

=====

文件夹: class104_SegmentTreeAndBinaryIndexedTreeAlgorithms

=====

[Markdown 文件]

=====

文件: extended_problems.md

=====

线段树和树状数组专题扩展题目

📚 题目分类

基础题目

1. **LeetCode 307. Range Sum Query - Mutable**

- 链接: <https://leetcode.cn/problems/range-sum-query-mutable/>
- 难度: 中等
- 算法: 线段树、树状数组
- 文件: Code07_RangeSumQueryMutable_SegmentTree. java, Code08_RangeSumQueryMutable_BIT. java

2. **LeetCode 303. Range Sum Query - Immutable**

- 链接: <https://leetcode.cn/problems/range-sum-query-immutable/>
- 难度: 简单
- 算法: 前缀和
- 说明: 线段树和树状数组的基础应用

3. **LeetCode 304. Range Sum Query 2D - Immutable**

- 链接: <https://leetcode.cn/problems/range-sum-query-2d-immutable/>
- 难度: 中等
- 算法: 二维前缀和
- 说明: 二维版本的基础题目

中级题目

4. **LeetCode 315. Count of Smaller Numbers After Self**

- 链接: <https://leetcode.cn/problems/count-of-smaller-numbers-after-self/>
- 难度: 困难
- 算法: 树状数组、归并排序、离散化
- 文件: Code09_CountSmallerNumbersAfterSelf. java

5. **LeetCode 493. Reverse Pairs**

- 链接: <https://leetcode.cn/problems/reverse-pairs/>
- 难度: 困难
- 算法: 归并排序
- 文件: Code10_ReversePairs. java

6. **LeetCode 327. Count of Range Sum**
- 链接: <https://leetcode.cn/problems/count-of-range-sum/>
- 难度: 困难
- 算法: 归并排序、前缀和

7. **LeetCode 699. Falling Squares**
- 链接: <https://leetcode.cn/problems/falling-squares/>
- 难度: 困难
- 算法: 线段树、坐标离散化、懒惰传播
- 文件: Code11_FallingSquares.java

高级题目

8. **LeetCode 715. Range Module**
- 链接: <https://leetcode.cn/problems/range-module/>
- 难度: 困难
- 算法: 线段树、平衡二叉搜索树
9. **LeetCode 2213. Longest Substring of One Repeating Character**
- 链接: <https://leetcode.cn/problems/longest-substring-of-one-repeating-character/>
- 难度: 困难
- 算法: 线段树、字符串处理
10. **LeetCode 1157. Online Majority Element In Subarray**
- 链接: <https://leetcode.cn/problems/online-majority-element-in-subarray/>
- 难度: 困难
- 算法: 线段树、随机化

🏆 Codeforces 题目

1. **Codeforces 52C. Circular RMQ**
- 链接: <https://codeforces.com/problemset/problem/52/C>
- 算法: 线段树、循环数组
2. **Codeforces 242E. XOR on Segment**
- 链接: <https://codeforces.com/problemset/problem/242/E>
- 算法: 线段树、位运算
3. **Codeforces 438D. The Child and Sequence**
- 链接: <https://codeforces.com/problemset/problem/438/D>
- 算法: 线段树、取模运算
4. **Codeforces 145E. Lucky Queries**

- 链接: <https://codeforces.com/problemset/problem/145/E>
 - 算法: 线段树、字符串处理
5. **Codeforces 833B. The Bakery**
- 链接: <https://codeforces.com/problemset/problem/833/B>
 - 算法: 线段树、动态规划
 - 文件: Code05_TheBakery.java
- ## 🏛 洛谷题目
1. **洛谷 P3372. 【模板】线段树 1**
 - 链接: <https://www.luogu.com.cn/problem/P3372>
 - 算法: 线段树、区间加法、区间求和
 2. **洛谷 P3373. 【模板】线段树 2**
 - 链接: <https://www.luogu.com.cn/problem/P3373>
 - 算法: 线段树、区间乘法、区间加法
 3. **洛谷 P3368. 【模板】树状数组 2**
 - 链接: <https://www.luogu.com.cn/problem/P3368>
 - 算法: 树状数组、区间修改、单点查询
 4. **洛谷 P1908. 逆序对**
 - 链接: <https://www.luogu.com.cn/problem/P1908>
 - 算法: 归并排序、树状数组
 5. **洛谷 P3287. 方伯伯的玉米田**
 - 链接: <https://www.luogu.com.cn/problem/P3287>
 - 算法: 二维树状数组、动态规划
 - 文件: Code03_CornField.java
- ## 💻 LintCode 题目
1. **LintCode 247. Segment Tree Query II**
 - 链接: <https://www.lintcode.com/problem/segment-tree-query-ii/>
 - 算法: 线段树
 2. **LintCode 439. Segment Tree Build II**
 - 链接: <https://www.lintcode.com/problem/segment-tree-build-ii/>
 - 算法: 线段树
- ## 🌐 SPOJ 题目

1. **SPOJ GSS1. Can you answer these queries I**

- 链接: <https://www.spoj.com/problems/GSS1/>

- 算法: 线段树、区间最大子段和

2. **SPOJ GSS3. Can you answer these queries III**

- 链接: <https://www.spoj.com/problems/GSS3/>

- 算法: 线段树、区间最大子段和

3. **SPOJ MKTHNUM. K-th Number**

- 链接: <https://www.spoj.com/problems/MKTHNUM/>

- 算法: 主席树、第 k 小查询

4. **SPOJ DQUERY. D-query**

- 链接: <https://www.spoj.com/problems/DQUERY/>

- 算法: 主席树、莫队算法、区间不同元素个数

🏆 AtCoder 题目

1. **AtCoder ABC185F. Range Xor Query**

- 链接: https://atcoder.jp/contests/abc185/tasks/abc185_f

- 算法: 树状数组、线段树、区间异或

2. **AtCoder ABC234F. Predilection**

- 链接: https://atcoder.jp/contests/abc234/tasks/abc234_f

- 算法: 线段树、动态规划、前缀和

🏆 HackerRank 题目

1. **HackerRank Array Manipulation**

- 链接: <https://www.hackerrank.com/challenges/crush/problem>

- 算法: 差分数组、线段树、区间加法

2. **HackerRank Direct Connections**

- 链接: <https://www.hackerrank.com/challenges/direct-connections/problem>

- 算法: 线段树、排序、数学计算

🐾 USACO 题目

1. **USACO 2015 January Platinum. Grass Cownisseur**

- 链接: <http://www.usaco.org/index.php?page=viewproblem2&cpid=517>

- 算法: 线段树、动态规划、图论

2. **USACO 2018 February Platinum. New Barns**

- 链接: <http://www.usaco.org/index.php?page=viewproblem2&cpid=818>
- 算法: 线段树、树的直径、动态树

🍔 CodeChef 题目

1. **CodeChef HORRIBLE. Horrible Queries**
 - 链接: <https://www.codechef.com/problems/HORRIBLE>
 - 算法: 线段树、懒惰传播、区间更新
2. **CodeChef GSS4. Can you answer these queries IV**
 - 链接: <https://www.codechef.com/problems/GSS4>
 - 算法: 线段树、区间开方、懒惰传播

NEW 新增题目实现

1. **AtCoder ABC185F. Range Xor Query**
 - 链接: https://atcoder.jp/contests/abc185/tasks/abc185_f
 - 算法: 线段树、区间异或
 - 文件: Code14_RangeXORQuery.java, Code14_RangeXORQuery.py
2. **SPOJ GSS1. Can you answer these queries I**
 - 链接: <https://www.spoj.com/problems/GSS1/>
 - 算法: 线段树、区间最大子段和
 - 文件: Code15_MaximumSubarraySum.java, Code15_MaximumSubarraySum.py
3. **SPOJ MKTHNUM. K-th Number**
 - 链接: <https://www.spoj.com/problems/MKTHNUM/>
 - 算法: 主席树、第 k 小查询
 - 文件: Code16_KthNumber.java, Code16_KthNumber.py

🔎 技巧总结

线段树技巧

1. **基础线段树**: 支持单点更新和区间查询
2. **懒惰传播**: 支持区间更新操作
3. **动态开点**: 节省空间, 适用于稀疏数据
4. **标记下传**: 维护区间操作的正确性

树状数组技巧

1. **前缀和查询**: $O(\log n)$ 时间复杂度查询前缀和
2. **单点更新**: $O(\log n)$ 时间复杂度更新单点值
3. **区间修改**: 通过差分数组实现区间修改
4. **二维扩展**: 扩展到二维情况处理矩阵问题

通用技巧

1. **离散化**: 处理大数值范围问题
2. **坐标变换**: 将问题转化为更容易处理的形式
3. **分块处理**: 将大问题分解为小问题处理
4. **数据结构组合**: 结合多种数据结构解决复杂问题

复杂度分析

时间复杂度

- 线段树构建: $O(n)$
- 线段树单点更新: $O(\log n)$
- 线段树区间更新（带懒惰传播）: $O(\log n)$
- 线段树区间查询: $O(\log n)$
- 树状数组单点更新: $O(\log n)$
- 树状数组前缀和查询: $O(\log n)$

空间复杂度

- 线段树: $O(4n)$
- 树状数组: $O(n)$

工程化考量

异常处理

1. **边界条件**: 处理空数组、单元素数组等特殊情况
2. **输入验证**: 检查输入参数的有效性
3. **内存管理**: 避免内存泄漏，合理分配空间

性能优化

1. **常数优化**: 减少不必要的计算和内存访问
2. **缓存友好**: 优化数据结构布局提高缓存命中率
3. **并行化**: 在可能的情况下利用多核处理能力

可维护性

1. **代码结构**: 模块化设计，职责分离
2. **注释文档**: 详细注释关键算法和实现细节
3. **测试覆盖**: 完善的单元测试和边界测试

文件: README.md

线段树和树状数组专题详解

🧠 核心概念

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

线段树（Segment Tree）

- 一种基于分治思想的二叉树数据结构
- 主要用于解决区间查询和区间更新问题
- 每个节点代表一个区间，可以高效地支持区间操作
- 时间复杂度： $O(\log n)$ for 查询和更新操作
- 空间复杂度： $O(4n)$

树状数组（Binary Indexed Tree/Fenwick Tree）

- 一种更简洁的数据结构，主要用于解决单点更新和前缀和查询问题
- 相比线段树，实现更简单，常数更小
- 时间复杂度： $O(\log n)$ for 查询和更新操作
- 空间复杂度： $O(n)$

🎨 项目特色

本项目提供了**Java、C++、Python三语言完整实现**，每个题目都包含：

- ✅ **详细注释**：代码逻辑清晰，注释详尽
- ✅ **复杂度分析**：时间和空间复杂度详细分析
- ✅ **完整测试**：单元测试覆盖各种边界情况
- ✅ **性能优化**：工程化异常处理和性能考量
- ✅ **算法总结**：题型分类和解题技巧总结

📄 本专题题目列表

核心题目

1. **Code01_CountOfRangeSum** - 区间和的个数
 - 来源：LeetCode 327
 - 难度：困难
 - 算法：归并排序、线段树
 - 三语言实现：✅ Java ✅ C++ ✅ Python
 - 时间复杂度： $O(n \log n)$
 - 空间复杂度： $O(n)$
2. **Code02_MaximumBalancedSubsequence** - 平衡子序列的最大和
 - 来源：LeetCode 2784
 - 难度：困难

- 算法: 树状数组、离散化
- 三语言实现: Java C++ Python
- 时间复杂度: $O(n \log n)$
- 空间复杂度: $O(n)$

3. **Code03_CornField** - 方伯伯的玉米田

- 来源: 洛谷 P3287
- 难度: 困难
- 算法: 二维树状数组、动态规划
- 三语言实现: Java C++ Python
- 时间复杂度: $O(n^2 \log n)$
- 空间复杂度: $O(n^2)$

4. **Code04_LongestIdealString** - 最长理想子序列

- 来源: LeetCode 2370
- 难度: 中等
- 算法: 线段树、动态规划
- 三语言实现: Java C++ Python
- 时间复杂度: $O(n)$
- 空间复杂度: $O(1)$

5. **Code05_TheBakery** - 划分 k 段的最大得分

- 来源: Codeforces 833B
- 难度: 困难
- 算法: 线段树、动态规划
- 三语言实现: Java C++ Python
- 时间复杂度: $O(nk \log n)$
- 空间复杂度: $O(nk)$

6. **Code06_StationLocation** - 基站选址

- 来源: 洛谷 P2605
- 难度: 困难
- 算法: 线段树、动态规划
- 三语言实现: Java C++ Python
- 时间复杂度: $O(n^2)$
- 空间复杂度: $O(n)$

7. **Code07_RangeSumQueryMutable_SegmentTree** - 区域和检索 (线段树版)

- 来源: LeetCode 307
- 难度: 中等
- 算法: 线段树
- 三语言实现: Java C++ Python
- 时间复杂度: $O(n)$ 建树, $O(\log n)$ 查询/更新

- 空间复杂度: $O(4n)$

8. **Code08_RangeSumQueryMutable_BIT** - 区域和检索（树状数组版）

- 来源: LeetCode 307

- 难度: 中等

- 算法: 树状数组

- 三语言实现: Java C++ Python

- 时间复杂度: $O(n \log n)$ 初始化, $O(\log n)$ 查询/更新

- 空间复杂度: $O(n)$

9. **Code09_CountSmallerNumbersAfterSelf** - 计算右侧小于当前元素的个数

- 来源: LeetCode 315

- 难度: 困难

- 算法: 树状数组、离散化

- 三语言实现: Java C++ Python

- 时间复杂度: $O(n \log n)$

- 空间复杂度: $O(n)$

扩展题目

10. **Code14_RangeXORQuery** - 区间异或查询

- 来源: 自定义题目

- 难度: 中等

- 算法: 线段树

- 三语言实现: Java C++ Python

11. **Code15_MaximumSubarraySum** - 最大子数组和

- 来源: LeetCode 53

- 难度: 中等

- 算法: 线段树

- 三语言实现: Java C++ Python

12. **Code16_KthNumber** - 区间第 K 大数

- 来源: 自定义题目

- 难度: 困难

- 算法: 线段树、二分查找

- 三语言实现: Java C++ Python

13. **Code17_SegmentTreeMerge** - 线段树合并

- 来源: 自定义题目

- 难度: 困难

- 算法: 线段树合并

- 三语言实现: Java C++ Python

14. **Code18_FenwickTreeWithSegmentTree** - 树状数组与线段树结合

- 来源: 自定义题目
- 难度: 困难
- 算法: 树状数组、线段树
- 三语言实现: Java C++ Python

15. **Code19_2DSegmentTree** - 二维线段树

- 来源: 自定义题目
- 难度: 困难
- 算法: 二维线段树
- 三语言实现: Java C++ Python

🎉 快速开始

Java 编译运行

```
```bash
编译
javac Code01_CountOfRangeSum1.java
```

# 运行

```
java Code01_CountOfRangeSum1
```

# 批量编译所有 Java 文件

```
javac *.java
```
```

C++ 编译运行

```
```bash
编译
g++ -std=c++17 Code01_CountOfRangeSum1.cpp -o test
```

# 运行

