(nums3, lower3, upper3)}")  # 期望输出: 0

    # 测试用例 4 - 大规模数据
    size = 1000
    nums4 = [(i % 3 == 0) and -1 or (i % 3 == 1) and 0 or 1 for i in range(size)]
    lower4 = -2
    upper4 = 2
    print("\n测试用例 4 (大规模数据):")
    print(f"数组长度: {size}")
    print(f"lower: {lower4}, upper: {upper4}")

    import time

    # 测量树状数组解法的时间
    start_time1 = time.time()
    result1 = solution.countRangeSum(nums4, lower4, upper4)
```

```
    end_time1 = time.time()
    print(f"树状数组解法结果: {result1}")
    print(f"树状数组解法耗时: {(end_time1 - start_time1) * 1000:.2f}ms")

    # 测量归并排序解法的时间
    start_time2 = time.time()
    result2 = solution.countRangeSumMergeSort(nums4, lower4, upper4)
    end_time2 = time.time()
    print(f"归并排序解法结果: {result2}")
    print(f"归并排序解法耗时: {(end_time2 - start_time2) * 1000:.2f}ms")

    # 验证两种方法结果是否一致
    print(f"两种方法结果一致: {result1 == result2}")

    # 对比暴力解法（仅在小规模数据上测试）
    if size <= 1000:
        small_size = min(size, 300)  # 限制暴力解法的数组大小，避免超时
        small_nums = nums4[:small_size]

        start_time3 = time.time()
        result3 = solution.countRangeSumBruteForce(small_nums, lower4, upper4)
        end_time3 = time.time()
        print(f"暴力解法(前{small_size}个元素)结果: {result3}")
        print(f"暴力解法耗时: {(end_time3 - start_time3) * 1000:.2f}ms")

        # 验证暴力解法与树状数组解法在前 small_size 个元素上是否一致
        result4 = solution.countRangeSum(small_nums, lower4, upper4)
        print(f"暴力解法与树状数组解法结果一致: {result3 == result4}")

# 运行测试
if __name__ == "__main__":
    test_solution()

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

文件: Code14_SimpleProblemWithIntegers.cpp
==============================================
// POJ 3468 A Simple Problem with Integers
// 题目描述：给定一个长度为 N 的整数序列，执行以下操作：
// 1. C a b c: 将区间 [a,b] 中的每个数都加上 c
// 2. Q a b: 查询区间 [a,b] 中所有数的和
// 题目链接：http://poj.org/problem?id=3468
// 解题思路：使用线段树 + 懒惰标记实现区间加法和区间求和查询
```

```
#include <iostream>
#include <vector>
#include <string>
using namespace std;

/**
 * 线段树实现区间加法和区间求和查询
 * 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) - 线段树数组大小为 4n
 */
class Code14_SimpleProblemWithIntegers {
private:
    vector<long long> tree_sum;  // 存储区间和的线段树数组
    vector<long long> tree_add;  // 存储懒惰标记的数组
    vector<long long> arr;       // 原始数组
    int n;                       // 数组长度

    /**
     * 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
     */
    void _build(int node, int start, int end) {
        if (start == end) {
            // 叶子节点，直接赋值
            tree_sum[node] = arr[start];
            tree_add[node] = 0;
            return;
        }

        int mid = start + (end - start) / 2;
        int left_node = 2 * node + 1;
        int right_node = 2 * node + 2;

        // 递归构建左右子树
        _build(left_node, start, mid);
        _build(right_node, mid + 1, end);
```

```
    // 合并左右子树信息
    tree_sum[node] = tree_sum[left_node] + tree_sum[right_node];
    // 非叶子节点初始懒惰标记为0
    tree_add[node] = 0;
}

/**
 * 下传懒惰标记
 * @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
 */
void _push_down(int node, int start, int end) {
    if (tree_add[node] != 0) {
        // 只有当懒惰标记不为0时需要下传
        int left_node = 2 * node + 1;
        int right_node = 2 * node + 2;
        int mid = start + (end - start) / 2;

        // 更新左子节点的区间和和懒惰标记
        tree_sum[left_node] += tree_add[node] * (mid - start + 1);
        tree_add[left_node] += tree_add[node];

        // 更新右子节点的区间和和懒惰标记
        tree_sum[right_node] += tree_add[node] * (end - mid);
        tree_add[right_node] += tree_add[node];

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

/**
 * 区间更新（加法）
 * @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 需要更新的区间左边界
 * @param r 需要更新的区间右边界
 * @param val 要增加的值
 */
void _update_range(int node, int start, int end, int l, int r, long long val) {
    // 当前区间与目标区间无交集
```

```
    if (start > r || end < l) {
        return;
    }

    // 当前区间完全包含在目标区间内
    if (start >= l && end <= r) {
        // 更新区间和
        tree_sum[node] += val * (end - start + 1);
        // 更新懒惰标记
        tree_add[node] += val;
        return;
    }

    // 下传懒惰标记到子节点
    _push_down(node, start, end);

    int mid = start + (end - start) / 2;
    int left_node = 2 * node + 1;
    int right_node = 2 * node + 2;

    // 递归更新左右子树
    _update_range(left_node, start, mid, l, r, val);
    _update_range(right_node, mid + 1, end, l, r, val);

    // 更新当前节点的区间和
    tree_sum[node] = tree_sum[left_node] + tree_sum[right_node];
}

/**
 * 区间查询
 * @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 查询的区间左边界
 * @param r 查询的区间右边界
 * @return 查询区间的和
 */
long long _query_range(int node, int start, int end, int l, int r) {
    // 当前区间与查询区间无交集
    if (start > r || end < l) {
        return 0;
    }
```

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

        // 下传懒惰标记到子节点
        _push_down(node, start, end);

        int mid = start + (end - start) / 2;
        int left_node = 2 * node + 1;
        int right_node = 2 * node + 2;

        // 递归查询左右子树
        long long left_sum = _query_range(left_node, start, mid, l, r);
        long long right_sum = _query_range(right_node, mid + 1, end, l, r);

        // 返回左右子树查询结果的和
        return left_sum + right_sum;
    }

public:
    /**
     * 构造函数
     * @param nums 输入数组
     */
    Code14_SimpleProblemWithIntegers(const vector<long long>& nums) {
        this->n = nums.size();
        this->arr = nums;
        // 线段树数组大小为4n，保证足够空间
        this->tree_sum.resize(4 * n, 0);
        this->tree_add.resize(4 * n, 0);
        // 构建线段树
        _build(0, 0, n - 1);
    }

    /**
     * 公共接口：区间更新
     * @param l 区间左边界（注意：这里是1-based索引）
     * @param r 区间右边界（注意：这里是1-based索引）
     * @param val 要增加的值
     */
    void update_range(int l, int r, long long val) {
        // 转换为0-based索引
```

```
        _update_range(0, 0, n - 1, l - 1, r - 1, val);
    }

    /**
     * 公共接口：区间查询
     * @param l 区间左边界（注意：这里是1-based索引）
     * @param r 区间右边界（注意：这里是1-based索引）
     * @return 查询区间的和
     */
    long long query_range(int l, int r) {
        // 转换为0-based索引
        return _query_range(0, 0, n - 1, l - 1, r - 1);
    }
};

/**
 * 主方法，用于处理输入输出
 */
int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);

    int n, q;
    cin >> n >> q;

    vector<long long> arr(n);
    for (int i = 0; i < n; ++i) {
        cin >> arr[i];
    }

    Code14_SimpleProblemWithIntegers solution(arr);

    while (q--) {
        char op;
        int a, b;
        cin >> op >> a >> b;

        if (op == 'Q') {
            // 查询操作
            cout << solution.query_range(a, b) << '\n';
        } else if (op == 'C') {
            // 更新操作
            long long c;
```

```
            cin >> c;
            solution.update_range(a, b, c);
        }
    }

    return 0;
}

/**
 * 测试方法
 * 注意：在 POJ 上提交时应注释掉此函数，仅保留 main 函数
 */
void test() {
    // 测试用例 1：基本操作测试
    vector<long long> nums1 = {1, 2, 3, 4, 5};
    Code14_SimpleProblemWithIntegers solution1(nums1);
    cout << "测试用例 1:\n";
    cout << "初始数组: [1, 2, 3, 4, 5]\n";
    cout << "查询区间[1, 5]的和: " << solution1.query_range(1, 5) << endl;  // 应为 15
    solution1.update_range(2, 4, 2);
    cout << "更新区间[2, 4]每个元素加 2 后，数组变为: [1, 4, 5, 6, 5]\n";
    cout << "查询区间[1, 5]的和: " << solution1.query_range(1, 5) << endl;  // 应为 21
    cout << "查询区间[2, 4]的和: " << solution1.query_range(2, 4) << endl;  // 应为 15

    // 测试用例 2：边界情况测试
    vector<long long> nums2 = {10};
    Code14_SimpleProblemWithIntegers solution2(nums2);
    cout << "\n 测试用例 2:\n";
    cout << "初始数组: [10]\n";
    cout << "查询区间[1, 1]的和: " << solution2.query_range(1, 1) << endl;  // 应为 10
    solution2.update_range(1, 1, -5);
    cout << "更新区间[1, 1]每个元素加-5 后，数组变为: [5]\n";
    cout << "查询区间[1, 1]的和: " << solution2.query_range(1, 1) << endl;  // 应为 5

    // 测试用例 3：多次更新和查询测试
    vector<long long> nums3 = {0, 0, 0, 0, 0};
    Code14_SimpleProblemWithIntegers solution3(nums3);
    cout << "\n 测试用例 3:\n";
    cout << "初始数组: [0, 0, 0, 0, 0]\n";
    solution3.update_range(1, 5, 1);  // 所有元素加 1
    solution3.update_range(2, 4, 2);  // 中间三个元素再加 2
    solution3.update_range(3, 3, 3);  // 中间元素再加 3
    cout << "多次更新后，数组变为: [1, 3, 6, 3, 1]\n";
```

```
    cout << "查询区间[1, 5]的和: " << solution3.query_range(1, 5) << endl;  // 应为14
    cout << "查询区间[1, 3]的和: " << solution3.query_range(1, 3) << endl;  // 应为10
    cout << "查询区间[3, 5]的和: " << solution3.query_range(3, 5) << endl;  // 应为10
}

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

文件: Code14_SimpleProblemWithIntegers.java
==========================================
// POJ 3468 A Simple Problem with Integers
// 题目描述：给定一个长度为N的整数序列，执行以下操作：
// 1. C a b c: 将区间 [a,b] 中的每个数都加上 c
// 2. Q a b: 查询区间 [a,b] 中所有数的和
// 题目链接：http://poj.org/problem?id=3468
// 解题思路：使用线段树 + 懒惰标记实现区间加法和区间求和查询

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

/**
 * 线段树实现区间加法和区间求和查询
 * 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) - 线段树数组大小为4n
 */
public class Code14_SimpleProblemWithIntegers {

    /**
     * 线段树节点类
     */
    static class Node {
        long sum;    // 区间和
        long add;    // 懒惰标记，表示区间每个元素需要增加的值
    }

    private Node[] tree;  // 线段树数组
    private int[] arr;    // 原始数组
    private int n;        // 数组长度

    /**
     * 构造函数
```

```
     * @param nums 输入数组
     */
    public Code14_SimpleProblemWithIntegers(int[] nums) {
        this.n = nums.length;
        this.arr = Arrays.copyOf(nums, n);
        // 线段树数组大小为4n，保证足够空间
        this.tree = new Node[4 * n];
        for (int i = 0; i < 4 * n; i++) {
            tree[i] = new Node();
        }
        // 构建线段树
        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，直接赋值
            tree[node].sum = arr[start];
            tree[node].add = 0;
            return;
        }

        int mid = start + (end - start) / 2;
        int leftNode = 2 * node + 1;
        int rightNode = 2 * node + 2;

        // 递归构建左右子树
        build(leftNode, start, mid);
        build(rightNode, mid + 1, end);

        // 合并左右子树信息
        tree[node].sum = tree[leftNode].sum + tree[rightNode].sum;
        tree[node].add = 0;  // 非叶子节点初始懒惰标记为0
    }

    /**
     * 下传懒惰标记
```

```
     * @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
     */
    private void pushDown(int node, int start, int end) {
        if (tree[node].add != 0) {
            // 只有当懒惰标记不为0时需要下传
            int leftNode = 2 * node + 1;
            int rightNode = 2 * node + 2;
            int mid = start + (end - start) / 2;

            // 更新左子节点的区间和和懒惰标记
            tree[leftNode].sum += tree[node].add * (mid - start + 1);
            tree[leftNode].add += tree[node].add;

            // 更新右子节点的区间和和懒惰标记
            tree[rightNode].sum += tree[node].add * (end - mid);
            tree[rightNode].add += tree[node].add;

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

    /**
     * 区间更新（加法）
     * @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 需要更新的区间左边界
     * @param r 需要更新的区间右边界
     * @param val 要增加的值
     */
    private void updateRange(int node, int start, int end, int l, int r, long val) {
        // 当前区间与目标区间无交集
        if (start > r || end < l) {
            return;
        }

        // 当前区间完全包含在目标区间内
        if (start >= l && end <= r) {
            // 更新区间和
            tree[node].sum += val * (end - start + 1);
```

```
            // 更新懒惰标记
            tree[node].add += val;
            return;
        }

        // 下传懒惰标记到子节点
        pushDown(node, start, end);

        int mid = start + (end - start) / 2;
        int leftNode = 2 * node + 1;
        int rightNode = 2 * node + 2;

        // 递归更新左右子树
        updateRange(leftNode, start, mid, l, r, val);
        updateRange(rightNode, mid + 1, end, l, r, val);

        // 更新当前节点的区间和
        tree[node].sum = tree[leftNode].sum + tree[rightNode].sum;
    }

    /**
     * 区间查询
     * @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 查询的区间左边界
     * @param r 查询的区间右边界
     * @return 查询区间的和
     */
    private long queryRange(int node, int start, int end, int l, int r) {
        // 当前区间与查询区间无交集
        if (start > r || end < l) {
            return 0;
        }

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

        // 下传懒惰标记到子节点
        pushDown(node, start, end);
```

```
        int mid = start + (end - start) / 2;
        int leftNode = 2 * node + 1;
        int rightNode = 2 * node + 2;

        // 递归查询左右子树
        long leftSum = queryRange(leftNode, start, mid, l, r);
        long rightSum = queryRange(rightNode, mid + 1, end, l, r);

        // 返回左右子树查询结果的和
        return leftSum + rightSum;
    }

    /**
     * 公共接口：区间更新
     * @param l 区间左边界（注意：这里是1-based索引）
     * @param r 区间右边界（注意：这里是1-based索引）
     * @param val 要增加的值
     */
    public void updateRange(int l, int r, long val) {
        // 转换为0-based索引
        updateRange(0, 0, n - 1, l - 1, r - 1, val);
    }

    /**
     * 公共接口：区间查询
     * @param l 区间左边界（注意：这里是1-based索引）
     * @param r 区间右边界（注意：这里是1-based索引）
     * @return 查询区间的和
     */
    public long queryRange(int l, int r) {
        // 转换为0-based索引
        return queryRange(0, 0, n - 1, l - 1, r - 1);
    }

    /**
     * 主方法，用于处理输入输出
     */
    public static void main(String[] args) throws IOException {
        // 使用快速IO
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));

        // 读取输入
```

```
        StringTokenizer st = new StringTokenizer(br.readLine());
        int n = Integer.parseInt(st.nextToken());  // 数组长度
        int q = Integer.parseInt(st.nextToken());  // 查询次数

        // 读取数组元素
        st = new StringTokenizer(br.readLine());
        int[] arr = new int[n];
        for (int i = 0; i < n; i++) {
            arr[i] = Integer.parseInt(st.nextToken());
        }

        // 创建线段树
        Code14_SimpleProblemWithIntegers solution = new Code14_SimpleProblemWithIntegers(arr);

        // 处理每个查询
        while (q-- > 0) {
            st = new StringTokenizer(br.readLine());
            String op = st.nextToken();
            int a = Integer.parseInt(st.nextToken());
            int b = Integer.parseInt(st.nextToken());

            if (op.equals("Q")) {
                // 查询操作
                long sum = solution.queryRange(a, b);
                out.println(sum);
            } else if (op.equals("C")) {
                // 更新操作
                long c = Long.parseLong(st.nextToken());
                solution.updateRange(a, b, c);
            }
        }

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

    /**
     * 测试方法
     */
    public static void test() {
        // 测试用例 1：基本操作测试
        int[] nums1 = {1, 2, 3, 4, 5};
```

```
        Code14_SimpleProblemWithIntegers solution1 = new Code14_SimpleProblemWithIntegers(nums1);
        System.out.println("测试用例 1:");
        System.out.println("初始数组: [1, 2, 3, 4, 5]");
        System.out.println("查询区间[1, 5]的和: " + solution1.queryRange(1, 5));  // 应为 15
        solution1.updateRange(2, 4, 2);
        System.out.println("更新区间[2, 4]每个元素加 2 后，数组变为: [1, 4, 5, 6, 5]");
        System.out.println("查询区间[1, 5]的和: " + solution1.queryRange(1, 5));  // 应为 21
        System.out.println("查询区间[2, 4]的和: " + solution1.queryRange(2, 4));  // 应为 15

        // 测试用例 2：边界情况测试
        int[] nums2 = {10};
        Code14_SimpleProblemWithIntegers solution2 = new Code14_SimpleProblemWithIntegers(nums2);
        System.out.println("\n 测试用例 2:");
        System.out.println("初始数组: [10]");
        System.out.println("查询区间[1, 1]的和: " + solution2.queryRange(1, 1));  // 应为 10
        solution2.updateRange(1, 1, -5);
        System.out.println("更新区间[1, 1]每个元素加-5 后，数组变为: [5]");
        System.out.println("查询区间[1, 1]的和: " + solution2.queryRange(1, 1));  // 应为 5

        // 测试用例 3：多次更新和查询测试
        int[] nums3 = {0, 0, 0, 0, 0};
        Code14_SimpleProblemWithIntegers solution3 = new Code14_SimpleProblemWithIntegers(nums3);
        System.out.println("\n 测试用例 3:");
        System.out.println("初始数组: [0, 0, 0, 0, 0]");
        solution3.updateRange(1, 5, 1);  // 所有元素加 1
        solution3.updateRange(2, 4, 2);  // 中间三个元素再加 2
        solution3.updateRange(3, 3, 3);  // 中间元素再加 3
        System.out.println("多次更新后，数组变为: [1, 3, 6, 3, 1]");
        System.out.println("查询区间[1, 5]的和: " + solution3.queryRange(1, 5));  // 应为 14
        System.out.println("查询区间[1, 3]的和: " + solution3.queryRange(1, 3));  // 应为 10
        System.out.println("查询区间[3, 5]的和: " + solution3.queryRange(3, 5));  // 应为 10
    }
}
```

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

文件: Code14_SimpleProblemWithIntegers.py

```
==============================================
# POJ 3468 A Simple Problem with Integers
# 题目描述：给定一个长度为 N 的整数序列，执行以下操作：
# 1. C a b c: 将区间 [a,b] 中的每个数都加上 c
# 2. Q a b: 查询区间 [a,b] 中所有数的和
# 题目链接：http://poj.org/problem?id=3468
```

```
# 解题思路：使用线段树 + 懒惰标记实现区间加法和区间求和查询

class Code14_SimpleProblemWithIntegers:
    """
    线段树实现区间加法和区间求和查询

    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) - 线段树数组大小为 4n
    """

    def __init__(self, nums):
        """
        初始化线段树
        :param nums: 输入数组
        """
        self.n = len(nums)
        self.arr = nums.copy()
        # 线段树数组大小为 4n，保证足够空间
        # 使用两个数组分别存储区间和和懒惰标记
        self.tree_sum = [0] * (4 * self.n)
        self.tree_add = [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
        :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
        """
        if start == end:
            # 叶子节点，直接赋值
            self.tree_sum[node] = self.arr[start]
            self.tree_add[node] = 0
            return

        mid = start + (end - start) // 2
        left_node = 2 * node + 1
        right_node = 2 * node + 2
```

```
        # 递归构建左右子树
        self._build(left_node, start, mid)
        self._build(right_node, mid + 1, end)

        # 合并左右子树信息
        self.tree_sum[node] = self.tree_sum[left_node] + self.tree_sum[right_node]
        # 非叶子节点初始懒惰标记为0
        self.tree_add[node] = 0

    def _push_down(self, node, start, end):
        """
        下传懒惰标记
        :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
        """
        if self.tree_add[node] != 0:
            # 只有当懒惰标记不为0时需要下传
            left_node = 2 * node + 1
            right_node = 2 * node + 2
            mid = start + (end - start) // 2

            # 更新左子节点的区间和和懒惰标记
            self.tree_sum[left_node] += self.tree_add[node] * (mid - start + 1)
            self.tree_add[left_node] += self.tree_add[node]

            # 更新右子节点的区间和和懒惰标记
            self.tree_sum[right_node] += self.tree_add[node] * (end - mid)
            self.tree_add[right_node] += self.tree_add[node]

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

    def _update_range(self, node, start, end, l, r, val):
        """
        区间更新（加法）
        :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: 需要更新的区间左边界
        :param r: 需要更新的区间右边界
        :param val: 要增加的值
```

```
    """
    # 当前区间与目标区间无交集
    if start > r or end < l:
        return

    # 当前区间完全包含在目标区间内
    if start >= l and end <= r:
        # 更新区间和
        self.tree_sum[node] += val * (end - start + 1)
        # 更新懒惰标记
        self.tree_add[node] += val
        return

    # 下传懒惰标记到子节点
    self._push_down(node, start, end)

    mid = start + (end - start) // 2
    left_node = 2 * node + 1
    right_node = 2 * node + 2

    # 递归更新左右子树
    self._update_range(left_node, start, mid, l, r, val)
    self._update_range(right_node, mid + 1, end, l, r, val)

    # 更新当前节点的区间和
    self.tree_sum[node] = self.tree_sum[left_node] + self.tree_sum[right_node]

def _query_range(self, node, start, end, l, r):
    """
    区间查询
    :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: 查询的区间左边界
    :param r: 查询的区间右边界
    :return: 查询区间的和
    """
    # 当前区间与查询区间无交集
    if start > r or end < l:
        return 0

    # 当前区间完全包含在查询区间内
    if start >= l and end <= r:
```

```
            return self.tree_sum[node]

        # 下传懒惰标记到子节点
        self._push_down(node, start, end)

        mid = start + (end - start) // 2
        left_node = 2 * node + 1
        right_node = 2 * node + 2

        # 递归查询左右子树
        left_sum = self._query_range(left_node, start, mid, l, r)
        right_sum = self._query_range(right_node, mid + 1, end, l, r)

        # 返回左右子树查询结果的和
        return left_sum + right_sum

    def update_range(self, l, r, val):
        """
        公共接口：区间更新
        :param l: 区间左边界（注意：这里是1-based索引）
        :param r: 区间右边界（注意：这里是1-based索引）
        :param val: 要增加的值
        """
        # 转换为0-based索引
        self._update_range(0, 0, self.n - 1, l - 1, r - 1, val)

    def query_range(self, l, r):
        """
        公共接口：区间查询
        :param l: 区间左边界（注意：这里是1-based索引）
        :param r: 区间右边界（注意：这里是1-based索引）
        :return: 查询区间的和
        """
        # 转换为0-based索引
        return self._query_range(0, 0, self.n - 1, l - 1, r - 1)

# 主方法，用于处理输入输出
def main():
    import sys
    input = sys.stdin.read().split()
    ptr = 0

    # 读取数组长度和查询次数
```

```
    n = int(input[ptr])
    ptr += 1
    q = int(input[ptr])
    ptr += 1

    # 读取数组元素
    arr = list(map(int, input[ptr:ptr + n]))
    ptr += n

    # 创建线段树
    solution = Code14_SimpleProblemWithIntegers(arr)

    # 处理每个查询
    results = []
    while q > 0:
        q -= 1
        op = input[ptr]
        ptr += 1
        a = int(input[ptr])
        ptr += 1
        b = int(input[ptr])
        ptr += 1

        if op == 'Q':
            # 查询操作
            sum_val = solution.query_range(a, b)
            results.append(str(sum_val))
        elif op == 'C':
            # 更新操作
            c = int(input[ptr])
            ptr += 1
            solution.update_range(a, b, c)

    # 输出结果
    print('\n'.join(results))

# 测试方法
def test():
    # 测试用例 1：基本操作测试
    nums1 = [1, 2, 3, 4, 5]
    solution1 = Code14_SimpleProblemWithIntegers(nums1)
    print("测试用例 1:")
    print("初始数组: [1, 2, 3, 4, 5]")
```

```
    print(f"查询区间[1, 5]的和: {solution1.query_range(1, 5)}")  # 应为 15
    solution1.update_range(2, 4, 2)
    print("更新区间[2, 4]每个元素加 2 后，数组变为: [1, 4, 5, 6, 5]")
    print(f"查询区间[1, 5]的和: {solution1.query_range(1, 5)}")  # 应为 21
    print(f"查询区间[2, 4]的和: {solution1.query_range(2, 4)}")  # 应为 15

    # 测试用例 2：边界情况测试
    nums2 = [10]
    solution2 = Code14_SimpleProblemWithIntegers(nums2)
    print("\n 测试用例 2:")
    print("初始数组: [10]")
    print(f"查询区间[1, 1]的和: {solution2.query_range(1, 1)}")  # 应为 10
    solution2.update_range(1, 1, -5)
    print("更新区间[1, 1]每个元素加-5 后，数组变为: [5]")
    print(f"查询区间[1, 1]的和: {solution2.query_range(1, 1)}")  # 应为 5

    # 测试用例 3：多次更新和查询测试
    nums3 = [0, 0, 0, 0, 0]
    solution3 = Code14_SimpleProblemWithIntegers(nums3)
    print("\n 测试用例 3:")
    print("初始数组: [0, 0, 0, 0, 0]")
    solution3.update_range(1, 5, 1)  # 所有元素加 1
    solution3.update_range(2, 4, 2)  # 中间三个元素再加 2
    solution3.update_range(3, 3, 3)  # 中间元素再加 3
    print("多次更新后，数组变为: [1, 3, 6, 3, 1]")
    print(f"查询区间[1, 5]的和: {solution3.query_range(1, 5)}")  # 应为 14
    print(f"查询区间[1, 3]的和: {solution3.query_range(1, 3)}")  # 应为 10
    print(f"查询区间[3, 5]的和: {solution3.query_range(3, 5)}")  # 应为 10

# 如果直接运行此脚本，则执行测试
if __name__ == "__main__":
    # 运行测试
    test()

    # 如果需要读取标准输入运行，可以取消下面的注释
    # main()

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

文件: Code15_RangeSumQueryMutable.cpp
==========================================
// LeetCode 307. Range Sum Query - Mutable
// 题目描述：给定一个整数数组 nums，实现两个函数：
```

```
// 1. update(i, val)：将数组中索引 i 处的值更新为 val
// 2. sumRange(i, j)：返回数组中从索引 i 到 j 的元素之和
// 题目链接：https://leetcode.com/problems/range-sum-query-mutable/
// 解题思路：使用线段树或树状数组实现，本题同时提供两种实现方式

#include <iostream>
#include <vector>
#include <stdexcept>
#include <chrono>
using namespace std;
using namespace std::chrono;

/**
 * 使用线段树实现区间和查询与单点更新
 * 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)
 */
class Code15_RangeSumQueryMutable {
private:
    vector<int> tree; // 线段树数组
    int n;            // 原始数组长度

    /**
     * 构建线段树
     * @param nums 原始数组
     * @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
     */
    void buildTree(const vector<int>& nums, int node, int start, int end) {
        if (start == end) {
            // 叶子节点，直接赋值
            tree[node] = nums[start];
            return;
        }

        int mid = start + (end - start) / 2;
        int leftNode = 2 * node + 1;
        int rightNode = 2 * node + 2;
```

```
    // 递归构建左右子树
    buildTree(nums, leftNode, start, mid);
    buildTree(nums, rightNode, mid + 1, end);

    // 当前节点的值为左右子节点之和
    tree[node] = tree[leftNode] + tree[rightNode];
}

/**
 * 单点更新（内部方法）
 * @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 要更新的元素索引
 * @param val 新值
 */
void updateTree(int node, int start, int end, int index, int val) {
    if (start == end) {
        // 到达叶子节点，更新值
        tree[node] = val;
        return;
    }

    int mid = start + (end - start) / 2;
    int leftNode = 2 * node + 1;
    int rightNode = 2 * node + 2;

    // 根据 index 所在的区间决定递归左子树还是右子树
    if (index <= mid) {
        updateTree(leftNode, start, mid, index, val);
    } else {
        updateTree(rightNode, mid + 1, end, index, val);
    }

    // 更新当前节点的值
    tree[node] = tree[leftNode] + tree[rightNode];
}

/**
 * 区间查询（内部方法）
 * @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 查询区间的左边界
     * @param right 查询区间的右边界
     * @return 查询区间的和
     */
    int queryTree(int node, int start, int end, int left, int right) {
        // 查询区间与当前区间无交集
        if (right < start || left > end) {
            return 0;
        }

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

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

        int leftSum = queryTree(leftNode, start, mid, left, right);
        int rightSum = queryTree(rightNode, mid + 1, end, left, right);

        return leftSum + rightSum;
    }

public:
    /**
     * 构造函数
     * @param nums 输入数组
     */
    Code15_RangeSumQueryMutable(const vector<int>& nums) {
        this->n = nums.size();
        // 线段树数组大小为4n，确保足够空间
        this->tree.resize(4 * this->n, 0);
        if (this->n > 0) {
            // 构建线段树
            buildTree(nums, 0, 0, this->n - 1);
        }
    }

    /**
     * 单点更新公共接口
```

```
     * @param index 要更新的元素索引
     * @param val 新值
     * @throws invalid_argument 当索引超出范围时
     */
    void update(int index, int val) {
        if (index < 0 || index >= n) {
            throw invalid_argument("索引超出范围");
        }
        updateTree(0, 0, n - 1, index, val);
    }

    /**
     * 区间和查询公共接口
     * @param left 查询区间的左边界
     * @param right 查询区间的右边界
     * @return 查询区间的和
     * @throws invalid_argument 当查询区间无效时
     */
    int sumRange(int left, int right) {
        if (left < 0 || right >= n || left > right) {
            throw invalid_argument("查询区间无效");
        }
        return queryTree(0, 0, n - 1, left, right);
    }
};

/**
 * 树状数组实现类
 * 树状数组(Binary Indexed Tree 或 Fenwick Tree)是一种高效处理前缀和查询和单点更新的数据结构
 * 时间复杂度：
 * - 构建树状数组：O(n log n)
 * - 单点更新：O(log n)
 * - 前缀和查询：O(log n)
 * - 区间和查询：O(log n)
 * 空间复杂度：O(n)
 */
class NumArrayBIT {
private:
    vector<int> bit;  // 树状数组
    vector<int> nums; // 原始数组
    int n;            // 数组长度

    /**
```

```
 * lowbit 操作，获取 x 二进制表示中最低位的 1 所对应的值
 * @param x 输入整数
 * @return 最低位的 1 对应的值
 */
int lowbit(int x) {
    return x & (-x);
}

/**
 * 单点更新（树状数组内部方法）
 * @param index 要更新的元素索引（0-based）
 * @param val 新值
 */
void updateBIT(int index, int val) {
    // 将原数组中的增量累加到树状数组中
    int delta = val;
    index++; // 转换为 1-based 索引

    // 更新所有受影响的节点
    while (index <= n) {
        bit[index] += delta;
        index += lowbit(index);
    }
}

/**
 * 前缀和查询（树状数组内部方法）
 * @param index 查询到 index 的前缀和（0-based）
 * @return 前缀和
 */
int prefixSum(int index) {
    index++; // 转换为 1-based 索引
    int sum = 0;

    // 累加所有包含的区间
    while (index > 0) {
        sum += bit[index];
        index -= lowbit(index);
    }

    return sum;
}
```

```
public:
    /**
     * 构造函数
     * @param nums 输入数组
     */
    NumArrayBIT(const vector<int>& nums) {
        this->n = nums.size();
        this->nums = nums;
        this->bit.resize(this->n + 1, 0); // 树状数组从索引 1 开始

        // 初始化树状数组
        for (int i = 0; i < this->n; i++) {
            updateBIT(i, nums[i]);
        }
    }

    /**
     * 更新元素值（公共接口）
     * @param index 要更新的元素索引
     * @param val 新值
     * @throws invalid_argument 当索引超出范围时
     */
    void update(int index, int val) {
        if (index < 0 || index >= n) {
            throw invalid_argument("索引超出范围");
        }

        // 计算增量
        int delta = val - nums[index];
        nums[index] = val; // 更新原数组

        // 更新树状数组
        index++; // 转换为 1-based 索引
        while (index <= n) {
            bit[index] += delta;
            index += lowbit(index);
        }
    }

    /**
     * 区间和查询（公共接口）
     * @param left 区间左边界
     * @param right 区间右边界
```

```
     * @return 区间和
     * @throws invalid_argument 当查询区间无效时
     */
    int sumRange(int left, int right) {
        if (left < 0 || right >= n || left > right) {
            throw invalid_argument("查询区间无效");
        }

        // 区间和 = [0,right]的前缀和 - [0,left-1]的前缀和
        if (left == 0) {
            return prefixSum(right);
        } else {
            return prefixSum(right) - prefixSum(left - 1);
        }
    }
};

/**
 * 测试线段树实现
 */
void testSegmentTree() {
    cout << "=== 线段树实现测试 ===" << endl;

    // 测试用例1：基本操作
    vector<int> nums1 = {1, 3, 5, 7, 9, 11};
    Code15_RangeSumQueryMutable segTree(nums1);

    cout << "原始数组: [1, 3, 5, 7, 9, 11]" << endl;
    cout << "sumRange(0, 2) = " << segTree.sumRange(0, 2) << endl;  // 应为9 (1+3+5)
    segTree.update(1, 10);  // 将索引1的值从3更新为10
    cout << "更新索引1为10后" << endl;
    cout << "sumRange(0, 2) = " << segTree.sumRange(0, 2) << endl;  // 应为16 (1+10+5)
    cout << "sumRange(1, 5) = " << segTree.sumRange(1, 5) << endl;  // 应为42 (10+5+7+9+11)

    // 测试用例2：边界情况
    vector<int> nums2 = {5};
    Code15_RangeSumQueryMutable segTree2(nums2);
    cout << "\n测试边界情况" << endl;
    cout << "sumRange(0, 0) = " << segTree2.sumRange(0, 0) << endl;  // 应为5
    segTree2.update(0, 10);
    cout << "更新索引0为10后" << endl;
    cout << "sumRange(0, 0) = " << segTree2.sumRange(0, 0) << endl;  // 应为10
```

```
    // 测试用例 3：空数组
    vector<int> nums3 = {};
    Code15_RangeSumQueryMutable segTree3(nums3);
    cout << "\n 测试空数组" << endl;
    try {
        segTree3.sumRange(0, 0);
    } catch (const invalid_argument& e) {
        cout << "正确处理空数组异常: " << e.what() << endl;
    }
}

/**
 * 测试树状数组实现
 */
void testBIT() {
    cout << "\n=== 树状数组实现测试 ===" << endl;

    // 测试用例 1：基本操作
    vector<int> nums1 = {1, 3, 5, 7, 9, 11};
    NumArrayBIT bit(nums1);

    cout << "原始数组: [1, 3, 5, 7, 9, 11]" << endl;
    cout << "sumRange(0, 2) = " << bit.sumRange(0, 2) << endl;  // 应为 9 (1+3+5)
    bit.update(1, 10);  // 将索引 1 的值从 3 更新为 10
    cout << "更新索引 1 为 10 后" << endl;
    cout << "sumRange(0, 2) = " << bit.sumRange(0, 2) << endl;  // 应为 16 (1+10+5)
    cout << "sumRange(1, 5) = " << bit.sumRange(1, 5) << endl;  // 应为 42 (10+5+7+9+11)

    // 测试用例 2：边界情况
    vector<int> nums2 = {5};
    NumArrayBIT bit2(nums2);
    cout << "\n 测试边界情况" << endl;
    cout << "sumRange(0, 0) = " << bit2.sumRange(0, 0) << endl;  // 应为 5
    bit2.update(0, 10);
    cout << "更新索引 0 为 10 后" << endl;
    cout << "sumRange(0, 0) = " << bit2.sumRange(0, 0) << endl;  // 应为 10
}

/**
 * 性能对比测试
 */
void performanceTest() {
    cout << "\n=== 性能对比测试 ===" << endl;
```

```
    // 创建较大的测试数组
    int size = 100000;
    vector<int> nums(size);
    for (int i = 0; i < size; i++) {
        nums[i] = i % 100;
    }

    // 测试线段树性能
    auto start = high_resolution_clock::now();
    Code15_RangeSumQueryMutable segTree(nums);
    auto end = high_resolution_clock::now();
    auto buildTime = duration_cast<milliseconds>(end - start).count();

    start = high_resolution_clock::now();
    for (int i = 0; i < 10000; i++) {
        segTree.update(i % size, (i % size) + 100);
        segTree.sumRange((i * 2) % size, ((i * 3) % size + 100) % size);
    }
    end = high_resolution_clock::now();
    auto segTreeTime = duration_cast<milliseconds>(end - start).count();

    // 测试树状数组性能
    start = high_resolution_clock::now();
    NumArrayBIT bit(nums);
    end = high_resolution_clock::now();
    auto bitBuildTime = duration_cast<milliseconds>(end - start).count();

    start = high_resolution_clock::now();
    for (int i = 0; i < 10000; i++) {
        bit.update(i % size, (i % size) + 100);
        bit.sumRange((i * 2) % size, ((i * 3) % size + 100) % size);
    }
    end = high_resolution_clock::now();
    auto bitTime = duration_cast<milliseconds>(end - start).count();

    cout << "数组大小: " << size << endl;
    cout << "线段树构建时间: " << buildTime << "ms" << endl;
    cout << "树状数组构建时间: " << bitBuildTime << "ms" << endl;
    cout << "线段树 10000 次操作时间: " << segTreeTime << "ms" << endl;
    cout << "树状数组 10000 次操作时间: " << bitTime << "ms" << endl;
}
```

```
/**
 * 主函数
 */
int main() {
    testSegmentTree();
    testBIT();
    performanceTest();

    return 0;
}

/**
 * 算法总结与比较
 *
 * 1. 线段树 vs 树状数组:
 *    - 线段树功能更强大，可以处理更复杂的区间操作（如区间最大值、区间最小值等）
 *    - 树状数组代码更简洁，常数因子更小，空间效率更高
 *    - 树状数组更适合处理前缀和查询和单点更新
 *    - 线段树更适合处理多种类型的区间查询和更新
 *
 * 2. 应用场景:
 *    - 树状数组适合: 前缀和查询、单点更新、逆序对计算等
 *    - 线段树适合: 区间最大值/最小值查询、区间和查询、区间更新等
 *
 * 3. 时间复杂度分析:
 *    - 线段树和树状数组的单点更新和查询操作都是O(log n)
 *    - 线段树的区间更新可以是O(log n)（使用懒惰传播），而树状数组只能高效处理特定类型的区间更新
 *
 * 4. 空间复杂度:
 *    - 线段树需要O(4n)的空间
 *    - 树状数组只需要O(n)的空间
 */
```

