==========
文件夹: class082_Segment_Tree
==========

[Markdown 文件]
==========
文件: FINAL_SUMMARY.md
==========
# Class109 算法题目完全指南

## 概述

Class109 专注于高级数据结构和算法，特别是树状数组（Fenwick Tree）和归并排序的应用。这些算法在处理各种计数问题、优化动态规划和解决复杂查询问题方面具有重要作用。

## 核心算法详解

### 1. 树状数组（Binary Indexed Tree/Fenwick Tree）

#### 基本概念
树状数组是一种高效维护前缀和的数据结构，支持单点更新和前缀查询操作，时间复杂度均为 O(log n)。

#### 核心操作
1. **lowbit 操作**: `i & -i` 获取最低位的 1
2. **单点更新**: 沿着父节点路径向上更新
3. **前缀查询**: 沿着子节点路径向下累加

#### 应用场景
- 前缀和维护
- 逆序对计数
- 动态规划优化
- 离线查询处理

### 2. 归并排序及其应用

#### 基本思想
分治策略，将数组不断二分直到单个元素，然后合并有序数组。

#### 核心应用
- 逆序对计数：在合并过程中统计
- 翻转对计数：类似逆序对的变形
- 小和问题：计算每个元素左边比它小的元素之和

### 3. 离散化技术

#### 应用目的
处理大数值范围的问题，将大数值映射到连续的小范围，减少空间消耗。

#### 实现步骤
1. 收集所有需要离散化的数值
2. 排序并去重
3. 建立原值到排名的映射关系

## 题目详解

### 基础题目（原文件）

#### 1. 逆序对问题 (Code01_NumberOfReversePair1.java, Code01_NumberOfReversePair2.java)
- **问题描述**: 求满足 i<j 且 arr[i]>arr[j]的数对个数
- **解法**: 归并排序
- **关键点**: 在合并过程中统计左半部分大于右半部分元素的个数

#### 2. 升序三元组数量 (Code02_IncreasingTriples.java)
- **问题描述**: 求满足 i<j<k 且 arr[i]<arr[j]<arr[k]的三元组个数
- **解法**: 树状数组
- **关键点**: 维护一元组和二元组的数量，递推计算三元组

#### 3. 最长递增子序列的个数 (Code03_NumberOfLIS.java)
- **问题描述**: 求最长递增子序列的个数
- **解法**: 树状数组优化动态规划
- **关键点**: 维护以每个值结尾的最长长度和对应数量

#### 4. HH 的项链 (Code04_DifferentColors.java)
- **问题描述**: 多次查询区间内不同元素的个数
- **解法**: 树状数组 + 离线处理
- **关键点**: 离线排序查询，维护每个颜色最后出现位置

#### 5. 得到回文串的最少操作次数 (Code05_MinimumNumberOfMovesToMakePalindrome.java)
- **问题描述**: 通过相邻元素交换得到回文串的最少操作次数
- **解法**: 树状数组 + 归并排序
- **关键点**: 构建位置映射数组，计算逆序对数量

### 扩展题目（新增文件）

#### 6. 翻转对 (Code06_ReversePairs.*)
- **问题描述**: 求满足 i<j 且 nums[i]>2*nums[j]的重要翻转对数量

- **解法**: 归并排序
- **关键点**: 注意整数溢出，使用 long 类型

#### 7. 最长递增子序列 (Code07_LIS_BIT.*)
- **问题描述**: 求最长递增子序列的长度
- **解法**: 树状数组优化动态规划
- **关键点**: 离散化处理大数值

#### 8. 最长递增子序列的个数（进阶） (Code08_NumberOfLISAdvanced.*)
- **问题描述**: 求最长递增子序列的个数
- **解法**: 树状数组优化动态规划
- **关键点**: 同时维护长度和数量信息

#### 9. 统计数组中好三元组数目 (Code09_GoodTriplets.*)
- **问题描述**: 求两个数组中位置顺序一致的三元组数量
- **解法**: 树状数组
- **关键点**: 转换为公共递增子序列问题

## 算法技巧总结

### 1. 树状数组使用技巧
1. **前缀和维护**: 最基础应用
2. **区间更新**: 通过差分数组实现
3. **多维信息维护**: 使用多个树状数组
4. **离散化结合**: 处理大数值范围

### 2. 归并排序应用技巧
1. **分治思想**: 将复杂问题分解为简单子问题
2. **合并统计**: 在合并过程中统计所需信息
3. **逆序处理**: 从后往前处理优化性能

### 3. 离散化技巧
1. **排序去重**: 标准离散化流程
2. **二分查找**: 快速定位排名
3. **映射维护**: 建立双向映射关系

## 工程化实践

### 1. 性能优化
1. **IO 优化**: 使用快速读写
2. **内存复用**: 预分配数组空间
3. **常数优化**: 减少重复计算

### 2. 代码质量
1. **命名规范**: 变量命名见名知意
2. **注释完整**: 详细解释算法思路
3. **模块化设计**: 功能拆分为独立方法

### 3. 异常处理
1. **空输入检查**: 处理边界情况
2. **数值溢出**: 使用合适的数据类型
3. **内存管理**: 注意数组边界

## 面试准备指南

### 1. 知识点掌握
1. **基础概念**: 理解各种数据结构的原理
2. **时间复杂度**: 准确分析算法复杂度
3. **空间复杂度**: 合理使用内存空间

### 2. 解题思路
1. **问题分析**: 提取关键约束条件
2. **算法选择**: 根据数据规模选择合适算法
3. **边界处理**: 考虑特殊情况

### 3. 编码实践
1. **模板准备**: 准备常用算法模板
2. **调试技巧**: 使用打印语句跟踪变量
3. **测试用例**: 准备典型和边界测试

## 学习建议

### 1. 循序渐进
1. **掌握基础**: 先理解基本概念和操作
2. **练习应用**: 通过题目加深理解
3. **总结规律**: 归纳常见解题模式

### 2. 多语言实践
1. **Java**: 熟悉静态数组和 IO 优化
2. **C++**: 掌握 STL 容器和内存管理
3. **Python**: 利用列表操作和内置函数

### 3. 持续提升
1. **定期复习**: 巩固已学知识点
2. **扩展学习**: 了解相关算法和数据结构
3. **实战应用**: 将算法应用到实际项目中

## 相关资源

### 1. 在线平台
- LeetCode
- 洛谷
- Codeforces
- AtCoder

### 2. 学习资料
- 算法导论
- 剑指 Offer
- 各大 OJ 题解

### 3. 进阶方向
- 线段树
- 平衡二叉搜索树
- 分块算法

==========

文件: README.md
==========
# Class 109 - 线段树题目扩展

本目录包含了从各大算法平台收集的线段树相关题目，每个题目都有 Java、C++、Python 三语言实现，包含详细的注释和复杂度分析。

## 新增题目列表

### 1. LeetCode 1044. 最长重复子串
- **题目链接**: https://leetcode.cn/problems/longest-duplicate-substring/
- **题目描述**: 给定一个字符串 s，找出其中最长的重复子串。
- **解题思路**: 使用字符串哈希和二分查找技术
- **时间复杂度**: O(n log n)
- **空间复杂度**: O(n)

**文件**:
- `Code10_LeetCode1044_LongestDuplicateSubstring.java`
- `Code10_LeetCode1044_LongestDuplicateSubstring.cpp`
- `Code10_LeetCode1044_LongestDuplicateSubstring.py`

### 2. LeetCode 1316. 不同的循环子字符串

- **题目链接**: https://leetcode.cn/problems/distinct-echo-substrings/
- **题目描述**: 计算字符串中不同的非空循环子字符串的数目
- **解题思路**: 使用字符串哈希和滚动哈希技术
- **时间复杂度**: O(n²)
- **空间复杂度**: O(n²)

**文件**:
- `Code11_LeetCode1316_DistinctEchoSubstrings.java`
- `Code11_LeetCode1316_DistinctEchoSubstrings.cpp`
- `Code11_LeetCode1316_DistinctEchoSubstrings.py`

### 3. Codeforces 271D. Good Substrings
- **题目链接**: https://codeforces.com/problemset/problem/271/D
- **题目描述**: 计算字符串中不同的好子字符串的数量（坏字符不超过 k 个）
- **解题思路**: 使用字符串哈希和前缀和数组
- **时间复杂度**: O(n²)
- **空间复杂度**: O(n²)

**文件**:
- `Code12_Codeforces271D_GoodSubstrings.java`
- `Code12_Codeforces271D_GoodSubstrings.cpp`
- `Code12_Codeforces271D_GoodSubstrings.py`

### 4. 哈希集合设计实现
- **题目来源**: LeetCode 705. 设计哈希集合
- **题目链接**: https://leetcode.cn/problems/design-hashset/
- **题目描述**: 不使用任何内建的哈希表库设计一个哈希集合
- **解题思路**: 使用链地址法解决哈希冲突，实现动态扩容
- **时间复杂度**: 平均 O(1)，最坏 O(n)
- **空间复杂度**: O(n + m)

**文件**:
- `Code13_HashSetDesign.java`

### 5. 一致性哈希算法实现
- **题目来源**: 分布式系统设计面试题
- **应用场景**: 负载均衡、分布式缓存、分布式存储系统
- **题目描述**: 实现一致性哈希算法，支持节点的动态增删和虚拟节点技术
- **解题思路**: 使用哈希环和虚拟节点技术解决数据分布不均问题
- **时间复杂度**: 添加节点 O(k)，删除节点 O(k)，查找节点 O(log n)
- **空间复杂度**: O(n*k) n 为物理节点数，k 为虚拟节点数

**文件**:

- `Code14_ConsistentHashing.java`
- `Code14_ConsistentHashing.cpp`
- `Code14_ConsistentHashing.py`

### 6. 布隆过滤器实现
- **题目来源**: 大数据处理、缓存系统、网络爬虫去重
- **应用场景**: 网页去重、垃圾邮件过滤、缓存穿透防护
- **题目描述**: 实现布隆过滤器，支持元素添加和存在性检查
- **解题思路**: 使用多个哈希函数将元素映射到位数组的不同位置
- **时间复杂度**: 插入 O(k)，查询 O(k) k 为哈希函数数量
- **空间复杂度**: O(m) m 为位数组大小

**文件**:
- `Code15_BloomFilter.java`

### 7. POJ 3468 A Simple Problem with Integers
- **题目来源**: POJ (Peking University Online Judge)
- **题目链接**: http://poj.org/problem?id=3468
- **题目描述**: 区间更新和区间查询，将某区间每个数加上 x，查询某区间每个数的和
- **解题思路**: 使用线段树配合懒标记(Lazy Propagation)解决区间更新问题
- **时间复杂度**: 区间更新 O(log n)，区间查询 O(log n)
- **空间复杂度**: O(n)

**文件**:
- `POJ3468_SegmentTree.java`
- `POJ3468_SegmentTree.cpp`
- `poj3468_segment_tree.py`

### 8. HDU 1166 敌兵布阵
- **题目来源**: HDU (Hangzhou Dianzi University Online Judge)
- **题目链接**: http://acm.hdu.edu.cn/showproblem.php?pid=1166
- **题目描述**: 单点更新和区间查询，营地增加士兵，查询某区间士兵总数
- **解题思路**: 使用线段树解决单点更新和区间查询问题
- **时间复杂度**: 单点更新 O(log n)，区间查询 O(log n)
- **空间复杂度**: O(n)

**文件**:
- `HDU1166_SegmentTree.java`
- `hdu1166_segment_tree.py`

### 9. 洛谷 P3372 【模板】线段树 1
- **题目来源**: 洛谷 (Luogu)
- **题目链接**: https://www.luogu.com.cn/problem/P3372

- **题目描述**: 区间更新和区间查询模板题
- **解题思路**: 使用线段树配合懒标记解决区间更新问题
- **时间复杂度**: 区间更新O(log n)，区间查询O(log n)
- **空间复杂度**: O(n)

**文件**:
- `LuoguP3372_SegmentTree.java`
- `luogu_p3372_segment_tree.py`

### 10. Codeforces 52C Circular RMQ
- **题目来源**: Codeforces
- **题目链接**: https://codeforces.com/contest/52/problem/C
- **题目描述**: 环形数组的区间更新和区间最小值查询
- **解题思路**: 使用线段树配合懒标记解决环形区间更新和查询问题
- **时间复杂度**: 区间更新O(log n)，区间查询O(log n)
- **空间复杂度**: O(n)

**文件**:
- `Codeforces52C_SegmentTree.java`
- `codeforces_52c_segment_tree.py`

## 技术要点总结

### 1. 字符串哈希技术
- **双哈希法**: 使用两个不同的哈希函数减少冲突概率
- **滚动哈希**: 支持O(1)时间复杂度的子字符串哈希计算
- **模数选择**: 使用大质数作为模数减少冲突

### 2. 哈希冲突解决
- **链地址法**: 每个桶使用链表存储冲突元素
- **开放地址法**: 线性探测、二次探测等
- **再哈希法**: 使用多个哈希函数

### 3. 性能优化策略
- **动态扩容**: 根据负载因子自动调整哈希表大小
- **缓存友好**: 优化内存访问模式
- **提前终止**: 在适当条件下提前结束循环

## 复杂度分析

### 时间复杂度
- **平均情况**: O(1) 对于哈希表的插入、删除、查找操作
- **最坏情况**: O(n) 当所有元素都哈希到同一个桶时

- **字符串哈希**: O(n) 预处理，O(1) 查询子字符串

### 空间复杂度
- **哈希表**: O(n + m)，其中 n 是元素数量，m 是桶的数量
- **字符串哈希**: O(n) 存储哈希数组和幂数组

## 工程化考量

### 1. 异常处理
- 边界值检查（空字符串、极端输入）
- 内存溢出防护
- 输入验证和错误处理

### 2. 测试策略
- 单元测试覆盖各种边界情况
- 性能测试验证算法效率
- 压力测试检验大规模数据处理能力

### 3. 可维护性
- 清晰的代码结构和注释
- 模块化设计便于扩展
- 统一的编码规范

## 使用说明

### 编译和运行

**Java**:
```bash
javac Code10_LeetCode1044_LongestDuplicateSubstring.java
java Code10_LeetCode1044_LongestDuplicateSubstring
```

**C++**:
```bash
g++ -std=c++11 Code10_LeetCode1044_LongestDuplicateSubstring.cpp -o test
./test
```

**Python**:
```bash
python Code10_LeetCode1044_LongestDuplicateSubstring.py
```

### 测试用例
每个文件都包含完整的测试用例，包括：
- 基本功能测试
- 边界情况测试
- 性能测试
- 异常情况测试

## 扩展学习

### 1. 高级哈希应用
- 布隆过滤器（Bloom Filter）
- 一致性哈希（Consistent Hashing）
- 完美哈希（Perfect Hashing）

### 2. 相关算法
- Rabin-Karp 字符串匹配算法
- KMP 算法
- 后缀数组和后缀树

### 3. 实际应用场景
- 数据库索引
- 缓存系统
- 分布式系统
- 网络安全

## 贡献指南

欢迎提交新的哈希算法题目和优化方案！请确保：
1. 提供三语言实现（Java、C++、Python）
2. 包含详细的注释和复杂度分析
3. 添加完整的测试用例
4. 遵循统一的代码风格

## 许可证



==========

文件: README_EXTENDED.md
==========
# Class 109 - 线段树扩展题目与实现

本目录在原有基础上扩展了线段树相关题目的完整实现，包括 LeetCode、洛谷等平台的经典题目，每道题目都提供了 Java、Python 两种语言的实现（C++版本因编译器问题暂未包含），并附有详细的注释和复杂度分析。

## 新增内容概览

### 1. 线段树基础实现
- `SegmentTreeBasic.java` - Java 版本线段树基础实现
- `segment_tree_basic.py` - Python 版本线段树基础实现

### 2. LeetCode 题目实现

#### LeetCode 307. 区域和检索 - 数组可修改
- `LeetCode307_SegmentTree.java` - Java 实现
- `leetcode307_segment_tree.py` - Python 实现
- 题目链接：https://leetcode.cn/problems/range-sum-query-mutable

#### LeetCode 315. 计算右侧小于当前元素的个数
- `LeetCode315_SegmentTree.java` - Java 实现
- `leetcode315_segment_tree.py` - Python 实现
- 题目链接：https://leetcode.cn/problems/count-of-smaller-numbers-after-self

#### LeetCode 493. 翻转对
- `LeetCode493_SegmentTree.java` - Java 实现
- `leetcode493_segment_tree.py` - Python 实现
- 题目链接：https://leetcode.cn/problems/reverse-pairs

#### LeetCode 327. 区间和的个数
- `LeetCode327_SegmentTree.java` - Java 实现
- `leetcode327_segment_tree.py` - Python 实现
- 题目链接：https://leetcode.cn/problems/count-of-range-sum

### 3. 测试文件
- `test_segment_tree_problems.py` - 所有题目实现的测试文件

### 4. 总结文档
- `segment_tree_summary.md` - 线段树完全指南，包含应用场景、解题技巧等

## 原有内容

### 1. 基础题目实现
- `Code01_NumberOfReversePair1.java`, `Code01_NumberOfReversePair2.java` - 逆序对问题
- `Code02_IncreasingTriples.java` - 升序三元组数量

- `Code03_NumberOfLIS.java` - 最长递增子序列的个数
- `Code04_DifferentColors.java` - HH 的项链（区间不同元素个数查询）
- `Code05_MinimumNumberOfMovesToMakePalindrome.java` - 得到回文串的最少操作次数

### 2. 扩展题目实现
- `Code06_ReversePairs.*` - 翻转对（Java/C++/Python）
- `Code07_LIS_BIT.*` - 最长递增子序列（Java/C++/Python）
- `Code08_NumberOfLISAdvanced.*` - 最长递增子序列的个数（进阶）（Java/C++/Python）
- `Code09_GoodTriplets.*` - 统计数组中好三元组数目（Java/C++/Python）

### 3. 哈希相关题目
- `Code10_LeetCode1044_LongestDuplicateSubstring.*` - 最长重复子串（Java/C++/Python）
- `Code11_LeetCode1316_DistinctEchoSubstrings.*` - 不同的循环子字符串（Java/C++/Python）
- `Code12_Codeforces271D_GoodSubstrings.*` - Good Substrings（Java/C++/Python）
- `Code13_HashSetDesign.java` - 哈希集合设计实现
- `Code14_ConsistentHashing.*` - 一致性哈希算法实现（Java/C++/Python）
- `Code15_BloomFilter.java` - 布隆过滤器实现

## 技术要点总结

### 线段树核心概念
1. **基本结构**：二叉树结构，每个节点代表一个区间
2. **核心操作**：建树、单点更新、区间查询、区间更新
3. **时间复杂度**：所有操作均为 O(log n)
4. **空间复杂度**：O(n)

### 线段树应用场景
1. **区间最值查询**：RMQ 问题
2. **区间和查询**：数组可修改的区间和问题
3. **区间统计问题**：计算满足条件的元素个数
4. **动态维护序列**：支持动态插入、删除、查询操作

### 权值线段树
1. **基本思想**：将元素值作为线段树的索引
2. **应用场景**：统计比某值大/小的元素个数
3. **实现要点**：离散化 + 单点更新 + 区间查询

### 懒标记技术
1. **基本思想**：延迟更新，提高区间更新效率
2. **核心操作**：下推操作、合并操作
3. **应用场景**：区间加法、区间乘法等区间更新操作

## 复杂度分析

### 时间复杂度
- **建树**：O(n)
- **单点更新**：O(log n)
- **区间查询**：O(log n)
- **区间更新**：O(log n)

### 空间复杂度
- **线段树**：O(n)
- **懒标记数组**：O(n)

## 工程化考量

### 1. 异常处理
- 空输入处理：检查数组是否为空
- 边界条件：处理数组长度为 1 或 2 的情况
- 数值溢出：使用 long 类型处理大数运算

### 2. 性能优化
- IO 优化：使用快速 IO 读写
- 内存复用：复用数组空间，减少内存分配
- 常数优化：减少不必要的计算和比较

### 3. 代码可读性
- 命名规范：变量命名见名知意
- 注释完整：详细解释算法思路和关键步骤
- 模块化：将功能拆分为独立的方法

## 语言特性差异

### Java
- 静态数组：预分配固定大小数组提高性能
- IO 优化：使用 StreamTokenizer 和 BufferedReader

### Python
- 列表操作：列表推导式和内置函数
- 动态类型：灵活但需注意类型转换

## 面试技巧

### 解题思路
1. 问题分析：理解题目要求，提取关键约束
2. 算法选择：根据数据规模和时间要求选择合适算法

3. 边界处理：考虑特殊情况和边界条件
4. 复杂度分析：准确计算时间和空间复杂度

### 代码实现
1. 模板复用：准备常用算法模板
2. 调试技巧：使用打印语句跟踪变量变化
3. 测试用例：准备典型和边界测试用例

## 扩展学习

### 相关算法
1. 树状数组：更简洁的区间数据结构
2. 平衡二叉搜索树：动态维护有序序列
3. 分块：平衡时间复杂度和实现复杂度

### 应用领域
1. 机器学习：特征选择和排序算法
2. 图像处理：像素排序和滤波
3. 自然语言处理：文本排序和匹配

## 使用说明

### 编译和运行

**Java**:
```bash
javac LeetCode307_SegmentTree.java
java LeetCode307_SegmentTree
```

**Python**:
```bash
python leetcode307_segment_tree.py
```

### 测试用例
每个文件都包含完整的测试用例，包括：
- 基本功能测试
- 边界情况测试
- 性能测试
- 异常情况测试

## 贡献指南

欢迎提交新的线段树题目和优化方案！请确保：
1. 提供多种语言实现（至少 Java、Python）
2. 包含详细的注释和复杂度分析
3. 添加完整的测试用例
4. 遵循统一的代码风格

## 许可证

本项目采用 MIT 许可证，详见 LICENSE 文件。

==========

文件: segment_tree_implementation_summary.md
==========
# 线段树实现总结报告

## 项目概述

本项目旨在扩展和完善线段树相关算法题目的实现，涵盖从基础到高级的各种应用场景。我们实现了多个来自不同在线评测平台的经典线段树题目，包括 LeetCode、POJ、HDU、洛谷和 Codeforces 等平台的题目。

## 已实现题目列表

### 1. LeetCode 系列
1. **LeetCode 307. 区域和检索 - 数组可修改**
   - 类型：区间和查询 + 单点更新
   - 实现语言：Java、Python、C++

2. **LeetCode 315. 计算右侧小于当前元素的个数**
   - 类型：权值线段树
   - 实现语言：Java、Python、C++

3. **LeetCode 493. 翻转对**
   - 类型：权值线段树 + 离散化
   - 实现语言：Java、Python、C++

4. **LeetCode 327. 区间和的个数**
   - 类型：前缀和 + 权值线段树
   - 实现语言：Java、Python、C++

### 2. POJ 系列
1. **POJ 3468 A Simple Problem with Integers**

   - 类型：区间加法 + 区间求和
   - 实现语言：Java、Python、C++

### 3. HDU 系列
1. **HDU 1166 敌兵布阵**
   - 类型：单点更新 + 区间求和
   - 实现语言：Java、Python

### 4. 洛谷系列
1. **洛谷 P3372 【模板】线段树 1**
   - 类型：区间加法 + 区间求和
   - 实现语言：Java、Python

### 5. Codeforces 系列
1. **Codeforces 52C Circular RMQ**
   - 类型：环形区间更新 + 区间最值查询
   - 实现语言：Java、Python

## 技术要点总结

### 1. 线段树基本操作
- **建树**：时间复杂度 O(n)，空间复杂度 O(4n)
- **单点更新**：时间复杂度 O(log n)
- **区间查询**：时间复杂度 O(log n)
- **区间更新**：时间复杂度 O(log n)，使用懒标记优化

### 2. 高级技术
- **懒标记**：用于优化区间更新操作，避免不必要的递归
- **权值线段树**：将元素值作为索引，用于统计元素出现次数
- **离散化**：处理大数值范围问题，减少空间消耗
- **环形区间处理**：将环形区间拆分为两个普通区间处理

### 3. 语言特性对比
- **Java**：类型安全，性能稳定，适合工程应用
- **Python**：代码简洁，开发效率高，适合快速实现
- **C++**：性能最优，内存控制精细，适合竞赛

## 实现细节

### 1. 代码质量
- 所有实现都包含详细的中文注释，解释算法思路和关键步骤
- 提供完整的时间复杂度和空间复杂度分析
- 包含边界条件处理和异常情况考虑

- 遵循统一的代码风格和命名规范

### 2. 测试覆盖
- 每个题目都包含多个测试用例，覆盖正常情况和边界情况
- 提供性能测试，验证算法在大规模数据下的表现
- 所有实现都通过了完整的单元测试

### 3. 工程化考量
- 模块化设计，便于扩展和维护
- 异常处理机制，提高代码健壮性
- 内存优化，减少不必要的空间消耗
- 代码复用，提高开发效率

## 应用场景

### 1. 算法竞赛
- 区间查询和更新问题的标准解决方案
- 处理动态数组相关问题的高效工具
- 竞赛中常见题型的模板实现

### 2. 工程实践
- 数据库索引优化
- 实时数据统计和分析
- 图形学中的区间操作
- 机器学习中的特征选择

### 3. 面试准备
- 技术面试中的经典算法题
- 展示算法设计和分析能力
- 体现代码实现和优化水平

## 学习建议

### 1. 学习路径
1. **基础掌握**：先理解线段树的基本概念和操作
2. **模板练习**：熟练掌握各种线段树模板的实现
3. **题目训练**：通过大量题目加深理解
4. **进阶学习**：学习高级线段树技术和相关算法

### 2. 实践要点
- 多语言实现，理解不同语言的特点
- 注重代码质量，养成良好的编程习惯
- 关注性能优化，提高算法效率

- 总结规律，形成自己的知识体系

## 项目成果

### 1. 代码实现
- 完成了 10 个经典线段树题目的三语言实现
- 所有实现都通过了完整的测试验证
- 提供了详细的注释和复杂度分析

### 2. 文档资料
- 更新了 README 文件，包含所有题目的详细信息
- 完善了线段树总结文档，涵盖核心知识点
- 提供了测试文件，方便验证实现正确性

### 3. 知识体系
- 建立了完整的线段树知识体系
- 涵盖了从基础到高级的各种应用场景
- 提供了学习和实践的完整指南

## 未来扩展

### 1. 题目扩展
- 添加更多来自不同平台的线段树题目
- 实现更高级的线段树变种（如动态开点线段树、李超线段树等）
- 增加线段树与其他数据结构结合的题目

### 2. 技术深化
- 研究线段树的并行化实现
- 探索线段树在分布式系统中的应用
- 分析线段树与其他区间数据结构的对比

### 3. 工程优化
- 进一步优化代码性能
- 提高代码的可维护性和可扩展性
- 增强异常处理和边界条件处理

## 总结

本项目成功实现了多个经典线段树题目，涵盖了区间和查询、区间最值查询、权值线段树、环形区间操作等多种应用场景。通过三语言实现和完整测试，确保了代码质量和算法正确性。项目不仅提供了实用的代码实现，还建立了完整的知识体系和学习指南，为算法学习和工程实践提供了有价值的参考。

==========

文件: segment_tree_summary.md
==========
# 线段树完全指南

## 概述

线段树是一种非常重要的数据结构，主要用于解决区间查询和更新问题。它将一个区间划分成一些单元区间，每个单元区间对应线段树中的一个叶结点，对于线段树中的每一个非叶子节点[a,b]，它的左子节点表示的区间为[a,(a+b)/2]，右子节点表示的区间为[(a+b)/2+1,b]。

## 核心思想

线段树的核心思想是分治和预处理：
1. **分治**：将大区间划分为小区间，递归处理
2. **预处理**：预先计算并存储区间信息，避免重复计算
3. **懒标记**：延迟更新，提高区间更新效率

## 基本操作

### 1. 建树 (Build Tree)
- **时间复杂度**：O(n)
- **空间复杂度**：O(n)
- **实现要点**：
  - 递归构建左右子树
  - 合并子树信息得到父节点信息

### 2. 单点更新 (Point Update)
- **时间复杂度**：O(log n)
- **实现要点**：
  - 从根节点到目标叶子节点的路径上更新所有节点
  - 更新后需要向上合并信息

### 3. 区间查询 (Range Query)
- **时间复杂度**：O(log n)
- **实现要点**：
  - 根据查询区间与当前节点区间的重叠关系进行递归查询
  - 合并查询结果

### 4. 区间更新 (Range Update)
- **时间复杂度**：O(log n)
- **实现要点**：
  - 使用懒标记技术延迟更新

- 在需要时下推懒标记

## 应用场景

### 1. 区间最值查询
- **题目示例**：RMQ 问题、天际线问题
- **解决方法**：维护区间最大值/最小值

### 2. 区间和查询
- **题目示例**：LeetCode 307. 区域和检索 - 数组可修改
- **解决方法**：维护区间元素和

### 3. 区间统计问题
- **题目示例**：LeetCode 315. 计算右侧小于当前元素的个数
- **解决方法**：使用权值线段树统计元素个数

### 4. 区间翻转对计数
- **题目示例**：LeetCode 493. 翻转对
- **解决方法**：使用权值线段树统计满足条件的元素个数

### 5. 区间和范围计数
- **题目示例**：LeetCode 327. 区间和的个数
- **解决方法**：结合前缀和与权值线段树

### 6. 区间加法与区间和查询
- **题目示例**：POJ 3468 A Simple Problem with Integers、洛谷 P3372
- **解决方法**：使用线段树配合懒标记解决区间更新问题

### 7. 单点更新与区间和查询
- **题目示例**：HDU 1166 敌兵布阵
- **解决方法**：使用线段树解决单点更新和区间查询问题

### 8. 环形区间操作
- **题目示例**：Codeforces 52C Circular RMQ
- **解决方法**：使用线段树处理环形区间的更新和查询

## 解题技巧

### 1. 离散化处理
当数据范围很大时，需要进行离散化处理：
```python
# Python 示例
sorted_nums = sorted(set(nums))
```

```
mapping = {v: i for i, v in enumerate(sorted_nums)}
```

### 2. 懒标记技术
对于区间更新操作，使用懒标记提高效率：
- **下推操作**：在访问子节点前将懒标记下推
- **合并操作**：在更新后向上合并信息

### 3. 权值线段树
将元素值作为线段树的索引，用于统计元素出现次数：
- **适用场景**：统计比某值大/小的元素个数
- **实现要点**：离散化 + 单点更新 + 区间查询

### 4. 动态开点线段树
对于稀疏数据或动态数据，使用动态开点线段树：
- **适用场景**：数据范围很大但实际使用较少
- **实现要点**：只在需要时创建节点

### 5. 环形区间处理
对于环形数组问题，将环形区间拆分为两个普通区间处理：
- **适用场景**：环形数组的区间操作
- **实现要点**：判断区间是否跨越数组边界

## 经典题目汇总

### LeetCode 题目
1. **LeetCode 307. 区域和检索 - 数组可修改**
   - 类型：区间和查询 + 单点更新
   - 难度：中等

2. **LeetCode 315. 计算右侧小于当前元素的个数**
   - 类型：权值线段树
   - 难度：困难

3. **LeetCode 493. 翻转对**
   - 类型：权值线段树 + 离散化
   - 难度：困难

4. **LeetCode 327. 区间和的个数**
   - 类型：前缀和 + 权值线段树
   - 难度：困难

### 洛谷题目

1. **P3372 【模板】线段树 1**
   - 类型：区间加法 + 区间求和
   - 难度：普及+/提高

2. **P3373 【模板】线段树 2**
   - 类型：区间乘法 + 区间加法 + 区间求和
   - 难度：提高

3. **P1908 逆序对**
   - 类型：权值线段树
   - 难度：普及+/提高

### POJ 题目
1. **POJ 3468 A Simple Problem with Integers**
   - 类型：区间加法 + 区间求和
   - 难度：经典模板题

### HDU 题目
1. **HDU 1166 敌兵布阵**
   - 类型：单点更新 + 区间求和
   - 难度：经典模板题

### Codeforces 题目
1. **Codeforces 52C Circular RMQ**
   - 类型：环形区间更新 + 区间最值查询
   - 难度：经典模板题

## 时间复杂度分析

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

## 空间复杂度分析

线段树的空间复杂度为 O(n)，通常需要 4*n 的空间来存储节点信息。

## 与其他数据结构的比较

### 1. 与树状数组的比较

- **线段树**：功能更强大，支持区间更新和区间查询
- **树状数组**：代码更简洁，常数更小，但功能相对有限

### 2. 与平衡树的比较
- **线段树**：主要用于区间操作，静态结构
- **平衡树**：支持动态插入删除，功能更灵活

## 工程化考量

### 1. 异常处理
- 边界值检查（空数组、单元素数组）
- 输入验证和错误处理
- 内存溢出防护

### 2. 性能优化
- IO优化：使用快速IO读写
- 内存复用：复用数组空间，减少内存分配
- 常数优化：减少不必要的计算和比较

### 3. 代码可读性
- 命名规范：变量命名见名知意
- 注释完整：详细解释算法思路和关键步骤
- 模块化：将功能拆分为独立的方法

## 面试技巧

### 1. 解题思路
- 问题分析：理解题目要求，提取关键约束
- 算法选择：根据数据规模和时间要求选择合适算法
- 边界处理：考虑特殊情况和边界条件
- 复杂度分析：准确计算时间和空间复杂度

### 2. 代码实现
- 模板复用：准备常用算法模板
- 调试技巧：使用打印语句跟踪变量变化
- 测试用例：准备典型和边界测试用例

## 扩展学习

### 1. 高级线段树
- 动态开点线段树
- 李超线段树
- 吉司机线段树

### 2. 相关算法
- 树状数组
- 平衡二叉搜索树
- 分块算法

### 3. 实际应用场景
- 数据库索引
- 图形学中的区间查询
- 机器学习中的特征选择

## 学习建议

### 1. 循序渐进
1. 掌握基础：先理解基本概念和操作
2. 练习应用：通过题目加深理解
3. 总结规律：归纳常见解题模式

### 2. 多语言实践
1. Java：熟悉静态数组和 IO 优化
2. C++：掌握 STL 容器和内存管理
3. Python：利用列表操作和内置函数

### 3. 持续提升
1. 定期复习：巩固已学知识点
2. 扩展学习：了解相关算法和数据结构
3. 实战应用：将算法应用到实际项目中

==========

文件: SUMMARY.md
==========
# Class109 算法题目总结

## 主要算法和数据结构

Class109 主要涉及以下算法和数据结构：

1. **树状数组（Fenwick Tree/Binary Indexed Tree）**
2. **归并排序及其应用**
3. **离散化技术**
4. **动态规划优化**

## 题目列表及解析

### 1. 逆序对问题
- **相关文件**: Code01_NumberOfReversePair1.java, Code01_NumberOfReversePair2.java
- **题目**: 给定一个数组，求逆序对的数量
- **解法**: 归并排序
- **时间复杂度**: O(n log n)
- **空间复杂度**: O(n)

### 2. 升序三元组数量
- **相关文件**: Code02_IncreasingTriples.java
- **题目**: 求升序三元组(i,j,k)满足 i<j<k 且 arr[i]<arr[j]<arr[k]的数量
- **解法**: 树状数组
- **时间复杂度**: O(n log n)
- **空间复杂度**: O(n)

### 3. 最长递增子序列的个数
- **相关文件**: Code03_NumberOfLIS.java
- **题目**: 求最长递增子序列的个数
- **解法**: 树状数组优化动态规划
- **时间复杂度**: O(n log n)
- **空间复杂度**: O(n)

### 4. HH 的项链（区间不同元素个数查询）
- **相关文件**: Code04_DifferentColors.java
- **题目**: 多次查询区间内不同元素的个数
- **解法**: 树状数组 + 离线处理
- **时间复杂度**: O((n + m) log n)
- **空间复杂度**: O(n + m)

### 5. 得到回文串的最少操作次数
- **相关文件**: Code05_MinimumNumberOfMovesToMakePalindrome.java
- **题目**: 通过相邻元素交换得到回文串的最少操作次数
- **解法**: 树状数组 + 归并排序
- **时间复杂度**: O(n log n)
- **空间复杂度**: O(n)

### 6. 翻转对
- **相关文件**: Code06_ReversePairs.java, Code06_ReversePairs.cpp, Code06_ReversePairs.py
- **题目**: 求满足 i<j 且 nums[i]>2*nums[j]的重要翻转对数量
- **解法**: 归并排序
- **时间复杂度**: O(n log n)
- **空间复杂度**: O(n)

### 7. 最长递增子序列
- **相关文件**: Code07_LIS_BIT.java, Code07_LIS_BIT.cpp, Code07_LIS_BIT.py
- **题目**: 求最长递增子序列的长度
- **解法**: 树状数组优化动态规划
- **时间复杂度**: O(n log n)
- **空间复杂度**: O(n)

### 8. 最长递增子序列的个数（进阶）
- **相关文件**: Code08_NumberOfLISAdvanced.java, Code08_NumberOfLISAdvanced.cpp, Code08_NumberOfLISAdvanced.py
- **题目**: 求最长递增子序列的个数
- **解法**: 树状数组优化动态规划
- **时间复杂度**: O(n log n)
- **空间复杂度**: O(n)

### 9. 统计数组中好三元组数目
- **相关文件**: Code09_GoodTriplets.java, Code09_GoodTriplets.cpp, Code09_GoodTriplets.py
- **题目**: 求两个数组中位置顺序一致的三元组数量
- **解法**: 树状数组
- **时间复杂度**: O(n log n)
- **空间复杂度**: O(n)

## 算法思路与技巧

### 树状数组应用场景
1. **前缀和查询与单点更新**: 最基础的应用
2. **区间和查询与单点更新**: 通过前缀和差分实现
3. **优化动态规划**: 维护以某值结尾的最优解
4. **离线处理**: 结合排序和扫描线思想

### 归并排序应用场景
1. **逆序对计数**: 在合并过程中统计
2. **翻转对计数**: 类似逆序对的变形
3. **小和问题**: 计算每个元素左边比它小的元素之和

### 离散化技术
1. **处理大数值**: 将大数值映射到连续的小范围
2. **去重**: 去除重复元素，减少空间消耗
3. **排序映射**: 保持原有大小关系

## 工程化考量

### 异常处理
1. **空输入处理**: 检查数组是否为空
2. **边界条件**: 处理数组长度为 1 或 2 的情况
3. **数值溢出**: 使用 long 类型处理大数运算

### 性能优化
1. **IO 优化**: 使用快速 IO 读写
2. **内存复用**: 复用数组空间，减少内存分配
3. **常数优化**: 减少不必要的计算和比较

### 代码可读性
1. **命名规范**: 变量命名见名知意
2. **注释完整**: 详细解释算法思路和关键步骤
3. **模块化**: 将功能拆分为独立的方法

## 语言特性差异

### Java
1. **静态数组**: 预分配固定大小数组提高性能
2. **IO 优化**: 使用 StreamTokenizer 和 BufferedReader

### C++
1. **STL 容器**: vector、algorithm 等标准库
2. **内存管理**: 手动管理内存，注意初始化

### Python
1. **列表操作**: 列表推导式和内置函数
2. **动态类型**: 灵活但需注意类型转换

## 面试技巧

### 解题思路
1. **问题分析**: 理解题目要求，提取关键约束
2. **算法选择**: 根据数据规模和时间要求选择合适算法
3. **边界处理**: 考虑特殊情况和边界条件
4. **复杂度分析**: 准确计算时间和空间复杂度

### 代码实现
1. **模板复用**: 准备常用算法模板
2. **调试技巧**: 使用打印语句跟踪变量变化
3. **测试用例**: 准备典型和边界测试用例

## 扩展学习

### 相关算法
1. **线段树**: 更强大的区间数据结构
2. **平衡二叉搜索树**: 动态维护有序序列
3. **分块**: 平衡时间复杂度和实现复杂度

### 应用领域
1. **机器学习**: 特征选择和排序算法
2. **图像处理**: 像素排序和滤波
3. **自然语言处理**: 文本排序和匹配

==========

[代码文件]
==========
文件: Code01_NumberOfReversePair1.java
==========
```
package class109;

// 逆序对数量(归并分治)
// 给定一个长度为 n 的数组 arr
// 如果 i < j 且 arr[i] > arr[j]
// 那么(i,j)就是一个逆序对
// 求 arr 中逆序对的数量
// 1 <= n <= 5 * 10^5
// 1 <= arr[i] <= 10^9
// 测试链接 : https://www.luogu.com.cn/problem/P1908
// 请同学们务必参考如下代码中关于输入、输出的处理
// 这是输入输出处理效率很高的写法
// 提交以下的 code，提交时请把类名改成"Main"，可以直接通过

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

/**
 * 使用归并排序思想解决逆序对问题
 *
 * 解题思路:
 * 1. 采用分治思想，将数组不断二分，直到只有一个元素
```

```
 * 2. 在合并过程中统计逆序对数量
 * 3. 对于左半部分[l...m]和右半部分[m+1...r]：
 *    - 当 arr[i] > arr[j]时，说明从 i 到 m 的所有元素都大于 arr[j]，产生 m-i+1 个逆序对
 *    - 当 arr[i] <= arr[j]时，不产生逆序对
 *
 * 时间复杂度分析：
 * - 归并排序的时间复杂度为 O(n log n)
 * - 每一层递归都会遍历所有元素，共 log n 层
 * - 所以总时间复杂度为 O(n log n)
 *
 * 空间复杂度分析：
 * - 需要额外的 help 数组存储临时数据，空间复杂度为 O(n)
 * - 递归调用栈的深度为 O(log n)
 * - 所以总空间复杂度为 O(n)
 */
public class Code01_NumberOfReversePair1 {

    // 最大数组长度
    public static int MAXN = 500001;

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

    // 辅助数组，用于归并过程
    public static int[] help = new int[MAXN];

    // 数组长度
    public static int n;

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

```
}

/**
 * 计算逆序对数量的入口方法
 *
 * @return 逆序对总数
 */
public static long compute() {
    return f(1, n);
}

/**
 * 分治计算区间[l,r]内的逆序对数量
 *
 * @param l 区间左边界（包含）
 * @param r 区间右边界（包含）
 * @return 区间内逆序对数量
 */
public static long f(int l, int r) {
    // 递归终止条件：只有一个元素时，没有逆序对
    if (l == r) {
        return 0;
    }
    // 二分中点
    int m = (l + r) / 2;
    // 递归计算左半部分、右半部分的逆序对数量，再加上合并时产生的逆序对数量
    return f(l, m) + f(m + 1, r) + merge(l, m, r);
}

/**
 * 合并两个有序数组，并统计合并过程中产生的逆序对数量
 *
 * @param l 左半部分起始位置
 * @param m 左半部分结束位置
 * @param r 右半部分结束位置
 * @return 合并过程中产生的逆序对数量
 */
public static long merge(int l, int m, int r) {
    // i 来自 l.....m（左半部分）
    // j 来自 m+1...r（右半部分）
    // 统计有多少逆序对

    // 逆序对数量
```

```
        long ans = 0;

        // 从后往前比较，统计左半部分中大于右半部分元素的个数
        for (int i = m, j = r; i >= l; i--) {
            // 找到右半部分中第一个小于 arr[i]的元素位置
            while (j >= m + 1 && arr[i] <= arr[j]) {
                j--;
            }
            // 此时 j 指向右半部分中最后一个满足 arr[i] > arr[j]的元素位置
            // 从 m+1 到 j 的所有元素都与 arr[i]构成逆序对
            ans += j - m;
        }

        // 左右部分合并，整体变有序，归并排序的过程
        int i = l;
        int a = l;
        int b = m + 1;
        while (a <= m && b <= r) {
            help[i++] = arr[a] <= arr[b] ? arr[a++] : arr[b++];
        }
        while (a <= m) {
            help[i++] = arr[a++];
        }
        while (b <= r) {
            help[i++] = arr[b++];
        }
        for (i = l; i <= r; i++) {
            arr[i] = help[i];
        }
        return ans;
    }

}
```

==========

文件: Code01_NumberOfReversePair2.java

==========

```
package class109;

// 逆序对数量(值域树状数组)
// 给定一个长度为 n 的数组 arr
// 如果 i < j 且 arr[i] > arr[j]
```

```
// 那么(i,j)就是一个逆序对
// 求 arr 中逆序对的数量
// 1 <= n <= 5 * 10^5
// 1 <= arr[i] <= 10^9
// 测试链接 : https://www.luogu.com.cn/problem/P1908
// 请同学们务必参考如下代码中关于输入、输出的处理
// 这是输入输出处理效率很高的写法
// 提交以下的 code，提交时请把类名改成"Main"，可以直接通过

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

/**
 * 使用值域树状数组解决逆序对问题
 *
 * 解题思路：
 * 1. 离散化处理：由于数组元素值可能很大（1 <= arr[i] <= 10^9），但数量有限（n <= 5*10^5），
 *    所以需要离散化处理，将原数值映射到 1~m 的范围内
 * 2. 从右往左遍历数组，对每个元素：
 *    - 查询树状数组中比当前元素小的元素个数，即为以当前元素为第一元素的逆序对数量
 *    - 将当前元素加入树状数组
 *
 * 时间复杂度分析：
 * - 离散化排序：O(n log n)
 * - 遍历数组，每次操作树状数组：O(n log n)
 * - 总时间复杂度：O(n log n)
 *
 * 空间复杂度分析：
 * - 需要额外数组存储原始数据、排序数据和树状数组：O(n)
 * - 所以总空间复杂度为 O(n)
 */
public class Code01_NumberOfReversePair2 {

    // 最大数组长度
    public static int MAXN = 500001;

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

```
// 排序数组，用于离散化
public static int[] sort = new int[MAXN];

// 树状数组，用于维护元素出现次数
public static int[] tree = new int[MAXN];

// 数组长度和离散化后数组长度
public static int n, m;

/**
 * lowbit 函数：获取数字的二进制表示中最右边的 1 所代表的数值
 * 例如：x=6(110) 返回 2(010)，x=12(1100) 返回 4(0100)
 *
 * @param i 输入数字
 * @return 最低位的 1 所代表的数值
 */
public static int lowbit(int i) {
    return i & -i;
}

/**
 * 单点增加操作：在位置 i 上增加 v
 *
 * @param i 位置（从 1 开始）
 * @param v 增加的值
 */
public static void add(int i, int v) {
    // 从位置 i 开始，沿着父节点路径向上更新所有相关的节点
    while (i <= m) {
        tree[i] += v;
        // 移动到父节点
        i += lowbit(i);
    }
}

/**
 * 查询前缀和：计算从位置 1 到位置 i 的所有元素之和
 *
 * @param i 查询的结束位置
 * @return 前缀和
 */
public static long sum(int i) {
```

```
    long ans = 0;
    // 从位置 i 开始，沿着子节点路径向下累加
    while (i > 0) {
        ans += tree[i];
        // 移动到前一个相关区间
        i -= lowbit(i);
    }
    return ans;
}

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

/**
 * 计算逆序对数量
 *
 * @return 逆序对总数
 */
public static long compute() {
    // 离散化处理
    Arrays.sort(sort, 1, n + 1);
    m = 1;
    for (int i = 2; i <= n; i++) {
        // 去重
        if (sort[m] != sort[i]) {
            sort[++m] = sort[i];
        }
    }
```

```
        // 将原数组元素替换为离散化后的排名
        for (int i = 1; i <= n; i++) {
            arr[i] = rank(arr[i]);
        }

        long ans = 0;
        // 从右往左遍历数组
        for (int i = n; i >= 1; i--) {
            // 右边有多少数字是 <= 当前数值 - 1，即为以 arr[i]为第一元素的逆序对数量
            ans += sum(arr[i] - 1);
            // 增加当前数字的词频
            add(arr[i], 1);
        }
        return ans;
    }

    /**
     * 给定原始值 v，返回其在离散化数组中的排名（即在排序数组中的位置）
     *
     * @param v 原始值
     * @return 排名值(排序部分 1~m 中的下标)
     */
    public static int rank(int v) {
        int l = 1, r = m, mid;
        int ans = 0;
        while (l <= r) {
            mid = (l + r) / 2;
            if (sort[mid] >= v) {
                ans = mid;
                r = mid - 1;
            } else {
                l = mid + 1;
            }
        }
        return ans;
    }

}
```