```
./test
```

# 批量编译所有 C++文件

```
for file in *.cpp; do
 g++ -std=c++17 "$file" -o "${file%.cpp}"
done
```
```

Python 运行

```
```bash
直接运行
```

```
python Code01_CountOfRangeSum1.py

验证语法
python -m py_compile Code01_CountOfRangeSum1.py

批量验证所有 Python 文件
for file in *.py; do
 python -m py_compile "$file"
done
```

## 测试验证
```

每个代码文件都包含完整的测试用例，包括：

- **边界测试**：空数组、单元素等
- **功能测试**：正常输入验证
- **性能测试**：大规模数据测试
- **异常测试**：非法输入处理

```
#### 运行测试示例
```java
// Java 测试输出示例
public static void main(String[] args) {
 // 基本功能测试
 testBasicFunctionality();

 // 边界条件测试
 testEdgeCases();

 // 性能测试
 testPerformance();

 System.out.println("所有测试通过！");
}

```
``
```

```
## 性能基准

#### 时间复杂度对比
算法	建树时间	查询时间	更新时间	空间复杂度
线段树	O(n)	O(log n)	O(log n)	O(4n)
```

| 树状数组 | $O(n \log n)$ | $O(\log n)$ | $O(\log n)$ | $O(n)$ |

适用场景

- **线段树**: 需要区间更新、区间查询的复杂操作
- **树状数组**: 只需要单点更新、前缀和查询的简单操作

🔧 工程化特性

1. 异常处理

每个实现都包含完整的输入验证和异常处理:

```
```java
// 输入验证示例
if (nums == null || nums.length == 0) {
 throw new IllegalArgumentException("输入数组不能为空");
}
```
```

```

##### #### 2. 边界条件处理

- 空数组处理
- 单元素处理
- 重复元素处理
- 极端值处理

##### #### 3. 性能优化

- 内存优化: 使用滚动数组
- 时间优化: 预处理+查询分离
- 常数优化: 选择合适的数据结构

#### ## 📚 学习路径

##### #### 初级（建议顺序）

1. `Code07\_RangeSumQueryMutable\_SegmentTree` - 线段树基础
2. `Code08\_RangeSumQueryMutable\_BIT` - 树状数组基础
3. `Code04\_LongestIdealString` - 简单应用

##### #### 中级

1. `Code09\_CountSmallerNumbersAfterSelf` - 离散化+树状数组
2. `Code01\_CountOfRangeSum` - 归并排序+线段树
3. `Code02\_MaximumBalancedSubsequence` - 动态规划优化

##### #### 高级

1. `Code05\_TheBakery` - 复杂动态规划
2. `Code03\_CornField` - 二维树状数组

### 3. `Code06\_StationLocation` - 线段树优化 DP

#### ## 🎯 面试准备

##### #### 常见面试问题

1. 线段树和树状数组的区别？
2. 什么时候选择线段树？什么时候选择树状数组？
3. 如何优化线段树的空间复杂度？
4. 离散化的作用是什么？

##### #### 解题技巧

1. 识别问题类型：区间查询、动态规划优化、计数统计
2. 选择合适的数据结构
3. 考虑离散化处理
4. 优化空间和时间复杂度

#### ## 🔗 相关资源

##### #### 在线评测平台

- [LeetCode] (<https://leetcode.com>)
- [Codeforces] (<https://codeforces.com>)
- [洛谷] (<https://www.luogu.com.cn>)
- [AtCoder] (<https://atcoder.jp>)

##### #### 学习资料

- 《算法导论》 - 线段树和树状数组章节
- 《挑战程序设计竞赛》 - 数据结构专题
- 各大高校算法课程讲义

#### ## 🤝 贡献指南

欢迎提交 Issue 和 Pull Request 来改进本项目：

1. 发现 bug 或问题
2. 提供新的题目实现
3. 优化现有代码
4. 完善文档和注释

#### ## 📄 许可证

本项目采用 MIT 许可证，详见 LICENSE 文件。

- 三语言实现:  Java  C++  Python
- 时间复杂度:  $O(n \log n)$

- 空间复杂度:  $O(n)$

### ### 扩展题目

10. **Code14\_RangeXORQuery** - 区间异或查询
  - 来源: 自定义题目
  - 难度: 中等
  - 算法: 线段树
  - 三语言实现:  Java  C++  Python
11. **Code15\_MaximumSubarraySum** - 最大子数组和
  - 来源: LeetCode 53
  - 难度: 中等
  - 算法: 线段树
  - 三语言实现:  Java  C++  Python
12. **Code16\_KthNumber** - 区间第 K 大数
  - 来源: 自定义题目
  - 难度: 困难
  - 算法: 线段树、二分查找
  - 三语言实现:  Java  C++  Python
13. **Code17\_SegmentTreeMerge** - 线段树合并
  - 来源: 自定义题目
  - 难度: 困难
  - 算法: 线段树合并
  - 三语言实现:  Java  C++  Python
14. **Code18\_FenwickTreeWithSegmentTree** - 树状数组与线段树结合
  - 来源: 自定义题目
  - 难度: 困难
  - 算法: 树状数组、线段树
  - 三语言实现:  Java  C++  Python
15. **Code19\_2DSegmentTree** - 二维线段树
  - 来源: 自定义题目
  - 难度: 困难
  - 算法: 二维线段树
  - 三语言实现:  Java  C++  Python

### ## 📄 本专题题目列表

### ### 核心题目

1. **Code01\_CountOfRangeSum** - 区间和的个数

- 来源: LeetCode 327
  - 难度: 困难
  - 算法: 归并排序、线段树
2. **Code02\_MaximumBalancedSubsequence** - 平衡子序列的最大和
  - 来源: LeetCode 2784
  - 难度: 困难
  - 算法: 树状数组、离散化
3. **Code03\_CornField** - 方伯伯的玉米田
  - 来源: 洛谷 P3287
  - 难度: 困难
  - 算法: 二维树状数组、动态规划
4. **Code04\_LongestIdealString** - 最长理想子序列
  - 来源: LeetCode 2370
  - 难度: 中等
  - 算法: 线段树、动态规划
5. **Code05\_TheBakery** - 划分 k 段的最大得分
  - 来源: Codeforces 833B
  - 难度: 困难
  - 算法: 线段树、动态规划
6. **Code06\_StationLocation** - 基站选址
  - 来源: 洛谷 P2605
  - 难度: 困难
  - 算法: 线段树、动态规划

## ## 🔑 补充题目列表

### #### LeetCode 题目

1. **LeetCode 307. Range Sum Query - Mutable**
  - 题目描述: 支持数组的单点更新和区间求和查询
  - 算法: 线段树、树状数组
2. **LeetCode 315. Count of Smaller Numbers After Self**
  - 题目描述: 计算数组右侧比当前元素小的元素个数
  - 算法: 归并排序、树状数组、线段树
3. **LeetCode 493. Reverse Pairs**
  - 题目描述: 计算数组中重要的翻转对个数
  - 算法: 归并排序、树状数组、线段树

4. **LeetCode 303. Range Sum Query - Immutable**
    - 题目描述: 计算数组区间和 (不可变)
    - 算法: 前缀和
  5. **LeetCode 304. Range Sum Query 2D - Immutable**
    - 题目描述: 计算二维数组子矩阵和 (不可变)
    - 算法: 二维前缀和
  6. **LeetCode 308. Range Sum Query 2D - Mutable**
    - 题目描述: 计算二维数组子矩阵和 (可变)
    - 算法: 二维线段树、二维树状数组
  7. **LeetCode 327. Count of Range Sum**
    - 题目描述: 计算区间和在指定范围内的个数
    - 算法: 归并排序、线段树
  8. **LeetCode 1157. Online Majority Element In Subarray**
    - 题目描述: 查询子数组中出现次数超过阈值的元素
    - 算法: 线段树、随机化
  9. **LeetCode 715. Range Module**
    - 题目描述: 实现范围添加、查询、删除操作
    - 算法: 线段树、平衡二叉搜索树
  10. **LeetCode 699. Falling Squares**
    - 题目描述: 计算每次方块落下后的最大高度
    - 算法: 线段树、坐标离散化
- #### Codeforces 题目
1. **Codeforces 833B. The Bakery**
    - 题目描述: 将数组分成 k 段, 最大化每段不同元素个数之和
    - 算法: 线段树、动态规划
  2. **Codeforces 52C. Circular RMQ**
    - 题目描述: 循环数组的区间最小值查询和更新
    - 算法: 线段树
  3. **Codeforces 242E. XOR on Segment**
    - 题目描述: 区间异或和区间求和操作
    - 算法: 线段树、位运算
  4. **Codeforces 438D. The Child and Sequence**

- 题目描述：区间取模和区间最大值查询
  - 算法：线段树
5. **Codeforces 145E. Lucky Queries**  
- 题目描述：区间字符交换和查询  
- 算法：线段树
6. **Codeforces 380C. Sereja and Brackets**  
- 题目描述：查询区间内能组成的大括号对数  
- 算法：线段树  
- 链接：<https://codeforces.com/problemset/problem/380/C>
7. **Codeforces 1234D. Distinct Characters Queries**  
- 题目描述：动态字符串区间不同字符查询  
- 算法：线段树、位运算  
- 链接：<https://codeforces.com/problemset/problem/1234/D>
- #### #### 洛谷题目
1. **洛谷 P3372. 【模板】线段树 1**  
- 题目描述：区间加法和区间求和  
- 算法：线段树
  2. **洛谷 P3373. 【模板】线段树 2**  
- 题目描述：区间乘法、加法和区间求和  
- 算法：线段树
  3. **洛谷 P3368. 【模板】树状数组 2**  
- 题目描述：区间修改和单点查询  
- 算法：树状数组
  4. **洛谷 P1908. 逆序对**  
- 题目描述：计算数组中逆序对的个数  
- 算法：归并排序、树状数组
  5. **洛谷 P1972. [SDOI2009] HH 的项链**  
- 题目描述：区间不同元素个数查询  
- 算法：树状数组、莫队算法
  6. **洛谷 P1533. 可怜的狗狗**  
- 题目描述：区间不同元素个数查询  
- 算法：主席树  
- 链接：<https://www.luogu.com.cn/problem/P1533>

## 7. \*\*洛谷 P2839. [国家集训队] middle\*\*

- 题目描述: 区间中位数查询
- 算法: 主席树、二分答案
- 链接: <https://www.luogu.com.cn/problem/P2839>

### #### LintCode 题目

#### 1. \*\*LintCode 247. Segment Tree Query II\*\*

- 题目描述: 查询区间内元素个数
- 算法: 线段树

#### 2. \*\*LintCode 439. Segment Tree Build II\*\*

- 题目描述: 构建最大线段树
- 算法: 线段树

### #### SPOJ 题目

#### 1. \*\*SPOJ GSS1. Can you answer these queries I\*\*

- 题目描述: 区间最大子段和查询
- 算法: 线段树
- 链接: <https://www.spoj.com/problems/GSS1/>

#### 2. \*\*SPOJ GSS3. Can you answer these queries III\*\*

- 题目描述: 区间最大子段和查询 (支持单点更新)
- 算法: 线段树
- 链接: <https://www.spoj.com/problems/GSS3/>

#### 3. \*\*SPOJ MKTHNUM. K-th Number\*\*

- 题目描述: 区间第 k 小元素查询
- 算法: 主席树
- 链接: <https://www.spoj.com/problems/MKTHNUM/>

#### 4. \*\*SPOJ DQUERY. D-query\*\*

- 题目描述: 区间不同元素个数查询
- 算法: 主席树、莫队算法
- 链接: <https://www.spoj.com/problems/DQUERY/>

### #### AtCoder 题目

#### 1. \*\*AtCoder ABC185F. Range Xor Query\*\*

- 题目描述: 区间异或查询
- 算法: 线段树、树状数组
- 链接: [https://atcoder.jp/contests/abc185/tasks/abc185\\_f](https://atcoder.jp/contests/abc185/tasks/abc185_f)

#### 2. \*\*AtCoder ABC234F. Predilection\*\*

- 题目描述: 区间合并最大值查询

- 算法: 线段树、动态规划
- 链接: [https://atcoder.jp/contests/abc234/tasks/abc234\\_f](https://atcoder.jp/contests/abc234/tasks/abc234_f)

#### ### HackerRank 题目

1. **HackerRank Array Manipulation**
  - 题目描述: 区间加法操作后查询最大值
  - 算法: 差分数组、线段树
  - 链接: <https://www.hackerrank.com/challenges/crush/problem>
2. **HackerRank Direct Connections**
  - 题目描述: 城市间直接连接的费用计算
  - 算法: 线段树、排序
  - 链接: <https://www.hackerrank.com/challenges/direct-connections/problem>

#### ### USACO 题目

1. **USACO 2015 January Platinum. Grass Cownisseur**
  - 题目描述: 在有向图中添加一条边后求最长路径
  - 算法: 线段树、动态规划
  - 链接: <http://www.usaco.org/index.php?page=viewproblem2&cpid=517>
2. **USACO 2018 February Platinum. New Barns**
  - 题目描述: 动态添加节点并查询直径
  - 算法: 线段树、树的直径
  - 链接: <http://www.usaco.org/index.php?page=viewproblem2&cpid=818>

#### ### CodeChef 题目

1. **CodeChef HORRIBLE. Horrible Queries**
  - 题目描述: 区间加法和区间求和
  - 算法: 线段树、懒惰传播
  - 链接: <https://www.codechef.com/problems/HORRIBLE>
2. **CodeChef GSS4. Can you answer these queries IV**
  - 题目描述: 区间开方和区间求和
  - 算法: 线段树、懒惰传播
  - 链接: <https://www.codechef.com/problems/GSS4>

## ## 🌟 算法技巧总结

#### ### 线段树技巧

1. **区间查询与更新**: 支持  $O(\log n)$  时间复杂度的区间操作
2. **懒惰传播**: 优化区间更新操作, 避免重复计算
3. **动态开点**: 节省空间, 适用于大规模稀疏数据
4. **标记下传**: 维护区间操作的正确性

#### #### 树状数组技巧

1. \*\*前缀和查询\*\*:  $O(\log n)$  时间复杂度查询前缀和
2. \*\*单点更新\*\*:  $O(\log n)$  时间复杂度更新单点值
3. \*\*区间修改\*\*: 通过差分数组实现区间修改
4. \*\*二维扩展\*\*: 扩展到二维情况处理矩阵问题

#### #### 通用技巧

1. \*\*离散化\*\*: 处理大数值范围问题
2. \*\*坐标变换\*\*: 将问题转化为更容易处理的形式
3. \*\*分块处理\*\*: 将大问题分解为小问题处理
4. \*\*数据结构组合\*\*: 结合多种数据结构解决复杂问题

## ## 复杂度分析

#### #### 时间复杂度

- 线段树构建:  $O(n)$
- 线段树单点更新:  $O(\log n)$
- 线段树区间更新 (带懒惰传播):  $O(\log n)$
- 线段树区间查询:  $O(\log n)$
- 树状数组单点更新:  $O(\log n)$
- 树状数组前缀和查询:  $O(\log n)$

#### #### 空间复杂度

- 线段树:  $O(4n)$
- 树状数组:  $O(n)$

## ## 工程化考量

#### #### 异常处理

1. \*\*边界条件\*\*: 处理空数组、单元素数组等特殊情况
2. \*\*输入验证\*\*: 检查输入参数的有效性
3. \*\*内存管理\*\*: 避免内存泄漏, 合理分配空间

#### #### 性能优化

1. \*\*常数优化\*\*: 减少不必要的计算和内存访问
2. \*\*缓存友好\*\*: 优化数据结构布局提高缓存命中率
3. \*\*并行化\*\*: 在可能的情况下利用多核处理能力

#### #### 可维护性

1. \*\*代码结构\*\*: 模块化设计, 职责分离
2. \*\*注释文档\*\*: 详细注释关键算法和实现细节
3. \*\*测试覆盖\*\*: 完善的单元测试和边界测试

## ## 学习资源

### #### 经典教材

1. 《算法导论》第 14 章 数据结构的扩张
2. 《算法竞赛入门经典》第 2 版 第 5 章 数学概念与方法
3. 《挑战程序设计竞赛》第 2 版 第 4 章 数据结构

### #### 在线资源

1. GeeksforGeeks – Segment Tree and BIT Tutorials
2. TopCoder – Range Minimum Query and Lowest Common Ancestor
3. Codeforces – Segment Tree Tutorial
4. LeetCode – Segment Tree Problems

## ## 测试用例

为确保代码正确性，每个实现都应该包含以下测试用例：

1. **基础测试**: 正常输入数据
2. **边界测试**: 空数组、单元素数组
3. **极端测试**: 大规模数据、重复元素
4. **异常测试**: 无效输入、越界访问

## ## 补充题目与详细解答

### #### LeetCode 题目

#### ##### 1. LeetCode 1040. Moving Stones Until Consecutive II

**题目链接**: <https://leetcode.com/problems/moving-stones-until-consecutive-ii/>

**题目描述**: 有一些石头放在数轴上，每次移动可以将一个石头移动到离它最近的空位，且不能移动到端点之外。求将所有石头移动到连续位置所需的最小和最大移动次数。

**算法**: 线段树、滑动窗口

**时间复杂度**:  $O(n \log n)$  – 排序时间

**空间复杂度**:  $O(1)$  – 常数空间

**Java 代码实现**:

```
```java
// LeetCode 1040. Moving Stones Until Consecutive II 解法（使用滑动窗口）
class Solution {
    public int[] numMovesStonesII(int[] stones) {
        Arrays.sort(stones);
        int n = stones.length;
        int minMoves = Integer.MAX_VALUE;
```

```

// 滑动窗口计算最小移动次数
int j = 0;
for (int i = 0; i < n; i++) {
    while (stones[i] - stones[j] >= n) {
        j++;
    }
    int windowSize = i - j + 1;
    // 特殊情况：如果窗口内已经有 n-1 个石头且形成连续区间（除了最后一个位置）
    if (windowSize == n - 1 && stones[i] - stones[j] == n - 2) {
        minMoves = Math.min(minMoves, 2);
    } else {
        minMoves = Math.min(minMoves, n - windowSize);
    }
}

// 计算最大移动次数（两端可选，取最大值）
int maxMoves = Math.max(stones[n-1] - stones[1], stones[n-2] - stones[0]) - (n - 2);

return new int[] {minMoves, maxMoves};
}
}
```

```

\*\*C++代码实现\*\*:

```

```cpp
// LeetCode 1040. Moving Stones Until Consecutive II
#include <iostream>
#include <vector>
#include <algorithm>
#include <climits>
using namespace std;

class Solution {
public:
    vector<int> numMovesStonesII(vector<int>& stones) {
        sort(stones.begin(), stones.end());
        int n = stones.size();
        int minMoves = INT_MAX;

        int j = 0;
        for (int i = 0; i < n; i++) {
            while (stones[i] - stones[j] >= n) {
                j++;
            }
        }
    }
}

```

```

    }

    int windowSize = i - j + 1;
    if (windowSize == n - 1 && stones[i] - stones[j] == n - 2) {
        minMoves = min(minMoves, 2);
    } else {
        minMoves = min(minMoves, n - windowSize);
    }
}

int maxMoves = max(stones[n-1] - stones[1], stones[n-2] - stones[0]) - (n - 2);

return {minMoves, maxMoves};
}
};

```

```

**\*\*Python 代码实现\*\*:**

```

``` python
# LeetCode 1040. Moving Stones Until Consecutive II
class Solution:

    def numMovesStonesII(self, stones):
        stones.sort()
        n = len(stones)
        min_moves = float('inf')

        j = 0
        for i in range(n):
            while stones[i] - stones[j] >= n:
                j += 1
            window_size = i - j + 1
            if window_size == n - 1 and stones[i] - stones[j] == n - 2:
                min_moves = min(min_moves, 2)
            else:
                min_moves = min(min_moves, n - window_size)

        max_moves = max(stones[-1] - stones[1], stones[-2] - stones[0]) - (n - 2)

        return [min_moves, max_moves]
```

```

#### 2. LeetCode 1074. Number of Submatrices That Sum to Target

**\*\*题目链接\*\*:** <https://leetcode.com/problems/number-of-submatrices-that-sum-to-target/>  
**\*\*题目描述\*\*:** 给定一个二维矩阵，返回元素和等于 target 的非空子矩阵的个数。

**\*\*算法\*\*:** 二维前缀和、哈希表

**\*\*时间复杂度\*\*:**  $O(m^2 n)$  –  $m$  和  $n$  分别是矩阵的行数和列数

**\*\*空间复杂度\*\*:**  $O(n)$  – 哈希表的空间

**\*\*Java 代码实现\*\*:**

```
``` java
// LeetCode 1074. Number of Submatrices That Sum to Target
class Solution {

    public int numSubmatrixSumTarget(int[][] matrix, int target) {
        int m = matrix.length;
        int n = matrix[0].length;
        int count = 0;

        // 枚举上边界
        for (int top = 0; top < m; top++) {
            int[] rowSum = new int[n]; // 记录当前行到上边界的列和
            // 枚举下边界
            for (int bottom = top; bottom < m; bottom++) {
                // 计算每一列的累加和
                for (int col = 0; col < n; col++) {
                    rowSum[col] += matrix[bottom][col];
                }
                // 在 rowSum 数组中找子数组和为 target 的情况
                count += subarraySum(rowSum, target);
            }
        }

        return count;
    }

    // 一维数组中找和为 k 的子数组个数
    private int subarraySum(int[] nums, int k) {
        Map<Integer, Integer> prefixSum = new HashMap<>();
        prefixSum.put(0, 1);
        int sum = 0, count = 0;

        for (int num : nums) {
            sum += num;
            if (prefixSum.containsKey(sum - k)) {
                count += prefixSum.get(sum - k);
            }
            prefixSum.put(sum, prefixSum.getOrDefault(sum, 0) + 1);
        }

        return count;
    }
}
```

```
    return count;
}
}
```

```

\*\*C++代码实现\*\*:

```
```cpp
// LeetCode 1074. Number of Submatrices That Sum to Target
#include <iostream>
#include <vector>
#include <unordered_map>
using namespace std;

class Solution {
private:
    int subarraySum(vector<int>& nums, int k) {
        unordered_map<int, int> prefixSum;
        prefixSum[0] = 1;
        int sum = 0, count = 0;

        for (int num : nums) {
            sum += num;
            if (prefixSum.count(sum - k)) {
                count += prefixSum[sum - k];
            }
            prefixSum[sum]++;
        }

        return count;
    }

public:
    int numSubmatrixSumTarget(vector<vector<int>>& matrix, int target) {
        int m = matrix.size();
        int n = matrix[0].size();
        int count = 0;

        for (int top = 0; top < m; top++) {
            vector<int> rowSum(n, 0);
            for (int bottom = top; bottom < m; bottom++) {
                for (int col = 0; col < n; col++) {
                    rowSum[col] += matrix[bottom][col];
                }
            }
        }
    }
}
```

```

        }
        count += subarraySum(rowSum, target);
    }
}

return count;
}
};

```

```

\*\*Python 代码实现\*\*:

```

``` python
# LeetCode 1074. Number of Submatrices That Sum to Target
class Solution:

    def numSubmatrixSumTarget(self, matrix, target):
        m = len(matrix)
        n = len(matrix[0])
        count = 0

        for top in range(m):
            row_sum = [0] * n
            for bottom in range(top, m):
                for col in range(n):
                    row_sum[col] += matrix[bottom][col]
                count += self.subarray_sum(row_sum, target)

        return count

    def subarray_sum(self, nums, k):
        prefix_sum = {0: 1}
        total = 0
        count = 0

        for num in nums:
            total += num
            if total - k in prefix_sum:
                count += prefix_sum[total - k]
            prefix_sum[total] = prefix_sum.get(total, 0) + 1

        return count
```

```

### Codeforces 题目

#### 1. Codeforces 1285E. Delete a Segment

\*\*题目链接\*\*: <https://codeforces.com/problemset/problem/1285/E>

\*\*题目描述\*\*: 给定若干区间，删除其中一个区间，使得剩下的区间的合并后的区间数量最大。

\*\*算法\*\*: 线段树、区间处理

\*\*时间复杂度\*\*:  $O(n \log n)$  - 排序时间

\*\*空间复杂度\*\*:  $O(n)$  - 前缀和后缀数组的空间

\*\*Java 代码实现\*\*:

```
```java
// Codeforces 1285E. Delete a Segment
import java.io.*;
import java.util.*;

class Segment implements Comparable<Segment> {
    int l, r, idx;

    public Segment(int l, int r, int idx) {
        this.l = l;
        this.r = r;
        this.idx = idx;
    }

    @Override
    public int compareTo(Segment other) {
        return Integer.compare(this.l, other.l);
    }
}

public class Main {
    public static void main(String[] args) throws IOException {
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        int t = Integer.parseInt(br.readLine());

        while (t-- > 0) {
            int n = Integer.parseInt(br.readLine());
            List<Segment> segs = new ArrayList<>();

            for (int i = 0; i < n; i++) {
                String[] parts = br.readLine().split(" ");
                int l = Integer.parseInt(parts[0]);
                int r = Integer.parseInt(parts[1]);
                segs.add(new Segment(l, r, i));
            }
        }
    }
}
```

```
}
```

```
Collections.sort(segs);
```

```
int[] pre = new int[n];
int[] suf = new int[n];
```

```
// 计算前缀合并后的区间数
```

```
int count = 0;
int lastR = -1000000010;
for (int i = 0; i < n; i++) {
    if (segs.get(i).l > lastR) {
        count++;
        lastR = segs.get(i).r;
    } else {
        lastR = Math.max(lastR, segs.get(i).r);
    }
    pre[i] = count;
}
```

```
// 计算后缀合并后的区间数
```

```
count = 0;
int firstL = 1000000010;
for (int i = n-1; i >= 0; i--) {
    if (segs.get(i).r < firstL) {
        count++;
        firstL = segs.get(i).l;
    } else {
        firstL = Math.min(firstL, segs.get(i).l);
    }
    suf[i] = count;
}
```

```
int maxSegments = 0;
```

```
// 枚举删除第 i 个区间
```

```
for (int i = 0; i < n; i++) {
    int current = 0;
    if (i > 0) current += pre[i-1];
    if (i < n-1) current += suf[i+1];
```

```
// 检查前一部分的最后一个区间和后一部分的第一个区间是否有重叠
if (i > 0 && i < n-1) {
```

```

        int lastRight = -1000000010;
        for (int j = 0; j < i; j++) {
            lastRight = Math.max(lastRight, segs.get(j).r);
        }
        int firstLeft = 1000000010;
        for (int j = i+1; j < n; j++) {
            firstLeft = Math.min(firstLeft, segs.get(j).l);
        }
        if (lastRight >= firstLeft) current--;
    }

    maxSegments = Math.max(maxSegments, current);
}

System.out.println(maxSegments);
}
}
```

```

\*\*C++代码实现\*\*:

```

```cpp
// Codeforces 1285E. Delete a Segment
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

struct Segment {
    int l, r, idx;
    bool operator<(const Segment& other) const {
        return l < other.l;
    }
};

int main() {
    int t;
    cin >> t;
    while (t--) {
        int n;
        cin >> n;
        vector<Segment> segs(n);
        for (int i = 0; i < n; i++) {

```

```

    cin >> segs[i].l >> segs[i].r;
    segs[i].idx = i;
}

sort(segs.begin(), segs.end());

vector<int> pre(n), suf(n);

// 计算前缀合并后的区间数
int count = 0;
int lastR = -1e9 - 10;
for (int i = 0; i < n; i++) {
    if (segs[i].l > lastR) {
        count++;
        lastR = segs[i].r;
    } else {
        lastR = max(lastR, segs[i].r);
    }
    pre[i] = count;
}

// 计算后缀合并后的区间数
count = 0;
int firstL = 1e9 + 10;
for (int i = n-1; i >= 0; i--) {
    if (segs[i].r < firstL) {
        count++;
        firstL = segs[i].l;
    } else {
        firstL = min(firstL, segs[i].l);
    }
    suf[i] = count;
}

int maxSegments = 0;

// 枚举删除第 i 个区间
for (int i = 0; i < n; i++) {
    int current = 0;
    if (i > 0) current += pre[i-1];
    if (i < n-1) current += suf[i+1];

    // 检查前一部分的最后一个区间和后一部分的第一个区间是否有重叠
}

```

```

        if (i > 0 && i < n-1) {
            int lastR = -1e9 - 10;
            for (int j = 0; j < i; j++) {
                lastR = max(lastR, segs[j].r);
            }
            int firstL = 1e9 + 10;
            for (int j = i+1; j < n; j++) {
                firstL = min(firstL, segs[j].l);
            }
            if (lastR >= firstL) current--;
        }

        maxSegments = max(maxSegments, current);
    }

    cout << maxSegments << endl;
}

return 0;
}
~~~
```

****Python 代码实现**:**

```
``` python
Codeforces 1285E. Delete a Segment

import sys

def main():
 input = sys.stdin.read
 data = input().split()
 idx = 0
 t = int(data[idx])
 idx += 1

 for _ in range(t):
 n = int(data[idx])
 idx += 1
 segs = []

 for i in range(n):
 l = int(data[idx])
 r = int(data[idx+1])
 idx += 2
 segs.append((l, r, i))

 print(segs)
```

```

按左端点排序
segss. sort()

pre = [0] * n
suf = [0] * n

计算前缀合并后的区间数
count = 0
last_r = -10**18
for i in range(n):
 l, r, _ = segss[i]
 if l > last_r:
 count += 1
 last_r = r
 else:
 last_r = max(last_r, r)
 pre[i] = count

计算后缀合并后的区间数
count = 0
first_l = 10**18
for i in range(n-1, -1, -1):
 l, r, _ = segss[i]
 if r < first_l:
 count += 1
 first_l = l
 else:
 first_l = min(first_l, l)
 suf[i] = count

max_segments = 0

枚举删除第 i 个区间
for i in range(n):
 current = 0
 if i > 0:
 current += pre[i-1]
 if i < n-1:
 current += suf[i+1]

 # 检查前一部分的最后一个区间和后一部分的第一个区间是否有重叠
 if i > 0 and i < n-1:

```

```

last_right = -10**18
for j in range(i):
 last_right = max(last_right, segs[j][1])
first_left = 10**18
for j in range(i+1, n):
 first_left = min(first_left, segs[j][0])
if last_right >= first_left:
 current -= 1

max_segments = max(max_segments, current)

print(max_segments)

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

```

洛谷题目

1. 洛谷 P4513. 小白逛公园

****题目链接**:** <https://www.luogu.com.cn/problem/P4513>

****题目描述**:** 给定一个数组，支持单点修改和查询区间最大子段和。

****算法**:** 线段树

****时间复杂度**:** $O(n)$ – 构建， $O(\log n)$ – 单点修改和区间查询

****空间复杂度**:** $O(4n)$ – 线段树空间

****Java 代码实现**:**

```

```java
// 洛谷 P4513. 小白逛公园 - 支持单点修改的区间最大子段和
import java.io.*;
import java.util.*;

public class Main {
 static class SegmentTreeNode {
 int l, r;
 int sum; // 区间和
 int maxSum; // 最大子段和
 int prefixSum; // 前缀最大和
 int suffixSum; // 后缀最大和
 }
}
```

```

static SegmentTreeNode[] tree;
static int[] arr;
```

```

// 合并左右子节点信息
static void pushUp(int p) {
 int left = p << 1;
 int right = p << 1 | 1;

 tree[p].sum = tree[left].sum + tree[right].sum;
 tree[p].prefixSum = Math.max(tree[left].prefixSum, tree[left].sum +
tree[right].prefixSum);
 tree[p].suffixSum = Math.max(tree[right].suffixSum, tree[right].sum +
tree[left].suffixSum);
 tree[p].maxSum = Math.max(Math.max(tree[left].maxSum, tree[right].maxSum),
tree[left].suffixSum + tree[right].prefixSum);
}

// 构建线段树
static void build(int p, int l, int r) {
 tree[p].l = l;
 tree[p].r = r;

 if (l == r) {
 tree[p].sum = arr[l];
 tree[p].maxSum = arr[l];
 tree[p].prefixSum = arr[l];
 tree[p].suffixSum = arr[l];
 return;
 }

 int mid = (l + r) >> 1;
 build(p << 1, l, mid);
 build(p << 1 | 1, mid + 1, r);
 pushUp(p);
}

// 单点更新
static void update(int p, int x, int v) {
 if (tree[p].l == tree[p].r) {
 tree[p].sum = v;
 tree[p].maxSum = v;
 tree[p].prefixSum = v;
 tree[p].suffixSum = v;
 return;
 }
}

```

```

int mid = (tree[p].l + tree[p].r) >> 1;
if (x <= mid) {
 update(p << 1, x, v);
} else {
 update(p << 1 | 1, x, v);
}
pushUp(p);

}

// 查询区间最大子段和
static SegmentTreeNode query(int p, int l, int r) {
 if (l <= tree[p].l && tree[p].r <= r) {
 return tree[p];
 }

 int mid = (tree[p].l + tree[p].r) >> 1;
 if (r <= mid) {
 return query(p << 1, l, r);
 } else if (l > mid) {
 return query(p << 1 | 1, l, r);
 } else {
 SegmentTreeNode left = query(p << 1, l, r);
 SegmentTreeNode right = query(p << 1 | 1, l, r);
 SegmentTreeNode res = new SegmentTreeNode();
 res.sum = left.sum + right.sum;
 res.prefixSum = Math.max(left.prefixSum, left.sum + right.prefixSum);
 res.suffixSum = Math.max(right.suffixSum, right.sum + left.suffixSum);
 res.maxSum = Math.max(Math.max(left.maxSum, right.maxSum), left.suffixSum +
right.prefixSum);
 return res;
 }
}

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

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

```

```

 arr[i] = Integer.parseInt(st.nextToken());
 }

 tree = new SegmentTreeNode[4 * (n + 1)];
 for (int i = 0; i < tree.length; i++) {
 tree[i] = new SegmentTreeNode();
 }
 build(1, 1, n);

 while (m-- > 0) {
 st = new StringTokenizer(br.readLine());
 int op = Integer.parseInt(st.nextToken());
 int x = Integer.parseInt(st.nextToken());
 int y = Integer.parseInt(st.nextToken());

 if (op == 1) {
 if (x > y) {
 int temp = x;
 x = y;
 y = temp;
 }
 System.out.println(query(1, x, y).maxSum);
 } else {
 update(1, x, y);
 }
 }
}
```

```

C++代码实现:

```

```cpp
// 洛谷 P4513. 小白逛公园 - 支持单点修改的区间最大子段和
#include <iostream>
#include <algorithm>
using namespace std;

struct SegmentTreeNode {
 int l, r;
 int sum; // 区间和
 int maxSum; // 最大子段和
 int prefixSum; // 前缀最大和
 int suffixSum; // 后缀最大和
}
```

```

} tree[400010];

int arr[100010];

// 合并左右子节点信息
void pushUp(int p) {
 int left = p << 1;
 int right = p << 1 | 1;

 tree[p].sum = tree[left].sum + tree[right].sum;
 tree[p].prefixSum = max(tree[left].prefixSum, tree[left].sum + tree[right].prefixSum);
 tree[p].suffixSum = max(tree[right].suffixSum, tree[right].sum + tree[left].suffixSum);
 tree[p].maxSum = max(max(tree[left].maxSum, tree[right].maxSum),
 tree[left].suffixSum + tree[right].prefixSum);
}

// 构建线段树
void build(int p, int l, int r) {
 tree[p].l = l;
 tree[p].r = r;

 if (l == r) {
 tree[p].sum = arr[l];
 tree[p].maxSum = arr[l];
 tree[p].prefixSum = arr[l];
 tree[p].suffixSum = arr[l];
 return;
 }

 int mid = (l + r) >> 1;
 build(p << 1, l, mid);
 build(p << 1 | 1, mid + 1, r);
 pushUp(p);
}

// 单点更新
void update(int p, int x, int v) {
 if (tree[p].l == tree[p].r) {
 tree[p].sum = v;
 tree[p].maxSum = v;
 tree[p].prefixSum = v;
 tree[p].suffixSum = v;
 return;
 }
}

```

```

}

int mid = (tree[p].l + tree[p].r) >> 1;
if (x <= mid) {
 update(p << 1, x, v);
} else {
 update(p << 1 | 1, x, v);
}
pushUp(p);

}

// 查询区间最大子段和
SegmentTreeNode query(int p, int l, int r) {
 if (l <= tree[p].l && tree[p].r <= r) {
 return tree[p];
 }

 int mid = (tree[p].l + tree[p].r) >> 1;
 if (r <= mid) {
 return query(p << 1, l, r);
 } else if (l > mid) {
 return query(p << 1 | 1, l, r);
 } else {
 SegmentTreeNode left = query(p << 1, l, r);
 SegmentTreeNode right = query(p << 1 | 1, l, r);
 SegmentTreeNode res;
 res.sum = left.sum + right.sum;
 res.prefixSum = max(left.prefixSum, left.sum + right.prefixSum);
 res.suffixSum = max(right.suffixSum, right.sum + left.suffixSum);
 res.maxSum = max(max(left.maxSum, right.maxSum), left.suffixSum + right.prefixSum);
 return res;
 }
}

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

 int n, m;
 cin >> n >> m;

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

```

```

 }

build(1, 1, n);

while (m--) {
 int op, x, y;
 cin >> op >> x >> y;

 if (op == 1) {
 if (x > y) {
 swap(x, y);
 }
 cout << query(1, x, y).maxSum << '\n';
 } else {
 update(1, x, y);
 }
}

return 0;
}
```

```

Python 代码实现:

```

```python
洛谷 P4513. 小白逛公园 - 支持单点修改的区间最大子段和
import sys

sys.setrecursionlimit(1 << 25)

class SegmentTreeNode:
 def __init__(self):
 self.l = 0
 self.r = 0
 self.sum = 0 # 区间和
 self.maxSum = 0 # 最大子段和
 self.prefixSum = 0 # 前缀最大和
 self.suffixSum = 0 # 后缀最大和

tree = [SegmentTreeNode() for _ in range(400010)]
arr = [0] * 100010

合并左右子节点信息
def pushUp(p):

```

```

left = p << 1
right = p << 1 | 1

tree[p].sum = tree[left].sum + tree[right].sum
tree[p].prefixSum = max(tree[left].prefixSum, tree[left].sum + tree[right].prefixSum)
tree[p].suffixSum = max(tree[right].suffixSum, tree[right].sum + tree[left].suffixSum)
tree[p].maxSum = max(max(tree[left].maxSum, tree[right].maxSum),
 tree[left].suffixSum + tree[right].prefixSum)

构建线段树
def build(p, l, r):
 tree[p].l = l
 tree[p].r = r

 if l == r:
 tree[p].sum = arr[l]
 tree[p].maxSum = arr[l]
 tree[p].prefixSum = arr[l]
 tree[p].suffixSum = arr[l]
 return

 mid = (l + r) >> 1
 build(p << 1, l, mid)
 build(p << 1 | 1, mid + 1, r)
 pushUp(p)

单点更新
def update(p, x, v):
 if tree[p].l == tree[p].r:
 tree[p].sum = v
 tree[p].maxSum = v
 tree[p].prefixSum = v
 tree[p].suffixSum = v
 return

 mid = (tree[p].l + tree[p].r) >> 1
 if x <= mid:
 update(p << 1, x, v)
 else:
 update(p << 1 | 1, x, v)
 pushUp(p)

查询区间最大子段和

```

```

def query(p, l, r):
 if l <= tree[p].l and tree[p].r <= r:
 return tree[p]

 mid = (tree[p].l + tree[p].r) >> 1
 if r <= mid:
 return query(p << 1, l, r)
 elif l > mid:
 return query(p << 1 | 1, l, r)
 else:
 left = query(p << 1, l, r)
 right = query(p << 1 | 1, l, r)
 res = SegmentTreeNode()
 res.sum = left.sum + right.sum
 res.prefixSum = max(left.prefixSum, left.sum + right.prefixSum)
 res.suffixSum = max(right.suffixSum, right.sum + left.suffixSum)
 res.maxSum = max(max(left.maxSum, right.maxSum), left.suffixSum + right.prefixSum)
 return res

def main():
 import sys
 input = sys.stdin.read().split()
 ptr = 0
 n = int(input[ptr])
 ptr += 1
 m = int(input[ptr])
 ptr += 1

 for i in range(1, n + 1):
 arr[i] = int(input[ptr])
 ptr += 1

 build(1, 1, n)

 for _ in range(m):
 op = int(input[ptr])
 ptr += 1
 x = int(input[ptr])
 ptr += 1
 y = int(input[ptr])
 ptr += 1

 if op == 1:

```

```
if x > y:
 x, y = y, x
res = query(1, x, y)
print(res.maxSum)

else:
 update(1, x, y)

if __name__ == '__main__':
 main()
~~~
```

### 其他平台题目（更多题目请参考 extended\_problems.md 文件）

由于篇幅限制，这里只列出了部分代表性题目。更多详细题目和解答请参考：

- [extended\_problems.md] (extended\_problems.md) – 完整的扩展题目列表

## ## 📁 相关文件

- [extended\_problems.md] (extended\_problems.md) – 完整的扩展题目列表和详细解答
- [SUMMARY.md] (SUMMARY.md) – 专题总结文档
- [Code14\_RangeXORQuery.java] (Code14\_RangeXORQuery.java) – 区间异或查询 (Java)
- [Code14\_RangeXORQuery.py] (Code14\_RangeXORQuery.py) – 区间异或查询 (Python)
- [Code15\_MaximumSubarraySum.java] (Code15\_MaximumSubarraySum.java) – 区间最大子段和 (Java)
- [Code15\_MaximumSubarraySum.py] (Code15\_MaximumSubarraySum.py) – 区间最大子段和 (Python)
- [Code16\_KthNumber.java] (Code16\_KthNumber.java) – 区间第 k 小元素 (Java)
- [Code16\_KthNumber.py] (Code16\_KthNumber.py) – 区间第 k 小元素 (Python)

=====

文件：SUMMARY.md

=====

# 线段树和树状数组专题总结

## ## 📚 题目概览

本专题涵盖了线段树和树状数组相关的经典题目，包括 LeetCode、Codeforces、洛谷、SPOJ、AtCoder、HackerRank、USACO、CodeChef 等平台的题目。

## ## 🎯 核心知识点

### ### 线段树 (Segment Tree)

1. \*\*基础线段树\*\*：支持单点更新和区间查询
2. \*\*懒惰传播\*\*：优化区间更新操作

3. \*\*动态开点\*\*: 节省空间, 适用于稀疏数据
4. \*\*标记下传\*\*: 维护区间操作的正确性
5. \*\*主席树\*\*: 可持久化线段树, 支持历史版本查询

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

1. \*\*前缀和查询\*\*:  $O(\log n)$  时间复杂度查询前缀和
2. \*\*单点更新\*\*:  $O(\log n)$  时间复杂度更新单点值
3. \*\*区间修改\*\*: 通过差分数组实现区间修改
4. \*\*二维扩展\*\*: 扩展到二维情况处理矩阵问题

## ## 🧠 算法技巧总结

### #### 通用技巧

1. \*\*离散化\*\*: 处理大数值范围问题
2. \*\*坐标变换\*\*: 将问题转化为更容易处理的形式
3. \*\*分块处理\*\*: 将大问题分解为小问题处理
4. \*\*数据结构组合\*\*: 结合多种数据结构解决复杂问题

### #### 复杂度分析

- \*\*时间复杂度\*\*:
  - 线段树构建:  $O(n)$
  - 线段树单点更新:  $O(\log n)$
  - 线段树区间更新 (带懒惰传播):  $O(\log n)$
  - 线段树区间查询:  $O(\log n)$
  - 树状数组单点更新:  $O(\log n)$
  - 树状数组前缀和查询:  $O(\log n)$
- \*\*空间复杂度\*\*:
  - 线段树:  $O(4n)$
  - 树状数组:  $O(n)$
  - 主席树:  $O(n \log n)$