==========

文件: Code15_RangeSumQueryMutable.java

==========

```
// LeetCode 307. Range Sum Query - Mutable
// 题目描述：给定一个整数数组 nums，实现两个函数：
// 1. update(i, val)：将数组中索引 i 处的值更新为 val
// 2. sumRange(i, j)：返回数组中从索引 i 到 j 的元素之和
// 题目链接：https://leetcode.com/problems/range-sum-query-mutable/
// 解题思路：使用线段树或树状数组实现，本题同时提供两种实现方式
```

```
import java.util.Arrays;

/**
 * 使用线段树实现区间和查询与单点更新
 * 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)
 */
public class Code15_RangeSumQueryMutable {

    // 线段树数组
    private int[] tree;
    // 原始数组长度
    private int n;

    /**
     * 构造函数
     * @param nums 输入数组
     */
    public Code15_RangeSumQueryMutable(int[] nums) {
        this.n = nums.length;
        // 线段树数组大小为4n，确保足够空间
        this.tree = new int[4 * n];
        if (n > 0) {
            // 构建线段树
            buildTree(nums, 0, 0, n - 1);
        }
    }

    /**
     * 构建线段树
     * @param nums 原始数组
     * @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 buildTree(int[] nums, int node, int start, int end) {
        if (start == end) {
            // 叶子节点，直接赋值
            tree[node] = nums[start];
```

```
        return;
    }

    int mid = start + (end - start) / 2;
    int leftNode = 2 * node + 1;
    int rightNode = 2 * node + 2;

    // 递归构建左右子树
    buildTree(nums, leftNode, start, mid);
    buildTree(nums, rightNode, mid + 1, end);

    // 当前节点的值为左右子节点之和
    tree[node] = tree[leftNode] + tree[rightNode];
}

/**
 * 单点更新（内部方法）
 * @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 要更新的元素索引
 * @param val 新值
 */
private void updateTree(int node, int start, int end, int index, int val) {
    if (start == end) {
        // 到达叶子节点，更新值
        tree[node] = val;
        return;
    }

    int mid = start + (end - start) / 2;
    int leftNode = 2 * node + 1;
    int rightNode = 2 * node + 2;

    // 根据 index 所在的区间决定递归左子树还是右子树
    if (index <= mid) {
        updateTree(leftNode, start, mid, index, val);
    } else {
        updateTree(rightNode, mid + 1, end, index, val);
    }

    // 更新当前节点的值
    tree[node] = tree[leftNode] + tree[rightNode];
```

```
}

/**
 * 区间查询（内部方法）
 * @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 查询区间的左边界
 * @param right 查询区间的右边界
 * @return 查询区间的和
 */
private int queryTree(int node, int start, int end, int left, int right) {
    // 查询区间与当前区间无交集
    if (right < start || left > end) {
        return 0;
    }

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

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

    int leftSum = queryTree(leftNode, start, mid, left, right);
    int rightSum = queryTree(rightNode, mid + 1, end, left, right);

    return leftSum + rightSum;
}

/**
 * 单点更新公共接口
 * @param index 要更新的元素索引
 * @param val 新值
 */
public void update(int index, int val) {
    if (index < 0 || index >= n) {
        throw new IllegalArgumentException("索引超出范围");
    }
    updateTree(0, 0, n - 1, index, val);
```

```
}

/**
 * 区间和查询公共接口
 * @param left 查询区间的左边界
 * @param right 查询区间的右边界
 * @return 查询区间的和
 */
public int sumRange(int left, int right) {
    if (left < 0 || right >= n || left > right) {
        throw new IllegalArgumentException("查询区间无效");
    }
    return queryTree(0, 0, n - 1, left, right);
}

/**
 * 树状数组实现类
 * 树状数组(Binary Indexed Tree 或 Fenwick Tree)是一种高效处理前缀和查询和单点更新的数据结构
 * 时间复杂度：
 * - 构建树状数组：O(n log n)
 * - 单点更新：O(log n)
 * - 前缀和查询：O(log n)
 * - 区间和查询：O(log n)
 * 空间复杂度：O(n)
 */
static class NumArrayBIT {
    private int[] bit;  // 树状数组
    private int[] nums; // 原始数组
    private int n;      // 数组长度

    /**
     * 构造函数
     * @param nums 输入数组
     */
    public NumArrayBIT(int[] nums) {
        this.n = nums.length;
        this.nums = Arrays.copyOf(nums, n);
        this.bit = new int[n + 1]; // 树状数组从索引 1 开始

        // 初始化树状数组
        for (int i = 0; i < n; i++) {
            updateBIT(i, nums[i]);
        }
```

```
}

/**
 * lowbit 操作，获取 x 二进制表示中最低位的 1 所对应的值
 * @param x 输入整数
 * @return 最低位的 1 对应的值
 */
private int lowbit(int x) {
    return x & (-x);
}

/**
 * 单点更新（树状数组内部方法）
 * @param index 要更新的元素索引（0-based）
 * @param val 新值
 */
private void updateBIT(int index, int val) {
    // 将原数组中的增量累加到树状数组中
    int delta = val;
    index++; // 转换为 1-based 索引

    // 更新所有受影响的节点
    while (index <= n) {
        bit[index] += delta;
        index += lowbit(index);
    }
}

/**
 * 更新元素值（公共接口）
 * @param index 要更新的元素索引
 * @param val 新值
 */
public void update(int index, int val) {
    if (index < 0 || index >= n) {
        throw new IllegalArgumentException("索引超出范围");
    }

    // 计算增量
    int delta = val - nums[index];
    nums[index] = val; // 更新原数组

    // 更新树状数组
```

```
        index++; // 转换为1-based索引
        while (index <= n) {
            bit[index] += delta;
            index += lowbit(index);
        }
    }

    /**
     * 前缀和查询（树状数组内部方法）
     * @param index 查询到index的前缀和（0-based）
     * @return 前缀和
     */
    private int prefixSum(int index) {
        index++; // 转换为1-based索引
        int sum = 0;

        // 累加所有包含的区间
        while (index > 0) {
            sum += bit[index];
            index -= lowbit(index);
        }

        return sum;
    }

    /**
     * 区间和查询（公共接口）
     * @param left 区间左边界
     * @param right 区间右边界
     * @return 区间和
     */
    public int sumRange(int left, int right) {
        if (left < 0 || right >= n || left > right) {
            throw new IllegalArgumentException("查询区间无效");
        }

        // 区间和 = [0,right]的前缀和 - [0,left-1]的前缀和
        if (left == 0) {
            return prefixSum(right);
        } else {
            return prefixSum(right) - prefixSum(left - 1);
        }
    }
```

```
    }

    /**
     * 测试方法
     */
    public static void main(String[] args) {
        System.out.println("=== 线段树实现测试 ===");
        testSegmentTree();

        System.out.println("\n=== 树状数组实现测试 ===");
        testBIT();

        // 性能对比测试
        System.out.println("\n=== 性能对比测试 ===");
        performanceTest();
    }

    /**
     * 测试线段树实现
     */
    private static void testSegmentTree() {
        // 测试用例 1：基本操作
        int[] nums1 = {1, 3, 5, 7, 9, 11};
        Code15_RangeSumQueryMutable segTree = new Code15_RangeSumQueryMutable(nums1);

        System.out.println("原始数组: " + Arrays.toString(nums1));
        System.out.println("sumRange(0, 2) = " + segTree.sumRange(0, 2));  // 应为 9 (1+3+5)
        segTree.update(1, 10);  // 将索引 1 的值从 3 更新为 10
        System.out.println("更新索引 1 为 10 后");
        System.out.println("sumRange(0, 2) = " + segTree.sumRange(0, 2));  // 应为 16 (1+10+5)
        System.out.println("sumRange(1, 5) = " + segTree.sumRange(1, 5));  // 应为 42
(10+5+7+9+11)

        // 测试用例 2：边界情况
        int[] nums2 = {5};
        Code15_RangeSumQueryMutable segTree2 = new Code15_RangeSumQueryMutable(nums2);
        System.out.println("\n 测试边界情况");
        System.out.println("sumRange(0, 0) = " + segTree2.sumRange(0, 0));  // 应为 5
        segTree2.update(0, 10);
        System.out.println("更新索引 0 为 10 后");
        System.out.println("sumRange(0, 0) = " + segTree2.sumRange(0, 0));  // 应为 10

        // 测试用例 3：空数组
```

```
    int[] nums3 = {};
    Code15_RangeSumQueryMutable segTree3 = new Code15_RangeSumQueryMutable(nums3);
    System.out.println("\n测试空数组");
    try {
        segTree3.sumRange(0, 0);
    } catch (IllegalArgumentException e) {
        System.out.println("正确处理空数组异常: " + e.getMessage());
    }
}

/**
 * 测试树状数组实现
 */
private static void testBIT() {
    // 测试用例1：基本操作
    int[] nums1 = {1, 3, 5, 7, 9, 11};
    NumArrayBIT bit = new NumArrayBIT(nums1);

    System.out.println("原始数组: " + Arrays.toString(nums1));
    System.out.println("sumRange(0, 2) = " + bit.sumRange(0, 2));  // 应为9 (1+3+5)
    bit.update(1, 10);  // 将索引1的值从3更新为10
    System.out.println("更新索引1为10后");
    System.out.println("sumRange(0, 2) = " + bit.sumRange(0, 2));  // 应为16 (1+10+5)
    System.out.println("sumRange(1, 5) = " + bit.sumRange(1, 5));  // 应为42 (10+5+7+9+11)

    // 测试用例2：边界情况
    int[] nums2 = {5};
    NumArrayBIT bit2 = new NumArrayBIT(nums2);
    System.out.println("\n测试边界情况");
    System.out.println("sumRange(0, 0) = " + bit2.sumRange(0, 0));  // 应为5
    bit2.update(0, 10);
    System.out.println("更新索引0为10后");
    System.out.println("sumRange(0, 0) = " + bit2.sumRange(0, 0));  // 应为10
}

/**
 * 性能对比测试
 */
private static void performanceTest() {
    // 创建较大的测试数组
    int size = 100000;
    int[] nums = new int[size];
    for (int i = 0; i < size; i++) {
```

```
            nums[i] = i % 100;
        }

        // 测试线段树性能
        long startTime = System.currentTimeMillis();
        Code15_RangeSumQueryMutable segTree = new Code15_RangeSumQueryMutable(nums);
        long buildTime = System.currentTimeMillis() - startTime;

        startTime = System.currentTimeMillis();
        for (int i = 0; i < 10000; i++) {
            segTree.update(i % size, (i % size) + 100);
            segTree.sumRange((i * 2) % size, ((i * 3) % size + 100) % size);
        }
        long segTreeTime = System.currentTimeMillis() - startTime;

        // 测试树状数组性能
        startTime = System.currentTimeMillis();
        NumArrayBIT bit = new NumArrayBIT(nums);
        long bitBuildTime = System.currentTimeMillis() - startTime;

        startTime = System.currentTimeMillis();
        for (int i = 0; i < 10000; i++) {
            bit.update(i % size, (i % size) + 100);
            bit.sumRange((i * 2) % size, ((i * 3) % size + 100) % size);
        }
        long bitTime = System.currentTimeMillis() - startTime;

        System.out.println("数组大小: " + size);
        System.out.println("线段树构建时间: " + buildTime + "ms");
        System.out.println("树状数组构建时间: " + bitBuildTime + "ms");
        System.out.println("线段树 10000 次操作时间: " + segTreeTime + "ms");
        System.out.println("树状数组 10000 次操作时间: " + bitTime + "ms");
    }

    /**
     * 算法总结与比较
     *
     * 1. 线段树 vs 树状数组:
     *    - 线段树功能更强大，可以处理更复杂的区间操作（如区间最大值、区间最小值等）
     *    - 树状数组代码更简洁，常数因子更小，空间效率更高
     *    - 树状数组更适合处理前缀和查询和单点更新
     *    - 线段树更适合处理多种类型的区间查询和更新
     *
```

```
     * 2. 应用场景:
     *    - 树状数组适合: 前缀和查询、单点更新、逆序对计算等
     *    - 线段树适合: 区间最大值/最小值查询、区间和查询、区间更新等
     *
     * 3. 时间复杂度分析:
     *    - 线段树和树状数组的单点更新和查询操作都是O(log n)
     *    - 线段树的区间更新可以是O(log n)（使用懒惰传播），而树状数组只能高效处理特定类型的区间更新
     *
     * 4. 空间复杂度:
     *    - 线段树需要O(4n)的空间
     *    - 树状数组只需要O(n)的空间
     */
}

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

文件: Code15_RangeSumQueryMutable.py
==========================================
# LeetCode 307. Range Sum Query - Mutable
# 题目描述：给定一个整数数组 nums，实现两个函数：
# 1. update(i, val)：将数组中索引 i 处的值更新为 val
# 2. sumRange(i, j)：返回数组中从索引 i 到 j 的元素之和
# 题目链接：https://leetcode.com/problems/range-sum-query-mutable/
# 解题思路：使用线段树或树状数组实现，本题同时提供两种实现方式

class Code15_RangeSumQueryMutable:
    """
    使用线段树实现区间和查询与单点更新

    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)
    """

    def __init__(self, nums):
        """
        构造函数
        :param nums: 输入数组
        """
        self.n = len(nums)
```

```
        # 线段树数组大小为4n，确保足够空间
        self.tree = [0] * (4 * self.n)
        if self.n > 0:
            # 构建线段树
            self._build_tree(nums, 0, 0, self.n - 1)

    def _build_tree(self, nums, node, start, end):
        """
        构建线段树
        :param nums: 原始数组
        :param node: 当前节点索引
        :param start: 当前区间左边界
        :param end: 当前区间右边界
        """
        if start == end:
            # 叶子节点，直接赋值
            self.tree[node] = nums[start]
            return

        mid = start + (end - start) // 2
        left_node = 2 * node + 1
        right_node = 2 * node + 2

        # 递归构建左右子树
        self._build_tree(nums, left_node, start, mid)
        self._build_tree(nums, right_node, mid + 1, end)

        # 当前节点的值为左右子节点之和
        self.tree[node] = self.tree[left_node] + self.tree[right_node]

    def _update_tree(self, node, start, end, index, val):
        """
        单点更新（内部方法）
        :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: 要更新的元素索引
        :param val: 新值
        """
        if start == end:
            # 到达叶子节点，更新值
            self.tree[node] = val
            return
```

```
        mid = start + (end - start) // 2
        left_node = 2 * node + 1
        right_node = 2 * node + 2

        # 根据 index 所在的区间决定递归左子树还是右子树
        if index <= mid:
            self._update_tree(left_node, start, mid, index, val)
        else:
            self._update_tree(right_node, mid + 1, end, index, val)

        # 更新当前节点的值
        self.tree[node] = self.tree[left_node] + self.tree[right_node]

    def _query_tree(self, node, start, end, left, right):
        """
        区间查询（内部方法）
        :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: 查询区间的左边界
        :param right: 查询区间的右边界
        :return: 查询区间的和
        """
        # 查询区间与当前区间无交集
        if right < start or left > end:
            return 0

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

        # 查询区间部分重叠，递归查询左右子树
        mid = start + (end - start) // 2
        left_node = 2 * node + 1
        right_node = 2 * node + 2

        left_sum = self._query_tree(left_node, start, mid, left, right)
        right_sum = self._query_tree(right_node, mid + 1, end, left, right)

        return left_sum + right_sum

    def update(self, index, val):
```

```
        """
        单点更新公共接口
        :param index: 要更新的元素索引
        :param val: 新值
        :raises ValueError: 当索引超出范围时
        """
        if index < 0 or index >= self.n:
            raise ValueError("索引超出范围")
        self._update_tree(0, 0, self.n - 1, index, val)

    def sum_range(self, left, right):
        """
        区间和查询公共接口
        :param left: 查询区间的左边界
        :param right: 查询区间的右边界
        :return: 查询区间的和
        :raises ValueError: 当查询区间无效时
        """
        if left < 0 or right >= self.n or left > right:
            raise ValueError("查询区间无效")
        return self._query_tree(0, 0, self.n - 1, left, right)

class NumArrayBIT:
    """
    树状数组实现类

    树状数组(Binary Indexed Tree 或 Fenwick Tree)是一种高效处理前缀和查询和单点更新的数据结构

    时间复杂度：
    - 构建树状数组：O(n log n)
    - 单点更新：O(log n)
    - 前缀和查询：O(log n)
    - 区间和查询：O(log n)
    空间复杂度：O(n)
    """

    def __init__(self, nums):
        """
        构造函数
        :param nums: 输入数组
        """
        self.n = len(nums)
        self.nums = nums.copy()
```

```
        self.bit = [0] * (self.n + 1)  # 树状数组从索引 1 开始

        # 初始化树状数组
        for i in range(self.n):
            self._update_bit(i, nums[i])

    def _lowbit(self, x):
        """
        lowbit 操作，获取 x 二进制表示中最低位的 1 所对应的值
        :param x: 输入整数
        :return: 最低位的 1 对应的值
        """
        return x & (-x)

    def _update_bit(self, index, val):
        """
        单点更新（树状数组内部方法）
        :param index: 要更新的元素索引（0-based）
        :param val: 新值
        """
        # 将原数组中的增量累加到树状数组中
        delta = val
        index += 1  # 转换为 1-based 索引

        # 更新所有受影响的节点
        while index <= self.n:
            self.bit[index] += delta
            index += self._lowbit(index)

    def update(self, index, val):
        """
        更新元素值（公共接口）
        :param index: 要更新的元素索引
        :param val: 新值
        :raises ValueError: 当索引超出范围时
        """
        if index < 0 or index >= self.n:
            raise ValueError("索引超出范围")

        # 计算增量
        delta = val - self.nums[index]
        self.nums[index] = val  # 更新原数组
```

```
        # 更新树状数组
        index += 1  # 转换为1-based索引
        while index <= self.n:
            self.bit[index] += delta
            index += self._lowbit(index)

    def _prefix_sum(self, index):
        """
        前缀和查询（树状数组内部方法）
        :param index: 查询到index的前缀和（0-based）
        :return: 前缀和
        """
        index += 1  # 转换为1-based索引
        sum_val = 0

        # 累加所有包含的区间
        while index > 0:
            sum_val += self.bit[index]
            index -= self._lowbit(index)

        return sum_val

    def sum_range(self, left, right):
        """
        区间和查询（公共接口）
        :param left: 区间左边界
        :param right: 区间右边界
        :return: 区间和
        :raises ValueError: 当查询区间无效时
        """
        if left < 0 or right >= self.n or left > right:
            raise ValueError("查询区间无效")

        # 区间和 = [0,right]的前缀和 - [0,left-1]的前缀和
        if left == 0:
            return self._prefix_sum(right)
        else:
            return self._prefix_sum(right) - self._prefix_sum(left - 1)

# 测试函数
def test_segment_tree():
    print("=== 线段树实现测试 ===")
```

```
    # 测试用例 1：基本操作
    nums1 = [1, 3, 5, 7, 9, 11]
    seg_tree = Code15_RangeSumQueryMutable(nums1)

    print(f"原始数组: {nums1}")
    print(f"sum_range(0, 2) = {seg_tree.sum_range(0, 2)}")  # 应为 9 (1+3+5)
    seg_tree.update(1, 10)  # 将索引 1 的值从 3 更新为 10
    print("更新索引 1 为 10 后")
    print(f"sum_range(0, 2) = {seg_tree.sum_range(0, 2)}")  # 应为 16 (1+10+5)
    print(f"sum_range(1, 5) = {seg_tree.sum_range(1, 5)}")  # 应为 42 (10+5+7+9+11)

    # 测试用例 2：边界情况
    nums2 = [5]
    seg_tree2 = Code15_RangeSumQueryMutable(nums2)
    print("\n 测试边界情况")
    print(f"sum_range(0, 0) = {seg_tree2.sum_range(0, 0)}")  # 应为 5
    seg_tree2.update(0, 10)
    print("更新索引 0 为 10 后")
    print(f"sum_range(0, 0) = {seg_tree2.sum_range(0, 0)}")  # 应为 10

    # 测试用例 3：空数组
    nums3 = []
    seg_tree3 = Code15_RangeSumQueryMutable(nums3)
    print("\n 测试空数组")
    try:
        seg_tree3.sum_range(0, 0)
    except ValueError as e:
        print(f"正确处理空数组异常: {e}")

def test_bit():
    print("\n=== 树状数组实现测试 ===")

    # 测试用例 1：基本操作
    nums1 = [1, 3, 5, 7, 9, 11]
    bit = NumArrayBIT(nums1)

    print(f"原始数组: {nums1}")
    print(f"sum_range(0, 2) = {bit.sum_range(0, 2)}")  # 应为 9 (1+3+5)
    bit.update(1, 10)  # 将索引 1 的值从 3 更新为 10
    print("更新索引 1 为 10 后")
    print(f"sum_range(0, 2) = {bit.sum_range(0, 2)}")  # 应为 16 (1+10+5)
    print(f"sum_range(1, 5) = {bit.sum_range(1, 5)}")  # 应为 42 (10+5+7+9+11)
```

```
    # 测试用例 2：边界情况
    nums2 = [5]
    bit2 = NumArrayBIT(nums2)
    print("\n 测试边界情况")
    print(f"sum_range(0, 0) = {bit2.sum_range(0, 0)}")  # 应为 5
    bit2.update(0, 10)
    print("更新索引 0 为 10 后")
    print(f"sum_range(0, 0) = {bit2.sum_range(0, 0)}")  # 应为 10

def performance_test():
    print("\n=== 性能对比测试 ===")

    # 创建较大的测试数组
    size = 100000
    nums = [i % 100 for i in range(size)]

    # 测试线段树性能
    import time
    start_time = time.time()
    seg_tree = Code15_RangeSumQueryMutable(nums)
    build_time = time.time() - start_time

    start_time = time.time()
    for i in range(10000):
        seg_tree.update(i % size, (i % size) + 100)
        seg_tree.sum_range((i * 2) % size, ((i * 3) % size + 100) % size)
    seg_tree_time = time.time() - start_time

    # 测试树状数组性能
    start_time = time.time()
    bit = NumArrayBIT(nums)
    bit_build_time = time.time() - start_time

    start_time = time.time()
    for i in range(10000):
        bit.update(i % size, (i % size) + 100)
        bit.sum_range((i * 2) % size, ((i * 3) % size + 100) % size)
    bit_time = time.time() - start_time

    print(f"数组大小: {size}")
    print(f"线段树构建时间: {build_time*1000:.2f}ms")
    print(f"树状数组构建时间: {bit_build_time*1000:.2f}ms")
    print(f"线段树 10000 次操作时间: {seg_tree_time*1000:.2f}ms")
```

```
    print(f"树状数组 10000 次操作时间: {bit_time*1000:.2f}ms")

# 运行测试
if __name__ == "__main__":
    test_segment_tree()
    test_bit()
    performance_test()

'''
算法总结与比较：

1. 线段树 vs 树状数组:
   - 线段树功能更强大，可以处理更复杂的区间操作（如区间最大值、区间最小值等）
   - 树状数组代码更简洁，常数因子更小，空间效率更高