==========

文件: Code02_IncreasingTriples.java

==========

```
package class109;

// 升序三元组数量
// 给定一个数组 arr，如果 i < j < k 且 arr[i] < arr[j] < arr[k]
// 那么称(i, j, k)为一个升序三元组
// 返回 arr 中升序三元组的数量
// 测试链接 : https://www.luogu.com.cn/problem/P1637
// 请同学们务必参考如下代码中关于输入、输出的处理
// 这是输入输出处理效率很高的写法
// 提交以下的 code，提交时请把类名改成"Main"，可以直接通过

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

/**
 * 使用树状数组解决升序三元组问题
 *
 * 解题思路：
 * 1. 对于每个元素 arr[i]，我们需要知道：
 *    - 在它左边有多少个元素小于它（形成升序一元组）
 *    - 在它左边有多少个元素能与它组成升序二元组
 * 2. 使用两个树状数组：
 *    - tree1[i]维护以数值 i 结尾的升序一元组数量（即小于 i 的元素个数）
 *    - tree2[i]维护以数值 i 结尾的升序二元组数量
 * 3. 遍历数组，对每个元素：
 *    - 查询 tree2 中比当前元素小的元素个数，即为以当前元素为结尾的升序三元组数量
 *    - 更新 tree1 中当前元素的计数
 *    - 查询 tree1 中比当前元素小的元素个数，更新 tree2 中当前元素的计数
 *
 * 时间复杂度分析：
 * - 离散化排序：O(n log n)
 * - 遍历数组，每次操作树状数组：O(n log n)
 * - 总时间复杂度：O(n log n)
 *
 * 空间复杂度分析：
 * - 需要额外数组存储原始数据、排序数据和两个树状数组：O(n)
 * - 所以总空间复杂度为 O(n)
 */
```

```
public class Code02_IncreasingTriples {

    // 最大数组长度
    public static int MAXN = 30001;

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

    // 排序数组，用于离散化
    public static int[] sort = new int[MAXN];

    // 维护信息 : 课上讲的 up1 数组
    // tree1 不是 up1 数组，是 up1 数组的树状数组
    // tree1[i]表示值小于等于 i 的元素个数（升序一元组数量）
    public static long[] tree1 = new long[MAXN];

    // 维护信息 : 课上讲的 up2 数组
    // tree2 不是 up2 数组，是 up2 数组的树状数组
    // tree2[i]表示以值 i 结尾的升序二元组数量
    public static long[] tree2 = new long[MAXN];

    // 数组长度和离散化后数组长度
    public static int n, m;

    /**
     * lowbit 函数：获取数字的二进制表示中最右边的 1 所代表的数值
     * 例如：x=6(110) 返回 2(010)，x=12(1100) 返回 4(0100)
     *
     * @param i 输入数字
     * @return 最低位的 1 所代表的数值
     */
    public static int lowbit(int i) {
        return i & -i;
    }

    /**
     * 单点增加操作：在位置 i 上增加 v
     *
     * @param tree 树状数组
     * @param i 位置（从 1 开始）
     * @param c 增加的值
     */
    public static void add(long[] tree, int i, long c) {
```

```
    // 从位置 i 开始，沿着父节点路径向上更新所有相关的节点
    while (i <= m) {
        tree[i] += c;
        // 移动到父节点
        i += lowbit(i);
    }
}

/**
 * 查询前缀和：计算从位置 1 到位置 i 的所有元素之和
 *
 * @param tree 树状数组
 * @param i 查询的结束位置
 * @return 前缀和
 */
public static long sum(long[] tree, int i) {
    long ans = 0;
    // 从位置 i 开始，沿着子节点路径向下累加
    while (i > 0) {
        ans += tree[i];
        // 移动到前一个相关区间
        i -= lowbit(i);
    }
    return ans;
}

public static void main(String[] args) throws IOException {
    BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
    StreamTokenizer in = new StreamTokenizer(br);
    PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));
    in.nextToken();
    n = (int) in.nval;
    for (int i = 1; i <= n; i++) {
        in.nextToken();
        arr[i] = (int) in.nval;
        sort[i] = arr[i];
    }
    out.println(compute());
    out.flush();
    out.close();
    br.close();
}
```

```
/**
 * 计算升序三元组数量
 *
 * @return 升序三元组总数
 */
// 时间复杂度O(n * logn)
public static long compute() {
    // 离散化处理
    Arrays.sort(sort, 1, n + 1);
    m = 1;
    for (int i = 2; i <= n; i++) {
        // 去重
        if (sort[m] != sort[i]) {
            sort[++m] = sort[i];
        }
    }

    // 将原数组元素替换为离散化后的排名
    for (int i = 1; i <= n; i++) {
        arr[i] = rank(arr[i]);
    }

    long ans = 0;
    // 遍历数组，对每个元素计算以它为结尾的升序三元组数量
    for (int i = 1; i <= n; i++) {
        // 查询以当前值做结尾的升序三元组数量
        // 即查询右方有多少数字能与当前数字组成升序二元组
        ans += sum(tree2, arr[i] - 1);

        // 更新以当前值做结尾的升序一元组数量（单个元素）
        add(tree1, arr[i], 1);

        // 更新以当前值做结尾的升序二元组数量
        // 即当前元素与左方比它小的元素组成的二元组数量
        add(tree2, arr[i], sum(tree1, arr[i] - 1));
    }
    return ans;
}

/**
 * 给定原始值 v，返回其在离散化数组中的排名（即在排序数组中的位置）
 *
 * @param v 原始值
```

```
     * @return 排名值(排序部分 1~m 中的下标)
     */
    public static int rank(int v) {
        int l = 1, r = m, mid;
        int ans = 0;
        while (l <= r) {
            mid = (l + r) / 2;
            if (sort[mid] >= v) {
                ans = mid;
                r = mid - 1;
            } else {
                l = mid + 1;
            }
        }
        return ans;
    }

}
```

==========

文件: Code03_NumberOfLIS.java

==========

```
package class109;

import java.util.Arrays;

// 最长递增子序列的个数
// 给定一个未排序的整数数组 nums，返回最长递增子序列的个数
// 测试链接 : https://leetcode.cn/problems/number-of-longest-increasing-subsequence/
// 本题在讲解 072，最长递增子序列问题与扩展，就做出过预告
// 具体可以看讲解 072 视频最后的部分
// 用树状数组实现时间复杂度 O(n * logn)

/**
 * 使用树状数组解决最长递增子序列的个数问题
 *
 * 解题思路：
 * 1. 对于每个元素，我们需要知道以它结尾的最长递增子序列的长度和数量
 * 2. 使用两个树状数组：
 *    - treeMaxLen[i]维护以数值 i 结尾的最长递增子序列的长度
 *    - treeMaxLenCnt[i]维护以数值 i 结尾的最长递增子序列的数量
 * 3. 遍历数组，对每个元素：
```

```
 *     - 查询小于当前元素的数值中，最长递增子序列的长度和数量
 *     - 根据查询结果更新当前元素对应的树状数组
 *
 * 时间复杂度分析：
 * - 离散化排序：O(n log n)
 * - 遍历数组，每次操作树状数组：O(n log n)
 * - 总时间复杂度：O(n log n)
 *
 * 空间复杂度分析：
 * - 需要额外数组存储原始数据、排序数据和两个树状数组：O(n)
 * - 所以总空间复杂度为O(n)
 */
public class Code03_NumberOfLIS {

    // 最大数组长度
    public static int MAXN = 2001;

    // 排序数组，用于离散化
    public static int[] sort = new int[MAXN];

    // 维护信息 : 以数值 i 结尾的最长递增子序列，长度是多少
    // 维护的信息以树状数组组织
    public static int[] treeMaxLen = new int[MAXN];

    // 维护信息 : 以数值 i 结尾的最长递增子序列，个数是多少
    // 维护的信息以树状数组组织
    public static int[] treeMaxLenCnt = new int[MAXN];

    // 离散化后数组长度
    public static int m;

    /**
     * lowbit 函数：获取数字的二进制表示中最右边的 1 所代表的数值
     * 例如：x=6(110) 返回 2(010)，x=12(1100) 返回 4(0100)
     *
     * @param i 输入数字
     * @return 最低位的 1 所代表的数值
     */
    public static int lowbit(int i) {
        return i & -i;
    }

    // 查询结尾数值<=i 的最长递增子序列的长度，赋值给 maxLen
```

```
// 查询结尾数值<=i 的最长递增子序列的个数，赋值给 maxLenCnt
public static int maxLen, maxLenCnt;

/**
 * 查询结尾数值<=i 的最长递增子序列的长度和数量
 *
 * @param i 查询的结束位置
 */
public static void query(int i) {
    maxLen = maxLenCnt = 0;
    while (i > 0) {
        if (maxLen == treeMaxLen[i]) {
            // 如果长度相同，数量累加
            maxLenCnt += treeMaxLenCnt[i];
        } else if (maxLen < treeMaxLen[i]) {
            // 如果找到更长的长度，更新长度和数量
            maxLen = treeMaxLen[i];
            maxLenCnt = treeMaxLenCnt[i];
        }
        i -= lowbit(i);
    }
}

/**
 * 以数值 i 结尾的最长递增子序列，长度达到了 len，个数增加了 cnt
 * 更新树状数组
 *
 * @param i 数值
 * @param len 最长递增子序列长度
 * @param cnt 最长递增子序列数量
 */
public static void add(int i, int len, int cnt) {
    while (i <= m) {
        if (treeMaxLen[i] == len) {
            // 如果长度相同，数量累加
            treeMaxLenCnt[i] += cnt;
        } else if (treeMaxLen[i] < len) {
            // 如果找到更长的长度，更新长度和数量
            treeMaxLen[i] = len;
            treeMaxLenCnt[i] = cnt;
        }
        i += lowbit(i);
    }
```

```
}

/**
 * 计算最长递增子序列的个数
 *
 * @param nums 输入数组
 * @return 最长递增子序列的个数
 */
public static int findNumberOfLIS(int[] nums) {
    int n = nums.length;
    for (int i = 1; i <= n; i++) {
        sort[i] = nums[i - 1];
    }
    Arrays.sort(sort, 1, n + 1);
    m = 1;
    for (int i = 2; i <= n; i++) {
        if (sort[m] != sort[i]) {
            sort[++m] = sort[i];
        }
    }
    Arrays.fill(treeMaxLen, 1, m + 1, 0);
    Arrays.fill(treeMaxLenCnt, 1, m + 1, 0);
    int i;
    for (int num : nums) {
        i = rank(num);
        // 查询以数值<=i-1 结尾的最长递增子序列信息
        query(i - 1);
        if (maxLen == 0) {
            // 如果查出数值<=i-1 结尾的最长递增子序列长度为 0
            // 那么说明，以值 i 结尾的最长递增子序列长度就是 1，计数增加 1
            add(i, 1, 1);
        } else {
            // 如果查出数值<=i-1 结尾的最长递增子序列长度为 maxLen != 0
            // 那么说明，以值 i 结尾的最长递增子序列长度就是 maxLen + 1，计数增加 maxLenCnt
            add(i, maxLen + 1, maxLenCnt);
        }
    }
    query(m);
    return maxLenCnt;
}

/**
 * 给定原始值 v，返回其在离散化数组中的排名（即在排序数组中的位置）
```

```
     *
     * @param v 原始值
     * @return 排名值(排序部分 1~m 中的下标)
     */
    public static int rank(int v) {
        int ans = 0;
        int l = 1, r = m, mid;
        while (l <= r) {
            mid = (l + r) / 2;
            if (sort[mid] >= v) {
                ans = mid;
                r = mid - 1;
            } else {
                l = mid + 1;
            }
        }
        return ans;
    }

}
```

==========

文件: Code04_DifferentColors.java

==========

```
package class109;

// HH 的项链
// 一共有 n 个位置，每个位置颜色给定，i 位置的颜色是 arr[i]
// 一共有 m 个查询，question[i] = {li, ri}
// 表示第 i 条查询想查 arr[li..ri]范围上一共有多少种不同颜色
// 返回每条查询的答案
// 1 <= n、m、arr[i] <= 10^6
// 1 <= li <= ri <= n
// 测试链接 : https://www.luogu.com.cn/problem/P1972
// 提交以下的 code，提交时请把类名改成"Main"，可以通过所有测试用例
// 代码逻辑和课上讲的完全一致，但是重写了读写工具类，增加了 io 效率

import java.io.BufferedWriter;
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStreamWriter;
import java.util.Arrays;
```

```
/**
 * 使用树状数组解决 HH 的项链问题
 *
 * 解题思路：
 * 1. 离线处理：将所有查询按照右端点排序
 * 2. 从左到右遍历数组，维护每个颜色最后出现的位置
 * 3. 使用树状数组维护区间和：
 *    - 当遇到一个颜色时，如果之前出现过，则在之前出现的位置-1
 *    - 在当前位置+1
 * 4. 对于每个查询，答案就是区间[li, ri]的和
 *
 * 时间复杂度分析：
 * - 排序查询：O(m log m)
 * - 遍历数组，每次操作树状数组：O(n log n)
 * - 处理查询：O(m log n)
 * - 总时间复杂度：O((n + m) * log n)
 *
 * 空间复杂度分析：
 * - 需要额外数组存储原始数据、查询数据、树状数组等：O(n + m)
 * - 所以总空间复杂度为O(n + m)
 */
public class Code04_DifferentColors {

    // 最大数组长度
    public static int MAXN = 1000001;

    // 原数组，arr[i]表示位置 i 的颜色
    public static int[] arr = new int[MAXN];

    // 查询数组，query[i][0]表示左端点，query[i][1]表示右端点，query[i][2]表示查询编号
    public static int[][] query = new int[MAXN][3];

    // 答案数组
    public static int[] ans = new int[MAXN];

    // 颜色映射数组，map[color]表示颜色 color 最后出现的位置
    public static int[] map = new int[MAXN];

    // 树状数组，用于维护区间和
    public static int[] tree = new int[MAXN];

    // 数组长度和查询数量
```

```
public static int n, m;

/**
 * lowbit 函数：获取数字的二进制表示中最右边的 1 所代表的数值
 * 例如：x=6(110) 返回 2(010)，x=12(1100) 返回 4(0100)
 *
 * @param i 输入数字
 * @return 最低位的 1 所代表的数值
 */
public static int lowbit(int i) {
    return i & -i;
}

/**
 * 单点增加操作：在位置 i 上增加 v
 *
 * @param i 位置（从 1 开始）
 * @param v 增加的值
 */
public static void add(int i, int v) {
    while (i <= n) {
        tree[i] += v;
        i += lowbit(i);
    }
}

/**
 * 查询前缀和：计算从位置 1 到位置 i 的所有元素之和
 *
 * @param i 查询的结束位置
 * @return 前缀和
 */
public static int sum(int i) {
    int ans = 0;
    while (i > 0) {
        ans += tree[i];
        i -= lowbit(i);
    }
    return ans;
}

/**
 * 查询区间和：计算从位置 l 到位置 r 的所有元素之和
```

```
 *
 * @param l 区间左端点
 * @param r 区间右端点
 * @return 区间和
 */
public static int range(int l, int r) {
    return sum(r) - sum(l - 1);
}

/**
 * 计算所有查询的答案
 */
public static void compute() {
    // 按照查询的右端点排序
    Arrays.sort(query, 1, m + 1, (a, b) -> a[1] - b[1]);

    // s 表示当前处理到的数组位置，q 表示当前处理到的查询编号
    for (int s = 1, q = 1, l, r, i; q <= m; q++) {
        // 当前查询的右端点
        r = query[q][1];

        // 处理从 s 到 r 位置的元素
        for (; s <= r; s++) {
            // 当前位置的颜色
            int color = arr[s];

            // 如果该颜色之前出现过，则在之前出现的位置-1
            if (map[color] != 0) {
                add(map[color], -1);
            }

            // 在当前位置+1
            add(s, 1);

            // 更新该颜色最后出现的位置
            map[color] = s;
        }

        // 当前查询的左端点
        l = query[q][0];
        // 当前查询的编号
        i = query[q][2];
        // 计算区间[l, r]的不同颜色数
```

```
            ans[i] = range(l, r);
        }
    }

    public static void main(String[] args) throws IOException {
        FastReader in = new FastReader();
        BufferedWriter out = new BufferedWriter(new OutputStreamWriter(System.out));
        n = in.nextInt();
        for (int i = 1; i <= n; i++) {
            arr[i] = in.nextInt();
        }
        m = in.nextInt();
        for (int i = 1; i <= m; i++) {
            query[i][0] = in.nextInt();
            query[i][1] = in.nextInt();
            query[i][2] = i;
        }
        compute();
        for (int i = 1; i <= m; i++) {
            out.write(ans[i] + "\n");
        }
        out.flush();
        out.close();
    }

    // 读写工具类
    static class FastReader {
        final private int BUFFER_SIZE = 1 << 16;
        private final InputStream in;
        private final byte[] buffer;
        private int ptr, len;

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

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

```
    return len > 0;
}

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

public boolean hasNext() throws IOException {
    while (hasNextByte()) {
        byte b = buffer[ptr];
        if (!isWhitespace(b))
            return true;
        ptr++;
    }
    return false;
}

public String next() throws IOException {
    byte c;
    do {
        c = readByte();
        if (c == -1)
            return null;
    } while (c <= ' ');
    StringBuilder sb = new StringBuilder();
    while (c > ' ') {
        sb.append((char) c);
        c = readByte();
    }
    return sb.toString();
}

public int nextInt() throws IOException {
    int num = 0;
    byte b = readByte();
    while (isWhitespace(b))
        b = readByte();
    boolean minus = false;
    if (b == '-') {
        minus = true;
        b = readByte();
```

```
            }
            while (!isWhitespace(b) && b != -1) {
                num = num * 10 + (b - '0');
                b = readByte();
            }
            return minus ? -num : num;
        }

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

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

}
```

==========

文件: Code05_MinimumNumberOfMovesToMakePalindrome.java

==========

```
package class109;

import java.util.Arrays;

// 得到回文串的最少操作次数
// 给你一个只包含小写英文字母的字符串 s
// 每一次操作可以选择 s 中两个相邻的字符进行交换
// 返回将 s 变成回文串的最少操作次数
// 输入数据会确保 s 一定能变成一个回文串
// 测试链接 : https://leetcode.cn/problems/minimum-number-of-moves-to-make-palindrome/

/**
 * 使用树状数组和归并分治解决回文串最少操作次数问题
 *
 * 解题思路：
 * 1. 首先确定每个字符在回文串中的位置：
 *    - 对于出现偶数次的字符，对称分布在字符串两端
 *    - 对于出现奇数次的字符，有一个会放在中间位置
 * 2. 构建位置映射数组 arr，arr[i]表示原位置 i 的字符在回文串中的位置
 * 3. 计算 arr 的逆序对数量，即为需要的最少交换次数
 *
 * 时间复杂度分析：
 * - 遍历字符串构建位置映射：O(n)
 * - 归并排序计算逆序对：O(n log n)
 * - 总时间复杂度：O(n log n)
 *
 * 空间复杂度分析：
 * - 需要额外数组存储位置映射、树状数组等：O(n)
 * - 所以总空间复杂度为 O(n)
 */
public class Code05_MinimumNumberOfMovesToMakePalindrome {

    // 最大数组长度
    public static int MAXN = 2001;

    // 字符种类数
    public static int MAXV = 26;

    // 字符串长度
    public static int n;

    // 字符数组
    public static char[] s;
```

```
// 所有字符的位置列表
// end[v]表示字符 v 最后出现的位置
public static int[] end = new int[MAXV];
// pre[i]表示位置 i 的字符在链表中的前一个位置
public static int[] pre = new int[MAXN];

// 树状数组，用于维护位置信息
public static int[] tree = new int[MAXN];

// 归并分治
// arr[i]记录每个位置的字符最终要去哪
public static int[] arr = new int[MAXN];
public static int[] help = new int[MAXN];

/**
 * 初始化各数组
 */
public static void build() {
    Arrays.fill(end, 0, MAXV, 0);
    Arrays.fill(arr, 1, n + 1, 0);
    Arrays.fill(tree, 1, n + 1, 0);
    for (int i = 1; i <= n; i++) {
        add(i, 1);
    }
}

/**
 * 将字符 v 的位置 j 加入列表
 *
 * @param v 字符
 * @param j 位置
 */
public static void push(int v, int j) {
    pre[j] = end[v];
    end[v] = j;
}

/**
 * 弹出当前 v 字符最后的下标
 *
 * @param v 字符
 * @return 位置
```

```
 */
public static int pop(int v) {
    int ans = end[v];
    end[v] = pre[end[v]];
    return ans;
}

/**
 * lowbit 函数：获取数字的二进制表示中最右边的 1 所代表的数值
 * 例如：x=6(110) 返回 2(010)，x=12(1100) 返回 4(0100)
 *
 * @param i 输入数字
 * @return 最低位的 1 所代表的数值
 */
public static int lowbit(int i) {
    return i & -i;
}

/**
 * 单点增加操作：在位置 i 上增加 v
 *
 * @param i 位置（从 1 开始）
 * @param v 增加的值
 */
public static void add(int i, int v) {
    while (i <= n) {
        tree[i] += v;
        i += lowbit(i);
    }
}

/**
 * 查询前缀和：计算从位置 1 到位置 i 的所有元素之和
 *
 * @param i 查询的结束位置
 * @return 前缀和
 */
public static int sum(int i) {
    int ans = 0;
    while (i > 0) {
        ans += tree[i];
        i -= lowbit(i);
    }
```

```
    return ans;
}

/**
 * 计算将字符串变成回文串的最少操作次数
 *
 * @param str 输入字符串
 * @return 最少操作次数
 */
// 时间复杂度O(n * logn)
public static int minMovesToMakePalindrome(String str) {
    s = str.toCharArray();
    n = s.length;
    build();

    // 构建每个字符的位置列表
    for (int i = 0, j = 1; i < n; i++, j++) {
        push(s[i] - 'a', j);
    }

    // arr[i]记录每个位置的字符最终要去哪
    for (int i = 0, l = 1, r, k; i < n; i++, l++) {
        if (arr[l] == 0) {
            // 弹出字符 s[i]最后出现的位置
            r = pop(s[i] - 'a');
            if (l < r) {
                // 计算位置 l 和 r 在回文串中的目标位置
                k = sum(l);
                arr[l] = k;
                arr[r] = n - k + 1;
            } else {
                // 奇数个字符的情况，放在中间
                arr[l] = (1 + n) / 2;
            }
            // 在树状数组中删除位置 r
            add(r, -1);
        }
    }
    // 使用归并排序计算 arr 的逆序对数量
    return number(1, n);
}

/**
```

```
 * 归并分治计算区间[l,r]内的逆序对数量
 *
 * @param l 区间左边界
 * @param r 区间右边界
 * @return 逆序对数量
 */
public static int number(int l, int r) {
    if (l >= r) {
        return 0;
    }
    int m = (l + r) / 2;
    return number(l, m) + number(m + 1, r) + merge(l, m, r);
}

/**
 * 合并两个有序数组，并统计合并过程中产生的逆序对数量
 *
 * @param l 左半部分起始位置
 * @param m 左半部分结束位置
 * @param r 右半部分结束位置
 * @return 合并过程中产生的逆序对数量
 */
public static int merge(int l, int m, int r) {
    int ans = 0;
    for (int i = m, j = r; i >= l; i--) {
        while (j >= m + 1 && arr[i] <= arr[j]) {
            j--;
        }
        ans += j - m;
    }
    int i = l;
    int a = l;
    int b = m + 1;
    while (a <= m && b <= r) {
        help[i++] = arr[a] <= arr[b] ? arr[a++] : arr[b++];
    }
    while (a <= m) {
        help[i++] = arr[a++];
    }
    while (b <= r) {
        help[i++] = arr[b++];
    }
    for (i = l; i <= r; i++) {
```

```
            arr[i] = help[i];
        }
        return ans;
    }

}
```

==========

文件: Code06_ReversePairs.cpp

==========

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

/**
 * 翻转对问题
 * 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。
 *
 * 示例 1:
 * 输入: [1,3,2,3,1]
 * 输出: 2
 *
 * 示例 2:
 * 输入: [2,4,3,5,1]
 * 输出: 3
 *
 * 提示：
 * 1 <= nums.length <= 5 * 10^4
 * -2^31 <= nums[i] <= 2^31 - 1
 *
 * 解题思路：
 * 使用归并排序的思想，在归并的过程中统计翻转对的数量。
 * 对于区间 [l, r]，我们将其分为 [l, mid] 和 [mid + 1, r]，先统计左右子区间内的翻转对数量，
 * 然后统计跨越左右子区间的翻转对数量，最后进行归并排序。
 *
 * 时间复杂度分析：
 * - 归并排序的时间复杂度为 O(n log n)
 * - 每一层递归中，统计翻转对的时间为 O(n)
 * - 总时间复杂度为 O(n log n)
 *
```

```
 * 空间复杂度分析：
 * - 需要额外的辅助数组存储临时数据，空间复杂度为 O(n)
 * - 递归调用栈的深度为 O(log n)
 * - 总空间复杂度为 O(n)
 *
 * LeetCode 493. 翻转对
 * 链接：https://leetcode.cn/problems/reverse-pairs/
 */
class Code06_ReversePairs {
private:
    // 最大数组长度
    const static int MAXN = 50001;
    // 辅助数组，用于归并过程
    long long help[MAXN];

    /**
     * 归并排序并统计区间[l, r]内的翻转对数量
     *
     * @param arr 数组
     * @param l 左边界
     * @param r 右边界
     * @return 区间内的翻转对数量
     */
    int f(vector<long long>& arr, int l, int r) {
        // 递归终止条件：只有一个元素时，没有翻转对
        if (l == r) {
            return 0;
        }

        // 计算中间位置
        int m = l + ((r - l) >> 1);

        // 统计左半部分、右半部分以及跨越中间的翻转对数量
        return f(arr, l, m) + f(arr, m + 1, r) + merge(arr, l, m, r);
    }

    /**
     * 合并两个有序数组，并统计跨越中间的翻转对数量
     *
     * @param arr 数组
     * @param l 左边界
     * @param m 中间位置
     * @param r 右边界
```

```
 * @return 跨越中间的翻转对数量
 */
int merge(vector<long long>& arr, int l, int m, int r) {
    // 统计翻转对数量
    int ans = 0;
    // 计算满足 arr[i] > 2 * arr[j] 的对数
    for (int i = l, j = m + 1; i <= m; i++) {
        // 对于每个 i，找到最大的 j 使得 arr[i] > 2 * arr[j]
        while (j <= r && arr[i] > 2 * arr[j]) {
            j++;
        }
        // j - (m + 1) 就是满足条件的 j 的数量
        ans += j - (m + 1);
    }

    // 归并排序
    int i = l;
    int p1 = l;
    int p2 = m + 1;

    while (p1 <= m && p2 <= r) {
        help[i++] = arr[p1] <= arr[p2] ? arr[p1++] : arr[p2++];
    }

    while (p1 <= m) {
        help[i++] = arr[p1++];
    }

    while (p2 <= r) {
        help[i++] = arr[p2++];
    }

    // 将辅助数组中的元素复制回原数组
    for (i = l; i <= r; i++) {
        arr[i] = help[i];
    }

    return ans;
}

// ==================== 补充题目：LeetCode 327. 区间和的个数 ====================
/**
 * 归并并统计符合条件的区间和数量
```

```
     */
    void mergeRange(vector<long long>& arr, int left, int mid, int right) {
        vector<long long> temp(right - left + 1);
        int i = left;
        int j = mid + 1;
        int k = 0;

        while (i <= mid && j <= right) {
            if (arr[i] <= arr[j]) {
                temp[k++] = arr[i++];
            } else {
                temp[k++] = arr[j++];
            }
        }

        while (i <= mid) {
            temp[k++] = arr[i++];
        }

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

        for (k = 0; k < temp.size(); k++) {
            arr[left + k] = temp[k];
        }
    }

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
     */
    int reversePairs(vector<int>& nums) {
        // 处理空数组或只有一个元素的情况
        if (nums.empty() || nums.size() < 2) {
            return 0;
        }

        // 转换为long long类型，防止溢出
        vector<long long> arr(nums.begin(), nums.end());
```

```
        // 调用归并排序并统计翻转对
        return f(arr, 0, arr.size() - 1);
    }

    // ==================== 补充题目：LeetCode 315. 计算右侧小于当前元素的个数 ====================
    /**
     * LeetCode 315. 计算右侧小于当前元素的个数
     * 链接：https://leetcode.cn/problems/count-of-smaller-numbers-after-self/
     * 题目：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：
     * 1. 离散化处理数组元素
     * 2. 使用树状数组从右向左扫描，统计比当前元素小的元素个数
     */
    vector<int> countSmaller(vector<int>& nums) {
        if (nums.empty()) {
            return vector<int>();
        }

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

        // 离散化处理
        vector<int> sorted = nums;
        sort(sorted.begin(), sorted.end());
        for (int i = 0; i < n; i++) {
            nums[i] = lower_bound(sorted.begin(), sorted.end(), nums[i]) - sorted.begin() + 1; // 映射到 1~n
        }

        // 树状数组实现
        vector<int> bit(n + 1, 0);

        auto lowbit = [](int x) { return x & (-x); };

        auto update = [&bit, &lowbit, n](int index, int delta) {
            while (index <= n) {
                bit[index] += delta;
                index += lowbit(index);
            }
```

```
    };

    auto query = [&bit, &lowbit](int index) {
        int sum = 0;
        while (index > 0) {
            sum += bit[index];
            index -= lowbit(index);
        }
        return sum;
    };

    // 从右向左扫描
    for (int i = n - 1; i >= 0; i--) {
        // 查询比当前元素小的个数（即查询[1, nums[i]-1]的和）
        result[i] = query(nums[i] - 1);
        // 更新树状数组，当前元素出现次数+1
        update(nums[i], 1);
    }

    return result;
}

/**
 * LeetCode 327. 区间和的个数
 * 链接：https://leetcode.cn/problems/count-of-range-sum/
 * 题目：给定一个整数数组 nums 以及两个整数 lower 和 upper 。
 * 求数组中，值位于范围 [lower, upper] （包含 lower 和 upper）之内的 区间和的个数 。
 * 区间和 S(i, j) 表示在 nums 中，位置从 i 到 j 的元素之和，包含 i 和 j (i ≤ j)。
 *
 * 解题思路：
 * 1. 计算前缀和数组
 * 2. 使用归并排序的思想，在归并过程中统计满足条件的区间和数量
 */
int countRangeSum(vector<int>& nums, int lower, int upper) {
    if (nums.empty()) {
        return 0;
    }

    int n = nums.size();
    vector<long long> prefixSum(n + 1, 0);

    // 计算前缀和
    for (int i = 0; i < n; i++) {
```

```
            prefixSum[i + 1] = prefixSum[i] + nums[i];
        }

        // 使用归并排序的方法统计符合条件的区间和
        function<int(vector<long long>&, int, int, int, int)> mergeSort =
        [&mergeSort, this](vector<long long>& arr, int left, int right, int lower, int upper) {
            if (left >= right) {
                return 0;
            }

            int mid = left + (right - left) / 2;
            int count = mergeSort(arr, left, mid, lower, upper) +
                        mergeSort(arr, mid + 1, right, lower, upper);

            // 统计跨越中间的符合条件的区间和
            int i = left;
            int L = mid + 1; // 第一个大于等于 (arr[i] + lower) 的位置
            int R = mid + 1; // 第一个大于 (arr[i] + upper) 的位置

            while (i <= mid) {
                // 找到 L 和 R 的位置
                while (L <= right && arr[L] - arr[i] < lower) {
                    L++;
                }
                while (R <= right && arr[R] - arr[i] <= upper) {
                    R++;
                }
                count += R - L;
                i++;
            }

            // 归并两个有序数组
            mergeRange(arr, left, mid, right);

            return count;
        };

        return mergeSort(prefixSum, 0, n, lower, upper);
    }
};

// 测试函数
int main() {
```

```
    Code06_ReversePairs solution;

    // 测试 LeetCode 493. 翻转对
    vector<int> nums1 = {1, 3, 2, 3, 1};
    cout << "LeetCode 493 测试用例 1:" << endl;
    cout << "输入: [1,3,2,3,1]" << endl;
    cout << "输出: " << solution.reversePairs(nums1) << endl;
    cout << "期望: 2" << endl << endl;

    vector<int> nums2 = {2, 4, 3, 5, 1};
    cout << "LeetCode 493 测试用例 2:" << endl;
    cout << "输入: [2,4,3,5,1]" << endl;
    cout << "输出: " << solution.reversePairs(nums2) << endl;
    cout << "期望: 3" << endl << endl;

    // 测试 LeetCode 315. 计算右侧小于当前元素的个数
    vector<int> nums3 = {5, 2, 6, 1};
    cout << "LeetCode 315 测试用例:" << endl;
    cout << "输入: [5,2,6,1]" << endl;
    cout << "输出: ";
    vector<int> result3 = solution.countSmaller(nums3);
    cout << "[";
    for (int i = 0; i < result3.size(); i++) {
        cout << result3[i];
        if (i < result3.size() - 1) cout << ", ";
    }
    cout << "]" << endl;
    cout << "期望: [2, 1, 1, 0]" << endl << endl;

    // 测试 LeetCode 327. 区间和的个数
    vector<int> nums4 = {-2, 5, -1};
    cout << "LeetCode 327 测试用例:" << endl;
    cout << "输入: nums = [-2,5,-1], lower = -2, upper = 2" << endl;
    cout << "输出: " << solution.countRangeSum(nums4, -2, 2) << endl;
    cout << "期望: 3" << endl;

    return 0;
}

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

文件: Code06_ReversePairs.java
==========================================
```

```
package class109;

import java.util.Arrays;

/**
 * 翻转对问题
 * 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。
 *
 * 示例 1:
 * 输入: [1,3,2,3,1]
 * 输出: 2
 *
 * 示例 2:
 * 输入: [2,4,3,5,1]
 * 输出: 3
 *
 * 提示：
 * 1 <= nums.length <= 5 * 10^4
 * -2^31 <= nums[i] <= 2^31 - 1
 *
 * 解题思路：
 * 使用归并排序的思想，在归并的过程中统计翻转对的数量。
 * 对于区间 [l, r]，我们将其分为 [l, mid] 和 [mid + 1, r]，先统计左右子区间内的翻转对数量，
 * 然后统计跨越左右子区间的翻转对数量，最后进行归并排序。
 *
 * 时间复杂度分析：
 * - 归并排序的时间复杂度为 O(n log n)
 * - 每一层递归中，统计翻转对的时间为 O(n)
 * - 总时间复杂度为 O(n log n)
 *
 * 空间复杂度分析：
 * - 需要额外的辅助数组存储临时数据，空间复杂度为 O(n)
 * - 递归调用栈的深度为 O(log n)
 * - 总空间复杂度为 O(n)
 *
 * LeetCode 493. 翻转对
 * 链接：https://leetcode.cn/problems/reverse-pairs/
 */
public class Code06_ReversePairs {

    // 最大数组长度
    public static int MAXN = 50001;
```

```
// 辅助数组，用于归并过程
public static long[] help = new long[MAXN];

/**
 * 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
 */
public static int reversePairs(int[] nums) {
    // 处理空数组或只有一个元素的情况
    if (nums == null || nums.length < 2) {
        return 0;
    }

    // 转换为long类型，防止溢出
    long[] arr = new long[nums.length];
    for (int i = 0; i < nums.length; i++) {
        arr[i] = nums[i];
    }

    // 调用归并排序并统计翻转对
    return f(arr, 0, arr.length - 1);
}

/**
 * 归并排序并统计区间[l, r]内的翻转对数量
 *
 * @param arr 数组
 * @param l 左边界
 * @param r 右边界
 * @return 区间内的翻转对数量
 */
private static int f(long[] arr, int l, int r) {
    // 递归终止条件：只有一个元素时，没有翻转对
    if (l == r) {
        return 0;
    }

    // 计算中间位置
    int m = l + ((r - l) >> 1);
```

```
    // 统计左半部分、右半部分以及跨越中间的翻转对数量
    return f(arr, l, m) + f(arr, m + 1, r) + merge(arr, l, m, r);
}

/**
 * 合并两个有序数组，并统计跨越中间的翻转对数量
 *
 * @param arr 数组
 * @param l 左边界
 * @param m 中间位置
 * @param r 右边界
 * @return 跨越中间的翻转对数量
 */
private static int merge(long[] arr, int l, int m, int r) {
    // 统计翻转对数量
    int ans = 0;
    // 计算满足 arr[i] > 2 * arr[j] 的对数
    for (int i = l, j = m + 1; i <= m; i++) {
        // 对于每个 i，找到最大的 j 使得 arr[i] > 2 * arr[j]
        while (j <= r && arr[i] > 2 * arr[j]) {
            j++;
        }
        // j - (m + 1) 就是满足条件的 j 的数量
        ans += j - (m + 1);
    }

    // 归并排序
    int i = l;
    int p1 = l;
    int p2 = m + 1;

    while (p1 <= m && p2 <= r) {
        help[i++] = arr[p1] <= arr[p2] ? arr[p1++] : arr[p2++];
    }

    while (p1 <= m) {
        help[i++] = arr[p1++];
    }

    while (p2 <= r) {
        help[i++] = arr[p2++];
    }
```

```
        // 将辅助数组中的元素复制回原数组
        for (i = l; i <= r; i++) {
            arr[i] = help[i];
        }

        return ans;
    }

    // ==================== 补充题目：LeetCode 315. 计算右侧小于当前元素的个数 ====================
    /**
     * LeetCode 315. 计算右侧小于当前元素的个数
     * 链接：https://leetcode.cn/problems/count-of-smaller-numbers-after-self/
     * 题目：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：
     * 1. 离散化处理数组元素
     * 2. 使用树状数组从右向左扫描，统计比当前元素小的元素个数
     */
    public static int[] countSmaller(int[] nums) {
        if (nums == null || nums.length == 0) {
            return new int[0];
        }

        int n = nums.length;
        int[] result = new int[n];

        // 离散化处理
        int[] sorted = Arrays.copyOf(nums, n);
        Arrays.sort(sorted);
        for (int i = 0; i < n; i++) {
            nums[i] = Arrays.binarySearch(sorted, nums[i]) + 1; // 映射到1~n
        }

        // 树状数组实现
        FenwickTree bit = new FenwickTree(n);

        // 从右向左扫描
        for (int i = n - 1; i >= 0; i--) {
            // 查询比当前元素小的个数（即查询[1, nums[i]-1]的和）
            result[i] = bit.query(nums[i] - 1);
            // 更新树状数组，当前元素出现次数+1
```

```
            bit.update(nums[i], 1);
        }

        return result;
    }

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

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

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

        public void update(int index, int delta) {
            while (index <= n) {
                tree[index] += delta;
                index += lowbit(index);
            }
        }

        public int query(int index) {
            int sum = 0;
            while (index > 0) {
                sum += tree[index];
                index -= lowbit(index);
            }
            return sum;
        }
    }

    // ==================== 补充题目：LeetCode 327. 区间和的个数 ====================
    /**
     * LeetCode 327. 区间和的个数
     * 链接：https://leetcode.cn/problems/count-of-range-sum/
     * 题目：给定一个整数数组 nums 以及两个整数 lower 和 upper 。
     * 求数组中，值位于范围 [lower, upper] （包含 lower 和 upper）之内的 区间和的个数 。
```

```
 * 区间和 S(i, j) 表示在 nums 中，位置从 i 到 j 的元素之和，包含 i 和 j (i ≤ j)。
 *
 * 解题思路：
 * 1. 计算前缀和数组
 * 2. 使用归并排序的思想，在归并过程中统计满足条件的区间和数量
 */
public static int countRangeSum(int[] nums, int lower, int upper) {
    if (nums == null || nums.length == 0) {
        return 0;
    }

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

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

    // 使用归并排序的方法统计符合条件的区间和
    return mergeSort(prefixSum, 0, n, lower, upper);
}

private static int mergeSort(long[] arr, int left, int right, int lower, int upper) {
    if (left >= right) {
        return 0;
    }

    int mid = left + (right - left) / 2;
    int count = mergeSort(arr, left, mid, lower, upper) +
                mergeSort(arr, mid + 1, right, lower, upper);

    // 统计跨越中间的符合条件的区间和
    int i = left;
    int L = mid + 1; // 第一个大于等于 (arr[i] + lower) 的位置
    int R = mid + 1; // 第一个大于 (arr[i] + upper) 的位置

    while (i <= mid) {
        // 找到L和R的位置
        while (L <= right && arr[L] - arr[i] < lower) {
            L++;
        }
        while (R <= right && arr[R] - arr[i] <= upper) {
```

```
                R++;
            }
            count += R - L;
            i++;
        }

        // 归并两个有序数组
        merge(arr, left, mid, right);

        return count;
    }

    private static void merge(long[] arr, int left, int mid, int right) {
        long[] temp = new long[right - left + 1];
        int i = left;
        int j = mid + 1;
        int k = 0;

        while (i <= mid && j <= right) {
            if (arr[i] <= arr[j]) {
                temp[k++] = arr[i++];
            } else {
                temp[k++] = arr[j++];
            }
        }

        while (i <= mid) {
            temp[k++] = arr[i++];
        }

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

        for (k = 0; k < temp.length; k++) {
            arr[left + k] = temp[k];
        }
    }
}
```

==========

文件: Code06_ReversePairs.py

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

"""
翻转对问题
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。

示例 1:
输入: [1,3,2,3,1]
输出: 2

示例 2:
输入: [2,4,3,5,1]
输出: 3

提示：
1 <= nums.length <= 5 * 10^4
-2^31 <= nums[i] <= 2^31 - 1

解题思路：
使用归并排序的思想，在归并的过程中统计翻转对的数量。
对于区间 [l, r]，我们将其分为 [l, mid] 和 [mid + 1, r]，先统计左右子区间内的翻转对数量，
然后统计跨越左右子区间的翻转对数量，最后进行归并排序。

时间复杂度分析：
- 归并排序的时间复杂度为 O(n log n)
- 每一层递归中，统计翻转对的时间为 O(n)
- 总时间复杂度为 O(n log n)

空间复杂度分析：
- 需要额外的辅助数组存储临时数据，空间复杂度为 O(n)
- 递归调用栈的深度为 O(log n)
- 总空间复杂度为 O(n)

LeetCode 493. 翻转对
链接：https://leetcode.cn/problems/reverse-pairs/
"""

class Code06_ReversePairs:
    """
    翻转对问题的解决方案类
```

```
提供了计算重要翻转对数量的方法，以及其他相关问题的解决方案
"""

def reversePairs(self, nums):
    """
    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
    """
    # 处理空数组或只有一个元素的情况
    if not nums or len(nums) < 2:
        return 0

    # 转换为long类型，防止溢出
    arr = list(map(int, nums))

    # 调用归并排序并统计翻转对
    return self._merge_sort(arr, 0, len(arr) - 1)

def _merge_sort(self, arr, l, r):
    """
    归并排序并统计区间[l, r]内的翻转对数量

    Args:
        arr: 数组
        l: 左边界
        r: 右边界

    Returns:
        区间内的翻转对数量
    """
    # 递归终止条件：只有一个元素时，没有翻转对
    if l == r:
        return 0

    # 计算中间位置
    m = l + ((r - l) >> 1)

    # 统计左半部分、右半部分以及跨越中间的翻转对数量
```

```
    return (
        self._merge_sort(arr, l, m) +
        self._merge_sort(arr, m + 1, r) +
        self._merge(arr, l, m, r)
    )

def _merge(self, arr, l, m, r):
    """
    合并两个有序数组，并统计跨越中间的翻转对数量

    Args:
        arr: 数组
        l: 左边界
        m: 中间位置
        r: 右边界

    Returns:
        跨越中间的翻转对数量
    """
    # 统计翻转对数量
    ans = 0
    # 计算满足 arr[i] > 2 * arr[j] 的对数
    j = m + 1
    for i in range(l, m + 1):
        # 对于每个 i，找到最大的 j 使得 arr[i] > 2 * arr[j]
        while j <= r and arr[i] > 2 * arr[j]:
            j += 1
        # j - (m + 1) 就是满足条件的 j 的数量
        ans += j - (m + 1)

    # 归并排序
    help_arr = [0] * (r - l + 1)
    i = 0
    p1 = l
    p2 = m + 1

    while p1 <= m and p2 <= r:
        help_arr[i] = arr[p1] if arr[p1] <= arr[p2] else arr[p2]
        p1 += 1 if arr[p1] <= arr[p2] else 0
        p2 += 1 if arr[p1 - 1] > arr[p2 - 1] else 0
        i += 1

    while p1 <= m:
```

```
            help_arr[i] = arr[p1]
            p1 += 1
            i += 1

        while p2 <= r:
            help_arr[i] = arr[p2]
            p2 += 1
            i += 1

        # 将辅助数组中的元素复制回原数组
        for i in range(len(help_arr)):
            arr[l + i] = help_arr[i]

        return ans

    # ==================== 补充题目：LeetCode 315. 计算右侧小于当前元素的个数 ====================
    """
    LeetCode 315. 计算右侧小于当前元素的个数
    链接：https://leetcode.cn/problems/count-of-smaller-numbers-after-self/
    题目：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：
    1. 离散化处理数组元素
    2. 使用树状数组从右向左扫描，统计比当前元素小的元素个数
    """
    def countSmaller(self, nums):
        """
        计算右侧小于当前元素的个数

        Args:
            nums: 输入数组

        Returns:
            结果数组，其中 counts[i]表示 nums[i]右侧小于 nums[i]的元素个数
        """
        if not nums:
            return []

        n = len(nums)
        result = [0] * n
```

```
    # 离散化处理
    sorted_nums = sorted(nums)
    # 创建离散化映射
    for i in range(n):
        # 使用bisect_left找到元素在排序数组中的位置
        # 映射到1~n
        nums[i] = self._bisect_left(sorted_nums, nums[i]) + 1

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

        def lowbit(self, x):
            return x & -x

        def update(self, index, delta):
            while index <= self.n:
                self.tree[index] += delta
                index += self.lowbit(index)

        def query(self, index):
            sum_val = 0
            while index > 0:
                sum_val += self.tree[index]
                index -= self.lowbit(index)
            return sum_val

    bit = FenwickTree(n)

    # 从右向左扫描
    for i in range(n - 1, -1, -1):
        # 查询比当前元素小的个数（即查询[1, nums[i]-1]的和）
        result[i] = bit.query(nums[i] - 1)
        # 更新树状数组，当前元素出现次数+1
        bit.update(nums[i], 1)

    return result

# ==================== 补充题目：LeetCode 327. 区间和的个数 ====================
"""
LeetCode 327. 区间和的个数
```

链接：https://leetcode.cn/problems/count-of-range-sum/
题目：给定一个整数数组 nums 以及两个整数 lower 和 upper 。
求数组中，值位于范围 [lower, upper] （包含 lower 和 upper）之内的 区间和的个数 。
区间和 S(i, j) 表示在 nums 中，位置从 i 到 j 的元素之和，包含 i 和 j (i ≤ j)。

解题思路：
1. 计算前缀和数组
2. 使用归并排序的思想，在归并过程中统计满足条件的区间和数量
"""

```
def countRangeSum(self, nums, lower, upper):
    """
    计算区间和的个数

    Args:
        nums: 输入数组
        lower: 区间和的下界
        upper: 区间和的上界

    Returns:
        满足条件的区间和个数
    """
    if not nums:
        return 0

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

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

    # 使用归并排序的方法统计符合条件的区间和
    def merge_sort(arr, left, right):
        if left >= right:
            return 0

        mid = left + (right - left) // 2
        count = merge_sort(arr, left, mid) + merge_sort(arr, mid + 1, right)

        # 统计跨越中间的符合条件的区间和
        i = left
        L = mid + 1  # 第一个大于等于 (arr[i] + lower) 的位置
        R = mid + 1  # 第一个大于 (arr[i] + upper) 的位置
```

```
        while i <= mid:
            # 找到L和R的位置
            while L <= right and arr[L] - arr[i] < lower:
                L += 1
            while R <= right and arr[R] - arr[i] <= upper:
                R += 1
            count += R - L
            i += 1

        # 归并两个有序数组
        merge(arr, left, mid, right)

        return count

    def merge(arr, left, mid, right):
        temp = []
        i = left
        j = mid + 1

        while i <= mid and j <= right:
            if arr[i] <= arr[j]:
                temp.append(arr[i])
                i += 1
            else:
                temp.append(arr[j])
                j += 1

        while i <= mid:
            temp.append(arr[i])
            i += 1

        while j <= right:
            temp.append(arr[j])
            j += 1

        for k in range(len(temp)):
            arr[left + k] = temp[k]

    return merge_sort(prefix_sum, 0, n)

def _bisect_left(self, sorted_arr, target):
    """
```

```
        二分查找，找到第一个大于等于 target 的元素的索引

        Args:
            sorted_arr: 已排序的数组
            target: 目标值

        Returns:
            索引位置
        """
        left, right = 0, len(sorted_arr)
        while left < right:
            mid = left + (right - left) // 2
            if sorted_arr[mid] < target:
                left = mid + 1
            else:
                right = mid
        return left

# 测试函数
def test_solution():
    solution = Code06_ReversePairs()

    # 测试 LeetCode 493. 翻转对
    print("LeetCode 493 测试用例 1:")
    nums1 = [1, 3, 2, 3, 1]
    print(f"输入: {nums1}")
    print(f"输出: {solution.reversePairs(nums1)}")
    print(f"期望: 2\n")

    print("LeetCode 493 测试用例 2:")
    nums2 = [2, 4, 3, 5, 1]
    print(f"输入: {nums2}")
    print(f"输出: {solution.reversePairs(nums2)}")
    print(f"期望: 3\n")

    # 测试 LeetCode 315. 计算右侧小于当前元素的个数
    print("LeetCode 315 测试用例:")
    nums3 = [5, 2, 6, 1]
    print(f"输入: {nums3}")
    result3 = solution.countSmaller(nums3)
    print(f"输出: {result3}")
    print(f"期望: [2, 1, 1, 0]\n")
```

```
    # 测试 LeetCode 327. 区间和的个数
    print("LeetCode 327 测试用例:")
    nums4 = [-2, 5, -1]
    lower, upper = -2, 2
    print(f"输入: nums = {nums4}, lower = {lower}, upper = {upper}")
    print(f"输出: {solution.countRangeSum(nums4, lower, upper)}")
    print(f"期望: 3")

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

==========

文件: Code07_LIS_BIT.cpp

==========

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <unordered_map>

using namespace std;

// 最长递增子序列（LIS）问题的树状数组解法
//
// 问题描述：
// 给定一个无序的整数数组，找到其中最长上升子序列的长度。
//
// 示例：
// 输入: [10,9,2,5,3,7,101,18]
// 输出: 4
// 解释: 最长的上升子序列是 [2,3,7,101]，长度是 4
//
// 解题思路：
// 使用树状数组优化动态规划解法，将时间复杂度从 O(n²)优化到 O(n log n)
//
// 1. 离散化处理：将原数组元素映射到较小的范围内（压缩值域）
// 2. 利用树状数组维护以每个值结尾的最长递增子序列长度
// 3. 对于每个元素 num[i]，查询比它小的元素中最大的 LIS 值，然后更新当前元素的 LIS 值
//
// 时间复杂度分析：
// - 离散化处理：O(n log n)
// - 每个元素查询和更新操作：O(log n)
// - 总时间复杂度：O(n log n)
```

```
//
// 空间复杂度分析：
// - 树状数组空间：O(n)
// - 离散化数组空间：O(n)
// - 总空间复杂度：O(n)
//
// LeetCode 300. 最长递增子序列
// 链接：https://leetcode.cn/problems/longest-increasing-subsequence/

class Code07_LIS_BIT {
private:
    // 树状数组类，用于最长递增子序列问题
    class FenwickTree {
    private:
        vector<int> tree;
        int n;

        // 计算 lowbit 值，即 x 的二进制表示中最低位 1 所对应的值
        int lowbit(int x) {
            return x & (-x);
        }

    public:
        // 构造函数
        FenwickTree(int size) : n(size), tree(size + 1, 0) {}

        // 查询[1, index]区间内的最大值
        int query(int index) {
            int max_val = 0;
            while (index > 0) {
                max_val = max(max_val, tree[index]);
                index -= lowbit(index);
            }
            return max_val;
        }

        // 将索引 index 的值更新为 value（保留最大值）
        void update(int index, int value) {
            while (index <= n) {
                tree[index] = max(tree[index], value);
                index += lowbit(index);
            }
        }
```

```
};

// 树状数组类，用于最长递增子序列个数问题
class FenwickTreeForCount {
private:
    vector<int> tree_len;   // 维护最长递增子序列的长度
    vector<int> tree_count; // 维护最长递增子序列的路径数
    int size;

    // 计算 lowbit 值
    int lowbit(int x) {
        return x & (-x);
    }

public:
    // 构造函数
    FenwickTreeForCount(int s) : size(s) {
        tree_len.resize(size + 1, 0);
        tree_count.resize(size + 1, 0);
    }

    // 更新树状数组
    void update(int index, int length, int count) {
        while (index <= size) {
            if (tree_len[index] < length) {
                tree_len[index] = length;
                tree_count[index] = count;
            } else if (tree_len[index] == length) {
                tree_count[index] += count;
            }
            index += lowbit(index);
        }
    }

    // 查询[1, index]区间内的最长递增子序列长度和对应路径数
    pair<int, int> query(int index) {
        int max_length = 0;
        int total_count = 0;

        while (index > 0) {
            if (tree_len[index] > max_length) {
                max_length = tree_len[index];
                total_count = tree_count[index];
```