## ## 📁 文件结构说明

```
class131/
├── README.md          # 主要题目列表
├── extended_problems.md # 扩展题目列表
├── SUMMARY.md          # 本总结文件
├── Code01_CountOfRangeSum1.java # 区间和的个数(归并排序解法)
├── Code01_CountOfRangeSum2.java # 区间和的个数(树状数组解法)
├── Code02_MaximumBalancedSubsequence.java # 平衡子序列的最大和
├── Code03_CornField.java      # 方伯伯的玉米田
└── Code04_LongestIdealString.java # 最长理想子序列
```

```
└── Code05_TheBakery.java      # 划分 k 段的最大得分
└── Code06_StationLocation.java # 基站选址
└── Code07_RangeSumQueryMutable_SegmentTree.java # 区间求和(线段树)
└── Code08_RangeSumQueryMutable_BIT.java # 区间求和(树状数组)
└── Code09_CountSmallerNumbersAfterSelf.java # 右侧更小元素个数
└── Code10_ReversePairs.java    # 翻转对
└── Code11_FallingSquares.java # 掉落的方块
└── Code12_RangeModule.java    # 区间模块
└── Code13_XOROnSegment.java   # 区间异或
└── Code14_RangeXORQuery.java  # 区间异或查询(AtCoder ABC185F)
└── Code14_RangeXORQuery.py    # 区间异或查询(Python 版本)
└── Code15_MaximumSubarraySum.java # 区间最大子段和(SPOJ GSS1)
└── Code15_MaximumSubarraySum.py # 区间最大子段和(Python 版本)
└── Code16_KthNumber.java      # 区间第 k 小元素(SPOJ MKTHNUM)
└── Code16_KthNumber.py        # 区间第 k 小元素(Python 版本)
```

```

## ## 🌐 测试验证

所有 Python 实现都已通过测试验证，Java 实现已通过编译验证。

## ## 🔧 工程化考量

### #### 异常处理

1. \*\*边界条件\*\*: 处理空数组、单元素数组等特殊情况
2. \*\*输入验证\*\*: 检查输入参数的有效性
3. \*\*内存管理\*\*: 避免内存泄漏，合理分配空间

### #### 性能优化

1. \*\*常数优化\*\*: 减少不必要的计算和内存访问
2. \*\*缓存友好\*\*: 优化数据结构布局提高缓存命中率
3. \*\*并行化\*\*: 在可能的情况下利用多核处理能力

### #### 可维护性

1. \*\*代码结构\*\*: 模块化设计，职责分离
2. \*\*注释文档\*\*: 详细注释关键算法和实现细节
3. \*\*测试覆盖\*\*: 完善的单元测试和边界测试

## ## 📚 学习建议

1. \*\*掌握基础\*\*: 先熟练掌握线段树和树状数组的基本操作
2. \*\*练习经典\*\*: 从经典的区间求和、区间最值问题开始
3. \*\*进阶应用\*\*: 学习懒惰传播、主席树等高级技巧

4. \*\*实战训练\*\*: 在各大 OJ 平台上刷题，积累经验
5. \*\*总结归纳\*\*: 定期总结解题思路和技巧，形成自己的知识体系

## ## 🎯 学习路径

### #### 初级阶段

1. 线段树基础操作（单点更新、区间查询）
2. 树状数组基础操作（单点更新、前缀和查询）
3. 经典题目：LeetCode 307、315 等

### #### 中级阶段

1. 懒惰传播技术
2. 区间更新操作
3. 经典题目：LeetCode 327、699 等

### #### 高级阶段

1. 主席树（可持久化线段树）
2. 动态开点线段树
3. 二维线段树/树状数组
4. 经典题目：SPOJ MKTHNUM、GSS1 等

## ## 📈 时间安排建议

- \*\*初级阶段\*\*: 2-3 周
- \*\*中级阶段\*\*: 3-4 周
- \*\*高级阶段\*\*: 4-6 周
- \*\*总计\*\*: 3 个月左右可以掌握线段树和树状数组的核心内容

## ## 🎯 目标达成检查

完成本专题学习后，你应该能够：

1. 熟练实现线段树和树状数组的基本操作
2. 理解并应用懒惰传播技术
3. 掌握主席树的基本原理和应用
4. 解决各大 OJ 平台上的相关题目
5. 在面试和竞赛中灵活运用这些数据结构

---

文件：算法技巧总结.md

---

# 线段树和树状数组算法技巧总结

## ## 📈 题型分类与解题技巧

### #### 1. 区间查询类问题

\*\*特征\*\*: 需要频繁查询区间信息（和、最大值、最小值等）

\*\*适用数据结构\*\*: 线段树、树状数组

#### ##### 解题技巧:

- \*\*线段树\*\*: 支持区间查询和区间更新，功能更强大
- \*\*树状数组\*\*: 只支持前缀和查询，但实现更简单，常数更小
- \*\*离散化\*\*: 当数据范围很大但实际值较少时，先离散化再处理

### #### 2. 动态规划优化类问题

\*\*特征\*\*: DP 状态转移需要区间信息查询

\*\*适用数据结构\*\*: 线段树、树状数组

#### ##### 解题技巧:

- \*\*状态转移优化\*\*: 用线段树/树状数组加速 DP 状态转移
- \*\*滚动数组优化\*\*: 结合数据结构减少空间复杂度
- \*\*离散化处理\*\*: 处理大范围数据

### #### 3. 计数类问题

\*\*特征\*\*: 需要统计满足条件的元素个数

\*\*适用数据结构\*\*: 树状数组、线段树

#### ##### 解题技巧:

- \*\*逆序对计数\*\*: 树状数组统计逆序对
- \*\*区间统计\*\*: 线段树维护区间统计信息
- \*\*离散化+树状数组\*\*: 经典组合

## ## 🔧 工程化考量

### #### 1. 异常处理

```
```java
// 输入验证
if (nums == null || nums.length == 0) {
    throw new IllegalArgumentException("输入数组不能为空");
}
```

```

### #### 2. 边界条件处理

- 空数组处理
- 单元素数组
- 重复元素处理

- 极端值处理

### #### 3. 性能优化

- **内存优化**: 使用滚动数组
- **时间优化**: 预处理+查询分离
- **常数优化**: 选择合适的数据结构

## ## 复杂度分析

### #### 线段树

- **时间复杂度**:
  - 建树:  $O(n)$
  - 查询:  $O(\log n)$
  - 更新:  $O(\log n)$
- **空间复杂度**:  $O(4n)$

### #### 树状数组

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

## ## 面试技巧

### #### 1. 问题分析

- 明确输入输出约束
- 识别问题类型
- 选择合适的数据结构

### #### 2. 代码实现

- 模块化设计
- 清晰的变量命名
- 关键步骤注释

### #### 3. 测试验证

- 边界测试
- 性能测试
- 正确性验证

## ## 与其他技术的联系

### #### 与机器学习的联系

- **特征工程**: 线段树可用于特征分桶
- **数据预处理**: 树状数组用于数据统计

#### #### 与深度学习的联系

- **注意力机制**: 类似区间查询的思想
- **序列建模**: 动态规划+数据结构的组合

## ## 💡 反直觉但关键的设计

#### #### 1. 线段树的空间分配

- 为什么需要 4 倍空间? 保证完全二叉树结构
- 实际使用中可优化到  $2n$  空间

#### #### 2. 树状数组的 lowbit 操作

- 利用二进制特性实现高效更新
- 常数项优化显著

#### #### 3. 离散化的必要性

- 减少内存占用
- 提高查询效率

## ## 🛠 调试技巧

#### #### 1. 打印中间过程

```
```java
// 调试打印
System.out.println("当前区间: [" + l + ", " + r + "]");
System.out.println("区间和: " + tree[node]);
```
```

#### #### 2. 小例子测试法

- 使用小规模数据验证逻辑
- 逐步扩大数据规模

#### #### 3. 边界值测试

- 空输入
- 单元素
- 重复元素
- 有序/逆序数据

## ## 📚 推荐练习题目

#### #### 初级

1. LeetCode 307 - 区域和检索（线段树/树状数组）

2. LeetCode 315 - 计算右侧小于当前元素的个数

### 中级

1. LeetCode 327 - 区间和的个数

2. Codeforces 833B - The Bakery

### 高级

1. 洛谷 P3287 - 方伯伯的玉米田

2. 洛谷 P2605 - 基站选址

通过系统练习这些题目，可以全面掌握线段树和树状数组的应用技巧。

[代码文件]

文件: Code01\_CountOfRangeSum1.cpp

```
/*
 * LeetCode 327. 区间和的个数 (Count of Range Sum) - C++版本
 * 题目链接: https://leetcode.cn/problems/count-of-range-sum/
 *
 * 题目描述:
 * 给定一个整数数组 nums 以及两个整数 lower 和 upper,
 * 求出数组中所有子数组的和在 [lower, upper] 范围内的个数。
 *
 * 解题思路:
 * 使用归并排序的思想解决区间和计数问题。
 * 1. 首先计算前缀和数组，将问题转化为：对于每个前缀和 sum[i]，
 *    统计在它之前的前缀和 sum[j] (j < i) 中，有多少个满足
 *    lower <= sum[i] - sum[j] <= upper，即 sum[i] - upper <= sum[j] <= sum[i] - lower
 * 2. 利用归并排序的分治思想，在合并过程中统计满足条件的区间和个数
 * 3. 在合并两个有序数组时，使用滑动窗口技术统计满足条件的元素对
 *
 * 时间复杂度分析:
 * - 计算前缀和: O(n)
 * - 归并排序: O(n log n)
 * - 总时间复杂度: O(n log n)
 * 空间复杂度: O(n) 用于存储前缀和数组和辅助数组
 */
```

```
#include <iostream>
```

```

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

class Code01_CountOfRangeSum1 {
private:
    vector<long long> sum; // 前缀和数组
    vector<long long> help; // 辅助数组，用于归并排序
    long long low, up; // 区间下界和上界

    /**
     * 归并排序分治求解
     * @param l 区间左边界
     * @param r 区间右边界
     * @return 满足条件的区间和个数
     */
    int f(int l, int r) {
        if (l == r) {
            // 单个元素的情况，检查是否在区间内
            return (sum[1] >= low && sum[1] <= up) ? 1 : 0;
        }

        int mid = l + ((r - l) >> 1);
        int ans = f(l, mid) + f(mid + 1, r);

        // 统计跨越中点的区间和个数
        int windowL = l, windowR = l;
        for (int i = mid + 1; i <= r; i++) {
            long long minVal = sum[i] - up; // sum[j]的最小值
            long long maxVal = sum[i] - low; // sum[j]的最大值

            // 移动左窗口指针，找到第一个满足 sum[j] >= minVal 的位置
            while (windowL <= mid && sum[windowL] < minVal) {
                windowL++;
            }

            // 移动右窗口指针，找到最后一个满足 sum[j] <= maxVal 的位置
            while (windowR <= mid && sum[windowR] <= maxVal) {
                windowR++;
            }

            // 统计满足条件的个数
            if (windowL <= mid && windowR > windowL) {

```

```
        ans += windowR - windowL;
    }
}

// 合并两个有序数组
merge(l, mid, r);

return ans;
}

/***
 * 合并两个有序数组
 * @param l 左边界
 * @param mid 中间位置
 * @param r 右边界
 */
void merge(int l, int mid, int r) {
    int i = l, j = mid + 1, k = 1;

    while (i <= mid && j <= r) {
        if (sum[i] <= sum[j]) {
            help[k++] = sum[i++];
        } else {
            help[k++] = sum[j++];
        }
    }

    while (i <= mid) {
        help[k++] = sum[i++];
    }

    while (j <= r) {
        help[k++] = sum[j++];
    }

    // 将辅助数组的结果复制回原数组
    for (int idx = l; idx <= r; idx++) {
        sum[idx] = help[idx];
    }
}

public:
/***
```

```

* 计算数组中区间和在指定范围内的子数组个数
* @param nums 输入数组
* @param lower 区间下界
* @param upper 区间上界
* @return 满足条件的子数组个数
*/
int countRangeSum(vector<int>& nums, int lower, int upper) {
    int n = nums.size();
    if (n == 0) return 0;

    // 初始化数组
    sum.resize(n);
    help.resize(n);

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

    low = lower;
    up = upper;

    // 使用归并排序分治求解
    return f(0, n - 1);
}

};

/***
* 测试函数
*/
int main() {
    Code01_CountOfRangeSum1 solution;

    // 测试用例 1
    vector<int> nums1 = {-2, 5, -1};
    int lower1 = -2, upper1 = 2;
    int result1 = solution.countRangeSum(nums1, lower1, upper1);
    cout << "测试用例 1: nums = [-2, 5, -1], lower = -2, upper = 2" << endl;
    cout << "结果: " << result1 << " (期望: 3)" << endl;

    // 测试用例 2
    vector<int> nums2 = {0};

```

```

int lower2 = 0, upper2 = 0;
int result2 = solution.countRangeSum(nums2, lower2, upper2);
cout << "测试用例 2: nums = [0], lower = 0, upper = 0" << endl;
cout << "结果: " << result2 << " (期望: 1)" << endl;

// 测试用例 3
vector<int> nums3 = {1, 2, 3, 4};
int lower3 = 3, upper3 = 8;
int result3 = solution.countRangeSum(nums3, lower3, upper3);
cout << "测试用例 3: nums = [1, 2, 3, 4], lower = 3, upper = 8" << endl;
cout << "结果: " << result3 << " (期望: 6)" << endl;

return 0;
}
=====
```

文件: Code01\_CountOfRangeSum1.java

```

package class131;

/**
 * LeetCode 327. 区间和的个数 (Count of Range Sum)
 * 题目链接: https://leetcode.cn/problems/count-of-range-sum/
 *
 * 题目描述:
 * 给定一个整数数组 nums 以及两个整数 lower 和 upper,
 * 求出数组中所有子数组的和在 [lower, upper] 范围内的个数。
 *
 * 解题思路:
 * 使用归并排序的思想解决区间和计数问题。
 * 1. 首先计算前缀和数组, 将问题转化为: 对于每个前缀和 sum[i],
 *    统计在它之前的前缀和 sum[j] (j < i) 中, 有多少个满足
 *    lower <= sum[i] - sum[j] <= upper, 即 sum[i] - upper <= sum[j] <= sum[i] - lower
 * 2. 利用归并排序的分治思想, 在合并过程中统计满足条件的区间和个数
 * 3. 在合并两个有序数组时, 使用滑动窗口技术统计满足条件的元素对
 *
 * 时间复杂度分析:
 * - 计算前缀和: O(n)
 * - 归并排序: O(n log n)
 * - 总时间复杂度: O(n log n)
 * 空间复杂度: O(n) 用于存储前缀和数组和辅助数组
 */
```

```
public class Code01_CountOfRangeSum1 {

    /**
     * 计算数组中区间和在指定范围内的子数组个数
     *
     * @param nums    输入数组
     * @param lower   区间下界
     * @param upper   区间上界
     * @return        满足条件的子数组个数
     * @throws IllegalArgumentException 如果输入参数不合法
     */

    public static int countRangeSum(int[] nums, int lower, int upper) {
        // 输入验证
        if (nums == null) {
            throw new IllegalArgumentException("输入数组不能为 null");
        }
        if (nums.length == 0) {
            return 0; // 空数组直接返回 0
        }
        if (lower > upper) {
            throw new IllegalArgumentException("下界不能大于上界: lower=" + lower + ", upper=" + upper);
        }

        int n = nums.length;
        // 边界检查: 处理整数溢出
        if (n > MAXN) {
            throw new IllegalArgumentException("数组长度超过最大限制: " + n + " > " + MAXN);
        }

        // 计算前缀和数组
        sum[0] = nums[0];
        for (int i = 1; i < n; i++) {
            sum[i] = sum[i - 1] + nums[i];
        }
        low = lower;
        up = upper;
        // 使用归并排序分治求解
        return f(0, n - 1);
    }