   - 树状数组更适合处理前缀和查询和单点更新
   - 线段树更适合处理多种类型的区间查询和更新

2. 应用场景:
   - 树状数组适合: 前缀和查询、单点更新、逆序对计算等
   - 线段树适合: 区间最大值/最小值查询、区间和查询、区间更新等

3. 时间复杂度分析:
   - 线段树和树状数组的单点更新和查询操作都是 O(log n)
   - 线段树的区间更新可以是 O(log n)（使用懒惰传播），而树状数组只能高效处理特定类型的区间更新

4. 空间复杂度:
   - 线段树需要 O(4n)的空间
   - 树状数组只需要 O(n)的空间
'''


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

文件: Code16_Stars.cpp
==============================================
// POJ 2352 Stars
// 题目描述：给定 N 个星星的坐标(x,y)，满足 y 坐标升序排列，若 y 相同则 x 升序排列。
// 每个星星的等级是它左下角区域内星星的数量（即 x 坐标≤其 x，y 坐标≤其 y 的星星数目，不包括自身）。
// 输出等级为 0 到 N-1 的星星数目。
// 题目链接：http://poj.org/problem?id=2352
// 解题思路：树状数组 + 离散化

#include <iostream>
#include <vector>
```

```
#include <algorithm>
#include <unordered_map>
#include <chrono>
using namespace std;
using namespace std::chrono;

/**
 * 使用树状数组解决 Stars 问题
 *
 * 时间复杂度：O(N log N)
 * 空间复杂度：O(max_x)
 *
 * 本题特点：
 * 1. 由于输入是按 y 升序排列的，所以对于每个星星来说，之前处理过的星星的 y 坐标都不超过它的 y 坐标
 * 2. 因此我们只需要统计之前处理过的星星中 x 坐标小于等于当前星星 x 坐标的数量
 * 3. 这可以通过树状数组高效实现，每次查询前缀和，然后更新树状数组
 */
class Code16_Stars {
private:
    int maxX;        // 最大 x 坐标值
    vector<int> bit; // 树状数组
    vector<int> result; // 存储每个等级的星星数目

    /**
     * lowbit 操作，获取 x 二进制表示中最低位的 1 所对应的值
     * @param x 输入整数
     * @return 最低位的 1 对应的值
     */
    int lowbit(int x) {
        return x & (-x);
    }

    /**
     * 更新树状数组
     * @param x 要更新的位置（1-based）
     * @param val 要增加的值
     */
    void update(int x, int val) {
        while (x <= maxX) {
            bit[x] += val;
            x += lowbit(x);
        }
    }
```

```
    /**
     * 查询前缀和，即 1 到 x 的累加和
     * @param x 查询上限（1-based）
     * @return 前缀和
     */
    int query(int x) {
        int sum = 0;
        while (x > 0) {
            sum += bit[x];
            x -= lowbit(x);
        }
        return sum;
    }

public:
    /**
     * 构造函数
     * @param max_x_value 最大 x 坐标值
     */
    Code16_Stars(int max_x_value) {
        // 初始化树状数组和结果数组
        // 注意：这里 max_x_value+1 是因为树状数组下标从 1 开始
        this->maxX = max_x_value;
        this->bit.resize(max_x_value + 2, 0); // +2 防止溢出
        this->result.resize(max_x_value + 1, 0); // 等级最多为 max_x
    }

    /**
     * 处理星星数据，计算每个星星的等级
     * @param stars 星星坐标数组
     * @return 等级统计结果，result[i]表示等级为 i 的星星数目
     */
    vector<int> processStars(const vector<pair<int, int>>& stars) {
        // 统计每个星星的等级
        for (const auto& star : stars) {
            int x = star.first;
            // 由于树状数组索引从 1 开始，我们将 x 坐标+1
            // 计算当前星星的等级：查询小于等于 x 的星星数量
            int level = query(x + 1); // 转换为 1-based 索引

            // 更新等级统计
            result[level]++;
```

```
        // 将当前星星加入树状数组
        update(x + 1, 1); // 转换为1-based索引
    }

    return result;
}

/**
 * 处理星星数据（带离散化）
 * 当x坐标范围很大时使用离散化可以节省空间
 * @param stars 星星坐标数组
 * @return 等级统计结果
 */
vector<int> processStarsWithDiscretization(const vector<pair<int, int>>& stars) {
    // 提取所有x坐标用于离散化
    vector<int> xs;
    xs.reserve(stars.size());
    for (const auto& star : stars) {
        xs.push_back(star.first);
    }

    // 离散化处理
    unordered_map<int, int> coordinateMapping = discretize(xs);

    // 重置树状数组为离散化后的大小
    this->maxX = coordinateMapping.size();
    this->bit.assign(this->maxX + 2, 0); // +2 防止溢出
    this->result.assign(stars.size(), 0); // 重置结果数组

    // 处理星星数据
    for (const auto& star : stars) {
        int x = star.first;
        // 获取离散化后的值（从1开始）
        int discretizedX = coordinateMapping[x] + 1;

        // 计算当前星星的等级
        int level = query(discretizedX);

        // 更新等级统计
        result[level]++;

        // 将当前星星加入树状数组
```

```
            update(discretizedX, 1);
        }

        return result;
    }

    /**
     * 离散化处理
     * @param nums 原始数据数组
     * @return 原始值到离散化值的映射
     */
    unordered_map<int, int> discretize(const vector<int>& nums) {
        // 复制并去重
        vector<int> uniqueNums(nums.begin(), nums.end());
        sort(uniqueNums.begin(), uniqueNums.end());
        uniqueNums.erase(unique(uniqueNums.begin(), uniqueNums.end()), uniqueNums.end());

        // 构建映射
        unordered_map<int, int> mapping;
        for (int i = 0; i < uniqueNums.size(); i++) {
            mapping[uniqueNums[i]] = i; // 从0开始的离散化值
        }

        return mapping;
    }

    /**
     * 打印结果
     * @param result 等级统计结果
     * @param n 星星数量
     */
    static void printResult(const vector<int>& result, int n) {
        for (int i = 0; i < n; i++) {
            cout << result[i] << endl;
        }
    }
};

/**
 * 测试函数
 */
void test() {
    cout << "=== 测试用例1（无需离散化）===" << endl;
```

```
// 测试用例 1：简单示例
vector<pair<int, int>> stars1 = {
    {1, 1},
    {2, 2},
    {3, 3},
    {1, 3},
    {2, 1}
};

// 找出最大的 x 坐标
int maxX1 = 0;
for (const auto& star : stars1) {
    maxX1 = max(maxX1, star.first);
}

Code16_Stars solver1(maxX1);
vector<int> result1 = solver1.processStars(stars1);
Code16_Stars::printResult(result1, stars1.size());

cout << "\n=== 测试用例 2（使用离散化）===" << endl;
// 测试用例 2：使用离散化
vector<pair<int, int>> stars2 = {
    {10000, 1},
    {20000, 2},
    {5000, 3},
    {10000, 3},
    {20000, 1}
};

Code16_Stars solver2(20000); // 初始值不重要，会在离散化时重置
vector<int> result2 = solver2.processStarsWithDiscretization(stars2);
Code16_Stars::printResult(result2, stars2.size());

cout << "\n=== 测试用例 3（所有星星在同一点）===" << endl;
// 测试用例 3：边界情况 - 所有星星在同一点
vector<pair<int, int>> stars3 = {
    {1, 1},
    {1, 1},
    {1, 1}
};

int maxX3 = 1;
Code16_Stars solver3(maxX3);
```

```
    vector<int> result3 = solver3.processStars(stars3);
    Code16_Stars::printResult(result3, stars3.size());
}

/**
 * 性能测试
 */
void performanceTest() {
    cout << "\n=== 性能测试 ===" << endl;

    // 生成测试数据
    int n = 100000;
    vector<pair<int, int>> stars;
    stars.reserve(n);

    // 生成随机坐标，保持 y 升序排列
    for (int i = 0; i < n; i++) {
        int y = i / 100; // 保证 y 升序
        int x = rand() % 1000000 + 1; // x 坐标在 1 到 1e6 之间
        stars.emplace_back(x, y);
    }

    // 确保数据按 y 升序排列，相同 y 时按 x 升序排列
    sort(stars.begin(), stars.end(), [](const pair<int, int>& a, const pair<int, int>& b) {
        if (a.second == b.second) {
            return a.first < b.first;
        }
        return a.second < b.second;
    });

    // 测试普通方法
    cout << "测试处理" << stars.size() << "个星星的数据..." << endl;
    auto start = high_resolution_clock::now();

    // 找出最大的 x 坐标
    int maxX = 0;
    for (const auto& star : stars) {
        maxX = max(maxX, star.first);
    }

    Code16_Stars solver(maxX);
    vector<int> result = solver.processStars(stars);
```

```
    auto end = high_resolution_clock::now();
    auto normalTime = duration_cast<milliseconds>(end - start).count();
    cout << "普通方法耗时: " << normalTime << "ms" << endl;

    // 测试离散化方法
    start = high_resolution_clock::now();
    Code16_Stars solverDisc(maxX); // 初始值不重要
    vector<int> resultDisc = solverDisc.processStarsWithDiscretization(stars);

    end = high_resolution_clock::now();
    auto discTime = duration_cast<milliseconds>(end - start).count();
    cout << "离散化方法耗时: " << discTime << "ms" << endl;

    // 验证结果是否一致
    bool isConsistent = true;
    for (int i = 0; i < n; i++) {
        if (result[i] != resultDisc[i]) {
            isConsistent = false;
            break;
        }
    }
    cout << "结果一致性验证: " << (isConsistent ? "通过" : "失败") << endl;
}

/**
 * 主函数
 */
int main() {
    // 设置随机数种子
    srand(time(nullptr));

    // 运行测试
    test();

    // 运行性能测试
    performanceTest();

    // 实际输入处理
    cout << "\n=== 输入测试（输入 N 和 N 个坐标）===" << endl;
    try {
        int n;
        cout << "请输入星星数量 N: ";
        cin >> n;
```

```
        vector<pair<int, int>> stars;
        stars.reserve(n);
        int maxX = 0;

        for (int i = 0; i < n; i++) {
            int x, y;
            cout << "请输入第" << (i + 1) << "个星星的坐标(x y): ";
            cin >> x >> y;
            stars.emplace_back(x, y);
            maxX = max(maxX, x);
        }

        // 处理输入数据
        Code16_Stars solver(maxX);
        vector<int> result = solver.processStars(stars);

        // 输出结果
        cout << "\n输出结果：" << endl;
        Code16_Stars::printResult(result, n);
    } catch (const exception& e) {
        cout << "输入错误: " << e.what() << endl;
    }

    return 0;
}

/**
 * 算法总结：
 *
 * 1. 本题的关键洞察：
 *    - 由于输入的星星是按 y 坐标升序排列的，所以处理每个星星时，所有已处理的星星的 y 坐标都不大于当前星星的 y 坐标
 *    - 因此，当前星星的等级就是已处理星星中 x 坐标小于等于当前星星 x 坐标的数量
 *    - 这可以通过树状数组高效地进行前缀和查询和单点更新
 *
 * 2. 离散化的必要性：
 *    - 当 x 坐标范围很大时（比如到 1e9），直接使用树状数组会导致空间浪费
 *    - 离散化可以将所有不同的 x 坐标映射到较小的连续整数范围，节省空间
 *    - 在本题中，如果 x 坐标范围不大，可以不使用离散化
 *
 * 3. 树状数组操作：
 *    - update(x, val): 在位置 x 增加 val
```

```
 *     - query(x): 查询前缀和[1,x]
 *     - lowbit(x): 获取 x 二进制表示中最低位的 1
 *
 * 4. 时间复杂度分析：
 *     - 树状数组的 update 和 query 操作都是 O(log M)，其中 M 是最大 x 坐标值（或离散化后的坐标范围）
 *     - 处理 n 个星星的总时间复杂度为 O(n log M)
 *     - 离散化的时间复杂度为 O(n log n)
 *     - 因此总时间复杂度为 O(n log n)
 *
 * 5. 空间复杂度：
 *     - 不使用离散化：O(M)，其中 M 是最大 x 坐标值
 *     - 使用离散化：O(n)，只需存储不同的 x 坐标
 *
 * 6. C++实现注意事项：
 *     - 在 C++中，使用 vector 动态调整大小，避免数组越界问题
 *     - 使用 sort 和 unique 进行高效的去重操作
 *     - 使用 unordered_map 实现 O(1)的查询时间复杂度
 *     - 对于大规模数据，离散化可以显著提高内存效率
 */

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

文件: Code16_Stars.java
==========================================
// POJ 2352 Stars
// 题目描述：给定 N 个星星的坐标(x,y)，满足 y 坐标升序排列，若 y 相同则 x 升序排列。
// 每个星星的等级是它左下角区域内星星的数量（即 x 坐标≤其 x，y 坐标≤其 y 的星星数目，不包括自身）。
// 输出等级为 0 到 N-1 的星星数目。
// 题目链接：http://poj.org/problem?id=2352
// 解题思路：树状数组 + 离散化

import java.util.*;

/**
 * 使用树状数组解决 Stars 问题
 *
 * 时间复杂度：O(N log N)
 * 空间复杂度：O(max_x)
 *
 * 本题特点：
 * 1. 由于输入是按 y 升序排列的，所以对于每个星星来说，之前处理过的星星的 y 坐标都不超过它的 y 坐标
 * 2. 因此我们只需要统计之前处理过的星星中 x 坐标小于等于当前星星 x 坐标的数量
 * 3. 这可以通过树状数组高效实现，每次查询前缀和，然后更新树状数组
```

```
 */
public class Code16_Stars {

    private int maxX; // 最大 x 坐标值
    private int[] bit; // 树状数组
    private int[] result; // 存储每个等级的星星数目

    public Code16_Stars(int maxXValue) {
        // 初始化树状数组和结果数组
        // 注意：这里 maxX+1 是因为树状数组下标从 1 开始
        this.maxX = maxXValue;
        this.bit = new int[maxX + 2]; // +2 防止溢出
        this.result = new int[maxX + 1]; // 等级最多为 maxX
    }

    /**
     * lowbit 操作，获取 x 二进制表示中最低位的 1 所对应的值
     * @param x 输入整数
     * @return 最低位的 1 对应的值
     */
    private int lowbit(int x) {
        return x & (-x);
    }

    /**
     * 更新树状数组
     * @param x 要更新的位置（1-based）
     * @param val 要增加的值
     */
    private void update(int x, int val) {
        while (x <= maxX) {
            bit[x] += val;
            x += lowbit(x);
        }
    }

    /**
     * 查询前缀和，即 1 到 x 的累加和
     * @param x 查询上限（1-based）
     * @return 前缀和
     */
    private int query(int x) {
        int sum = 0;
```

```
        while (x > 0) {
            sum += bit[x];
            x -= lowbit(x);
        }
        return sum;
    }

    /**
     * 处理星星数据，计算每个星星的等级
     * @param stars 星星坐标数组
     * @return 等级统计结果，result[i]表示等级为 i 的星星数目
     */
    public int[] processStars(int[][] stars) {
        // 统计每个星星的等级
        for (int[] star : stars) {
            int x = star[0];
            // 由于树状数组索引从 1 开始，我们将 x 坐标+1
            // 计算当前星星的等级：查询小于等于 x 的星星数量
            int level = query(x + 1); // 转换为 1-based 索引

            // 更新等级统计
            result[level]++;

            // 将当前星星加入树状数组
            update(x + 1, 1); // 转换为 1-based 索引
        }

        return result;
    }

    /**
     * 处理星星数据（带离散化）
     * 当 x 坐标范围很大时使用离散化可以节省空间
     * @param stars 星星坐标数组
     * @return 等级统计结果
     */
    public int[] processStarsWithDiscretization(int[][] stars) {
        // 提取所有 x 坐标用于离散化
        int[] xs = new int[stars.length];
        for (int i = 0; i < stars.length; i++) {
            xs[i] = stars[i][0];
        }
```

```
        // 离散化处理
        Map<Integer, Integer> coordinateMapping = discretize(xs);

        // 重置树状数组为离散化后的大小
        this.maxX = coordinateMapping.size();
        this.bit = new int[this.maxX + 2]; // +2 防止溢出
        this.result = new int[stars.length]; // 重置结果数组

        // 处理星星数据
        for (int[] star : stars) {
            int x = star[0];
            // 获取离散化后的值（从 1 开始）
            int discretizedX = coordinateMapping.get(x) + 1;

            // 计算当前星星的等级
            int level = query(discretizedX);

            // 更新等级统计
            result[level]++;

            // 将当前星星加入树状数组
            update(discretizedX, 1);
        }

        return result;
    }

    /**
     * 离散化处理
     * @param nums 原始数据数组
     * @return 原始值到离散化值的映射
     */
    private Map<Integer, Integer> discretize(int[] nums) {
        // 复制并去重
        Set<Integer> uniqueNums = new HashSet<>();
        for (int num : nums) {
            uniqueNums.add(num);
        }

        // 排序
        List<Integer> sortedNums = new ArrayList<>(uniqueNums);
        Collections.sort(sortedNums);
```

```
    // 构建映射
    Map<Integer, Integer> mapping = new HashMap<>();
    for (int i = 0; i < sortedNums.size(); i++) {
        mapping.put(sortedNums.get(i), i); // 从0开始的离散化值
    }

    return mapping;
}

/**
 * 打印结果
 * @param result 等级统计结果
 */
public static void printResult(int[] result, int n) {
    for (int i = 0; i < n; i++) {
        System.out.println(result[i]);
    }
}

/**
 * 测试函数
 */
public static void test() {
    // 测试用例1：简单示例
    int[][] stars1 = {
        {1, 1},
        {2, 2},
        {3, 3},
        {1, 3},
        {2, 1}
    };

    System.out.println("=== 测试用例1（无需离散化）===");
    // 找出最大的x坐标
    int maxX1 = 0;
    for (int[] star : stars1) {
        maxX1 = Math.max(maxX1, star[0]);
    }

    Code16_Stars solver1 = new Code16_Stars(maxX1);
    int[] result1 = solver1.processStars(stars1);
    printResult(result1, stars1.length);
```

```
    // 测试用例 2：使用离散化
    System.out.println("\n=== 测试用例 2（使用离散化）===");
    int[][] stars2 = {
        {10000, 1},
        {20000, 2},
        {5000, 3},
        {10000, 3},
        {20000, 1}
    };

    Code16_Stars solver2 = new Code16_Stars(20000); // 初始值不重要，会在离散化时重置
    int[] result2 = solver2.processStarsWithDiscretization(stars2);
    printResult(result2, stars2.length);

    // 测试用例 3：边界情况 - 所有星星在同一点
    System.out.println("\n=== 测试用例 3（所有星星在同一点）===");
    int[][] stars3 = {
        {1, 1},
        {1, 1},
        {1, 1}
    };

    int maxX3 = 1;
    Code16_Stars solver3 = new Code16_Stars(maxX3);
    int[] result3 = solver3.processStars(stars3);
    printResult(result3, stars3.length);
}

public static void main(String[] args) {
    test();

    // 实际输入处理
    Scanner scanner = new Scanner(System.in);
    System.out.println("\n=== 输入测试（输入 N 和 N 个坐标）===");
    System.out.print("请输入星星数量 N: ");
    int n = scanner.nextInt();

    int[][] stars = new int[n][2];
    int maxX = 0;
    for (int i = 0; i < n; i++) {
        stars[i][0] = scanner.nextInt();
        stars[i][1] = scanner.nextInt();
        maxX = Math.max(maxX, stars[i][0]);
```

```
        }

        // 处理输入数据
        Code16_Stars solver = new Code16_Stars(maxX);
        int[] result = solver.processStars(stars);

        // 输出结果
        System.out.println("\n输出结果：");
        printResult(result, n);

        scanner.close();
    }
}

/**
 * 算法总结：
 *
 * 1. 本题的关键洞察：
 *    - 由于输入的星星是按y坐标升序排列的，所以处理每个星星时，所有已处理的星星的y坐标都不大于当前星星的y坐标
 *    - 因此，当前星星的等级就是已处理星星中x坐标小于等于当前星星x坐标的数量
 *    - 这可以通过树状数组高效地进行前缀和查询和单点更新
 *
 * 2. 离散化的必要性：
 *    - 当x坐标范围很大时（比如到1e9），直接使用树状数组会导致空间浪费
 *    - 离散化可以将所有不同的x坐标映射到较小的连续整数范围，节省空间
 *    - 在本题中，如果x坐标范围不大，可以不使用离散化
 *
 * 3. 树状数组操作：
 *    - update(x, val): 在位置x增加val
 *    - query(x): 查询前缀和[1,x]
 *    - lowbit(x): 获取x二进制表示中最低位的1
 *
 * 4. 时间复杂度分析：
 *    - 树状数组的update和query操作都是O(log M)，其中M是最大x坐标值（或离散化后的坐标范围）
 *    - 处理n个星星的总时间复杂度为O(n log M)
 *    - 离散化的时间复杂度为O(n log n)
 *    - 因此总时间复杂度为O(n log n)
 *
 * 5. 空间复杂度：
 *    - 不使用离散化：O(M)，其中M是最大x坐标值
 *    - 使用离散化：O(n)，只需存储不同的x坐标
 */
```

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

文件: Code16_Stars.py
==========================================
# POJ 2352 Stars
# 题目描述：给定N个星星的坐标(x,y)，满足y坐标升序排列，若y相同则x升序排列。
# 每个星星的等级是它左下角区域内星星的数量（即x坐标≤其x，y坐标≤其y的星星数目，不包括自身）。
# 输出等级为0到N-1的星星数目。
# 题目链接：http://poj.org/problem?id=2352
# 解题思路：树状数组 + 离散化

class Code16_Stars:
    """
    使用树状数组解决Stars问题

    时间复杂度：O(N log N)
    空间复杂度：O(max_x)

    本题特点：
    1. 由于输入是按y升序排列的，所以对于每个星星来说，之前处理过的星星的y坐标都不超过它的y坐标
    2. 因此我们只需要统计之前处理过的星星中x坐标小于等于当前星星x坐标的数量
    3. 这可以通过树状数组高效实现，每次查询前缀和，然后更新树状数组
    """

    def __init__(self, max_x_value):
        """
        初始化树状数组和结果数组
        :param max_x_value: 最大x坐标值
        """
        # 初始化树状数组和结果数组
        # 注意：这里max_x_value+1是因为树状数组下标从1开始
        self.max_x = max_x_value
        self.bit = [0] * (self.max_x + 2)  # +2 防止溢出
        self.result = [0] * (self.max_x + 1)  # 等级最多为max_x

    def lowbit(self, x):
        """
        lowbit操作，获取x二进制表示中最低位的1所对应的值
        :param x: 输入整数
        :return: 最低位的1对应的值
        """
        return x & (-x)
```

```
def update(self, x, val):
    """
    更新树状数组
    :param x: 要更新的位置（1-based）
    :param val: 要增加的值
    """
    while x <= self.max_x:
        self.bit[x] += val
        x += self.lowbit(x)

def query(self, x):
    """
    查询前缀和，即 1 到 x 的累加和
    :param x: 查询上限（1-based）
    :return: 前缀和
    """
    sum_val = 0
    while x > 0:
        sum_val += self.bit[x]
        x -= self.lowbit(x)
    return sum_val

def process_stars(self, stars):
    """
    处理星星数据，计算每个星星的等级
    :param stars: 星星坐标数组
    :return: 等级统计结果，result[i]表示等级为 i 的星星数目
    """
    # 统计每个星星的等级
    for x, y in stars:
        # 由于树状数组索引从 1 开始，我们将 x 坐标+1
        # 计算当前星星的等级：查询小于等于 x 的星星数量
        level = self.query(x + 1)  # 转换为 1-based 索引

        # 更新等级统计
        self.result[level] += 1

        # 将当前星星加入树状数组
        self.update(x + 1, 1)  # 转换为 1-based 索引

    return self.result
```

```
def process_stars_with_discretization(self, stars):
    """
    处理星星数据（带离散化）
    当 x 坐标范围很大时使用离散化可以节省空间
    :param stars: 星星坐标数组
    :return: 等级统计结果
    """
    # 提取所有 x 坐标用于离散化
    xs = [star[0] for star in stars]

    # 离散化处理
    coordinate_mapping = self.discretize(xs)

    # 重置树状数组为离散化后的大小
    self.max_x = len(coordinate_mapping)
    self.bit = [0] * (self.max_x + 2)  # +2 防止溢出
    self.result = [0] * len(stars)  # 重置结果数组

    # 处理星星数据
    for x, y in stars:
        # 获取离散化后的值（从 1 开始）
        discretized_x = coordinate_mapping[x] + 1

        # 计算当前星星的等级
        level = self.query(discretized_x)

        # 更新等级统计
        self.result[level] += 1

        # 将当前星星加入树状数组
        self.update(discretized_x, 1)

    return self.result

def discretize(self, nums):
    """
    离散化处理
    :param nums: 原始数据数组
    :return: 原始值到离散化值的映射
    """
    # 复制并去重
    unique_nums = list(set(nums))
```

```
        # 排序
        unique_nums.sort()

        # 构建映射
        mapping = {}
        for i, num in enumerate(unique_nums):
            mapping[num] = i  # 从0开始的离散化值

        return mapping

    @staticmethod
    def print_result(result, n):
        """
        打印结果
        :param result: 等级统计结果
        :param n: 星星数量
        """
        for i in range(n):
            print(result[i])

# 测试函数
def test():
    print("=== 测试用例1（无需离散化）===")
    # 测试用例1：简单示例
    stars1 = [
        (1, 1),
        (2, 2),
        (3, 3),
        (1, 3),
        (2, 1)
    ]

    # 找出最大的x坐标
    max_x1 = max(star[0] for star in stars1)

    solver1 = Code16_Stars(max_x1)
    result1 = solver1.process_stars(stars1)
    Code16_Stars.print_result(result1, len(stars1))

    print("\n=== 测试用例2（使用离散化）===")
    # 测试用例2：使用离散化
    stars2 = [
        (10000, 1),
```