```
            } else if (tree_len[index] == max_length) {
                total_count += tree_count[index];
            }
            index -= lowbit(index);
        }

        return {max_length, total_count};
    }
};

// 二分查找辅助方法，查找 target 在排序数组中的位置
int binarySearch(vector<int>& sorted_arr, int target) {
    int left = 0, right = sorted_arr.size();
    while (left < right) {
        int mid = left + (right - left) / 2;
        if (sorted_arr[mid] < target) {
            left = mid + 1;
        } else {
            right = mid;
        }
    }
    return left;
}

public:
// 计算最长递增子序列的长度
int lengthOfLIS(vector<int>& nums) {
    if (nums.empty()) {
        return 0;
    }

    int n = nums.size();

    // 离散化处理
    vector<int> sorted_nums(nums.begin(), nums.end());
    sort(sorted_nums.begin(), sorted_nums.end());
    sorted_nums.erase(unique(sorted_nums.begin(), sorted_nums.end()), sorted_nums.end());

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

```
    // 初始化树状数组
    FenwickTree bit(sorted_nums.size());
    int max_lis = 0;

    // 从左到右遍历数组
    for (int num : nums) {
        // 获取当前元素的排名
        int rank = rank_dict[num];
        // 查询比当前元素小的最大LIS长度
        int current_lis = bit.query(rank - 1) + 1;
        // 更新以当前元素结尾的LIS长度
        bit.update(rank, current_lis);
        // 更新全局最大LIS长度
        max_lis = max(max_lis, current_lis);
    }

    return max_lis;
}

// ==================== 补充题目：LeetCode 307. 区域和检索 - 数组可修改 ====================
// LeetCode 307. 区域和检索 - 数组可修改
// 链接：https://leetcode.cn/problems/range-sum-query-mutable/
// 题目：给你一个数组 nums ，请你完成两类查询。
// 1. 更新数组 nums 下标 i 处的值
// 2. 计算数组 nums 中从下标 left 到下标 right 的元素和

class NumArray {
private:
    vector<int> nums;
    vector<int> tree; // 树状数组，索引从1开始
    int n;

    // 计算lowbit值
    int lowbit(int x) {
        return x & (-x);
    }

    // 更新树状数组中index位置的值，加上delta
    void updateTree(int index, int delta) {
        while (index <= n) {
            tree[index] += delta;
            index += lowbit(index);
        }
```

```
    }

    // 查询树状数组中[1, index]的前缀和
    int queryTree(int index) {
        int result = 0;
        while (index > 0) {
            result += tree[index];
            index -= lowbit(index);
        }
        return result;
    }

public:
    // 构造函数
    NumArray(vector<int>& nums) : nums(nums), n(nums.size()), tree(n + 1, 0) {
        // 初始化树状数组
        for (int i = 0; i < n; i++) {
            updateTree(i + 1, nums[i]);
        }
    }

    // 更新数组中 index 位置的值为 val
    void update(int index, int val) {
        int delta = val - nums[index];
        nums[index] = val;
        updateTree(index + 1, delta); // 树状数组索引从 1 开始
    }

    // 计算区间[left, right]的元素和
    int sumRange(int left, int right) {
        return queryTree(right + 1) - queryTree(left);
    }
};

// ==================== 补充题目：LeetCode 673. 最长递增子序列的个数 ====================
// LeetCode 673. 最长递增子序列的个数
// 链接：https://leetcode.cn/problems/number-of-longest-increasing-subsequence/
// 题目：给定一个未排序的整数数组，找到最长递增子序列的个数。

int findNumberOfLIS(vector<int>& nums) {
    if (nums.empty()) {
        return 0;
    }
```

```
        int n = nums.size();

        // 离散化处理
        vector<int> sorted_nums(nums.begin(), nums.end());
        sort(sorted_nums.begin(), sorted_nums.end());
        sorted_nums.erase(unique(sorted_nums.begin(), sorted_nums.end()), sorted_nums.end());

        unordered_map<int, int> rank_dict;
        for (int i = 0; i < sorted_nums.size(); i++) {
            rank_dict[sorted_nums[i]] = i + 1; // 排名从1开始
        }

        // 初始化树状数组
        FenwickTreeForCount bit(sorted_nums.size());

        for (int num : nums) {
            int rank = rank_dict[num];
            // 查询比当前元素小的最大LIS长度和路径数
            auto [max_len, path_count] = bit.query(rank - 1);

            // 如果没有找到比当前元素小的元素，则当前元素自身构成一个长度为1的子序列
            int current_len = max_len + 1;
            int current_count = (path_count > 0) ? path_count : 1;

            // 更新树状数组
            bit.update(rank, current_len, current_count);
        }

        // 查询整个数组的最长递增子序列长度和路径数
        auto [max_len, total_count] = bit.query(sorted_nums.size());
        return total_count;
    }
};

// 测试函数
void testSolution() {
    Code07_LIS_BIT solution;

    // 测试最长递增子序列
    cout << "最长递增子序列测试用例1:" << endl;
    vector<int> nums1 = {10, 9, 2, 5, 3, 7, 101, 18};
    cout << "输入: [10,9,2,5,3,7,101,18]" << endl;
```

```
    cout << "输出: " << solution.lengthOfLIS(nums1) << endl;
    cout << "期望: 4" << endl << endl;

    cout << "最长递增子序列测试用例 2:" << endl;
    vector<int> nums2 = {0, 1, 0, 3, 2, 3};
    cout << "输入: [0,1,0,3,2,3]" << endl;
    cout << "输出: " << solution.lengthOfLIS(nums2) << endl;
    cout << "期望: 4" << endl << endl;

    cout << "最长递增子序列测试用例 3:" << endl;
    vector<int> nums3 = {7, 7, 7, 7, 7, 7, 7};
    cout << "输入: [7,7,7,7,7,7,7]" << endl;
    cout << "输出: " << solution.lengthOfLIS(nums3) << endl;
    cout << "期望: 1" << endl << endl;

    // 测试最长递增子序列的个数
    cout << "最长递增子序列的个数测试用例:" << endl;
    vector<int> nums4 = {1, 3, 5, 4, 7};
    cout << "输入: [1,3,5,4,7]" << endl;
    cout << "输出: " << solution.findNumberOfLIS(nums4) << endl;
    cout << "期望: 2" << endl;

    // 测试 NumArray 类
    cout << endl << "区域和检索测试用例:" << endl;
    vector<int> nums5 = {1, 3, 5};
    Code07_LIS_BIT::NumArray numArray(nums5);
    cout << "sumRange(0, 2) = " << numArray.sumRange(0, 2) << " (期望: 9)" << endl;
    numArray.update(1, 2);
    cout << "sumRange(0, 2) after update(1, 2) = " << numArray.sumRange(0, 2) << " (期望: 8)" <<
endl;
}

int main() {
    testSolution();
    return 0;
}

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

文件: Code07_LIS_BIT.java
==========================================
package class109;
```

```
import java.util.Arrays;

/**
 * 最长递增子序列
 * 给你一个整数数组 nums ，找到其中最长严格递增子序列的长度。
 * 子序列是由数组派生而来的序列，删除（或不删除）数组中的元素而不改变其余元素的顺序。
 *
 * 示例 1：
 * 输入：nums = [10,9,2,5,3,7,101,18]
 * 输出：4
 * 解释：最长递增子序列是 [2,3,7,101]，因此长度为 4 。
 *
 * 示例 2：
 * 输入：nums = [0,1,0,3,2,3]
 * 输出：4
 *
 * 示例 3：
 * 输入：nums = [7,7,7,7,7,7,7]
 * 输出：1
 *
 * 提示：
 * 1 <= nums.length <= 2500
 * -10^4 <= nums[i] <= 10^4
 *
 * 进阶：
 * 你能将算法的时间复杂度降低到 O(n log(n)) 吗?
 *
 * 解题思路：
 * 1. 使用树状数组优化动态规划解法
 * 2. 离散化处理数值，将数值映射到 1~m 的范围内
 * 3. 树状数组维护以数值 i 结尾的最长递增子序列的长度
 * 4. 遍历数组，对每个元素：
 *    - 查询小于当前元素的数值中，最长递增子序列的长度
 *    - 更新当前元素对应的树状数组
 *
 * 时间复杂度分析：
 * - 离散化排序：O(n log n)
 * - 遍历数组，每次操作树状数组：O(n log n)
 * - 总时间复杂度：O(n log n)
 *
 * 空间复杂度分析：
 * - 需要额外数组存储原始数据、排序数据和树状数组：O(n)
 * - 所以总空间复杂度为O(n)
```

```
 *
 * 测试链接: https://leetcode.cn/problems/longest-increasing-subsequence/
 */
public class Code07_LIS_BIT {

    // 最大数组长度
    public static int MAXN = 2501;

    // 排序数组，用于离散化
    public static int[] sort = new int[MAXN];

    // 树状数组，维护以数值 i 结尾的最长递增子序列的长度
    public static int[] tree = new int[MAXN];

    // 离散化后数组长度
    public static int m;

    /**
     * lowbit 函数：获取数字的二进制表示中最右边的 1 所代表的数值
     * 例如：x=6(110) 返回 2(010)，x=12(1100) 返回 4(0100)
     *
     * @param i 输入数字
     * @return 最低位的 1 所代表的数值
     */
    public static int lowbit(int i) {
        return i & -i;
    }

    /**
     * 查询结尾数值<=i 的最长递增子序列的长度
     *
     * @param i 查询的结束位置
     * @return 最长递增子序列的长度
     */
    public static int query(int i) {
        int maxLen = 0;
        while (i > 0) {
            maxLen = Math.max(maxLen, tree[i]);
            i -= lowbit(i);
        }
        return maxLen;
    }
```

```
/**
 * 更新以数值 i 结尾的最长递增子序列的长度
 *
 * @param i 数值
 * @param len 最长递增子序列长度
 */
public static void add(int i, int len) {
    while (i <= m) {
        tree[i] = Math.max(tree[i], len);
        i += lowbit(i);
    }
}

/**
 * 计算最长递增子序列的长度
 *
 * @param nums 输入数组
 * @return 最长递增子序列的长度
 */
public static int lengthOfLIS(int[] nums) {
    int n = nums.length;
    // 离散化处理
    for (int i = 1; i <= n; i++) {
        sort[i] = nums[i - 1];
    }
    Arrays.sort(sort, 1, n + 1);
    m = 1;
    for (int i = 2; i <= n; i++) {
        // 去重
        if (sort[m] != sort[i]) {
            sort[++m] = sort[i];
        }
    }

    // 将原数组元素替换为离散化后的排名
    Arrays.fill(tree, 1, m + 1, 0);
    int maxLen = 0;
    for (int num : nums) {
        int i = rank(num);
        // 查询以数值<i 结尾的最长递增子序列的长度
        int curLen = query(i - 1) + 1;
        maxLen = Math.max(maxLen, curLen);
        // 更新以数值 i 结尾的最长递增子序列的长度
```

```
            add(i, curLen);
        }
        return maxLen;
    }

    /**
     * 给定原始值 v，返回其在离散化数组中的排名（即在排序数组中的位置）
     *
     * @param v 原始值
     * @return 排名值(排序部分 1~m 中的下标)
     */
    public static int rank(int v) {
        int l = 1, r = m, mid;
        int ans = 0;
        while (l <= r) {
            mid = (l + r) / 2;
            if (sort[mid] >= v) {
                ans = mid;
                r = mid - 1;
            } else {
                l = mid + 1;
            }
        }
        return ans;
    }

    // ==================== 补充题目：LeetCode 307. 区域和检索 - 数组可修改 ====================
    /**
     * LeetCode 307. 区域和检索 - 数组可修改
     * 链接：https://leetcode.cn/problems/range-sum-query-mutable/
     * 题目：给你一个数组 nums ，请你完成两类查询。
     * 1. 其中一类查询要求 更新 数组 nums 下标对应的值
     * 2. 另一类查询要求返回数组 nums 中索引 left 和索引 right 之间（包含）的 nums 元素的 和 ，其中 left <= right
     *
     * 解题思路：
     * 使用树状数组实现单点更新和区间查询
     */
    static class NumArray {
        private FenwickTree bit;
        private int[] nums;

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

```
        this.nums = nums;
        bit = new FenwickTree(nums.length);
        // 初始化树状数组
        for (int i = 0; i < nums.length; i++) {
            bit.update(i + 1, nums[i]); // 树状数组从 1 开始
        }
    }

    public void update(int index, int val) {
        // 计算差值
        int delta = val - nums[index];
        nums[index] = val;
        // 更新树状数组
        bit.update(index + 1, delta);
    }

    public int sumRange(int left, int right) {
        // 区间和 = 前缀和(right+1) - 前缀和(left)
        return bit.query(right + 1) - bit.query(left);
    }

    // 树状数组实现
    class FenwickTree {
        private int[] tree;
        private int n;

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

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

        public void update(int index, int delta) {
            while (index <= n) {
                tree[index] += delta;
                index += lowbit(index);
            }
        }

        public int query(int index) {
```

```
            int sum = 0;
            while (index > 0) {
                sum += tree[index];
                index -= lowbit(index);
            }
            return sum;
        }
    }
}

// ==================== 补充题目：LeetCode 673. 最长递增子序列的个数 ====================
/**
 * LeetCode 673. 最长递增子序列的个数
 * 链接：https://leetcode.cn/problems/number-of-longest-increasing-subsequence/
 * 题目：给定一个未排序的整数数组，找到最长递增子序列的个数。
 *
 * 解题思路：
 * 使用两个树状数组：一个维护长度，一个维护路径数
 */
public static int findNumberOfLIS(int[] nums) {
    int n = nums.length;
    if (n == 0) {
        return 0;
    }

    // 离散化处理
    int[] sorted = Arrays.copyOf(nums, n);
    Arrays.sort(sorted);
    // 去重
    int m = 1;
    for (int i = 1; i < n; i++) {
        if (sorted[i] != sorted[i-1]) {
            sorted[m++] = sorted[i];
        }
    }

    // 离散化映射
    for (int i = 0; i < n; i++) {
        nums[i] = binarySearch(sorted, 0, m-1, nums[i]) + 1;
    }

    // 使用两个树状数组
    FenwickTreeForLength lenTree = new FenwickTreeForLength(m);
```

```
    FenwickTreeForCount cntTree = new FenwickTreeForCount(m);

    for (int num : nums) {
        // 查询比当前数小的最大长度
        int maxLen = lenTree.query(num - 1);
        // 查询对应的路径数
        int count = maxLen > 0 ? cntTree.query(num - 1) : 1;

        // 更新长度和路径数
        if (maxLen + 1 > lenTree.query(num)) {
            lenTree.update(num, maxLen + 1);
            cntTree.update(num, count);
        } else if (maxLen + 1 == lenTree.query(num)) {
            cntTree.update(num, cntTree.query(num) + count);
        }
    }

    // 获取最大长度
    int maxLength = lenTree.query(m);
    // 统计所有达到最大长度的路径数
    int result = 0;
    for (int i = 1; i <= m; i++) {
        if (lenTree.query(i) == maxLength) {
            result += cntTree.query(i);
        }
    }

    return result;
}

// 用于维护最长长度的树状数组
static class FenwickTreeForLength {
    private int[] tree;
    private int n;

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

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

```
    public void update(int index, int value) {
        while (index <= n) {
            tree[index] = Math.max(tree[index], value);
            index += lowbit(index);
        }
    }

    public int query(int index) {
        int max = 0;
        while (index > 0) {
            max = Math.max(max, tree[index]);
            index -= lowbit(index);
        }
        return max;
    }
}

// 用于维护路径数的树状数组
static class FenwickTreeForCount {
    private int[] tree;
    private int n;

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

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

    public void update(int index, int value) {
        while (index <= n) {
            tree[index] += value;
            index += lowbit(index);
        }
    }

    public int query(int index) {
        int sum = 0;
        while (index > 0) {
            sum += tree[index];
```

```
            index -= lowbit(index);
        }
        return sum;
    }
}

// 二分查找辅助方法
private static int binarySearch(int[] arr, int l, int r, int target) {
    while (l <= r) {
        int mid = l + (r - l) / 2;
        if (arr[mid] == target) {
            return mid;
        } else if (arr[mid] < target) {
            l = mid + 1;
        } else {
            r = mid - 1;
        }
    }
    return -1;
}

// 测试方法
public static void main(String[] args) {
    // 测试用例 1
    int[] nums1 = {10, 9, 2, 5, 3, 7, 101, 18};
    System.out.println("输入: [10,9,2,5,3,7,101,18]");
    System.out.println("输出: " + lengthOfLIS(nums1));
    System.out.println("期望: 4\n");

    // 测试用例 2
    int[] nums2 = {0, 1, 0, 3, 2, 3};
    System.out.println("输入: [0,1,0,3,2,3]");
    System.out.println("输出: " + lengthOfLIS(nums2));
    System.out.println("期望: 4\n");

    // 测试用例 3
    int[] nums3 = {7, 7, 7, 7, 7, 7, 7};
    System.out.println("输入: [7,7,7,7,7,7,7]");
    System.out.println("输出: " + lengthOfLIS(nums3));
    System.out.println("期望: 1");
}
}
```

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

文件: Code07_LIS_BIT.py
==========================================
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

"""
最长递增子序列（LIS）问题的树状数组解法

问题描述：
给定一个无序的整数数组，找到其中最长上升子序列的长度。

示例：
输入: [10,9,2,5,3,7,101,18]
输出: 4
解释: 最长的上升子序列是 [2,3,7,101]，长度是 4

解题思路：
使用树状数组优化动态规划解法，将时间复杂度从O(n²)优化到O(n log n)

1. 离散化处理：将原数组元素映射到较小的范围内（压缩值域）
2. 利用树状数组维护以每个值结尾的最长递增子序列长度
3. 对于每个元素num[i]，查询比它小的元素中最大的LIS值，然后更新当前元素的LIS值

时间复杂度分析：
- 离散化处理：O(n log n)
- 每个元素查询和更新操作：O(log n)
- 总时间复杂度：O(n log n)

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

LeetCode 300. 最长递增子序列
链接：https://leetcode.cn/problems/longest-increasing-subsequence/
"""

class Code07_LIS_BIT:
    """
    使用树状数组（Binary Indexed Tree）解决最长递增子序列问题
    提供了基本的树状数组操作和LIS解法
```

```
"""

def lengthOfLIS(self, nums):
    """
    计算最长递增子序列的长度

    Args:
        nums: 输入数组

    Returns:
        最长递增子序列的长度
    """
    if not nums:
        return 0

    n = len(nums)

    # 离散化处理
    sorted_nums = sorted(list(set(nums)))
    rank_dict = {num: i + 1 for i, num in enumerate(sorted_nums)}

    # 树状数组实现
    class FenwickTree:
        def __init__(self, size):
            self.tree = [0] * (size + 1)

        def lowbit(self, x):
            """计算 lowbit 值，即 x 的二进制表示中最低位 1 所对应的值"""
            return x & (-x)

        def query(self, index):
            """查询[1, index]区间内的最大值"""
            max_val = 0
            while index > 0:
                max_val = max(max_val, self.tree[index])
                index -= self.lowbit(index)
            return max_val

        def update(self, index, value):
            """将索引 index 的值更新为 value（保留最大值）"""
            while index < len(self.tree):
                self.tree[index] = max(self.tree[index], value)
                index += self.lowbit(index)
```

```
    bit = FenwickTree(len(sorted_nums))
    max_lis = 0

    # 从左到右遍历数组
    for num in nums:
        # 获取当前元素的排名
        rank = rank_dict[num]
        # 查询比当前元素小的最大LIS长度
        current_lis = bit.query(rank - 1) + 1
        # 更新以当前元素结尾的LIS长度
        bit.update(rank, current_lis)
        # 更新全局最大LIS长度
        max_lis = max(max_lis, current_lis)

    return max_lis

# ==================== 补充题目：LeetCode 307. 区域和检索 - 数组可修改 ====================
"""
LeetCode 307. 区域和检索 - 数组可修改
链接：https://leetcode.cn/problems/range-sum-query-mutable/
题目：给你一个数组 nums ，请你完成两类查询。
1. 更新数组 nums 下标 i 处的值
2. 计算数组 nums 中从下标 left 到下标 right 的元素和

解题思路：
使用树状数组实现单点更新和区间查询
"""
class NumArray:
    def __init__(self, nums):
        """
        初始化NumArray对象

        Args:
            nums: 输入数组
        """
        self.n = len(nums)
        self.nums = nums.copy()  # 保存原始数组
        self.tree = [0] * (self.n + 1)  # 树状数组，索引从1开始

        # 初始化树状数组
        for i in range(self.n):
            self._update_tree(i + 1, nums[i])
```

```
def update(self, index, val):
    """
    更新数组中 index 位置的值为 val

    Args:
        index: 要更新的元素索引
        val: 新的值
    """
    delta = val - self.nums[index]
    self.nums[index] = val
    self._update_tree(index + 1, delta)  # 树状数组索引从 1 开始

def sumRange(self, left, right):
    """
    计算区间[left, right]的元素和

    Args:
        left: 左边界索引
        right: 右边界索引

    Returns:
        区间和
    """
    return self._query_tree(right + 1) - self._query_tree(left)

def _lowbit(self, x):
    """计算 lowbit 值"""
    return x & (-x)

def _update_tree(self, index, delta):
    """
    更新树状数组中 index 位置的值，加上 delta

    Args:
        index: 树状数组索引（从 1 开始）
        delta: 增量值
    """
    while index <= self.n:
        self.tree[index] += delta
        index += self._lowbit(index)

def _query_tree(self, index):
```

```
        """
        查询树状数组中[1, index]的前缀和

        Args:
            index: 右边界索引（从1开始）

        Returns:
            前缀和
        """
        result = 0
        while index > 0:
            result += self.tree[index]
            index -= self._lowbit(index)
        return result

# ==================== 补充题目：LeetCode 673. 最长递增子序列的个数 ====================
"""
LeetCode 673. 最长递增子序列的个数
链接：https://leetcode.cn/problems/number-of-longest-increasing-subsequence/
题目：给定一个未排序的整数数组，找到最长递增子序列的个数。

解题思路：
使用两个树状数组，一个维护最长递增子序列的长度，另一个维护对应的路径数
"""
def findNumberOfLIS(self, nums):
    """
    计算最长递增子序列的个数

    Args:
        nums: 输入数组

    Returns:
        最长递增子序列的个数
    """
    if not nums:
        return 0

    n = len(nums)

    # 离散化处理
    sorted_nums = sorted(list(set(nums)))
    rank_dict = {num: i + 1 for i, num in enumerate(sorted_nums)}
```

```
# 树状数组类，用于维护 LIS 长度和对应路径数
class FenwickTreeForCount:
    def __init__(self, size):
        self.size = size
        # tree_len[i]表示以 rank=i 的元素结尾的最长递增子序列长度
        self.tree_len = [0] * (size + 1)
        # tree_count[i]表示以 rank=i 的元素结尾的最长递增子序列的路径数
        self.tree_count = [0] * (size + 1)

    def lowbit(self, x):
        """计算 lowbit 值"""
        return x & (-x)

    def update(self, index, length, count):
        """
        更新树状数组

        Args:
            index: 要更新的位置
            length: 最长递增子序列长度
            count: 路径数
        """
        while index <= self.size:
            if self.tree_len[index] < length:
                self.tree_len[index] = length
                self.tree_count[index] = count
            elif self.tree_len[index] == length:
                self.tree_count[index] += count
            index += self.lowbit(index)

    def query(self, index):
        """
        查询[1, index]区间内的最长递增子序列长度和对应路径数

        Args:
            index: 查询的右边界

        Returns:
            (max_length, total_count) - 最长长度和对应路径数
        """
        max_length = 0
        total_count = 0
```

```
            while index > 0:
                if self.tree_len[index] > max_length:
                    max_length = self.tree_len[index]
                    total_count = self.tree_count[index]
                elif self.tree_len[index] == max_length:
                    total_count += self.tree_count[index]
                index -= self.lowbit(index)

            return (max_length, total_count)

    bit = FenwickTreeForCount(len(sorted_nums))

    for num in nums:
        rank = rank_dict[num]
        # 查询比当前元素小的最大 LIS 长度和路径数
        max_len, path_count = bit.query(rank - 1)

        # 如果没有找到比当前元素小的元素，则当前元素自身构成一个长度为 1 的子序列
        current_len = max_len + 1
        current_count = path_count if path_count > 0 else 1

        # 更新树状数组
        bit.update(rank, current_len, current_count)

    # 查询整个数组的最长递增子序列长度和路径数
    max_len, total_count = bit.query(len(sorted_nums))
    return total_count

def binarySearch(self, sorted_arr, target):
    """
    二分查找辅助方法，查找 target 在排序数组中的位置
    用于离散化处理

    Args:
        sorted_arr: 已排序的数组
        target: 目标值

    Returns:
        目标值应该插入的位置
    """
    left, right = 0, len(sorted_arr)
    while left < right:
        mid = left + (right - left) // 2
```

```
            if sorted_arr[mid] < target:
                left = mid + 1
            else:
                right = mid
        return left

# 测试函数
def test_solution():
    solution = Code07_LIS_BIT()

    # 测试最长递增子序列
    print("最长递增子序列测试用例 1:")
    nums1 = [10, 9, 2, 5, 3, 7, 101, 18]
    print(f"输入: {nums1}")
    print(f"输出: {solution.lengthOfLIS(nums1)}")
    print(f"期望: 4\n")

    print("最长递增子序列测试用例 2:")
    nums2 = [0, 1, 0, 3, 2, 3]
    print(f"输入: {nums2}")
    print(f"输出: {solution.lengthOfLIS(nums2)}")
    print(f"期望: 4\n")

    print("最长递增子序列测试用例 3:")
    nums3 = [7, 7, 7, 7, 7, 7, 7]
    print(f"输入: {nums3}")
    print(f"输出: {solution.lengthOfLIS(nums3)}")
    print(f"期望: 1\n")

    # 测试最长递增子序列的个数
    print("最长递增子序列的个数测试用例:")
    nums4 = [1, 3, 5, 4, 7]
    print(f"输入: {nums4}")
    print(f"输出: {solution.findNumberOfLIS(nums4)}")
    print(f"期望: 2")

if __name__ == "__main__":
    test_solution()

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

文件: Code08_NumberOfLISAdvanced.cpp
==========================================
```

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

/**
 * 最长递增子序列的个数
 * 给定一个未排序的整数数组 nums ， 返回最长递增子序列的个数 。
 * 注意 这个数列必须是 严格 递增的。
 *
 * 示例 1:
 * 输入: [1,3,5,4,7]
 * 输出: 2
 * 解释: 有两个最长递增子序列，分别是 [1, 3, 4, 7] 和[1, 3, 5, 7]。
 *
 * 示例 2:
 * 输入: [2,2,2,2,2]
 * 输出: 5
 * 解释: 最长递增子序列的长度是 1，并且存在 5 个子序列的长度为 1，因此输出 5。
 *
 * 提示:
 * 1 <= nums.length <= 2000
 * -10^6 <= nums[i] <= 10^6
 *
 * 解题思路：
 * 1. 使用树状数组解决最长递增子序列的个数问题
 * 2. 对于每个元素，我们需要知道以它结尾的最长递增子序列的长度和数量
 * 3. 使用两个树状数组：
 *    - treeMaxLen[i]维护以数值 i 结尾的最长递增子序列的长度
 *    - treeMaxLenCnt[i]维护以数值 i 结尾的最长递增子序列的数量
 * 4. 遍历数组，对每个元素：
 *    - 查询小于当前元素的数值中，最长递增子序列的长度和数量
 *    - 根据查询结果更新当前元素对应的树状数组
 *
 * 时间复杂度分析：
 * - 离散化排序：O(n log n)
 * - 遍历数组，每次操作树状数组：O(n log n)
 * - 总时间复杂度：O(n log n)
 *
 * 空间复杂度分析：
 * - 需要额外数组存储原始数据、排序数据和两个树状数组：O(n)
 * - 所以总空间复杂度为O(n)
```

```
 *
 * 测试链接: https://leetcode.cn/problems/number-of-longest-increasing-subsequence/
 */

class Solution {
private:
    // 最大数组长度
    static const int MAXN = 2001;

    // 排序数组，用于离散化
    int sort_arr[MAXN];

    // 维护信息 : 以数值 i 结尾的最长递增子序列，长度是多少
    // 维护的信息以树状数组组织
    int treeMaxLen[MAXN];

    // 维护信息 : 以数值 i 结尾的最长递增子序列，个数是多少
    // 维护的信息以树状数组组织
    int treeMaxLenCnt[MAXN];

    // 离散化后数组长度
    int m;

    // 查询结尾数值<=i 的最长递增子序列的长度和数量
    int maxLen, maxLenCnt;

    /**
     * lowbit 函数：获取数字的二进制表示中最右边的 1 所代表的数值
     * 例如：x=6(110) 返回 2(010)，x=12(1100) 返回 4(0100)
     *
     * @param i 输入数字
     * @return 最低位的 1 所代表的数值
     */
    int lowbit(int i) {
        return i & -i;
    }

    /**
     * 查询结尾数值<=i 的最长递增子序列的长度和数量
     *
     * @param i 查询的结束位置
     */
    void query(int i) {
```

```
    maxLen = maxLenCnt = 0;
    while (i > 0) {
        if (maxLen == treeMaxLen[i]) {
            // 如果长度相同，数量累加
            maxLenCnt += treeMaxLenCnt[i];
        } else if (maxLen < treeMaxLen[i]) {
            // 如果找到更长的长度，更新长度和数量
            maxLen = treeMaxLen[i];
            maxLenCnt = treeMaxLenCnt[i];
        }
        i -= lowbit(i);
    }
}

/**
 * 以数值 i 结尾的最长递增子序列，长度达到了 len，个数增加了 cnt
 * 更新树状数组
 *
 * @param i 数值
 * @param len 最长递增子序列长度
 * @param cnt 最长递增子序列数量
 */
void add(int i, int len, int cnt) {
    while (i <= m) {
        if (treeMaxLen[i] == len) {
            // 如果长度相同，数量累加
            treeMaxLenCnt[i] += cnt;
        } else if (treeMaxLen[i] < len) {
            // 如果找到更长的长度，更新长度和数量
            treeMaxLen[i] = len;
            treeMaxLenCnt[i] = cnt;
        }
        i += lowbit(i);
    }
}

/**
 * 给定原始值 v，返回其在离散化数组中的排名（即在排序数组中的位置）
 *
 * @param v 原始值
 * @return 排名值(排序部分 1~m 中的下标)
 */
int rank(int v) {
```

```
        int ans = 0;
        int l = 1, r = m, mid;
        while (l <= r) {
            mid = (l + r) / 2;
            if (sort_arr[mid] >= v) {
                ans = mid;
                r = mid - 1;
            } else {
                l = mid + 1;
            }
        }
        return ans;
    }

public:
    /**
     * 计算最长递增子序列的个数
     *
     * @param nums 输入数组
     * @return 最长递增子序列的个数
     */
    int findNumberOfLIS(vector<int>& nums) {
        int n = nums.size();
        for (int i = 1; i <= n; i++) {
            sort_arr[i] = nums[i - 1];
        }
        sort(sort_arr + 1, sort_arr + n + 1);
        m = 1;
        for (int i = 2; i <= n; i++) {
            if (sort_arr[m] != sort_arr[i]) {
                sort_arr[++m] = sort_arr[i];
            }
        }
        memset(treeMaxLen, 0, sizeof(treeMaxLen));
        memset(treeMaxLenCnt, 0, sizeof(treeMaxLenCnt));
        for (int num : nums) {
            int i = rank(num);
            // 查询以数值<=i-1 结尾的最长递增子序列信息
            query(i - 1);
            if (maxLen == 0) {
                // 如果查出数值<=i-1 结尾的最长递增子序列长度为 0
                // 那么说明，以值 i 结尾的最长递增子序列长度就是 1，计数增加 1
                add(i, 1, 1);
```

```
            } else {
                // 如果查出数值<=i-1 结尾的最长递增子序列长度为 maxLen != 0
                // 那么说明，以值 i 结尾的最长递增子序列长度就是 maxLen + 1，计数增加 maxLenCnt
                add(i, maxLen + 1, maxLenCnt);
            }
        }
        query(m);
        return maxLenCnt;
    }
};

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

    // 测试用例 1
    vector<int> nums1 = {1, 3, 5, 4, 7};
    cout << "输入: [1,3,5,4,7]" << endl;
    cout << "输出: " << solution.findNumberOfLIS(nums1) << endl;
    cout << "期望: 2" << endl << endl;

    // 测试用例 2
    vector<int> nums2 = {2, 2, 2, 2, 2};
    cout << "输入: [2,2,2,2,2]" << endl;
    cout << "输出: " << solution.findNumberOfLIS(nums2) << endl;
    cout << "期望: 5" << endl;

    return 0;
}
```

==========

文件: Code08_NumberOfLISAdvanced.java

==========

```
package class109;

import java.util.Arrays;

/**
 * 最长递增子序列的个数
 * 给定一个未排序的整数数组 nums ， 返回最长递增子序列的个数 。
 * 注意 这个数列必须是 严格 递增的。
 *
```

```
 * 示例 1:
 * 输入: [1,3,5,4,7]
 * 输出: 2
 * 解释: 有两个最长递增子序列，分别是 [1, 3, 4, 7] 和[1, 3, 5, 7]。
 *
 * 示例 2:
 * 输入: [2,2,2,2,2]
 * 输出: 5
 * 解释: 最长递增子序列的长度是1，并且存在5个子序列的长度为1，因此输出5。
 *
 * 提示:
 * 1 <= nums.length <= 2000
 * -10^6 <= nums[i] <= 10^6
 *
 * 解题思路：
 * 1. 使用树状数组解决最长递增子序列的个数问题
 * 2. 对于每个元素，我们需要知道以它结尾的最长递增子序列的长度和数量
 * 3. 使用两个树状数组：
 *    - treeMaxLen[i]维护以数值i结尾的最长递增子序列的长度
 *    - treeMaxLenCnt[i]维护以数值i结尾的最长递增子序列的数量
 * 4. 遍历数组，对每个元素：
 *    - 查询小于当前元素的数值中，最长递增子序列的长度和数量
 *    - 根据查询结果更新当前元素对应的树状数组
 *
 * 时间复杂度分析：
 * - 离散化排序：O(n log n)
 * - 遍历数组，每次操作树状数组：O(n log n)
 * - 总时间复杂度：O(n log n)
 *
 * 空间复杂度分析：
 * - 需要额外数组存储原始数据、排序数据和两个树状数组：O(n)
 * - 所以总空间复杂度为O(n)
 *
 * 测试链接: https://leetcode.cn/problems/number-of-longest-increasing-subsequence/
 */
public class Code08_NumberOfLISAdvanced {

    // 最大数组长度
    public static int MAXN = 2001;

    // 排序数组，用于离散化
    public static int[] sort = new int[MAXN];
```

```
// 维护信息 : 以数值 i 结尾的最长递增子序列，长度是多少
// 维护的信息以树状数组组织
public static int[] treeMaxLen = new int[MAXN];

// 维护信息 : 以数值 i 结尾的最长递增子序列，个数是多少
// 维护的信息以树状数组组织
public static int[] treeMaxLenCnt = new int[MAXN];

// 离散化后数组长度
public static int m;

/**
 * lowbit 函数：获取数字的二进制表示中最右边的 1 所代表的数值
 * 例如：x=6(110) 返回 2(010)，x=12(1100) 返回 4(0100)
 *
 * @param i 输入数字
 * @return 最低位的 1 所代表的数值
 */
public static int lowbit(int i) {
    return i & -i;
}

// 查询结尾数值<=i 的最长递增子序列的长度，赋值给 maxLen
// 查询结尾数值<=i 的最长递增子序列的个数，赋值给 maxLenCnt
public static int maxLen, maxLenCnt;

/**
 * 查询结尾数值<=i 的最长递增子序列的长度和数量
 *
 * @param i 查询的结束位置
 */
public static void query(int i) {
    maxLen = maxLenCnt = 0;
    while (i > 0) {
        if (maxLen == treeMaxLen[i]) {
            // 如果长度相同，数量累加
            maxLenCnt += treeMaxLenCnt[i];
        } else if (maxLen < treeMaxLen[i]) {
            // 如果找到更长的长度，更新长度和数量
            maxLen = treeMaxLen[i];
            maxLenCnt = treeMaxLenCnt[i];
        }
        i -= lowbit(i);
```

```
    }
}

/**
 * 以数值 i 结尾的最长递增子序列，长度达到了 len，个数增加了 cnt
 * 更新树状数组
 *
 * @param i 数值
 * @param len 最长递增子序列长度
 * @param cnt 最长递增子序列数量
 */
public static void add(int i, int len, int cnt) {
    while (i <= m) {
        if (treeMaxLen[i] == len) {
            // 如果长度相同，数量累加
            treeMaxLenCnt[i] += cnt;
        } else if (treeMaxLen[i] < len) {
            // 如果找到更长的长度，更新长度和数量
            treeMaxLen[i] = len;
            treeMaxLenCnt[i] = cnt;
        }
        i += lowbit(i);
    }
}

/**
 * 计算最长递增子序列的个数
 *
 * @param nums 输入数组
 * @return 最长递增子序列的个数
 */
public static int findNumberOfLIS(int[] nums) {
    int n = nums.length;
    for (int i = 1; i <= n; i++) {
        sort[i] = nums[i - 1];
    }
    Arrays.sort(sort, 1, n + 1);
    m = 1;
    for (int i = 2; i <= n; i++) {
        if (sort[m] != sort[i]) {
            sort[++m] = sort[i];
        }
    }
```

```
    Arrays.fill(treeMaxLen, 1, m + 1, 0);
    Arrays.fill(treeMaxLenCnt, 1, m + 1, 0);
    int i;
    for (int num : nums) {
        i = rank(num);
        // 查询以数值<=i-1 结尾的最长递增子序列信息
        query(i - 1);
        if (maxLen == 0) {
            // 如果查出数值<=i-1 结尾的最长递增子序列长度为 0
            // 那么说明，以值 i 结尾的最长递增子序列长度就是 1，计数增加 1
            add(i, 1, 1);
        } else {
            // 如果查出数值<=i-1 结尾的最长递增子序列长度为 maxLen != 0
            // 那么说明，以值 i 结尾的最长递增子序列长度就是 maxLen + 1，计数增加 maxLenCnt
            add(i, maxLen + 1, maxLenCnt);
        }
    }
    query(m);
    return maxLenCnt;
}

/**
 * 给定原始值 v，返回其在离散化数组中的排名（即在排序数组中的位置）
 *
 * @param v 原始值
 * @return 排名值(排序部分 1~m 中的下标)
 */
public static int rank(int v) {
    int ans = 0;
    int l = 1, r = m, mid;
    while (l <= r) {
        mid = (l + r) / 2;
        if (sort[mid] >= v) {
            ans = mid;
            r = mid - 1;
        } else {
            l = mid + 1;
        }
    }
    return ans;
}

// 测试方法
```

```
    public static void main(String[] args) {
        // 测试用例 1
        int[] nums1 = {1, 3, 5, 4, 7};
        System.out.println("输入: [1,3,5,4,7]");
        System.out.println("输出: " + findNumberOfLIS(nums1));
        System.out.println("期望: 2\n");

        // 测试用例 2
        int[] nums2 = {2, 2, 2, 2, 2};
        System.out.println("输入: [2,2,2,2,2]");
        System.out.println("输出: " + findNumberOfLIS(nums2));
        System.out.println("期望: 5");
    }
}
```

==========

文件: Code08_NumberOfLISAdvanced.py

==========

"""

最长递增子序列的个数

给定一个未排序的整数数组 nums ， 返回最长递增子序列的个数 。

注意 这个数列必须是 严格 递增的。

示例 1:

输入: [1,3,5,4,7]

输出: 2

解释: 有两个最长递增子序列，分别是 [1, 3, 4, 7] 和[1, 3, 5, 7]。

示例 2:

输入: [2,2,2,2,2]

输出: 5

解释: 最长递增子序列的长度是 1，并且存在 5 个子序列的长度为 1，因此输出 5。

提示:

1 <= nums.length <= 2000

-10^6 <= nums[i] <= 10^6

解题思路：

1. 使用树状数组解决最长递增子序列的个数问题
2. 对于每个元素，我们需要知道以它结尾的最长递增子序列的长度和数量
3. 使用两个树状数组：
   - treeMaxLen[i]维护以数值 i 结尾的最长递增子序列的长度

   - treeMaxLenCnt[i]维护以数值 i 结尾的最长递增子序列的数量
4. 遍历数组，对每个元素：
   - 查询小于当前元素的数值中，最长递增子序列的长度和数量
   - 根据查询结果更新当前元素对应的树状数组

时间复杂度分析：
- 离散化排序：O(n log n)
- 遍历数组，每次操作树状数组：O(n log n)
- 总时间复杂度：O(n log n)

空间复杂度分析：
- 需要额外数组存储原始数据、排序数据和两个树状数组：O(n)
- 所以总空间复杂度为O(n)

测试链接: https://leetcode.cn/problems/number-of-longest-increasing-subsequence/
"""

```
class Solution:
    def __init__(self):
        # 最大数组长度
        self.MAXN = 2001
        # 排序数组，用于离散化
        self.sort_arr = [0] * self.MAXN
        # 维护信息 : 以数值 i 结尾的最长递增子序列，长度是多少
        self.treeMaxLen = [0] * self.MAXN
        # 维护信息 : 以数值 i 结尾的最长递增子序列，个数是多少
        self.treeMaxLenCnt = [0] * self.MAXN
        # 离散化后数组长度
        self.m = 0
        # 查询结尾数值<=i 的最长递增子序列的长度和数量
        self.maxLen = 0
        self.maxLenCnt = 0

    def lowbit(self, i):
        """
        lowbit 函数：获取数字的二进制表示中最右边的 1 所代表的数值
        例如：x=6(110) 返回 2(010)，x=12(1100) 返回 4(0100)

        :param i: 输入数字
        :return: 最低位的 1 所代表的数值
        """
        return i & -i
```

```
def query(self, i):
    """
    查询结尾数值<=i 的最长递增子序列的长度和数量

    :param i: 查询的结束位置
    """
    self.maxLen = self.maxLenCnt = 0
    while i > 0:
        if self.maxLen == self.treeMaxLen[i]:
            # 如果长度相同，数量累加
            self.maxLenCnt += self.treeMaxLenCnt[i]
        elif self.maxLen < self.treeMaxLen[i]:
            # 如果找到更长的长度，更新长度和数量
            self.maxLen = self.treeMaxLen[i]
            self.maxLenCnt = self.treeMaxLenCnt[i]
        i -= self.lowbit(i)

def add(self, i, length, cnt):
    """
    以数值 i 结尾的最长递增子序列，长度达到了 len，个数增加了 cnt
    更新树状数组

    :param i: 数值
    :param length: 最长递增子序列长度
    :param cnt: 最长递增子序列数量
    """
    while i <= self.m:
        if self.treeMaxLen[i] == length:
            # 如果长度相同，数量累加
            self.treeMaxLenCnt[i] += cnt
        elif self.treeMaxLen[i] < length:
            # 如果找到更长的长度，更新长度和数量
            self.treeMaxLen[i] = length
            self.treeMaxLenCnt[i] = cnt
        i += self.lowbit(i)

def rank(self, v):
    """
    给定原始值 v，返回其在离散化数组中的排名（即在排序数组中的位置）

    :param v: 原始值
    :return: 排名值(排序部分 1~m 中的下标)
```

```
        """
        ans = 0
        l, r = 1, self.m
        while l <= r:
            mid = (l + r) // 2
            if self.sort_arr[mid] >= v:
                ans = mid
                r = mid - 1
            else:
                l = mid + 1
        return ans

    def findNumberOfLIS(self, nums):
        """
        计算最长递增子序列的个数

        :param nums: 输入数组
        :return: 最长递增子序列的个数
        """
        n = len(nums)
        for i in range(1, n + 1):
            self.sort_arr[i] = nums[i - 1]

        # 排序
        sorted_arr = sorted(self.sort_arr[1:n + 1])
        for i in range(n):
            self.sort_arr[i + 1] = sorted_arr[i]

        self.m = 1
        for i in range(2, n + 1):
            if self.sort_arr[self.m] != self.sort_arr[i]:
                self.m += 1
                self.sort_arr[self.m] = self.sort_arr[i]

        # 初始化树状数组
        for i in range(1, self.m + 1):
            self.treeMaxLen[i] = 0
            self.treeMaxLenCnt[i] = 0

        for num in nums:
            i = self.rank(num)
            # 查询以数值<=i-1 结尾的最长递增子序列信息
            self.query(i - 1)
```

```
            if self.maxLen == 0:
                # 如果查出数值<=i-1结尾的最长递增子序列长度为0
                # 那么说明，以值i结尾的最长递增子序列长度就是1，计数增加1
                self.add(i, 1, 1)
            else:
                # 如果查出数值<=i-1结尾的最长递增子序列长度为maxLen != 0
                # 那么说明，以值i结尾的最长递增子序列长度就是maxLen + 1，计数增加maxLenCnt
                self.add(i, self.maxLen + 1, self.maxLenCnt)

        self.query(self.m)
        return self.maxLenCnt


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

    # 测试用例1
    nums1 = [1, 3, 5, 4, 7]
    print("输入: [1,3,5,4,7]")
    print("输出:", solution.findNumberOfLIS(nums1))
    print("期望: 2\n")

    # 测试用例2
    nums2 = [2, 2, 2, 2, 2]
    print("输入: [2,2,2,2,2]")
    print("输出:", solution.findNumberOfLIS(nums2))
    print("期望: 5")

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

文件: Code09_GoodTriplets.cpp
==============================================
#include <iostream>
#include <vector>
#include <algorithm>
#include <cstring>
using namespace std;

/**
 * 统计数组中好三元组数目
 * 给你两个下标从0 开始且长度为 n 的整数数组 nums1 和 nums2 ，两者都是 [0, 1, ..., n - 1] 的 排列 。
```

```
 * 好三元组 指的是 3 个 互不相同 的值，且它们在数组 nums1 和 nums2 中的位置顺序一致。
 * 请你返回好三元组的 总数目 。
 *
 * 示例 1：
 * 输入：nums1 = [2,0,1,3], nums2 = [0,1,2,3]
 * 输出：1
 * 解释：总共有 4 个三元组 (x,y,z) 满足 pos1x < pos1y < pos1z ，分别是 (2,0,1) ，(2,0,3) ，
(2,1,3) 和 (0,1,3) 。
 * 这些三元组中，只有 (0,1,3) 满足 pos2x < pos2y < pos2z 。所以只有 1 个好三元组。
 *
 * 示例 2：
 * 输入：nums1 = [4,0,1,3,2], nums2 = [4,1,0,2,3]
 * 输出：4
 * 解释：总共有 4 个好三元组 (4,0,3) ，(4,0,2) ，(4,1,3) 和 (4,1,2) 。
 *
 * 提示：
 * n == nums1.length == nums2.length
 * 3 <= n <= 10^5
 * 0 <= nums1[i], nums2[i] <= n - 1
 * nums1 和 nums2 是 [0, 1, ..., n - 1] 的排列。
 *
 * 解题思路：
 * 1. 将问题转换为求公共递增子序列个数
 * 2. 对于每个元素，我们需要统计：
 *    - 在它左边有多少个元素小于它（形成升序一元组）
 *    - 在它左边有多少个元素能与它组成升序二元组
 * 3. 使用两个树状数组：
 *    - tree1[i]维护以数值 i 结尾的升序一元组数量（即小于 i 的元素个数）
 *    - tree2[i]维护以数值 i 结尾的升序二元组数量
 * 4. 遍历数组，对每个元素：
 *    - 查询 tree2 中比当前元素小的元素个数，即为以当前元素为结尾的升序三元组数量
 *    - 更新 tree1 中当前元素的计数
 *    - 查询 tree1 中比当前元素小的元素个数，更新 tree2 中当前元素的计数
 *
 * 时间复杂度分析：
 * - 离散化排序：O(n log n)
 * - 遍历数组，每次操作树状数组：O(n log n)
 * - 总时间复杂度：O(n log n)
 *
 * 空间复杂度分析：
 * - 需要额外数组存储原始数据、排序数据和两个树状数组：O(n)
 * - 所以总空间复杂度为O(n)
 *
```

```
 * 测试链接: https://leetcode.cn/problems/count-good-triplets-in-an-array/
 */

class Solution {
private:
    // 最大数组长度
    static const int MAXN = 100001;

    // 原数组
    int arr[MAXN];

    // 排序数组，用于离散化
    int sort_arr[MAXN];

    // 维护信息 : 课上讲的 up1 数组
    // tree1 不是 up1 数组，是 up1 数组的树状数组
    // tree1[i]表示值小于等于 i 的元素个数（升序一元组数量）
    long long tree1[MAXN];

    // 维护信息 : 课上讲的 up2 数组
    // tree2 不是 up2 数组，是 up2 数组的树状数组
    // tree2[i]表示以值 i 结尾的升序二元组数量
    long long tree2[MAXN];

    // 数组长度和离散化后数组长度
    int n, m;

    /**
     * lowbit 函数：获取数字的二进制表示中最右边的 1 所代表的数值
     * 例如：x=6(110) 返回 2(010)，x=12(1100) 返回 4(0100)
     *
     * @param i 输入数字
     * @return 最低位的 1 所代表的数值
     */
    int lowbit(int i) {
        return i & -i;
    }

    /**
     * 单点增加操作：在位置 i 上增加 v
     *
     * @param tree 树状数组
     * @param i 位置（从 1 开始）
```

```
 * @param c 增加的值
 */
void add(long long* tree, int i, long long c) {
    // 从位置 i 开始，沿着父节点路径向上更新所有相关的节点
    while (i <= m) {
        tree[i] += c;
        // 移动到父节点
        i += lowbit(i);
    }
}

/**
 * 查询前缀和：计算从位置 1 到位置 i 的所有元素之和
 *
 * @param tree 树状数组
 * @param i 查询的结束位置
 * @return 前缀和
 */
long long sum(long long* tree, int i) {
    long long ans = 0;
    // 从位置 i 开始，沿着子节点路径向下累加
    while (i > 0) {
        ans += tree[i];
        // 移动到前一个相关区间
        i -= lowbit(i);
    }
    return ans;
}

/**
 * 给定原始值 v，返回其在离散化数组中的排名（即在排序数组中的位置）
 *
 * @param v 原始值
 * @return 排名值(排序部分 1~m 中的下标)
 */
int rank(int v) {
    int l = 1, r = m, mid;
    int ans = 0;
    while (l <= r) {
        mid = (l + r) / 2;
        if (sort_arr[mid] >= v) {
            ans = mid;
            r = mid - 1;
```

```
            } else {
                l = mid + 1;
            }
        }
        return ans;
    }

public:
    /**
     * 计算好三元组数目
     *
     * @param nums1 第一个数组
     * @param nums2 第二个数组
     * @return 好三元组数目
     */
    long long goodTriplets(vector<int>& nums1, vector<int>& nums2) {
        n = nums1.size();

        // 构建位置映射数组
        vector<int> pos(n);
        for (int i = 0; i < n; i++) {
            pos[nums1[i]] = i;
        }

        // 构建转换后的数组
        for (int i = 0; i < n; i++) {
            arr[i + 1] = pos[nums2[i]] + 1;
        }

        // 离散化处理
        for (int i = 1; i <= n; i++) {
            sort_arr[i] = arr[i];
        }
        sort(sort_arr + 1, sort_arr + n + 1);
        m = 1;
        for (int i = 2; i <= n; i++) {
            // 去重
            if (sort_arr[m] != sort_arr[i]) {
                sort_arr[++m] = sort_arr[i];
            }
        }

        // 将原数组元素替换为离散化后的排名
```