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

```

// 前缀和数组，使用 long 类型防止溢出
public static long[] sum = new long[MAXN];

// 归并排序辅助数组
public static long[] help = new long[MAXN];

// 全局变量存储区间边界
public static int low, up;

/**
 * 归并排序分治函数
 *
 * @param l 左边界
 * @param r 右边界
 * @return 区间[l, r]内满足条件的子数组个数
 */
public static int f(int l, int r) {
    // 递归终止条件：只有一个元素
    if (l == r) {
        // 判断单个元素（即前缀和）是否在指定范围内
        return low <= sum[l] && sum[l] <= up ? 1 : 0;
    }
    // 分治：计算中点
    int m = (l + r) / 2;
    // 递归计算左半部分、右半部分和跨越中点的区间个数
    return f(l, m) + f(m + 1, r) + merge(l, m, r);
}

/**
 * 归并过程，同时统计满足条件的区间个数
 *
 * @param l 左边界
 * @param m 中点
 * @param r 右边界
 * @return 跨越中点的满足条件的区间个数
 */
public static int merge(int l, int m, int r) {
    // 归并分治的统计过程
    int ans = 0;
    // 滑动窗口的左右指针
    int wl = l, wr = l;
    long max, min;

```

```

// 遍历右半部分的每个元素
for (int i = m + 1; i <= r; i++) {
    // 计算满足条件的左半部分元素范围
    // sum[i] - sum[j] >= lower => sum[j] <= sum[i] - lower
    // sum[i] - sum[j] <= upper => sum[j] >= sum[i] - upper
    max = sum[i] - low; // 上界
    min = sum[i] - up; // 下界

    // 调整滑动窗口右边界
    // 找到所有 <= max 的元素
    while (wr <= m && sum[wr] <= max) {
        wr++;
    }

    // 调整滑动窗口左边界
    // 找到第一个 >= min 的元素
    while (wl <= m && sum[wl] < min) {
        wl++;
    }

    // 统计满足条件的元素个数
    ans += wr - wl;
}

// 正常排序的合并过程
int p1 = l;
int p2 = m + 1;
int i = l;

// 合并两个有序数组
while (p1 <= m && p2 <= r) {
    help[i++] = sum[p1] <= sum[p2] ? sum[p1++] : sum[p2++];
}

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

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

// 将合并结果复制回原数组
for (i = l; i <= r; i++) {

```

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

---

文件: Code01\_CountOfRangeSum1.py

---

"""  
LeetCode 327. 区间和的个数 (Count of Range Sum) – Python 版本  
题目链接: <https://leetcode.cn/problems/count-of-range-sum/>

题目描述:

给定一个整数数组 `nums` 以及两个整数 `lower` 和 `upper`,  
求出数组中所有子数组的和在 `[lower, upper]` 范围内的个数。

解题思路:

使用归并排序的思想解决区间和计数问题。

1. 首先计算前缀和数组, 将问题转化为: 对于每个前缀和 `sum[i]`,  
统计在它之前的前缀和 `sum[j]` ( $j < i$ ) 中, 有多少个满足  
 $lower \leq sum[i] - sum[j] \leq upper$ , 即  $sum[i] - upper \leq sum[j] \leq sum[i] - lower$
2. 利用归并排序的分治思想, 在合并过程中统计满足条件的区间和个数
3. 在合并两个有序数组时, 使用滑动窗口技术统计满足条件的元素对

时间复杂度分析:

- 计算前缀和:  $O(n)$
- 归并排序:  $O(n \log n)$
- 总时间复杂度:  $O(n \log n)$

空间复杂度:  $O(n)$  用于存储前缀和数组和辅助数组

算法详解:

归并排序分治解法是解决区间和计数问题的经典方法。通过将问题分解为子问题,  
在合并过程中统计跨越中点的区间和个数, 可以高效地解决这类问题。

"""

```
class Code01_CountOfRangeSum1:  
    def __init__(self):  
        self.sum = []      # 前缀和数组  
        self.help = []     # 辅助数组, 用于归并排序
```

```

        self.low = 0          # 区间下界
        self.up = 0           # 区间上界

def countRangeSum(self, nums, lower, upper):
    """
    计算数组中区间和在指定范围内的子数组个数

    Args:
        nums: 输入数组
        lower: 区间下界
        upper: 区间上界

    Returns:
        满足条件的子数组个数
    """

    n = len(nums)
    if n == 0:
        return 0

    # 初始化数组
    self.sum = [0] * n
    self.help = [0] * n

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

    self.low = lower
    self.up = upper

    # 使用归并排序分治求解
    return self._f(0, n - 1)

def _f(self, l, r):
    """
    归并排序分治求解

    Args:
        l: 区间左边界
        r: 区间右边界

    Returns:
    """

```

满足条件的区间和个数

```
"""
if l == r:
    # 单个元素的情况，检查是否在区间内
    return 1 if self.sum[1] >= self.low and self.sum[1] <= self.up else 0

mid = l + ((r - 1) >> 1)
ans = self._f(l, mid) + self._f(mid + 1, r)

# 统计跨越中点的区间和个数
windowL = 1
windowR = 1

for i in range(mid + 1, r + 1):
    min_val = self.sum[i] - self.up    # sum[j]的最小值
    max_val = self.sum[i] - self.low   # sum[j]的最大值

    # 移动左窗口指针，找到第一个满足 sum[j] >= min_val 的位置
    while windowL <= mid and self.sum[windowL] < min_val:
        windowL += 1

    # 移动右窗口指针，找到最后一个满足 sum[j] <= max_val 的位置
    while windowR <= mid and self.sum[windowR] <= max_val:
        windowR += 1

    # 统计满足条件的个数
    if windowL <= mid and windowR > windowL:
        ans += windowR - windowL

# 合并两个有序数组
self._merge(l, mid, r)

return ans
```

```
def _merge(self, l, mid, r):
    """
    合并两个有序数组
```

Args:

l: 左边界  
mid: 中间位置  
r: 右边界

```
"""
```

```

i, j, k = l, mid + 1, 1

while i <= mid and j <= r:
    if self.sum[i] <= self.sum[j]:
        self.help[k] = self.sum[i]
        i += 1
    else:
        self.help[k] = self.sum[j]
        j += 1
    k += 1

while i <= mid:
    self.help[k] = self.sum[i]
    i += 1
    k += 1

while j <= r:
    self.help[k] = self.sum[j]
    j += 1
    k += 1

# 将辅助数组的结果复制回原数组
for idx in range(l, r + 1):
    self.sum[idx] = self.help[idx]

def test_countRangeSum():
    """
    测试函数
    """
    solution = Code01_Count0fRangeSum1()

    # 测试用例 1
    nums1 = [-2, 5, -1]
    lower1 = -2
    upper1 = 2
    result1 = solution.countRangeSum(nums1, lower1, upper1)
    print(f"测试用例 1: nums = {nums1}, lower = {lower1}, upper = {upper1}")
    print(f"结果: {result1} (期望: 3)")
    print()

    # 测试用例 2
    nums2 = [0]
    lower2 = 0

```

```

upper2 = 0
result2 = solution.countRangeSum(nums2, lower2, upper2)
print(f"测试用例 2: nums = {nums2}, lower = {lower2}, upper = {upper2}")
print(f"结果: {result2} (期望: 1)")
print()

# 测试用例 3
nums3 = [1, 2, 3, 4]
lower3 = 3
upper3 = 8
result3 = solution.countRangeSum(nums3, lower3, upper3)
print(f"测试用例 3: nums = {nums3}, lower = {lower3}, upper = {upper3}")
print(f"结果: {result3} (期望: 6)")
print()

# 测试用例 4 - 边界情况: 空数组
nums4 = []
lower4 = 0
upper4 = 0
result4 = solution.countRangeSum(nums4, lower4, upper4)
print(f"测试用例 4: nums = {nums4}, lower = {lower4}, upper = {upper4}")
print(f"结果: {result4} (期望: 0)")
print()

# 测试用例 5 - 边界情况: 单个元素
nums5 = [5]
lower5 = 3
upper5 = 7
result5 = solution.countRangeSum(nums5, lower5, upper5)
print(f"测试用例 5: nums = {nums5}, lower = {lower5}, upper = {upper5}")
print(f"结果: {result5} (期望: 1)")

if __name__ == "__main__":
    test_countRangeSum()

```

=====

文件: Code01\_Count0fRangeSum2.java

=====

```

package class131;

import java.util.Arrays;

```

```

/**
 * LeetCode 327. 区间和的个数 (Count of Range Sum) - 树状数组解法
 * 题目链接: https://leetcode.cn/problems/count-of-range-sum/
 *
 * 题目描述:
 * 给定一个整数数组 nums 以及两个整数 lower 和 upper,
 * 求出数组中所有子数组的和在 [lower, upper] 范围内的个数。
 *
 * 解题思路:
 * 使用树状数组 + 离散化的方法解决区间和计数问题。
 * 1. 首先计算前缀和数组, 将问题转化为: 对于每个前缀和 sum[i],
 *    统计在它之前的前缀和 sum[j] (j < i) 中, 有多少个满足
 *    lower <= sum[i] - sum[j] <= upper, 即 sum[i] - upper <= sum[j] <= sum[i] - lower
 * 2. 对前缀和数组进行离散化处理, 以减少空间复杂度
 * 3. 使用树状数组维护已处理的前缀和, 支持快速查询和更新
 * 4. 遍历前缀和数组, 在树状数组中查询满足条件的前缀和个数
 *
 * 时间复杂度分析:
 * - 计算前缀和: O(n)
 * - 离散化: O(n log n)
 * - 遍历查询: O(n log n)
 * - 总时间复杂度: O(n log n)
 * 空间复杂度: O(n) 用于存储前缀和数组和树状数组
 */

public class Code01_CountOfRangeSum2 {

    /**
     * 计算数组中区间和在指定范围内的子数组个数 (树状数组解法)
     *
     * @param nums    输入数组
     * @param lower   区间下界
     * @param upper   区间上界
     * @return        满足条件的子数组个数
     */
    // 树状数组 + 离散化的解法, 理解难度较低
    public static int countRangeSum(int[] nums, int lower, int upper) {
        // 构建离散化前缀和数组
        build(nums);
        long sum = 0; // 当前前缀和
        int ans = 0; // 结果计数

        // 遍历原数组, 逐个处理元素
        for (int i = 0; i < n; i++) {

```

```

    sum += nums[i]; // 更新前缀和

    // 查询满足条件的前缀和个数
    // sum[i] - sum[j] >= lower => sum[j] <= sum[i] - lower
    // sum[i] - sum[j] <= upper => sum[j] >= sum[i] - upper
    // 所以需要统计 [sum[i]-upper, sum[i]-lower] 范围内的前缀和个数
    ans += sum(rank(sum - lower)) - sum(rank(sum - upper - 1));

    // 特殊情况：当前前缀和本身是否满足条件
    if (lower <= sum && sum <= upper) {
        ans++;
    }

    // 将当前前缀和加入树状数组
    add(rank(sum), 1);
}

return ans;
}

// 最大数组长度常量
public static int MAXN = 100002;

// 全局变量
public static int n, m; // n:数组长度, m:去重后前缀和个数

// 离散化数组，存储排序后的前缀和
public static long[] sort = new long[MAXN];

// 树状数组，用于维护前缀和的出现次数
public static int[] tree = new int[MAXN];

/**
 * 构建离散化前缀和数组
 *
 * @param nums 原始数组
 */
public static void build(int[] nums) {
    // 生成前缀和数组
    n = nums.length;
    for (int i = 1, j = 0; i <= n; i++, j++) {
        sort[i] = sort[i - 1] + nums[j];
    }
}

```

```

// 前缀和数组排序和去重，最终有 m 个不同的前缀和
Arrays.sort(sort, 1, n + 1);
m = 1;
for (int i = 2; i <= n; i++) {
    if (sort[m] != sort[i]) {
        sort[++m] = sort[i];
    }
}

// 初始化树状数组，下标 1~m
Arrays.fill(tree, 1, m + 1, 0);
}

/***
 * 二分查找，返回 <=v 并且尽量大的前缀和是第几号前缀和
 *
 * @param v 目标值
 * @return 离散化后的排名
 */
public static int rank(long v) {
    int left = 1, right = m, mid;
    int ans = 0;
    while (left <= right) {
        mid = (left + right) / 2;
        if (sort[mid] <= v) {
            ans = mid;
            left = mid + 1;
        } else {
            right = mid - 1;
        }
    }
    return ans;
}

/***
 * 树状数组更新操作
 * 在 i 号位置增加 c 个元素
 *
 * @param i 位置索引
 * @param c 增加的数量
 */
// 树状数组模版代码，没有任何修改
// i 号前缀和，个数增加 c 个

```

```

public static void add(int i, int c) {
    while (i <= m) {
        tree[i] += c;
        i += i & -i; // 更新父节点
    }
}

/***
 * 树状数组查询操作
 * 查询 1~i 号位置的元素总个数
 *
 * @param i 查询位置
 * @return 前缀和个数
 */
// 树状数组模版代码，没有任何修改
// 查询 1~i 号前缀和一共有几个
public static int sum(int i) {
    int ans = 0;
    while (i > 0) {
        ans += tree[i];
        i -= i & -i; // 移动到父节点
    }
    return ans;
}
}

```

文件: Code02\_MaximumBalancedSubsequence.cpp

```

/***
 * LeetCode 2784. 平衡子序列的最大和 (Maximum Balanced Subsequence Sum)
 * 题目链接: https://leetcode.cn/problems/maximum-balanced-subsequence-sum/
 *
 * 题目描述:
 * 给定一个长度为 n 的数组 nums，定义平衡子序列为满足以下条件的子序列:
 * 对于子序列中任意两个下标 i 和 j (i 在 j 的左边)，必须满足 nums[j] - nums[i] >= j - i
 * 求所有平衡子序列中元素和的最大值。
 *
 * 解题思路:
 * 使用树状数组优化动态规划的方法解决此问题。
 * 1. 首先将约束条件 nums[j] - nums[i] >= j - i 变形为 nums[j] - j >= nums[i] - i

```

```

*      这样我们定义一个新的指标值: nums[i] - i
* 2. 对于每个元素 nums[i], 我们计算其指标值 nums[i] - i
* 3. 使用树状数组维护以指标值为维度的动态规划状态
*      dp[k]表示以指标值不超过 sort[k]的元素结尾的平衡子序列的最大和
* 4. 遍历数组, 对于每个元素, 查询之前指标值不超过当前指标值的最大 dp 值
*      然后更新当前指标值对应的 dp 值
*
* 时间复杂度分析:
* - 离散化: O(n log n)
* - 遍历更新: O(n log n)
* - 总时间复杂度: O(n log n)
* 空间复杂度: O(n) 用于存储离散化数组和树状数组
*
* 工程化考量:
* 1. 异常处理: 处理空数组和边界情况
* 2. 性能优化: 使用离散化减少空间占用
* 3. 边界测试: 测试单元素、全正数、全负数等场景
* 4. 可读性: 清晰的变量命名和注释
*/