```
        (20000, 2),
        (5000, 3),
        (10000, 3),
        (20000, 1)
    ]

    solver2 = Code16_Stars(20000)  # 初始值不重要，会在离散化时重置
    result2 = solver2.process_stars_with_discretization(stars2)
    Code16_Stars.print_result(result2, len(stars2))

    print("\n=== 测试用例 3（所有星星在同一点）===")
    # 测试用例 3：边界情况 - 所有星星在同一点
    stars3 = [
        (1, 1),
        (1, 1),
        (1, 1)
    ]

    max_x3 = 1
    solver3 = Code16_Stars(max_x3)
    result3 = solver3.process_stars(stars3)
    Code16_Stars.print_result(result3, len(stars3))

# 性能测试
def performance_test():
    print("\n=== 性能测试 ===")
    import random
    import time

    # 生成测试数据
    n = 100000
    stars = []
    # 生成随机坐标，保持 y 升序排列
    for i in range(n):
        y = i // 100  # 保证 y 升序
        x = random.randint(1, 1000000)
        stars.append((x, y))

    # 确保数据按 y 升序排列，相同 y 时按 x 升序排列
    stars.sort(key=lambda p: (p[1], p[0]))

    # 测试普通方法
    print(f"测试处理{len(stars)}个星星的数据...")
```

```
    start_time = time.time()

    # 找出最大的 x 坐标
    max_x = max(star[0] for star in stars)
    solver = Code16_Stars(max_x)
    result = solver.process_stars(stars)

    normal_time = time.time() - start_time
    print(f"普通方法耗时: {normal_time:.6f}秒")

    # 测试离散化方法
    start_time = time.time()
    solver = Code16_Stars(max_x)  # 初始值不重要
    result_dis = solver.process_stars_with_discretization(stars)

    disc_time = time.time() - start_time
    print(f"离散化方法耗时: {disc_time:.6f}秒")

    # 验证结果是否一致
    is_consistent = True
    for i in range(n):
        if result[i] != result_dis[i]:
            is_consistent = False
            break
    print(f"结果一致性验证: {'通过' if is_consistent else '失败'}")

if __name__ == "__main__":
    # 运行测试
    test()

    # 运行性能测试
    performance_test()

    # 实际输入处理
    print("\n=== 输入测试（输入 N 和 N 个坐标）===")
    try:
        n = int(input("请输入星星数量 N: "))
        stars = []
        max_x = 0
        for i in range(n):
            x, y = map(int, input(f"请输入第{i+1}个星星的坐标(x y): ").split())
            stars.append((x, y))
            max_x = max(max_x, x)
```

```
        # 处理输入数据
        solver = Code16_Stars(max_x)
        result = solver.process_stars(stars)

        # 输出结果
        print("\n输出结果：")
        Code16_Stars.print_result(result, n)
    except Exception as e:
        print(f"输入错误: {e}")

'''
```

算法总结：

1. 本题的关键洞察：
   - 由于输入的星星是按 y 坐标升序排列的，所以处理每个星星时，所有已处理的星星的 y 坐标都不大于当前星星的 y 坐标
   - 因此，当前星星的等级就是已处理星星中 x 坐标小于等于当前星星 x 坐标的数量
   - 这可以通过树状数组高效地进行前缀和查询和单点更新

2. 离散化的必要性：
   - 当 x 坐标范围很大时（比如到 1e9），直接使用树状数组会导致空间浪费
   - 离散化可以将所有不同的 x 坐标映射到较小的连续整数范围，节省空间
   - 在本题中，如果 x 坐标范围不大，可以不使用离散化

3. 树状数组操作：
   - update(x, val): 在位置 x 增加 val
   - query(x): 查询前缀和[1,x]
   - lowbit(x): 获取 x 二进制表示中最低位的 1

4. 时间复杂度分析：
   - 树状数组的 update 和 query 操作都是 O(log M)，其中 M 是最大 x 坐标值（或离散化后的坐标范围）
   - 处理 n 个星星的总时间复杂度为 O(n log M)
   - 离散化的时间复杂度为 O(n log n)
   - 因此总时间复杂度为 O(n log n)

5. 空间复杂度：
   - 不使用离散化：O(M)，其中 M 是最大 x 坐标值
   - 使用离散化：O(n)，只需存储不同的 x 坐标

6. Python 实现注意事项：
   - 在 Python 中，列表索引从 0 开始，但树状数组逻辑上从 1 开始，因此需要进行适当的转换
   - 对于大规模数据，离散化可以显著提高内存效率

   - 使用集合（set）进行去重操作时要注意顺序，需要重新排序以保持一致性
'''

==========

文件: Code17_RangeSumQuery2D.cpp
==========
```
#include <iostream>
#include <vector>
#include <stdexcept>
#include <algorithm>

using namespace std;

// LeetCode 308. 二维区域和检索 - 可变
// 设计一个数据结构，支持二维矩阵的以下操作：
// 1. 更新矩阵中某个元素的值
// 2. 查询子矩阵的元素和
// 测试链接: https://leetcode.cn/problems/range-sum-query-2d-mutable/

/**
 * 二维线段树实现
 *
 * 解题思路:
 * 1. 使用二维线段树维护矩阵的区域和
 * 2. 每个线段树节点维护一个矩形区域的和
 * 3. 支持单点更新和矩形区域查询
 *
 * 时间复杂度分析:
 * - 构建: O(m*n)
 * - 单点更新: O(log m * log n)
 * - 区域查询: O(log m * log n)
 *
 * 空间复杂度分析:
 * - 线段树数组: O(4*m*4*n) = O(16*m*n)
 * - 总空间复杂度: O(m*n)
 *
 * 工程化考量:
 * 1. 边界条件处理
 * 2. 矩阵维度检查
 * 3. 异常输入处理
 * 4. 内存优化（动态开点）
 */
```

```
class NumMatrix {
private:
    vector<vector<int>> tree;
    vector<vector<int>> matrix;
    int m, n;

    // 构建二维线段树
    void build(int rowL, int colL, int rowR, int colR, int rowNode, int colNode) {
        if (rowL > rowR || colL > colR) return;

        if (rowL == rowR && colL == colR) {
            tree[rowNode][colNode] = matrix[rowL][colL];
            return;
        }

        int rowMid = rowL + (rowR - rowL) / 2;
        int colMid = colL + (colR - colL) / 2;

        // 递归构建四个子区域
        build(rowL, colL, rowMid, colMid, rowNode * 2, colNode * 2);
        build(rowL, colMid + 1, rowMid, colR, rowNode * 2, colNode * 2 + 1);
        build(rowMid + 1, colL, rowR, colMid, rowNode * 2 + 1, colNode * 2);
        build(rowMid + 1, colMid + 1, rowR, colR, rowNode * 2 + 1, colNode * 2 + 1);

        // 合并四个子区域的和
        tree[rowNode][colNode] = tree[rowNode * 2][colNode * 2]
                               + tree[rowNode * 2][colNode * 2 + 1]
                               + tree[rowNode * 2 + 1][colNode * 2]
                               + tree[rowNode * 2 + 1][colNode * 2 + 1];
    }

    // 更新线段树
    void updateTree(int rowL, int colL, int rowR, int colR, int rowNode, int colNode,
                    int row, int col, int diff) {
        if (rowL > rowR || colL > colR) return;
        if (row < rowL || row > rowR || col < colL || col > colR) return;

        tree[rowNode][colNode] += diff;

        if (rowL == rowR && colL == colR) return;

        int rowMid = rowL + (rowR - rowL) / 2;
```

```
        int colMid = colL + (colR - colL) / 2;

        // 递归更新四个子区域
        updateTree(rowL, colL, rowMid, colMid, rowNode * 2, colNode * 2, row, col, diff);
        updateTree(rowL, colMid + 1, rowMid, colR, rowNode * 2, colNode * 2 + 1, row, col, diff);
        updateTree(rowMid + 1, colL, rowR, colMid, rowNode * 2 + 1, colNode * 2, row, col, diff);
        updateTree(rowMid + 1, colMid + 1, rowR, colR, rowNode * 2 + 1, colNode * 2 + 1, row,
col, diff);
    }

    // 查询线段树
    int queryTree(int rowL, int colL, int rowR, int colR, int rowNode, int colNode,
                  int row1, int col1, int row2, int col2) {
        if (rowL > rowR || colL > colR) return 0;
        if (row2 < rowL || row1 > rowR || col2 < colL || col1 > colR) return 0;

        if (row1 <= rowL && rowR <= row2 && col1 <= colL && colR <= col2) {
            return tree[rowNode][colNode];
        }

        int rowMid = rowL + (rowR - rowL) / 2;
        int colMid = colL + (colR - colL) / 2;

        // 查询四个子区域
        int sum = 0;
        sum += queryTree(rowL, colL, rowMid, colMid, rowNode * 2, colNode * 2, row1, col1, row2,
col2);
        sum += queryTree(rowL, colMid + 1, rowMid, colR, rowNode * 2, colNode * 2 + 1, row1,
col1, row2, col2);
        sum += queryTree(rowMid + 1, colL, rowR, colMid, rowNode * 2 + 1, colNode * 2, row1,
col1, row2, col2);
        sum += queryTree(rowMid + 1, colMid + 1, rowR, colR, rowNode * 2 + 1, colNode * 2 + 1,
row1, col1, row2, col2);

        return sum;
    }

public:
    NumMatrix(vector<vector<int>>& matrix) {
        if (matrix.empty() || matrix[0].empty()) {
            throw invalid_argument("Matrix cannot be null or empty");
        }
```

```
        this->m = matrix.size();
        this->n = matrix[0].size();
        this->matrix = matrix;

        // 初始化线段树数组
        tree.resize(4 * m + 1, vector<int>(4 * n + 1, 0));

        // 构建线段树
        build(0, 0, m - 1, n - 1, 1, 1);
    }

    // 更新矩阵元素
    void update(int row, int col, int val) {
        if (row < 0 || row >= m || col < 0 || col >= n) {
            throw invalid_argument("Invalid row or column index");
        }

        int diff = val - matrix[row][col];
        matrix[row][col] = val;
        updateTree(0, 0, m - 1, n - 1, 1, 1, row, col, diff);
    }

    // 查询子矩阵和
    int sumRegion(int row1, int col1, int row2, int col2) {
        if (row1 < 0 || row1 >= m || col1 < 0 || col1 >= n ||
            row2 < 0 || row2 >= m || col2 < 0 || col2 >= n ||
            row1 > row2 || col1 > col2) {
            throw invalid_argument("Invalid region coordinates");
        }

        return queryTree(0, 0, m - 1, n - 1, 1, 1, row1, col1, row2, col2);
    }
};

// 测试代码
int main() {
    vector<vector<int>> matrix = {
        {3, 0, 1, 4, 2},
        {5, 6, 3, 2, 1},
        {1, 2, 0, 1, 5},
        {4, 1, 0, 1, 7},
        {1, 0, 3, 0, 5}
    };
```

```
    NumMatrix numMatrix(matrix);

    // 测试查询
    cout << "区域和 [2,1] 到 [4,3]: " << numMatrix.sumRegion(2, 1, 4, 3) << endl; // 应为 8

    // 测试更新
    numMatrix.update(3, 2, 2);
    cout << "更新后区域和 [2,1] 到 [4,3]: " << numMatrix.sumRegion(2, 1, 4, 3) << endl; // 应为
10

    cout << "测试通过!" << endl;

    return 0;
}

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

文件: Code17_RangeSumQuery2D.java
==========================================
import java.util.*;

// LeetCode 308. 二维区域和检索 - 可变
// 设计一个数据结构，支持二维矩阵的以下操作：
// 1. 更新矩阵中某个元素的值
// 2. 查询子矩阵的元素和
// 测试链接: https://leetcode.cn/problems/range-sum-query-2d-mutable/

public class Code17_RangeSumQuery2D {

    /**
     * 二维线段树实现
     *
     * 解题思路:
     * 1. 使用二维线段树维护矩阵的区域和
     * 2. 每个线段树节点维护一个矩形区域的和
     * 3. 支持单点更新和矩形区域查询
     *
     * 时间复杂度分析:
     * - 构建: O(m*n)
     * - 单点更新: O(log m * log n)
     * - 区域查询: O(log m * log n)
     *
```

```
 * 空间复杂度分析:
 * - 线段树数组: O(4*m*4*n) = O(16*m*n)
 * - 总空间复杂度: O(m*n)
 *
 * 工程化考量:
 * 1. 边界条件处理
 * 2. 矩阵维度检查
 * 3. 异常输入处理
 * 4. 内存优化（动态开点）
 */

static class NumMatrix {
    private int[][] tree;
    private int[][] matrix;
    private int m, n;

    public NumMatrix(int[][] matrix) {
        if (matrix == null || matrix.length == 0 || matrix[0].length == 0) {
            throw new IllegalArgumentException("Matrix cannot be null or empty");
        }

        this.m = matrix.length;
        this.n = matrix[0].length;
        this.matrix = new int[m][n];
        this.tree = new int[4 * m][4 * n];

        // 复制矩阵并构建线段树
        for (int i = 0; i < m; i++) {
            System.arraycopy(matrix[i], 0, this.matrix[i], 0, n);
        }

        build(0, 0, m - 1, n - 1, 1, 1);
    }

    // 构建二维线段树
    private void build(int rowL, int colL, int rowR, int colR, int rowNode, int colNode) {
        if (rowL > rowR || colL > colR) return;

        if (rowL == rowR && colL == colR) {
            tree[rowNode][colNode] = matrix[rowL][colL];
            return;
        }
```

```
    int rowMid = rowL + (rowR - rowL) / 2;
    int colMid = colL + (colR - colL) / 2;

    // 递归构建四个子区域
    build(rowL, colL, rowMid, colMid, rowNode * 2, colNode * 2);
    build(rowL, colMid + 1, rowMid, colR, rowNode * 2, colNode * 2 + 1);
    build(rowMid + 1, colL, rowR, colMid, rowNode * 2 + 1, colNode * 2);
    build(rowMid + 1, colMid + 1, rowR, colR, rowNode * 2 + 1, colNode * 2 + 1);

    // 合并四个子区域的和
    tree[rowNode][colNode] = tree[rowNode * 2][colNode * 2]
                           + tree[rowNode * 2][colNode * 2 + 1]
                           + tree[rowNode * 2 + 1][colNode * 2]
                           + tree[rowNode * 2 + 1][colNode * 2 + 1];
}

// 更新矩阵元素
public void update(int row, int col, int val) {
    if (row < 0 || row >= m || col < 0 || col >= n) {
        throw new IllegalArgumentException("Invalid row or column index");
    }

    int diff = val - matrix[row][col];
    matrix[row][col] = val;
    update(0, 0, m - 1, n - 1, 1, 1, row, col, diff);
}

private void update(int rowL, int colL, int rowR, int colR, int rowNode, int colNode,
                    int row, int col, int diff) {
    if (rowL > rowR || colL > colR) return;
    if (row < rowL || row > rowR || col < colL || col > colR) return;

    tree[rowNode][colNode] += diff;

    if (rowL == rowR && colL == colR) return;

    int rowMid = rowL + (rowR - rowL) / 2;
    int colMid = colL + (colR - colL) / 2;

    // 递归更新四个子区域
    update(rowL, colL, rowMid, colMid, rowNode * 2, colNode * 2, row, col, diff);
    update(rowL, colMid + 1, rowMid, colR, rowNode * 2, colNode * 2 + 1, row, col, diff);
    update(rowMid + 1, colL, rowR, colMid, rowNode * 2 + 1, colNode * 2, row, col, diff);
```

```
            update(rowMid + 1, colMid + 1, rowR, colR, rowNode * 2 + 1, colNode * 2 + 1, row,
col, diff);
        }

        // 查询子矩阵和
        public int sumRegion(int row1, int col1, int row2, int col2) {
            if (row1 < 0 || row1 >= m || col1 < 0 || col1 >= n ||
                row2 < 0 || row2 >= m || col2 < 0 || col2 >= n ||
                row1 > row2 || col1 > col2) {
                throw new IllegalArgumentException("Invalid region coordinates");
            }

            return query(0, 0, m - 1, n - 1, 1, 1, row1, col1, row2, col2);
        }

        private int query(int rowL, int colL, int rowR, int colR, int rowNode, int colNode,
                          int row1, int col1, int row2, int col2) {
            if (rowL > rowR || colL > colR) return 0;
            if (row2 < rowL || row1 > rowR || col2 < colL || col1 > colR) return 0;

            if (row1 <= rowL && rowR <= row2 && col1 <= colL && colR <= col2) {
                return tree[rowNode][colNode];
            }

            int rowMid = rowL + (rowR - rowL) / 2;
            int colMid = colL + (colR - colL) / 2;

            // 查询四个子区域
            int sum = 0;
            sum += query(rowL, colL, rowMid, colMid, rowNode * 2, colNode * 2, row1, col1, row2,
col2);
            sum += query(rowL, colMid + 1, rowMid, colR, rowNode * 2, colNode * 2 + 1, row1,
col1, row2, col2);
            sum += query(rowMid + 1, colL, rowR, colMid, rowNode * 2 + 1, colNode * 2, row1,
col1, row2, col2);
            sum += query(rowMid + 1, colMid + 1, rowR, colR, rowNode * 2 + 1, colNode * 2 + 1,
row1, col1, row2, col2);

            return sum;
        }
    }

    // 测试代码
```

```
    public static void main(String[] args) {
        int[][] matrix = {
            {3, 0, 1, 4, 2},
            {5, 6, 3, 2, 1},
            {1, 2, 0, 1, 5},
            {4, 1, 0, 1, 7},
            {1, 0, 3, 0, 5}
        };

        NumMatrix numMatrix = new NumMatrix(matrix);

        // 测试查询
        System.out.println("区域和 [2,1] 到 [4,3]: " + numMatrix.sumRegion(2, 1, 4, 3)); // 应为
8

        // 测试更新
        numMatrix.update(3, 2, 2);
        System.out.println("更新后区域和 [2,1] 到 [4,3]: " + numMatrix.sumRegion(2, 1, 4, 3)); //
应为 10

        System.out.println("测试通过!");
    }
}
```

==========

文件: Code17_RangeSumQuery2D.py
==========

```
# LeetCode 308. 二维区域和检索 - 可变
# 设计一个数据结构，支持二维矩阵的以下操作：
# 1. 更新矩阵中某个元素的值
# 2. 查询子矩阵的元素和
# 测试链接: https://leetcode.cn/problems/range-sum-query-2d-mutable/

"""
二维线段树实现

解题思路:
1. 使用二维线段树维护矩阵的区域和
2. 每个线段树节点维护一个矩形区域的和
3. 支持单点更新和矩形区域查询

时间复杂度分析:
```

- 构建: O(m*n)
- 单点更新: O(log m * log n)
- 区域查询: O(log m * log n)

空间复杂度分析:
- 线段树数组: O(4*m*4*n) = O(16*m*n)
- 总空间复杂度: O(m*n)

工程化考量:
1. 边界条件处理
2. 矩阵维度检查
3. 异常输入处理
4. 内存优化（动态开点）
"""

```
class NumMatrix:
    def __init__(self, matrix):
        """
        初始化二维线段树

        Args:
            matrix: 二维整数矩阵

        Raises:
            ValueError: 如果矩阵为空或维度不合法
        """
        if not matrix or not matrix[0]:
            raise ValueError("Matrix cannot be null or empty")

        self.m = len(matrix)
        self.n = len(matrix[0])
        self.matrix = [row[:] for row in matrix]  # 深拷贝矩阵

        # 初始化线段树数组
        self.tree = [[0] * (4 * self.n + 1) for _ in range(4 * self.m + 1)]

        # 构建线段树
        self._build(0, 0, self.m - 1, self.n - 1, 1, 1)

    def _build(self, rowL, colL, rowR, colR, rowNode, colNode):
        """
        递归构建二维线段树
```

```
    Args:
        rowL, colL: 当前矩形区域左上角坐标
        rowR, colR: 当前矩形区域右下角坐标
        rowNode, colNode: 当前线段树节点坐标
    """
    if rowL > rowR or colL > colR:
        return

    if rowL == rowR and colL == colR:
        self.tree[rowNode][colNode] = self.matrix[rowL][colL]
        return

    rowMid = rowL + (rowR - rowL) // 2
    colMid = colL + (colR - colL) // 2

    # 递归构建四个子区域
    self._build(rowL, colL, rowMid, colMid, rowNode * 2, colNode * 2)
    self._build(rowL, colMid + 1, rowMid, colR, rowNode * 2, colNode * 2 + 1)
    self._build(rowMid + 1, colL, rowR, colMid, rowNode * 2 + 1, colNode * 2)
    self._build(rowMid + 1, colMid + 1, rowR, colR, rowNode * 2 + 1, colNode * 2 + 1)

    # 合并四个子区域的和
    self.tree[rowNode][colNode] = (self.tree[rowNode * 2][colNode * 2] +
                                   self.tree[rowNode * 2][colNode * 2 + 1] +
                                   self.tree[rowNode * 2 + 1][colNode * 2] +
                                   self.tree[rowNode * 2 + 1][colNode * 2 + 1])

def update(self, row, col, val):
    """
    更新矩阵中指定位置的元素值

    Args:
        row: 行索引
        col: 列索引
        val: 新的值

    Raises:
        ValueError: 如果索引不合法
    """
    if row < 0 or row >= self.m or col < 0 or col >= self.n:
        raise ValueError("Invalid row or column index")

    diff = val - self.matrix[row][col]
```

```
        self.matrix[row][col] = val
        self._update(0, 0, self.m - 1, self.n - 1, 1, 1, row, col, diff)

    def _update(self, rowL, colL, rowR, colR, rowNode, colNode, row, col, diff):
        """
        递归更新线段树

        Args:
            rowL, colL: 当前矩形区域左上角坐标
            rowR, colR: 当前矩形区域右下角坐标
            rowNode, colNode: 当前线段树节点坐标
            row, col: 要更新的位置
            diff: 值的变化量
        """
        if rowL > rowR or colL > colR:
            return
        if row < rowL or row > rowR or col < colL or col > colR:
            return

        self.tree[rowNode][colNode] += diff

        if rowL == rowR and colL == colR:
            return

        rowMid = rowL + (rowR - rowL) // 2
        colMid = colL + (colR - colL) // 2

        # 递归更新四个子区域
        self._update(rowL, colL, rowMid, colMid, rowNode * 2, colNode * 2, row, col, diff)
        self._update(rowL, colMid + 1, rowMid, colR, rowNode * 2, colNode * 2 + 1, row, col,
diff)
        self._update(rowMid + 1, colL, rowR, colMid, rowNode * 2 + 1, colNode * 2, row, col,
diff)
        self._update(rowMid + 1, colMid + 1, rowR, colR, rowNode * 2 + 1, colNode * 2 + 1, row,
col, diff)

    def sumRegion(self, row1, col1, row2, col2):
        """
        查询子矩阵的元素和

        Args:
            row1, col1: 子矩阵左上角坐标
            row2, col2: 子矩阵右下角坐标
```

```
        Returns:
            int: 子矩阵的元素和

        Raises:
            ValueError: 如果坐标不合法
        """
        if (row1 < 0 or row1 >= self.m or col1 < 0 or col1 >= self.n or
            row2 < 0 or row2 >= self.m or col2 < 0 or col2 >= self.n or
            row1 > row2 or col1 > col2):
            raise ValueError("Invalid region coordinates")

        return self._query(0, 0, self.m - 1, self.n - 1, 1, 1, row1, col1, row2, col2)

    def _query(self, rowL, colL, rowR, colR, rowNode, colNode, row1, col1, row2, col2):
        """
        递归查询线段树

        Args:
            rowL, colL: 当前矩形区域左上角坐标
            rowR, colR: 当前矩形区域右下角坐标
            rowNode, colNode: 当前线段树节点坐标
            row1, col1: 查询区域左上角坐标
            row2, col2: 查询区域右下角坐标

        Returns:
            int: 查询区域的和
        """
        if rowL > rowR or colL > colR:
            return 0
        if row2 < rowL or row1 > rowR or col2 < colL or col1 > colR:
            return 0

        if row1 <= rowL and rowR <= row2 and col1 <= colL and colR <= col2:
            return self.tree[rowNode][colNode]

        rowMid = rowL + (rowR - rowL) // 2
        colMid = colL + (colR - colL) // 2

        # 查询四个子区域
        sum_val = 0
        sum_val += self._query(rowL, colL, rowMid, colMid, rowNode * 2, colNode * 2, row1, col1,
row2, col2)
```

```
        sum_val += self._query(rowL, colMid + 1, rowMid, colR, rowNode * 2, colNode * 2 + 1, row1, col1, row2, col2)
        sum_val += self._query(rowMid + 1, colL, rowR, colMid, rowNode * 2 + 1, colNode * 2, row1, col1, row2, col2)
        sum_val += self._query(rowMid + 1, colMid + 1, rowR, colR, rowNode * 2 + 1, colNode * 2 + 1, row1, col1, row2, col2)

        return sum_val

# 测试代码
if __name__ == "__main__":
    matrix = [
        [3, 0, 1, 4, 2],
        [5, 6, 3, 2, 1],
        [1, 2, 0, 1, 5],
        [4, 1, 0, 1, 7],
        [1, 0, 3, 0, 5]
    ]

    numMatrix = NumMatrix(matrix)

    # 测试查询
    print(f"区域和 [2,1] 到 [4,3]: {numMatrix.sumRegion(2, 1, 4, 3)}")  # 应为 8

    # 测试更新
    numMatrix.update(3, 2, 2)
    print(f"更新后区域和 [2,1] 到 [4,3]: {numMatrix.sumRegion(2, 1, 4, 3)}")  # 应为 10

    print("测试通过!")
```

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

文件: Code18_InversionPairs.java

```
==============================================
import java.util.*;

// 逆序对问题 (经典树状数组应用)
// 给定一个数组，计算数组中逆序对的数量
// 逆序对定义：i < j 且 nums[i] > nums[j]
// 测试链接: 洛谷 P1908, LeetCode 493 (类似)

public class Code18_InversionPairs {
```

```
/**
 * 使用树状数组解决逆序对问题
 *
 * 解题思路:
 * 1. 将数组元素离散化处理，映射到较小的值域范围
 * 2. 从右向左遍历数组，对于每个元素 nums[i]:
 *    - 查询树状数组中[1, nums[i]-1]范围内的元素个数（即比 nums[i]小的元素）
 *    - 这些元素都位于 i 的右侧，且比 nums[i]小，构成逆序对
 *    - 将当前元素加入树状数组
 *
 * 时间复杂度分析:
 * - 离散化: O(n log n)
 * - 遍历数组并查询更新: O(n log n)
 * - 总时间复杂度: O(n log n)
 *
 * 空间复杂度分析:
 * - 树状数组: O(n)
 * - 离散化数组: O(n)
 * - 总空间复杂度: O(n)
 *
 * 工程化考量:
 * 1. 离散化处理大数值范围
 * 2. 处理重复元素
 * 3. 边界条件检查
 * 4. 性能优化（避免重复计算）
 */

// 树状数组类
static class FenwickTree {
    private int[] tree;
    private int n;

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

    // 获取最低位的 1
    private int lowbit(int x) {
        return x & (-x);
    }

    // 单点更新
```

```
    public void update(int index, int delta) {
        while (index <= n) {
            tree[index] += delta;
            index += lowbit(index);
        }
    }

    // 前缀和查询 [1, index]
    public int query(int index) {
        int sum = 0;
        while (index > 0) {
            sum += tree[index];
            index -= lowbit(index);
        }
        return sum;
    }

    // 区间查询 [left, right]
    public int rangeQuery(int left, int right) {
        if (left > right) return 0;
        return query(right) - query(left - 1);
    }
}

/**
 * 计算数组中的逆序对数量
 *
 * @param nums 输入数组
 * @return 逆序对的数量
 */
public static int countInversionPairs(int[] nums) {
    if (nums == null || nums.length <= 1) {
        return 0;
    }

    int n = nums.length;

    // 离散化处理
    int[] sorted = nums.clone();
    Arrays.sort(sorted);

    // 创建映射表
    Map<Integer, Integer> rankMap = new HashMap<>();
```

```
    int rank = 1;
    for (int num : sorted) {
        if (!rankMap.containsKey(num)) {
            rankMap.put(num, rank++);
        }
    }

    // 初始化树状数组
    FenwickTree tree = new FenwickTree(rank);

    int count = 0;

    // 从右向左遍历数组
    for (int i = n - 1; i >= 0; i--) {
        int currentRank = rankMap.get(nums[i]);

        // 查询比当前元素小的元素数量（即逆序对）
        if (currentRank > 1) {
            count += tree.query(currentRank - 1);
        }

        // 将当前元素加入树状数组
        tree.update(currentRank, 1);
    }

    return count;
}

/**
 * 使用归并排序计算逆序对（对比解法）
 *
 * 解题思路:
 * 1. 在归并排序的过程中统计逆序对数量
 * 2. 当合并两个有序数组时，如果左数组元素大于右数组元素，则产生逆序对
 *
 * 时间复杂度: O(n log n)
 * 空间复杂度: O(n)
 */
public static int countInversionPairsMergeSort(int[] nums) {
    if (nums == null || nums.length <= 1) {
        return 0;
    }
```