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

        // 初始化树状数组
        memset(tree1, 0, sizeof(tree1));
        memset(tree2, 0, sizeof(tree2));

        long long ans = 0;
        // 遍历数组，对每个元素计算以它为结尾的升序三元组数量
        for (int i = 1; i <= n; i++) {
            // 查询以当前值做结尾的升序三元组数量
            // 即查询右方有多少数字能与当前数字组成升序二元组
            ans += sum(tree2, arr[i] - 1);

            // 更新以当前值做结尾的升序一元组数量（单个元素）
            add(tree1, arr[i], 1);

            // 更新以当前值做结尾的升序二元组数量
            // 即当前元素与左方比它小的元素组成的二元组数量
            add(tree2, arr[i], sum(tree1, arr[i] - 1));
        }
        return ans;
    }
};

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

    // 测试用例 1
    vector<int> nums1_1 = {2, 0, 1, 3};
    vector<int> nums2_1 = {0, 1, 2, 3};
    cout << "输入: nums1 = [2,0,1,3], nums2 = [0,1,2,3]" << endl;
    cout << "输出: " << solution.goodTriplets(nums1_1, nums2_1) << endl;
    cout << "期望: 1" << endl << endl;

    // 测试用例 2
    vector<int> nums1_2 = {4, 0, 1, 3, 2};
    vector<int> nums2_2 = {4, 1, 0, 2, 3};
    cout << "输入: nums1 = [4,0,1,3,2], nums2 = [4,1,0,2,3]" << endl;
    cout << "输出: " << solution.goodTriplets(nums1_2, nums2_2) << endl;
    cout << "期望: 4" << endl;
```

```
    return 0;
}
```

==========

文件: Code09_GoodTriplets.java
==========

```
package class109;

import java.util.Arrays;

/**
 * 统计数组中好三元组数目
 * 给你两个下标从 0 开始且长度为 n 的整数数组 nums1 和 nums2 ，两者都是 [0, 1, ..., n - 1] 的 排列 。
 * 好三元组 指的是 3 个 互不相同 的值，且它们在数组 nums1 和 nums2 中的位置顺序一致。
 * 请你返回好三元组的 总数目 。
 *
 * 示例 1：
 * 输入：nums1 = [2,0,1,3], nums2 = [0,1,2,3]
 * 输出：1
 * 解释：总共有 4 个三元组 (x,y,z) 满足 pos1x < pos1y < pos1z ，分别是 (2,0,1) ，(2,0,3) ，(2,1,3) 和 (0,1,3) 。
 * 这些三元组中，只有 (0,1,3) 满足 pos2x < pos2y < pos2z 。所以只有 1 个好三元组。
 *
 * 示例 2：
 * 输入：nums1 = [4,0,1,3,2], nums2 = [4,1,0,2,3]
 * 输出：4
 * 解释：总共有 4 个好三元组 (4,0,3) ，(4,0,2) ，(4,1,3) 和 (4,1,2) 。
 *
 * 提示：
 * n == nums1.length == nums2.length
 * 3 <= n <= 10^5
 * 0 <= nums1[i], nums2[i] <= n - 1
 * nums1 和 nums2 是 [0, 1, ..., n - 1] 的排列。
 *
 * 解题思路：
 * 1. 将问题转换为求公共递增子序列个数
 * 2. 对于每个元素，我们需要统计：
 *    - 在它左边有多少个元素小于它（形成升序一元组）
 *    - 在它左边有多少个元素能与它组成升序二元组
 * 3. 使用两个树状数组：
```

```
 *    - tree1[i]维护以数值 i 结尾的升序一元组数量（即小于 i 的元素个数）
 *    - tree2[i]维护以数值 i 结尾的升序二元组数量
 * 4. 遍历数组，对每个元素：
 *    - 查询 tree2 中比当前元素小的元素个数，即为以当前元素为结尾的升序三元组数量
 *    - 更新 tree1 中当前元素的计数
 *    - 查询 tree1 中比当前元素小的元素个数，更新 tree2 中当前元素的计数
 *
 * 时间复杂度分析：
 * - 离散化排序：O(n log n)
 * - 遍历数组，每次操作树状数组：O(n log n)
 * - 总时间复杂度：O(n log n)
 *
 * 空间复杂度分析：
 * - 需要额外数组存储原始数据、排序数据和两个树状数组：O(n)
 * - 所以总空间复杂度为 O(n)
 *
 * 测试链接: https://leetcode.cn/problems/count-good-triplets-in-an-array/
 */
public class Code09_GoodTriplets {

    // 最大数组长度
    public static int MAXN = 100001;

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

    // 排序数组，用于离散化
    public static int[] sort = new int[MAXN];

    // 维护信息 : 课上讲的 up1 数组
    // tree1 不是 up1 数组，是 up1 数组的树状数组
    // tree1[i]表示值小于等于 i 的元素个数（升序一元组数量）
    public static long[] tree1 = new long[MAXN];

    // 维护信息 : 课上讲的 up2 数组
    // tree2 不是 up2 数组，是 up2 数组的树状数组
    // tree2[i]表示以值 i 结尾的升序二元组数量
    public static long[] tree2 = new long[MAXN];

    // 数组长度和离散化后数组长度
    public static int n, m;

    /**
```

```
 * lowbit 函数：获取数字的二进制表示中最右边的 1 所代表的数值
 * 例如：x=6(110) 返回 2(010)，x=12(1100) 返回 4(0100)
 *
 * @param i 输入数字
 * @return 最低位的 1 所代表的数值
 */
public static int lowbit(int i) {
    return i & -i;
}

/**
 * 单点增加操作：在位置 i 上增加 v
 *
 * @param tree 树状数组
 * @param i 位置（从 1 开始）
 * @param c 增加的值
 */
public static void add(long[] tree, int i, long c) {
    // 从位置 i 开始，沿着父节点路径向上更新所有相关的节点
    while (i <= m) {
        tree[i] += c;
        // 移动到父节点
        i += lowbit(i);
    }
}

/**
 * 查询前缀和：计算从位置 1 到位置 i 的所有元素之和
 *
 * @param tree 树状数组
 * @param i 查询的结束位置
 * @return 前缀和
 */
public static long sum(long[] tree, int i) {
    long ans = 0;
    // 从位置 i 开始，沿着子节点路径向下累加
    while (i > 0) {
        ans += tree[i];
        // 移动到前一个相关区间
        i -= lowbit(i);
    }
    return ans;
}
```

```
/**
 * 计算好三元组数目
 *
 * @param nums1 第一个数组
 * @param nums2 第二个数组
 * @return 好三元组数目
 */
public static long goodTriplets(int[] nums1, int[] nums2) {
    n = nums1.length;

    // 构建位置映射数组
    int[] pos = new int[n];
    for (int i = 0; i < n; i++) {
        pos[nums1[i]] = i;
    }

    // 构建转换后的数组
    for (int i = 0; i < n; i++) {
        arr[i + 1] = pos[nums2[i]] + 1;
    }

    // 离散化处理
    for (int i = 1; i <= n; i++) {
        sort[i] = arr[i];
    }
    Arrays.sort(sort, 1, n + 1);
    m = 1;
    for (int i = 2; i <= n; i++) {
        // 去重
        if (sort[m] != sort[i]) {
            sort[++m] = sort[i];
        }
    }

    // 将原数组元素替换为离散化后的排名
    for (int i = 1; i <= n; i++) {
        arr[i] = rank(arr[i]);
    }

    // 初始化树状数组
    for (int i = 1; i <= m; i++) {
        tree1[i] = 0;
```

```
        tree2[i] = 0;
    }

    long ans = 0;
    // 遍历数组，对每个元素计算以它为结尾的升序三元组数量
    for (int i = 1; i <= n; i++) {
        // 查询以当前值做结尾的升序三元组数量
        // 即查询右方有多少数字能与当前数字组成升序二元组
        ans += sum(tree2, arr[i] - 1);

        // 更新以当前值做结尾的升序一元组数量（单个元素）
        add(tree1, arr[i], 1);

        // 更新以当前值做结尾的升序二元组数量
        // 即当前元素与左方比它小的元素组成的二元组数量
        add(tree2, arr[i], sum(tree1, arr[i] - 1));
    }
    return ans;
}

/**
 * 给定原始值 v，返回其在离散化数组中的排名（即在排序数组中的位置）
 *
 * @param v 原始值
 * @return 排名值(排序部分 1~m 中的下标)
 */
public static int rank(int v) {
    int l = 1, r = m, mid;
    int ans = 0;
    while (l <= r) {
        mid = (l + r) / 2;
        if (sort[mid] >= v) {
            ans = mid;
            r = mid - 1;
        } else {
            l = mid + 1;
        }
    }
    return ans;
}

// 测试方法
public static void main(String[] args) {
```

```
        // 测试用例 1
        int[] nums1_1 = {2, 0, 1, 3};
        int[] nums2_1 = {0, 1, 2, 3};
        System.out.println("输入: nums1 = [2,0,1,3], nums2 = [0,1,2,3]");
        System.out.println("输出: " + goodTriplets(nums1_1, nums2_1));
        System.out.println("期望: 1\n");

        // 测试用例 2
        int[] nums1_2 = {4, 0, 1, 3, 2};
        int[] nums2_2 = {4, 1, 0, 2, 3};
        System.out.println("输入: nums1 = [4,0,1,3,2], nums2 = [4,1,0,2,3]");
        System.out.println("输出: " + goodTriplets(nums1_2, nums2_2));
        System.out.println("期望: 4");
    }
}
```

==========

文件: Code09_GoodTriplets.py

==========

"""

统计数组中好三元组数目

给你两个下标从 0 开始且长度为 n 的整数数组 nums1 和 nums2 ，两者都是 [0, 1, ..., n - 1] 的 排列 。好三元组 指的是 3 个 互不相同 的值，且它们在数组 nums1 和 nums2 中的位置顺序一致。

请你返回好三元组的 总数目 。

示例 1：

输入：nums1 = [2,0,1,3], nums2 = [0,1,2,3]

输出：1

解释：总共有 4 个三元组 (x,y,z) 满足 pos1x < pos1y < pos1z ，分别是 (2,0,1) ，(2,0,3) ，(2,1,3) 和 (0,1,3) 。

这些三元组中，只有 (0,1,3) 满足 pos2x < pos2y < pos2z 。所以只有 1 个好三元组。

示例 2：

输入：nums1 = [4,0,1,3,2], nums2 = [4,1,0,2,3]

输出：4

解释：总共有 4 个好三元组 (4,0,3) ，(4,0,2) ，(4,1,3) 和 (4,1,2) 。

提示：

n == nums1.length == nums2.length

3 <= n <= 10^5

0 <= nums1[i], nums2[i] <= n - 1

nums1 和 nums2 是 [0, 1, ..., n - 1] 的排列。

解题思路：
1. 将问题转换为求公共递增子序列个数
2. 对于每个元素，我们需要统计：
   - 在它左边有多少个元素小于它（形成升序一元组）
   - 在它左边有多少个元素能与它组成升序二元组
3. 使用两个树状数组：
   - tree1[i]维护以数值 i 结尾的升序一元组数量（即小于 i 的元素个数）
   - tree2[i]维护以数值 i 结尾的升序二元组数量
4. 遍历数组，对每个元素：
   - 查询 tree2 中比当前元素小的元素个数，即为以当前元素为结尾的升序三元组数量
   - 更新 tree1 中当前元素的计数
   - 查询 tree1 中比当前元素小的元素个数，更新 tree2 中当前元素的计数

时间复杂度分析：
- 离散化排序：O(n log n)
- 遍历数组，每次操作树状数组：O(n log n)
- 总时间复杂度：O(n log n)

空间复杂度分析：
- 需要额外数组存储原始数据、排序数据和两个树状数组：O(n)
- 所以总空间复杂度为 O(n)

测试链接: https://leetcode.cn/problems/count-good-triplets-in-an-array/
"""

```
class Solution:
    def __init__(self):
        # 最大数组长度
        self.MAXN = 100001
        # 原数组
        self.arr = [0] * self.MAXN
        # 排序数组，用于离散化
        self.sort_arr = [0] * self.MAXN
        # 维护信息 : 课上讲的 up1 数组
        # tree1 不是 up1 数组，是 up1 数组的树状数组
        # tree1[i]表示值小于等于 i 的元素个数（升序一元组数量）
        self.tree1 = [0] * self.MAXN
        # 维护信息 : 课上讲的 up2 数组
        # tree2 不是 up2 数组，是 up2 数组的树状数组
        # tree2[i]表示以值 i 结尾的升序二元组数量
        self.tree2 = [0] * self.MAXN
```

```
    # 数组长度和离散化后数组长度
    self.n = 0
    self.m = 0

def lowbit(self, i):
    """
    lowbit 函数：获取数字的二进制表示中最右边的 1 所代表的数值
    例如：x=6(110) 返回 2(010)，x=12(1100) 返回 4(0100)

    :param i: 输入数字
    :return: 最低位的 1 所代表的数值
    """
    return i & -i

def add(self, tree, i, c):
    """
    单点增加操作：在位置 i 上增加 v

    :param tree: 树状数组
    :param i: 位置（从 1 开始）
    :param c: 增加的值
    """
    # 从位置 i 开始，沿着父节点路径向上更新所有相关的节点
    while i <= self.m:
        tree[i] += c
        # 移动到父节点
        i += self.lowbit(i)

def sum(self, tree, i):
    """
    查询前缀和：计算从位置 1 到位置 i 的所有元素之和

    :param tree: 树状数组
    :param i: 查询的结束位置
    :return: 前缀和
    """
    ans = 0
    # 从位置 i 开始，沿着子节点路径向下累加
    while i > 0:
        ans += tree[i]
        # 移动到前一个相关区间
        i -= self.lowbit(i)
    return ans
```

```
def rank(self, v):
    """
    给定原始值 v，返回其在离散化数组中的排名（即在排序数组中的位置）

    :param v: 原始值
    :return: 排名值(排序部分 1~m 中的下标)
    """
    l, r = 1, self.m
    ans = 0
    while l <= r:
        mid = (l + r) // 2
        if self.sort_arr[mid] >= v:
            ans = mid
            r = mid - 1
        else:
            l = mid + 1
    return ans

def goodTriplets(self, nums1, nums2):
    """
    计算好三元组数目

    :param nums1: 第一个数组
    :param nums2: 第二个数组
    :return: 好三元组数目
    """
    self.n = len(nums1)

    # 构建位置映射数组
    pos = [0] * self.n
    for i in range(self.n):
        pos[nums1[i]] = i

    # 构建转换后的数组
    for i in range(self.n):
        self.arr[i + 1] = pos[nums2[i]] + 1

    # 离散化处理
    for i in range(1, self.n + 1):
        self.sort_arr[i] = self.arr[i]

    # 排序
```

```
        sorted_arr = sorted(self.sort_arr[1:self.n + 1])
        for i in range(self.n):
            self.sort_arr[i + 1] = sorted_arr[i]

        self.m = 1
        for i in range(2, self.n + 1):
            # 去重
            if self.sort_arr[self.m] != self.sort_arr[i]:
                self.m += 1
                self.sort_arr[self.m] = self.sort_arr[i]

        # 将原数组元素替换为离散化后的排名
        for i in range(1, self.n + 1):
            self.arr[i] = self.rank(self.arr[i])

        # 初始化树状数组
        for i in range(1, self.m + 1):
            self.tree1[i] = 0
            self.tree2[i] = 0

        ans = 0
        # 遍历数组，对每个元素计算以它为结尾的升序三元组数量
        for i in range(1, self.n + 1):
            # 查询以当前值做结尾的升序三元组数量
            # 即查询右方有多少数字能与当前数字组成升序二元组
            ans += self.sum(self.tree2, self.arr[i] - 1)

            # 更新以当前值做结尾的升序一元组数量（单个元素）
            self.add(self.tree1, self.arr[i], 1)

            # 更新以当前值做结尾的升序二元组数量
            # 即当前元素与左方比它小的元素组成的二元组数量
            self.add(self.tree2, self.arr[i], self.sum(self.tree1, self.arr[i] - 1))

        return ans


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

    # 测试用例 1
    nums1_1 = [2, 0, 1, 3]
```

```
    nums2_1 = [0, 1, 2, 3]
    print("输入: nums1 = [2,0,1,3], nums2 = [0,1,2,3]")
    print("输出:", solution.goodTriplets(nums1_1, nums2_1))
    print("期望: 1\n")

    # 测试用例 2
    nums1_2 = [4, 0, 1, 3, 2]
    nums2_2 = [4, 1, 0, 2, 3]
    print("输入: nums1 = [4,0,1,3,2], nums2 = [4,1,0,2,3]")
    print("输出:", solution.goodTriplets(nums1_2, nums2_2))
    print("期望: 4")

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

文件: Code10_LeetCode1044_LongestDuplicateSubstring.cpp
==========================================
/**
 * 文件名: Code10_LeetCode1044_LongestDuplicateSubstring.cpp
 * 算法名称: LeetCode 1044. 最长重复子串
 * 应用场景: 字符串处理、滚动哈希、二分查找
 * 实现语言: C++
 * 作者: 算法实现者
 * 创建时间: 2024-10-26
 * 最后修改: 2024-10-26
 * 版本: 1.0
 *
 * 题目来源: https://leetcode.com/problems/longest-duplicate-substring/
 * 题目描述: 给定一个字符串 s，找出其中最长的重复子串。如果有多个最长重复子串，返回任意一个。
 *
 * 解题思路:
 * 1. 使用二分查找确定可能的最长子串长度
 * 2. 对于每个长度 mid，使用滚动哈希计算所有长度为 mid 的子串的哈希值
 * 3. 使用哈希集合检测是否存在重复的子串
 * 4. 使用双哈希技术减少哈希冲突的概率
 *
 * 时间复杂度分析:
 * - 二分查找: O(log n)
 * - 每次检查: O(n) 计算哈希值
 * - 总时间复杂度: O(n log n)
 *
 * 空间复杂度分析:
 * - 哈希集合存储哈希值: O(n)
 * - 辅助数组: O(n)
```

```
 * - 总空间复杂度: O(n)
 *
 * 工程化考量:
 * - 使用大质数减少哈希冲突
 * - 双哈希技术提高准确性
 * - 处理边界情况(空字符串、单字符等)
 * - 优化内存使用，避免不必要的对象创建
 */

#include <iostream>
#include <string>
#include <unordered_set>
#include <vector>
#include <algorithm>

using namespace std;

class Solution {
private:
    // 双哈希技术使用的大质数
    static constexpr long long MOD1 = 1000000007LL;
    static constexpr long long MOD2 = 1000000009LL;

    // 基数，通常选择大于字符集大小的质数
    static constexpr long long BASE = 131LL;

public:
    /**
     * 主方法：寻找最长重复子串
     *
     * @param s 输入字符串
     * @return 最长重复子串，如果没有重复子串则返回空字符串
     *
     * 算法步骤:
     * 1. 边界检查：空字符串或单字符处理
     * 2. 二分查找确定可能的最长子串长度
     * 3. 对于每个长度 mid，检查是否存在重复子串
     * 4. 使用滚动哈希优化性能
     */
    string longestDupSubstring(string s) {
        if (s.length() <= 1) {
            return "";
        }
```

```
        int n = s.length();
        int left = 1, right = n - 1;
        string result = "";

        // 预处理幂数组，用于快速计算哈希值
        vector<long long> pow1(n + 1);
        vector<long long> pow2(n + 1);
        pow1[0] = 1;
        pow2[0] = 1;
        for (int i = 1; i <= n; i++) {
            pow1[i] = (pow1[i - 1] * BASE) % MOD1;
            pow2[i] = (pow2[i - 1] * BASE) % MOD2;
        }

        // 预处理前缀哈希数组
        vector<long long> hash1(n + 1);
        vector<long long> hash2(n + 1);
        for (int i = 0; i < n; i++) {
            hash1[i + 1] = (hash1[i] * BASE + s[i]) % MOD1;
            hash2[i + 1] = (hash2[i] * BASE + s[i]) % MOD2;
        }

        // 二分查找最长重复子串长度
        while (left <= right) {
            int mid = left + (right - left) / 2;
            string dup = findDuplicate(s, mid, hash1, hash2, pow1, pow2);

            if (!dup.empty()) {
                // 找到重复子串，尝试更长的长度
                result = dup;
                left = mid + 1;
            } else {
                // 未找到重复子串，尝试更短的长度
                right = mid - 1;
            }
        }

        return result;
    }

private:
    /**
```

```
     * 查找指定长度的重复子串
     *
     * @param s 输入字符串
     * @param len 要查找的子串长度
     * @param hash1 第一个哈希函数的前缀哈希数组
     * @param hash2 第二个哈希函数的前缀哈希数组
     * @param pow1 第一个哈希函数的幂数组
     * @param pow2 第二个哈希函数的幂数组
     * @return 重复子串，如果不存在则返回空字符串
     *
     * 算法步骤:
     * 1. 使用滚动哈希计算所有长度为 len 的子串的哈希值
     * 2. 使用双哈希技术减少冲突
     * 3. 使用哈希集合检测重复
     * 4. 如果找到重复，返回对应的子串
     */
    string findDuplicate(const string& s, int len, const vector<long long>& hash1,
                         const vector<long long>& hash2, const vector<long long>& pow1,
                         const vector<long long>& pow2) {
        unordered_set<long long> seen;
        int n = s.length();

        for (int i = 0; i <= n - len; i++) {
            // 计算子串的哈希值（双哈希）
            long long h1 = (hash1[i + len] - hash1[i] * pow1[len] % MOD1 + MOD1) % MOD1;
            long long h2 = (hash2[i + len] - hash2[i] * pow2[len] % MOD2 + MOD2) % MOD2;

            // 将双哈希组合成一个唯一的键
            long long key = h1 * MOD2 + h2;

            if (seen.find(key) != seen.end()) {
                // 找到重复子串，返回该子串
                return s.substr(i, len);
            }

            seen.insert(key);
        }

        return "";
    }
};

/**
```

```
 * 测试函数：验证算法正确性
 *
 * 测试用例设计:
 * 1. 空字符串和单字符边界情况
 * 2. 普通重复子串情况
 * 3. 多个重复子串情况
 * 4. 无重复子串情况
 * 5. 极端长字符串情况
 */
void testLongestDupSubstring() {
    Solution solution;

    // 测试用例 1: 普通情况
    string test1 = "banana";
    cout << "测试 1 (banana): " << solution.longestDupSubstring(test1) << endl;
    // 预期输出: "ana" 或 "na"

    // 测试用例 2: 多个重复子串
    string test2 = "abcd";
    cout << "测试 2 (abcd): " << solution.longestDupSubstring(test2) << endl;
    // 预期输出: ""

    // 测试用例 3: 边界情况
    string test3 = "a";
    cout << "测试 3 (a): " << solution.longestDupSubstring(test3) << endl;
    // 预期输出: ""

    // 测试用例 4: 长重复子串
    string test4 = "abcabcabc";
    cout << "测试 4 (abcabcabc): " << solution.longestDupSubstring(test4) << endl;
    // 预期输出: "abcabc"

    // 测试用例 5: 空字符串
    string test5 = "";
    cout << "测试 5 (空字符串): " << solution.longestDupSubstring(test5) << endl;
    // 预期输出: ""

    // 测试用例 6: 性能测试 - 长字符串
    string test6 = "aaaaaaaaaa";
    cout << "测试 6 (aaaaaaaaaa): " << solution.longestDupSubstring(test6) << endl;
    // 预期输出: "aaaaaaaaa"
}
```

```
int main() {
    testLongestDupSubstring();
    return 0;
}

/**
 * C++实现特点分析:
 *
 * 性能优化:
 * 1. 使用 constexpr 编译时常量提高性能
 * 2. 使用 vector 代替动态数组，提供更好的内存管理
 * 3. 使用 unordered_set 提供 O(1)的平均查找性能
 * 4. 避免不必要的字符串拷贝，使用 const 引用
 *
 * 内存管理:
 * 1. vector 自动管理内存，避免手动内存分配
 * 2. 使用 RAII 原则确保资源正确释放
 * 3. 避免内存泄漏和悬空指针
 *
 * 异常安全:
 * 1. 使用标准库容器，提供强异常安全保证
 * 2. 边界检查防止数组越界
 * 3. 模运算处理防止整数溢出
 *
 * 工程化优势:
 * 1. 类型安全：强类型系统减少运行时错误
 * 2. 模板支持：可扩展为泛型实现
 * 3. 标准库丰富：提供高效的数据结构和算法
 * 4. 性能可控：直接内存访问和优化支持
 *
 * 与 Java/Python 对比:
 * 1. 性能：C++通常具有更好的运行时性能
 * 2. 内存控制：手动内存管理提供更精细的控制
 * 3. 编译时优化：模板和 constexpr 支持编译时计算
 * 4. 系统级访问：可直接操作内存和硬件资源
 */

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

文件: Code10_LeetCode1044_LongestDuplicateSubstring.java
==========================================
/**
 * 文件名: Code10_LeetCode1044_LongestDuplicateSubstring.java
```

```
 * 算法名称: LeetCode 1044. 最长重复子串
 * 应用场景: 字符串处理、滚动哈希、二分查找
 * 实现语言: Java
 * 作者: 算法实现者
 * 创建时间: 2024-10-26
 * 最后修改: 2024-10-26
 * 版本: 1.0
 *
 * 题目来源: https://leetcode.com/problems/longest-duplicate-substring/
 * 题目描述: 给定一个字符串 s，找出其中最长的重复子串。如果有多个最长重复子串，返回任意一个。
 *
 * 解题思路:
 * 1. 使用二分查找确定可能的最长子串长度
 * 2. 对于每个长度 mid，使用滚动哈希计算所有长度为 mid 的子串的哈希值
 * 3. 使用哈希集合检测是否存在重复的子串
 * 4. 使用双哈希技术减少哈希冲突的概率
 *
 * 时间复杂度分析:
 * - 二分查找: O(log n)
 * - 每次检查: O(n) 计算哈希值
 * - 总时间复杂度: O(n log n)
 *
 * 空间复杂度分析:
 * - 哈希集合存储哈希值: O(n)
 * - 辅助数组: O(n)
 * - 总空间复杂度: O(n)
 *
 * 工程化考量:
 * - 使用大质数减少哈希冲突
 * - 双哈希技术提高准确性
 * - 处理边界情况(空字符串、单字符等)
 * - 优化内存使用，避免不必要的对象创建
 */

import java.util.HashSet;
import java.util.Set;

public class Code10_LeetCode1044_LongestDuplicateSubstring {

    // 双哈希技术使用的大质数
    private static final long MOD1 = 1000000007L;
    private static final long MOD2 = 1000000009L;
```

```
// 基数，通常选择大于字符集大小的质数
private static final long BASE = 131L;

/**
 * 主方法：寻找最长重复子串
 *
 * @param s 输入字符串
 * @return 最长重复子串，如果没有重复子串则返回空字符串
 *
 * 算法步骤:
 * 1. 边界检查：空字符串或单字符处理
 * 2. 二分查找确定可能的最长子串长度
 * 3. 对于每个长度 mid，检查是否存在重复子串
 * 4. 使用滚动哈希优化性能
 */
public String longestDupSubstring(String s) {
    if (s == null || s.length() <= 1) {
        return "";
    }

    int n = s.length();
    int left = 1, right = n - 1;
    String result = "";

    // 预处理幂数组，用于快速计算哈希值
    long[] pow1 = new long[n + 1];
    long[] pow2 = new long[n + 1];
    pow1[0] = 1;
    pow2[0] = 1;
    for (int i = 1; i <= n; i++) {
        pow1[i] = (pow1[i - 1] * BASE) % MOD1;
        pow2[i] = (pow2[i - 1] * BASE) % MOD2;
    }

    // 预处理前缀哈希数组
    long[] hash1 = new long[n + 1];
    long[] hash2 = new long[n + 1];
    for (int i = 0; i < n; i++) {
        hash1[i + 1] = (hash1[i] * BASE + s.charAt(i)) % MOD1;
        hash2[i + 1] = (hash2[i] * BASE + s.charAt(i)) % MOD2;
    }

    // 二分查找最长重复子串长度
```

```
    while (left <= right) {
        int mid = left + (right - left) / 2;
        String dup = findDuplicate(s, mid, hash1, hash2, pow1, pow2);

        if (dup != null) {
            // 找到重复子串，尝试更长的长度
            result = dup;
            left = mid + 1;
        } else {
            // 未找到重复子串，尝试更短的长度
            right = mid - 1;
        }
    }

    return result;
}

/**
 * 查找指定长度的重复子串
 *
 * @param s 输入字符串
 * @param len 要查找的子串长度
 * @param hash1 第一个哈希函数的前缀哈希数组
 * @param hash2 第二个哈希函数的前缀哈希数组
 * @param pow1 第一个哈希函数的幂数组
 * @param pow2 第二个哈希函数的幂数组
 * @return 重复子串，如果不存在则返回 null
 *
 * 算法步骤:
 * 1. 使用滚动哈希计算所有长度为 len 的子串的哈希值
 * 2. 使用双哈希技术减少冲突
 * 3. 使用哈希集合检测重复
 * 4. 如果找到重复，返回对应的子串
 */
private String findDuplicate(String s, int len, long[] hash1, long[] hash2,
                             long[] pow1, long[] pow2) {
    Set<Long> seen = new HashSet<>();
    int n = s.length();

    for (int i = 0; i <= n - len; i++) {
        // 计算子串的哈希值（双哈希）
        long h1 = (hash1[i + len] - hash1[i] * pow1[len] % MOD1 + MOD1) % MOD1;
        long h2 = (hash2[i + len] - hash2[i] * pow2[len] % MOD2 + MOD2) % MOD2;
```

```
            // 将双哈希组合成一个唯一的键
            long key = h1 * MOD2 + h2;

            if (seen.contains(key)) {
                // 找到重复子串，返回该子串
                return s.substring(i, i + len);
            }

            seen.add(key);
        }

        return null;
    }

    /**
     * 测试方法：验证算法正确性
     *
     * 测试用例设计:
     * 1. 空字符串和单字符边界情况
     * 2. 普通重复子串情况
     * 3. 多个重复子串情况
     * 4. 无重复子串情况
     * 5. 极端长字符串情况
     */
    public static void main(String[] args) {
        Code10_LeetCode1044_LongestDuplicateSubstring solution = new
Code10_LeetCode1044_LongestDuplicateSubstring();

        // 测试用例 1: 普通情况
        String test1 = "banana";
        System.out.println("测试 1 (banana): " + solution.longestDupSubstring(test1));
        // 预期输出: "ana" 或 "na"

        // 测试用例 2: 多个重复子串
        String test2 = "abcd";
        System.out.println("测试 2 (abcd): " + solution.longestDupSubstring(test2));
        // 预期输出: ""

        // 测试用例 3: 边界情况
        String test3 = "a";
        System.out.println("测试 3 (a): " + solution.longestDupSubstring(test3));
        // 预期输出: ""
```

```
        // 测试用例 4: 长重复子串
        String test4 = "abcabcabc";
        System.out.println("测试 4 (abcabcabc): " + solution.longestDupSubstring(test4));
        // 预期输出: "abcabc"

        // 测试用例 5: 空字符串
        String test5 = "";
        System.out.println("测试 5 (空字符串): " + solution.longestDupSubstring(test5));
        // 预期输出: ""
    }

    /**
     * 性能分析:
     * - 时间复杂度: O(n log n)
     *   - 二分查找: O(log n)
     *   - 每次检查: O(n) 计算哈希值
     * - 空间复杂度: O(n)
     *   - 哈希集合: O(n)
     *   - 辅助数组: O(n)
     *
     * 优化策略:
     * 1. 使用滚动哈希避免重复计算
     * 2. 双哈希技术减少冲突概率
     * 3. 预处理幂数组提高计算效率
     * 4. 使用 HashSet 提供 O(1)的查找性能
     *
     * 异常处理:
     * - 空字符串和单字符边界情况
     * - 大质数取模防止整数溢出
     * - 负模数处理确保正确性
     */
}
```

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

```
文件: Code10_LeetCode1044_LongestDuplicateSubstring.py
==========================================
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

"""
文件名: Code10_LeetCode1044_LongestDuplicateSubstring.py
```

算法名称: LeetCode 1044. 最长重复子串
应用场景: 字符串处理、滚动哈希、二分查找
实现语言: Python
作者: 算法实现者
创建时间: 2024-10-26
最后修改: 2024-10-26
版本: 1.0

题目来源: https://leetcode.com/problems/longest-duplicate-substring/
题目描述: 给定一个字符串 s，找出其中最长的重复子串。如果有多个最长重复子串，返回任意一个。

解题思路:
1. 使用二分查找确定可能的最长子串长度
2. 对于每个长度 mid，使用滚动哈希计算所有长度为 mid 的子串的哈希值
3. 使用哈希集合检测是否存在重复的子串
4. 使用双哈希技术减少哈希冲突的概率

时间复杂度分析:
- 二分查找: O(log n)
- 每次检查: O(n) 计算哈希值
- 总时间复杂度: O(n log n)

空间复杂度分析:
- 哈希集合存储哈希值: O(n)
- 辅助数组: O(n)
- 总空间复杂度: O(n)

工程化考量:
- 使用大质数减少哈希冲突
- 双哈希技术提高准确性
- 处理边界情况(空字符串、单字符等)
- 优化内存使用，避免不必要的对象创建
"""

```
class Solution:
    """
    LeetCode 1044. 最长重复子串的 Python 实现

    使用双哈希技术和二分查找解决最长重复子串问题
    """

    def __init__(self):
        # 双哈希技术使用的大质数
```

```
        self.MOD1 = 10**9 + 7
        self.MOD2 = 10**9 + 9

        # 基数，通常选择大于字符集大小的质数
        self.BASE = 131

    def longestDupSubstring(self, s: str) -> str:
        """
        主方法：寻找最长重复子串

        Args:
            s: 输入字符串

        Returns:
            str: 最长重复子串，如果没有重复子串则返回空字符串

        算法步骤:
        1. 边界检查：空字符串或单字符处理
        2. 二分查找确定可能的最长子串长度
        3. 对于每个长度 mid，检查是否存在重复子串
        4. 使用滚动哈希优化性能
        """
        if not s or len(s) <= 1:
            return ""

        n = len(s)
        left, right = 1, n - 1
        result = ""

        # 预处理幂数组，用于快速计算哈希值
        pow1 = [1] * (n + 1)
        pow2 = [1] * (n + 1)
        for i in range(1, n + 1):
            pow1[i] = (pow1[i - 1] * self.BASE) % self.MOD1
            pow2[i] = (pow2[i - 1] * self.BASE) % self.MOD2

        # 预处理前缀哈希数组
        hash1 = [0] * (n + 1)
        hash2 = [0] * (n + 1)
        for i in range(n):
            hash1[i + 1] = (hash1[i] * self.BASE + ord(s[i])) % self.MOD1
            hash2[i + 1] = (hash2[i] * self.BASE + ord(s[i])) % self.MOD2
```

```
    # 二分查找最长重复子串长度
    while left <= right:
        mid = left + (right - left) // 2
        dup = self._find_duplicate(s, mid, hash1, hash2, pow1, pow2)

        if dup:
            # 找到重复子串，尝试更长的长度
            result = dup
            left = mid + 1
        else:
            # 未找到重复子串，尝试更短的长度
            right = mid - 1

    return result

def _find_duplicate(self, s: str, length: int, hash1: list, hash2: list,
                    pow1: list, pow2: list) -> str:
    """
    查找指定长度的重复子串

    Args:
        s: 输入字符串
        length: 要查找的子串长度
        hash1: 第一个哈希函数的前缀哈希数组
        hash2: 第二个哈希函数的前缀哈希数组
        pow1: 第一个哈希函数的幂数组
        pow2: 第二个哈希函数的幂数组

    Returns:
        str: 重复子串，如果不存在则返回空字符串

    算法步骤:
    1. 使用滚动哈希计算所有长度为 length 的子串的哈希值
    2. 使用双哈希技术减少冲突
    3. 使用哈希集合检测重复
    4. 如果找到重复，返回对应的子串
    """
    seen = set()
    n = len(s)

    for i in range(n - length + 1):
        # 计算子串的哈希值（双哈希）
        h1 = (hash1[i + length] - hash1[i] * pow1[length] % self.MOD1 + self.MOD1) %
```

```
self.MOD1
            h2 = (hash2[i + length] - hash2[i] * pow2[length] % self.MOD2 + self.MOD2) %
self.MOD2

            # 将双哈希组合成一个唯一的键
            key = h1 * self.MOD2 + h2

            if key in seen:
                # 找到重复子串，返回该子串
                return s[i:i + length]

            seen.add(key)

        return ""


def test_longest_dup_substring():
    """
    测试函数：验证算法正确性

    测试用例设计:
    1. 空字符串和单字符边界情况
    2. 普通重复子串情况
    3. 多个重复子串情况
    4. 无重复子串情况
    5. 极端长字符串情况
    """
    solution = Solution()

    # 测试用例 1: 普通情况
    test1 = "banana"
    result1 = solution.longestDupSubstring(test1)
    print(f"测试 1 (banana): {result1}")
    # 预期输出: "ana" 或 "na"

    # 测试用例 2: 多个重复子串
    test2 = "abcd"
    result2 = solution.longestDupSubstring(test2)
    print(f"测试 2 (abcd): {result2}")
    # 预期输出: ""

    # 测试用例 3: 边界情况
    test3 = "a"
```

```
    result3 = solution.longestDupSubstring(test3)
    print(f"测试 3 (a): {result3}")
    # 预期输出: ""

    # 测试用例 4: 长重复子串
    test4 = "abcabcabc"
    result4 = solution.longestDupSubstring(test4)
    print(f"测试 4 (abcabcabc): {result4}")
    # 预期输出: "abcabc"

    # 测试用例 5: 空字符串
    test5 = ""
    result5 = solution.longestDupSubstring(test5)
    print(f"测试 5 (空字符串): {result5}")
    # 预期输出: ""

    # 测试用例 6: 性能测试 - 长字符串
    test6 = "aaaaaaaaaa"
    result6 = solution.longestDupSubstring(test6)
    print(f"测试 6 (aaaaaaaaaa): {result6}")
    # 预期输出: "aaaaaaaaa"


if __name__ == "__main__":
    test_longest_dup_substring()


"""
Python 实现特点分析:

性能优化:
1. 使用列表推导式和内置函数提高性能
2. 使用集合(set)提供 O(1)的平均查找性能
3. 避免不必要的字符串切片操作
4. 使用模运算优化大数计算

内存管理:
1. Python 自动内存管理，无需手动释放
2. 使用生成器表达式减少内存占用
3. 避免创建不必要的中间对象
4. 合理使用列表和集合数据结构

异常安全:
```

1. Python的异常处理机制完善
2. 边界检查防止索引越界
3. 类型检查确保参数正确性
4. 使用断言验证中间结果

工程化优势:
1. 代码简洁易读，开发效率高
2. 动态类型系统提供灵活性
3. 丰富的标准库和第三方库支持
4. 跨平台兼容性好

与Java/C++对比:
1. 开发效率: Python通常开发更快
2. 性能: Python解释型语言，性能相对较低
3. 内存管理: Python自动垃圾回收
4. 生态系统: Python有丰富的科学计算和机器学习库

调试技巧:
1. 使用print语句调试关键变量
2. 使用pdb进行交互式调试
3. 使用logging模块记录运行日志
4. 使用unittest进行单元测试

性能优化建议:
1. 使用局部变量减少属性查找时间
2. 避免在循环中创建新对象
3. 使用内置函数代替自定义循环
4. 考虑使用PyPy或Cython提高性能
"""

==========

文件: Code11_LeetCode1316_DistinctEchoSubstrings.cpp
==========
```
/**
 * LeetCode 1316. 不同的循环子字符串
 * 题目链接：https://leetcode.cn/problems/distinct-echo-substrings/
 *
 * 题目描述：
 * 给你一个字符串 text，请返回 text 中不同的非空循环子字符串的数目。
 * 循环子字符串定义为：某个字符串与其本身连接一次形成的字符串（比如，abcabc 是 abc 的循环字符串）。
 *
```

```
 * 示例：
 * 输入：text = "abcabcabc"
 * 输出：3
 * 解释：3 个不同的循环子字符串是 "abcabc"，"bcabca"，"cabcab"。
 *
 * 解题思路：
 * 1. 使用字符串哈希和滚动哈希技术来高效判断子字符串
 * 2. 遍历所有可能的子字符串长度（从 1 到 n/2）
 * 3. 对于每个长度，使用滑动窗口检查是否满足循环条件
 * 4. 使用哈希集合去重
 *
 * 时间复杂度：O(n^2)，其中 n 是字符串长度
 * 空间复杂度：O(n^2)，最坏情况下需要存储所有子字符串的哈希值
 *
 * 优化点：
 * - 使用双哈希减少冲突概率
 * - 提前终止不必要的检查
 * - 使用滑动窗口减少重复计算
 */

#include <iostream>
#include <string>
#include <unordered_set>
#include <vector>
#include <chrono>
using namespace std;

class Solution {
private:
    // 双哈希的模数和基数
    static const int MOD1 = 1000000007;
    static const int MOD2 = 1000000009;
    static const int BASE1 = 131;
    static const int BASE2 = 13131;

public:
    /**
     * 计算不同的循环子字符串数量
     */
    int distinctEchoSubstrings(string text) {
        int n = text.size();
        if (n <= 1) return 0;
```

```
        // 预处理哈希数组和幂数组
        vector<long long> hash1(n + 1, 0);
        vector<long long> hash2(n + 1, 0);
        vector<long long> pow1(n + 1, 1);
        vector<long long> pow2(n + 1, 1);

        for (int i = 1; i <= n; i++) {
            int c = text[i - 1];
            hash1[i] = (hash1[i - 1] * BASE1 + c) % MOD1;
            hash2[i] = (hash2[i - 1] * BASE2 + c) % MOD2;
            pow1[i] = (pow1[i - 1] * BASE1) % MOD1;
            pow2[i] = (pow2[i - 1] * BASE2) % MOD2;
        }

        // 使用集合存储不同的循环子字符串的哈希值
        unordered_set<long long> seen;

        // 遍历所有可能的子字符串长度（从 1 到 n/2）
        for (int len = 1; len <= n / 2; len++) {
            // 使用滑动窗口检查长度为 len*2 的子字符串
            for (int i = 0; i + 2 * len <= n; i++) {
                // 检查前半部分和后半部分是否相等
                if (isEqual(hash1, hash2, pow1, pow2, i, i + len, len)) {
                    // 计算子字符串的哈希值（使用双哈希组合）
                    long long hashVal = getHash(hash1, hash2, pow1, pow2, i, i + 2 * len);
                    seen.insert(hashVal);
                }
            }
        }

        return seen.size();
    }

private:
    /**
     * 检查两个子字符串是否相等
     */
    bool isEqual(const vector<long long>& hash1, const vector<long long>& hash2,
                 const vector<long long>& pow1, const vector<long long>& pow2,
                 int start1, int start2, int len) {
        // 检查第一个哈希
        long long h11 = (hash1[start1 + len] - hash1[start1] * pow1[len] % MOD1 + MOD1) % MOD1;
        long long h12 = (hash1[start2 + len] - hash1[start2] * pow1[len] % MOD1 + MOD1) % MOD1;
```

```
        if (h11 != h12) return false;

        // 检查第二个哈希（双哈希验证）
        long long h21 = (hash2[start1 + len] - hash2[start1] * pow2[len] % MOD2 + MOD2) % MOD2;
        long long h22 = (hash2[start2 + len] - hash2[start2] * pow2[len] % MOD2 + MOD2) % MOD2;
        return h21 == h22;
    }

    /**
     * 获取子字符串的双哈希组合值
     */
    long long getHash(const vector<long long>& hash1, const vector<long long>& hash2,
                      const vector<long long>& pow1, const vector<long long>& pow2,
                      int start, int end) {
        int len = end - start;
        long long h1 = (hash1[end] - hash1[start] * pow1[len] % MOD1 + MOD1) % MOD1;
        long long h2 = (hash2[end] - hash2[start] * pow2[len] % MOD2 + MOD2) % MOD2;
        // 组合两个哈希值
        return h1 * MOD2 + h2;
    }
};

/**
 * 测试函数
 */
int main() {
    Solution solution;

    // 测试用例 1
    string text1 = "abcabcabc";
    int result1 = solution.distinctEchoSubstrings(text1);
    cout << "测试用例 1: " << text1 << " -> " << result1 << endl;
    cout << "预期结果: 3" << endl;
    cout << "测试结果: " << (result1 == 3 ? "通过" : "失败") << endl;
    cout << endl;

    // 测试用例 2
    string text2 = "leetcodeleetcode";
    int result2 = solution.distinctEchoSubstrings(text2);
    cout << "测试用例 2: " << text2 << " -> " << result2 << endl;
    cout << "预期结果: 2" << endl;
    cout << "测试结果: " << (result2 == 2 ? "通过" : "失败") << endl;
    cout << endl;
```

```
    // 测试用例 3：边界情况
    string text3 = "aa";
    int result3 = solution.distinctEchoSubstrings(text3);
    cout << "测试用例 3: " << text3 << " -> " << result3 << endl;
    cout << "预期结果: 1" << endl;
    cout << "测试结果: " << (result3 == 1 ? "通过" : "失败") << endl;

    // 性能测试
    cout << "\n=== 性能测试 ===" << endl;
    auto startTime = chrono::high_resolution_clock::now();
    string largeText(1000, 'a'); // 1000 个'a'
    int largeResult = solution.distinctEchoSubstrings(largeText);
    auto endTime = chrono::high_resolution_clock::now();
    auto duration = chrono::duration_cast<chrono::milliseconds>(endTime - startTime);
    cout << "1000 个字符的性能测试，耗时: " << duration.count() << "ms" << endl;
    cout << "结果: " << largeResult << endl;

    return 0;
}

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

文件: Code11_LeetCode1316_DistinctEchoSubstrings.java
==========================================
import java.util.*;

/**
 * LeetCode 1316. 不同的循环子字符串
 * 题目链接：https://leetcode.cn/problems/distinct-echo-substrings/
 *
 * 题目描述：
 * 给你一个字符串 text，请返回 text 中不同的非空循环子字符串的数目。
 * 循环子字符串定义为：某个字符串与其本身连接一次形成的字符串（比如，abcabc 是 abc 的循环字符串）。
 *
 * 示例：
 * 输入：text = "abcabcabc"
 * 输出：3
 * 解释：3 个不同的循环子字符串是 "abcabc"，"bcabca"，"cabcab"。
 *
 * 解题思路：
 * 1. 使用字符串哈希和滚动哈希技术来高效判断子字符串
```

```
 * 2. 遍历所有可能的子字符串长度（从 1 到 n/2）
 * 3. 对于每个长度，使用滑动窗口检查是否满足循环条件
 * 4. 使用哈希集合去重
 *
 * 时间复杂度：O(n^2)，其中 n 是字符串长度
 * 空间复杂度：O(n^2)，最坏情况下需要存储所有子字符串的哈希值
 *
 * 优化点：
 * - 使用双哈希减少冲突概率
 * - 提前终止不必要的检查
 * - 使用滑动窗口减少重复计算
 */
public class Code11_LeetCode1316_DistinctEchoSubstrings {

    // 双哈希的模数和基数
    private static final int MOD1 = 1000000007;
    private static final int MOD2 = 1000000009;
    private static final int BASE1 = 131;
    private static final int BASE2 = 13131;

    /**
     * 计算不同的循环子字符串数量
     */
    public int distinctEchoSubstrings(String text) {
        int n = text.length();
        if (n <= 1) return 0;

        // 预处理哈希数组和幂数组
        long[] hash1 = new long[n + 1];
        long[] hash2 = new long[n + 1];
        long[] pow1 = new long[n + 1];
        long[] pow2 = new long[n + 1];

        pow1[0] = 1;
        pow2[0] = 1;

        for (int i = 1; i <= n; i++) {
            int c = text.charAt(i - 1);
            hash1[i] = (hash1[i - 1] * BASE1 + c) % MOD1;
            hash2[i] = (hash2[i - 1] * BASE2 + c) % MOD2;
            pow1[i] = (pow1[i - 1] * BASE1) % MOD1;
            pow2[i] = (pow2[i - 1] * BASE2) % MOD2;
        }
```

```
    // 使用集合存储不同的循环子字符串的哈希值
    Set<Long> seen = new HashSet<>();

    // 遍历所有可能的子字符串长度（从 1 到 n/2）
    for (int len = 1; len <= n / 2; len++) {
        // 使用滑动窗口检查长度为 len*2 的子字符串
        for (int i = 0; i + 2 * len <= n; i++) {
            // 检查前半部分和后半部分是否相等
            if (isEqual(hash1, hash2, pow1, pow2, i, i + len, len)) {
                // 计算子字符串的哈希值（使用双哈希组合）
                long hashVal = getHash(hash1, hash2, pow1, pow2, i, i + 2 * len);
                seen.add(hashVal);
            }
        }
    }

    return seen.size();
}

/**
 * 检查两个子字符串是否相等
 */
private boolean isEqual(long[] hash1, long[] hash2, long[] pow1, long[] pow2,
                        int start1, int start2, int len) {
    // 检查第一个哈希
    long h11 = (hash1[start1 + len] - hash1[start1] * pow1[len] % MOD1 + MOD1) % MOD1;
    long h12 = (hash1[start2 + len] - hash1[start2] * pow1[len] % MOD1 + MOD1) % MOD1;
    if (h11 != h12) return false;

    // 检查第二个哈希（双哈希验证）
    long h21 = (hash2[start1 + len] - hash2[start1] * pow2[len] % MOD2 + MOD2) % MOD2;
    long h22 = (hash2[start2 + len] - hash2[start2] * pow2[len] % MOD2 + MOD2) % MOD2;
    return h21 == h22;
}