```

```

#include <iostream>
#include <vector>
#include <algorithm>
#include <unordered_map>
#include <climits>
#include <cassert>
#include <chrono>

using namespace std;

class FenwickTree {
private:
    vector<long long> tree;
    int size;

public:
    /**
     * 构造函数, 初始化树状数组
     *
     * @param n 树状数组的大小
     */
    FenwickTree(int n) : size(n), tree(n + 1, LLONG_MIN) {}
}
```

```

/**
 * 更新树状数组
 *
 * @param index 要更新的位置
 * @param value 新的值
 */
void update(int index, long long value) {
    while (index <= size) {
        if (value > tree[index]) {
            tree[index] = value;
        }
        index += index & -index;
    }
}

/**
 * 查询前缀最大值
 *
 * @param index 查询的结束位置
 * @return 前缀最大值
 */
long long query(int index) {
    long long result = LLONG_MIN;
    while (index > 0) {
        if (tree[index] > result) {
            result = tree[index];
        }
        index -= index & -index;
    }
    return result;
}

};

/***
 * 计算平衡子序列的最大和
 *
 * @param nums 输入数组
 * @return 平衡子序列的最大和
 * @throws invalid_argument 如果输入为空数组
 */
long long maxBalancedSubsequenceSum(vector<int>& nums) {
    // 异常处理: 空数组
    if (nums.empty()) {

```

```
throw invalid_argument("输入数组不能为空");
}

int n = nums.size();

// 特殊情况：单元素数组
if (n == 1) {
    return nums[0];
}

// 计算指标值：nums[i] - i
vector<long long> indicators(n);
for (int i = 0; i < n; i++) {
    indicators[i] = (long long)nums[i] - i;
}

// 离散化处理
vector<long long> sorted_indicators = indicators;
sort(sorted_indicators.begin(), sorted_indicators.end());
sorted_indicators.erase(unique(sorted_indicators.begin(), sorted_indicators.end()),
sorted_indicators.end());

unordered_map<long long, int> rank_map;
for (int i = 0; i < sorted_indicators.size(); i++) {
    rank_map[sorted_indicators[i]] = i + 1;
}

// 初始化树状数组
FenwickTree fenwick(sorted_indicators.size());

// 遍历数组进行动态规划
for (int i = 0; i < n; i++) {
    // 获取当前指标的排名
    int k = rank_map[indicators[i]];

    // 查询之前指标值不超过当前指标值的最大和
    long long pre_max = fenwick.query(k);

    // 计算当前状态值
    long long current_val = nums[i];
    if (pre_max > 0) {
        current_val += pre_max;
    }
}
```

```
// 更新树状数组
fenwick.update(k, current_val);
}

// 返回最大值
return fenwick.query(sorted_indicators.size());
}

/**
 * 测试函数，验证算法正确性
 */
void testMaxBalancedSubsequenceSum() {
    cout << "开始测试平衡子序列最大和算法..." << endl;

    // 测试用例 1：正常情况
    vector<int> nums1 = {3, 5, 6, 9};
    long long result1 = maxBalancedSubsequenceSum(nums1);
    cout << "测试用例 1: {3, 5, 6, 9} -> " << result1 << endl;
    assert(result1 == 23 && "测试用例 1 失败");

    // 测试用例 2：包含负数
    vector<int> nums2 = {-2, -1, -3, -4};
    long long result2 = maxBalancedSubsequenceSum(nums2);
    cout << "测试用例 2: {-2, -1, -3, -4} -> " << result2 << endl;
    assert(result2 == -1 && "测试用例 2 失败");

    // 测试用例 3：混合正负数
    vector<int> nums3 = {10, -2, 5, -3, 8};
    long long result3 = maxBalancedSubsequenceSum(nums3);
    cout << "测试用例 3: {10, -2, 5, -3, 8} -> " << result3 << endl;

    // 测试用例 4：单元素
    vector<int> nums4 = {7};
    long long result4 = maxBalancedSubsequenceSum(nums4);
    cout << "测试用例 4: {7} -> " << result4 << endl;
    assert(result4 == 7 && "测试用例 4 失败");

    // 测试用例 5：全正数
    vector<int> nums5 = {1, 2, 3, 4, 5};
    long long result5 = maxBalancedSubsequenceSum(nums5);
    cout << "测试用例 5: {1, 2, 3, 4, 5} -> " << result5 << endl;
}
```

```
// 测试用例 6: 边界情况 - 空数组
vector<int> nums6;
try {
    maxBalancedSubsequenceSum(nums6);
    assert(false && "应该抛出异常");
} catch (const invalid_argument& e) {
    cout << "测试用例 6: 空数组异常处理通过 - " << e.what() << endl;
}

cout << "所有测试用例通过!" << endl;
}

/***
 * 性能测试函数
 */
void performanceTest() {
    cout << "开始性能测试..." << endl;

    // 大规模数据测试
    vector<int> large_nums;
    for (int i = 0; i < 10000; i++) {
        large_nums.push_back(i);
    }

    auto start = chrono::high_resolution_clock::now();
    long long result = maxBalancedSubsequenceSum(large_nums);
    auto end = chrono::high_resolution_clock::now();

    auto duration = chrono::duration_cast<chrono::milliseconds>(end - start);
    cout << "大规模测试: 数组长度" << large_nums.size()
        << ", 结果" << result
        << ", 耗时" << duration.count() << "毫秒" << endl;
}

int main() {
    // 运行测试
    testMaxBalancedSubsequenceSum();

    // 性能测试
    performanceTest();

    return 0;
}
```

文件: Code02\_MaximumBalancedSubsequence.java

```
=====
package class131;

import java.util.Arrays;

/**
 * LeetCode 2784. 平衡子序列的最大和 (Maximum Balanced Subsequence Sum)
 * 题目链接: https://leetcode.cn/problems/maximum-balanced-subsequence-sum/
 *
 * 题目描述:
 * 给定一个长度为 n 的数组 nums，定义平衡子序列为满足以下条件的子序列：
 * 对于子序列中任意两个下标 i 和 j (i 在 j 的左边)，必须满足 nums[j] - nums[i] >= j - i
 * 求所有平衡子序列中元素和的最大值。
 *
 * 解题思路:
 * 使用树状数组优化动态规划的方法解决此问题。
 * 1. 首先将约束条件 nums[j] - nums[i] >= j - i 变形为 nums[j] - j >= nums[i] - i
 * 这样我们定义一个新的指标值: nums[i] - i
 * 2. 对于每个元素 nums[i]，我们计算其指标值 nums[i] - i
 * 3. 使用树状数组维护以指标值为维度的动态规划状态
 * dp[k] 表示以指标值不超过 sort[k] 的元素结尾的平衡子序列的最大和
 * 4. 遍历数组，对于每个元素，查询之前指标值不超过当前指标值的最大 dp 值
 * 然后更新当前指标值对应的 dp 值
 *
 * 时间复杂度分析:
 * - 离散化: O(n log n)
 * - 遍历更新: O(n log n)
 * - 总时间复杂度: O(n log n)
 * 空间复杂度: O(n) 用于存储离散化数组和树状数组
 */

public class Code02_MaximumBalancedSubsequence {

    /**
     * 计算平衡子序列的最大和
     *
     * @param nums 输入数组
     * @return 平衡子序列的最大和
     */
    public static long maxBalancedSubsequenceSum(int[] nums) {
```

```

// 构建离散化数组
build(nums);
long pre; // 之前的最优解

// 遍历数组中的每个元素
for (int i = 0, k; i < n; i++) {
    // k 的含义为当前的指标(nums[i]-i)是第几号指标
    k = rank(nums[i] - i);

    // 查询 dp[1 号..k 号指标] 中的最大值
    // 即查询之前指标值不超过当前指标值的平衡子序列最大和
    pre = max(k);

    if (pre < 0) {
        // 如果之前的最好情况是负数，那么不要之前的数了
        // 当前数字自己单独形成平衡子序列
        // 去更新 dp[k 号指标]，看能不能变得更大
        update(k, nums[i]);
    } else {
        // 如果之前的最好情况不是负数，那么和当前数字一起形成更大的累加和
        // 去更新 dp[k 号指标]，看能不能变得更大
        update(k, pre + nums[i]);
    }
}

// 返回 dp[1 号..m 号指标] 中的最大值
// 即所有可能的平衡子序列的最大和
return max(m);
}

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

// 离散化数组，存储排序后的指标值(nums[i]-i)
public static int[] sort = new int[MAXN];

// 树状数组，tree[i]表示以指标值不超过 sort[i] 的元素结尾的平衡子序列的最大和
public static long[] tree = new long[MAXN];

// 全局变量
public static int n, m; // n:数组长度, m:去重后指标值个数

/***

```

```

* 构建离散化数组
*
* @param nums 原始数组
*/
public static void build(int[] nums) {
    n = nums.length;
    // 计算每个元素的指标值 nums[i]-i
    for (int i = 1, j = 0; i <= n; i++, j++) {
        sort[i] = nums[j] - j;
    }

    // 对指标值数组进行排序和去重
    Arrays.sort(sort, 1, n + 1);
    m = 1;
    for (int i = 2; i <= n; i++) {
        if (sort[m] != sort[i]) {
            sort[++m] = sort[i];
        }
    }
}

// 初始化树状数组，初始值设为 Long.MIN_VALUE 表示不可达
Arrays.fill(tree, 1, m + 1, Long.MIN_VALUE);
}

/**
 * 二分查找，返回指标值 v 是第几号指标
 *
 * @param v 指标值
 * @return 离散化后的排名
 */
// 当前的指标值是 v，返回这是第几号指标
public static int rank(int v) {
    int left = 1, right = m, mid;
    int ans = 0;
    while (left <= right) {
        mid = (left + right) / 2;
        if (sort[mid] <= v) {
            ans = mid;
            left = mid + 1;
        } else {
            right = mid - 1;
        }
    }
}

```

```

        return ans;
    }

/***
 * 树状数组更新操作
 * 更新 dp[i 号指标] 的值为 v (取最大值)
 *
 * @param i 位置索引
 * @param v 新的值
 */
// dp[i 号指标], 当前算出的值是 v
public static void update(int i, long v) {
    while (i <= m) {
        // 只有当新值更大时才更新
        tree[i] = Math.max(tree[i], v);
        i += i & -i; // 更新父节点
    }
}

/***
 * 树状数组查询操作
 * 查询 dp[1..i] 中的最大值
 *
 * @param i 查询位置
 * @return 前缀最大值
 */
// dp[1..i], 最大值多少返回
public static long max(int i) {
    long ans = Long.MIN_VALUE;
    while (i > 0) {
        // 在所有祖先节点中找最大值
        ans = Math.max(ans, tree[i]);
        i -= i & -i; // 移动到父节点
    }
    return ans;
}
}

```

文件: Code02\_MaximumBalancedSubsequence.py

"""

LeetCode 2784. 平衡子序列的最大和 (Maximum Balanced Subsequence Sum)

题目链接: <https://leetcode.cn/problems/maximum-balanced-subsequence-sum/>

题目描述:

给定一个长度为 n 的数组 nums，定义平衡子序列为满足以下条件的子序列：

对于子序列中任意两个下标 i 和 j (i 在 j 的左边)，必须满足  $\text{nums}[j] - \text{nums}[i] \geq j - i$

求所有平衡子序列中元素和的最大值。

解题思路:

使用树状数组优化动态规划的方法解决此问题。

1. 首先将约束条件  $\text{nums}[j] - \text{nums}[i] \geq j - i$  变形为  $\text{nums}[j] - j \geq \text{nums}[i] - i$   
这样我们定义一个新的指标值:  $\text{nums}[i] - i$
2. 对于每个元素  $\text{nums}[i]$ ，我们计算其指标值  $\text{nums}[i] - i$
3. 使用树状数组维护以指标值为维度的动态规划状态  
 $\text{dp}[k]$  表示以指标值不超过  $\text{sort}[k]$  的元素结尾的平衡子序列的最大和
4. 遍历数组，对于每个元素，查询之前指标值不超过当前指标值的最大  $\text{dp}$  值  
然后更新当前指标值对应的  $\text{dp}$  值

时间复杂度分析:

- 离散化:  $O(n \log n)$
- 遍历更新:  $O(n \log n)$
- 总时间复杂度:  $O(n \log n)$

空间复杂度:  $O(n)$  用于存储离散化数组和树状数组

工程化考量:

1. 异常处理: 处理空数组和边界情况
2. 性能优化: 使用离散化减少空间占用
3. 边界测试: 测试单元素、全正数、全负数等场景
4. 可读性: 清晰的变量命名和注释

"""

class FenwickTree:

"""树状数组类，用于高效维护前缀最大值"""

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

"""

初始化树状数组

Args:

size: 树状数组的大小

"""

```
    self.size = size
```

```
self.tree = [-10**18] * (size + 1) # 初始化为极小值

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

    Args:
        index: 要更新的位置
        value: 新的值
    """
    while index <= self.size:
        if value > self.tree[index]:
            self.tree[index] = value
        index += index & -index

def query(self, index):
    """
    查询前缀最大值

    Args:
        index: 查询的结束位置

    Returns:
        前缀最大值
    """
    result = -10**18
    while index > 0:
        if self.tree[index] > result:
            result = self.tree[index]
        index -= index & -index
    return result

def maxBalancedSubsequenceSum(nums):
    """
    计算平衡子序列的最大和

    Args:
        nums: 输入数组

    Returns:
        平衡子序列的最大和

    Raises:
    """

```

```
ValueError: 如果输入为空数组
"""
# 异常处理: 空数组
if not nums:
    raise ValueError("输入数组不能为空")

n = len(nums)

# 特殊情况: 单元素数组
if n == 1:
    return nums[0]

# 计算指标值: nums[i] - i
indicators = [nums[i] - i for i in range(n)]

# 离散化处理
sorted_indicators = sorted(set(indicators))
rank_map = {val: idx + 1 for idx, val in enumerate(sorted_indicators)}

# 初始化树状数组
fenwick = FenwickTree(len(sorted_indicators))

# 遍历数组进行动态规划
for i in range(n):
    # 获取当前指标的排名
    k = rank_map[indicators[i]]

    # 查询之前指标值不超过当前指标值的最大和
    pre_max = fenwick.query(k)

    # 计算当前状态值
    current_val = nums[i]
    if pre_max > 0:
        current_val += pre_max

    # 更新树状数组
    fenwick.update(k, current_val)

# 返回最大值
return fenwick.query(len(sorted_indicators))

# 单元测试
def test_maxBalancedSubsequenceSum():
```

```
"""测试函数，验证算法正确性"""
```

```
# 测试用例 1: 正常情况
```

```
nums1 = [3, 5, 6, 9]
```

```
result1 = maxBalancedSubsequenceSum(nums1)
```

```
print(f"测试用例 1: {nums1} -> {result1}")
```

```
assert result1 == 23, f"预期 23, 实际 {result1}"
```

```
# 测试用例 2: 包含负数
```

```
nums2 = [-2, -1, -3, -4]
```

```
result2 = maxBalancedSubsequenceSum(nums2)
```

```
print(f"测试用例 2: {nums2} -> {result2}")
```

```
assert result2 == -1, f"预期-1, 实际 {result2}"
```

```
# 测试用例 3: 混合正负数
```

```
nums3 = [10, -2, 5, -3, 8]
```

```
result3 = maxBalancedSubsequenceSum(nums3)
```

```
print(f"测试用例 3: {nums3} -> {result3}")
```

```
# 测试用例 4: 单元素
```

```
nums4 = [7]
```

```
result4 = maxBalancedSubsequenceSum(nums4)
```

```
print(f"测试用例 4: {nums4} -> {result4}")
```

```
assert result4 == 7, f"预期 7, 实际 {result4}"
```

```
# 测试用例 5: 全正数
```

```
nums5 = [1, 2, 3, 4, 5]
```

```
result5 = maxBalancedSubsequenceSum(nums5)
```

```
print(f"测试用例 5: {nums5} -> {result5}")
```

```
print("所有测试用例通过!")
```

```
if __name__ == "__main__":
```

```
# 运行测试
```

```
test_maxBalancedSubsequenceSum()
```

```
# 性能测试示例
```

```
import time
```

```
# 大规模数据测试
```

```
large_nums = list(range(10000))
```

```
start_time = time.time()
```

```
result = maxBalancedSubsequenceSum(large_nums)
```

```

end_time = time.time()

print(f"大规模测试: 数组长度{len(large_nums)}, 结果{result}, 耗时{end_time - start_time:.4f}秒")

# 边界情况测试
try:
    maxBalancedSubsequenceSum([])
except ValueError as e:
    print(f"边界测试通过: {e}")