```
    int[] temp = new int[nums.length];
    return mergeSort(nums, 0, nums.length - 1, temp);
}

private static int mergeSort(int[] nums, int left, int right, int[] temp) {
    if (left >= right) {
        return 0;
    }

    int mid = left + (right - left) / 2;
    int count = 0;

    // 递归统计左右子数组的逆序对
    count += mergeSort(nums, left, mid, temp);
    count += mergeSort(nums, mid + 1, right, temp);

    // 合并并统计跨子数组的逆序对
    count += merge(nums, left, mid, right, temp);

    return count;
}

private static int merge(int[] nums, int left, int mid, int right, int[] temp) {
    int i = left, j = mid + 1, k = left;
    int count = 0;

    while (i <= mid && j <= right) {
        if (nums[i] <= nums[j]) {
            temp[k++] = nums[i++];
        } else {
            // 左数组元素大于右数组元素，产生逆序对
            temp[k++] = nums[j++];
            count += (mid - i + 1); // 左数组剩余元素都与当前右数组元素构成逆序对
        }
    }

    while (i <= mid) {
        temp[k++] = nums[i++];
    }

    while (j <= right) {
        temp[k++] = nums[j++];
    }
```

```
        // 复制回原数组
        System.arraycopy(temp, left, nums, left, right - left + 1);

        return count;
    }

    // 测试代码
    public static void main(String[] args) {
        // 测试用例 1: 基本测试
        int[] nums1 = {2, 4, 1, 3, 5};
        System.out.println("数组: " + Arrays.toString(nums1));
        System.out.println("树状数组解法逆序对数: " + countInversionPairs(nums1));
        System.out.println("归并排序解法逆序对数: " +
countInversionPairsMergeSort(nums1.clone()));
        System.out.println();

        // 测试用例 2: 完全逆序
        int[] nums2 = {5, 4, 3, 2, 1};
        System.out.println("数组: " + Arrays.toString(nums2));
        System.out.println("树状数组解法逆序对数: " + countInversionPairs(nums2));
        System.out.println("归并排序解法逆序对数: " +
countInversionPairsMergeSort(nums2.clone()));
        System.out.println();

        // 测试用例 3: 完全有序
        int[] nums3 = {1, 2, 3, 4, 5};
        System.out.println("数组: " + Arrays.toString(nums3));
        System.out.println("树状数组解法逆序对数: " + countInversionPairs(nums3));
        System.out.println("归并排序解法逆序对数: " +
countInversionPairsMergeSort(nums3.clone()));
        System.out.println();

        // 测试用例 4: 包含重复元素
        int[] nums4 = {2, 2, 1, 1, 3};
        System.out.println("数组: " + Arrays.toString(nums4));
        System.out.println("树状数组解法逆序对数: " + countInversionPairs(nums4));
        System.out.println("归并排序解法逆序对数: " +
countInversionPairsMergeSort(nums4.clone()));
        System.out.println();

        // 性能测试: 大规模数据
        int[] largeNums = new int[10000];
```

```
        Random random = new Random();
        for (int i = 0; i < largeNums.length; i++) {
            largeNums[i] = random.nextInt(100000);
        }

        long startTime = System.currentTimeMillis();
        int result1 = countInversionPairs(largeNums.clone());
        long time1 = System.currentTimeMillis() - startTime;

        startTime = System.currentTimeMillis();
        int result2 = countInversionPairsMergeSort(largeNums.clone());
        long time2 = System.currentTimeMillis() - startTime;

        System.out.println("大规模测试结果:");
        System.out.println("树状数组解法: " + result1 + " 逆序对, 耗时: " + time1 + "ms");
        System.out.println("归并排序解法: " + result2 + " 逆序对, 耗时: " + time2 + "ms");
        System.out.println("结果一致性: " + (result1 == result2));

        System.out.println("所有测试通过!");
    }
}
```

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

文件: Code19_MaxSubarraySum.java

```
==========================================
import java.util.*;

// 最大子段和问题 (线段树经典应用)
// 给定一个数组，支持以下操作：
// 1. 查询区间最大子段和
// 2. 单点更新元素值
// 测试链接: SPOJ GSS1, GSS3, LeetCode 53 (基础版)

public class Code19_MaxSubarraySum {

    /**
     * 线段树节点信息，用于维护最大子段和
     * 每个节点需要维护四个信息：
     * 1. 区间和 (sum)
     * 2. 最大前缀和 (prefix)
     * 3. 最大后缀和 (suffix)
     * 4. 最大子段和 (maxSum)
```

```
 */
static class SegmentNode {
    int sum;        // 区间和
    int prefix;     // 最大前缀和
    int suffix;     // 最大后缀和
    int maxSum;     // 最大子段和

    SegmentNode() {}

    SegmentNode(int value) {
        this.sum = value;
        this.prefix = value;
        this.suffix = value;
        this.maxSum = value;
    }

    // 合并两个子节点
    static SegmentNode merge(SegmentNode left, SegmentNode right) {
        if (left == null) return right;
        if (right == null) return left;

        SegmentNode node = new SegmentNode();

        // 区间和 = 左区间和 + 右区间和
        node.sum = left.sum + right.sum;

        // 最大前缀和 = max(左前缀和, 左区间和 + 右前缀和)
        node.prefix = Math.max(left.prefix, left.sum + right.prefix);

        // 最大后缀和 = max(右后缀和, 右区间和 + 左后缀和)
        node.suffix = Math.max(right.suffix, right.sum + left.suffix);

        // 最大子段和 = max(左最大子段和, 右最大子段和, 左后缀和 + 右前缀和)
        node.maxSum = Math.max(Math.max(left.maxSum, right.maxSum),
                               left.suffix + right.prefix);

        return node;
    }
}

/**
 * 最大子段和线段树
 *
```

```
 * 解题思路:
 * 1. 线段树每个节点维护区间最大子段和相关信息
 * 2. 支持单点更新和区间查询
 * 3. 合并操作需要特殊处理四个值的合并
 *
 * 时间复杂度分析:
 * - 构建: O(n)
 * - 单点更新: O(log n)
 * - 区间查询: O(log n)
 *
 * 空间复杂度分析:
 * - 线段树数组: O(4n)
 * - 总空间复杂度: O(n)
 *
 * 工程化考量:
 * 1. 边界条件处理
 * 2. 负数和零的处理
 * 3. 空区间处理
 * 4. 性能优化
 */
static class MaxSubarraySegmentTree {
    private SegmentNode[] tree;
    private int n;

    public MaxSubarraySegmentTree(int[] nums) {
        if (nums == null || nums.length == 0) {
            throw new IllegalArgumentException("Array cannot be null or empty");
        }

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

    private void build(int node, int left, int right, int[] nums) {
        if (left == right) {
            tree[node] = new SegmentNode(nums[left]);
            return;
        }

        int mid = left + (right - left) / 2;
        build(node * 2, left, mid, nums);
        build(node * 2 + 1, mid + 1, right, nums);
```

```
        tree[node] = SegmentNode.merge(tree[node * 2], tree[node * 2 + 1]);
    }

    // 单点更新
    public void update(int index, int value) {
        if (index < 0 || index >= n) {
            throw new IllegalArgumentException("Invalid index");
        }
        update(1, 0, n - 1, index, value);
    }

    private void update(int node, int left, int right, int index, int value) {
        if (left == right) {
            tree[node] = new SegmentNode(value);
            return;
        }

        int mid = left + (right - left) / 2;
        if (index <= mid) {
            update(node * 2, left, mid, index, value);
        } else {
            update(node * 2 + 1, mid + 1, right, index, value);
        }

        tree[node] = SegmentNode.merge(tree[node * 2], tree[node * 2 + 1]);
    }

    // 查询区间最大子段和
    public int queryMaxSubarray(int queryLeft, int queryRight) {
        if (queryLeft < 0 || queryRight >= n || queryLeft > queryRight) {
            throw new IllegalArgumentException("Invalid query range");
        }

        SegmentNode result = query(1, 0, n - 1, queryLeft, queryRight);
        return result.maxSum;
    }

    private SegmentNode query(int node, int left, int right, int queryLeft, int queryRight) {
        if (queryLeft <= left && right <= queryRight) {
            return tree[node];
        }
```

```
            int mid = left + (right - left) / 2;

            if (queryRight <= mid) {
                return query(node * 2, left, mid, queryLeft, queryRight);
            } else if (queryLeft > mid) {
                return query(node * 2 + 1, mid + 1, right, queryLeft, queryRight);
            } else {
                SegmentNode leftResult = query(node * 2, left, mid, queryLeft, queryRight);
                SegmentNode rightResult = query(node * 2 + 1, mid + 1, right, queryLeft,
queryRight);
                return SegmentNode.merge(leftResult, rightResult);
            }
        }
    }

    /**
     * 动态规划解法（对比解法）
     * Kadane 算法：O(n)时间求最大子段和
     *
     * 解题思路:
     * 1. 遍历数组，维护当前子段和
     * 2. 如果当前子段和小于 0，则重新开始
     * 3. 记录最大子段和
     *
     * 时间复杂度: O(n)
     * 空间复杂度: O(1)
     */
    public static int kadaneAlgorithm(int[] nums) {
        if (nums == null || nums.length == 0) {
            return 0;
        }

        int maxSum = nums[0];
        int currentSum = nums[0];

        for (int i = 1; i < nums.length; i++) {
            // 如果当前和小于 0，重新开始
            currentSum = Math.max(nums[i], currentSum + nums[i]);
            maxSum = Math.max(maxSum, currentSum);
        }

        return maxSum;
    }
```

```
/**
 * 分治解法（对比解法）
 * 将数组分成两半，最大子段和可能出现在：
 * 1. 左半部分
 * 2. 右半部分
 * 3. 跨越中间的部分
 *
 * 时间复杂度: O(n log n)
 * 空间复杂度: O(log n)
 */
public static int divideAndConquer(int[] nums) {
    if (nums == null || nums.length == 0) {
        return 0;
    }
    return divideAndConquer(nums, 0, nums.length - 1);
}

private static int divideAndConquer(int[] nums, int left, int right) {
    if (left == right) {
        return nums[left];
    }

    int mid = left + (right - left) / 2;

    // 左半部分最大子段和
    int leftMax = divideAndConquer(nums, left, mid);

    // 右半部分最大子段和
    int rightMax = divideAndConquer(nums, mid + 1, right);

    // 跨越中间的最大子段和
    int crossMax = maxCrossingSum(nums, left, mid, right);

    return Math.max(Math.max(leftMax, rightMax), crossMax);
}

private static int maxCrossingSum(int[] nums, int left, int mid, int right) {
    // 从中间向左的最大后缀和
    int leftSum = Integer.MIN_VALUE;
    int sum = 0;
    for (int i = mid; i >= left; i--) {
        sum += nums[i];
```

```
            leftSum = Math.max(leftSum, sum);
        }

        // 从中间向右的最大前缀和
        int rightSum = Integer.MIN_VALUE;
        sum = 0;
        for (int i = mid + 1; i <= right; i++) {
            sum += nums[i];
            rightSum = Math.max(rightSum, sum);
        }

        return leftSum + rightSum;
    }

    // 测试代码
    public static void main(String[] args) {
        // 测试用例 1: 基础测试
        int[] nums1 = {-2, 1, -3, 4, -1, 2, 1, -5, 4};
        System.out.println("数组: " + Arrays.toString(nums1));

        MaxSubarraySegmentTree tree = new MaxSubarraySegmentTree(nums1);
        System.out.println("线段树解法最大子段和: " + tree.queryMaxSubarray(0, nums1.length -
1));
        System.out.println("Kadane 算法最大子段和: " + kadaneAlgorithm(nums1));
        System.out.println("分治解法最大子段和: " + divideAndConquer(nums1));
        System.out.println();

        // 测试用例 2: 全负数
        int[] nums2 = {-1, -2, -3, -4};
        System.out.println("数组: " + Arrays.toString(nums2));

        tree = new MaxSubarraySegmentTree(nums2);
        System.out.println("线段树解法最大子段和: " + tree.queryMaxSubarray(0, nums2.length -
1));
        System.out.println("Kadane 算法最大子段和: " + kadaneAlgorithm(nums2));
        System.out.println("分治解法最大子段和: " + divideAndConquer(nums2));
        System.out.println();

        // 测试用例 3: 全正数
        int[] nums3 = {1, 2, 3, 4};
        System.out.println("数组: " + Arrays.toString(nums3));

        tree = new MaxSubarraySegmentTree(nums3);
```

```
        System.out.println("线段树解法最大子段和: " + tree.queryMaxSubarray(0, nums3.length - 1));
        System.out.println("Kadane 算法最大子段和: " + kadaneAlgorithm(nums3));
        System.out.println("分治解法最大子段和: " + divideAndConquer(nums3));
        System.out.println();

        // 测试用例 4: 包含零
        int[] nums4 = {0, -1, 2, -3, 4, 0, -2};
        System.out.println("数组: " + Arrays.toString(nums4));

        tree = new MaxSubarraySegmentTree(nums4);
        System.out.println("线段树解法最大子段和: " + tree.queryMaxSubarray(0, nums4.length - 1));
        System.out.println("Kadane 算法最大子段和: " + kadaneAlgorithm(nums4));
        System.out.println("分治解法最大子段和: " + divideAndConquer(nums4));
        System.out.println();

        // 测试更新操作
        int[] nums5 = {1, -2, 3, -4, 5};
        tree = new MaxSubarraySegmentTree(nums5);
        System.out.println("原始数组: " + Arrays.toString(nums5));
        System.out.println("原始最大子段和: " + tree.queryMaxSubarray(0, nums5.length - 1));

        // 更新中间元素
        tree.update(2, 10);
        System.out.println("更新索引 2 为 10 后最大子段和: " + tree.queryMaxSubarray(0, nums5.length - 1));

        // 测试区间查询
        System.out.println("区间[1,3]最大子段和: " + tree.queryMaxSubarray(1, 3));
        System.out.println("区间[0,2]最大子段和: " + tree.queryMaxSubarray(0, 2));

        System.out.println("所有测试通过!");
    }
}
```

==========

文件: Code20_RangeModule.java

==========

```
import java.util.*;

// LeetCode 715. Range 模块
```

```
// 设计一个数据结构，支持以下操作：
// 1. 添加区间 [left, right)
// 2. 删除区间 [left, right)
// 3. 查询区间 [left, right) 是否完全被覆盖
// 测试链接: https://leetcode.cn/problems/range-module/

public class Code20_RangeModule {

    /**
     * 动态开点线段树实现 Range 模块
     *
     * 解题思路:
     * 1. 使用动态开点线段树维护区间覆盖状态
     * 2. 每个节点维护三个信息：
     *    - 区间是否被完全覆盖 (covered)
     *    - 懒惰标记 (lazy)
     *    - 左右子节点引用
     * 3. 支持区间添加、删除和查询操作
     *
     * 时间复杂度分析:
     * - 添加区间: O(log R) 其中 R 是值域范围
     * - 删除区间: O(log R)
     * - 查询区间: O(log R)
     *
     * 空间复杂度分析:
     * - 动态开点线段树: O(n log R)，其中 n 是操作次数
     * - 总空间复杂度: O(n log R)
     *
     * 工程化考量:
     * 1. 动态开点节省内存
     * 2. 懒惰传播提高效率
     * 3. 边界条件处理
     * 4. 大值域范围处理
     */

    static class SegmentNode {
        boolean covered;  // 当前区间是否被完全覆盖
        int lazy;         // 懒惰标记: 0-无操作, 1-添加, 2-删除
        SegmentNode left, right;

        SegmentNode() {
            this.covered = false;
            this.lazy = 0;
```

```
        this.left = null;
        this.right = null;
    }
}

static class RangeModule {
    private SegmentNode root;
    private static final int MIN = 1;
    private static final int MAX = 1000000000; // 10^9

    public RangeModule() {
        root = new SegmentNode();
    }

    /**
     * 添加区间 [left, right)
     */
    public void addRange(int left, int right) {
        update(root, MIN, MAX, left, right - 1, 1);
    }

    /**
     * 查询区间 [left, right) 是否完全被覆盖
     */
    public boolean queryRange(int left, int right) {
        return query(root, MIN, MAX, left, right - 1);
    }

    /**
     * 删除区间 [left, right)
     */
    public void removeRange(int left, int right) {
        update(root, MIN, MAX, left, right - 1, 2);
    }

    /**
     * 更新线段树
     *
     * @param node 当前节点
     * @param l 当前节点区间左边界
     * @param r 当前节点区间右边界
     * @param ql 查询区间左边界
     * @param qr 查询区间右边界
```

```
     * @param op 操作类型: 1-添加, 2-删除
     */
    private void update(SegmentNode node, int l, int r, int ql, int qr, int op) {
        if (ql <= l && r <= qr) {
            // 当前区间完全在查询区间内
            if (op == 1) {
                // 添加操作: 标记为完全覆盖
                node.covered = true;
                node.lazy = 1;
            } else {
                // 删除操作: 标记为未覆盖
                node.covered = false;
                node.lazy = 2;
            }
            return;
        }

        // 懒惰传播
        pushDown(node, l, r);

        int mid = l + (r - l) / 2;

        if (ql <= mid) {
            update(getLeft(node), l, mid, ql, qr, op);
        }
        if (qr > mid) {
            update(getRight(node), mid + 1, r, ql, qr, op);
        }

        // 合并子节点状态
        pushUp(node);
    }

    /**
     * 查询区间是否完全被覆盖
     */
    private boolean query(SegmentNode node, int l, int r, int ql, int qr) {
        if (ql <= l && r <= qr) {
            // 当前区间完全在查询区间内，直接返回覆盖状态
            return node.covered;
        }

        // 懒惰传播
```

```
    pushDown(node, l, r);

    int mid = l + (r - l) / 2;
    boolean result = true;

    if (ql <= mid) {
        result = result && query(getLeft(node), l, mid, ql, qr);
    }
    if (qr > mid) {
        result = result && query(getRight(node), mid + 1, r, ql, qr);
    }

    return result;
}

/**
 * 懒惰传播
 */
private void pushDown(SegmentNode node, int l, int r) {
    if (node.lazy != 0) {
        int mid = l + (r - l) / 2;

        SegmentNode leftNode = getLeft(node);
        SegmentNode rightNode = getRight(node);

        if (node.lazy == 1) {
            // 添加操作传播
            leftNode.covered = true;
            rightNode.covered = true;
            leftNode.lazy = 1;
            rightNode.lazy = 1;
        } else {
            // 删除操作传播
            leftNode.covered = false;
            rightNode.covered = false;
            leftNode.lazy = 2;
            rightNode.lazy = 2;
        }

        node.lazy = 0; // 清除懒惰标记
    }
}
```

```
    /**
     * 合并子节点状态
     */
    private void pushUp(SegmentNode node) {
        node.covered = getLeft(node).covered && getRight(node).covered;
    }

    /**
     * 获取左子节点（动态开点）
     */
    private SegmentNode getLeft(SegmentNode node) {
        if (node.left == null) {
            node.left = new SegmentNode();
        }
        return node.left;
    }

    /**
     * 获取右子节点（动态开点）
     */
    private SegmentNode getRight(SegmentNode node) {
        if (node.right == null) {
            node.right = new SegmentNode();
        }
        return node.right;
    }
}

/**
 * 有序集合实现（对比解法）
 * 使用 TreeMap 维护不相交区间
 *
 * 解题思路:
 * 1. 使用 TreeMap 存储区间，key 为区间左端点，value 为区间右端点
 * 2. 添加区间时合并重叠区间
 * 3. 删除区间时分割现有区间
 * 4. 查询时检查区间是否完全覆盖
 *
 * 时间复杂度分析:
 * - 添加区间: O(n) 最坏情况需要合并多个区间
 * - 删除区间: O(n) 最坏情况需要分割多个区间
 * - 查询区间: O(log n)
 *
```

```
 * 空间复杂度分析:
 * - TreeMap: O(n)
 * - 总空间复杂度: O(n)
 */
static class RangeModuleTreeMap {
    private TreeMap<Integer, Integer> intervals;

    public RangeModuleTreeMap() {
        intervals = new TreeMap<>();
    }

    public void addRange(int left, int right) {
        // 找到第一个左端点小于等于 right 的区间
        Map.Entry<Integer, Integer> entry = intervals.floorEntry(right);

        while (entry != null && entry.getValue() >= left) {
            // 合并重叠区间
            left = Math.min(left, entry.getKey());
            right = Math.max(right, entry.getValue());
            intervals.remove(entry.getKey());
            entry = intervals.floorEntry(right);
        }

        intervals.put(left, right);
    }

    public boolean queryRange(int left, int right) {
        // 找到第一个左端点小于等于 left 的区间
        Map.Entry<Integer, Integer> entry = intervals.floorEntry(left);
        return entry != null && entry.getValue() >= right;
    }

    public void removeRange(int left, int right) {
        // 找到第一个左端点小于 right 的区间
        Map.Entry<Integer, Integer> entry = intervals.lowerEntry(right);

        while (entry != null && entry.getValue() > left) {
            // 分割区间
            intervals.remove(entry.getKey());

            if (entry.getKey() < left) {
                intervals.put(entry.getKey(), left);
            }
```

```
            if (entry.getValue() > right) {
                intervals.put(right, entry.getValue());
            }

            entry = intervals.lowerEntry(right);
        }
    }
}

// 测试代码
public static void main(String[] args) {
    System.out.println("=== 线段树实现测试 ===");
    testSegmentTreeImplementation();

    System.out.println("\n=== TreeMap 实现测试 ===");
    testTreeMapImplementation();

    System.out.println("\n=== 性能对比测试 ===");
    performanceComparison();

    System.out.println("\n 所有测试通过!");
}

private static void testSegmentTreeImplementation() {
    RangeModule rangeModule = new RangeModule();

    // 测试添加区间
    rangeModule.addRange(10, 20);
    System.out.println("添加区间 [10, 20)");

    // 测试查询
    System.out.println("查询 [14, 16): " + rangeModule.queryRange(14, 16)); // true
    System.out.println("查询 [16, 17): " + rangeModule.queryRange(16, 17)); // true
    System.out.println("查询 [20, 21): " + rangeModule.queryRange(20, 21)); // false

    // 测试添加重叠区间
    rangeModule.addRange(15, 25);
    System.out.println("添加重叠区间 [15, 25)");
    System.out.println("查询 [20, 21): " + rangeModule.queryRange(20, 21)); // true

    // 测试删除区间
    rangeModule.removeRange(14, 16);
```

```
        System.out.println("删除区间 [14, 16)");
        System.out.println("查询 [14, 15): " + rangeModule.queryRange(14, 15)); // false
        System.out.println("查询 [16, 17): " + rangeModule.queryRange(16, 17)); // true

        // 测试复杂操作
        rangeModule.addRange(5, 8);
        rangeModule.addRange(1, 3);
        System.out.println("查询 [1, 8): " + rangeModule.queryRange(1, 8)); // false
        System.out.println("查询 [1, 3): " + rangeModule.queryRange(1, 3)); // true
        System.out.println("查询 [5, 8): " + rangeModule.queryRange(5, 8)); // true
    }

    private static void testTreeMapImplementation() {
        RangeModuleTreeMap rangeModule = new RangeModuleTreeMap();

        // 测试添加区间
        rangeModule.addRange(10, 20);
        System.out.println("添加区间 [10, 20)");

        // 测试查询
        System.out.println("查询 [14, 16): " + rangeModule.queryRange(14, 16)); // true
        System.out.println("查询 [16, 17): " + rangeModule.queryRange(16, 17)); // true
        System.out.println("查询 [20, 21): " + rangeModule.queryRange(20, 21)); // false

        // 测试添加重叠区间
        rangeModule.addRange(15, 25);
        System.out.println("添加重叠区间 [15, 25)");
        System.out.println("查询 [20, 21): " + rangeModule.queryRange(20, 21)); // true

        // 测试删除区间
        rangeModule.removeRange(14, 16);
        System.out.println("删除区间 [14, 16)");
        System.out.println("查询 [14, 15): " + rangeModule.queryRange(14, 15)); // false
        System.out.println("查询 [16, 17): " + rangeModule.queryRange(16, 17)); // true
    }

    private static void performanceComparison() {
        int n = 10000; // 操作次数

        // 线段树实现性能测试
        long startTime = System.currentTimeMillis();
        RangeModule segmentTree = new RangeModule();
        for (int i = 0; i < n; i++) {
```

```
            int left = i * 10;
            int right = left + 5;
            segmentTree.addRange(left, right);
            segmentTree.queryRange(left, right + 1);
            segmentTree.removeRange(left + 2, right - 2);
        }
        long segmentTreeTime = System.currentTimeMillis() - startTime;

        // TreeMap 实现性能测试
        startTime = System.currentTimeMillis();
        RangeModuleTreeMap treeMap = new RangeModuleTreeMap();
        for (int i = 0; i < n; i++) {
            int left = i * 10;
            int right = left + 5;
            treeMap.addRange(left, right);
            treeMap.queryRange(left, right + 1);
            treeMap.removeRange(left + 2, right - 2);
        }
        long treeMapTime = System.currentTimeMillis() - startTime;

        System.out.println("线段树实现耗时: " + segmentTreeTime + "ms");
        System.out.println("TreeMap 实现耗时: " + treeMapTime + "ms");
        System.out.println("性能差异: " + (segmentTreeTime - treeMapTime) + "ms");
    }
}
```

==========

文件: Code21_RangeSumQuery2DMutable.cpp

==========

```
/**
 * LeetCode 308. 二维区域和检索 - 可变 (Range Sum Query 2D - Mutable)
 *
 * 题目描述：
 * 设计一个数据结构，支持以下操作：
 * 1. 更新矩阵中某个元素的值
 * 2. 查询子矩阵的元素和
 *
 * 解题思路：
 * 使用二维树状数组（Fenwick Tree）来高效支持更新和查询操作。
 * 二维树状数组通过维护二维前缀和数组，可以在 O(log m * log n)时间内完成更新和查询。
 *
 * 时间复杂度分析：
```

```
 * - 构造函数：O(m * n)，需要初始化二维树状数组
 * - update 操作：O(log m * log n)，需要更新相关的前缀和
 * - sumRegion 操作：O(log m * log n)，通过二维前缀和计算子矩阵和
 *
 * 空间复杂度分析：
 * - O(m * n)，用于存储二维树状数组
 *
 * 工程化考量：
 * 1. 边界条件处理：检查行列索引是否越界
 * 2. 异常处理：处理非法输入参数
 * 3. 性能优化：使用位运算加速索引计算
 * 4. 可读性：变量命名清晰，注释详细
 * 5. 内存管理：使用 vector 容器自动管理内存
 *
 * 算法技巧：
 * - 二维树状数组的核心思想是将二维前缀和分解为多个一维前缀和的组合
 * - 使用 lowbit 操作快速定位需要更新的位置
 * - 通过容斥原理计算子矩阵和
 *
 * 适用场景：
 * - 需要频繁更新和查询二维矩阵的子矩阵和
 * - 数据规模较大，需要高效的数据结构支持
 * - 对实时性要求较高的应用场景
 *
 * 测试用例：
 * 输入：
 * matrix = [
 *   [3, 0, 1, 4, 2],
 *   [5, 6, 3, 2, 1],
 *   [1, 2, 0, 1, 5],
 *   [4, 1, 0, 1, 7],
 *   [1, 0, 3, 0, 5]
 * ]
 * 操作：
 * sumRegion(2, 1, 4, 3) -> 8
 * update(3, 2, 2)
 * sumRegion(2, 1, 4, 3) -> 10
 */

#include <iostream>
#include <vector>
#include <stdexcept>
```

```
using namespace std;

class NumMatrix {
private:
    vector<vector<int>> tree;  // 二维树状数组
    vector<vector<int>> matrix; // 原始矩阵
    int m, n;                   // 矩阵的行数和列数

    /**
     * 计算数字的 lowbit（最低位的 1）
     *
     * @param x 输入数字
     * @return lowbit 值
     *
     * 算法原理：
     * - x & -x 可以快速得到 x 的最低位的 1
     * - 这是树状数组的核心操作，用于快速定位需要更新的位置
     *
     * 时间复杂度：O(1)
     * 空间复杂度：O(1)
     */
    int lowbit(int x) {
        return x & -x;
    }

    /**
     * 查询从(0,0)到(row,col)的子矩阵和
     *
     * @param row 行索引
     * @param col 列索引
     * @return 前缀和
     *
     * 算法步骤：
     * 1. 处理边界情况
     * 2. 使用树状数组查询前缀和
     * 3. 累加相关位置的值
     *
     * 时间复杂度：O(log m * log n)
     * 空间复杂度：O(1)
     */
    int query(int row, int col) {
        if (row < 0 || col < 0) return 0;
```

```
        int sum = 0;
        for (int i = row + 1; i > 0; i -= lowbit(i)) {
            for (int j = col + 1; j > 0; j -= lowbit(j)) {
                sum += tree[i][j];
            }
        }
        return sum;
    }

public:
    /**
     * 构造函数：初始化二维树状数组
     *
     * @param matrix 输入的二维矩阵
     *
     * 算法步骤：
     * 1. 检查输入矩阵是否为空
     * 2. 初始化树状数组和原始矩阵
     * 3. 构建二维树状数组
     *
     * 时间复杂度：O(m * n * log m * log n)
     * 空间复杂度：O(m * n)
     */
    NumMatrix(vector<vector<int>>& matrix) {
        if (matrix.empty() || matrix[0].empty()) {
            throw invalid_argument("Matrix cannot be empty");
        }

        this->m = matrix.size();
        this->n = matrix[0].size();
        this->matrix = matrix;
        this->tree = vector<vector<int>>(m + 1, vector<int>(n + 1, 0));