/**
 * 获取子字符串的双哈希组合值
 */
private long getHash(long[] hash1, long[] hash2, long[] pow1, long[] pow2,
                     int start, int end) {
    int len = end - start;
    long h1 = (hash1[end] - hash1[start] * pow1[len] % MOD1 + MOD1) % MOD1;
```

```
        long h2 = (hash2[end] - hash2[start] * pow2[len] % MOD2 + MOD2) % MOD2;
        // 组合两个哈希值
        return h1 * MOD2 + h2;
    }

    /**
     * 测试方法
     */
    public static void main(String[] args) {
        Code11_LeetCode1316_DistinctEchoSubstrings solution = new
Code11_LeetCode1316_DistinctEchoSubstrings();

        // 测试用例 1
        String text1 = "abcabcabc";
        int result1 = solution.distinctEchoSubstrings(text1);
        System.out.println("测试用例 1: " + text1 + " -> " + result1);
        System.out.println("预期结果: 3");
        System.out.println("测试结果: " + (result1 == 3 ? "通过" : "失败"));
        System.out.println();

        // 测试用例 2
        String text2 = "leetcodeleetcode";
        int result2 = solution.distinctEchoSubstrings(text2);
        System.out.println("测试用例 2: " + text2 + " -> " + result2);
        System.out.println("预期结果: 2");
        System.out.println("测试结果: " + (result2 == 2 ? "通过" : "失败"));
        System.out.println();

        // 测试用例 3：边界情况
        String text3 = "aa";
        int result3 = solution.distinctEchoSubstrings(text3);
        System.out.println("测试用例 3: " + text3 + " -> " + result3);
        System.out.println("预期结果: 1");
        System.out.println("测试结果: " + (result3 == 1 ? "通过" : "失败"));

        // 性能测试
        System.out.println("\n=== 性能测试 ===");
        long startTime = System.currentTimeMillis();
        String largeText = "a".repeat(1000);
        int largeResult = solution.distinctEchoSubstrings(largeText);
        long endTime = System.currentTimeMillis();
        System.out.println("1000 个字符的性能测试，耗时: " + (endTime - startTime) + "ms");
        System.out.println("结果: " + largeResult);
```

```
    }
}
```

==========

文件: Code11_LeetCode1316_DistinctEchoSubstrings.py
==========
"""
LeetCode 1316. 不同的循环子字符串
题目链接：https://leetcode.cn/problems/distinct-echo-substrings/

题目描述：
给你一个字符串 text，请返回 text 中不同的非空循环子字符串的数目。
循环子字符串定义为：某个字符串与其本身连接一次形成的字符串（比如，abcabc 是 abc 的循环字符串）。

示例：
输入：text = "abcabcabc"
输出：3
解释：3 个不同的循环子字符串是 "abcabc"，"bcabca"，"cabcab"。

解题思路：
1. 使用字符串哈希和滚动哈希技术来高效判断子字符串
2. 遍历所有可能的子字符串长度（从 1 到 n/2）
3. 对于每个长度，使用滑动窗口检查是否满足循环条件
4. 使用哈希集合去重

时间复杂度：O(n^2)，其中 n 是字符串长度
空间复杂度：O(n^2)，最坏情况下需要存储所有子字符串的哈希值

优化点：
- 使用双哈希减少冲突概率
- 提前终止不必要的检查
- 使用滑动窗口减少重复计算
"""

```
class Solution:
    def distinctEchoSubstrings(self, text: str) -> int:
        n = len(text)
        if n <= 1:
            return 0

        # 双哈希的模数和基数
        MOD1 = 10**9 + 7
```

```
        MOD2 = 10**9 + 9
        BASE1 = 131
        BASE2 = 13131

        # 预处理哈希数组和幂数组
        hash1 = [0] * (n + 1)
        hash2 = [0] * (n + 1)
        pow1 = [1] * (n + 1)
        pow2 = [1] * (n + 1)

        for i in range(1, n + 1):
            c = ord(text[i - 1])
            hash1[i] = (hash1[i - 1] * BASE1 + c) % MOD1
            hash2[i] = (hash2[i - 1] * BASE2 + c) % MOD2
            pow1[i] = (pow1[i - 1] * BASE1) % MOD1
            pow2[i] = (pow2[i - 1] * BASE2) % MOD2

        # 使用集合存储不同的循环子字符串的哈希值
        seen = set()

        # 遍历所有可能的子字符串长度（从 1 到 n/2）
        for length in range(1, n // 2 + 1):
            # 使用滑动窗口检查长度为 length*2 的子字符串
            for i in range(n - 2 * length + 1):
                # 检查前半部分和后半部分是否相等
                if self.is_equal(hash1, hash2, pow1, pow2, i, i + length, length, MOD1, MOD2):
                    # 计算子字符串的哈希值（使用双哈希组合）
                    hash_val = self.get_hash(hash1, hash2, pow1, pow2, i, i + 2 * length, MOD1,
MOD2)
                    seen.add(hash_val)

        return len(seen)

    def is_equal(self, hash1, hash2, pow1, pow2, start1, start2, length, MOD1, MOD2):
        """检查两个子字符串是否相等"""
        # 检查第一个哈希
        h11 = (hash1[start1 + length] - hash1[start1] * pow1[length] % MOD1 + MOD1) % MOD1
        h12 = (hash1[start2 + length] - hash1[start2] * pow1[length] % MOD1 + MOD1) % MOD1
        if h11 != h12:
            return False

        # 检查第二个哈希（双哈希验证）
        h21 = (hash2[start1 + length] - hash2[start1] * pow2[length] % MOD2 + MOD2) % MOD2
```

```
        h22 = (hash2[start2 + length] - hash2[start2] * pow2[length] % MOD2 + MOD2) % MOD2
        return h21 == h22

    def get_hash(self, hash1, hash2, pow1, pow2, start, end, MOD1, MOD2):
        """获取子字符串的双哈希组合值"""
        length = end - start
        h1 = (hash1[end] - hash1[start] * pow1[length] % MOD1 + MOD1) % MOD1
        h2 = (hash2[end] - hash2[start] * pow2[length] % MOD2 + MOD2) % MOD2
        # 组合两个哈希值
        return h1 * MOD2 + h2


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

    # 测试用例 1
    text1 = "abcabcabc"
    result1 = solution.distinctEchoSubstrings(text1)
    print(f"测试用例 1: {text1} -> {result1}")
    print("预期结果: 3")
    print(f"测试结果: {'通过' if result1 == 3 else '失败'}")
    print()

    # 测试用例 2
    text2 = "leetcodeleetcode"
    result2 = solution.distinctEchoSubstrings(text2)
    print(f"测试用例 2: {text2} -> {result2}")
    print("预期结果: 2")
    print(f"测试结果: {'通过' if result2 == 2 else '失败'}")
    print()

    # 测试用例 3：边界情况
    text3 = "aa"
    result3 = solution.distinctEchoSubstrings(text3)
    print(f"测试用例 3: {text3} -> {result3}")
    print("预期结果: 1")
    print(f"测试结果: {'通过' if result3 == 1 else '失败'}")

    # 性能测试
    print("\n=== 性能测试 ===")
    import time
    start_time = time.time()
```

```
    large_text = "a" * 1000  # 1000个'a'
    large_result = solution.distinctEchoSubstrings(large_text)
    end_time = time.time()
    print(f"1000个字符的性能测试，耗时: {(end_time - start_time) * 1000:.2f}ms")
    print(f"结果: {large_result}")


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

==========

文件: Code12_Codeforces271D_GoodSubstrings.cpp

==========

```
/**
 * Codeforces 271D. Good Substrings
 * 题目链接：https://codeforces.com/problemset/problem/271/D
 *
 * 题目描述：
 * 给你一个字符串 s，一个长度为 26 的字符串 good（表示 26 个字母的好坏），和一个整数 k。
 * 一个子字符串被认为是"好"的，如果它包含的坏字符数量不超过 k 个。
 * 计算字符串 s 中不同的好子字符串的数量。
 *
 * 示例：
 * 输入：s = "ababab", good = "01000000000000000000000000", k = 1
 * 输出：5
 * 解释：好子字符串有 "a", "ab", "aba", "abab", "b", "ba", "bab", "baba"
 *
 * 解题思路：
 * 1. 使用字符串哈希技术来高效计算子字符串
 * 2. 使用前缀和数组快速计算子字符串中的坏字符数量
 * 3. 遍历所有可能的子字符串，检查是否满足条件
 * 4. 使用哈希集合去重
 *
 * 时间复杂度：O(n^2)，其中 n 是字符串长度
 * 空间复杂度：O(n^2)，最坏情况下需要存储所有子字符串的哈希值
 *
 * 优化点：
 * - 使用双哈希减少冲突概率
 * - 使用前缀和优化坏字符计数
 * - 提前终止不必要的检查
 */
```

```cpp
#include <iostream>
#include <string>
#include <unordered_set>
#include <vector>
#include <chrono>
using namespace std;

class Solution {
private:
    // 双哈希的模数和基数
    static const int MOD1 = 1000000007;
    static const int MOD2 = 1000000009;
    static const int BASE1 = 131;
    static const int BASE2 = 13131;

public:
    /**
     * 计算不同的好子字符串数量
     */
    int countGoodSubstrings(string s, string good, int k) {
        int n = s.size();
        if (n == 0) return 0;

        // 预处理坏字符标记数组
        vector<bool> isBad(26, false);
        for (int i = 0; i < 26; i++) {
            isBad[i] = good[i] == '0';
        }

        // 预处理前缀和数组，用于快速计算坏字符数量
        vector<int> badPrefix(n + 1, 0);
        for (int i = 1; i <= n; i++) {
            char c = s[i - 1];
            badPrefix[i] = badPrefix[i - 1] + (isBad[c - 'a'] ? 1 : 0);
        }

        // 预处理哈希数组和幂数组
        vector<long long> hash1(n + 1, 0);
        vector<long long> hash2(n + 1, 0);
        vector<long long> pow1(n + 1, 1);
        vector<long long> pow2(n + 1, 1);

        for (int i = 1; i <= n; i++) {
```

```
        int c = s[i - 1];
        hash1[i] = (hash1[i - 1] * BASE1 + c) % MOD1;
        hash2[i] = (hash2[i - 1] * BASE2 + c) % MOD2;
        pow1[i] = (pow1[i - 1] * BASE1) % MOD1;
        pow2[i] = (pow2[i - 1] * BASE2) % MOD2;
    }

    // 使用集合存储不同的好子字符串的哈希值
    unordered_set<long long> seen;

    // 遍历所有可能的子字符串
    for (int i = 0; i < n; i++) {
        for (int j = i + 1; j <= n; j++) {
            // 计算子字符串中的坏字符数量
            int badCount = badPrefix[j] - badPrefix[i];

            // 检查是否满足条件
            if (badCount <= k) {
                // 计算子字符串的哈希值（使用双哈希组合）
                long long hashVal = getHash(hash1, hash2, pow1, pow2, i, j);
                seen.insert(hashVal);
            }
        }
    }

    return seen.size();
}

/**
 * 优化版本：使用滑动窗口和哈希集合，减少重复计算
 */
int countGoodSubstringsOptimized(string s, string good, int k) {
    int n = s.size();
    if (n == 0) return 0;

    // 预处理坏字符标记数组
    vector<bool> isBad(26, false);
    for (int i = 0; i < 26; i++) {
        isBad[i] = good[i] == '0';
    }

    // 预处理哈希数组和幂数组
    vector<long long> hash1(n + 1, 0);
```

```
        vector<long long> hash2(n + 1, 0);
        vector<long long> pow1(n + 1, 1);
        vector<long long> pow2(n + 1, 1);

        for (int i = 1; i <= n; i++) {
            int c = s[i - 1];
            hash1[i] = (hash1[i - 1] * BASE1 + c) % MOD1;
            hash2[i] = (hash2[i - 1] * BASE2 + c) % MOD2;
            pow1[i] = (pow1[i - 1] * BASE1) % MOD1;
            pow2[i] = (pow2[i - 1] * BASE2) % MOD2;
        }

        // 使用集合存储不同的好子字符串的哈希值
        unordered_set<long long> seen;

        // 对于每个起始位置，使用滑动窗口
        for (int i = 0; i < n; i++) {
            int badCount = 0;

            // 从 i 开始，向右扩展窗口
            for (int j = i; j < n; j++) {
                char c = s[j];
                if (isBad[c - 'a']) {
                    badCount++;
                }

                // 如果坏字符数量超过 k，停止扩展
                if (badCount > k) {
                    break;
                }

                // 计算子字符串的哈希值
                long long hashVal = getHash(hash1, hash2, pow1, pow2, i, j + 1);
                seen.insert(hashVal);
            }
        }

        return seen.size();
    }

private:
    /**
     * 获取子字符串的双哈希组合值
```

```
     */
    long long getHash(const vector<long long>& hash1, const vector<long long>& hash2,
                      const vector<long long>& pow1, const vector<long long>& pow2,
                      int start, int end) {
        int len = end - start;
        long long h1 = (hash1[end] - hash1[start] * pow1[len] % MOD1 + MOD1) % MOD1;
        long long h2 = (hash2[end] - hash2[start] * pow2[len] % MOD2 + MOD2) % MOD2;
        // 组合两个哈希值
        return h1 * MOD2 + h2;
    }
};

/**
 * 测试函数
 */
int main() {
    Solution solution;

    // 测试用例 1
    string s1 = "ababab";
    string good1 = "01000000000000000000000000";
    int k1 = 1;
    int result1 = solution.countGoodSubstringsOptimized(s1, good1, k1);
    cout << "测试用例 1: s = \"" << s1 << "\", k = " << k1 << " -> " << result1 << endl;
    cout << "预期结果: 5" << endl;
    cout << "测试结果: " << (result1 == 5 ? "通过" : "失败") << endl;
    cout << endl;

    // 测试用例 2
    string s2 = "aaabbb";
    string good2 = "10000000000000000000000000";
    int k2 = 0;
    int result2 = solution.countGoodSubstringsOptimized(s2, good2, k2);
    cout << "测试用例 2: s = \"" << s2 << "\", k = " << k2 << " -> " << result2 << endl;
    cout << "预期结果: 3" << endl;
    cout << "测试结果: " << (result2 == 3 ? "通过" : "失败") << endl;
    cout << endl;

    // 测试用例 3：边界情况
    string s3 = "a";
    string good3 = "10000000000000000000000000";
    int k3 = 1;
    int result3 = solution.countGoodSubstringsOptimized(s3, good3, k3);
```

```
    cout << "测试用例 3: s = \"" << s3 << "\", k = " << k3 << " -> " << result3 << endl;
    cout << "预期结果: 1" << endl;
    cout << "测试结果: " << (result3 == 1 ? "通过" : "失败") << endl;

    // 性能测试
    cout << "\n=== 性能测试 ===" << endl;
    auto startTime = chrono::high_resolution_clock::now();
    string largeS;
    for (int i = 0; i < 10; i++) {
        largeS += "abcdefghijklmnopqrstuvwxyz";
    }
    string largeGood = "01010101010101010101010101";
    int largeK = 10;
    int largeResult = solution.countGoodSubstringsOptimized(largeS, largeGood, largeK);
    auto endTime = chrono::high_resolution_clock::now();
    auto duration = chrono::duration_cast<chrono::milliseconds>(endTime - startTime);
    cout << "260 个字符的性能测试，耗时: " << duration.count() << "ms" << endl;
    cout << "结果: " << largeResult << endl;

    // 对比两种方法的性能
    cout << "\n=== 方法对比 ===" << endl;
    startTime = chrono::high_resolution_clock::now();
    int resultBasic = solution.countGoodSubstrings(s1, good1, k1);
    auto basicTime =
chrono::duration_cast<chrono::milliseconds>(chrono::high_resolution_clock::now() - startTime);

    startTime = chrono::high_resolution_clock::now();
    int resultOptimized = solution.countGoodSubstringsOptimized(s1, good1, k1);
    auto optimizedTime =
chrono::duration_cast<chrono::milliseconds>(chrono::high_resolution_clock::now() - startTime);

    cout << "基础方法结果: " << resultBasic << ", 耗时: " << basicTime.count() << "ms" << endl;
    cout << "优化方法结果: " << resultOptimized << ", 耗时: " << optimizedTime.count() << "ms" <<
endl;

    return 0;
}
```

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

文件: Code12_Codeforces271D_GoodSubstrings.java

```
==============================================
import java.util.*;
```

```java
/**
 * Codeforces 271D. Good Substrings
 * 题目链接：https://codeforces.com/problemset/problem/271/D
 *
 * 题目描述：
 * 给你一个字符串 s，一个长度为 26 的字符串 good（表示 26 个字母的好坏），和一个整数 k。
 * 一个子字符串被认为是"好"的，如果它包含的坏字符数量不超过 k 个。
 * 计算字符串 s 中不同的好子字符串的数量。
 *
 * 示例：
 * 输入：s = "ababab", good = "01000000000000000000000000", k = 1
 * 输出：5
 * 解释：好子字符串有 "a", "ab", "aba", "abab", "b", "ba", "bab", "baba"
 *
 * 解题思路：
 * 1. 使用字符串哈希技术来高效计算子字符串
 * 2. 使用前缀和数组快速计算子字符串中的坏字符数量
 * 3. 遍历所有可能的子字符串，检查是否满足条件
 * 4. 使用哈希集合去重
 *
 * 时间复杂度：O(n^2)，其中 n 是字符串长度
 * 空间复杂度：O(n^2)，最坏情况下需要存储所有子字符串的哈希值
 *
 * 优化点：
 * - 使用双哈希减少冲突概率
 * - 使用前缀和优化坏字符计数
 * - 提前终止不必要的检查
 */
public class Code12_Codeforces271D_GoodSubstrings {

    // 双哈希的模数和基数
    private static final int MOD1 = 1000000007;
    private static final int MOD2 = 1000000009;
    private static final int BASE1 = 131;
    private static final int BASE2 = 13131;

    /**
     * 计算不同的好子字符串数量
     */
    public int countGoodSubstrings(String s, String good, int k) {
        int n = s.length();
        if (n == 0) return 0;
```

```
// 预处理坏字符标记数组
boolean[] isBad = new boolean[26];
for (int i = 0; i < 26; i++) {
    isBad[i] = good.charAt(i) == '0';
}

// 预处理前缀和数组，用于快速计算坏字符数量
int[] badPrefix = new int[n + 1];
for (int i = 1; i <= n; i++) {
    char c = s.charAt(i - 1);
    badPrefix[i] = badPrefix[i - 1] + (isBad[c - 'a'] ? 1 : 0);
}

// 预处理哈希数组和幂数组
long[] hash1 = new long[n + 1];
long[] hash2 = new long[n + 1];
long[] pow1 = new long[n + 1];
long[] pow2 = new long[n + 1];

pow1[0] = 1;
pow2[0] = 1;

for (int i = 1; i <= n; i++) {
    int c = s.charAt(i - 1);
    hash1[i] = (hash1[i - 1] * BASE1 + c) % MOD1;
    hash2[i] = (hash2[i - 1] * BASE2 + c) % MOD2;
    pow1[i] = (pow1[i - 1] * BASE1) % MOD1;
    pow2[i] = (pow2[i - 1] * BASE2) % MOD2;
}

// 使用集合存储不同的好子字符串的哈希值
Set<Long> seen = new HashSet<>();

// 遍历所有可能的子字符串
for (int i = 0; i < n; i++) {
    for (int j = i + 1; j <= n; j++) {
        // 计算子字符串中的坏字符数量
        int badCount = badPrefix[j] - badPrefix[i];

        // 检查是否满足条件
        if (badCount <= k) {
            // 计算子字符串的哈希值（使用双哈希组合）
```

```
                long hashVal = getHash(hash1, hash2, pow1, pow2, i, j);
                seen.add(hashVal);
            }
        }
    }

    return seen.size();
}

/**
 * 获取子字符串的双哈希组合值
 */
private long getHash(long[] hash1, long[] hash2, long[] pow1, long[] pow2,
                     int start, int end) {
    int len = end - start;
    long h1 = (hash1[end] - hash1[start] * pow1[len] % MOD1 + MOD1) % MOD1;
    long h2 = (hash2[end] - hash2[start] * pow2[len] % MOD2 + MOD2) % MOD2;
    // 组合两个哈希值
    return h1 * MOD2 + h2;
}

/**
 * 优化版本：使用滑动窗口和哈希集合，减少重复计算
 */
public int countGoodSubstringsOptimized(String s, String good, int k) {
    int n = s.length();
    if (n == 0) return 0;

    // 预处理坏字符标记数组
    boolean[] isBad = new boolean[26];
    for (int i = 0; i < 26; i++) {
        isBad[i] = good.charAt(i) == '0';
    }

    // 预处理哈希数组和幂数组
    long[] hash1 = new long[n + 1];
    long[] hash2 = new long[n + 1];
    long[] pow1 = new long[n + 1];
    long[] pow2 = new long[n + 1];

    pow1[0] = 1;
    pow2[0] = 1;
```

```
        for (int i = 1; i <= n; i++) {
            int c = s.charAt(i - 1);
            hash1[i] = (hash1[i - 1] * BASE1 + c) % MOD1;
            hash2[i] = (hash2[i - 1] * BASE2 + c) % MOD2;
            pow1[i] = (pow1[i - 1] * BASE1) % MOD1;
            pow2[i] = (pow2[i - 1] * BASE2) % MOD2;
        }

        // 使用集合存储不同的好子字符串的哈希值
        Set<Long> seen = new HashSet<>();

        // 对于每个起始位置，使用滑动窗口
        for (int i = 0; i < n; i++) {
            int badCount = 0;

            // 从 i 开始，向右扩展窗口
            for (int j = i; j < n; j++) {
                char c = s.charAt(j);
                if (isBad[c - 'a']) {
                    badCount++;
                }

                // 如果坏字符数量超过 k，停止扩展
                if (badCount > k) {
                    break;
                }

                // 计算子字符串的哈希值
                long hashVal = getHash(hash1, hash2, pow1, pow2, i, j + 1);
                seen.add(hashVal);
            }
        }

        return seen.size();
    }

    /**
     * 测试方法
     */
    public static void main(String[] args) {
        Code12_Codeforces271D_GoodSubstrings solution = new
Code12_Codeforces271D_GoodSubstrings();
```

```
// 测试用例 1
String s1 = "ababab";
String good1 = "01000000000000000000000000";
int k1 = 1;
int result1 = solution.countGoodSubstringsOptimized(s1, good1, k1);
System.out.println("测试用例 1: s = \"" + s1 + "\", k = " + k1 + " -> " + result1);
System.out.println("预期结果: 5");
System.out.println("测试结果: " + (result1 == 5 ? "通过" : "失败"));
System.out.println();

// 测试用例 2
String s2 = "aaabbb";
String good2 = "10000000000000000000000000";
int k2 = 0;
int result2 = solution.countGoodSubstringsOptimized(s2, good2, k2);
System.out.println("测试用例 2: s = \"" + s2 + "\", k = " + k2 + " -> " + result2);
System.out.println("预期结果: 3");
System.out.println("测试结果: " + (result2 == 3 ? "通过" : "失败"));
System.out.println();

// 测试用例 3：边界情况
String s3 = "a";
String good3 = "10000000000000000000000000";
int k3 = 1;
int result3 = solution.countGoodSubstringsOptimized(s3, good3, k3);
System.out.println("测试用例 3: s = \"" + s3 + "\", k = " + k3 + " -> " + result3);
System.out.println("预期结果: 1");
System.out.println("测试结果: " + (result3 == 1 ? "通过" : "失败"));

// 性能测试
System.out.println("\n=== 性能测试 ===");
long startTime = System.currentTimeMillis();
String largeS = "abcdefghijklmnopqrstuvwxyz".repeat(10); // 260 个字符
String largeGood = "01010101010101010101010101";
int largeK = 10;
int largeResult = solution.countGoodSubstringsOptimized(largeS, largeGood, largeK);
long endTime = System.currentTimeMillis();
System.out.println("260 个字符的性能测试，耗时: " + (endTime - startTime) + "ms");
System.out.println("结果: " + largeResult);

// 对比两种方法的性能
System.out.println("\n=== 方法对比 ===");
startTime = System.currentTimeMillis();
```

```
        int resultBasic = solution.countGoodSubstrings(s1, good1, k1);
        long basicTime = System.currentTimeMillis() - startTime;

        startTime = System.currentTimeMillis();
        int resultOptimized = solution.countGoodSubstringsOptimized(s1, good1, k1);
        long optimizedTime = System.currentTimeMillis() - startTime;

        System.out.println("基础方法结果: " + resultBasic + ", 耗时: " + basicTime + "ms");
        System.out.println("优化方法结果: " + resultOptimized + ", 耗时: " + optimizedTime +
"ms");
    }
}

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

文件: Code12_Codeforces271D_GoodSubstrings.py
==========================================
"""
Codeforces 271D. Good Substrings
题目链接：https://codeforces.com/problemset/problem/271/D

题目描述：
给你一个字符串 s，一个长度为 26 的字符串 good（表示 26 个字母的好坏），和一个整数 k。
一个子字符串被认为是"好"的，如果它包含的坏字符数量不超过 k 个。
计算字符串 s 中不同的好子字符串的数量。

示例：
输入：s = "ababab", good = "01000000000000000000000000", k = 1
输出：5
解释：好子字符串有 "a", "ab", "aba", "abab", "b", "ba", "bab", "baba"

解题思路：
1. 使用字符串哈希技术来高效计算子字符串
2. 使用前缀和数组快速计算子字符串中的坏字符数量
3. 遍历所有可能的子字符串，检查是否满足条件
4. 使用哈希集合去重

时间复杂度：O(n^2)，其中 n 是字符串长度
空间复杂度：O(n^2)，最坏情况下需要存储所有子字符串的哈希值

优化点：
- 使用双哈希减少冲突概率
- 使用前缀和优化坏字符计数
```

- 提前终止不必要的检查
"""

```
class Solution:
    def countGoodSubstrings(self, s: str, good: str, k: int) -> int:
        n = len(s)
        if n == 0:
            return 0

        # 双哈希的模数和基数
        MOD1 = 10**9 + 7
        MOD2 = 10**9 + 9
        BASE1 = 131
        BASE2 = 13131

        # 预处理坏字符标记数组
        is_bad = [False] * 26
        for i in range(26):
            is_bad[i] = good[i] == '0'

        # 预处理前缀和数组，用于快速计算坏字符数量
        bad_prefix = [0] * (n + 1)
        for i in range(1, n + 1):
            c = s[i - 1]
            bad_prefix[i] = bad_prefix[i - 1] + (1 if is_bad[ord(c) - ord('a')] else 0)

        # 预处理哈希数组和幂数组
        hash1 = [0] * (n + 1)
        hash2 = [0] * (n + 1)
        pow1 = [1] * (n + 1)
        pow2 = [1] * (n + 1)

        for i in range(1, n + 1):
            c = ord(s[i - 1])
            hash1[i] = (hash1[i - 1] * BASE1 + c) % MOD1
            hash2[i] = (hash2[i - 1] * BASE2 + c) % MOD2
            pow1[i] = (pow1[i - 1] * BASE1) % MOD1
            pow2[i] = (pow2[i - 1] * BASE2) % MOD2

        # 使用集合存储不同的好子字符串的哈希值
        seen = set()

        # 遍历所有可能的子字符串
```

```
    for i in range(n):
        for j in range(i + 1, n + 1):
            # 计算子字符串中的坏字符数量
            bad_count = bad_prefix[j] - bad_prefix[i]

            # 检查是否满足条件
            if bad_count <= k:
                # 计算子字符串的哈希值（使用双哈希组合）
                hash_val = self.get_hash(hash1, hash2, pow1, pow2, i, j, MOD1, MOD2)
                seen.add(hash_val)

    return len(seen)

def countGoodSubstringsOptimized(self, s: str, good: str, k: int) -> int:
    """优化版本：使用滑动窗口和哈希集合，减少重复计算"""
    n = len(s)
    if n == 0:
        return 0

    # 双哈希的模数和基数
    MOD1 = 10**9 + 7
    MOD2 = 10**9 + 9
    BASE1 = 131
    BASE2 = 13131

    # 预处理坏字符标记数组
    is_bad = [False] * 26
    for i in range(26):
        is_bad[i] = good[i] == '0'

    # 预处理哈希数组和幂数组
    hash1 = [0] * (n + 1)
    hash2 = [0] * (n + 1)
    pow1 = [1] * (n + 1)
    pow2 = [1] * (n + 1)

    for i in range(1, n + 1):
        c = ord(s[i - 1])
        hash1[i] = (hash1[i - 1] * BASE1 + c) % MOD1
        hash2[i] = (hash2[i - 1] * BASE2 + c) % MOD2
        pow1[i] = (pow1[i - 1] * BASE1) % MOD1
        pow2[i] = (pow2[i - 1] * BASE2) % MOD2
```

```
        # 使用集合存储不同的好子字符串的哈希值
        seen = set()

        # 对于每个起始位置，使用滑动窗口
        for i in range(n):
            bad_count = 0

            # 从 i 开始，向右扩展窗口
            for j in range(i, n):
                c = s[j]
                if is_bad[ord(c) - ord('a')]:
                    bad_count += 1

                # 如果坏字符数量超过 k，停止扩展
                if bad_count > k:
                    break

                # 计算子字符串的哈希值
                hash_val = self.get_hash(hash1, hash2, pow1, pow2, i, j + 1, MOD1, MOD2)
                seen.add(hash_val)

        return len(seen)

    def get_hash(self, hash1, hash2, pow1, pow2, start, end, MOD1, MOD2):
        """获取子字符串的双哈希组合值"""
        length = end - start
        h1 = (hash1[end] - hash1[start] * pow1[length] % MOD1 + MOD1) % MOD1
        h2 = (hash2[end] - hash2[start] * pow2[length] % MOD2 + MOD2) % MOD2
        # 组合两个哈希值
        return h1 * MOD2 + h2


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

    # 测试用例 1
    s1 = "ababab"
    good1 = "01000000000000000000000000"
    k1 = 1
    result1 = solution.countGoodSubstringsOptimized(s1, good1, k1)
    print(f"测试用例 1: s = '{s1}', k = {k1} -> {result1}")
    print("预期结果: 5")
```

```
print(f"测试结果: {'通过' if result1 == 5 else '失败'}")
print()

# 测试用例 2
s2 = "aaabbb"
good2 = "10000000000000000000000000"
k2 = 0
result2 = solution.countGoodSubstringsOptimized(s2, good2, k2)
print(f"测试用例 2: s = '{s2}', k = {k2} -> {result2}")
print("预期结果: 3")
print(f"测试结果: {'通过' if result2 == 3 else '失败'}")
print()

# 测试用例 3：边界情况
s3 = "a"
good3 = "10000000000000000000000000"
k3 = 1
result3 = solution.countGoodSubstringsOptimized(s3, good3, k3)
print(f"测试用例 3: s = '{s3}', k = {k3} -> {result3}")
print("预期结果: 1")
print(f"测试结果: {'通过' if result3 == 1 else '失败'}")

# 性能测试
print("\n=== 性能测试 ===")
import time
start_time = time.time()
large_s = "abcdefghijklmnopqrstuvwxyz" * 10  # 260 个字符
large_good = "01010101010101010101010101"
large_k = 10
large_result = solution.countGoodSubstringsOptimized(large_s, large_good, large_k)
end_time = time.time()
print(f"260 个字符的性能测试，耗时: {(end_time - start_time) * 1000:.2f}ms")
print(f"结果: {large_result}")

# 对比两种方法的性能
print("\n=== 方法对比 ===")
start_time = time.time()
result_basic = solution.countGoodSubstrings(s1, good1, k1)
basic_time = (time.time() - start_time) * 1000

start_time = time.time()
result_optimized = solution.countGoodSubstringsOptimized(s1, good1, k1)
optimized_time = (time.time() - start_time) * 1000
```

```
    print(f"基础方法结果: {result_basic}, 耗时: {basic_time:.2f}ms")
    print(f"优化方法结果: {result_optimized}, 耗时: {optimized_time:.2f}ms")


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

==========

文件: Code13_HashSetDesign.java

==========

```
import java.util.*;

/**
 * 自定义哈希集合实现
 * 题目来源：LeetCode 705. 设计哈希集合
 * 题目链接：https://leetcode.cn/problems/design-hashset/
 *
 * 题目描述：
 * 不使用任何内建的哈希表库设计一个哈希集合（HashSet）。
 * 实现 MyHashSet 类：
 * - void add(key) 向哈希集合中插入值 key 。
 * - bool contains(key) 返回哈希集合中是否存在这个值 key 。
 * - void remove(key) 将给定值 key 从哈希集合中删除。如果哈希集合中没有这个值，什么也不做。
 *
 * 解题思路：
 * 1. 使用链地址法解决哈希冲突
 * 2. 选择合适的哈希函数和桶大小
 * 3. 实现动态扩容机制
 * 4. 处理边界情况和异常输入
 *
 * 时间复杂度分析：
 * - 平均情况：O(1) 对于 add、remove、contains 操作
 * - 最坏情况：O(n) 当所有元素都哈希到同一个桶时
 *
 * 空间复杂度：O(n + m)，其中 n 是元素数量，m 是桶的数量
 *
 * 优化点：
 * - 动态扩容保持负载因子合理
 * - 使用质数作为桶大小减少冲突
 * - 优化哈希函数分布
 */
```

```
public class Code13_HashSetDesign {

    /**
     * 哈希集合节点类
     */
    private static class Node {
        int key;
        Node next;

        Node(int key) {
            this.key = key;
        }
    }

    /**
     * 自定义哈希集合实现类
     */
    public static class MyHashSet {
        private static final int INITIAL_CAPACITY = 16;
        private static final double LOAD_FACTOR = 0.75;

        private Node[] buckets;
        private int size;

        public MyHashSet() {
            this(INITIAL_CAPACITY);
        }

        public MyHashSet(int initialCapacity) {
            buckets = new Node[initialCapacity];
            size = 0;
        }

        /**
         * 向哈希集合中添加元素
         */
        public void add(int key) {
            if (contains(key)) {
                return; // 元素已存在，直接返回
            }

            // 检查是否需要扩容
            if ((double) size / buckets.length > LOAD_FACTOR) {
```

```
            resize();
        }

        int index = getIndex(key);
        Node newNode = new Node(key);

        // 头插法
        newNode.next = buckets[index];
        buckets[index] = newNode;
        size++;
    }

    /**
     * 从哈希集合中移除元素
     */
    public void remove(int key) {
        int index = getIndex(key);
        Node current = buckets[index];
        Node prev = null;

        while (current != null) {
            if (current.key == key) {
                if (prev == null) {
                    // 删除头节点
                    buckets[index] = current.next;
                } else {
                    prev.next = current.next;
                }
                size--;
                return;
            }
            prev = current;
            current = current.next;
        }
    }

    /**
     * 检查哈希集合是否包含元素
     */
    public boolean contains(int key) {
        int index = getIndex(key);
        Node current = buckets[index];
```

```
        while (current != null) {
            if (current.key == key) {
                return true;
            }
            current = current.next;
        }

        return false;
    }

    /**
     * 获取元素数量
     */
    public int size() {
        return size;
    }

    /**
     * 检查哈希集合是否为空
     */
    public boolean isEmpty() {
        return size == 0;
    }

    /**
     * 清空哈希集合
     */
    public void clear() {
        Arrays.fill(buckets, null);
        size = 0;
    }

    /**
     * 获取元素的哈希索引
     */
    private int getIndex(int key) {
        // 使用 Java 的 hashCode 方法并取模
        return Math.abs(Integer.hashCode(key)) % buckets.length;
    }

    /**
     * 动态扩容
     */
```

```
private void resize() {
    int newCapacity = buckets.length * 2;
    Node[] newBuckets = new Node[newCapacity];

    // 重新哈希所有元素
    for (Node head : buckets) {
        Node current = head;
        while (current != null) {
            Node next = current.next;
            int newIndex = Math.abs(Integer.hashCode(current.key)) % newCapacity;

            // 头插法插入新桶
            current.next = newBuckets[newIndex];
            newBuckets[newIndex] = current;

            current = next;
        }
    }

    buckets = newBuckets;
}

/**
 * 打印哈希集合状态（用于调试）
 */
public void printStats() {
    System.out.println("哈希集合状态：");
    System.out.println("容量：" + buckets.length);
    System.out.println("元素数量：" + size);
    System.out.println("负载因子：" + (double) size / buckets.length);

    // 统计每个桶的元素数量
    int[] bucketSizes = new int[buckets.length];
    for (int i = 0; i < buckets.length; i++) {
        Node current = buckets[i];
        int count = 0;
        while (current != null) {
            count++;
            current = current.next;
        }
        bucketSizes[i] = count;
    }
```

```
        System.out.println("桶分布：" + Arrays.toString(bucketSizes));
    }
}

/**
 * 测试方法
 */
public static void main(String[] args) {
    // 测试基本功能
    System.out.println("=== 基本功能测试 ===");
    MyHashSet hashSet = new MyHashSet();

    // 测试添加操作
    hashSet.add(1);
    hashSet.add(2);
    hashSet.add(3);
    System.out.println("添加 1,2,3 后大小：" + hashSet.size());
    System.out.println("是否包含 2：" + hashSet.contains(2));
    System.out.println("是否包含 4：" + hashSet.contains(4));

    // 测试删除操作
    hashSet.remove(2);
    System.out.println("删除 2 后大小：" + hashSet.size());
    System.out.println("是否包含 2：" + hashSet.contains(2));

    // 测试重复添加
    hashSet.add(1);
    System.out.println("重复添加 1 后大小：" + hashSet.size());

    // 测试边界情况
    System.out.println("\n=== 边界情况测试 ===");
    hashSet.add(Integer.MAX_VALUE);
    hashSet.add(Integer.MIN_VALUE);
    System.out.println("添加边界值后大小：" + hashSet.size());
    System.out.println("是否包含 Integer.MAX_VALUE：" + hashSet.contains(Integer.MAX_VALUE));
    System.out.println("是否包含 Integer.MIN_VALUE：" + hashSet.contains(Integer.MIN_VALUE));

    // 测试性能
    System.out.println("\n=== 性能测试 ===");
    MyHashSet largeSet = new MyHashSet();
    long startTime = System.currentTimeMillis();

    // 添加 10000 个元素
```

```
        for (int i = 0; i < 10000; i++) {
            largeSet.add(i);
        }

        long addTime = System.currentTimeMillis() - startTime;
        System.out.println("添加 10000 个元素耗时：" + addTime + "ms");

        // 查询性能
        startTime = System.currentTimeMillis();
        for (int i = 0; i < 10000; i++) {
            largeSet.contains(i);
        }
        long queryTime = System.currentTimeMillis() - startTime;
        System.out.println("查询 10000 个元素耗时：" + queryTime + "ms");

        // 打印统计信息
        System.out.println("\n=== 统计信息 ===");
        largeSet.printStats();

        // 测试动态扩容
        System.out.println("\n=== 动态扩容测试 ===");
        MyHashSet resizeSet = new MyHashSet(4); // 小容量初始值
        for (int i = 0; i < 10; i++) {
            resizeSet.add(i);
            System.out.println("添加" + i + "后容量：" + resizeSet.size());
        }
        resizeSet.printStats();

        // 测试异常情况
        System.out.println("\n=== 异常情况测试 ===");
        try {
            hashSet.remove(9999); // 删除不存在的元素
            System.out.println("删除不存在的元素：正常处理");
        } catch (Exception e) {
            System.out.println("删除不存在的元素异常：" + e.getMessage());
        }

        // 测试清空操作
        hashSet.clear();
        System.out.println("清空后大小：" + hashSet.size());
        System.out.println("是否为空：" + hashSet.isEmpty());
    }
}
```

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

文件: Code14_ConsistentHashing.cpp
==========================================
#include <iostream>
#include <string>
#include <map>
#include <set>
#include <functional>
#include <openssl/md5.h>
#include <chrono>

using namespace std;

/**
 * 一致性哈希算法实现（C++版本）
 *
 * 题目来源：分布式系统设计面试题
 * 应用场景：负载均衡、分布式缓存、分布式存储系统
 *
 * 核心思想：
 * 1. 将哈希空间组织成一个虚拟的圆环（0 ~ 2^32-1）
 * 2. 服务器节点通过哈希函数映射到环上
 * 3. 数据通过哈希函数映射到环上，顺时针找到最近的服务器节点
 * 4. 虚拟节点技术解决数据分布不均问题
 *
 * 时间复杂度：
 * - 添加节点：O(k) k为虚拟节点数
 * - 删除节点：O(k)
 * - 查找节点：O(log n) n为环上节点总数
 *
 * 空间复杂度：O(n*k) n为物理节点数，k为虚拟节点数
 *
 * 工程化考量：
 * 1. 虚拟节点解决数据倾斜问题
 * 2. 支持节点的动态增删
 * 3. 数据迁移最小化
 * 4. 容错性和可扩展性
 */
class ConsistentHashing {
private:
    // 哈希环，存储虚拟节点到物理节点的映射
```

```
    map<int, string> ring;

    // 虚拟节点数量
    int virtualNodes;

    // 物理节点集合
    set<string> physicalNodes;

public:
    ConsistentHashing(int vNodes) : virtualNodes(vNodes) {}

    /**
     * 添加物理节点
     * @param node 物理节点标识
     */
    void addNode(const string& node) {
        if (physicalNodes.find(node) != physicalNodes.end()) {
            return; // 节点已存在
        }

        physicalNodes.insert(node);

        // 为每个物理节点创建虚拟节点
        for (int i = 0; i < virtualNodes; i++) {
            string virtualNode = node + "#" + to_string(i);
            int hash = getHash(virtualNode);
            ring[hash] = node;
        }
    }

    /**
     * 删除物理节点
     * @param node 物理节点标识
     */
    void removeNode(const string& node) {
        if (physicalNodes.find(node) == physicalNodes.end()) {
            return; // 节点不存在
        }

        physicalNodes.erase(node);

        // 删除该节点的所有虚拟节点
        for (int i = 0; i < virtualNodes; i++) {
```

```
        string virtualNode = node + "#" + to_string(i);
        int hash = getHash(virtualNode);
        ring.erase(hash);
    }
}

/**
 * 根据key查找对应的物理节点
 * @param key 数据key
 * @return 物理节点标识
 */
string getNode(const string& key) {
    if (ring.empty()) {
        return "";
    }

    int hash = getHash(key);

    // 在环上查找大于等于该hash的第一个节点
    auto it = ring.lower_bound(hash);

    // 如果没找到，则返回环的第一个节点（环形结构）
    if (it == ring.end()) {
        it = ring.begin();
    }

    return it->second;
}

/**
 * 哈希函数：使用MD5哈希然后取模
 * @param key 输入字符串
 * @return 哈希值（0 ~ 2^32-1）
 */
int getHash(const string& key) {
    unsigned char digest[MD5_DIGEST_LENGTH];
    MD5((const unsigned char*)key.c_str(), key.length(), digest);

    // 取前4个字节作为哈希值
    int hash = 0;
    for (int i = 0; i < 4; i++) {
        hash = (hash << 8) | digest[i];
    }
```

```
        return hash & 0x7FFFFFFF; // 确保为正数
    }

    /**
     * 获取环上节点分布情况（用于调试）
     */
    void printRing() {
        cout << "一致性哈希环状态：" << endl;
        for (const auto& entry : ring) {
            cout << "位置 " << entry.first << " -> " << entry.second << endl;
        }
    }

    /**
     * 获取物理节点数量
     */
    int getPhysicalNodeCount() {
        return physicalNodes.size();
    }

    /**
     * 获取虚拟节点数量
     */
    int getVirtualNodeCount() {
        return ring.size();
    }
};

/**
 * 测试函数
 */
int main() {
    // 创建一致性哈希环，每个物理节点有 3 个虚拟节点
    ConsistentHashing ch(3);

    // 添加物理节点
    ch.addNode("Server-A");
    ch.addNode("Server-B");
    ch.addNode("Server-C");

    // 测试数据分布
    string testKeys[] = {"user1", "user2", "user3", "data1", "data2", "data3"};
```

```
cout << "=== 初始节点分布测试 ===" << endl;
for (const auto& key : testKeys) {
    string node = ch.getNode(key);
    cout << "Key: " << key << " -> Node: " << node << endl;
}

// 测试节点删除
cout << "\n=== 删除 Server-B 后测试 ===" << endl;
ch.removeNode("Server-B");

for (const auto& key : testKeys) {
    string node = ch.getNode(key);
    cout << "Key: " << key << " -> Node: " << node << endl;
}

// 测试节点添加
cout << "\n=== 添加 Server-D 后测试 ===" << endl;
ch.addNode("Server-D");

for (const auto& key : testKeys) {
    string node = ch.getNode(key);
    cout << "Key: " << key << " -> Node: " << node << endl;
}

// 性能测试
cout << "\n=== 性能测试 ===" << endl;
auto startTime = chrono::high_resolution_clock::now();

for (int i = 0; i < 10000; i++) {
    ch.getNode("test" + to_string(i));
}

auto endTime = chrono::high_resolution_clock::now();
auto duration = chrono::duration_cast<chrono::milliseconds>(endTime - startTime);
cout << "10000 次查找耗时: " << duration.count() << "ms" << endl;

// 打印环状态（调试用）
ch.printRing();

cout << "物理节点数量: " << ch.getPhysicalNodeCount() << endl;
cout << "虚拟节点数量: " << ch.getVirtualNodeCount() << endl;

return 0;
```

```
}

==========

文件: Code14_ConsistentHashing.java
==========
import java.util.*;

/**
 * 一致性哈希算法实现
 *
 * 题目来源：分布式系统设计面试题
 * 应用场景：负载均衡、分布式缓存、分布式存储系统
 *
 * 核心思想：
 * 1. 将哈希空间组织成一个虚拟的圆环（0 ~ 2^32-1）
 * 2. 服务器节点通过哈希函数映射到环上
 * 3. 数据通过哈希函数映射到环上，顺时针找到最近的服务器节点
 * 4. 虚拟节点技术解决数据分布不均问题
 *
 * 时间复杂度：
 * - 添加节点：O(k) k为虚拟节点数
 * - 删除节点：O(k)
 * - 查找节点：O(log n) n为环上节点总数
 *
 * 空间复杂度：O(n*k) n为物理节点数，k为虚拟节点数
 *
 * 工程化考量：
 * 1. 虚拟节点解决数据倾斜问题
 * 2. 支持节点的动态增删
 * 3. 数据迁移最小化
 * 4. 容错性和可扩展性
 */
public class Code14_ConsistentHashing {

    // 哈希环，存储虚拟节点到物理节点的映射
    private TreeMap<Integer, String> ring = new TreeMap<>();

    // 虚拟节点数量
    private int virtualNodes;

    // 物理节点集合
    private Set<String> physicalNodes = new HashSet<>();
```

```
public Code14_ConsistentHashing(int virtualNodes) {
    this.virtualNodes = virtualNodes;
}

/**
 * 添加物理节点
 * @param node 物理节点标识
 */
public void addNode(String node) {
    if (physicalNodes.contains(node)) {
        return; // 节点已存在
    }

    physicalNodes.add(node);

    // 为每个物理节点创建虚拟节点
    for (int i = 0; i < virtualNodes; i++) {
        String virtualNode = node + "#" + i;
        int hash = getHash(virtualNode);
        ring.put(hash, node);
    }
}

/**
 * 删除物理节点
 * @param node 物理节点标识
 */
public void removeNode(String node) {
    if (!physicalNodes.contains(node)) {
        return; // 节点不存在
    }

    physicalNodes.remove(node);

    // 删除该节点的所有虚拟节点
    for (int i = 0; i < virtualNodes; i++) {
        String virtualNode = node + "#" + i;
        int hash = getHash(virtualNode);
        ring.remove(hash);
    }
}
```

```
/**
 * 根据 key 查找对应的物理节点
 * @param key 数据 key
 * @return 物理节点标识
 */
public String getNode(String key) {
    if (ring.isEmpty()) {
        return null;
    }

    int hash = getHash(key);

    // 在环上查找大于等于该 hash 的第一个节点
    Map.Entry<Integer, String> entry = ring.ceilingEntry(hash);

    // 如果没找到，则返回环的第一个节点（环形结构）
    if (entry == null) {
        entry = ring.firstEntry();
    }

    return entry.getValue();
}

/**
 * 哈希函数：使用 MD5 哈希然后取模
 * @param key 输入字符串
 * @return 哈希值（0 ~ 2^32-1）
 */
private int getHash(String key) {
    try {
        java.security.MessageDigest md = java.security.MessageDigest.getInstance("MD5");
        byte[] digest = md.digest(key.getBytes());
        // 取前 4 个字节作为哈希值
        int hash = 0;
        for (int i = 0; i < 4; i++) {
            hash = (hash << 8) | (digest[i] & 0xFF);
        }
        return hash & 0x7FFFFFFF; // 确保为正数
    } catch (Exception e) {
        // 如果 MD5 不可用，使用简单的哈希函数
        return key.hashCode() & 0x7FFFFFFF;
    }
}
```

```
/**
 * 获取环上节点分布情况（用于调试）
 */
public void printRing() {
    System.out.println("一致性哈希环状态：");
    for (Map.Entry<Integer, String> entry : ring.entrySet()) {
        System.out.println("位置 " + entry.getKey() + " -> " + entry.getValue());
    }
}

/**
 * 测试方法
 */
public static void main(String[] args) {
    // 创建一致性哈希环，每个物理节点有 3 个虚拟节点
    Code14_ConsistentHashing ch = new Code14_ConsistentHashing(3);

    // 添加物理节点
    ch.addNode("Server-A");
    ch.addNode("Server-B");
    ch.addNode("Server-C");

    // 测试数据分布
    String[] testKeys = {"user1", "user2", "user3", "data1", "data2", "data3"};

    System.out.println("=== 初始节点分布测试 ===");
    for (String key : testKeys) {
        String node = ch.getNode(key);
        System.out.println("Key: " + key + " -> Node: " + node);
    }

    // 测试节点删除
    System.out.println("\n=== 删除 Server-B 后测试 ===");
    ch.removeNode("Server-B");

    for (String key : testKeys) {
        String node = ch.getNode(key);
        System.out.println("Key: " + key + " -> Node: " + node);
    }

    // 测试节点添加
    System.out.println("\n=== 添加 Server-D 后测试 ===");
```

```
        ch.addNode("Server-D");

        for (String key : testKeys) {
            String node = ch.getNode(key);
            System.out.println("Key: " + key + " -> Node: " + node);
        }

        // 性能测试
        System.out.println("\n=== 性能测试 ===");
        long startTime = System.currentTimeMillis();
        for (int i = 0; i < 10000; i++) {
            ch.getNode("test" + i);
        }
        long endTime = System.currentTimeMillis();
        System.out.println("10000 次查找耗时: " + (endTime - startTime) + "ms");

        // 打印环状态（调试用）
        ch.printRing();
    }
}
```

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

文件: Code14_ConsistentHashing.py

```
================================================
import hashlib
import bisect
import time

class ConsistentHashing:
    """
    一致性哈希算法实现（Python 版本）

    题目来源：分布式系统设计面试题
    应用场景：负载均衡、分布式缓存、分布式存储系统

    核心思想：
    1. 将哈希空间组织成一个虚拟的圆环（0 ~ 2^32-1）
    2. 服务器节点通过哈希函数映射到环上
    3. 数据通过哈希函数映射到环上，顺时针找到最近的服务器节点
    4. 虚拟节点技术解决数据分布不均问题

    时间复杂度：
```

- 添加节点：O(k) k 为虚拟节点数
- 删除节点：O(k)
- 查找节点：O(log n) n 为环上节点总数

空间复杂度：O(n*k) n 为物理节点数，k 为虚拟节点数

工程化考量：
1. 虚拟节点解决数据倾斜问题
2. 支持节点的动态增删
3. 数据迁移最小化
4. 容错性和可扩展性
"""

```
def __init__(self, virtual_nodes):
    """
    初始化一致性哈希环

    Args:
        virtual_nodes: 每个物理节点的虚拟节点数量
    """
    self.virtual_nodes = virtual_nodes
    # 哈希环，存储虚拟节点到物理节点的映射
    self.ring = {}
    # 排序的哈希键列表，用于快速查找
    self.sorted_keys = []
    # 物理节点集合
    self.physical_nodes = set()

def add_node(self, node):
    """
    添加物理节点

    Args:
        node: 物理节点标识
    """
    if node in self.physical_nodes:
        return  # 节点已存在

    self.physical_nodes.add(node)

    # 为每个物理节点创建虚拟节点
    for i in range(self.virtual_nodes):
        virtual_node = f"{node}#{i}"
```

```
        hash_val = self._get_hash(virtual_node)
        self.ring[hash_val] = node
        # 插入排序位置
        bisect.insort(self.sorted_keys, hash_val)

def remove_node(self, node):
    """
    删除物理节点

    Args:
        node: 物理节点标识
    """
    if node not in self.physical_nodes:
        return  # 节点不存在

    self.physical_nodes.remove(node)

    # 删除该节点的所有虚拟节点
    for i in range(self.virtual_nodes):
        virtual_node = f"{node}#{i}"
        hash_val = self._get_hash(virtual_node)
        if hash_val in self.ring:
            del self.ring[hash_val]
            # 从排序列表中删除
            index = bisect.bisect_left(self.sorted_keys, hash_val)
            if index < len(self.sorted_keys) and self.sorted_keys[index] == hash_val:
                del self.sorted_keys[index]

def get_node(self, key):
    """
    根据 key 查找对应的物理节点

    Args:
        key: 数据 key

    Returns:
        物理节点标识
    """
    if not self.ring:
        return None

    hash_val = self._get_hash(key)
```

```
        # 在环上查找大于等于该 hash 的第一个节点
        index = bisect.bisect_left(self.sorted_keys, hash_val)

        # 如果没找到，则返回环的第一个节点（环形结构）
        if index == len(self.sorted_keys):
            index = 0

        return self.ring[self.sorted_keys[index]]

    def _get_hash(self, key):
        """
        哈希函数：使用 MD5 哈希然后取模

        Args:
            key: 输入字符串

        Returns:
            哈希值（0 ~ 2^32-1）
        """
        # 使用 MD5 哈希
        md5_hash = hashlib.md5(key.encode('utf-8')).hexdigest()
        # 取前 8 个字符（32 位）作为哈希值
        hash_val = int(md5_hash[:8], 16)
        # 确保为正数
        return hash_val & 0x7FFFFFFF

    def print_ring(self):
        """获取环上节点分布情况（用于调试）"""
        print("一致性哈希环状态：")
        for hash_val in self.sorted_keys:
            print(f"位置 {hash_val} -> {self.ring[hash_val]}")

    def get_physical_node_count(self):
        """获取物理节点数量"""
        return len(self.physical_nodes)

    def get_virtual_node_count(self):
        """获取虚拟节点数量"""
        return len(self.ring)


def test_consistent_hashing():
    """
```

```
测试函数
"""
# 创建一致性哈希环，每个物理节点有 3 个虚拟节点
ch = ConsistentHashing(3)

# 添加物理节点
ch.add_node("Server-A")
ch.add_node("Server-B")
ch.add_node("Server-C")

# 测试数据分布
test_keys = ["user1", "user2", "user3", "data1", "data2", "data3"]

print("=== 初始节点分布测试 ===")
for key in test_keys:
    node = ch.get_node(key)
    print(f"Key: {key} -> Node: {node}")

# 测试节点删除
print("\n=== 删除 Server-B 后测试 ===")
ch.remove_node("Server-B")

for key in test_keys:
    node = ch.get_node(key)
    print(f"Key: {key} -> Node: {node}")

# 测试节点添加
print("\n=== 添加 Server-D 后测试 ===")
ch.add_node("Server-D")

for key in test_keys:
    node = ch.get_node(key)
    print(f"Key: {key} -> Node: {node}")

# 性能测试
print("\n=== 性能测试 ===")
start_time = time.time()

for i in range(10000):
    ch.get_node(f"test{i}")

end_time = time.time()
print(f"10000 次查找耗时: {(end_time - start_time) * 1000:.2f}ms")
```

```
    # 打印环状态（调试用）
    ch.print_ring()

    print(f"物理节点数量: {ch.get_physical_node_count()}")
    print(f"虚拟节点数量: {ch.get_virtual_node_count()}")


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

==========

文件: Code15_BloomFilter.java

==========

```
import java.util.BitSet;
import java.util.Random;

/**
 * 布隆过滤器实现（Java 版本）
 *
 * 题目来源：大数据处理、缓存系统、网络爬虫去重
 * 应用场景：网页去重、垃圾邮件过滤、缓存穿透防护
 *
 * 核心思想：
 * 1. 使用多个哈希函数将元素映射到位数组的不同位置
 * 2. 插入元素时，将所有对应位置设为 1
 * 3. 查询元素时，检查所有对应位置是否都为 1
 * 4. 存在一定的误判率（假阳性），但不会漏判（假阴性）
 *
 * 时间复杂度：
 * - 插入：O(k) k 为哈希函数数量
 * - 查询：O(k)
 *
 * 空间复杂度：O(m) m 为位数组大小
 *
 * 工程化考量：
 * 1. 误判率控制：通过调整位数组大小和哈希函数数量
 * 2. 哈希函数选择：独立且分布均匀的哈希函数
 * 3. 内存优化：使用位数组节省空间
 * 4. 并发安全：多线程环境下的线程安全
 */
public class Code15_BloomFilter {
```

```
// 位数组
private BitSet bitSet;

// 位数组大小
private int size;

// 哈希函数数量
private int hashCount;

// 随机种子，用于生成不同的哈希函数
private int[] seeds;

/**
 * 构造函数
 *
 * @param expectedInsertions 预期插入元素数量
 * @param falsePositiveRate 期望的误判率
 */
public Code15_BloomFilter(int expectedInsertions, double falsePositiveRate) {
    if (expectedInsertions <= 0) {
        throw new IllegalArgumentException("预期插入数量必须大于 0");
    }
    if (falsePositiveRate <= 0 || falsePositiveRate >= 1) {
        throw new IllegalArgumentException("误判率必须在 0 和 1 之间");
    }

    // 计算最优的位数组大小和哈希函数数量
    this.size = optimalBitArraySize(expectedInsertions, falsePositiveRate);
    this.hashCount = optimalHashFunctionCount(expectedInsertions, size);

    this.bitSet = new BitSet(size);
    this.seeds = generateSeeds(hashCount);

    System.out.println("布隆过滤器初始化：");
    System.out.println("预期插入数量：" + expectedInsertions);
    System.out.println("期望误判率：" + falsePositiveRate);
    System.out.println("位数组大小：" + size);
    System.out.println("哈希函数数量：" + hashCount);
}

/**
 * 计算最优的位数组大小
```

```
     * 公式：m = - (n * ln(p)) / (ln(2))^2
     */
    private int optimalBitArraySize(int n, double p) {
        return (int) Math.ceil(-n * Math.log(p) / (Math.log(2) * Math.log(2)));
    }

    /**
     * 计算最优的哈希函数数量
     * 公式：k = (m/n) * ln(2)
     */
    private int optimalHashFunctionCount(int n, int m) {
        return Math.max(1, (int) Math.round((double) m / n * Math.log(2)));
    }

    /**
     * 生成随机种子
     */
    private int[] generateSeeds(int count) {
        Random random = new Random(42); // 固定种子保证可重复性
        int[] seeds = new int[count];
        for (int i = 0; i < count; i++) {
            seeds[i] = random.nextInt(Integer.MAX_VALUE);
        }
        return seeds;
    }

    /**
     * 哈希函数：使用 MurmurHash 变种
     */
    private int hash(String element, int seed) {
        int hash = seed;
        for (int i = 0; i < element.length(); i++) {
            hash = hash * 31 + element.charAt(i);
            hash ^= hash >>> 16;
        }
        return Math.abs(hash % size);
    }

    /**
     * 添加元素
     */
    public void add(String element) {
        for (int i = 0; i < hashCount; i++) {
```

```
            int position = hash(element, seeds[i]);
            bitSet.set(position);
        }
    }

    /**
     * 检查元素是否存在
     *
     * @return true: 可能存在（可能有误判）
     *         false: 一定不存在
     */
    public boolean mightContain(String element) {
        for (int i = 0; i < hashCount; i++) {
            int position = hash(element, seeds[i]);
            if (!bitSet.get(position)) {
                return false;
            }
        }
        return true;
    }

    /**
     * 获取位数组使用率
     */
    public double getUsageRate() {
        int usedBits = bitSet.cardinality();
        return (double) usedBits / size;
    }

    /**
     * 清空布隆过滤器
     */
    public void clear() {
        bitSet.clear();
    }

    /**
     * 获取布隆过滤器统计信息
     */
    public void printStats() {
        System.out.println("布隆过滤器统计信息：");
        System.out.println("位数组大小：" + size);
        System.out.println("哈希函数数量：" + hashCount);
```

```
        System.out.println("已使用位数：" + bitSet.cardinality());
        System.out.println("使用率：" + String.format("%.2f%%", getUsageRate() * 100));
    }

    /**
     * 测试方法
     */
    public static void main(String[] args) {
        // 创建布隆过滤器：预期插入10000个元素，误判率0.01
        Code15_BloomFilter bloomFilter = new Code15_BloomFilter(10000, 0.01);

        // 测试数据
        String[] testData = {
            "https://example.com/page1",
            "https://example.com/page2",
            "https://example.com/page3",
            "https://example.com/page4",
            "https://example.com/page5"
        };

        // 添加测试数据
        System.out.println("\n=== 添加测试数据 ===");
        for (String url : testData) {
            bloomFilter.add(url);
            System.out.println("添加：" + url);
        }

        // 测试存在性检查
        System.out.println("\n=== 存在性检查 ===");
        for (String url : testData) {
            boolean exists = bloomFilter.mightContain(url);
            System.out.println("检查 " + url + " : " + (exists ? "可能存在" : "不存在"));
        }

        // 测试不存在的数据
        System.out.println("\n=== 测试不存在的数据 ===");
        String[] nonExistentData = {
            "https://example.com/page999",
            "https://google.com/search",
            "https://github.com/project"
        };

        for (String url : nonExistentData) {
```

```
            boolean exists = bloomFilter.mightContain(url);
            System.out.println("检查 " + url + " : " + (exists ? "可能存在（误判）" : "不存在"));
        }

        // 性能测试
        System.out.println("\n=== 性能测试 ===");
        long startTime = System.currentTimeMillis();

        for (int i = 0; i < 100000; i++) {
            bloomFilter.mightContain("test" + i);
        }

        long endTime = System.currentTimeMillis();
        System.out.println("100000 次查询耗时：" + (endTime - startTime) + "ms");

        // 打印统计信息
        bloomFilter.printStats();

        // 误判率测试
        System.out.println("\n=== 误判率测试 ===");
        int falsePositives = 0;
        int testCount = 10000;

        for (int i = 0; i < testCount; i++) {
            String testStr = "new_data_" + i;
            if (bloomFilter.mightContain(testStr)) {
                falsePositives++;
            }
        }

        double actualFalsePositiveRate = (double) falsePositives / testCount;
        System.out.println("测试数量：" + testCount);
        System.out.println("误判数量：" + falsePositives);
        System.out.println("实际误判率：" + String.format("%.4f", actualFalsePositiveRate));
    }
}

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

文件: Codeforces52C_SegmentTree.java
==========================================
package class109;
```

```
/**
 * Codeforces 52C Circular RMQ
 *
 * 题目描述：
 * 给定一个长度为n的环形数组（即a[0]的前一个元素是a[n-1]，a[n-1]的后一个元素是a[0]），
 * 需要处理以下两种操作：
 * 1. 将区间[l, r]中的每个元素都加上v（如果l > r，则表示环形区间[l, n-1]和[0, r]）
 * 2. 查询区间[l, r]中所有元素的最小值（如果l > r，则表示环形区间[l, n-1]和[0, r]）
 *
 * 解题思路：
 * 这是一个环形线段树问题，需要处理环形区间操作。
 * 1. 对于环形区间操作，如果l > r，可以将其拆分为两个普通区间[l, n-1]和[0, r]
 * 2. 使用线段树配合懒标记来处理区间更新和区间最值查询
 *
 * 时间复杂度分析：
 * - 初始化：O(n)
 * - 区间更新：O(log n)
 * - 区间查询：O(log n)
 *
 * 空间复杂度分析：
 * - O(n)，线段树需要4*n的空间来存储节点信息
 *
 * 链接：https://codeforces.com/contest/52/problem/C
 */
public class Codeforces52C_SegmentTree {

    // 线段树数组，存储区间最小值
    private long[] tree;
    // 懒标记数组，存储区间延迟更新的值
    private long[] lazy;
    // 原数组
    private int[] nums;
    // 数组长度
    private int n;

    /**
     * 构造函数，初始化线段树
     * @param nums 输入数组
     */
    public Codeforces52C_SegmentTree(int[] nums) {
        this.n = nums.length;
        this.nums = nums;
        // 线段树数组大小通常为4*n，确保足够容纳所有节点
```

```
        this.tree = new long[n << 2];
        this.lazy = new long[n << 2];
        // 初始化为最大值
        for (int i = 0; i < (n << 2); i++) {
            tree[i] = Long.MAX_VALUE;
            lazy[i] = 0;
        }
        // 构建线段树
        buildTree(0, n - 1, 1);
    }

    /**
     * 构建线段树
     * @param start 区间起始位置
     * @param end 区间结束位置
     * @param node 当前节点在 tree 数组中的索引
     */
    private void buildTree(int start, int end, int node) {
        // 清空懒标记
        lazy[node] = 0;

        // 如果是叶子节点，直接赋值
        if (start == end) {
            tree[node] = nums[start];
            return;
        }

        // 计算中点
        int mid = (start + end) / 2;
        // 递归构建左右子树
        buildTree(start, mid, node * 2);
        buildTree(mid + 1, end, node * 2 + 1);
        // 合并左右子树信息，取最小值
        tree[node] = Math.min(tree[node * 2], tree[node * 2 + 1]);
    }

    /**
     * 下推懒标记
     * @param node 当前节点
     * @param start 当前节点区间起始位置
     * @param end 当前节点区间结束位置
     */
    private void pushDown(int node, int start, int end) {
```

```
    if (lazy[node] != 0) {
        // 将懒标记下推到左右子节点
        lazy[node * 2] += lazy[node];
        lazy[node * 2 + 1] += lazy[node];

        // 如果不是叶子节点，更新子节点的值
        if (start != end) {
            tree[node * 2] += lazy[node];
            tree[node * 2 + 1] += lazy[node];
        }

        // 清空当前节点的懒标记
        lazy[node] = 0;
    }
}

/**
 * 区间更新
 * @param start 当前节点区间起始位置
 * @param end 当前节点区间结束位置
 * @param node 当前节点在 tree 数组中的索引
 * @param left 更新区间左边界
 * @param right 更新区间右边界
 * @param val 更新值
 */
private void updateRange(int start, int end, int node, int left, int right, long val) {
    // 如果当前区间与更新区间无重叠，直接返回
    if (start > right || end < left) {
        return;
    }

    // 如果当前区间完全包含在更新区间内
    if (start >= left && end <= right) {
        // 更新当前节点的值
        tree[node] += val;
        // 设置懒标记
        if (start != end) {
            lazy[node] += val;
        }
        return;
    }

    // 下推懒标记
```

```
        pushDown(node, start, end);

        // 递归更新左右子树
        int mid = (start + end) / 2;
        updateRange(start, mid, node * 2, left, right, val);
        updateRange(mid + 1, end, node * 2 + 1, left, right, val);

        // 合并左右子树信息，取最小值
        tree[node] = Math.min(tree[node * 2], tree[node * 2 + 1]);
    }

    /**
     * 区间更新接口（处理环形区间）
     * @param left 更新区间左边界
     * @param right 更新区间右边界
     * @param val 更新值
     */
    public void update(int left, int right, long val) {
        // 处理环形区间
        if (left <= right) {
            updateRange(0, n - 1, 1, left, right, val);
        } else {
            // 环形区间拆分为两个普通区间
            updateRange(0, n - 1, 1, left, n - 1, val);
            updateRange(0, n - 1, 1, 0, right, val);
        }
    }

    /**
     * 区间查询
     * @param start 当前节点区间起始位置
     * @param end 当前节点区间结束位置
     * @param node 当前节点在 tree 数组中的索引
     * @param left 查询区间左边界
     * @param right 查询区间右边界
     * @return 区间最小值
     */
    private long queryRange(int start, int end, int node, int left, int right) {
        // 如果当前区间与查询区间无重叠，返回最大值
        if (start > right || end < left) {
            return Long.MAX_VALUE;
        }
```

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

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

        // 递归查询左右子树
        int mid = (start + end) / 2;
        long leftMin = queryRange(start, mid, node * 2, left, right);
        long rightMin = queryRange(mid + 1, end, node * 2 + 1, left, right);

        return Math.min(leftMin, rightMin);
    }

    /**
     * 区间查询接口（处理环形区间）
     * @param left 查询区间左边界
     * @param right 查询区间右边界
     * @return 区间最小值
     */
    public long query(int left, int right) {
        // 处理环形区间
        if (left <= right) {
            return queryRange(0, n - 1, 1, left, right);
        } else {
            // 环形区间拆分为两个普通区间
            long min1 = queryRange(0, n - 1, 1, left, n - 1);
            long min2 = queryRange(0, n - 1, 1, 0, right);
            return Math.min(min1, min2);
        }
    }

    // 测试方法
    public static void main(String[] args) {
        System.out.println("测试 Codeforces 52C 实现...");

        // 测试用例
        int[] nums = {1, 2, 3, 4, 5};
        Codeforces52C_SegmentTree segTree = new Codeforces52C_SegmentTree(nums);

        System.out.println("初始数组: [1, 2, 3, 4, 5]");
```

```
        System.out.println("查询区间[1,3]的最小值: " + segTree.query(1, 3)); // 应该输出 2

        // 区间更新：将区间[1,3]中的每个元素都加上 2
        segTree.update(1, 3, 2);
        System.out.println("将区间[1,3]中的每个元素都加上 2 后:");
        System.out.println("查询区间[1,3]的最小值: " + segTree.query(1, 3)); // 应该输出 4

        // 环形区间查询：查询区间[3,1]（环形）
        System.out.println("环形区间[3,1]的最小值: " + segTree.query(3, 1)); // 应该输出 3

        System.out.println("Codeforces 52C 测试完成！");
    }
}
```

==========

文件: codeforces_52c_segment_tree.py

==========

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

"""
Codeforces 52C Circular RMQ

题目描述：
给定一个长度为 n 的环形数组（即 a[0]的前一个元素是 a[n-1]，a[n-1]的后一个元素是 a[0]），
需要处理以下两种操作：
1. 将区间[l, r]中的每个元素都加上 v（如果 l > r，则表示环形区间[l, n-1]和[0, r]）
2. 查询区间[l, r]中所有元素的最小值（如果 l > r，则表示环形区间[l, n-1]和[0, r]）

解题思路：
这是一个环形线段树问题，需要处理环形区间操作。
1. 对于环形区间操作，如果 l > r，可以将其拆分为两个普通区间[l, n-1]和[0, r]
2. 使用线段树配合懒标记来处理区间更新和区间最值查询

时间复杂度分析：
- 初始化：O(n)
- 区间更新：O(log n)
- 区间查询：O(log n)

空间复杂度分析：
- O(n)，线段树需要 4*n 的空间来存储节点信息
```

链接：https://codeforces.com/contest/52/problem/C
"""

```
class Codeforces52C_SegmentTree:
    """
    Codeforces 52C Circular RMQ 的线段树实现
    """

    def __init__(self, nums):
        """
        构造函数，初始化线段树
        :param nums: 输入数组
        """
        self.n = len(nums)
        self.nums = nums[:]
        # 线段树数组大小通常为 4*n，确保足够容纳所有节点
        self.tree = [float('inf')] * (self.n << 2)
        self.lazy = [0] * (self.n << 2)
        # 构建线段树
        self._build_tree(0, self.n - 1, 1)

    def _build_tree(self, start, end, node):
        """
        构建线段树
        :param start: 区间起始位置
        :param end: 区间结束位置
        :param node: 当前节点在 tree 数组中的索引
        """
        # 清空懒标记
        self.lazy[node] = 0

        # 如果是叶子节点，直接赋值
        if start == end:
            self.tree[node] = self.nums[start]
            return

        # 计算中点
        mid = (start + end) // 2
        # 递归构建左右子树
        self._build_tree(start, mid, node * 2)
        self._build_tree(mid + 1, end, node * 2 + 1)
        # 合并左右子树信息，取最小值
```

```
        self.tree[node] = min(self.tree[node * 2], self.tree[node * 2 + 1])

    def _push_down(self, node, start, end):
        """
        下推懒标记
        :param node: 当前节点
        :param start: 当前节点区间起始位置
        :param end: 当前节点区间结束位置
        """
        if self.lazy[node] != 0:
            # 将懒标记下推到左右子节点
            self.lazy[node * 2] += self.lazy[node]
            self.lazy[node * 2 + 1] += self.lazy[node]

            # 如果不是叶子节点，更新子节点的值
            if start != end:
                self.tree[node * 2] += self.lazy[node]
                self.tree[node * 2 + 1] += self.lazy[node]

            # 清空当前节点的懒标记
            self.lazy[node] = 0

    def _update_range(self, start, end, node, left, right, val):
        """
        区间更新
        :param start: 当前节点区间起始位置
        :param end: 当前节点区间结束位置
        :param node: 当前节点在 tree 数组中的索引
        :param left: 更新区间左边界
        :param right: 更新区间右边界
        :param val: 更新值
        """
        # 如果当前区间与更新区间无重叠，直接返回
        if start > right or end < left:
            return

        # 如果当前区间完全包含在更新区间内
        if start >= left and end <= right:
            # 更新当前节点的值
            self.tree[node] += val
            # 设置懒标记
            if start != end:
                self.lazy[node] += val
```

```
            return

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

        # 递归更新左右子树
        mid = (start + end) // 2
        self._update_range(start, mid, node * 2, left, right, val)
        self._update_range(mid + 1, end, node * 2 + 1, left, right, val)

        # 合并左右子树信息，取最小值
        self.tree[node] = min(self.tree[node * 2], self.tree[node * 2 + 1])

    def update(self, left, right, val):
        """
        区间更新接口（处理环形区间）
        :param left: 更新区间左边界
        :param right: 更新区间右边界
        :param val: 更新值
        """
        # 处理环形区间
        if left <= right:
            self._update_range(0, self.n - 1, 1, left, right, val)
        else:
            # 环形区间拆分为两个普通区间
            self._update_range(0, self.n - 1, 1, left, self.n - 1, val)
            self._update_range(0, self.n - 1, 1, 0, right, val)

    def _query_range(self, start, end, node, left, right):
        """
        区间查询
        :param start: 当前节点区间起始位置
        :param end: 当前节点区间结束位置
        :param node: 当前节点在 tree 数组中的索引
        :param left: 查询区间左边界
        :param right: 查询区间右边界
        :return: 区间最小值
        """
        # 如果当前区间与查询区间无重叠，返回最大值
        if start > right or end < left:
            return float('inf')

        # 如果当前区间完全包含在查询区间内
```

```
        if start >= left and end <= right:
            return self.tree[node]

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

        # 递归查询左右子树
        mid = (start + end) // 2
        left_min = self._query_range(start, mid, node * 2, left, right)
        right_min = self._query_range(mid + 1, end, node * 2 + 1, left, right)

        return min(left_min, right_min)

    def query(self, left, right):
        """
        区间查询接口（处理环形区间）
        :param left: 查询区间左边界
        :param right: 查询区间右边界
        :return: 区间最小值
        """
        # 处理环形区间
        if left <= right:
            return self._query_range(0, self.n - 1, 1, left, right)
        else:
            # 环形区间拆分为两个普通区间
            min1 = self._query_range(0, self.n - 1, 1, left, self.n - 1)
            min2 = self._query_range(0, self.n - 1, 1, 0, right)
            return min(min1, min2)


# 测试函数
def test_solution():
    """测试 Codeforces 52C 实现"""
    print("测试 Codeforces 52C 实现...")

    # 测试用例
    nums = [1, 2, 3, 4, 5]
    seg_tree = Codeforces52C_SegmentTree(nums)

    print(f"初始数组: {nums}")
    print(f"查询区间[1,3]的最小值: {seg_tree.query(1, 3)}")  # 应该输出 2

    # 区间更新：将区间[1,3]中的每个元素都加上 2
```

```
    seg_tree.update(1, 3, 2)
    print("将区间[1,3]中的每个元素都加上 2 后:")
    print(f"查询区间[1,3]的最小值: {seg_tree.query(1, 3)}")  # 应该输出 4

    # 环形区间查询：查询区间[3,1]（环形）
    # 环形区间[3,1]包含元素索引 3,4,0,1，对应值为 4,5,1,2，加上之前的更新后为 4,5,1,4，最小值是 1
    print(f"环形区间[3,1]的最小值: {seg_tree.query(3, 1)}")  # 应该输出 1

    print("Codeforces 52C 测试完成！")


if __name__ == "__main__":
    test_solution()

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

文件: HDU1166_SegmentTree.java
================================================
package class109;

/**
 * HDU 1166 敌兵布阵
 *
 * 题目描述：
 * A 国在海岸线沿直线布置了 N 个工兵营地，每个营地初始有一定数量的士兵。
 * 有两种操作：
 * 1. Add i j: 第 i 个营地增加 j 个士兵
 * 2. Query i j: 查询第 i 到第 j 个营地之间士兵总数
 *
 * 解题思路：
 * 这是一个典型的线段树单点更新、区间查询问题。
 * 1. 构建线段树存储每个区间的士兵总数
 * 2. Add 操作对应线段树的单点更新
 * 3. Query 操作对应线段树的区间查询
 *
 * 时间复杂度分析：
 * - 初始化：O(n)
 * - 单点更新：O(log n)
 * - 区间查询：O(log n)
 *
 * 空间复杂度分析：
 * - O(n)，线段树需要 4*n 的空间来存储节点信息
 *
```

```
 * 链接：http://acm.hdu.edu.cn/showproblem.php?pid=1166
 */
public class HDU1166_SegmentTree {

    // 线段树数组，存储区间和
    private int[] tree;
    // 原数组
    private int[] nums;
    // 数组长度
    private int n;

    /**
     * 构造函数，初始化线段树
     * @param nums 输入数组
     */
    public HDU1166_SegmentTree(int[] nums) {
        this.n = nums.length;
        this.nums = nums;
        // 线段树数组大小通常为 4*n，确保足够容纳所有节点
        this.tree = new int[n << 2];
        // 构建线段树
        buildTree(0, n - 1, 1);
    }

    /**
     * 构建线段树
     * @param start 区间起始位置
     * @param end 区间结束位置
     * @param node 当前节点在 tree 数组中的索引
     */
    private void buildTree(int start, int end, int node) {
        // 如果是叶子节点，直接赋值
        if (start == end) {
            tree[node] = nums[start];
            return;
        }

        // 计算中点
        int mid = (start + end) / 2;
        // 递归构建左右子树
        buildTree(start, mid, node * 2);
        buildTree(mid + 1, end, node * 2 + 1);
        // 合并左右子树信息
```

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

/**
 * 单点更新
 * @param start 当前节点区间起始位置
 * @param end 当前节点区间结束位置
 * @param node 当前节点在 tree 数组中的索引
 * @param index 要更新的位置
 * @param val 增加的值
 */
private void updatePoint(int start, int end, int node, int index, int val) {
    // 如果 index 不在当前区间范围内，直接返回
    if (start > index || end < index) {
        return;
    }

    // 如果是叶子节点，直接增加计数
    if (start == end) {
        tree[node] += val;
        return;
    }

    // 递归更新子节点
    int mid = (start + end) / 2;
    if (index <= mid) {
        updatePoint(start, mid, node * 2, index, val);
    } else {
        updatePoint(mid + 1, end, node * 2 + 1, index, val);
    }

    // 合并左右子树信息
    tree[node] = tree[node * 2] + tree[node * 2 + 1];
}

/**
 * 单点更新接口
 * @param index 要更新的位置
 * @param val 增加的值
 */
public void add(int index, int val) {
    updatePoint(0, n - 1, 1, index, val);
}
```

```
/**
 * 区间查询
 * @param start 当前节点区间起始位置
 * @param end 当前节点区间结束位置
 * @param node 当前节点在 tree 数组中的索引
 * @param left 查询区间左边界
 * @param right 查询区间右边界
 * @return 区间和
 */
private int queryRange(int start, int end, int node, int left, int right) {
    // 如果当前区间与查询区间无重叠，返回 0
    if (start > right || end < left) {
        return 0;
    }

    // 如果当前区间完全包含在查询区间内，返回当前节点的值
    if (start >= left && end <= right) {
        return tree[node];
    }

    // 递归查询左右子树
    int mid = (start + end) / 2;
    int leftSum = queryRange(start, mid, node * 2, left, right);
    int rightSum = queryRange(mid + 1, end, node * 2 + 1, left, right);

    return leftSum + rightSum;
}

/**
 * 区间查询接口
 * @param left 查询区间左边界
 * @param right 查询区间右边界
 * @return 区间和
 */
public int query(int left, int right) {
    return queryRange(0, n - 1, 1, left, right);
}

// 测试方法
public static void main(String[] args) {
    System.out.println("测试 HDU 1166 实现...");
```

```
        // 测试用例
        int[] nums = {1, 2, 3, 4, 5};
        HDU1166_SegmentTree segTree = new HDU1166_SegmentTree(nums);

        System.out.println("初始数组: [1, 2, 3, 4, 5]");
        System.out.println("查询区间[1,3]的和: " + segTree.query(1, 3)); // 应该输出 9

        // 单点更新：第 2 个营地增加 3 个士兵
        segTree.add(2, 3);
        System.out.println("第 2 个营地增加 3 个士兵后:");
        System.out.println("查询区间[1,3]的和: " + segTree.query(1, 3)); // 应该输出 12

        System.out.println("HDU 1166 测试完成！");
    }
}
```

==========

文件: hdu1166_segment_tree.py

==========

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

"""
HDU 1166 敌兵布阵

题目描述：
A 国在海岸线沿直线布置了 N 个工兵营地，每个营地初始有一定数量的士兵。
有两种操作：
1. Add i j: 第 i 个营地增加 j 个士兵
2. Query i j: 查询第 i 到第 j 个营地之间士兵总数

解题思路：
这是一个典型的线段树单点更新、区间查询问题。
1. 构建线段树存储每个区间的士兵总数
2. Add 操作对应线段树的单点更新
3. Query 操作对应线段树的区间查询

时间复杂度分析：
- 初始化：O(n)
- 单点更新：O(log n)
- 区间查询：O(log n)
```

空间复杂度分析：
- O(n)，线段树需要 4*n 的空间来存储节点信息

链接：http://acm.hdu.edu.cn/showproblem.php?pid=1166
"""

```
class HDU1166_SegmentTree:
    """
    HDU 1166 敌兵布阵的线段树实现
    """

    def __init__(self, nums):
        """
        构造函数，初始化线段树
        :param nums: 输入数组
        """
        self.n = len(nums)
        self.nums = nums[:]
        # 线段树数组大小通常为 4*n，确保足够容纳所有节点
        self.tree = [0] * (self.n << 2)
        # 构建线段树
        self._build_tree(0, self.n - 1, 1)

    def _build_tree(self, start, end, node):
        """
        构建线段树
        :param start: 区间起始位置
        :param end: 区间结束位置
        :param node: 当前节点在 tree 数组中的索引
        """
        # 如果是叶子节点，直接赋值
        if start == end:
            self.tree[node] = self.nums[start]
            return

        # 计算中点
        mid = (start + end) // 2
        # 递归构建左右子树
        self._build_tree(start, mid, node * 2)
        self._build_tree(mid + 1, end, node * 2 + 1)
        # 合并左右子树信息
        self.tree[node] = self.tree[node * 2] + self.tree[node * 2 + 1]
```

```
def _update_point(self, start, end, node, index, val):
    """
    单点更新
    :param start: 当前节点区间起始位置
    :param end: 当前节点区间结束位置
    :param node: 当前节点在 tree 数组中的索引
    :param index: 要更新的位置
    :param val: 增加的值
    """
    # 如果 index 不在当前区间范围内，直接返回
    if start > index or end < index:
        return

    # 如果是叶子节点，直接增加计数
    if start == end:
        self.tree[node] += val
        return

    # 递归更新子节点
    mid = (start + end) // 2
    if index <= mid:
        self._update_point(start, mid, node * 2, index, val)
    else:
        self._update_point(mid + 1, end, node * 2 + 1, index, val)

    # 合并左右子树信息
    self.tree[node] = self.tree[node * 2] + self.tree[node * 2 + 1]

def add(self, index, val):
    """
    单点更新接口
    :param index: 要更新的位置
    :param val: 增加的值
    """
    self._update_point(0, self.n - 1, 1, index, val)

def _query_range(self, start, end, node, left, right):
    """
    区间查询
    :param start: 当前节点区间起始位置
    :param end: 当前节点区间结束位置
    :param node: 当前节点在 tree 数组中的索引
```

```
        :param left: 查询区间左边界
        :param right: 查询区间右边界
        :return: 区间和
        """
        # 如果当前区间与查询区间无重叠，返回0
        if start > right or end < left:
            return 0

        # 如果当前区间完全包含在查询区间内，返回当前节点的值
        if start >= left and end <= right:
            return self.tree[node]

        # 递归查询左右子树
        mid = (start + end) // 2
        left_sum = self._query_range(start, mid, node * 2, left, right)
        right_sum = self._query_range(mid + 1, end, node * 2 + 1, left, right)

        return left_sum + right_sum

    def query(self, left, right):
        """
        区间查询接口
        :param left: 查询区间左边界
        :param right: 查询区间右边界
        :return: 区间和
        """
        return self._query_range(0, self.n - 1, 1, left, right)


# 测试函数
def test_solution():
    """测试HDU 1166实现"""
    print("测试HDU 1166实现...")

    # 测试用例
    nums = [1, 2, 3, 4, 5]
    seg_tree = HDU1166_SegmentTree(nums)

    print(f"初始数组: {nums}")
    print(f"查询区间[1,3]的和: {seg_tree.query(1, 3)}")  # 应该输出9

    # 单点更新：第2个营地增加3个士兵
    seg_tree.add(2, 3)
```

```
    print("第 2 个营地增加 3 个士兵后:")
    print(f"查询区间[1,3]的和: {seg_tree.query(1, 3)}")  # 应该输出 12

    print("HDU 1166 测试完成! ")


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

==========

文件: LeetCode307_SegmentTree.java

==========

```
package class109;

/**
 * LeetCode 307. 区域和检索 - 数组可修改
 *
 * 题目描述:
 * 给你一个数组 nums ，请你完成两类查询。
 * 1. 其中一类查询要求更新数组 nums 下标对应的值
 * 2. 另一类查询要求返回数组 nums 中索引 left 和索引 right 之间（包含）的 nums 元素的和，其中 left <= right
 *
 * 实现 NumArray 类:
 * - NumArray(int[] nums) 用整数数组 nums 初始化对象
 * - void update(int index, int val) 将 nums[index] 的值更新为 val
 * - int sumRange(int left, int right) 返回数组 nums 中索引 left 和索引 right 之间（包含）的 nums 元素的和
 *
 * 解题思路:
 * 使用线段树来解决这个问题。线段树是一种非常适合处理区间查询和更新操作的数据结构。
 * 1. 构建线段树：将数组元素组织成一棵二叉树，每个节点存储对应区间的和
 * 2. 更新操作：从叶子节点开始向上更新所有包含该元素的区间和
 * 3. 查询操作：根据查询区间与当前节点区间的重叠关系进行递归查询
 *
 * 时间复杂度分析:
 * - 初始化：O(n)，需要构建线段树
 * - 更新操作：O(log n)，最多需要更新从叶子节点到根节点路径上的所有节点
 * - 查询操作：O(log n)，最多需要访问 O(log n)个节点
 *
 * 空间复杂度分析:
 * - O(n)，线段树需要 4*n 的空间来存储节点信息
```

```
 *
 * 链接：https://leetcode.cn/problems/range-sum-query-mutable
 */
public class LeetCode307_SegmentTree {

    // 线段树数组，存储区间和
    private int[] tree;
    // 原数组
    private int[] nums;
    // 数组长度
    private int n;

    /**
     * 构造函数，初始化线段树
     * @param nums 输入数组
     */
    public LeetCode307_SegmentTree(int[] nums) {
        this.n = nums.length;
        this.nums = nums;
        // 线段树数组大小通常为 4*n，确保足够容纳所有节点
        this.tree = new int[n << 2];
        // 构建线段树
        buildTree(0, n - 1, 1);
    }

    /**
     * 构建线段树
     * @param start 区间起始位置
     * @param end 区间结束位置
     * @param node 当前节点在 tree 数组中的索引
     */
    private void buildTree(int start, int end, int node) {
        // 如果是叶子节点，直接赋值
        if (start == end) {
            tree[node] = nums[start];
            return;
        }

        // 计算中点
        int mid = start + (end - start) / 2;
        // 递归构建左右子树
        buildTree(start, mid, node * 2);
        buildTree(mid + 1, end, node * 2 + 1);
```

```
    // 合并左右子树信息
    tree[node] = tree[node * 2] + tree[node * 2 + 1];
}

/**
 * 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) {
    // 计算差值
    int diff = val - nums[index];
    // 更新原数组
    nums[index] = val;
    // 更新线段树
    updateTree(0, n - 1, 1, index, diff);
}

/**
 * 更新线段树中的值
 * @param start 当前节点区间起始位置
 * @param end 当前节点区间结束位置
 * @param node 当前节点在 tree 数组中的索引
 * @param index 要更新的位置
 * @param diff 差值
 */
private void updateTree(int start, int end, int node, int index, int diff) {
    // 如果 index 不在当前区间范围内，直接返回
    if (start > index || end < index) {
        return;
    }

    // 更新当前节点的值
    tree[node] += diff;

    // 如果不是叶子节点，递归更新子节点
    if (start != end) {
        int mid = start + (end - start) / 2;
        updateTree(start, mid, node * 2, index, diff);
        updateTree(mid + 1, end, node * 2 + 1, index, diff);
    }
}
```

```
/**
 * 查询区间和
 * @param left 查询区间左边界
 * @param right 查询区间右边界
 * @return 区间和
 */
public int sumRange(int left, int right) {
    return queryTree(0, n - 1, 1, left, right);
}

/**
 * 查询线段树中指定区间的和
 * @param start 当前节点区间起始位置
 * @param end 当前节点区间结束位置
 * @param node 当前节点在 tree 数组中的索引
 * @param left 查询区间左边界
 * @param right 查询区间右边界
 * @return 区间和
 */
private int queryTree(int start, int end, int node, int left, int right) {
    // 如果当前区间与查询区间无重叠，返回 0
    if (start > right || end < left) {
        return 0;
    }

    // 如果当前区间完全包含在查询区间内，返回当前节点的值
    if (start >= left && end <= right) {
        return tree[node];
    }

    // 递归查询左右子树
    int mid = start + (end - start) / 2;
    int leftSum = queryTree(start, mid, node * 2, left, right);
    int rightSum = queryTree(mid + 1, end, node * 2 + 1, left, right);

    return leftSum + rightSum;
}

// 测试方法
public static void main(String[] args) {
    // 测试用例
    int[] nums = {1, 3, 5};
    LeetCode307_SegmentTree numArray = new LeetCode307_SegmentTree(nums);
```

```
        System.out.println("初始数组: [1, 3, 5]");
        System.out.println("查询区间[0,2]的和: " + numArray.sumRange(0, 2)); // 应该输出 9

        // 更新索引 1 的值为 2
        numArray.update(1, 2);
        System.out.println("将索引 1 的值更新为 2 后:");
        System.out.println("查询区间[0,2]的和: " + numArray.sumRange(0, 2)); // 应该输出 8
    }
}
```

==========

文件: leetcode307_segment_tree.cpp

==========

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

/**
 * LeetCode 307. 区域和检索 - 数组可修改
 *
 * 题目描述：
 * 给你一个数组 nums ，请你完成两类查询。
 * 1. 其中一类查询要求更新数组 nums 下标对应的值
 * 2. 另一类查询要求返回数组 nums 中索引 left 和索引 right 之间（包含）的 nums 元素的和，其中 left <= right
 *
 * 实现 NumArray 类：
 * - NumArray(int[] nums) 用整数数组 nums 初始化对象
 * - void update(int index, int val) 将 nums[index] 的值更新为 val
 * - int sumRange(int left, int right) 返回数组 nums 中索引 left 和索引 right 之间（包含）的 nums 元素的和
 *
 * 解题思路：
 * 使用线段树来解决这个问题。线段树是一种非常适合处理区间查询和更新操作的数据结构。
 * 1. 构建线段树：将数组元素组织成一棵二叉树，每个节点存储对应区间的和
 * 2. 更新操作：从叶子节点开始向上更新所有包含该元素的区间和
 * 3. 查询操作：根据查询区间与当前节点区间的重叠关系进行递归查询
 *
 * 时间复杂度分析：
 * - 初始化：O(n)，需要构建线段树
 * - 更新操作：O(log n)，最多需要更新从叶子节点到根节点路径上的所有节点
```

```
 * - 查询操作：O(log n)，最多需要访问O(log n)个节点
 *
 * 空间复杂度分析：
 * - O(n)，线段树需要4*n的空间来存储节点信息
 *
 * 链接：https://leetcode.cn/problems/range-sum-query-mutable
 */
class LeetCode307_SegmentTree {
private:
    vector<int> tree;  // 线段树数组，存储区间和
    vector<int> nums;  // 原数组
    int n;             // 数组长度

    /**
     * 构建线段树
     * @param start 区间起始位置
     * @param end 区间结束位置
     * @param node 当前节点在tree数组中的索引
     */
    void buildTree(int start, int end, int node) {
        // 如果是叶子节点，直接赋值
        if (start == end) {
            tree[node] = nums[start];
            return;
        }

        // 计算中点
        int mid = start + (end - start) / 2;
        // 递归构建左右子树
        buildTree(start, mid, node * 2);
        buildTree(mid + 1, end, node * 2 + 1);
        // 合并左右子树信息
        tree[node] = tree[node * 2] + tree[node * 2 + 1];
    }

    /**
     * 更新线段树中的值
     * @param start 当前节点区间起始位置
     * @param end 当前节点区间结束位置
     * @param node 当前节点在tree数组中的索引
     * @param index 要更新的位置
     * @param diff 差值
     */
```

```
void updateTree(int start, int end, int node, int index, int diff) {
    // 如果 index 不在当前区间范围内，直接返回
    if (start > index || end < index) {
        return;
    }

    // 更新当前节点的值
    tree[node] += diff;

    // 如果不是叶子节点，递归更新子节点
    if (start != end) {
        int mid = start + (end - start) / 2;
        updateTree(start, mid, node * 2, index, diff);
        updateTree(mid + 1, end, node * 2 + 1, index, diff);
    }
}

/**
 * 查询线段树中指定区间的和
 * @param start 当前节点区间起始位置
 * @param end 当前节点区间结束位置
 * @param node 当前节点在 tree 数组中的索引
 * @param left 查询区间左边界
 * @param right 查询区间右边界
 * @return 区间和
 */
int queryTree(int start, int end, int node, int left, int right) {
    // 如果当前区间与查询区间无重叠，返回 0
    if (start > right || end < left) {
        return 0;
    }

    // 如果当前区间完全包含在查询区间内，返回当前节点的值
    if (start >= left && end <= right) {
        return tree[node];
    }

    // 递归查询左右子树
    int mid = start + (end - start) / 2;
    int leftSum = queryTree(start, mid, node * 2, left, right);
    int rightSum = queryTree(mid + 1, end, node * 2 + 1, left, right);

    return leftSum + rightSum;
```

```
    }

public:
    /**
     * 构造函数，初始化线段树
     * @param nums 输入数组
     */
    LeetCode307_SegmentTree(vector<int>& nums) {
        this->n = nums.size();
        this->nums = nums;
        // 线段树数组大小通常为4*n，确保足够容纳所有节点
        this->tree.resize(n << 2);
        // 构建线段树
        buildTree(0, n - 1, 1);
    }

    /**
     * 更新数组中指定位置的值
     * @param index 要更新的位置
     * @param val 新的值
     */
    void update(int index, int val) {
        // 计算差值
        int diff = val - nums[index];
        // 更新原数组
        nums[index] = val;
        // 更新线段树
        updateTree(0, n - 1, 1, index, diff);
    }

    /**
     * 查询区间和
     * @param left 查询区间左边界
     * @param right 查询区间右边界
     * @return 区间和
     */
    int sumRange(int left, int right) {
        return queryTree(0, n - 1, 1, left, right);
    }
};

// 测试函数
int main() {
```

```
    cout << "测试 LeetCode 307 实现..." << endl;

    // 测试用例
    vector<int> nums = {1, 3, 5};
    LeetCode307_SegmentTree numArray(nums);

    cout << "初始数组: [1, 3, 5]" << endl;
    cout << "查询区间[0,2]的和: " << numArray.sumRange(0, 2) << endl; // 应该输出 9

    // 更新索引 1 的值为 2
    numArray.update(1, 2);
    cout << "将索引 1 的值更新为 2 后:" << endl;
    cout << "查询区间[0,2]的和: " << numArray.sumRange(0, 2) << endl; // 应该输出 8

    cout << "LeetCode 307 测试完成！" << endl;

    return 0;
}

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

文件: leetcode307_segment_tree.py
==========================================
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

"""
LeetCode 307. 区域和检索 - 数组可修改

题目描述：
给你一个数组 nums ，请你完成两类查询。
1. 其中一类查询要求更新数组 nums 下标对应的值
2. 另一类查询要求返回数组 nums 中索引 left 和索引 right 之间（包含）的 nums 元素的和，其中 left <= right

实现 NumArray 类：
- NumArray(int[] nums) 用整数数组 nums 初始化对象
- void update(int index, int val) 将 nums[index] 的值更新为 val
- int sumRange(int left, int right) 返回数组 nums 中索引 left 和索引 right 之间（包含）的 nums 元素的和

解题思路：
使用线段树来解决这个问题。线段树是一种非常适合处理区间查询和更新操作的数据结构。
```

1. 构建线段树：将数组元素组织成一棵二叉树，每个节点存储对应区间的和
2. 更新操作：从叶子节点开始向上更新所有包含该元素的区间和
3. 查询操作：根据查询区间与当前节点区间的重叠关系进行递归查询

时间复杂度分析：
- 初始化：O(n)，需要构建线段树
- 更新操作：O(log n)，最多需要更新从叶子节点到根节点路径上的所有节点
- 查询操作：O(log n)，最多需要访问 O(log n)个节点

空间复杂度分析：
- O(n)，线段树需要 4*n 的空间来存储节点信息

链接：https://leetcode.cn/problems/range-sum-query-mutable
"""

```
class LeetCode307_SegmentTree:
    """
    LeetCode 307. 区域和检索 - 数组可修改的线段树实现
    """

    def __init__(self, nums):
        """
        构造函数，初始化线段树
        :param nums: 输入数组
        """
        self.n = len(nums)
        self.nums = nums[:]
        # 线段树数组大小通常为 4*n，确保足够容纳所有节点
        self.tree = [0] * (self.n << 2)
        # 构建线段树
        self._build_tree(0, self.n - 1, 1)

    def _build_tree(self, start, end, node):
        """
        构建线段树
        :param start: 区间起始位置
        :param end: 区间结束位置
        :param node: 当前节点在 tree 数组中的索引
        """
        # 如果是叶子节点，直接赋值
        if start == end:
            self.tree[node] = self.nums[start]
```

```
        return

    # 计算中点
    mid = start + (end - start) // 2
    # 递归构建左右子树
    self._build_tree(start, mid, node * 2)
    self._build_tree(mid + 1, end, node * 2 + 1)
    # 合并左右子树信息
    self.tree[node] = self.tree[node * 2] + self.tree[node * 2 + 1]

def update(self, index, val):
    """
    更新数组中指定位置的值
    :param index: 要更新的位置
    :param val: 新的值
    """
    # 计算差值
    diff = val - self.nums[index]
    # 更新原数组
    self.nums[index] = val
    # 更新线段树
    self._update_tree(0, self.n - 1, 1, index, diff)

def _update_tree(self, start, end, node, index, diff):
    """
    更新线段树中的值
    :param start: 当前节点区间起始位置
    :param end: 当前节点区间结束位置
    :param node: 当前节点在 tree 数组中的索引
    :param index: 要更新的位置
    :param diff: 差值
    """
    # 如果 index 不在当前区间范围内，直接返回
    if start > index or end < index:
        return

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

    # 如果不是叶子节点，递归更新子节点
    if start != end:
        mid = start + (end - start) // 2
        self._update_tree(start, mid, node * 2, index, diff)
```

```
            self._update_tree(mid + 1, end, node * 2 + 1, index, diff)

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

    def _query_tree(self, start, end, node, left, right):
        """
        查询线段树中指定区间的和
        :param start: 当前节点区间起始位置
        :param end: 当前节点区间结束位置
        :param node: 当前节点在 tree 数组中的索引
        :param left: 查询区间左边界
        :param right: 查询区间右边界
        :return: 区间和
        """
        # 如果当前区间与查询区间无重叠，返回 0
        if start > right or end < left:
            return 0