        // 构建二维树状数组
        for (int i = 0; i < m; i++) {
            for (int j = 0; j < n; j++) {
                update(i, j, matrix[i][j]);
            }
        }
    }

    /**
     * 更新矩阵中指定位置的元素值
```

```
 *
 * @param row 行索引
 * @param col 列索引
 * @param val 新的值
 *
 * 算法步骤：
 * 1. 检查索引是否越界
 * 2. 计算值的差异
 * 3. 更新原始矩阵
 * 4. 更新二维树状数组
 *
 * 时间复杂度：O(log m * log n)
 * 空间复杂度：O(1)
 */
void update(int row, int col, int val) {
    if (row < 0 || row >= m || col < 0 || col >= n) {
        throw invalid_argument("Invalid row or column index");
    }

    int diff = val - matrix[row][col];
    matrix[row][col] = val;

    // 更新二维树状数组
    for (int i = row + 1; i <= m; i += lowbit(i)) {
        for (int j = col + 1; j <= n; j += lowbit(j)) {
            tree[i][j] += diff;
        }
    }
}

/**
 * 查询子矩阵的元素和
 *
 * @param row1 子矩阵左上角行索引
 * @param col1 子矩阵左上角列索引
 * @param row2 子矩阵右下角行索引
 * @param col2 子矩阵右下角列索引
 * @return 子矩阵的元素和
 *
 * 算法步骤：
 * 1. 检查索引是否越界
 * 2. 使用二维前缀和计算子矩阵和
 * 3. 应用容斥原理：sum = sum(row2,col2) - sum(row2,col1-1) - sum(row1-1,col2) + sum(row1-
```

```
1,col1-1)
     *
     * 时间复杂度：O(log m * log n)
     * 空间复杂度：O(1)
     */
    int sumRegion(int row1, int col1, int row2, int col2) {
        if (row1 < 0 || row2 >= m || col1 < 0 || col2 >= n || row1 > row2 || col1 > col2) {
            throw invalid_argument("Invalid region coordinates");
        }

        return query(row2, col2) - query(row2, col1 - 1) - query(row1 - 1, col2) + query(row1 -
1, col1 - 1);
    }
};

/**
 * 测试函数：验证二维树状数组的正确性
 *
 * 测试用例设计：
 * 1. 正常情况测试
 * 2. 边界情况测试
 * 3. 更新操作测试
 * 4. 查询操作测试
 */
int main() {
    vector<vector<int>> matrix = {
        {3, 0, 1, 4, 2},
        {5, 6, 3, 2, 1},
        {1, 2, 0, 1, 5},
        {4, 1, 0, 1, 7},
        {1, 0, 3, 0, 5}
    };

    try {
        NumMatrix numMatrix(matrix);

        // 测试查询操作
        cout << "初始查询结果：" << numMatrix.sumRegion(2, 1, 4, 3) << endl; // 期望：8

        // 测试更新操作
        numMatrix.update(3, 2, 2);
        cout << "更新后查询结果：" << numMatrix.sumRegion(2, 1, 4, 3) << endl; // 期望：10
```

```
        // 测试边界情况
        cout << "单点查询：" << numMatrix.sumRegion(0, 0, 0, 0) << endl; // 期望：3
        cout << "整行查询：" << numMatrix.sumRegion(0, 0, 0, 4) << endl; // 期望：10

        cout << "测试通过！" << endl;
    } catch (const exception& e) {
        cerr << "错误：" << e.what() << endl;
        return 1;
    }

    return 0;
}

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

文件: Code21_RangeSumQuery2DMutable.java
==========================================
package class132;

/**
 * LeetCode 308. 二维区域和检索 - 可变 (Range Sum Query 2D - Mutable)
 *
 * 题目描述：
 * 设计一个数据结构，支持以下操作：
 * 1. 更新矩阵中某个元素的值
 * 2. 查询子矩阵的元素和
 *
 * 解题思路：
 * 使用二维树状数组（Fenwick Tree）来高效支持更新和查询操作。
 * 二维树状数组通过维护二维前缀和数组，可以在O(log m * log n)时间内完成更新和查询。
 *
 * 时间复杂度分析：
 * - 构造函数：O(m * n)，需要初始化二维树状数组
 * - update操作：O(log m * log n)，需要更新相关的前缀和
 * - sumRegion操作：O(log m * log n)，通过二维前缀和计算子矩阵和
 *
 * 空间复杂度分析：
 * - O(m * n)，用于存储二维树状数组
 *
 * 工程化考量：
 * 1. 边界条件处理：检查行列索引是否越界
 * 2. 异常处理：处理非法输入参数
 * 3. 性能优化：使用位运算加速索引计算
```

```
 * 4. 可读性：变量命名清晰，注释详细
 *
 * 算法技巧：
 * - 二维树状数组的核心思想是将二维前缀和分解为多个一维前缀和的组合
 * - 使用 lowbit 操作快速定位需要更新的位置
 * - 通过容斥原理计算子矩阵和
 *
 * 适用场景：
 * - 需要频繁更新和查询二维矩阵的子矩阵和
 * - 数据规模较大，需要高效的数据结构支持
 * - 对实时性要求较高的应用场景
 *
 * 测试用例：
 * 输入：
 * matrix = [
 *   [3, 0, 1, 4, 2],
 *   [5, 6, 3, 2, 1],
 *   [1, 2, 0, 1, 5],
 *   [4, 1, 0, 1, 7],
 *   [1, 0, 3, 0, 5]
 * ]
 * 操作：
 * sumRegion(2, 1, 4, 3) -> 8
 * update(3, 2, 2)
 * sumRegion(2, 1, 4, 3) -> 10
 */
public class Code21_RangeSumQuery2DMutable {

    private int[][] tree;  // 二维树状数组
    private int[][] matrix; // 原始矩阵
    private int m, n;       // 矩阵的行数和列数

    /**
     * 构造函数：初始化二维树状数组
     *
     * @param matrix 输入的二维矩阵
     *
     * 算法步骤：
     * 1. 检查输入矩阵是否为空
     * 2. 初始化树状数组和原始矩阵
     * 3. 构建二维树状数组
     *
     * 时间复杂度：O(m * n * log m * log n)
```

```
     * 空间复杂度：O(m * n)
     */
    public Code21_RangeSumQuery2DMutable(int[][] matrix) {
        if (matrix == null || matrix.length == 0 || matrix[0].length == 0) {
            throw new IllegalArgumentException("Matrix cannot be null or empty");
        }

        this.m = matrix.length;
        this.n = matrix[0].length;
        this.matrix = new int[m][n];
        this.tree = new int[m + 1][n + 1];

        // 构建二维树状数组
        for (int i = 0; i < m; i++) {
            for (int j = 0; j < n; j++) {
                update(i, j, matrix[i][j]);
                this.matrix[i][j] = matrix[i][j];
            }
        }
    }

    /**
     * 更新矩阵中指定位置的元素值
     *
     * @param row 行索引
     * @param col 列索引
     * @param val 新的值
     *
     * 算法步骤：
     * 1. 检查索引是否越界
     * 2. 计算值的差异
     * 3. 更新原始矩阵
     * 4. 更新二维树状数组
     *
     * 时间复杂度：O(log m * log n)
     * 空间复杂度：O(1)
     */
    public void update(int row, int col, int val) {
        if (row < 0 || row >= m || col < 0 || col >= n) {
            throw new IllegalArgumentException("Invalid row or column index");
        }

        int diff = val - matrix[row][col];
```

```
        matrix[row][col] = val;

        // 更新二维树状数组
        for (int i = row + 1; i <= m; i += lowbit(i)) {
            for (int j = col + 1; j <= n; j += lowbit(j)) {
                tree[i][j] += diff;
            }
        }
    }

    /**
     * 查询子矩阵的元素和
     *
     * @param row1 子矩阵左上角行索引
     * @param col1 子矩阵左上角列索引
     * @param row2 子矩阵右下角行索引
     * @param col2 子矩阵右下角列索引
     * @return 子矩阵的元素和
     *
     * 算法步骤：
     * 1. 检查索引是否越界
     * 2. 使用二维前缀和计算子矩阵和
     * 3. 应用容斥原理：sum = sum(row2,col2) - sum(row2,col1-1) - sum(row1-1,col2) + sum(row1-1,col1-1)
     *
     * 时间复杂度：O(log m * log n)
     * 空间复杂度：O(1)
     */
    public int sumRegion(int row1, int col1, int row2, int col2) {
        if (row1 < 0 || row2 >= m || col1 < 0 || col2 >= n || row1 > row2 || col1 > col2) {
            throw new IllegalArgumentException("Invalid region coordinates");
        }

        return query(row2, col2) - query(row2, col1 - 1) - query(row1 - 1, col2) + query(row1 - 1, col1 - 1);
    }

    /**
     * 查询从(0,0)到(row,col)的子矩阵和
     *
     * @param row 行索引
     * @param col 列索引
     * @return 前缀和
```

```
     *
     * 算法步骤：
     * 1. 处理边界情况
     * 2. 使用树状数组查询前缀和
     * 3. 累加相关位置的值
     *
     * 时间复杂度：O(log m * log n)
     * 空间复杂度：O(1)
     */
    private int query(int row, int col) {
        if (row < 0 || col < 0) return 0;

        int sum = 0;
        for (int i = row + 1; i > 0; i -= lowbit(i)) {
            for (int j = col + 1; j > 0; j -= lowbit(j)) {
                sum += tree[i][j];
            }
        }
        return sum;
    }

    /**
     * 计算数字的 lowbit（最低位的 1）
     *
     * @param x 输入数字
     * @return lowbit 值
     *
     * 算法原理：
     * - x & -x 可以快速得到 x 的最低位的 1
     * - 这是树状数组的核心操作，用于快速定位需要更新的位置
     *
     * 时间复杂度：O(1)
     * 空间复杂度：O(1)
     */
    private int lowbit(int x) {
        return x & -x;
    }

    /**
     * 测试方法：验证二维树状数组的正确性
     *
     * 测试用例设计：
     * 1. 正常情况测试
```

```
     * 2. 边界情况测试
     * 3. 更新操作测试
     * 4. 查询操作测试
     */
    public static void main(String[] args) {
        int[][] matrix = {
            {3, 0, 1, 4, 2},
            {5, 6, 3, 2, 1},
            {1, 2, 0, 1, 5},
            {4, 1, 0, 1, 7},
            {1, 0, 3, 0, 5}
        };

        Code21_RangeSumQuery2DMutable numMatrix = new Code21_RangeSumQuery2DMutable(matrix);

        // 测试查询操作
        System.out.println("初始查询结果：" + numMatrix.sumRegion(2, 1, 4, 3)); // 期望：8

        // 测试更新操作
        numMatrix.update(3, 2, 2);
        System.out.println("更新后查询结果：" + numMatrix.sumRegion(2, 1, 4, 3)); // 期望：10

        // 测试边界情况
        System.out.println("单点查询：" + numMatrix.sumRegion(0, 0, 0, 0)); // 期望：3
        System.out.println("整行查询：" + numMatrix.sumRegion(0, 0, 0, 4)); // 期望：10

        System.out.println("测试通过！");
    }
}
```

==========

文件: Code21_RangeSumQuery2DMutable.py

==========

```
"""
LeetCode 308. 二维区域和检索 - 可变 (Range Sum Query 2D - Mutable)

题目描述：
设计一个数据结构，支持以下操作：
1. 更新矩阵中某个元素的值
2. 查询子矩阵的元素和

解题思路：
```

使用二维树状数组（Fenwick Tree）来高效支持更新和查询操作。
二维树状数组通过维护二维前缀和数组，可以在O(log m * log n)时间内完成更新和查询。

时间复杂度分析：
- 构造函数：O(m * n)，需要初始化二维树状数组
- update操作：O(log m * log n)，需要更新相关的前缀和
- sumRegion操作：O(log m * log n)，通过二维前缀和计算子矩阵和

空间复杂度分析：
- O(m * n)，用于存储二维树状数组

工程化考量：
1. 边界条件处理：检查行列索引是否越界
2. 异常处理：处理非法输入参数
3. 性能优化：使用位运算加速索引计算
4. 可读性：变量命名清晰，注释详细
5. Python特性：使用列表推导式初始化二维数组

算法技巧：
- 二维树状数组的核心思想是将二维前缀和分解为多个一维前缀和的组合
- 使用lowbit操作快速定位需要更新的位置
- 通过容斥原理计算子矩阵和

适用场景：
- 需要频繁更新和查询二维矩阵的子矩阵和
- 数据规模较大，需要高效的数据结构支持
- 对实时性要求较高的应用场景

测试用例：
输入：
```
matrix = [
  [3, 0, 1, 4, 2],
  [5, 6, 3, 2, 1],
  [1, 2, 0, 1, 5],
  [4, 1, 0, 1, 7],
  [1, 0, 3, 0, 5]
]
```
操作：
```
sumRegion(2, 1, 4, 3) -> 8
update(3, 2, 2)
sumRegion(2, 1, 4, 3) -> 10
"""
```

```
class NumMatrix:
    """
    二维区域和检索数据结构类

    使用二维树状数组实现高效的矩阵更新和查询操作
    """

    def __init__(self, matrix):
        """
        构造函数：初始化二维树状数组

        Args:
            matrix: 输入的二维矩阵

        算法步骤：
        1. 检查输入矩阵是否为空
        2. 初始化树状数组和原始矩阵
        3. 构建二维树状数组

        时间复杂度：O(m * n * log m * log n)
        空间复杂度：O(m * n)
        """
        if not matrix or not matrix[0]:
            raise ValueError("Matrix cannot be empty")

        self.m = len(matrix)
        self.n = len(matrix[0])
        self.matrix = [row[:] for row in matrix]  # 深拷贝原始矩阵

        # 初始化二维树状数组
        self.tree = [[0] * (self.n + 1) for _ in range(self.m + 1)]

        # 构建二维树状数组
        for i in range(self.m):
            for j in range(self.n):
                self._update_tree(i, j, matrix[i][j])

    def _lowbit(self, x):
        """
        计算数字的 lowbit（最低位的 1）

        Args:
            x: 输入数字
```

```
    Returns:
        lowbit 值

    算法原理：
    - x & -x 可以快速得到 x 的最低位的 1
    - 这是树状数组的核心操作，用于快速定位需要更新的位置

    时间复杂度：O(1)
    空间复杂度：O(1)
    """
    return x & -x

def _update_tree(self, row, col, val):
    """
    更新树状数组（内部方法）

    Args:
        row: 行索引
        col: 列索引
        val: 要增加的值

    算法步骤：
    1. 从指定位置开始，沿着树状数组的路径更新相关位置
    2. 使用 lowbit 操作快速定位需要更新的位置

    时间复杂度：O(log m * log n)
    空间复杂度：O(1)
    """
    i = row + 1
    while i <= self.m:
        j = col + 1
        while j <= self.n:
            self.tree[i][j] += val
            j += self._lowbit(j)
        i += self._lowbit(i)

def _query_tree(self, row, col):
    """
    查询树状数组前缀和（内部方法）

    Args:
        row: 行索引
```

```
        col: 列索引

    Returns:
        从(0,0)到(row,col)的子矩阵和

    算法步骤：
    1. 从指定位置开始，沿着树状数组的路径累加相关位置的值
    2. 使用 lowbit 操作快速定位需要累加的位置

    时间复杂度：O(log m * log n)
    空间复杂度：O(1)
    """
    if row < 0 or col < 0:
        return 0

    total = 0
    i = row + 1
    while i > 0:
        j = col + 1
        while j > 0:
            total += self.tree[i][j]
            j -= self._lowbit(j)
        i -= self._lowbit(i)
    return total

def update(self, row, col, val):
    """
    更新矩阵中指定位置的元素值

    Args:
        row: 行索引
        col: 列索引
        val: 新的值

    算法步骤：
    1. 检查索引是否越界
    2. 计算值的差异
    3. 更新原始矩阵
    4. 更新二维树状数组

    时间复杂度：O(log m * log n)
    空间复杂度：O(1)
    """
```

```
        if row < 0 or row >= self.m or col < 0 or col >= self.n:
            raise ValueError("Invalid row or column index")

        diff = val - self.matrix[row][col]
        self.matrix[row][col] = val

        # 更新二维树状数组
        self._update_tree(row, col, diff)

    def sumRegion(self, row1, col1, row2, col2):
        """
        查询子矩阵的元素和

        Args:
            row1: 子矩阵左上角行索引
            col1: 子矩阵左上角列索引
            row2: 子矩阵右下角行索引
            col2: 子矩阵右下角列索引

        Returns:
            子矩阵的元素和

        算法步骤：
        1. 检查索引是否越界
        2. 使用二维前缀和计算子矩阵和
        3. 应用容斥原理：sum = sum(row2,col2) - sum(row2,col1-1) - sum(row1-1,col2) + sum(row1-1,col1-1)

        时间复杂度：O(log m * log n)
        空间复杂度：O(1)
        """
        if (row1 < 0 or row2 >= self.m or col1 < 0 or col2 >= self.n or
            row1 > row2 or col1 > col2):
            raise ValueError("Invalid region coordinates")

        # 使用容斥原理计算子矩阵和
        return (self._query_tree(row2, col2) -
                self._query_tree(row2, col1 - 1) -
                self._query_tree(row1 - 1, col2) +
                self._query_tree(row1 - 1, col1 - 1))


def test_num_matrix():
```

```
    """
    测试函数：验证二维树状数组的正确性

    测试用例设计：
    1. 正常情况测试
    2. 边界情况测试
    3. 更新操作测试
    4. 查询操作测试
    """
    matrix = [
        [3, 0, 1, 4, 2],
        [5, 6, 3, 2, 1],
        [1, 2, 0, 1, 5],
        [4, 1, 0, 1, 7],
        [1, 0, 3, 0, 5]
    ]

    try:
        num_matrix = NumMatrix(matrix)

        # 测试查询操作
        result1 = num_matrix.sumRegion(2, 1, 4, 3)
        print(f"初始查询结果：{result1}")  # 期望：8

        # 测试更新操作
        num_matrix.update(3, 2, 2)
        result2 = num_matrix.sumRegion(2, 1, 4, 3)
        print(f"更新后查询结果：{result2}")  # 期望：10

        # 测试边界情况
        result3 = num_matrix.sumRegion(0, 0, 0, 0)
        print(f"单点查询：{result3}")  # 期望：3

        result4 = num_matrix.sumRegion(0, 0, 0, 4)
        print(f"整行查询：{result4}")  # 期望：10

        print("测试通过！")

    except ValueError as e:
        print(f"错误：{e}")


if __name__ == "__main__":
```

```
    test_num_matrix()

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

文件: Code22_CountOfRangeSum.cpp
================================================
/**
 * LeetCode 327. 区间和的个数 (Count of Range Sum)
 *
 * 题目描述：
 * 给定一个整数数组 nums，以及两个整数 lower 和 upper。
 * 返回区间和的值在区间 [lower, upper] 之间的区间个数（包含等于）。
 *
 * 解题思路：
 * 使用树状数组（Fenwick Tree）+ 离散化来高效统计满足条件的区间和个数。
 * 核心思想：
 * 1. 计算前缀和数组 prefixSum
 * 2. 对于每个前缀和 prefixSum[j]，需要统计有多少个 i < j 满足：
 *    lower <= prefixSum[j] - prefixSum[i] <= upper
 *    即：prefixSum[j] - upper <= prefixSum[i] <= prefixSum[j] - lower
 * 3. 使用树状数组维护前缀和的出现次数
 * 4. 通过离散化处理大数值范围问题
 *
 * 时间复杂度分析：
 * - 前缀和计算：O(n)
 * - 离散化处理：O(n log n)
 * - 树状数组操作：O(n log n)
 * - 总时间复杂度：O(n log n)
 *
 * 空间复杂度分析：
 * - 前缀和数组：O(n)
 * - 离散化数组：O(n)
 * - 树状数组：O(n)
 * - 总空间复杂度：O(n)
 *
 * 工程化考量：
 * 1. 边界条件处理：处理空数组、lower > upper 等情况
 * 2. 数值溢出处理：使用 long long 类型避免整数溢出
 * 3. 离散化优化：使用 set 去重并排序
 * 4. 异常处理：检查输入参数合法性
 *
 * 算法技巧：
 * - 离散化：将大范围的数值映射到小范围的索引
```

```
 * - 树状数组：高效统计前缀和的出现次数
 * - 容斥原理：通过区间查询统计满足条件的个数
 *
 * 适用场景：
 * - 需要统计满足特定条件的区间和个数
 * - 数值范围较大，需要离散化处理
 * - 对时间复杂度要求较高的场景
 *
 * 测试用例：
 * 输入：nums = [-2,5,-1], lower = -2, upper = 2
 * 输出：3
 * 解释：三个区间和满足条件：[0,0], [2,2], [0,2]
 */

#include <iostream>
#include <vector>
#include <set>
#include <map>
#include <algorithm>
#include <stdexcept>

using namespace std;

class Solution {
private:
    /**
     * 树状数组（Fenwick Tree）实现
     * 用于高效统计前缀和的出现次数
     */
    class FenwickTree {
    private:
        vector<int> tree;
        int size;

        /**
         * 计算 lowbit（最低位的 1）
         *
         * @param x 输入数字
         * @return lowbit 值
         */
        int lowbit(int x) {
            return x & -x;
        }
```

```
public:
    /**
     * 构造函数
     *
     * @param size 树状数组大小
     */
    FenwickTree(int size) : size(size) {
        tree.resize(size + 1, 0);
    }

    /**
     * 更新操作：在指定位置增加一个值
     *
     * @param index 位置索引
     * @param delta 增加值
     */
    void update(int index, int delta) {
        while (index <= size) {
            tree[index] += delta;
            index += lowbit(index);
        }
    }

    /**
     * 查询前缀和：从 1 到 index 的和
     *
     * @param index 位置索引
     * @return 前缀和
     */
    int query(int index) {
        int sum = 0;
        while (index > 0) {
            sum += tree[index];
            index -= lowbit(index);
        }
        return sum;
    }

    /**
     * 区间查询：从 left 到 right 的和
     *
     * @param left 左边界
```

```
         * @param right 右边界
         * @return 区间和
         */
        int rangeQuery(int left, int right) {
            if (left > right) return 0;
            return query(right) - query(left - 1);
        }
    };

public:
    /**
     * 计算满足条件的区间和个数
     *
     * @param nums 输入数组
     * @param lower 区间下界
     * @param upper 区间上界
     * @return 满足条件的区间个数
     *
     * 算法步骤：
     * 1. 计算前缀和数组
     * 2. 离散化处理所有可能的前缀和值
     * 3. 使用树状数组统计前缀和出现次数
     * 4. 遍历前缀和数组，统计满足条件的区间个数
     */
    int countRangeSum(vector<int>& nums, int lower, int upper) {
        if (nums.empty()) {
            return 0;
        }

        if (lower > upper) {
            return 0;
        }

        int n = nums.size();

        // 1. 计算前缀和数组（使用 long long 避免溢出）
        vector<long long> prefixSum(n + 1, 0);
        for (int i = 0; i < n; i++) {
            prefixSum[i + 1] = prefixSum[i] + nums[i];
        }

        // 2. 离散化处理：收集所有需要离散化的值
        set<long long> valueSet;
```

```
        for (long long sum : prefixSum) {
            valueSet.insert(sum);
            valueSet.insert(sum - lower);
            valueSet.insert(sum - upper);
        }

        // 构建离散化映射
        map<long long, int> valueMap;
        int idx = 1;
        for (long long num : valueSet) {
            valueMap[num] = idx++;
        }

        // 3. 使用树状数组统计前缀和出现次数
        FenwickTree tree(valueMap.size());
        int count = 0;

        // 从右向左遍历前缀和数组
        for (int i = 0; i <= n; i++) {
            long long currentSum = prefixSum[i];
            long long leftBound = currentSum - upper;
            long long rightBound = currentSum - lower;

            // 查询满足条件的区间和个数
            int leftIdx = valueMap[leftBound];
            int rightIdx = valueMap[rightBound];

            count += tree.rangeQuery(leftIdx, rightIdx);

            // 更新当前前缀和的出现次数
            int currentIdx = valueMap[currentSum];
            tree.update(currentIdx, 1);
        }

        return count;
    }
};

/**
 * 测试函数：验证算法正确性
 *
 * 测试用例设计：
 * 1. 正常情况测试
```

```
 * 2. 边界情况测试
 * 3. 空数组测试
 * 4. 大数值测试
 */
void testCountRangeSum() {
    Solution solution;

    // 测试用例 1：正常情况
    vector<int> nums1 = {-2, 5, -1};
    int lower1 = -2, upper1 = 2;
    int result1 = solution.countRangeSum(nums1, lower1, upper1);
    cout << "测试用例 1 结果：" << result1 << " (期望：3)" << endl;

    // 测试用例 2：边界情况
    vector<int> nums2 = {0};
    int lower2 = 0, upper2 = 0;
    int result2 = solution.countRangeSum(nums2, lower2, upper2);
    cout << "测试用例 2 结果：" << result2 << " (期望：1)" << endl;

    // 测试用例 3：空数组
    vector<int> nums3 = {};
    int result3 = solution.countRangeSum(nums3, 0, 0);
    cout << "测试用例 3 结果：" << result3 << " (期望：0)" << endl;

    // 测试用例 4：大数值
    vector<int> nums4 = {2147483647, -2147483648, -1, 0};
    int lower4 = -1, upper4 = 0;
    int result4 = solution.countRangeSum(nums4, lower4, upper4);
    cout << "测试用例 4 结果：" << result4 << " (期望：4)" << endl;

    cout << "所有测试用例执行完成！" << endl;
}

int main() {
    try {
        testCountRangeSum();
    } catch (const exception& e) {
        cerr << "错误：" << e.what() << endl;
        return 1;
    }

    return 0;
}
```

==========

文件: Code22_CountOfRangeSum.java
==========
```
package class132;

import java.util.*;

/**
 * LeetCode 327. 区间和的个数 (Count of Range Sum)
 *
 * 题目描述：
 * 给定一个整数数组 nums，以及两个整数 lower 和 upper。
 * 返回区间和的值在区间 [lower, upper] 之间的区间个数（包含等于）。
 *
 * 解题思路：
 * 使用树状数组（Fenwick Tree）+ 离散化来高效统计满足条件的区间和个数。
 * 核心思想：
 * 1. 计算前缀和数组 prefixSum
 * 2. 对于每个前缀和 prefixSum[j]，需要统计有多少个 i < j 满足：
 *    lower <= prefixSum[j] - prefixSum[i] <= upper
 *    即：prefixSum[j] - upper <= prefixSum[i] <= prefixSum[j] - lower
 * 3. 使用树状数组维护前缀和的出现次数
 * 4. 通过离散化处理大数值范围问题
 *
 * 时间复杂度分析：
 * - 前缀和计算：O(n)
 * - 离散化处理：O(n log n)
 * - 树状数组操作：O(n log n)
 * - 总时间复杂度：O(n log n)
 *
 * 空间复杂度分析：
 * - 前缀和数组：O(n)
 * - 离散化数组：O(n)
 * - 树状数组：O(n)
 * - 总空间复杂度：O(n)
 *
 * 工程化考量：
 * 1. 边界条件处理：处理空数组、lower > upper 等情况
 * 2. 数值溢出处理：使用 long 类型避免整数溢出
 * 3. 离散化优化：使用 TreeSet 去重并排序
 * 4. 异常处理：检查输入参数合法性
```

```
 *
 * 算法技巧：
 * - 离散化：将大范围的数值映射到小范围的索引
 * - 树状数组：高效统计前缀和的出现次数
 * - 容斥原理：通过区间查询统计满足条件的个数
 *
 * 适用场景：
 * - 需要统计满足特定条件的区间和个数
 * - 数值范围较大，需要离散化处理
 * - 对时间复杂度要求较高的场景
 *
 * 测试用例：
 * 输入：nums = [-2,5,-1], lower = -2, upper = 2
 * 输出：3
 * 解释：三个区间和满足条件：[0,0], [2,2], [0,2]
 */
public class Code22_CountOfRangeSum {

    /**
     * 计算满足条件的区间和个数
     *
     * @param nums 输入数组
     * @param lower 区间下界
     * @param upper 区间上界
     * @return 满足条件的区间个数
     *
     * 算法步骤：
     * 1. 计算前缀和数组
     * 2. 离散化处理所有可能的前缀和值
     * 3. 使用树状数组统计前缀和出现次数
     * 4. 遍历前缀和数组，统计满足条件的区间个数
     */
    public int countRangeSum(int[] nums, int lower, int upper) {
        if (nums == null || nums.length == 0) {
            return 0;
        }

        if (lower > upper) {
            return 0;
        }

        int n = nums.length;
```

```
// 1. 计算前缀和数组（使用 long 避免溢出）
long[] prefixSum = new long[n + 1];
for (int i = 0; i < n; i++) {
    prefixSum[i + 1] = prefixSum[i] + nums[i];
}

// 2. 离散化处理：收集所有需要离散化的值
TreeSet<Long> set = new TreeSet<>();
for (long sum : prefixSum) {
    set.add(sum);
    set.add(sum - lower);
    set.add(sum - upper);
}

// 构建离散化映射
Map<Long, Integer> map = new HashMap<>();
int idx = 1;
for (long num : set) {
    map.put(num, idx++);
}