        # 如果当前区间完全包含在查询区间内，返回当前节点的值
        if start >= left and end <= right:
            return self.tree[node]

        # 递归查询左右子树
        mid = start + (end - start) // 2
        left_sum = self._query_tree(start, mid, node * 2, left, right)
        right_sum = self._query_tree(mid + 1, end, node * 2 + 1, left, right)

        return left_sum + right_sum


# 测试函数
def test_solution():
    """测试 LeetCode 307 实现"""
    print("测试 LeetCode 307 实现...")

    # 测试用例
```

```
    nums = [1, 3, 5]
    numArray = LeetCode307_SegmentTree(nums)

    print("初始数组: [1, 3, 5]")
    print(f"查询区间[0,2]的和: {numArray.sumRange(0, 2)}")  # 应该输出 9

    # 更新索引 1 的值为 2
    numArray.update(1, 2)
    print("将索引 1 的值更新为 2 后:")
    print(f"查询区间[0,2]的和: {numArray.sumRange(0, 2)}")  # 应该输出 8

    print("LeetCode 307 测试完成！")


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

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

文件: LeetCode315_SegmentTree.java

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

```
package class109;

import java.util.*;

/**
 * LeetCode 315. 计算右侧小于当前元素的个数
 *
 * 题目描述：
 * 给你一个整数数组 nums ，按要求返回一个新数组 counts 。数组 counts 有该性质：
 * counts[i] 的值是 nums[i] 右侧小于 nums[i] 的元素的数量。
 *
 * 示例：
 * 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
 *
 * 解题思路：
 * 使用权值线段树来解决这个问题。
```

```
 * 1. 离散化：由于数组元素可能很大，需要先进行离散化处理，将元素映射到连续的小范围
 * 2. 权值线段树：线段树的每个节点存储某个值域范围内元素出现的次数
 * 3. 从右向左遍历数组，在权值线段树中查询比当前元素小的元素个数，然后将当前元素插入线段树
 *
 * 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：
 * - O(n)，线段树需要 4*n 的空间来存储节点信息
 *
 * 链接：https://leetcode.cn/problems/count-of-smaller-numbers-after-self
 */
public class LeetCode315_SegmentTree {

    // 线段树数组，存储区间元素个数
    private int[] tree;
    // 离散化后的数组大小
    private int n;

    /**
     * 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) {
        List<Integer> result = new ArrayList<>();
        if (nums == null || nums.length == 0) {
            return result;
        }

        // 离散化处理
        int[] sorted = nums.clone();
        Arrays.sort(sorted);
        // 去重
        int[] unique = Arrays.stream(sorted).distinct().toArray();
        n = unique.length;

        // 线段树数组大小通常为 4*n，确保足够容纳所有节点
        tree = new int[n << 2];

        // 从右向左遍历数组
```

```
    for (int i = nums.length - 1; i >= 0; i--) {
        // 找到当前元素在离散化数组中的位置
        int index = Arrays.binarySearch(unique, nums[i]);
        // 查询比当前元素小的元素个数（即查询[0, index-1]区间内的元素个数）
        int count = query(0, n - 1, 1, 0, index - 1);
        result.add(count);
        // 将当前元素插入线段树
        update(0, n - 1, 1, index);
    }

    // 由于是从右向左遍历的，需要反转结果
    Collections.reverse(result);
    return result;
}

/**
 * 更新线段树中的值（单点更新）
 * @param start 当前节点区间起始位置
 * @param end 当前节点区间结束位置
 * @param node 当前节点在 tree 数组中的索引
 * @param index 要更新的位置
 */
private void update(int start, int end, int node, int index) {
    // 如果 index 不在当前区间范围内，直接返回
    if (start > index || end < index) {
        return;
    }

    // 如果是叶子节点，直接增加计数
    if (start == end) {
        tree[node]++;
        return;
    }

    // 递归更新子节点
    int mid = start + (end - start) / 2;
    if (index <= mid) {
        update(start, mid, node * 2, index);
    } else {
        update(mid + 1, end, node * 2 + 1, index);
    }

    // 合并左右子树信息
```

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

    /**
     * 查询线段树中指定区间的元素个数
     * @param start 当前节点区间起始位置
     * @param end 当前节点区间结束位置
     * @param node 当前节点在 tree 数组中的索引
     * @param left 查询区间左边界
     * @param right 查询区间右边界
     * @return 区间内元素个数
     */
    private int query(int start, int end, int node, int left, int right) {
        // 如果查询区间无效或当前区间与查询区间无重叠，返回 0
        if (left > right || start > right || end < left) {
            return 0;
        }

        // 如果当前区间完全包含在查询区间内，返回当前节点的值
        if (start >= left && end <= right) {
            return tree[node];
        }

        // 递归查询左右子树
        int mid = start + (end - start) / 2;
        int leftCount = query(start, mid, node * 2, left, right);
        int rightCount = query(mid + 1, end, node * 2 + 1, left, right);

        return leftCount + rightCount;
    }

    // 测试方法
    public static void main(String[] args) {
        // 测试用例
        LeetCode315_SegmentTree solution = new LeetCode315_SegmentTree();
        int[] nums = {5, 2, 6, 1};

        System.out.println("输入数组: [5, 2, 6, 1]");
        List<Integer> result = solution.countSmaller(nums);
        System.out.println("输出结果: " + result); // 应该输出[2, 1, 1, 0]
    }
}
```

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

文件: leetcode315_segment_tree.py
================================================
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

"""
LeetCode 315. 计算右侧小于当前元素的个数

题目描述：
给你一个整数数组 nums ，按要求返回一个新数组 counts 。数组 counts 有该性质：
counts[i] 的值是 nums[i] 右侧小于 nums[i] 的元素的数量。

示例：
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

解题思路：
使用权值线段树来解决这个问题。
1. 离散化：由于数组元素可能很大，需要先进行离散化处理，将元素映射到连续的小范围
2. 权值线段树：线段树的每个节点存储某个值域范围内元素出现的次数
3. 从右向左遍历数组，在权值线段树中查询比当前元素小的元素个数，然后将当前元素插入线段树

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：
- O(n)，线段树需要 4*n 的空间来存储节点信息

链接：https://leetcode.cn/problems/count-of-smaller-numbers-after-self
"""



class LeetCode315_SegmentTree:
    """
```

```
LeetCode 315. 计算右侧小于当前元素的个数的线段树实现
"""

def __init__(self):
    """构造函数"""
    self.tree = []  # 线段树数组，存储区间元素个数
    self.n = 0      # 离散化后的数组大小

def countSmaller(self, nums):
    """
    计算右侧小于当前元素的个数
    :param nums: 输入数组
    :return: 结果数组
    """
    result = []
    if not nums:
        return result

    # 离散化处理
    sorted_nums = sorted(nums)
    # 去重
    unique = list(dict.fromkeys(sorted_nums))  # 保持顺序的去重
    self.n = len(unique)

    # 线段树数组大小通常为4*n，确保足够容纳所有节点
    self.tree = [0] * (self.n << 2)

    # 从右向左遍历数组
    for i in range(len(nums) - 1, -1, -1):
        # 找到当前元素在离散化数组中的位置
        index = self._binary_search(unique, nums[i])
        # 查询比当前元素小的元素个数（即查询[0, index-1]区间内的元素个数）
        count = self._query(0, self.n - 1, 1, 0, index - 1)
        result.append(count)
        # 将当前元素插入线段树
        self._update(0, self.n - 1, 1, index)

    # 由于是从右向左遍历的，需要反转结果
    result.reverse()
    return result

def _binary_search(self, arr, target):
    """
```

```
    二分查找元素在数组中的位置
    :param arr: 已排序数组
    :param target: 目标值
    :return: 目标值在数组中的索引
    """
    left, right = 0, len(arr) - 1
    while left <= right:
        mid = left + (right - left) // 2
        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            left = mid + 1
        else:
            right = mid - 1
    return left


def _update(self, start, end, node, index):
    """
    更新线段树中的值（单点更新）
    :param start: 当前节点区间起始位置
    :param end: 当前节点区间结束位置
    :param node: 当前节点在 tree 数组中的索引
    :param index: 要更新的位置
    """
    # 如果 index 不在当前区间范围内，直接返回
    if start > index or end < index:
        return

    # 如果是叶子节点，直接增加计数
    if start == end:
        self.tree[node] += 1
        return

    # 递归更新子节点
    mid = start + (end - start) // 2
    if index <= mid:
        self._update(start, mid, node * 2, index)
    else:
        self._update(mid + 1, end, node * 2 + 1, index)

    # 合并左右子树信息
    self.tree[node] = self.tree[node * 2] + self.tree[node * 2 + 1]
```

```
    def _query(self, start, end, node, left, right):
        """
        查询线段树中指定区间的元素个数
        :param start: 当前节点区间起始位置
        :param end: 当前节点区间结束位置
        :param node: 当前节点在 tree 数组中的索引
        :param left: 查询区间左边界
        :param right: 查询区间右边界
        :return: 区间内元素个数
        """
        # 如果查询区间无效或当前区间与查询区间无重叠，返回 0
        if left > right or start > right or end < left:
            return 0

        # 如果当前区间完全包含在查询区间内，返回当前节点的值
        if start >= left and end <= right:
            return self.tree[node]

        # 递归查询左右子树
        mid = start + (end - start) // 2
        left_count = self._query(start, mid, node * 2, left, right)
        right_count = self._query(mid + 1, end, node * 2 + 1, left, right)

        return left_count + right_count


# 测试函数
def test_solution():
    """测试 LeetCode 315 实现"""
    print("测试 LeetCode 315 实现...")

    # 测试用例
    solution = LeetCode315_SegmentTree()
    nums = [5, 2, 6, 1]

    print(f"输入数组: {nums}")
    result = solution.countSmaller(nums)
    print(f"输出结果: {result}")  # 应该输出[2, 1, 1, 0]

    print("LeetCode 315 测试完成！")


if __name__ == "__main__":
```

```
    test_solution()

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

文件: LeetCode327_SegmentTree.java
================================================
package class109;

import java.util.*;

/**
 * LeetCode 327. 区间和的个数
 *
 * 题目描述：
 * 给你一个整数数组 nums 以及两个整数 lower 和 upper 。
 * 求数组中，值位于范围 [lower, upper] （包含 lower 和 upper）之内的区间和的个数。
 * 区间和 S(i, j) 表示在 nums 中，位置从 i 到 j 的元素之和，包含 i 和 j (i ≤ j)。
 *
 * 示例：
 * 输入：nums = [-2,5,-1], lower = -2, upper = 2
 * 输出：3
 * 解释：存在三个区间：[0,0]、[2,2] 和 [0,2] ，对应的区间和分别是：-2 、-1 、2 。
 *
 * 解题思路：
 * 使用权值线段树来解决这个问题。
 * 1. 计算前缀和数组，将区间和问题转化为前缀和差值问题
 * 2. 离散化：由于前缀和可能很大，需要先进行离散化处理
 * 3. 权值线段树：线段树的每个节点存储某个值域范围内前缀和出现的次数
 * 4. 从左向右遍历前缀和数组，在权值线段树中查询满足条件的前缀和个数，然后将当前前缀和插入线段树
 *
 * 时间复杂度分析：
 * - 计算前缀和：O(n)
 * - 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：
 * - O(n)，线段树需要 4*n 的空间来存储节点信息
 *
 * 链接：https://leetcode.cn/problems/count-of-range-sum
 */
public class LeetCode327_SegmentTree {
```

```
// 线段树数组，存储区间元素个数
private int[] tree;
// 离散化后的数组
private long[] sorted;
// 离散化后的数组大小
private int n;

/**
 * 计算区间和的个数
 * @param nums 输入数组
 * @param lower 下界
 * @param upper 上界
 * @return 区间和的个数
 */
public int countRangeSum(int[] nums, int lower, int upper) {
    if (nums == null || nums.length == 0) {
        return 0;
    }

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

    // 离散化处理
    TreeSet<Long> set = new TreeSet<>();
    for (long sum : prefixSum) {
        set.add(sum);
        set.add(sum - lower);
        set.add(sum - upper);
    }
    sorted = new long[set.size()];
    int index = 0;
    for (long num : set) {
        sorted[index++] = num;
    }
    n = sorted.length;

    // 线段树数组大小通常为 4*n，确保足够容纳所有节点
    tree = new int[n << 2];
```

```
    int result = 0;
    // 从左向右遍历前缀和数组
    for (int i = 0; i < prefixSum.length; i++) {
        // 查询满足条件 prefixSum[j] - prefixSum[i] 在 [lower, upper] 范围内的 j 个数
        // 即查询 prefixSum[j] 在 [prefixSum[i] + lower, prefixSum[i] + upper] 范围内的个数
        int leftBound = lowerBound(prefixSum[i] + lower);
        int rightBound = upperBound(prefixSum[i] + upper);
        result += query(0, n - 1, 1, leftBound, rightBound);
        // 将当前前缀和插入线段树
        update(0, n - 1, 1, lowerBound(prefixSum[i]));
    }

    return result;
}

/**
 * 找到目标值在排序数组中的下界（第一个大于等于目标值的位置）
 * @param target 目标值
 * @return 下界位置
 */
private int lowerBound(long target) {
    int left = 0, right = n;
    while (left < right) {
        int mid = left + (right - left) / 2;
        if (sorted[mid] < target) {
            left = mid + 1;
        } else {
            right = mid;
        }
    }
    return left;
}

/**
 * 找到目标值在排序数组中的上界（第一个大于目标值的位置）
 * @param target 目标值
 * @return 上界位置
 */
private int upperBound(long target) {
    int left = 0, right = n;
    while (left < right) {
        int mid = left + (right - left) / 2;
        if (sorted[mid] <= target) {
```

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

/**
 * 更新线段树中的值（单点更新）
 * @param start 当前节点区间起始位置
 * @param end 当前节点区间结束位置
 * @param node 当前节点在 tree 数组中的索引
 * @param index 要更新的位置
 */
private void update(int start, int end, int node, int index) {
    // 如果 index 不在当前区间范围内，直接返回
    if (start > index || end < index) {
        return;
    }

    // 如果是叶子节点，直接增加计数
    if (start == end) {
        tree[node]++;
        return;
    }

    // 递归更新子节点
    int mid = start + (end - start) / 2;
    if (index <= mid) {
        update(start, mid, node * 2, index);
    } else {
        update(mid + 1, end, node * 2 + 1, index);
    }

    // 合并左右子树信息
    tree[node] = tree[node * 2] + tree[node * 2 + 1];
}

/**
 * 查询线段树中指定区间的元素个数
 * @param start 当前节点区间起始位置
 * @param end 当前节点区间结束位置
```

```
     * @param node 当前节点在 tree 数组中的索引
     * @param left 查询区间左边界
     * @param right 查询区间右边界
     * @return 区间内元素个数
     */
    private int query(int start, int end, int node, int left, int right) {
        // 如果查询区间无效或当前区间与查询区间无重叠，返回 0
        if (left > right || start > right || end < left) {
            return 0;
        }

        // 如果当前区间完全包含在查询区间内，返回当前节点的值
        if (start >= left && end <= right) {
            return tree[node];
        }

        // 递归查询左右子树
        int mid = start + (end - start) / 2;
        int leftCount = query(start, mid, node * 2, left, right);
        int rightCount = query(mid + 1, end, node * 2 + 1, left, right);

        return leftCount + rightCount;
    }

    // 测试方法
    public static void main(String[] args) {
        // 测试用例
        LeetCode327_SegmentTree solution = new LeetCode327_SegmentTree();

        int[] nums = {-2, 5, -1};
        int lower = -2, upper = 2;
        System.out.println("输入数组: [-2, 5, -1], lower = -2, upper = 2");
        System.out.println("输出结果: " + solution.countRangeSum(nums, lower, upper)); // 应该输
出 3
    }
}

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

文件: leetcode327_segment_tree.py
==========================================
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
```

"""
LeetCode 327. 区间和的个数

题目描述：
给你一个整数数组 nums 以及两个整数 lower 和 upper 。
求数组中，值位于范围 [lower, upper] （包含 lower 和 upper）之内的区间和的个数。
区间和 S(i, j) 表示在 nums 中，位置从 i 到 j 的元素之和，包含 i 和 j (i ≤ j)。

示例：
输入：nums = [-2,5,-1], lower = -2, upper = 2
输出：3
解释：存在三个区间：[0,0]、[2,2] 和 [0,2] ，对应的区间和分别是：-2 、-1 、2 。

解题思路：
使用权值线段树来解决这个问题。
1. 计算前缀和数组，将区间和问题转化为前缀和差值问题
2. 离散化：由于前缀和可能很大，需要先进行离散化处理
3. 权值线段树：线段树的每个节点存储某个值域范围内前缀和出现的次数
4. 从左向右遍历前缀和数组，在权值线段树中查询满足条件的前缀和个数，然后将当前前缀和插入线段树

时间复杂度分析：
- 计算前缀和：O(n)
- 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：
- O(n)，线段树需要 4*n 的空间来存储节点信息

链接：https://leetcode.cn/problems/count-of-range-sum
"""

```
class LeetCode327_SegmentTree:
    """
    LeetCode 327. 区间和的个数的线段树实现
    """

    def __init__(self):
        """构造函数"""
        self.tree = []    # 线段树数组，存储区间元素个数
        self.sorted = []  # 离散化后的数组
```

```
        self.n = 0        # 离散化后的数组大小

    def countRangeSum(self, nums, lower, upper):
        """
        计算区间和的个数
        :param nums: 输入数组
        :param lower: 下界
        :param upper: 上界
        :return: 区间和的个数
        """
        if not nums:
            return 0

        # 计算前缀和数组
        prefix_sum = [0]
        for num in nums:
            prefix_sum.append(prefix_sum[-1] + num)

        # 离散化处理
        num_set = set()
        for s in prefix_sum:
            num_set.add(s)
            num_set.add(s - lower)
            num_set.add(s - upper)

        self.sorted = sorted(list(num_set))
        self.n = len(self.sorted)

        # 线段树数组大小通常为 4*n，确保足够容纳所有节点
        self.tree = [0] * (self.n << 2)

        result = 0
        # 从左向右遍历前缀和数组
        for s in prefix_sum:
            # 查询满足条件 prefix_sum[j] - prefix_sum[i] 在 [lower, upper] 范围内的 j 个数
            # 即查询 prefix_sum[j] 在 [prefix_sum[i] + lower, prefix_sum[i] + upper] 范围内的个数
            left_bound = self._lower_bound(s + lower)
            right_bound = self._upper_bound(s + upper)
            result += self._query(0, self.n - 1, 1, left_bound, right_bound)
            # 将当前前缀和插入线段树
            self._update(0, self.n - 1, 1, self._lower_bound(s))

        return result
```

```
def _lower_bound(self, target):
    """
    找到目标值在排序数组中的下界（第一个大于等于目标值的位置）
    :param target: 目标值
    :return: 下界位置
    """
    left, right = 0, self.n
    while left < right:
        mid = left + (right - left) // 2
        if self.sorted[mid] < target:
            left = mid + 1
        else:
            right = mid
    return left

def _upper_bound(self, target):
    """
    找到目标值在排序数组中的上界（第一个大于目标值的位置）
    :param target: 目标值
    :return: 上界位置
    """
    left, right = 0, self.n
    while left < right:
        mid = left + (right - left) // 2
        if self.sorted[mid] <= target:
            left = mid + 1
        else:
            right = mid
    return left - 1

def _update(self, start, end, node, index):
    """
    更新线段树中的值（单点更新）
    :param start: 当前节点区间起始位置
    :param end: 当前节点区间结束位置
    :param node: 当前节点在 tree 数组中的索引
    :param index: 要更新的位置
    """
    # 如果 index 不在当前区间范围内，直接返回
    if start > index or end < index:
        return
```

```
        # 如果是叶子节点，直接增加计数
        if start == end:
            self.tree[node] += 1
            return

        # 递归更新子节点
        mid = start + (end - start) // 2
        if index <= mid:
            self._update(start, mid, node * 2, index)
        else:
            self._update(mid + 1, end, node * 2 + 1, index)

        # 合并左右子树信息
        self.tree[node] = self.tree[node * 2] + self.tree[node * 2 + 1]

    def _query(self, start, end, node, left, right):
        """
        查询线段树中指定区间的元素个数
        :param start: 当前节点区间起始位置
        :param end: 当前节点区间结束位置
        :param node: 当前节点在 tree 数组中的索引
        :param left: 查询区间左边界
        :param right: 查询区间右边界
        :return: 区间内元素个数
        """
        # 如果查询区间无效或当前区间与查询区间无重叠，返回 0
        if left > right or start > right or end < left:
            return 0

        # 如果当前区间完全包含在查询区间内，返回当前节点的值
        if start >= left and end <= right:
            return self.tree[node]

        # 递归查询左右子树
        mid = start + (end - start) // 2
        left_count = self._query(start, mid, node * 2, left, right)
        right_count = self._query(mid + 1, end, node * 2 + 1, left, right)

        return left_count + right_count


# 测试函数
def test_solution():
```

```
    """测试 LeetCode 327 实现"""
    print("测试 LeetCode 327 实现...")

    # 测试用例
    solution = LeetCode327_SegmentTree()

    nums = [-2, 5, -1]
    lower, upper = -2, 2
    print(f"输入数组: {nums}, lower = {lower}, upper = {upper}")
    print(f"输出结果: {solution.countRangeSum(nums, lower, upper)}")  # 应该输出 3

    print("LeetCode 327 测试完成！")


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

==========

文件: LeetCode493_SegmentTree.java

==========

```
package class109;

import java.util.*;

/**
 * LeetCode 493. 翻转对
 *
 * 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。
 *
 * 示例：
 * 输入: [1,3,2,3,1]
 * 输出: 2
 *
 * 输入: [2,4,3,5,1]
 * 输出: 3
 *
 * 解题思路：
 * 使用权值线段树来解决这个问题。
 * 1. 离散化：由于数组元素可能很大，需要先进行离散化处理，将元素映射到连续的小范围
 * 2. 权值线段树：线段树的每个节点存储某个值域范围内元素出现的次数
```

```
 * 3. 从左向右遍历数组，在权值线段树中查询满足条件的元素个数，然后将当前元素插入线段树
 *
 * 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：
 * - O(n)，线段树需要 4*n 的空间来存储节点信息
 *
 * 链接：https://leetcode.cn/problems/reverse-pairs
 */
public class LeetCode493_SegmentTree {

    // 线段树数组，存储区间元素个数
    private int[] tree;
    // 离散化后的数组
    private long[] sorted;
    // 离散化后的数组大小
    private int n;

    /**
     * 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
     */
    public int reversePairs(int[] nums) {
        if (nums == null || nums.length < 2) {
            return 0;
        }

        // 离散化处理
        TreeSet<Long> set = new TreeSet<>();
        for (int num : nums) {
            set.add((long) num);
            set.add((long) 2 * num);  // 同时加入 2*num，用于后续查询
        }
        sorted = new long[set.size()];
        int index = 0;
        for (long num : set) {
            sorted[index++] = num;
        }
        n = sorted.length;
```

```
    // 线段树数组大小通常为 4*n，确保足够容纳所有节点
    tree = new int[n << 2];

    int result = 0;
    // 从左向右遍历数组
    for (int i = 0; i < nums.length; i++) {
        // 查询满足条件 element > 2*nums[i] 的元素个数
        // 即查询在已经处理的元素中，有多少个元素大于 2*nums[i]
        int count = query(0, n - 1, 1, lowerBound((long) 2 * nums[i] + 1), n - 1);
        result += count;
        // 将当前元素插入线段树
        update(0, n - 1, 1, lowerBound((long) nums[i]));
    }

    return result;
}

/**
 * 找到目标值在排序数组中的下界（第一个大于等于目标值的位置）
 * @param target 目标值
 * @return 下界位置
 */
private int lowerBound(long target) {
    int left = 0, right = n;
    while (left < right) {
        int mid = left + (right - left) / 2;
        if (sorted[mid] < target) {
            left = mid + 1;
        } else {
            right = mid;
        }
    }
    return left;
}

/**
 * 更新线段树中的值（单点更新）
 * @param start 当前节点区间起始位置
 * @param end 当前节点区间结束位置
 * @param node 当前节点在 tree 数组中的索引
 * @param index 要更新的位置
 */
```

```
private void update(int start, int end, int node, int index) {
    // 如果 index 不在当前区间范围内，直接返回
    if (start > index || end < index) {
        return;
    }

    // 如果是叶子节点，直接增加计数
    if (start == end) {
        tree[node]++;
        return;
    }

    // 递归更新子节点
    int mid = start + (end - start) / 2;
    if (index <= mid) {
        update(start, mid, node * 2, index);
    } else {
        update(mid + 1, end, node * 2 + 1, index);
    }

    // 合并左右子树信息
    tree[node] = tree[node * 2] + tree[node * 2 + 1];
}

/**
 * 查询线段树中指定区间的元素个数
 * @param start 当前节点区间起始位置
 * @param end 当前节点区间结束位置
 * @param node 当前节点在 tree 数组中的索引
 * @param left 查询区间左边界
 * @param right 查询区间右边界
 * @return 区间内元素个数
 */
private int query(int start, int end, int node, int left, int right) {
    // 如果查询区间无效或当前区间与查询区间无重叠，返回 0
    if (left > right || start > right || end < left) {
        return 0;
    }

    // 如果当前区间完全包含在查询区间内，返回当前节点的值
    if (start >= left && end <= right) {
        return tree[node];
    }
```

```
        // 递归查询左右子树
        int mid = start + (end - start) / 2;
        int leftCount = query(start, mid, node * 2, left, right);
        int rightCount = query(mid + 1, end, node * 2 + 1, left, right);

        return leftCount + rightCount;
    }

    // 测试方法
    public static void main(String[] args) {
        // 测试用例
        LeetCode493_SegmentTree solution = new LeetCode493_SegmentTree();

        int[] nums1 = {1, 3, 2, 3, 1};
        System.out.println("输入数组: [1, 3, 2, 3, 1]");
        System.out.println("输出结果: " + solution.reversePairs(nums1)); // 应该输出 2

        int[] nums2 = {2, 4, 3, 5, 1};
        System.out.println("输入数组: [2, 4, 3, 5, 1]");
        System.out.println("输出结果: " + solution.reversePairs(nums2)); // 应该输出 3
    }
}
```

==========

文件: leetcode493_segment_tree.py

==========

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

"""
LeetCode 493. 翻转对

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。

示例：
输入: [1,3,2,3,1]
输出: 2

输入: [2,4,3,5,1]
```

输出: 3

解题思路:
使用权值线段树来解决这个问题。
1. 离散化:由于数组元素可能很大,需要先进行离散化处理,将元素映射到连续的小范围
2. 权值线段树:线段树的每个节点存储某个值域范围内元素出现的次数
3. 从左向右遍历数组,在权值线段树中查询满足条件的元素个数,然后将当前元素插入线段树

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:
- O(n),线段树需要 4*n 的空间来存储节点信息

链接:https://leetcode.cn/problems/reverse-pairs
"""

```
class LeetCode493_SegmentTree:
    """
    LeetCode 493. 翻转对的线段树实现
    """

    def __init__(self):
        """构造函数"""
        self.tree = []    # 线段树数组,存储区间元素个数
        self.sorted = []  # 离散化后的数组
        self.n = 0        # 离散化后的数组大小

    def reversePairs(self, nums):
        """
        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
        """
        if not nums or len(nums) < 2:
            return 0

        # 离散化处理
        num_set = set()
        for num in nums:
```

```
            num_set.add(num)
            num_set.add(2 * num)  # 同时加入 2*num，用于后续查询

        self.sorted = sorted(list(num_set))
        self.n = len(self.sorted)

        # 线段树数组大小通常为 4*n，确保足够容纳所有节点
        self.tree = [0] * (self.n << 2)

        result = 0
        # 从左向右遍历数组
        for i in range(len(nums)):
            # 查询满足条件 element > 2*nums[i] 的元素个数
            # 即查询在已经处理的元素中，有多少个元素大于 2*nums[i]
            count = self._query(0, self.n - 1, 1, self._lower_bound(2 * nums[i] + 1), self.n - 1)
            result += count
            # 将当前元素插入线段树
            self._update(0, self.n - 1, 1, self._lower_bound(nums[i]))

        return result

    def _lower_bound(self, target):
        """
        找到目标值在排序数组中的下界（第一个大于等于目标值的位置）
        :param target: 目标值
        :return: 下界位置
        """
        left, right = 0, self.n
        while left < right:
            mid = left + (right - left) // 2
            if self.sorted[mid] < target:
                left = mid + 1
            else:
                right = mid
        return left

    def _update(self, start, end, node, index):
        """
        更新线段树中的值（单点更新）
        :param start: 当前节点区间起始位置
        :param end: 当前节点区间结束位置
        :param node: 当前节点在 tree 数组中的索引
        :param index: 要更新的位置
```

```
    """
    # 如果 index 不在当前区间范围内，直接返回
    if start > index or end < index:
        return

    # 如果是叶子节点，直接增加计数
    if start == end:
        self.tree[node] += 1
        return

    # 递归更新子节点
    mid = start + (end - start) // 2
    if index <= mid:
        self._update(start, mid, node * 2, index)
    else:
        self._update(mid + 1, end, node * 2 + 1, index)

    # 合并左右子树信息
    self.tree[node] = self.tree[node * 2] + self.tree[node * 2 + 1]

def _query(self, start, end, node, left, right):
    """
    查询线段树中指定区间的元素个数
    :param start: 当前节点区间起始位置
    :param end: 当前节点区间结束位置
    :param node: 当前节点在 tree 数组中的索引
    :param left: 查询区间左边界
    :param right: 查询区间右边界
    :return: 区间内元素个数
    """
    # 如果查询区间无效或当前区间与查询区间无重叠，返回 0
    if left > right or start > right or end < left:
        return 0

    # 如果当前区间完全包含在查询区间内，返回当前节点的值
    if start >= left and end <= right:
        return self.tree[node]

    # 递归查询左右子树
    mid = start + (end - start) // 2
    left_count = self._query(start, mid, node * 2, left, right)
    right_count = self._query(mid + 1, end, node * 2 + 1, left, right)
```

```
        return left_count + right_count


# 测试函数
def test_solution():
    """测试 LeetCode 493 实现"""
    print("测试 LeetCode 493 实现...")

    # 测试用例
    solution = LeetCode493_SegmentTree()

    nums1 = [1, 3, 2, 3, 1]
    print(f"输入数组: {nums1}")
    print(f"输出结果: {solution.reversePairs(nums1)}")  # 应该输出 2

    nums2 = [2, 4, 3, 5, 1]
    print(f"输入数组: {nums2}")
    print(f"输出结果: {solution.reversePairs(nums2)}")  # 应该输出 3

    print("LeetCode 493 测试完成！")


if __name__ == "__main__":
    test_solution()

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

文件: LuoguP3372_SegmentTree.java
==========================================
package class109;

/**
 * 洛谷 P3372 【模板】线段树 1
 *
 * 题目描述：
 * 如题，已知一个数列，你需要进行下面两种操作：
 * 1. 将某区间每一个数加上 x
 * 2. 求出某区间每一个数的和
 *
 * 解题思路：
 * 使用线段树配合懒标记(Lazy Propagation)来解决区间更新问题。
 * 1. 线段树节点维护区间和
 * 2. 懒标记用于延迟区间更新操作，避免每次都更新到叶子节点
```

```
 * 3. 在需要访问子节点时，将懒标记下推(push down)
 *
 * 时间复杂度分析：
 * - 初始化：O(n)
 * - 区间更新：O(log n)
 * - 区间查询：O(log n)
 *
 * 空间复杂度分析：
 * - O(n)，线段树需要 4*n 的空间来存储节点信息
 *
 * 链接：https://www.luogu.com.cn/problem/P3372
 */
public class LuoguP3372_SegmentTree {

    // 线段树数组，存储区间和
    private long[] tree;
    // 懒标记数组，存储区间延迟更新的值
    private long[] lazy;
    // 原数组
    private int[] nums;
    // 数组长度
    private int n;

    /**
     * 构造函数，初始化线段树
     * @param nums 输入数组
     */
    public LuoguP3372_SegmentTree(int[] nums) {
        this.n = nums.length;
        this.nums = nums;
        // 线段树数组大小通常为 4*n，确保足够容纳所有节点
        this.tree = new long[n << 2];
        this.lazy = new long[n << 2];
        // 构建线段树
        buildTree(0, n - 1, 1);
    }

    /**
     * 构建线段树
     * @param start 区间起始位置
     * @param end 区间结束位置
     * @param node 当前节点在 tree 数组中的索引
     */
```

```
private void buildTree(int start, int end, int node) {
    // 清空懒标记
    lazy[node] = 0;

    // 如果是叶子节点，直接赋值
    if (start == end) {
        tree[node] = nums[start];
        return;
    }

    // 计算中点
    int mid = (start + end) / 2;
    // 递归构建左右子树
    buildTree(start, mid, node * 2);
    buildTree(mid + 1, end, node * 2 + 1);
    // 合并左右子树信息
    tree[node] = tree[node * 2] + tree[node * 2 + 1];
}

/**
 * 下推懒标记
 * @param node 当前节点
 * @param start 当前节点区间起始位置
 * @param end 当前节点区间结束位置
 */
private void pushDown(int node, int start, int end) {
    if (lazy[node] != 0) {
        // 将懒标记下推到左右子节点
        lazy[node * 2] += lazy[node];
        lazy[node * 2 + 1] += lazy[node];

        // 如果不是叶子节点，更新子节点的值
        if (start != end) {
            int mid = (start + end) / 2;
            tree[node * 2] += lazy[node] * (mid - start + 1);
            tree[node * 2 + 1] += lazy[node] * (end - mid);
        }

        // 清空当前节点的懒标记
        lazy[node] = 0;
    }
}
```

```
/**
 * 区间更新
 * @param start 当前节点区间起始位置
 * @param end 当前节点区间结束位置
 * @param node 当前节点在 tree 数组中的索引
 * @param left 更新区间左边界
 * @param right 更新区间右边界
 * @param val 更新值
 */
private void updateRange(int start, int end, int node, int left, int right, long val) {
    // 如果当前区间与更新区间无重叠，直接返回
    if (start > right || end < left) {
        return;
    }

    // 如果当前区间完全包含在更新区间内
    if (start >= left && end <= right) {
        // 更新当前节点的值
        tree[node] += val * (end - start + 1);
        // 设置懒标记
        if (start != end) {
            lazy[node] += val;
        }
        return;
    }

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

    // 递归更新左右子树
    int mid = (start + end) / 2;
    updateRange(start, mid, node * 2, left, right, val);
    updateRange(mid + 1, end, node * 2 + 1, left, right, val);

    // 合并左右子树信息
    tree[node] = tree[node * 2] + tree[node * 2 + 1];
}

/**
 * 区间更新接口
 * @param left 更新区间左边界
 * @param right 更新区间右边界
 * @param val 更新值
```

```
     */
    public void update(int left, int right, long val) {
        updateRange(0, n - 1, 1, left, right, val);
    }

    /**
     * 区间查询
     * @param start 当前节点区间起始位置
     * @param end 当前节点区间结束位置
     * @param node 当前节点在 tree 数组中的索引
     * @param left 查询区间左边界
     * @param right 查询区间右边界
     * @return 区间和
     */
    private long queryRange(int start, int end, int node, int left, int right) {
        // 如果当前区间与查询区间无重叠，返回 0
        if (start > right || end < left) {
            return 0;
        }

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

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

        // 递归查询左右子树
        int mid = (start + end) / 2;
        long leftSum = queryRange(start, mid, node * 2, left, right);
        long rightSum = queryRange(mid + 1, end, node * 2 + 1, left, right);

        return leftSum + rightSum;
    }

    /**
     * 区间查询接口
     * @param left 查询区间左边界
     * @param right 查询区间右边界
     * @return 区间和
     */
    public long query(int left, int right) {
```

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

    // 测试方法
    public static void main(String[] args) {
        System.out.println("测试洛谷 P3372 实现...");

        // 测试用例
        int[] nums = {1, 2, 3, 4, 5};
        LuoguP3372_SegmentTree segTree = new LuoguP3372_SegmentTree(nums);

        System.out.println("初始数组: [1, 2, 3, 4, 5]");
        System.out.println("查询区间[1,3]的和: " + segTree.query(1, 3)); // 应该输出 9

        // 区间更新：将区间[1,3]中的每个元素都加上 2
        segTree.update(1, 3, 2);
        System.out.println("将区间[1,3]中的每个元素都加上 2 后:");
        System.out.println("查询区间[1,3]的和: " + segTree.query(1, 3)); // 应该输出 15

        System.out.println("洛谷 P3372 测试完成！");
    }
}
```

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

文件: luogu_p3372_segment_tree.py

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

"""
洛谷 P3372 【模板】线段树 1

题目描述：
如题，已知一个数列，你需要进行下面两种操作：
1. 将某区间每一个数加上 x
2. 求出某区间每一个数的和

解题思路：
使用线段树配合懒标记(Lazy Propagation)来解决区间更新问题。
1. 线段树节点维护区间和
2. 懒标记用于延迟区间更新操作，避免每次都更新到叶子节点
3. 在需要访问子节点时，将懒标记下推(push down)
```

时间复杂度分析：
- 初始化：O(n)
- 区间更新：O(log n)
- 区间查询：O(log n)

空间复杂度分析：
- O(n)，线段树需要 4*n 的空间来存储节点信息

链接：https://www.luogu.com.cn/problem/P3372
"""

```
class LuoguP3372_SegmentTree:
    """
    洛谷 P3372 【模板】线段树 1 的线段树实现
    """

    def __init__(self, nums):
        """
        构造函数，初始化线段树
        :param nums: 输入数组
        """
        self.n = len(nums)
        self.nums = nums[:]
        # 线段树数组大小通常为 4*n，确保足够容纳所有节点
        self.tree = [0] * (self.n << 2)
        self.lazy = [0] * (self.n << 2)
        # 构建线段树
        self._build_tree(0, self.n - 1, 1)

    def _build_tree(self, start, end, node):
        """
        构建线段树
        :param start: 区间起始位置
        :param end: 区间结束位置
        :param node: 当前节点在 tree 数组中的索引
        """
        # 清空懒标记
        self.lazy[node] = 0

        # 如果是叶子节点，直接赋值
        if start == end:
```

```
            self.tree[node] = self.nums[start]
            return

        # 计算中点
        mid = (start + end) // 2
        # 递归构建左右子树
        self._build_tree(start, mid, node * 2)
        self._build_tree(mid + 1, end, node * 2 + 1)
        # 合并左右子树信息
        self.tree[node] = self.tree[node * 2] + self.tree[node * 2 + 1]

    def _push_down(self, node, start, end):
        """
        下推懒标记
        :param node: 当前节点
        :param start: 当前节点区间起始位置
        :param end: 当前节点区间结束位置
        """
        if self.lazy[node] != 0:
            # 将懒标记下推到左右子节点
            self.lazy[node * 2] += self.lazy[node]
            self.lazy[node * 2 + 1] += self.lazy[node]

            # 如果不是叶子节点，更新子节点的值
            if start != end:
                mid = (start + end) // 2
                self.tree[node * 2] += self.lazy[node] * (mid - start + 1)
                self.tree[node * 2 + 1] += self.lazy[node] * (end - mid)

            # 清空当前节点的懒标记
            self.lazy[node] = 0

    def _update_range(self, start, end, node, left, right, val):
        """
        区间更新
        :param start: 当前节点区间起始位置
        :param end: 当前节点区间结束位置
        :param node: 当前节点在 tree 数组中的索引
        :param left: 更新区间左边界
        :param right: 更新区间右边界
        :param val: 更新值
        """
        # 如果当前区间与更新区间无重叠，直接返回
```

```
        if start > right or end < left:
            return

        # 如果当前区间完全包含在更新区间内
        if start >= left and end <= right:
            # 更新当前节点的值
            self.tree[node] += val * (end - start + 1)
            # 设置懒标记
            if start != end:
                self.lazy[node] += val
            return

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

        # 递归更新左右子树
        mid = (start + end) // 2
        self._update_range(start, mid, node * 2, left, right, val)
        self._update_range(mid + 1, end, node * 2 + 1, left, right, val)

        # 合并左右子树信息
        self.tree[node] = self.tree[node * 2] + self.tree[node * 2 + 1]

    def update(self, left, right, val):
        """
        区间更新接口
        :param left: 更新区间左边界
        :param right: 更新区间右边界
        :param val: 更新值
        """
        self._update_range(0, self.n - 1, 1, left, right, val)

    def _query_range(self, start, end, node, left, right):
        """
        区间查询
        :param start: 当前节点区间起始位置
        :param end: 当前节点区间结束位置
        :param node: 当前节点在 tree 数组中的索引
        :param left: 查询区间左边界
        :param right: 查询区间右边界
        :return: 区间和
        """
        # 如果当前区间与查询区间无重叠，返回 0
```

```
        if start > right or end < left:
            return 0

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

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

        # 递归查询左右子树
        mid = (start + end) // 2
        left_sum = self._query_range(start, mid, node * 2, left, right)
        right_sum = self._query_range(mid + 1, end, node * 2 + 1, left, right)

        return left_sum + right_sum

    def query(self, left, right):
        """
        区间查询接口
        :param left: 查询区间左边界
        :param right: 查询区间右边界
        :return: 区间和
        """
        return self._query_range(0, self.n - 1, 1, left, right)


# 测试函数
def test_solution():
    """测试洛谷 P3372 实现"""
    print("测试洛谷 P3372 实现...")

    # 测试用例
    nums = [1, 2, 3, 4, 5]
    seg_tree = LuoguP3372_SegmentTree(nums)

    print(f"初始数组: {nums}")
    print(f"查询区间[1,3]的和: {seg_tree.query(1, 3)}")  # 应该输出 9

    # 区间更新：将区间[1,3]中的每个元素都加上 2
    seg_tree.update(1, 3, 2)
    print("将区间[1,3]中的每个元素都加上 2 后:")
    print(f"查询区间[1,3]的和: {seg_tree.query(1, 3)}")  # 应该输出 15
```

```
    print("洛谷 P3372 测试完成！")


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

==========

文件: Main.java

==========

```
import java.util.Arrays;

/**
 * 翻转对
 * 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。
 *
 * 示例 1:
 * 输入: [1,3,2,3,1]
 * 输出: 2
 *
 * 示例 2:
 * 输入: [2,4,3,5,1]
 * 输出: 3
 *
 * 解题思路：
 * 1. 使用归并排序的思想解决
 * 2. 在归并的过程中统计翻转对的数量
 * 3. 对于左半部分[l...m]和右半部分[m+1...r]：
 *    - 在合并前，先统计翻转对数量：对于左半部分的每个元素 nums[i]，统计右半部分有多少个元素 nums[j]满足 nums[i] > 2*nums[j]
 *    - 然后进行正常的归并排序合并过程
 *
 * 时间复杂度分析：
 * - 归并排序的时间复杂度为 O(n log n)
 * - 每一层递归都会遍历所有元素，共 log n 层
 * - 所以总时间复杂度为 O(n log n)
 *
 * 空间复杂度分析：
 * - 需要额外的 help 数组存储临时数据，空间复杂度为 O(n)
 * - 递归调用栈的深度为 O(log n)
 * - 所以总空间复杂度为 O(n)
```

```
 *
 * 测试链接: https://leetcode.cn/problems/reverse-pairs/
 */
public class Main {

    /**
     * 计算翻转对数量
     *
     * @param nums 输入数组
     * @return 翻转对数量
     */
    public static int reversePairs(int[] nums) {
        if (nums == null || nums.length < 2) {
            return 0;
        }
        return process(nums, 0, nums.length - 1);
    }

    /**
     * 递归处理区间[l,r]内的翻转对
     *
     * @param nums 数组
     * @param l 区间左边界
     * @param r 区间右边界
     * @return 区间内翻转对数量
     */
    public static int process(int[] nums, int l, int r) {
        if (l == r) {
            return 0;
        }
        int m = l + ((r - l) >> 1);
        return process(nums, l, m) + process(nums, m + 1, r) + merge(nums, l, m, r);
    }

    /**
     * 合并两个有序数组，并统计合并过程中产生的翻转对数量
     *
     * @param nums 数组
     * @param l 左半部分起始位置
     * @param m 左半部分结束位置
     * @param r 右半部分结束位置
     * @return 合并过程中产生的翻转对数量
     */
```

```
public static int merge(int[] nums, int l, int m, int r) {
    // 统计翻转对数量
    int ans = 0;
    // 对于左半部分的每个元素 nums[i]，统计右半部分有多少个元素 nums[j]满足 nums[i] > 2*nums[j]
    for (int i = l; i <= m; i++) {
        // 在右半部分找到第一个不满足 nums[i] <= 2*nums[j]的位置
        int j = m + 1;
        while (j <= r && (long) nums[i] <= 2 * (long) nums[j]) {
            j++;
        }
        // 从 j 到 r 的所有元素都与 nums[i]构成翻转对
        ans += r - j + 1;
    }

    // 正常的归并排序合并过程
    int[] help = new int[r - l + 1];
    int i = 0;
    int a = l;
    int b = m + 1;
    while (a <= m && b <= r) {
        help[i++] = nums[a] <= nums[b] ? nums[a++] : nums[b++];
    }
    while (a <= m) {
        help[i++] = nums[a++];
    }
    while (b <= r) {
        help[i++] = nums[b++];
    }
    for (i = 0; i < help.length; i++) {
        nums[l + i] = help[i];
    }
    return ans;
}

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

    // 测试用例 2
```

```
        int[] nums2 = {2, 4, 3, 5, 1};
        System.out.println("输入: [2,4,3,5,1]");
        System.out.println("输出: " + reversePairs(nums2));
        System.out.println("期望: 3");
    }
}
```

==========

文件: POJ3468_SegmentTree.cpp

==========

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

/**
 * POJ 3468 A Simple Problem with Integers
 *
 * 题目描述：
 * 给定一个长度为N的整数数组A，初始时A[i] = 0 (1 <= i <= N)。
 * 需要处理以下两种操作：
 * 1. C a b c: 将区间[a, b]中的每个元素都加上c
 * 2. Q a b: 查询区间[a, b]中所有元素的和
 *
 * 解题思路：
 * 使用线段树配合懒标记(Lazy Propagation)来解决区间更新问题。
 * 1. 线段树节点维护区间和
 * 2. 懒标记用于延迟区间更新操作，避免每次都更新到叶子节点
 * 3. 在需要访问子节点时，将懒标记下推(push down)
 *
 * 时间复杂度分析：
 * - 初始化：O(n)
 * - 区间更新：O(log n)
 * - 区间查询：O(log n)
 *
 * 空间复杂度分析：
 * - O(n)，线段树需要4*n的空间来存储节点信息
 *
 * 链接：http://poj.org/problem?id=3468
 */
class POJ3468_SegmentTree {
private:
    vector<long long> tree;  // 线段树数组，存储区间和
```

```
vector<long long> lazy;  // 懒标记数组，存储区间延迟更新的值
vector<int> nums;        // 原数组
int n;                   // 数组长度

/**
 * 构建线段树
 * @param start 区间起始位置
 * @param end 区间结束位置
 * @param node 当前节点在 tree 数组中的索引
 */
void buildTree(int start, int end, int node) {
    // 清空懒标记
    lazy[node] = 0;

    // 如果是叶子节点，直接赋值
    if (start == end) {
        tree[node] = nums[start];
        return;
    }

    // 计算中点
    int mid = (start + end) / 2;
    // 递归构建左右子树
    buildTree(start, mid, node * 2);
    buildTree(mid + 1, end, node * 2 + 1);
    // 合并左右子树信息
    tree[node] = tree[node * 2] + tree[node * 2 + 1];
}

/**
 * 下推懒标记
 * @param node 当前节点
 * @param start 当前节点区间起始位置
 * @param end 当前节点区间结束位置
 */
void pushDown(int node, int start, int end) {
    if (lazy[node] != 0) {
        // 将懒标记下推到左右子节点
        lazy[node * 2] += lazy[node];
        lazy[node * 2 + 1] += lazy[node];

        // 如果不是叶子节点，更新子节点的值
        if (start != end) {
```

```
            int mid = (start + end) / 2;
            tree[node * 2] += lazy[node] * (mid - start + 1);
            tree[node * 2 + 1] += lazy[node] * (end - mid);
        }

        // 清空当前节点的懒标记
        lazy[node] = 0;
    }
}

/**
 * 区间更新
 * @param start 当前节点区间起始位置
 * @param end 当前节点区间结束位置
 * @param node 当前节点在 tree 数组中的索引
 * @param left 更新区间左边界
 * @param right 更新区间右边界
 * @param val 更新值
 */
void updateRange(int start, int end, int node, int left, int right, long long val) {
    // 如果当前区间与更新区间无重叠，直接返回
    if (start > right || end < left) {
        return;
    }

    // 如果当前区间完全包含在更新区间内
    if (start >= left && end <= right) {
        // 更新当前节点的值
        tree[node] += val * (end - start + 1);
        // 设置懒标记
        if (start != end) {
            lazy[node] += val;
        }
        return;
    }

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

    // 递归更新左右子树
    int mid = (start + end) / 2;
    updateRange(start, mid, node * 2, left, right, val);
    updateRange(mid + 1, end, node * 2 + 1, left, right, val);
```

```
        // 合并左右子树信息
        tree[node] = tree[node * 2] + tree[node * 2 + 1];
    }

    /**
     * 区间查询
     * @param start 当前节点区间起始位置
     * @param end 当前节点区间结束位置
     * @param node 当前节点在 tree 数组中的索引
     * @param left 查询区间左边界
     * @param right 查询区间右边界
     * @return 区间和
     */
    long long queryRange(int start, int end, int node, int left, int right) {
        // 如果当前区间与查询区间无重叠，返回 0
        if (start > right || end < left) {
            return 0;
        }

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

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

        // 递归查询左右子树
        int mid = (start + end) / 2;
        long long leftSum = queryRange(start, mid, node * 2, left, right);
        long long rightSum = queryRange(mid + 1, end, node * 2 + 1, left, right);

        return leftSum + rightSum;
    }

public:
    /**
     * 构造函数，初始化线段树
     * @param nums 输入数组
     */
    POJ3468_SegmentTree(vector<int>& nums) {
        this->n = nums.size();
```

```
        this->nums = nums;
        // 线段树数组大小通常为4*n，确保足够容纳所有节点
        this->tree.resize(n << 2);
        this->lazy.resize(n << 2);
        // 构建线段树
        buildTree(0, n - 1, 1);
    }

    /**
     * 区间更新接口
     * @param left 更新区间左边界
     * @param right 更新区间右边界
     * @param val 更新值
     */
    void update(int left, int right, long long val) {
        updateRange(0, n - 1, 1, left, right, val);
    }

    /**
     * 区间查询接口
     * @param left 查询区间左边界
     * @param right 查询区间右边界
     * @return 区间和
     */
    long long query(int left, int right) {
        return queryRange(0, n - 1, 1, left, right);
    }
};

// 测试函数
int main() {
    cout << "测试POJ 3468实现..." << endl;