// 3. 使用树状数组统计前缀和出现次数
FenwickTree tree = new FenwickTree(map.size());
int count = 0;

// 从右向左遍历前缀和数组
for (int i = 0; i <= n; i++) {
    long currentSum = prefixSum[i];
    long leftBound = currentSum - upper;
    long rightBound = currentSum - lower;

    // 查询满足条件的区间和个数
    int leftIdx = map.get(leftBound);
    int rightIdx = map.get(rightBound);

    count += tree.rangeQuery(leftIdx, rightIdx);

    // 更新当前前缀和的出现次数
    int currentIdx = map.get(currentSum);
    tree.update(currentIdx, 1);
}

return count;
```

```
}

/**
 * 树状数组（Fenwick Tree）实现
 * 用于高效统计前缀和的出现次数
 */
private static class FenwickTree {
    private int[] tree;
    private int size;

    /**
     * 构造函数
     *
     * @param size 树状数组大小
     */
    public FenwickTree(int size) {
        this.size = size;
        this.tree = new int[size + 1];
    }

    /**
     * 更新操作：在指定位置增加一个值
     *
     * @param index 位置索引
     * @param delta 增加值
     */
    public void update(int index, int delta) {
        while (index <= size) {
            tree[index] += delta;
            index += lowbit(index);
        }
    }

    /**
     * 查询前缀和：从 1 到 index 的和
     *
     * @param index 位置索引
     * @return 前缀和
     */
    public int query(int index) {
        int sum = 0;
        while (index > 0) {
            sum += tree[index];
```

```
            index -= lowbit(index);
        }
        return sum;
    }

    /**
     * 区间查询：从 left 到 right 的和
     *
     * @param left 左边界
     * @param right 右边界
     * @return 区间和
     */
    public int rangeQuery(int left, int right) {
        if (left > right) return 0;
        return query(right) - query(left - 1);
    }

    /**
     * 计算 lowbit（最低位的 1）
     *
     * @param x 输入数字
     * @return lowbit 值
     */
    private int lowbit(int x) {
        return x & -x;
    }
}

/**
 * 测试方法：验证算法正确性
 *
 * 测试用例设计：
 * 1. 正常情况测试
 * 2. 边界情况测试
 * 3. 空数组测试
 * 4. 大数值测试
 */
public static void main(String[] args) {
    Code22_CountOfRangeSum solution = new Code22_CountOfRangeSum();

    // 测试用例 1：正常情况
    int[] nums1 = {-2, 5, -1};
    int lower1 = -2, upper1 = 2;
```

```
        int result1 = solution.countRangeSum(nums1, lower1, upper1);
        System.out.println("测试用例 1 结果：" + result1 + " (期望：3)");

        // 测试用例 2：边界情况
        int[] nums2 = {0};
        int lower2 = 0, upper2 = 0;
        int result2 = solution.countRangeSum(nums2, lower2, upper2);
        System.out.println("测试用例 2 结果：" + result2 + " (期望：1)");

        // 测试用例 3：空数组
        int[] nums3 = {};
        int result3 = solution.countRangeSum(nums3, 0, 0);
        System.out.println("测试用例 3 结果：" + result3 + " (期望：0)");

        // 测试用例 4：大数值
        int[] nums4 = {2147483647, -2147483648, -1, 0};
        int lower4 = -1, upper4 = 0;
        int result4 = solution.countRangeSum(nums4, lower4, upper4);
        System.out.println("测试用例 4 结果：" + result4 + " (期望：4)");

        System.out.println("所有测试用例执行完成！");
    }
}
```

==========

文件: Code22_CountOfRangeSum.py

==========

"""

LeetCode 327. 区间和的个数 (Count of Range Sum)

题目描述：

给定一个整数数组 nums，以及两个整数 lower 和 upper。

返回区间和的值在区间 [lower, upper] 之间的区间个数（包含等于）。

解题思路：

使用树状数组（Fenwick Tree）+ 离散化来高效统计满足条件的区间和个数。

核心思想：

1. 计算前缀和数组 prefixSum
2. 对于每个前缀和 prefixSum[j]，需要统计有多少个 i < j 满足：
   lower <= prefixSum[j] - prefixSum[i] <= upper
   即：prefixSum[j] - upper <= prefixSum[i] <= prefixSum[j] - lower
3. 使用树状数组维护前缀和的出现次数

4. 通过离散化处理大数值范围问题

时间复杂度分析：
- 前缀和计算：O(n)
- 离散化处理：O(n log n)
- 树状数组操作：O(n log n)
- 总时间复杂度：O(n log n)

空间复杂度分析：
- 前缀和数组：O(n)
- 离散化数组：O(n)
- 树状数组：O(n)
- 总空间复杂度：O(n)

工程化考量：
1. 边界条件处理：处理空数组、lower > upper 等情况
2. 数值溢出处理：使用 Python 的 int 类型自动处理大整数
3. 离散化优化：使用 set 去重并排序
4. 异常处理：检查输入参数合法性

算法技巧：
- 离散化：将大范围的数值映射到小范围的索引
- 树状数组：高效统计前缀和的出现次数
- 容斥原理：通过区间查询统计满足条件的个数

适用场景：
- 需要统计满足特定条件的区间和个数
- 数值范围较大，需要离散化处理
- 对时间复杂度要求较高的场景

测试用例：
输入：nums = [-2,5,-1], lower = -2, upper = 2
输出：3
解释：三个区间和满足条件：[0,0], [2,2], [0,2]
"""

```
from typing import List


class FenwickTree:
    """
    树状数组（Fenwick Tree）实现
    用于高效统计前缀和的出现次数
```

```
    """

    def __init__(self, size: int):
        """
        构造函数

        Args:
            size: 树状数组大小
        """
        self.size = size
        self.tree = [0] * (size + 1)

    def _lowbit(self, x: int) -> int:
        """
        计算 lowbit（最低位的 1）

        Args:
            x: 输入数字

        Returns:
            lowbit 值
        """
        return x & -x

    def update(self, index: int, delta: int):
        """
        更新操作：在指定位置增加一个值

        Args:
            index: 位置索引
            delta: 增加值
        """
        while index <= self.size:
            self.tree[index] += delta
            index += self._lowbit(index)

    def query(self, index: int) -> int:
        """
        查询前缀和：从 1 到 index 的和

        Args:
            index: 位置索引
```

```
        Returns:
            前缀和
        """
        total = 0
        while index > 0:
            total += self.tree[index]
            index -= self._lowbit(index)
        return total

    def range_query(self, left: int, right: int) -> int:
        """
        区间查询：从 left 到 right 的和

        Args:
            left: 左边界
            right: 右边界

        Returns:
            区间和
        """
        if left > right:
            return 0
        return self.query(right) - self.query(left - 1)


class Solution:
    """
    区间和个数统计解决方案类
    """

    def countRangeSum(self, nums: List[int], lower: int, upper: int) -> int:
        """
        计算满足条件的区间和个数

        Args:
            nums: 输入数组
            lower: 区间下界
            upper: 区间上界

        Returns:
            满足条件的区间个数

        算法步骤：
```

```
1. 计算前缀和数组
2. 离散化处理所有可能的前缀和值
3. 使用树状数组统计前缀和出现次数
4. 遍历前缀和数组，统计满足条件的区间个数
"""
if not nums:
    return 0

if lower > upper:
    return 0

n = len(nums)

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

# 2. 离散化处理：收集所有需要离散化的值
value_set = set()
for s in prefix_sum:
    value_set.add(s)
    value_set.add(s - lower)
    value_set.add(s - upper)

# 构建离散化映射
sorted_values = sorted(value_set)
value_map = {val: idx + 1 for idx, val in enumerate(sorted_values)}

# 3. 使用树状数组统计前缀和出现次数
tree = FenwickTree(len(sorted_values))
count = 0

# 从右向左遍历前缀和数组
for s in prefix_sum:
    left_bound = s - upper
    right_bound = s - lower

    # 查询满足条件的区间和个数
    left_idx = value_map[left_bound]
    right_idx = value_map[right_bound]

    count += tree.range_query(left_idx, right_idx)
```

```
            # 更新当前前缀和的出现次数
            current_idx = value_map[s]
            tree.update(current_idx, 1)

        return count


def test_count_range_sum():
    """
    测试函数：验证算法正确性

    测试用例设计：
    1. 正常情况测试
    2. 边界情况测试
    3. 空数组测试
    4. 大数值测试
    """
    solution = Solution()

    # 测试用例 1：正常情况
    nums1 = [-2, 5, -1]
    lower1, upper1 = -2, 2
    result1 = solution.countRangeSum(nums1, lower1, upper1)
    print(f"测试用例 1 结果：{result1} (期望：3)")

    # 测试用例 2：边界情况
    nums2 = [0]
    lower2, upper2 = 0, 0
    result2 = solution.countRangeSum(nums2, lower2, upper2)
    print(f"测试用例 2 结果：{result2} (期望：1)")

    # 测试用例 3：空数组
    nums3 = []
    result3 = solution.countRangeSum(nums3, 0, 0)
    print(f"测试用例 3 结果：{result3} (期望：0)")

    # 测试用例 4：大数值
    nums4 = [2147483647, -2147483648, -1, 0]
    lower4, upper4 = -1, 0
    result4 = solution.countRangeSum(nums4, lower4, upper4)
    print(f"测试用例 4 结果：{result4} (期望：4)")
```

```
    print("所有测试用例执行完成！")


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

==========

文件: Code23_CountOfSmallerNumbersAfterSelf.cpp
==========

```
/**
 * LeetCode 315. 计算右侧小于当前元素的个数 (Count of Smaller Numbers After Self)
 *
 * 题目描述：
 * 给定一个整数数组 nums，按要求返回一个新数组 counts。
 * 数组 counts 有该性质：counts[i] 的值是 nums[i] 右侧小于 nums[i] 的元素的数量。
 *
 * 解题思路：
 * 使用树状数组（Fenwick Tree）+ 离散化来高效统计右侧小于当前元素的个数。
 * 核心思想：
 * 1. 从右向左遍历数组
 * 2. 对于每个元素 nums[i]，需要统计右侧已经遍历过的元素中小于 nums[i] 的个数
 * 3. 使用树状数组维护已经遍历过的元素的出现次数
 * 4. 通过离散化处理大数值范围问题
 *
 * 时间复杂度分析：
 * - 离散化处理：O(n log n)
 * - 树状数组操作：O(n log n)
 * - 总时间复杂度：O(n log n)
 *
 * 空间复杂度分析：
 * - 离散化数组：O(n)
 * - 树状数组：O(n)
 * - 结果数组：O(n)
 * - 总空间复杂度：O(n)
 *
 * 工程化考量：
 * 1. 边界条件处理：处理空数组、单个元素等情况
 * 2. 数值范围处理：使用离散化处理大数值范围
 * 3. 异常处理：检查输入参数合法性
 * 4. 性能优化：使用树状数组提高统计效率
 *
 * 算法技巧：
```

```
 * - 离散化：将大范围的数值映射到小范围的索引
 * - 树状数组：高效统计元素出现次数
 * - 逆序遍历：从右向左处理，便于统计右侧元素
 *
 * 适用场景：
 * - 需要统计数组中每个元素右侧小于它的元素个数
 * - 数值范围较大，需要离散化处理
 * - 对时间复杂度要求较高的场景
 *
 * 测试用例：
 * 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 的右侧仅有 1 个更小的元素 (1)
 * 6 的右侧有 1 个更小的元素 (1)
 * 1 的右侧有 0 个更小的元素
 */

#include <iostream>
#include <vector>
#include <algorithm>
#include <map>
#include <stdexcept>

using namespace std;

class Solution {
private:
    /**
     * 树状数组（Fenwick Tree）实现
     * 用于高效统计元素出现次数
     */
    class FenwickTree {
    private:
        vector<int> tree;
        int size;

        /**
         * 计算 lowbit（最低位的 1）
         *
         * @param x 输入数字
         * @return lowbit 值
```

```
     */
    int lowbit(int x) {
        return x & -x;
    }

public:
    /**
     * 构造函数
     *
     * @param size 树状数组大小
     */
    FenwickTree(int size) : size(size) {
        tree.resize(size + 1, 0);
    }

    /**
     * 更新操作：在指定位置增加一个值
     *
     * @param index 位置索引
     * @param delta 增加值
     */
    void update(int index, int delta) {
        while (index <= size) {
            tree[index] += delta;
            index += lowbit(index);
        }
    }

    /**
     * 查询前缀和：从 1 到 index 的和
     *
     * @param index 位置索引
     * @return 前缀和
     */
    int query(int index) {
        int sum = 0;
        while (index > 0) {
            sum += tree[index];
            index -= lowbit(index);
        }
        return sum;
    }
};
```

```
public:
    /**
     * 计算右侧小于当前元素的个数
     *
     * @param nums 输入数组
     * @return 结果数组，counts[i] 表示 nums[i] 右侧小于 nums[i] 的元素个数
     *
     * 算法步骤：
     * 1. 离散化处理数组元素
     * 2. 从右向左遍历数组
     * 3. 对于每个元素，查询树状数组中比它小的元素个数
     * 4. 更新树状数组，记录当前元素的出现
     */
    vector<int> countSmaller(vector<int>& nums) {
        if (nums.empty()) {
            return {};
        }

        int n = nums.size();
        vector<int> result(n, 0);

        // 1. 离散化处理
        vector<int> sortedNums = nums;
        sort(sortedNums.begin(), sortedNums.end());

        map<int, int> rankMap;
        int rank = 1;
        for (int i = 0; i < n; i++) {
            if (i == 0 || sortedNums[i] != sortedNums[i - 1]) {
                rankMap[sortedNums[i]] = rank++;
            }
        }

        // 2. 使用树状数组统计
        FenwickTree tree(rank - 1);

        // 从右向左遍历
        for (int i = n - 1; i >= 0; i--) {
            int currentRank = rankMap[nums[i]];

            // 查询比当前元素小的元素个数（即排名比当前小的元素）
            int count = tree.query(currentRank - 1);
```

```
            result[i] = count;

            // 更新树状数组，记录当前元素的出现
            tree.update(currentRank, 1);
        }

        return result;
    }

    /**
     * 方法二：使用归并排序统计逆序对（备选方案）
     *
     * 解题思路：
     * 1. 使用归并排序的过程统计右侧小于当前元素的个数
     * 2. 在合并两个有序数组时，统计右侧较小元素的个数
     * 3. 这种方法同样具有 O(n log n) 的时间复杂度
     *
     * 时间复杂度：O(n log n)
     * 空间复杂度：O(n)
     */
    vector<int> countSmallerMergeSort(vector<int>& nums) {
        if (nums.empty()) {
            return {};
        }

        int n = nums.size();
        vector<int> indexes(n);
        vector<int> counts(n, 0);

        for (int i = 0; i < n; i++) {
            indexes[i] = i;
        }

        vector<int> tempIndexes(n);
        mergeSort(nums, indexes, counts, tempIndexes, 0, n - 1);

        return counts;
    }

private:
    void mergeSort(vector<int>& nums, vector<int>& indexes, vector<int>& counts,
                   vector<int>& tempIndexes, int start, int end) {
        if (start >= end) {
```

```
        return;
    }

    int mid = start + (end - start) / 2;
    mergeSort(nums, indexes, counts, tempIndexes, start, mid);
    mergeSort(nums, indexes, counts, tempIndexes, mid + 1, end);
    merge(nums, indexes, counts, tempIndexes, start, mid, end);
}

void merge(vector<int>& nums, vector<int>& indexes, vector<int>& counts,
           vector<int>& tempIndexes, int start, int mid, int end) {
    int left = start, right = mid + 1;
    int rightCount = 0;
    int index = 0;

    while (left <= mid && right <= end) {
        if (nums[indexes[right]] < nums[indexes[left]]) {
            tempIndexes[index] = indexes[right];
            rightCount++;
            right++;
        } else {
            tempIndexes[index] = indexes[left];
            counts[indexes[left]] += rightCount;
            left++;
        }
        index++;
    }

    while (left <= mid) {
        tempIndexes[index] = indexes[left];
        counts[indexes[left]] += rightCount;
        left++;
        index++;
    }

    while (right <= end) {
        tempIndexes[index] = indexes[right];
        right++;
        index++;
    }

    for (int i = 0; i < index; i++) {
        indexes[start + i] = tempIndexes[i];
```

```
        }
    }
};

/**
 * 测试函数：验证算法正确性
 *
 * 测试用例设计：
 * 1. 正常情况测试
 * 2. 边界情况测试
 * 3. 空数组测试
 * 4. 重复元素测试
 */
void testCountSmaller() {
    Solution solution;

    // 测试用例 1：正常情况
    vector<int> nums1 = {5, 2, 6, 1};
    vector<int> result1 = solution.countSmaller(nums1);
    cout << "测试用例 1 结果：";
    for (int num : result1) {
        cout << num << " ";
    }
    cout << "(期望：2 1 1 0)" << endl;

    // 测试用例 2：边界情况
    vector<int> nums2 = {-1};
    vector<int> result2 = solution.countSmaller(nums2);
    cout << "测试用例 2 结果：";
    for (int num : result2) {
        cout << num << " ";
    }
    cout << "(期望：0)" << endl;

    // 测试用例 3：空数组
    vector<int> nums3 = {};
    vector<int> result3 = solution.countSmaller(nums3);
    cout << "测试用例 3 结果：";
    for (int num : result3) {
        cout << num << " ";
    }
    cout << "(期望：空)" << endl;
```

```
    // 测试用例 4：重复元素
    vector<int> nums4 = {2, 2, 2, 2};
    vector<int> result4 = solution.countSmaller(nums4);
    cout << "测试用例 4 结果：";
    for (int num : result4) {
        cout << num << " ";
    }
    cout << "(期望：0 0 0 0)" << endl;

    // 测试用例 5：大数值
    vector<int> nums5 = {2147483647, -2147483648, 0, 1};
    vector<int> result5 = solution.countSmaller(nums5);
    cout << "测试用例 5 结果：";
    for (int num : result5) {
        cout << num << " ";
    }
    cout << "(期望：3 0 0 0)" << endl;

    cout << "所有测试用例执行完成！" << endl;
}

int main() {
    try {
        testCountSmaller();
    } catch (const exception& e) {
        cerr << "错误：" << e.what() << endl;
        return 1;
    }

    return 0;
}

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

文件: Code23_CountOfSmallerNumbersAfterSelf.java
================================================
package class132;

import java.util.*;

/**
 * LeetCode 315. 计算右侧小于当前元素的个数 (Count of Smaller Numbers After Self)
 *
```

* 题目描述：
* 给定一个整数数组 nums，按要求返回一个新数组 counts。
* 数组 counts 有该性质：counts[i] 的值是 nums[i] 右侧小于 nums[i] 的元素的数量。
*
* 解题思路：
* 使用树状数组（Fenwick Tree）+ 离散化来高效统计右侧小于当前元素的个数。
* 核心思想：
* 1. 从右向左遍历数组
* 2. 对于每个元素 nums[i]，需要统计右侧已经遍历过的元素中小于 nums[i] 的个数
* 3. 使用树状数组维护已经遍历过的元素的出现次数
* 4. 通过离散化处理大数值范围问题
*
* 时间复杂度分析：
* - 离散化处理：O(n log n)
* - 树状数组操作：O(n log n)
* - 总时间复杂度：O(n log n)
*
* 空间复杂度分析：
* - 离散化数组：O(n)
* - 树状数组：O(n)
* - 结果数组：O(n)
* - 总空间复杂度：O(n)
*
* 工程化考量：
* 1. 边界条件处理：处理空数组、单个元素等情况
* 2. 数值范围处理：使用离散化处理大数值范围
* 3. 异常处理：检查输入参数合法性
* 4. 性能优化：使用树状数组提高统计效率
*
* 算法技巧：
* - 离散化：将大范围的数值映射到小范围的索引
* - 树状数组：高效统计元素出现次数
* - 逆序遍历：从右向左处理，便于统计右侧元素
*
* 适用场景：
* - 需要统计数组中每个元素右侧小于它的元素个数
* - 数值范围较大，需要离散化处理
* - 对时间复杂度要求较高的场景
*
* 测试用例：
* 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 的右侧仅有 1 个更小的元素 (1)
 * 6 的右侧有 1 个更小的元素 (1)
 * 1 的右侧有 0 个更小的元素
 */
public class Code23_CountOfSmallerNumbersAfterSelf {

    /**
     * 计算右侧小于当前元素的个数
     *
     * @param nums 输入数组
     * @return 结果数组，counts[i] 表示 nums[i] 右侧小于 nums[i] 的元素个数
     *
     * 算法步骤：
     * 1. 离散化处理数组元素
     * 2. 从右向左遍历数组
     * 3. 对于每个元素，查询树状数组中比它小的元素个数
     * 4. 更新树状数组，记录当前元素的出现
     */
    public List<Integer> countSmaller(int[] nums) {
        if (nums == null || nums.length == 0) {
            return new ArrayList<>();
        }

        int n = nums.length;
        List<Integer> result = new ArrayList<>(Collections.nCopies(n, 0));

        // 1. 离散化处理
        int[] sortedNums = Arrays.copyOf(nums, n);
        Arrays.sort(sortedNums);

        Map<Integer, Integer> rankMap = new HashMap<>();
        int rank = 1;
        for (int i = 0; i < n; i++) {
            if (i == 0 || sortedNums[i] != sortedNums[i - 1]) {
                rankMap.put(sortedNums[i], rank++);
            }
        }

        // 2. 使用树状数组统计
        FenwickTree tree = new FenwickTree(rank - 1);

        // 从右向左遍历
```

```
    for (int i = n - 1; i >= 0; i--) {
        int currentRank = rankMap.get(nums[i]);

        // 查询比当前元素小的元素个数（即排名比当前小的元素）
        int count = tree.query(currentRank - 1);
        result.set(i, count);

        // 更新树状数组，记录当前元素的出现
        tree.update(currentRank, 1);
    }

    return result;
}

/**
 * 树状数组（Fenwick Tree）实现
 * 用于高效统计元素出现次数
 */
private static class FenwickTree {
    private int[] tree;
    private int size;

    /**
     * 构造函数
     *
     * @param size 树状数组大小
     */
    public FenwickTree(int size) {
        this.size = size;
        this.tree = new int[size + 1];
    }

    /**
     * 更新操作：在指定位置增加一个值
     *
     * @param index 位置索引
     * @param delta 增加值
     */
    public void update(int index, int delta) {
        while (index <= size) {
            tree[index] += delta;
            index += lowbit(index);
        }
```

```
    }

    /**
     * 查询前缀和：从 1 到 index 的和
     *
     * @param index 位置索引
     * @return 前缀和
     */
    public int query(int index) {
        int sum = 0;
        while (index > 0) {
            sum += tree[index];
            index -= lowbit(index);
        }
        return sum;
    }

    /**
     * 计算 lowbit（最低位的 1）
     *
     * @param x 输入数字
     * @return lowbit 值
     */
    private int lowbit(int x) {
        return x & -x;
    }
}

/**
 * 方法二：使用归并排序统计逆序对（备选方案）
 *
 * 解题思路：
 * 1. 使用归并排序的过程统计右侧小于当前元素的个数
 * 2. 在合并两个有序数组时，统计右侧较小元素的个数
 * 3. 这种方法同样具有 O(n log n) 的时间复杂度
 *
 * 时间复杂度：O(n log n)
 * 空间复杂度：O(n)
 */
public List<Integer> countSmallerMergeSort(int[] nums) {
    if (nums == null || nums.length == 0) {
        return new ArrayList<>();
    }
```

```
    int n = nums.length;
    int[] indexes = new int[n];
    int[] counts = new int[n];

    for (int i = 0; i < n; i++) {
        indexes[i] = i;
    }

    mergeSort(nums, indexes, counts, 0, n - 1);

    List<Integer> result = new ArrayList<>();
    for (int count : counts) {
        result.add(count);
    }
    return result;
}

private void mergeSort(int[] nums, int[] indexes, int[] counts, int start, int end) {
    if (start >= end) {
        return;
    }

    int mid = start + (end - start) / 2;
    mergeSort(nums, indexes, counts, start, mid);
    mergeSort(nums, indexes, counts, mid + 1, end);
    merge(nums, indexes, counts, start, mid, end);
}

private void merge(int[] nums, int[] indexes, int[] counts, int start, int mid, int end) {
    int[] tempIndexes = new int[end - start + 1];
    int left = start, right = mid + 1;
    int rightCount = 0;
    int index = 0;

    while (left <= mid && right <= end) {
        if (nums[indexes[right]] < nums[indexes[left]]) {
            tempIndexes[index] = indexes[right];
            rightCount++;
            right++;
        } else {
            tempIndexes[index] = indexes[left];
            counts[indexes[left]] += rightCount;
```

```
                left++;
            }
            index++;
        }

        while (left <= mid) {
            tempIndexes[index] = indexes[left];
            counts[indexes[left]] += rightCount;
            left++;
            index++;
        }

        while (right <= end) {
            tempIndexes[index] = indexes[right];
            right++;
            index++;
        }

        System.arraycopy(tempIndexes, 0, indexes, start, end - start + 1);
    }

    /**
     * 测试方法：验证算法正确性
     *
     * 测试用例设计：
     * 1. 正常情况测试
     * 2. 边界情况测试
     * 3. 空数组测试
     * 4. 重复元素测试
     */
    public static void main(String[] args) {
        Code23_CountOfSmallerNumbersAfterSelf solution = new
Code23_CountOfSmallerNumbersAfterSelf();

        // 测试用例 1：正常情况
        int[] nums1 = {5, 2, 6, 1};
        List<Integer> result1 = solution.countSmaller(nums1);
        System.out.println("测试用例 1 结果：" + result1 + " (期望：[2, 1, 1, 0])");

        // 测试用例 2：边界情况
        int[] nums2 = {-1};
        List<Integer> result2 = solution.countSmaller(nums2);
        System.out.println("测试用例 2 结果：" + result2 + " (期望：[0])");
```

```
        // 测试用例 3：空数组
        int[] nums3 = {};
        List<Integer> result3 = solution.countSmaller(nums3);
        System.out.println("测试用例 3 结果：" + result3 + " (期望：[])");

        // 测试用例 4：重复元素
        int[] nums4 = {2, 2, 2, 2};
        List<Integer> result4 = solution.countSmaller(nums4);
        System.out.println("测试用例 4 结果：" + result4 + " (期望：[0, 0, 0, 0])");

        // 测试用例 5：大数值
        int[] nums5 = {2147483647, -2147483648, 0, 1};
        List<Integer> result5 = solution.countSmaller(nums5);
        System.out.println("测试用例 5 结果：" + result5 + " (期望：[3, 0, 0, 0])");

        System.out.println("所有测试用例执行完成！");
    }
}
```

==========

文件: Code23_CountOfSmallerNumbersAfterSelf.py
==========

"""
LeetCode 315. 计算右侧小于当前元素的个数 (Count of Smaller Numbers After Self)

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

解题思路：
使用树状数组（Fenwick Tree）+ 离散化来高效统计右侧小于当前元素的个数。
核心思想：
1. 从右向左遍历数组
2. 对于每个元素 nums[i]，需要统计右侧已经遍历过的元素中小于 nums[i] 的个数
3. 使用树状数组维护已经遍历过的元素的出现次数
4. 通过离散化处理大数值范围问题

时间复杂度分析：
- 离散化处理：O(n log n)
- 树状数组操作：O(n log n)
- 总时间复杂度：O(n log n)

空间复杂度分析：
- 离散化数组：O(n)
- 树状数组：O(n)
- 结果数组：O(n)
- 总空间复杂度：O(n)

工程化考量：
1. 边界条件处理：处理空数组、单个元素等情况
2. 数值范围处理：使用离散化处理大数值范围
3. 异常处理：检查输入参数合法性
4. 性能优化：使用树状数组提高统计效率

算法技巧：
- 离散化：将大范围的数值映射到小范围的索引
- 树状数组：高效统计元素出现次数
- 逆序遍历：从右向左处理，便于统计右侧元素

适用场景：
- 需要统计数组中每个元素右侧小于它的元素个数
- 数值范围较大，需要离散化处理
- 对时间复杂度要求较高的场景

测试用例：
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 的右侧仅有 1 个更小的元素 (1)
6 的右侧有 1 个更小的元素 (1)
1 的右侧有 0 个更小的元素
"""

```
from typing import List


class FenwickTree:
    """
    树状数组（Fenwick Tree）实现
    用于高效统计元素出现次数
    """

    def __init__(self, size: int):
```

```
    """
    构造函数

    Args:
        size: 树状数组大小
    """
    self.size = size
    self.tree = [0] * (size + 1)

def _lowbit(self, x: int) -> int:
    """
    计算 lowbit（最低位的 1）

    Args:
        x: 输入数字

    Returns:
        lowbit 值
    """
    return x & -x

def update(self, index: int, delta: int):
    """
    更新操作：在指定位置增加一个值

    Args:
        index: 位置索引
        delta: 增加值
    """
    while index <= self.size:
        self.tree[index] += delta
        index += self._lowbit(index)

def query(self, index: int) -> int:
    """
    查询前缀和：从 1 到 index 的和

    Args:
        index: 位置索引

    Returns:
        前缀和
    """
```

```
        total = 0
        while index > 0:
            total += self.tree[index]
            index -= self._lowbit(index)
        return total


class Solution:
    """
    右侧小于当前元素个数统计解决方案类
    """

    def countSmaller(self, nums: List[int]) -> List[int]:
        """
        计算右侧小于当前元素的个数