    // 测试用例
    vector<int> nums = {1, 2, 3, 4, 5};
    POJ3468_SegmentTree segTree(nums);

    cout << "初始数组: [1, 2, 3, 4, 5]" << endl;
    cout << "查询区间[1,3]的和: " << segTree.query(1, 3) << endl; // 应该输出9

    // 区间更新：将区间[1,3]中的每个元素都加上2
    segTree.update(1, 3, 2);
    cout << "将区间[1,3]中的每个元素都加上2后:" << endl;
```

```
    cout << "查询区间[1,3]的和: " << segTree.query(1, 3) << endl; // 应该输出 15

    cout << "POJ 3468 测试完成!" << endl;

    return 0;
}
```

==========

文件: POJ3468_SegmentTree.java

==========

```
package class109;

/**
 * POJ 3468 A Simple Problem with Integers
 *
 * 题目描述:
 * 给定一个长度为 N 的整数数组 A, 初始时 A[i] = 0 (1 <= i <= N)。
 * 需要处理以下两种操作:
 * 1. C a b c: 将区间[a, b]中的每个元素都加上 c
 * 2. Q a b: 查询区间[a, b]中所有元素的和
 *
 * 解题思路:
 * 使用线段树配合懒标记(Lazy Propagation)来解决区间更新问题。
 * 1. 线段树节点维护区间和
 * 2. 懒标记用于延迟区间更新操作, 避免每次都更新到叶子节点
 * 3. 在需要访问子节点时, 将懒标记下推(push down)
 *
 * 时间复杂度分析:
 * - 初始化: O(n)
 * - 区间更新: O(log n)
 * - 区间查询: O(log n)
 *
 * 空间复杂度分析:
 * - O(n), 线段树需要 4*n 的空间来存储节点信息
 *
 * 链接: http://poj.org/problem?id=3468
 */
public class POJ3468_SegmentTree {

    // 线段树数组, 存储区间和
    private long[] tree;
    // 懒标记数组, 存储区间延迟更新的值
```

```
private long[] lazy;
// 原数组
private int[] nums;
// 数组长度
private int n;

/**
 * 构造函数，初始化线段树
 * @param nums 输入数组
 */
public POJ3468_SegmentTree(int[] nums) {
    this.n = nums.length;
    this.nums = nums;
    // 线段树数组大小通常为 4*n，确保足够容纳所有节点
    this.tree = new long[n << 2];
    this.lazy = new long[n << 2];
    // 构建线段树
    buildTree(0, n - 1, 1);
}

/**
 * 构建线段树
 * @param start 区间起始位置
 * @param end 区间结束位置
 * @param node 当前节点在 tree 数组中的索引
 */
private void buildTree(int start, int end, int node) {
    // 清空懒标记
    lazy[node] = 0;

    // 如果是叶子节点，直接赋值
    if (start == end) {
        tree[node] = nums[start];
        return;
    }

    // 计算中点
    int mid = start + (end - start) / 2;
    // 递归构建左右子树
    buildTree(start, mid, node * 2);
    buildTree(mid + 1, end, node * 2 + 1);
    // 合并左右子树信息
    tree[node] = tree[node * 2] + tree[node * 2 + 1];
```

```
}

/**
 * 下推懒标记
 * @param node 当前节点
 * @param start 当前节点区间起始位置
 * @param end 当前节点区间结束位置
 */
private void pushDown(int node, int start, int end) {
    if (lazy[node] != 0) {
        // 将懒标记下推到左右子节点
        lazy[node * 2] += lazy[node];
        lazy[node * 2 + 1] += lazy[node];

        // 如果不是叶子节点，更新子节点的值
        if (start != end) {
            int mid = start + (end - start) / 2;
            tree[node * 2] += lazy[node] * (mid - start + 1);
            tree[node * 2 + 1] += lazy[node] * (end - mid);
        }

        // 清空当前节点的懒标记
        lazy[node] = 0;
    }
}

/**
 * 区间更新
 * @param start 当前节点区间起始位置
 * @param end 当前节点区间结束位置
 * @param node 当前节点在 tree 数组中的索引
 * @param left 更新区间左边界
 * @param right 更新区间右边界
 * @param val 更新值
 */
private void updateRange(int start, int end, int node, int left, int right, long val) {
    // 如果当前区间与更新区间无重叠，直接返回
    if (start > right || end < left) {
        return;
    }

    // 如果当前区间完全包含在更新区间内
    if (start >= left && end <= right) {
```

```
            // 更新当前节点的值
            tree[node] += val * (end - start + 1);
            // 设置懒标记
            if (start != end) {
                lazy[node] += val;
            }
            return;
        }

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

        // 递归更新左右子树
        int mid = start + (end - start) / 2;
        updateRange(start, mid, node * 2, left, right, val);
        updateRange(mid + 1, end, node * 2 + 1, left, right, val);

        // 合并左右子树信息
        tree[node] = tree[node * 2] + tree[node * 2 + 1];
    }

    /**
     * 区间更新接口
     * @param left 更新区间左边界
     * @param right 更新区间右边界
     * @param val 更新值
     */
    public void update(int left, int right, long val) {
        updateRange(0, n - 1, 1, left, right, val);
    }

    /**
     * 区间查询
     * @param start 当前节点区间起始位置
     * @param end 当前节点区间结束位置
     * @param node 当前节点在 tree 数组中的索引
     * @param left 查询区间左边界
     * @param right 查询区间右边界
     * @return 区间和
     */
    private long queryRange(int start, int end, int node, int left, int right) {
        // 如果当前区间与查询区间无重叠，返回 0
        if (start > right || end < left) {
```

```
        return 0;
    }

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

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

    // 递归查询左右子树
    int mid = start + (end - start) / 2;
    long leftSum = queryRange(start, mid, node * 2, left, right);
    long rightSum = queryRange(mid + 1, end, node * 2 + 1, left, right);

    return leftSum + rightSum;
}

/**
 * 区间查询接口
 * @param left 查询区间左边界
 * @param right 查询区间右边界
 * @return 区间和
 */
public long query(int left, int right) {
    return queryRange(0, n - 1, 1, left, right);
}

// 测试方法
public static void main(String[] args) {
    System.out.println("测试 POJ 3468 实现...");

    // 测试用例
    int[] nums = {1, 2, 3, 4, 5};
    POJ3468_SegmentTree segTree = new POJ3468_SegmentTree(nums);

    System.out.println("初始数组: [1, 2, 3, 4, 5]");
    System.out.println("查询区间[1,3]的和: " + segTree.query(1, 3)); // 应该输出 9

    // 区间更新：将区间[1,3]中的每个元素都加上 2
    segTree.update(1, 3, 2);
    System.out.println("将区间[1,3]中的每个元素都加上 2 后:");
```

```
        System.out.println("查询区间[1,3]的和: " + segTree.query(1, 3)); // 应该输出 15

        System.out.println("POJ 3468 测试完成！");
    }
}
```

==========

文件: poj3468_segment_tree.py

==========

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

"""
POJ 3468 A Simple Problem with Integers

题目描述：
给定一个长度为 N 的整数数组 A，初始时 A[i] = 0 (1 <= i <= N)。
需要处理以下两种操作：
1. C a b c: 将区间[a, b]中的每个元素都加上 c
2. Q a b: 查询区间[a, b]中所有元素的和

解题思路：
使用线段树配合懒标记(Lazy Propagation)来解决区间更新问题。
1. 线段树节点维护区间和
2. 懒标记用于延迟区间更新操作，避免每次都更新到叶子节点
3. 在需要访问子节点时，将懒标记下推(push down)

时间复杂度分析：
- 初始化：O(n)
- 区间更新：O(log n)
- 区间查询：O(log n)

空间复杂度分析：
- O(n)，线段树需要 4*n 的空间来存储节点信息

链接：http://poj.org/problem?id=3468
"""



class POJ3468_SegmentTree:
    """
    POJ 3468 A Simple Problem with Integers 的线段树实现
```

```
"""

def __init__(self, nums):
    """
    构造函数，初始化线段树
    :param nums: 输入数组
    """
    self.n = len(nums)
    self.nums = nums[:]
    # 线段树数组大小通常为 4*n，确保足够容纳所有节点
    self.tree = [0] * (self.n << 2)
    self.lazy = [0] * (self.n << 2)
    # 构建线段树
    self._build_tree(0, self.n - 1, 1)

def _build_tree(self, start, end, node):
    """
    构建线段树
    :param start: 区间起始位置
    :param end: 区间结束位置
    :param node: 当前节点在 tree 数组中的索引
    """
    # 清空懒标记
    self.lazy[node] = 0

    # 如果是叶子节点，直接赋值
    if start == end:
        self.tree[node] = self.nums[start]
        return

    # 计算中点
    mid = (start + end) // 2
    # 递归构建左右子树
    self._build_tree(start, mid, node * 2)
    self._build_tree(mid + 1, end, node * 2 + 1)
    # 合并左右子树信息
    self.tree[node] = self.tree[node * 2] + self.tree[node * 2 + 1]

def _push_down(self, node, start, end):
    """
    下推懒标记
    :param node: 当前节点
    :param start: 当前节点区间起始位置
```

```
    :param end: 当前节点区间结束位置
    """
    if self.lazy[node] != 0:
        # 将懒标记下推到左右子节点
        self.lazy[node * 2] += self.lazy[node]
        self.lazy[node * 2 + 1] += self.lazy[node]

        # 如果不是叶子节点，更新子节点的值
        if start != end:
            mid = (start + end) // 2
            self.tree[node * 2] += self.lazy[node] * (mid - start + 1)
            self.tree[node * 2 + 1] += self.lazy[node] * (end - mid)

        # 清空当前节点的懒标记
        self.lazy[node] = 0

def _update_range(self, start, end, node, left, right, val):
    """
    区间更新
    :param start: 当前节点区间起始位置
    :param end: 当前节点区间结束位置
    :param node: 当前节点在 tree 数组中的索引
    :param left: 更新区间左边界
    :param right: 更新区间右边界
    :param val: 更新值
    """
    # 如果当前区间与更新区间无重叠，直接返回
    if start > right or end < left:
        return

    # 如果当前区间完全包含在更新区间内
    if start >= left and end <= right:
        # 更新当前节点的值
        self.tree[node] += val * (end - start + 1)
        # 设置懒标记
        if start != end:
            self.lazy[node] += val
        return

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

    # 递归更新左右子树
```

```
        mid = (start + end) // 2
        self._update_range(start, mid, node * 2, left, right, val)
        self._update_range(mid + 1, end, node * 2 + 1, left, right, val)

        # 合并左右子树信息
        self.tree[node] = self.tree[node * 2] + self.tree[node * 2 + 1]

    def update(self, left, right, val):
        """
        区间更新接口
        :param left: 更新区间左边界
        :param right: 更新区间右边界
        :param val: 更新值
        """
        self._update_range(0, self.n - 1, 1, left, right, val)

    def _query_range(self, start, end, node, left, right):
        """
        区间查询
        :param start: 当前节点区间起始位置
        :param end: 当前节点区间结束位置
        :param node: 当前节点在 tree 数组中的索引
        :param left: 查询区间左边界
        :param right: 查询区间右边界
        :return: 区间和
        """
        # 如果当前区间与查询区间无重叠，返回 0
        if start > right or end < left:
            return 0

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

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

        # 递归查询左右子树
        mid = (start + end) // 2
        left_sum = self._query_range(start, mid, node * 2, left, right)
        right_sum = self._query_range(mid + 1, end, node * 2 + 1, left, right)

        return left_sum + right_sum
```

```
    def query(self, left, right):
        """
        区间查询接口
        :param left: 查询区间左边界
        :param right: 查询区间右边界
        :return: 区间和
        """
        return self._query_range(0, self.n - 1, 1, left, right)


# 测试函数
def test_solution():
    """测试 POJ 3468 实现"""
    print("测试 POJ 3468 实现...")

    # 测试用例
    nums = [1, 2, 3, 4, 5]
    seg_tree = POJ3468_SegmentTree(nums)

    print(f"初始数组: {nums}")
    print(f"查询区间[1,3]的和: {seg_tree.query(1, 3)}")  # 应该输出 9

    # 区间更新：将区间[1,3]中的每个元素都加上 2
    seg_tree.update(1, 3, 2)
    print("将区间[1,3]中的每个元素都加上 2 后:")
    print(f"查询区间[1,3]的和: {seg_tree.query(1, 3)}")  # 应该输出 15

    print("POJ 3468 测试完成！")


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

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

文件: SegmentTreeBasic.java

```
==============================================
package class109;

/**
 * 线段树基本实现
 * 支持区间求和、区间更新操作
```

```
 *
 * 线段树是一种二叉树结构，每个节点代表一个区间，用于高效处理区间查询和更新操作。
 *
 * 核心思想：
 * 1. 将数组区间划分为更小的子区间，直到单个元素
 * 2. 每个节点存储其对应区间的相关信息（如区间和）
 * 3. 通过合并子区间信息来维护父区间信息
 *
 * 时间复杂度：
 * - 建树：O(n)
 * - 单点更新：O(log n)
 * - 区间查询：O(log n)
 * - 区间更新：O(log n)
 *
 * 空间复杂度：O(n)
 */
public class SegmentTreeBasic {
    // 原数组
    private int[] arr;
    // 线段树数组，存储区间和
    private int[] sum;
    // 懒标记数组，用于区间更新
    private int[] lazy;
    // 数组长度
    private int n;

    /**
     * 构造函数
     * @param nums 输入数组
     */
    public SegmentTreeBasic(int[] nums) {
        this.n = nums.length;
        this.arr = new int[n];
        // 线段树数组大小通常为 4*n，确保足够容纳所有节点
        this.sum = new int[n << 2];
        this.lazy = new int[n << 2];

        // 复制原数组
        for (int i = 0; i < n; i++) {
            arr[i] = nums[i];
        }

        // 构建线段树
```

```
    build(1, n, 1);
}

/**
 * 构建线段树
 * @param l 区间左边界（从 1 开始）
 * @param r 区间右边界
 * @param rt 当前节点在 sum 数组中的索引
 */
private void build(int l, int r, int rt) {
    // 如果是叶子节点，直接赋值
    if (l == r) {
        sum[rt] = arr[l - 1]; // 注意数组索引从 0 开始
        return;
    }

    // 计算中点
    int mid = l + ((r - l) >> 1);

    // 递归构建左右子树
    build(l, mid, rt << 1);
    build(mid + 1, r, rt << 1 | 1);

    // 合并左右子树信息
    pushUp(rt);
}

/**
 * 向上更新节点信息
 * @param rt 当前节点索引
 */
private void pushUp(int rt) {
    sum[rt] = sum[rt << 1] + sum[rt << 1 | 1];
}

/**
 * 下推懒标记
 * @param rt 当前节点索引
 * @param ln 左子树节点数量
 * @param rn 右子树节点数量
 */
private void pushDown(int rt, int ln, int rn) {
    // 如果当前节点有懒标记
```

```
    if (lazy[rt] != 0) {
        // 将懒标记传递给左右子节点
        lazy[rt << 1] += lazy[rt];
        lazy[rt << 1 | 1] += lazy[rt];

        // 更新左右子节点的区间和
        sum[rt << 1] += lazy[rt] * ln;
        sum[rt << 1 | 1] += lazy[rt] * rn;

        // 清除当前节点的懒标记
        lazy[rt] = 0;
    }
}

/**
 * 区间更新操作
 * @param L 更新区间左边界
 * @param R 更新区间右边界
 * @param C 更新值
 */
public void update(int L, int R, int C) {
    update(L, R, C, 1, n, 1);
}

/**
 * 区间更新操作（内部实现）
 * @param L 更新区间左边界
 * @param R 更新区间右边界
 * @param C 更新值
 * @param l 当前节点区间左边界
 * @param r 当前节点区间右边界
 * @param rt 当前节点索引
 */
private void update(int L, int R, int C, int l, int r, int rt) {
    // 如果当前节点区间完全包含在更新区间内
    if (L <= l && r <= R) {
        sum[rt] += C * (r - l + 1);
        lazy[rt] += C;
        return;
    }

    // 计算中点
    int mid = l + ((r - l) >> 1);
```

```
    // 下推懒标记
    pushDown(rt, mid - l + 1, r - mid);

    // 递归更新左右子树
    if (L <= mid) {
        update(L, R, C, l, mid, rt << 1);
    }
    if (R > mid) {
        update(L, R, C, mid + 1, r, rt << 1 | 1);
    }

    // 向上更新节点信息
    pushUp(rt);
}

/**
 * 区间查询操作
 * @param L 查询区间左边界
 * @param R 查询区间右边界
 * @return 区间和
 */
public int query(int L, int R) {
    return query(L, R, 1, n, 1);
}

/**
 * 区间查询操作（内部实现）
 * @param L 查询区间左边界
 * @param R 查询区间右边界
 * @param l 当前节点区间左边界
 * @param r 当前节点区间右边界
 * @param rt 当前节点索引
 * @return 区间和
 */
private int query(int L, int R, int l, int r, int rt) {
    // 如果当前节点区间完全包含在查询区间内
    if (L <= l && r <= R) {
        return sum[rt];
    }

    // 计算中点
    int mid = l + ((r - l) >> 1);
```

```
        // 下推懒标记
        pushDown(rt, mid - l + 1, r - mid);

        int ans = 0;

        // 递归查询左右子树
        if (L <= mid) {
            ans += query(L, R, l, mid, rt << 1);
        }
        if (R > mid) {
            ans += query(L, R, mid + 1, r, rt << 1 | 1);
        }

        return ans;
    }

    /**
     * 单点更新操作
     * @param index 更新位置（从1开始）
     * @param value 更新值
     */
    public void updatePoint(int index, int value) {
        // 先查询当前值，然后计算差值进行区间更新
        int oldValue = query(index, index);
        update(index, index, value - oldValue);
    }

    /**
     * 获取数组长度
     * @return 数组长度
     */
    public int size() {
        return n;
    }

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

        System.out.println("初始数组: [1, 3, 5, 7, 9, 11]");
```

```
        System.out.println("查询区间[1,3]的和: " + segTree.query(1, 3)); // 应该输出 9 (1+3+5)
        System.out.println("查询区间[2,5]的和: " + segTree.query(2, 5)); // 应该输出 24 (3+5+7+9)

        // 区间更新：将区间[2,4]都加上 2
        segTree.update(2, 4, 2);
        System.out.println("将区间[2,4]都加上 2 后:");
        System.out.println("查询区间[1,3]的和: " + segTree.query(1, 3)); // 应该输出 15 (1+5+9)
        System.out.println("查询区间[2,5]的和: " + segTree.query(2, 5)); // 应该输出 30 (5+9+11)

        // 单点更新：将位置 3 的值更新为 10
        segTree.updatePoint(3, 10);
        System.out.println("将位置 3 的值更新为 10 后:");
        System.out.println("查询区间[1,3]的和: " + segTree.query(1, 3)); // 应该输出 16 (1+5+10)
    }
}
```

==========

文件: segment_tree_basic.cpp

==========

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

/**
 * 线段树基本实现
 * 支持区间求和、区间更新操作
 *
 * 线段树是一种二叉树结构，每个节点代表一个区间，用于高效处理区间查询和更新操作。
 *
 * 核心思想：
 * 1. 将数组区间划分为更小的子区间，直到单个元素
 * 2. 每个节点存储其对应区间的相关信息（如区间和）
 * 3. 通过合并子区间信息来维护父区间信息
 *
 * 时间复杂度：
 * - 建树：O(n)
 * - 单点更新：O(log n)
 * - 区间查询：O(log n)
 * - 区间更新：O(log n)
 *
 * 空间复杂度：O(n)
 */
```

```
class SegmentTreeBasic {
private:
    vector<int> arr;  // 原数组
    vector<int> sum;  // 线段树数组，存储区间和
    vector<int> lazy; // 懒标记数组，用于区间更新
    int n;            // 数组长度

    /**
     * 构建线段树
     * @param l 区间左边界（从 1 开始）
     * @param r 区间右边界
     * @param rt 当前节点在 sum 数组中的索引
     */
    void build(int l, int r, int rt) {
        // 如果是叶子节点，直接赋值
        if (l == r) {
            sum[rt] = arr[l - 1]; // 注意数组索引从 0 开始
            return;
        }

        // 计算中点
        int mid = l + ((r - l) >> 1);

        // 递归构建左右子树
        build(l, mid, rt << 1);
        build(mid + 1, r, rt << 1 | 1);

        // 合并左右子树信息
        pushUp(rt);
    }

    /**
     * 向上更新节点信息
     * @param rt 当前节点索引
     */
    void pushUp(int rt) {
        sum[rt] = sum[rt << 1] + sum[rt << 1 | 1];
    }

    /**
     * 下推懒标记
     * @param rt 当前节点索引
     * @param ln 左子树节点数量
```

```
 * @param rn 右子树节点数量
 */
void pushDown(int rt, int ln, int rn) {
    // 如果当前节点有懒标记
    if (lazy[rt] != 0) {
        // 将懒标记传递给左右子节点
        lazy[rt << 1] += lazy[rt];
        lazy[rt << 1 | 1] += lazy[rt];

        // 更新左右子节点的区间和
        sum[rt << 1] += lazy[rt] * ln;
        sum[rt << 1 | 1] += lazy[rt] * rn;

        // 清除当前节点的懒标记
        lazy[rt] = 0;
    }
}

/**
 * 区间更新操作（内部实现）
 * @param L 更新区间左边界
 * @param R 更新区间右边界
 * @param C 更新值
 * @param l 当前节点区间左边界
 * @param r 当前节点区间右边界
 * @param rt 当前节点索引
 */
void update(int L, int R, int C, int l, int r, int rt) {
    // 如果当前节点区间完全包含在更新区间内
    if (L <= l && r <= R) {
        sum[rt] += C * (r - l + 1);
        lazy[rt] += C;
        return;
    }

    // 计算中点
    int mid = l + ((r - l) >> 1);

    // 下推懒标记
    pushDown(rt, mid - l + 1, r - mid);

    // 递归更新左右子树
    if (L <= mid) {
```

```
        update(L, R, C, l, mid, rt << 1);
    }
    if (R > mid) {
        update(L, R, C, mid + 1, r, rt << 1 | 1);
    }

    // 向上更新节点信息
    pushUp(rt);
}

/**
 * 区间查询操作（内部实现）
 * @param L 查询区间左边界
 * @param R 查询区间右边界
 * @param l 当前节点区间左边界
 * @param r 当前节点区间右边界
 * @param rt 当前节点索引
 * @return 区间和
 */
int query(int L, int R, int l, int r, int rt) {
    // 如果当前节点区间完全包含在查询区间内
    if (L <= l && r <= R) {
        return sum[rt];
    }

    // 计算中点
    int mid = l + ((r - l) >> 1);

    // 下推懒标记
    pushDown(rt, mid - l + 1, r - mid);

    int ans = 0;

    // 递归查询左右子树
    if (L <= mid) {
        ans += query(L, R, l, mid, rt << 1);
    }
    if (R > mid) {
        ans += query(L, R, mid + 1, r, rt << 1 | 1);
    }

    return ans;
}
```

```
public:
    /**
     * 构造函数
     * @param nums 输入数组
     */
    SegmentTreeBasic(vector<int>& nums) {
        this->n = nums.size();
        this->arr = nums;
        // 线段树数组大小通常为 4*n，确保足够容纳所有节点
        this->sum.resize(n << 2);
        this->lazy.resize(n << 2);

        // 构建线段树
        build(1, n, 1);
    }

    /**
     * 区间更新操作
     * @param L 更新区间左边界
     * @param R 更新区间右边界
     * @param C 更新值
     */
    void update(int L, int R, int C) {
        update(L, R, C, 1, n, 1);
    }

    /**
     * 区间查询操作
     * @param L 查询区间左边界
     * @param R 查询区间右边界
     * @return 区间和
     */
    int query(int L, int R) {
        return query(L, R, 1, n, 1);
    }

    /**
     * 单点更新操作
     * @param index 更新位置（从 1 开始）
     * @param value 更新值
     */
    void updatePoint(int index, int value) {
```

```
        // 先查询当前值，然后计算差值进行区间更新
        int oldValue = query(index, index);
        update(index, index, value - oldValue);
    }

    /**
     * 获取数组长度
     * @return 数组长度
     */
    int size() {
        return n;
    }
};

// 测试函数
int main() {
    cout << "测试线段树实现..." << endl;

    // 测试用例
    vector<int> nums = {1, 3, 5, 7, 9, 11};
    SegmentTreeBasic segTree(nums);

    cout << "初始数组: [1, 3, 5, 7, 9, 11]" << endl;
    cout << "查询区间[1,3]的和: " << segTree.query(1, 3) << endl; // 应该输出 9 (1+3+5)
    cout << "查询区间[2,5]的和: " << segTree.query(2, 5) << endl; // 应该输出 24 (3+5+7+9)

    // 区间更新：将区间[2,4]都加上 2
    segTree.update(2, 4, 2);
    cout << "将区间[2,4]都加上 2 后:" << endl;
    cout << "查询区间[1,3]的和: " << segTree.query(1, 3) << endl; // 应该输出 15 (1+5+9)
    cout << "查询区间[2,5]的和: " << segTree.query(2, 5) << endl; // 应该输出 30 (5+9+11)

    // 单点更新：将位置 3 的值更新为 10
    segTree.updatePoint(3, 10);
    cout << "将位置 3 的值更新为 10 后:" << endl;
    cout << "查询区间[1,3]的和: " << segTree.query(1, 3) << endl; // 应该输出 16 (1+5+10)

    cout << "线段树测试完成！" << endl;

    return 0;
}

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

文件: segment_tree_basic.py
==========

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

"""
线段树基本实现
支持区间求和、区间更新操作

线段树是一种二叉树结构，每个节点代表一个区间，用于高效处理区间查询和更新操作。

核心思想：
1. 将数组区间划分为更小的子区间，直到单个元素
2. 每个节点存储其对应区间的相关信息（如区间和）
3. 通过合并子区间信息来维护父区间信息

时间复杂度：
- 建树：O(n)
- 单点更新：O(log n)
- 区间查询：O(log n)
- 区间更新：O(log n)

空间复杂度：O(n)
"""



class SegmentTreeBasic:
    """
    线段树基本实现类
    支持区间求和、区间更新操作
    """

    def __init__(self, nums):
        """
        构造函数
        :param nums: 输入数组
        """
        self.n = len(nums)
        self.arr = nums[:]
        # 线段树数组大小通常为 4*n，确保足够容纳所有节点
        self.sum = [0] * (self.n << 2)
        self.lazy = [0] * (self.n << 2)
```

```
    # 构建线段树
    self._build(1, self.n, 1)

def _build(self, l, r, rt):
    """
    构建线段树
    :param l: 区间左边界（从 1 开始）
    :param r: 区间右边界
    :param rt: 当前节点在 sum 数组中的索引
    """
    # 如果是叶子节点，直接赋值
    if l == r:
        self.sum[rt] = self.arr[l - 1]  # 注意数组索引从 0 开始
        return

    # 计算中点
    mid = l + ((r - l) >> 1)

    # 递归构建左右子树
    self._build(l, mid, rt << 1)
    self._build(mid + 1, r, rt << 1 | 1)

    # 合并左右子树信息
    self._push_up(rt)

def _push_up(self, rt):
    """
    向上更新节点信息
    :param rt: 当前节点索引
    """
    self.sum[rt] = self.sum[rt << 1] + self.sum[rt << 1 | 1]

def _push_down(self, rt, ln, rn):
    """
    下推懒标记
    :param rt: 当前节点索引
    :param ln: 左子树节点数量
    :param rn: 右子树节点数量
    """
    # 如果当前节点有懒标记
    if self.lazy[rt] != 0:
        # 将懒标记传递给左右子节点
```

```
        self.lazy[rt << 1] += self.lazy[rt]
        self.lazy[rt << 1 | 1] += self.lazy[rt]

        # 更新左右子节点的区间和
        self.sum[rt << 1] += self.lazy[rt] * ln
        self.sum[rt << 1 | 1] += self.lazy[rt] * rn

        # 清除当前节点的懒标记
        self.lazy[rt] = 0

def update(self, L, R, C):
    """
    区间更新操作
    :param L: 更新区间左边界
    :param R: 更新区间右边界
    :param C: 更新值
    """
    self._update(L, R, C, 1, self.n, 1)

def _update(self, L, R, C, l, r, rt):
    """
    区间更新操作（内部实现）
    :param L: 更新区间左边界
    :param R: 更新区间右边界
    :param C: 更新值
    :param l: 当前节点区间左边界
    :param r: 当前节点区间右边界
    :param rt: 当前节点索引
    """
    # 如果当前节点区间完全包含在更新区间内
    if L <= l and r <= R:
        self.sum[rt] += C * (r - l + 1)
        self.lazy[rt] += C
        return

    # 计算中点
    mid = l + ((r - l) >> 1)

    # 下推懒标记
    self._push_down(rt, mid - l + 1, r - mid)

    # 递归更新左右子树
    if L <= mid:
```

```
            self._update(L, R, C, l, mid, rt << 1)
        if R > mid:
            self._update(L, R, C, mid + 1, r, rt << 1 | 1)

        # 向上更新节点信息
        self._push_up(rt)

    def query(self, L, R):
        """
        区间查询操作
        :param L: 查询区间左边界
        :param R: 查询区间右边界
        :return: 区间和
        """
        return self._query(L, R, 1, self.n, 1)

    def _query(self, L, R, l, r, rt):
        """
        区间查询操作（内部实现）
        :param L: 查询区间左边界
        :param R: 查询区间右边界
        :param l: 当前节点区间左边界
        :param r: 当前节点区间右边界
        :param rt: 当前节点索引
        :return: 区间和
        """
        # 如果当前节点区间完全包含在查询区间内
        if L <= l and r <= R:
            return self.sum[rt]

        # 计算中点
        mid = l + ((r - l) >> 1)

        # 下推懒标记
        self._push_down(rt, mid - l + 1, r - mid)

        ans = 0

        # 递归查询左右子树
        if L <= mid:
            ans += self._query(L, R, l, mid, rt << 1)
        if R > mid:
            ans += self._query(L, R, mid + 1, r, rt << 1 | 1)
```

```
        return ans

    def update_point(self, index, value):
        """
        单点更新操作
        :param index: 更新位置（从1开始）
        :param value: 更新值
        """
        # 先查询当前值，然后计算差值进行区间更新
        old_value = self.query(index, index)
        self.update(index, index, value - old_value)

    def size(self):
        """
        获取数组长度
        :return: 数组长度
        """
        return self.n


# 测试函数
def test_segment_tree():
    """测试线段树实现"""
    print("测试线段树实现...")

    # 测试用例
    nums = [1, 3, 5, 7, 9, 11]
    seg_tree = SegmentTreeBasic(nums)

    print("初始数组: [1, 3, 5, 7, 9, 11]")
    print(f"查询区间[1,3]的和: {seg_tree.query(1, 3)}")  # 应该输出9 (1+3+5)
    print(f"查询区间[2,5]的和: {seg_tree.query(2, 5)}")  # 应该输出24 (3+5+7+9)

    # 区间更新：将区间[2,4]都加上2
    seg_tree.update(2, 4, 2)
    print("将区间[2,4]都加上2后:")
    print(f"查询区间[1,3]的和: {seg_tree.query(1, 3)}")  # 应该输出15 (1+5+9)
    print(f"查询区间[2,5]的和: {seg_tree.query(2, 5)}")  # 应该输出30 (5+9+11)

    # 单点更新：将位置3的值更新为10
    seg_tree.update_point(3, 10)
    print("将位置3的值更新为10后:")
```

```
    print(f"查询区间[1,3]的和: {seg_tree.query(1, 3)}")  # 应该输出 16 (1+5+10)

    print("线段树测试完成！")


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

==========

文件: TestMain.java

==========

```
import java.util.Arrays;

/**
 * 翻转对
 * 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。
 *
 * 示例 1:
 * 输入: [1,3,2,3,1]
 * 输出: 2
 *
 * 示例 2:
 * 输入: [2,4,3,5,1]
 * 输出: 3
 *
 * 解题思路：
 * 1. 使用归并排序的思想解决
 * 2. 在归并的过程中统计翻转对的数量
 * 3. 对于左半部分[l...m]和右半部分[m+1...r]：
 *    - 在合并前，先统计翻转对数量：对于左半部分的每个元素 nums[i]，统计右半部分有多少个元素 nums[j]满足 nums[i] > 2*nums[j]
 *    - 然后进行正常的归并排序合并过程
 *
 * 时间复杂度分析：
 * - 归并排序的时间复杂度为 O(n log n)
 * - 每一层递归都会遍历所有元素，共 log n 层
 * - 所以总时间复杂度为 O(n log n)
 *
 * 空间复杂度分析：
 * - 需要额外的 help 数组存储临时数据，空间复杂度为 O(n)
 * - 递归调用栈的深度为 O(log n)
```

```
 * - 所以总空间复杂度为O(n)
 *
 * 测试链接: https://leetcode.cn/problems/reverse-pairs/
 */
public class TestMain {

    /**
     * 计算翻转对数量
     *
     * @param nums 输入数组
     * @return 翻转对数量
     */
    public static int reversePairs(int[] nums) {
        if (nums == null || nums.length < 2) {
            return 0;
        }
        return process(nums, 0, nums.length - 1);
    }

    /**
     * 递归处理区间[l,r]内的翻转对
     *
     * @param nums 数组
     * @param l 区间左边界
     * @param r 区间右边界
     * @return 区间内翻转对数量
     */
    public static int process(int[] nums, int l, int r) {
        if (l == r) {
            return 0;
        }
        int m = l + ((r - l) >> 1);
        return process(nums, l, m) + process(nums, m + 1, r) + merge(nums, l, m, r);
    }

    /**
     * 合并两个有序数组，并统计合并过程中产生的翻转对数量
     *
     * @param nums 数组
     * @param l 左半部分起始位置
     * @param m 左半部分结束位置
     * @param r 右半部分结束位置
     * @return 合并过程中产生的翻转对数量
```

```
 */
public static int merge(int[] nums, int l, int m, int r) {
    // 统计翻转对数量
    int ans = 0;
    // 对于左半部分的每个元素 nums[i]，统计右半部分有多少个元素 nums[j]满足 nums[i] > 2*nums[j]
    for (int i = l; i <= m; i++) {
        // 在右半部分找到第一个不满足 nums[i] <= 2*nums[j]的位置
        int j = m + 1;
        while (j <= r && (long) nums[i] <= 2 * (long) nums[j]) {
            j++;
        }
        // 从 j 到 r 的所有元素都与 nums[i]构成翻转对
        ans += r - j + 1;
    }

    // 正常的归并排序合并过程
    int[] help = new int[r - l + 1];
    int i = 0;
    int a = l;
    int b = m + 1;
    while (a <= m && b <= r) {
        help[i++] = nums[a] <= nums[b] ? nums[a++] : nums[b++];
    }
    while (a <= m) {
        help[i++] = nums[a++];
    }
    while (b <= r) {
        help[i++] = nums[b++];
    }
    for (i = 0; i < help.length; i++) {
        nums[l + i] = help[i];
    }
    return ans;
}

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

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

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

文件: TEST_ALL.py

```
==========================================
"""
测试所有算法实现的正确性
"""

import time

# 导入所有实现的算法
from Code06_ReversePairs import Solution as ReversePairsSolution
from Code07_LIS_BIT import Solution as LISSolution
from Code08_NumberOfLISAdvanced import Solution as NumberOfLISSolution
from Code09_GoodTriplets import Solution as GoodTripletsSolution

def test_reverse_pairs():
    """测试翻转对算法"""
    print("测试翻转对算法...")
    solution = ReversePairsSolution()

    # 测试用例 1
    nums1 = [1, 3, 2, 3, 1]
    result1 = solution.reversePairs(nums1)
    print(f"输入: {nums1}")
    print(f"输出: {result1}")
    print(f"期望: 2")
    assert result1 == 2, f"测试失败，期望 2，实际{result1}"

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

```
    assert result2 == 3, f"测试失败，期望 3，实际{result2}"

    print("翻转对算法测试通过！\n")

def test_lis():
    """测试最长递增子序列算法"""
    print("测试最长递增子序列算法...")
    solution = LISSolution()

    # 测试用例 1
    nums1 = [10, 9, 2, 5, 3, 7, 101, 18]
    result1 = solution.lengthOfLIS(nums1)
    print(f"输入: {nums1}")
    print(f"输出: {result1}")
    print(f"期望: 4")
    assert result1 == 4, f"测试失败，期望 4，实际{result1}"

    # 测试用例 2
    nums2 = [0, 1, 0, 3, 2, 3]
    result2 = solution.lengthOfLIS(nums2)
    print(f"输入: {nums2}")
    print(f"输出: {result2}")
    print(f"期望: 4")
    assert result2 == 4, f"测试失败，期望 4，实际{result2}"

    # 测试用例 3
    nums3 = [7, 7, 7, 7, 7, 7, 7]
    result3 = solution.lengthOfLIS(nums3)
    print(f"输入: {nums3}")
    print(f"输出: {result3}")
    print(f"期望: 1")
    assert result3 == 1, f"测试失败，期望 1，实际{result3}"

    print("最长递增子序列算法测试通过！\n")

def test_number_of_lis():
    """测试最长递增子序列的个数算法"""
    print("测试最长递增子序列的个数算法...")
    solution = NumberOfLISSolution()

    # 测试用例 1
    nums1 = [1, 3, 5, 4, 7]
    result1 = solution.findNumberOfLIS(nums1)
```

```
    print(f"输入: {nums1}")
    print(f"输出: {result1}")
    print(f"期望: 2")
    assert result1 == 2, f"测试失败，期望 2，实际{result1}"

    # 测试用例 2
    nums2 = [2, 2, 2, 2, 2]
    result2 = solution.findNumberOfLIS(nums2)
    print(f"输入: {nums2}")
    print(f"输出: {result2}")
    print(f"期望: 5")
    assert result2 == 5, f"测试失败，期望 5，实际{result2}"

    print("最长递增子序列的个数算法测试通过！\n")

def test_good_triplets():
    """测试好三元组数目算法"""
    print("测试好三元组数目算法...")
    solution = GoodTripletsSolution()

    # 测试用例 1
    nums1_1 = [2, 0, 1, 3]
    nums2_1 = [0, 1, 2, 3]
    result1 = solution.goodTriplets(nums1_1, nums2_1)
    print(f"输入: nums1 = {nums1_1}, nums2 = {nums2_1}")
    print(f"输出: {result1}")
    print(f"期望: 1")
    assert result1 == 1, f"测试失败，期望 1，实际{result1}"

    # 测试用例 2
    nums1_2 = [4, 0, 1, 3, 2]
    nums2_2 = [4, 1, 0, 2, 3]
    result2 = solution.goodTriplets(nums1_2, nums2_2)
    print(f"输入: nums1 = {nums1_2}, nums2 = {nums2_2}")
    print(f"输出: {result2}")
    print(f"期望: 4")
    assert result2 == 4, f"测试失败，期望 4，实际{result2}"

    print("好三元组数目算法测试通过！\n")

def performance_test():
    """性能测试"""
    print("性能测试...")
```

```
    # 生成大规模测试数据
    import random
    n = 10000
    nums = [random.randint(1, 100000) for _ in range(n)]

    # 测试翻转对算法性能
    solution = ReversePairsSolution()
    start_time = time.time()
    result = solution.reversePairs(nums)
    end_time = time.time()
    print(f"翻转对算法处理{n}个元素耗时: {end_time - start_time:.4f}秒")

    print("性能测试完成！\n")

def main():
    """主测试函数"""
    print("开始测试所有算法实现...\n")

    try:
        test_reverse_pairs()
        test_lis()
        test_number_of_lis()
        test_good_triplets()
        performance_test()

        print("所有测试通过！所有算法实现正确。")
    except Exception as e:
        print(f"测试失败: {e}")
        raise

if __name__ == "__main__":
    main()

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

文件: test_segment_tree_problems.py
================================================
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

"""
线段树相关题目测试文件
```

测试所有线段树实现的正确性
"""

```
import time
from leetcode307_segment_tree import LeetCode307_SegmentTree
from leetcode315_segment_tree import LeetCode315_SegmentTree
from leetcode493_segment_tree import LeetCode493_SegmentTree
from leetcode327_segment_tree import LeetCode327_SegmentTree
from poj3468_segment_tree import POJ3468_SegmentTree
from hdu1166_segment_tree import HDU1166_SegmentTree
from luogu_p3372_segment_tree import LuoguP3372_SegmentTree
from codeforces_52c_segment_tree import Codeforces52C_SegmentTree


def test_leetcode307():
    """测试 LeetCode 307. 区域和检索 - 数组可修改"""
    print("测试 LeetCode 307. 区域和检索 - 数组可修改...")

    # 测试用例 1
    nums1 = [1, 3, 5]
    numArray1 = LeetCode307_SegmentTree(nums1)

    result1 = numArray1.sumRange(0, 2)
    print(f"输入数组: {nums1}")
    print(f"查询区间[0,2]的和: {result1}")
    assert result1 == 9, f"测试失败，期望 9，实际{result1}"

    # 更新索引 1 的值为 2
    numArray1.update(1, 2)
    result2 = numArray1.sumRange(0, 2)
    print(f"将索引 1 的值更新为 2 后，查询区间[0,2]的和: {result2}")
    assert result2 == 8, f"测试失败，期望 8，实际{result2}"

    # 测试用例 2
    nums2 = [9, -8]
    numArray2 = LeetCode307_SegmentTree(nums2)

    numArray2.update(0, 3)
    result3 = numArray2.sumRange(1, 1)
    print(f"输入数组: {nums2}")
    print(f"将索引 0 的值更新为 3 后，查询区间[1,1]的和: {result3}")
    assert result3 == -8, f"测试失败，期望-8，实际{result3}"
```

```
    result4 = numArray2.sumRange(0, 1)
    print(f"查询区间[0,1]的和: {result4}")
    assert result4 == -5, f"测试失败，期望-5，实际{result4}"

    numArray2.update(1, -3)
    result5 = numArray2.sumRange(0, 1)
    print(f"将索引 1 的值更新为-3 后，查询区间[0,1]的和: {result5}")
    assert result5 == 0, f"测试失败，期望 0，实际{result5}"

    print("LeetCode 307 测试通过！\n")


def test_leetcode315():
    """测试 LeetCode 315. 计算右侧小于当前元素的个数"""
    print("测试 LeetCode 315. 计算右侧小于当前元素的个数...")

    # 测试用例 1
    solution1 = LeetCode315_SegmentTree()
    nums1 = [5, 2, 6, 1]
    result1 = solution1.countSmaller(nums1)
    print(f"输入数组: {nums1}")
    print(f"输出结果: {result1}")
    assert result1 == [2, 1, 1, 0], f"测试失败，期望[2, 1, 1, 0]，实际{result1}"

    # 测试用例 2
    solution2 = LeetCode315_SegmentTree()
    nums2 = [-1]
    result2 = solution2.countSmaller(nums2)
    print(f"输入数组: {nums2}")
    print(f"输出结果: {result2}")
    assert result2 == [0], f"测试失败，期望[0]，实际{result2}"

    # 测试用例 3
    solution3 = LeetCode315_SegmentTree()
    nums3 = [-1, -1]
    result3 = solution3.countSmaller(nums3)
    print(f"输入数组: {nums3}")
    print(f"输出结果: {result3}")
    assert result3 == [0, 0], f"测试失败，期望[0, 0]，实际{result3}"

    print("LeetCode 315 测试通过！\n")
```

```
def test_leetcode493():
    """测试 LeetCode 493. 翻转对"""
    print("测试 LeetCode 493. 翻转对...")

    # 测试用例 1
    solution1 = LeetCode493_SegmentTree()
    nums1 = [1, 3, 2, 3, 1]
    result1 = solution1.reversePairs(nums1)
    print(f"输入数组: {nums1}")
    print(f"输出结果: {result1}")
    assert result1 == 2, f"测试失败，期望 2，实际{result1}"

    # 测试用例 2
    solution2 = LeetCode493_SegmentTree()
    nums2 = [2, 4, 3, 5, 1]
    result2 = solution2.reversePairs(nums2)
    print(f"输入数组: {nums2}")
    print(f"输出结果: {result2}")
    assert result2 == 3, f"测试失败，期望 3，实际{result2}"

    # 测试用例 3
    solution3 = LeetCode493_SegmentTree()
    nums3 = [2147483647, 2147483647, 2147483647, 2147483647, 2147483647, 2147483647]
    result3 = solution3.reversePairs(nums3)
    print(f"输入数组: {nums3}")
    print(f"输出结果: {result3}")
    assert result3 == 0, f"测试失败，期望 0，实际{result3}"

    print("LeetCode 493 测试通过！\n")


def test_leetcode327():
    """测试 LeetCode 327. 区间和的个数"""
    print("测试 LeetCode 327. 区间和的个数...")

    # 测试用例 1
    solution1 = LeetCode327_SegmentTree()
    nums1 = [-2, 5, -1]
    lower1, upper1 = -2, 2
    result1 = solution1.countRangeSum(nums1, lower1, upper1)
    print(f"输入数组: {nums1}, lower = {lower1}, upper = {upper1}")
    print(f"输出结果: {result1}")
    assert result1 == 3, f"测试失败，期望 3，实际{result1}"
```

```
    # 测试用例 2
    solution2 = LeetCode327_SegmentTree()
    nums2 = [0]
    lower2, upper2 = 0, 0
    result2 = solution2.countRangeSum(nums2, lower2, upper2)
    print(f"输入数组: {nums2}, lower = {lower2}, upper = {upper2}")
    print(f"输出结果: {result2}")
    assert result2 == 1, f"测试失败，期望 1，实际{result2}"

    # 测试用例 3
    solution3 = LeetCode327_SegmentTree()
    nums3 = [2147483647, -2147483648, -1, 0]
    lower3, upper3 = -1, 0
    result3 = solution3.countRangeSum(nums3, lower3, upper3)
    print(f"输入数组: {nums3}, lower = {lower3}, upper = {upper3}")
    print(f"输出结果: {result3}")

    print("LeetCode 327 测试通过！\n")


def test_poj3468():
    """测试 POJ 3468 A Simple Problem with Integers"""
    print("测试 POJ 3468 A Simple Problem with Integers...")

    # 测试用例
    nums = [1, 2, 3, 4, 5]
    seg_tree = POJ3468_SegmentTree(nums)

    print(f"初始数组: {nums}")
    result1 = seg_tree.query(1, 3)
    print(f"查询区间[1,3]的和: {result1}")
    assert result1 == 9, f"测试失败，期望 9，实际{result1}"

    # 区间更新：将区间[1,3]中的每个元素都加上 2
    seg_tree.update(1, 3, 2)
    result2 = seg_tree.query(1, 3)
    print("将区间[1,3]中的每个元素都加上 2 后:")
    print(f"查询区间[1,3]的和: {result2}")
    assert result2 == 15, f"测试失败，期望 15，实际{result2}"

    print("POJ 3468 测试通过！\n")
```

```
def test_hdu1166():
    """测试 HDU 1166 敌兵布阵"""
    print("测试 HDU 1166 敌兵布阵...")

    # 测试用例
    nums = [1, 2, 3, 4, 5]
    seg_tree = HDU1166_SegmentTree(nums)

    print(f"初始数组: {nums}")
    result1 = seg_tree.query(1, 3)
    print(f"查询区间[1,3]的和: {result1}")
    assert result1 == 9, f"测试失败，期望 9，实际{result1}"

    # 单点更新：第 2 个营地增加 3 个士兵
    seg_tree.add(2, 3)
    result2 = seg_tree.query(1, 3)
    print("第 2 个营地增加 3 个士兵后:")
    print(f"查询区间[1,3]的和: {result2}")
    assert result2 == 12, f"测试失败，期望 12，实际{result2}"

    print("HDU 1166 测试通过！\n")


def test_luogu_p3372():
    """测试洛谷 P3372 【模板】线段树 1"""
    print("测试洛谷 P3372 【模板】线段树 1...")

    # 测试用例
    nums = [1, 2, 3, 4, 5]
    seg_tree = LuoguP3372_SegmentTree(nums)

    print(f"初始数组: {nums}")
    result1 = seg_tree.query(1, 3)
    print(f"查询区间[1,3]的和: {result1}")
    assert result1 == 9, f"测试失败，期望 9，实际{result1}"

    # 区间更新：将区间[1,3]中的每个元素都加上 2
    seg_tree.update(1, 3, 2)
    result2 = seg_tree.query(1, 3)
    print("将区间[1,3]中的每个元素都加上 2 后:")
    print(f"查询区间[1,3]的和: {result2}")
    assert result2 == 15, f"测试失败，期望 15，实际{result2}"
```

```
    print("洛谷 P3372 测试通过！\n")


def test_codeforces_52c():
    """测试 Codeforces 52C Circular RMQ"""
    print("测试 Codeforces 52C Circular RMQ...")

    # 测试用例
    nums = [1, 2, 3, 4, 5]
    seg_tree = Codeforces52C_SegmentTree(nums)

    print(f"初始数组: {nums}")
    result1 = seg_tree.query(1, 3)
    print(f"查询区间[1,3]的最小值: {result1}")
    assert result1 == 2, f"测试失败，期望 2，实际{result1}"

    # 区间更新：将区间[1,3]中的每个元素都加上 2
    seg_tree.update(1, 3, 2)
    result2 = seg_tree.query(1, 3)
    print("将区间[1,3]中的每个元素都加上 2 后:")
    print(f"查询区间[1,3]的最小值: {result2}")
    assert result2 == 4, f"测试失败，期望 4，实际{result2}"

    # 环形区间查询：查询区间[3,1]（环形）
    # 环形区间[3,1]包含元素索引 3,4,0,1，对应值为 4,5,1,2，加上之前的更新后为 4,5,1,4，最小值是 1
    result3 = seg_tree.query(3, 1)
    print(f"环形区间[3,1]的最小值: {result3}")
    assert result3 == 1, f"测试失败，期望 1，实际{result3}"

    print("Codeforces 52C 测试通过！\n")


def performance_test():
    """性能测试"""
    print("性能测试...")

    # 生成大规模测试数据
    import random
    n = 10000
    nums = [random.randint(-10000, 10000) for _ in range(n)]

    # 测试 LeetCode 315 性能
```

```
    solution = LeetCode315_SegmentTree()
    start_time = time.time()
    result = solution.countSmaller(nums)
    end_time = time.time()
    print(f"LeetCode 315 处理{n}个元素耗时: {end_time - start_time:.4f}秒")

    print("性能测试完成！\n")


def main():
    """主测试函数"""
    print("开始测试所有线段树相关题目实现...\n")

    try:
        test_leetcode307()
        test_leetcode315()
        test_leetcode493()
        test_leetcode327()
        test_poj3468()
        test_hdu1166()
        test_luogu_p3372()
        test_codeforces_52c()
        performance_test()

        print("所有测试通过！所有线段树实现正确。")
    except Exception as e:
        print(f"测试失败: {e}")
        raise


if __name__ == "__main__":
    main()

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