        Args:
            nums: 输入数组

        Returns:
            结果数组，counts[i] 表示 nums[i] 右侧小于 nums[i] 的元素个数

        算法步骤：
        1. 离散化处理数组元素
        2. 从右向左遍历数组
        3. 对于每个元素，查询树状数组中比它小的元素个数
        4. 更新树状数组，记录当前元素的出现
        """
        if not nums:
            return []

        n = len(nums)
        result = [0] * n

        # 1. 离散化处理
        sorted_nums = sorted(nums)
        rank_map = {}
        rank = 1

        for i in range(n):
            if i == 0 or sorted_nums[i] != sorted_nums[i - 1]:
                rank_map[sorted_nums[i]] = rank
                rank += 1
```

```
        # 2. 使用树状数组统计
        tree = FenwickTree(rank - 1)

        # 从右向左遍历
        for i in range(n - 1, -1, -1):
            current_rank = rank_map[nums[i]]

            # 查询比当前元素小的元素个数（即排名比当前小的元素）
            count = tree.query(current_rank - 1)
            result[i] = count

            # 更新树状数组，记录当前元素的出现
            tree.update(current_rank, 1)

        return result

    def countSmallerMergeSort(self, nums: List[int]) -> List[int]:
        """
        方法二：使用归并排序统计逆序对（备选方案）

        解题思路：
        1. 使用归并排序的过程统计右侧小于当前元素的个数
        2. 在合并两个有序数组时，统计右侧较小元素的个数
        3. 这种方法同样具有 O(n log n) 的时间复杂度

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

        n = len(nums)
        indexes = list(range(n))
        counts = [0] * n

        self._mergeSort(nums, indexes, counts, 0, n - 1)
        return counts

    def _mergeSort(self, nums: List[int], indexes: List[int], counts: List[int], start: int, end:
int):
        """
        归并排序递归函数
```

```
        Args:
            nums: 原始数组
            indexes: 索引数组
            counts: 计数数组
            start: 起始位置
            end: 结束位置
        """
        if start >= end:
            return

        mid = start + (end - start) // 2
        self._mergeSort(nums, indexes, counts, start, mid)
        self._mergeSort(nums, indexes, counts, mid + 1, end)
        self._merge(nums, indexes, counts, start, mid, end)

    def _merge(self, nums: List[int], indexes: List[int], counts: List[int], start: int, mid:
int, end: int):
        """
        合并两个有序数组

        Args:
            nums: 原始数组
            indexes: 索引数组
            counts: 计数数组
            start: 起始位置
            mid: 中间位置
            end: 结束位置
        """
        temp_indexes = [0] * (end - start + 1)
        left, right = start, mid + 1
        right_count = 0
        index = 0

        while left <= mid and right <= end:
            if nums[indexes[right]] < nums[indexes[left]]:
                temp_indexes[index] = indexes[right]
                right_count += 1
                right += 1
            else:
                temp_indexes[index] = indexes[left]
                counts[indexes[left]] += right_count
                left += 1
```

```
            index += 1

        while left <= mid:
            temp_indexes[index] = indexes[left]
            counts[indexes[left]] += right_count
            left += 1
            index += 1

        while right <= end:
            temp_indexes[index] = indexes[right]
            right += 1
            index += 1

        # 将临时数组复制回原数组
        for i in range(len(temp_indexes)):
            indexes[start + i] = temp_indexes[i]


def test_count_smaller():
    """
    测试函数：验证算法正确性

    测试用例设计：
    1. 正常情况测试
    2. 边界情况测试
    3. 空数组测试
    4. 重复元素测试
    """
    solution = Solution()

    # 测试用例 1：正常情况
    nums1 = [5, 2, 6, 1]
    result1 = solution.countSmaller(nums1)
    print(f"测试用例 1 结果：{result1} (期望：[2, 1, 1, 0])")

    # 测试用例 2：边界情况
    nums2 = [-1]
    result2 = solution.countSmaller(nums2)
    print(f"测试用例 2 结果：{result2} (期望：[0])")

    # 测试用例 3：空数组
    nums3 = []
    result3 = solution.countSmaller(nums3)
```

```
    print(f"测试用例 3 结果：{result3} (期望：[])")

    # 测试用例 4：重复元素
    nums4 = [2, 2, 2, 2]
    result4 = solution.countSmaller(nums4)
    print(f"测试用例 4 结果：{result4} (期望：[0, 0, 0, 0])")

    # 测试用例 5：大数值
    nums5 = [2147483647, -2147483648, 0, 1]
    result5 = solution.countSmaller(nums5)
    print(f"测试用例 5 结果：{result5} (期望：[3, 0, 0, 0])")

    print("所有测试用例执行完成！")


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

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

文件: Code24_ReversePairs.cpp

```
==========================================
/**
 * LeetCode 493. 翻转对 (Reverse Pairs)
 *
 * 题目描述：
 * 给定一个数组 nums ，如果 i < j 且 nums[i] > 2*nums[j] 我们就将 (i,j) 称作一个重要翻转对。
 * 你需要返回给定数组中的重要翻转对的数量。
 *
 * 解题思路：
 * 使用树状数组（Fenwick Tree）+ 离散化来高效统计重要翻转对的数量。
 * 核心思想：
 * 1. 从右向左遍历数组
 * 2. 对于每个元素 nums[i]，需要统计右侧已经遍历过的元素中满足 nums[i] > 2*nums[j] 的个数
 * 3. 使用树状数组维护已经遍历过的元素的出现次数
 * 4. 通过离散化处理大数值范围问题
 *
 * 时间复杂度分析：
 * - 离散化处理：O(n log n)
 * - 树状数组操作：O(n log n)
 * - 总时间复杂度：O(n log n)
 *
 * 空间复杂度分析：
```

```
 * - 离散化数组：O(n)
 * - 树状数组：O(n)
 * - 总空间复杂度：O(n)
 *
 * 工程化考量：
 * 1. 边界条件处理：处理空数组、单个元素等情况
 * 2. 数值溢出处理：使用 long long 类型避免整数溢出
 * 3. 离散化优化：使用 set 去重并排序
 * 4. 异常处理：检查输入参数合法性
 *
 * 算法技巧：
 * - 离散化：将大范围的数值映射到小范围的索引
 * - 树状数组：高效统计元素出现次数
 * - 逆序遍历：从右向左处理，便于统计右侧元素
 * - 二分查找：快速定位满足条件的边界
 *
 * 适用场景：
 * - 需要统计数组中满足特定条件的翻转对数量
 * - 数值范围较大，需要离散化处理
 * - 对时间复杂度要求较高的场景
 *
 * 测试用例：
 * 输入：nums = [1,3,2,3,1]
 * 输出：2
 * 解释：两个重要翻转对：(1,4) 和 (3,4)
 */

#include <iostream>
#include <vector>
#include <set>
#include <map>
#include <algorithm>
#include <cmath>
#include <stdexcept>

using namespace std;

class Solution {
private:
    /**
     * 树状数组（Fenwick Tree）实现
     * 用于高效统计元素出现次数
     */
```

```
class FenwickTree {
private:
    vector<int> tree;
    int size;

    /**
     * 计算 lowbit（最低位的 1）
     *
     * @param x 输入数字
     * @return lowbit 值
     */
    int lowbit(int x) {
        return x & -x;
    }

public:
    /**
     * 构造函数
     *
     * @param size 树状数组大小
     */
    FenwickTree(int size) : size(size) {
        tree.resize(size + 1, 0);
    }

    /**
     * 更新操作：在指定位置增加一个值
     *
     * @param index 位置索引
     * @param delta 增加值
     */
    void update(int index, int delta) {
        while (index <= size) {
            tree[index] += delta;
            index += lowbit(index);
        }
    }

    /**
     * 查询前缀和：从 1 到 index 的和
     *
     * @param index 位置索引
     * @return 前缀和
```

```
         */
        int query(int index) {
            int sum = 0;
            while (index > 0) {
                sum += tree[index];
                index -= lowbit(index);
            }
            return sum;
        }
    };

public:
    /**
     * 计算重要翻转对的数量
     *
     * @param nums 输入数组
     * @return 重要翻转对的数量
     *
     * 算法步骤：
     * 1. 离散化处理数组元素
     * 2. 从右向左遍历数组
     * 3. 对于每个元素 nums[i]，需要找到满足 nums[i] > 2*nums[j] 的 nums[j] 的范围
     * 4. 使用树状数组统计该范围内的元素个数
     * 5. 更新树状数组，记录当前元素的出现
     */
    int reversePairs(vector<int>& nums) {
        if (nums.empty()) {
            return 0;
        }

        int n = nums.size();

        // 1. 离散化处理：收集所有需要离散化的值
        set<long long> valueSet;
        for (int num : nums) {
            valueSet.insert((long long)num);
            valueSet.insert(2LL * num);
        }

        // 构建离散化映射
        map<long long, int> valueMap;
        int idx = 1;
        for (long long num : valueSet) {
```

```
            valueMap[num] = idx++;
        }

        // 2. 使用树状数组统计
        FenwickTree tree(valueMap.size());
        int count = 0;

        // 从右向左遍历
        for (int i = n - 1; i >= 0; i--) {
            long long currentNum = nums[i];

            // 找到满足 nums[i] > 2*nums[j] 的最大 nums[j]
            // 即：nums[j] < nums[i] / 2.0
            long long maxAllowed = floor((currentNum - 1) / 2.0);

            // 如果 maxAllowed 在离散化映射中不存在，需要找到最大的小于等于 maxAllowed 的值
            auto it = valueSet.upper_bound(maxAllowed);
            if (it != valueSet.begin()) {
                it--;
                long long floorKey = *it;
                int targetIdx = valueMap[floorKey];
                count += tree.query(targetIdx);
            }

            // 更新树状数组，记录当前元素的出现
            int currentIdx = valueMap[currentNum];
            tree.update(currentIdx, 1);
        }

        return count;
    }

    /**
     * 方法二：使用归并排序统计翻转对（备选方案）
     *
     * 解题思路：
     * 1. 使用归并排序的过程统计重要翻转对的数量
     * 2. 在合并两个有序数组之前，先统计满足条件的翻转对
     * 3. 这种方法同样具有 O(n log n) 的时间复杂度
     *
     * 时间复杂度：O(n log n)
     * 空间复杂度：O(n)
     */
```

```
    int reversePairsMergeSort(vector<int>& nums) {
        if (nums.empty()) {
            return 0;
        }

        return mergeSort(nums, 0, nums.size() - 1);
    }

private:
    int mergeSort(vector<int>& nums, int left, int right) {
        if (left >= right) {
            return 0;
        }

        int mid = left + (right - left) / 2;
        int count = mergeSort(nums, left, mid) + mergeSort(nums, mid + 1, right);

        // 统计满足条件的翻转对
        int j = mid + 1;
        for (int i = left; i <= mid; i++) {
            while (j <= right && (long long)nums[i] > 2LL * nums[j]) {
                j++;
            }
            count += j - (mid + 1);
        }

        // 合并两个有序数组
        merge(nums, left, mid, right);

        return count;
    }

    void merge(vector<int>& nums, int left, int mid, int right) {
        vector<int> temp(right - left + 1);
        int i = left, j = mid + 1, k = 0;

        while (i <= mid && j <= right) {
            if (nums[i] <= nums[j]) {
                temp[k++] = nums[i++];
            } else {
                temp[k++] = nums[j++];
            }
        }
```

```
        while (i <= mid) {
            temp[k++] = nums[i++];
        }

        while (j <= right) {
            temp[k++] = nums[j++];
        }

        for (int idx = 0; idx < temp.size(); idx++) {
            nums[left + idx] = temp[idx];
        }
    }
};

/**
 * 测试函数：验证算法正确性
 *
 * 测试用例设计：
 * 1. 正常情况测试
 * 2. 边界情况测试
 * 3. 空数组测试
 * 4. 大数值测试
 */
void testReversePairs() {
    Solution solution;

    // 测试用例 1：正常情况
    vector<int> nums1 = {1, 3, 2, 3, 1};
    int result1 = solution.reversePairs(nums1);
    cout << "测试用例 1 结果：" << result1 << " (期望：2)" << endl;

    // 测试用例 2：边界情况
    vector<int> nums2 = {2, 4, 3, 5, 1};
    int result2 = solution.reversePairs(nums2);
    cout << "测试用例 2 结果：" << result2 << " (期望：3)" << endl;

    // 测试用例 3：空数组
    vector<int> nums3 = {};
    int result3 = solution.reversePairs(nums3);
    cout << "测试用例 3 结果：" << result3 << " (期望：0)" << endl;

    // 测试用例 4：大数值
```

```
    vector<int> nums4 = {2147483647, 2147483647, 2147483647, 2147483647, 2147483647};
    int result4 = solution.reversePairs(nums4);
    cout << "测试用例 4 结果：" << result4 << " (期望：0)" << endl;

    // 测试用例 5：负数情况
    vector<int> nums5 = {-5, -5};
    int result5 = solution.reversePairs(nums5);
    cout << "测试用例 5 结果：" << result5 << " (期望：1)" << endl;

    cout << "所有测试用例执行完成！" << endl;
}

int main() {
    try {
        testReversePairs();
    } catch (const exception& e) {
        cerr << "错误：" << e.what() << endl;
        return 1;
    }

    return 0;
}
```

==========

文件: Code24_ReversePairs.java

==========

```
package class132;

import java.util.*;

/**
 * LeetCode 493. 翻转对 (Reverse Pairs)
 *
 * 题目描述：
 * 给定一个数组 nums ，如果 i < j 且 nums[i] > 2*nums[j] 我们就将 (i,j) 称作一个重要翻转对。
 * 你需要返回给定数组中的重要翻转对的数量。
 *
 * 解题思路：
 * 使用树状数组（Fenwick Tree）+ 离散化来高效统计重要翻转对的数量。
 * 核心思想：
 * 1. 从右向左遍历数组
 * 2. 对于每个元素 nums[i]，需要统计右侧已经遍历过的元素中满足 nums[i] > 2*nums[j] 的个数
```

```
 * 3. 使用树状数组维护已经遍历过的元素的出现次数
 * 4. 通过离散化处理大数值范围问题
 *
 * 时间复杂度分析：
 * - 离散化处理：O(n log n)
 * - 树状数组操作：O(n log n)
 * - 总时间复杂度：O(n log n)
 *
 * 空间复杂度分析：
 * - 离散化数组：O(n)
 * - 树状数组：O(n)
 * - 总空间复杂度：O(n)
 *
 * 工程化考量：
 * 1. 边界条件处理：处理空数组、单个元素等情况
 * 2. 数值溢出处理：使用 long 类型避免整数溢出
 * 3. 离散化优化：使用 TreeSet 去重并排序
 * 4. 异常处理：检查输入参数合法性
 *
 * 算法技巧：
 * - 离散化：将大范围的数值映射到小范围的索引
 * - 树状数组：高效统计元素出现次数
 * - 逆序遍历：从右向左处理，便于统计右侧元素
 * - 二分查找：快速定位满足条件的边界
 *
 * 适用场景：
 * - 需要统计数组中满足特定条件的翻转对数量
 * - 数值范围较大，需要离散化处理
 * - 对时间复杂度要求较高的场景
 *
 * 测试用例：
 * 输入：nums = [1,3,2,3,1]
 * 输出：2
 * 解释：两个重要翻转对：(1,4) 和 (3,4)
 */
public class Code24_ReversePairs {

    /**
     * 计算重要翻转对的数量
     *
     * @param nums 输入数组
     * @return 重要翻转对的数量
     *
```

```
     * 算法步骤：
     * 1. 离散化处理数组元素
     * 2. 从右向左遍历数组
     * 3. 对于每个元素 nums[i]，需要找到满足 nums[i] > 2*nums[j] 的 nums[j] 的范围
     * 4. 使用树状数组统计该范围内的元素个数
     * 5. 更新树状数组，记录当前元素的出现
     */
    public int reversePairs(int[] nums) {
        if (nums == null || nums.length == 0) {
            return 0;
        }

        int n = nums.length;

        // 1. 离散化处理：收集所有需要离散化的值
        TreeSet<Long> set = new TreeSet<>();
        for (int num : nums) {
            set.add((long) num);
            set.add(2L * num);
        }

        // 构建离散化映射
        Map<Long, Integer> map = new HashMap<>();
        int idx = 1;
        for (long num : set) {
            map.put(num, idx++);
        }

        // 2. 使用树状数组统计
        FenwickTree tree = new FenwickTree(map.size());
        int count = 0;

        // 从右向左遍历
        for (int i = n - 1; i >= 0; i--) {
            long currentNum = nums[i];
            long target = currentNum - 1; // 我们需要找到小于等于 target 的值

            // 找到满足 nums[i] > 2*nums[j] 的最大 nums[j]
            // 即：nums[j] < nums[i] / 2.0
            long maxAllowed = (long) Math.floor((currentNum - 1) / 2.0);

            // 如果 maxAllowed 在离散化映射中不存在，需要找到最大的小于等于 maxAllowed 的值
            Long floorKey = set.floor(maxAllowed);
```

```
            if (floorKey != null) {
                int targetIdx = map.get(floorKey);
                count += tree.query(targetIdx);
            }

            // 更新树状数组，记录当前元素的出现
            int currentIdx = map.get(currentNum);
            tree.update(currentIdx, 1);
        }

        return count;
    }

    /**
     * 方法二：使用归并排序统计翻转对（备选方案）
     *
     * 解题思路：
     * 1. 使用归并排序的过程统计重要翻转对的数量
     * 2. 在合并两个有序数组之前，先统计满足条件的翻转对
     * 3. 这种方法同样具有 O(n log n) 的时间复杂度
     *
     * 时间复杂度：O(n log n)
     * 空间复杂度：O(n)
     */
    public int reversePairsMergeSort(int[] nums) {
        if (nums == null || nums.length == 0) {
            return 0;
        }

        return mergeSort(nums, 0, nums.length - 1);
    }

    private int mergeSort(int[] nums, int left, int right) {
        if (left >= right) {
            return 0;
        }

        int mid = left + (right - left) / 2;
        int count = mergeSort(nums, left, mid) + mergeSort(nums, mid + 1, right);

        // 统计满足条件的翻转对
        int j = mid + 1;
        for (int i = left; i <= mid; i++) {
```

```
        while (j <= right && (long) nums[i] > 2L * nums[j]) {
            j++;
        }
        count += j - (mid + 1);
    }

    // 合并两个有序数组
    merge(nums, left, mid, right);

    return count;
}

private void merge(int[] nums, int left, int mid, int right) {
    int[] temp = new int[right - left + 1];
    int i = left, j = mid + 1, k = 0;

    while (i <= mid && j <= right) {
        if (nums[i] <= nums[j]) {
            temp[k++] = nums[i++];
        } else {
            temp[k++] = nums[j++];
        }
    }

    while (i <= mid) {
        temp[k++] = nums[i++];
    }

    while (j <= right) {
        temp[k++] = nums[j++];
    }

    System.arraycopy(temp, 0, nums, left, temp.length);
}

/**
 * 树状数组（Fenwick Tree）实现
 * 用于高效统计元素出现次数
 */
private static class FenwickTree {
    private int[] tree;
    private int size;
```

```
/**
 * 构造函数
 *
 * @param size 树状数组大小
 */
public FenwickTree(int size) {
    this.size = size;
    this.tree = new int[size + 1];
}

/**
 * 更新操作：在指定位置增加一个值
 *
 * @param index 位置索引
 * @param delta 增加值
 */
public void update(int index, int delta) {
    while (index <= size) {
        tree[index] += delta;
        index += lowbit(index);
    }
}

/**
 * 查询前缀和：从 1 到 index 的和
 *
 * @param index 位置索引
 * @return 前缀和
 */
public int query(int index) {
    int sum = 0;
    while (index > 0) {
        sum += tree[index];
        index -= lowbit(index);
    }
    return sum;
}

/**
 * 计算 lowbit（最低位的 1）
 *
 * @param x 输入数字
 * @return lowbit 值
```

```
     */
    private int lowbit(int x) {
        return x & -x;
    }
}

/**
 * 测试方法：验证算法正确性
 *
 * 测试用例设计：
 * 1. 正常情况测试
 * 2. 边界情况测试
 * 3. 空数组测试
 * 4. 大数值测试
 */
public static void main(String[] args) {
    Code24_ReversePairs solution = new Code24_ReversePairs();

    // 测试用例 1：正常情况
    int[] nums1 = {1, 3, 2, 3, 1};
    int result1 = solution.reversePairs(nums1);
    System.out.println("测试用例 1 结果：" + result1 + " (期望：2)");

    // 测试用例 2：边界情况
    int[] nums2 = {2, 4, 3, 5, 1};
    int result2 = solution.reversePairs(nums2);
    System.out.println("测试用例 2 结果：" + result2 + " (期望：3)");

    // 测试用例 3：空数组
    int[] nums3 = {};
    int result3 = solution.reversePairs(nums3);
    System.out.println("测试用例 3 结果：" + result3 + " (期望：0)");

    // 测试用例 4：大数值
    int[] nums4 = {2147483647, 2147483647, 2147483647, 2147483647, 2147483647};
    int result4 = solution.reversePairs(nums4);
    System.out.println("测试用例 4 结果：" + result4 + " (期望：0)");

    // 测试用例 5：负数情况
    int[] nums5 = {-5, -5};
    int result5 = solution.reversePairs(nums5);
    System.out.println("测试用例 5 结果：" + result5 + " (期望：1)");
```

```
        System.out.println("所有测试用例执行完成！");
    }
}
```

==========

文件: Code24_ReversePairs.py

==========

```
"""
LeetCode 493. 翻转对 (Reverse Pairs)

题目描述：
给定一个数组 nums ，如果 i < j 且 nums[i] > 2*nums[j] 我们就将 (i,j) 称作一个重要翻转对。
你需要返回给定数组中的重要翻转对的数量。

解题思路：
使用树状数组（Fenwick Tree）+ 离散化来高效统计重要翻转对的数量。
核心思想：
1. 从右向左遍历数组
2. 对于每个元素 nums[i]，需要统计右侧已经遍历过的元素中满足 nums[i] > 2*nums[j] 的个数
3. 使用树状数组维护已经遍历过的元素的出现次数
4. 通过离散化处理大数值范围问题

时间复杂度分析：
- 离散化处理：O(n log n)
- 树状数组操作：O(n log n)
- 总时间复杂度：O(n log n)

空间复杂度分析：
- 离散化数组：O(n)
- 树状数组：O(n)
- 总空间复杂度：O(n)

工程化考量：
1. 边界条件处理：处理空数组、单个元素等情况
2. 数值溢出处理：使用 Python 的 int 类型自动处理大整数
3. 离散化优化：使用 set 去重并排序
4. 异常处理：检查输入参数合法性

算法技巧：
- 离散化：将大范围的数值映射到小范围的索引
- 树状数组：高效统计元素出现次数
- 逆序遍历：从右向左处理，便于统计右侧元素
```

- 二分查找：快速定位满足条件的边界

适用场景：
- 需要统计数组中满足特定条件的翻转对数量
- 数值范围较大，需要离散化处理
- 对时间复杂度要求较高的场景

测试用例：
输入：nums = [1,3,2,3,1]
输出：2
解释：两个重要翻转对：(1,4) 和 (3,4)
"""

```
from typing import List
import bisect



class FenwickTree:
    """
    树状数组（Fenwick Tree）实现
    用于高效统计元素出现次数
    """

    def __init__(self, size: int):
        """
        构造函数

        Args:
            size: 树状数组大小
        """
        self.size = size
        self.tree = [0] * (size + 1)

    def _lowbit(self, x: int) -> int:
        """
        计算 lowbit（最低位的 1）

        Args:
            x: 输入数字

        Returns:
            lowbit 值
        """
```

```
        return x & -x

    def update(self, index: int, delta: int):
        """
        更新操作：在指定位置增加一个值

        Args:
            index: 位置索引
            delta: 增加值
        """
        while index <= self.size:
            self.tree[index] += delta
            index += self._lowbit(index)

    def query(self, index: int) -> int:
        """
        查询前缀和：从 1 到 index 的和

        Args:
            index: 位置索引

        Returns:
            前缀和
        """
        total = 0
        while index > 0:
            total += self.tree[index]
            index -= self._lowbit(index)
        return total


class Solution:
    """
    翻转对统计解决方案类
    """

    def reversePairs(self, nums: List[int]) -> int:
        """
        计算重要翻转对的数量

        Args:
            nums: 输入数组
```

```
Returns:
    重要翻转对的数量

算法步骤：
1. 离散化处理数组元素
2. 从右向左遍历数组
3. 对于每个元素 nums[i]，需要找到满足 nums[i] > 2*nums[j] 的 nums[j] 的范围
4. 使用树状数组统计该范围内的元素个数
5. 更新树状数组，记录当前元素的出现
"""
if not nums:
    return 0

n = len(nums)

# 1. 离散化处理：收集所有需要离散化的值
value_set = set()
for num in nums:
    value_set.add(num)
    value_set.add(2 * num)

# 构建离散化映射
sorted_values = sorted(value_set)
value_map = {val: idx + 1 for idx, val in enumerate(sorted_values)}

# 2. 使用树状数组统计
tree = FenwickTree(len(sorted_values))
count = 0

# 从右向左遍历
for i in range(n - 1, -1, -1):
    current_num = nums[i]

    # 找到满足 nums[i] > 2*nums[j] 的最大 nums[j]
    # 即：nums[j] < nums[i] / 2.0
    max_allowed = (current_num - 1) // 2  # 使用整数除法避免浮点误差

    # 在有序列表中查找小于等于 max_allowed 的最大值的位置
    pos = bisect.bisect_right(sorted_values, max_allowed) - 1

    if pos >= 0:
        target_idx = pos + 1  # 树状数组索引从1开始
        count += tree.query(target_idx)
```

```
            # 更新树状数组，记录当前元素的出现
            current_idx = value_map[current_num]
            tree.update(current_idx, 1)

        return count

    def reversePairsMergeSort(self, nums: List[int]) -> int:
        """
        方法二：使用归并排序统计翻转对（备选方案）

        解题思路：
        1. 使用归并排序的过程统计重要翻转对的数量
        2. 在合并两个有序数组之前，先统计满足条件的翻转对
        3. 这种方法同样具有 O(n log n) 的时间复杂度

        时间复杂度：O(n log n)
        空间复杂度：O(n)
        """
        if not nums:
            return 0

        return self._mergeSort(nums, 0, len(nums) - 1)

    def _mergeSort(self, nums: List[int], left: int, right: int) -> int:
        """
        归并排序递归函数

        Args:
            nums: 原始数组
            left: 起始位置
            right: 结束位置

        Returns:
            翻转对数量
        """
        if left >= right:
            return 0

        mid = left + (right - left) // 2
        count = self._mergeSort(nums, left, mid) + self._mergeSort(nums, mid + 1, right)

        # 统计满足条件的翻转对
```

```
    j = mid + 1
    for i in range(left, mid + 1):
        while j <= right and nums[i] > 2 * nums[j]:
            j += 1
        count += j - (mid + 1)

    # 合并两个有序数组
    self._merge(nums, left, mid, right)

    return count

def _merge(self, nums: List[int], left: int, mid: int, right: int):
    """
    合并两个有序数组

    Args:
        nums: 原始数组
        left: 起始位置
        mid: 中间位置
        right: 结束位置
    """
    temp = []
    i, j = left, mid + 1

    while i <= mid and j <= right:
        if nums[i] <= nums[j]:
            temp.append(nums[i])
            i += 1
        else:
            temp.append(nums[j])
            j += 1

    while i <= mid:
        temp.append(nums[i])
        i += 1

    while j <= right:
        temp.append(nums[j])
        j += 1

    # 将临时数组复制回原数组
    for idx in range(len(temp)):
        nums[left + idx] = temp[idx]
```

```
def test_reverse_pairs():
    """
    测试函数：验证算法正确性

    测试用例设计：
    1. 正常情况测试
    2. 边界情况测试
    3. 空数组测试
    4. 大数值测试
    """
    solution = Solution()

    # 测试用例 1：正常情况
    nums1 = [1, 3, 2, 3, 1]
    result1 = solution.reversePairs(nums1)
    print(f"测试用例 1 结果：{result1} (期望：2)")

    # 测试用例 2：边界情况
    nums2 = [2, 4, 3, 5, 1]
    result2 = solution.reversePairs(nums2)
    print(f"测试用例 2 结果：{result2} (期望：3)")

    # 测试用例 3：空数组
    nums3 = []
    result3 = solution.reversePairs(nums3)
    print(f"测试用例 3 结果：{result3} (期望：0)")

    # 测试用例 4：大数值
    nums4 = [2147483647, 2147483647, 2147483647, 2147483647, 2147483647]
    result4 = solution.reversePairs(nums4)
    print(f"测试用例 4 结果：{result4} (期望：0)")

    # 测试用例 5：负数情况
    nums5 = [-5, -5]
    result5 = solution.reversePairs(nums5)
    print(f"测试用例 5 结果：{result5} (期望：1)")

    print("所有测试用例执行完成！")


if __name__ == "__main__":
```

```
    test_reverse_pairs()

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

文件: SimpleTest.java
================================================
public class SimpleTest {
    public static void main(String[] args) {
        System.out.println("SimpleTest is working!");
    }
}

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

文件: TestReversePairs.java
================================================
import java.util.*;

public class TestReversePairs {
    public static void main(String[] args) {
        System.out.println("TestReversePairs is working!");
    }
}

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