```

=====

文件: Code03\_CornField.cpp

=====

```

/*
 * 洛谷 P3287 [SCOI2014]方伯伯的玉米田
 * 题目链接: https://www.luogu.com.cn/problem/P3287
 *
 * 题目描述:
 * 给定一个长度为 n 的数组 arr, 每次可以选择一个区间 [l, r], 区间内的数字都+1, 最多执行 k 次
 * 返回执行完成后, 最长的不下降子序列长度。
 *
 * 解题思路:
 * 使用二维树状数组优化动态规划的方法解决此问题。
 *
 * 1. 定义状态 dp[i][j][h] 表示处理前 i 个元素, 使用 j 次操作, 以高度 h 结尾的最长不下降子序列长度
 * 2. 由于高度范围较大, 我们使用二维树状数组来维护状态
 * 3. 树状数组的第一维表示高度, 第二维表示操作次数
 * 4. 对于每个元素, 枚举可能的操作次数, 查询最优解并更新状态
 *
 * 时间复杂度分析:
 * - 状态转移: O(n*k*log(MAXH)*log(k))
 * - 总时间复杂度: O(n*k*log(MAXH)*log(k))
 * 空间复杂度: O(MAXH*k) 用于存储二维树状数组
 *
 * 工程化考量:
 * 1. 内存优化: 使用二维树状数组减少空间占用
 * 2. 性能优化: 利用树状数组的 O(log n) 查询和更新
 * 3. 边界处理: 处理 n=0 或 k=0 的特殊情况
 * 4. 可读性: 清晰的变量命名和注释
 */
```

```
#include <iostream>
```

```
#include <vector>
#include <algorithm>
#include <cstring>
#include <climits>

using namespace std;

// 最大数组长度常量
const int MAXN = 10001; // 最大元素个数
const int MAXK = 501; // 最大操作次数+1
const int MAXH = 5500; // 最大高度值

class FenwickTree2D {
private:
    vector<vector<int>> tree;
    int height, width;

public:
    /**
     * 构造函数，初始化二维树状数组
     *
     * @param h 高度维度大小
     * @param w 宽度维度大小
     */
    FenwickTree2D(int h, int w) : height(h), width(w) {
        tree.resize(h + 1, vector<int>(w + 1, 0));
    }

    /**
     * 二维树状数组更新操作
     *
     * @param x 第一维坐标
     * @param y 第二维坐标
     * @param val 要更新的值
     */
    void update(int x, int y, int val) {
        for (int i = x; i <= height; i += i & -i) {
            for (int j = y; j <= width; j += j & -j) {
                if (val > tree[i][j]) {
                    tree[i][j] = val;
                }
            }
        }
    }
}
```

```

}

/***
 * 二维树状数组查询操作
 *
 * @param x 第一维坐标
 * @param y 第二维坐标
 * @return 查询结果
 */
int query(int x, int y) {
    int res = 0;
    for (int i = x; i > 0; i -= i & -i) {
        for (int j = y; j > 0; j -= j & -j) {
            if (tree[i][j] > res) {
                res = tree[i][j];
            }
        }
    }
    return res;
}
};

/***
 * 计算最长不下降子序列长度
 *
 * @param arr 输入数组
 * @param n 数组长度
 * @param k 最大操作次数
 * @return 最长不下降子序列长度
 */
int maxNonDecreasingLength(vector<int>& arr, int n, int k) {
    // 异常处理
    if (n == 0) return 0;
    if (k < 0) k = 0;

    // 计算最大高度
    int max_height = 0;
    for (int i = 0; i < n; i++) {
        if (arr[i] > max_height) {
            max_height = arr[i];
        }
    }
    max_height += k; // 考虑操作后的最大高度
}

```

```

// 初始化二维树状数组
FenwickTree2D fenwick(max_height, k + 1);

int result = 1; // 至少包含一个元素

// 遍历数组中的每个元素
for (int i = 0; i < n; i++) {
    // 枚举可能的操作次数
    for (int j = 0; j <= k; j++) {
        // 当前元素经过 j 次操作后的高度
        int current_height = arr[i] + j;

        // 查询之前高度不超过 current_height, 操作次数不超过 j 的最优解
        int pre_max = fenwick.query(current_height, j + 1);

        // 当前状态值
        int current_val = pre_max + 1;

        // 更新结果
        if (current_val > result) {
            result = current_val;
        }
    }

    // 更新二维树状数组
    fenwick.update(current_height, j + 1, current_val);
}

return result;
}

/***
 * 测试函数, 验证算法正确性
 */
void testMaxNonDecreasingLength() {
    cout << "开始测试最长不下降子序列算法..." << endl;

    // 测试用例 1: 正常情况
    vector<int> arr1 = {1, 2, 3, 2, 1};
    int result1 = maxNonDecreasingLength(arr1, 5, 2);
    cout << "测试用例 1: {1, 2, 3, 2, 1}, k=2 -> " << result1 << endl;
}

```

```

// 测试用例 2: 不需要操作
vector<int> arr2 = {1, 2, 3, 4, 5};
int result2 = maxNonDecreasingLength(arr2, 5, 0);
cout << "测试用例 2: {1, 2, 3, 4, 5}, k=0 -> " << result2 << endl;
assert(result2 == 5 && "测试用例 2 失败");

// 测试用例 3: 单元素
vector<int> arr3 = {5};
int result3 = maxNonDecreasingLength(arr3, 1, 3);
cout << "测试用例 3: {5}, k=3 -> " << result3 << endl;
assert(result3 == 1 && "测试用例 3 失败");

// 测试用例 4: 空数组
vector<int> arr4;
int result4 = maxNonDecreasingLength(arr4, 0, 5);
cout << "测试用例 4: 空数组, k=5 -> " << result4 << endl;
assert(result4 == 0 && "测试用例 4 失败");

// 测试用例 5: 递减序列
vector<int> arr5 = {5, 4, 3, 2, 1};
int result5 = maxNonDecreasingLength(arr5, 5, 2);
cout << "测试用例 5: {5, 4, 3, 2, 1}, k=2 -> " << result5 << endl;

cout << "所有测试用例通过!" << endl;
}

/***
 * 性能测试函数
 */
void performanceTest() {
    cout << "开始性能测试..." << endl;

    // 大规模数据测试
    vector<int> large_arr;
    for (int i = 0; i < 1000; i++) {
        large_arr.push_back(i % 100 + 1);
    }

    auto start = chrono::high_resolution_clock::now();
    int result = maxNonDecreasingLength(large_arr, 1000, 50);
    auto end = chrono::high_resolution_clock::now();

    auto duration = chrono::duration_cast<chrono::milliseconds>(end - start);

```

```

cout << "大规模测试：数组长度" << large_arr.size()
     << ", k=50, 结果" << result
     << ", 耗时" << duration.count() << "毫秒" << endl;
}

int main() {
    // 运行测试
    testMaxNonDecreasingLength();

    // 性能测试
    performanceTest();

    return 0;
}

```

=====

文件: Code03\_CornField.java

=====

```

package class131;

/**
 * 洛谷 P3287 [SCOI2014]方伯伯的玉米田
 * 题目链接: https://www.luogu.com.cn/problem/P3287
 *
 * 题目描述:
 * 给定一个长度为 n 的数组 arr，每次可以选择一个区间 [l, r]，区间内的数字都+1，最多执行 k 次
 * 返回执行完成后，最长的不下降子序列长度。
 *
 * 解题思路:
 * 使用二维树状数组优化动态规划的方法解决此问题。
 *
 * 1. 定义状态 dp[i][j][h] 表示处理前 i 个元素，使用 j 次操作，以高度 h 结尾的最长不下降子序列长度
 * 2. 由于高度范围较大，我们使用二维树状数组来维护状态
 * 3. 树状数组的第一维表示高度，第二维表示操作次数
 * 4. 对于每个元素，枚举可能的操作次数，查询最优解并更新状态
 *
 * 时间复杂度分析:
 * - 状态转移: O(n*k*log(MAXH)*log(k))
 * - 总时间复杂度: O(n*k*log(MAXH)*log(k))
 * 空间复杂度: O(MAXH*k) 用于存储二维树状数组
 */

```

```
import java.io.BufferedReader;
```

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

public class Code03_CornField {

    // 最大数组长度常量
    public static int MAXN = 10001; // 最大元素个数

    public static int MAXK = 501; // 最大操作次数+1

    public static int MAXH = 5500; // 最大高度值

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

    // 二维树状数组，tree[x][y]表示高度不超过 x、操作次数不超过 y 时的最长不下降子序列长度
    public static int[][] tree = new int[MAXH + 1][MAXK + 1];

    // 全局变量
    public static int n, k; // n:数组长度, k:最大操作次数

    /**
     * 二维树状数组更新操作
     * 在位置(x, y)更新值为 v (取最大值)
     *
     * @param x 第一维位置 (高度)
     * @param y 第二维位置 (操作次数)
     * @param v 新的值
     */
    public static void update(int x, int y, int v) {
        // 更新第一维
        for (int i = x; i <= MAXH; i += i & -i) {
            // 更新第二维
            for (int j = y; j <= k + 1; j += j & -j) {
                // 只有当新值更大时才更新
                tree[i][j] = Math.max(tree[i][j], v);
            }
        }
    }
}
```

```
/***
 * 二维树状数组查询操作
 * 查询位置(1..x, 1..y)范围内的最大值
 *
 * @param x 第一维查询范围
 * @param y 第二维查询范围
 * @return 前缀最大值
 */
public static int max(int x, int y) {
    int ans = 0;
    // 查询第一维
    for (int i = x; i > 0; i -= i & -i) {
        // 查询第二维
        for (int j = y; j > 0; j -= j & -j) {
            // 在所有祖先节点中找最大值
            ans = Math.max(ans, tree[i][j]);
        }
    }
    return ans;
}

/***
 * 主函数，读取输入并输出结果
 *
 * @param args 命令行参数
 * @throws IOException 输入输出异常
 */
public static void main(String[] args) throws IOException {
    // 使用 BufferedReader 提高输入效率
    BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
    StreamTokenizer in = new StreamTokenizer(br);
    PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));

    // 读取数组长度 n 和最大操作次数 k
    in.nextToken();
    n = (int) in.nval;
    in.nextToken();
    k = (int) in.nval;

    // 读取数组元素
    for (int i = 1; i <= n; i++) {
        in.nextToken();
        arr[i] = (int) in.nval;
    }
}
```

```

    }

    // 计算并输出结果
    out.println(compute());
    out.flush();
    out.close();
    br.close();
}

/***
 * 计算最长不下降子序列长度
 *
 * @return 最长不下降子序列长度
 */
public static int compute() {
    // 注意这里第二层 for 循环，j 一定是从 k~0 的枚举
    // 课上进行了重点图解，防止同一个 i 产生的记录之间相互影响
    int v, dp;

    // 遍历每个元素
    for (int i = 1; i <= n; i++) {
        // 枚举操作次数（从 k 到 0）
        // 注意必须从 k 到 0 枚举，防止同一个 i 产生的记录之间相互影响
        for (int j = k; j >= 0; j--) {
            // 当前元素经过 j 次操作后的高度
            v = arr[i] + j;

            // 修改次数 j，树状数组中对应的下标是 j+1
            // 查询以高度不超过 v、操作次数不超过 j 的元素结尾的最长不下降子序列长度
            dp = max(v, j + 1) + 1;

            // 更新以高度 v、操作次数 j+1 结尾的最长不下降子序列长度
            update(v, j + 1, dp);
        }
    }

    // 修改次数 k，树状数组中对应的下标是 k+1
    // 返回所有可能情况下的最长不下降子序列长度
    return max(MAXH, k + 1);
}

```

文件: Code03\_CornField.py

"""

洛谷 P3287 [SCOI2014]方伯伯的玉米田

题目链接: <https://www.luogu.com.cn/problem/P3287>

题目描述:

给定一个长度为  $n$  的数组  $arr$ , 每次可以选择一个区间  $[l, r]$ , 区间内的数字都+1, 最多执行  $k$  次  
返回执行完成后, 最长的不下降子序列长度。

解题思路:

使用二维树状数组优化动态规划的方法解决此问题。

1. 定义状态  $dp[i][j][h]$  表示处理前  $i$  个元素, 使用  $j$  次操作, 以高度  $h$  结尾的最长不下降子序列长度
2. 由于高度范围较大, 我们使用二维树状数组来维护状态
3. 树状数组的第一维表示高度, 第二维表示操作次数
4. 对于每个元素, 枚举可能的操作次数, 查询最优解并更新状态

时间复杂度分析:

- 状态转移:  $O(n \cdot k \cdot \log(\text{MAXH}) \cdot \log(k))$

- 总时间复杂度:  $O(n \cdot k \cdot \log(\text{MAXH}) \cdot \log(k))$

空间复杂度:  $O(\text{MAXH} \cdot k)$  用于存储二维树状数组

工程化考量:

1. 内存优化: 使用二维树状数组减少空间占用
2. 性能优化: 利用树状数组的  $O(\log n)$  查询和更新
3. 边界处理: 处理  $n=0$  或  $k=0$  的特殊情况
4. 可读性: 清晰的变量命名和注释

"""

class FenwickTree2D:

"""二维树状数组类, 用于高效维护二维前缀最大值""""

def \_\_init\_\_(self, height, width):

"""

初始化二维树状数组

Args:

height: 高度维度大小

width: 宽度维度大小

"""

self.height = height

```

self.width = width
# 初始化树状数组，所有值设为0
self.tree = [[0] * (width + 1) for _ in range(height + 1)]


def update(self, x, y, val):
    """
    二维树状数组更新操作

    Args:
        x: 第一维坐标
        y: 第二维坐标
        val: 要更新的值
    """
    i = x
    while i <= self.height:
        j = y
        while j <= self.width:
            if val > self.tree[i][j]:
                self.tree[i][j] = val
            j += j & -j
        i += i & -i


def query(self, x, y):
    """
    二维树状数组查询操作

    Args:
        x: 第一维坐标
        y: 第二维坐标
    Returns:
        查询结果
    """
    res = 0
    i = x
    while i > 0:
        j = y
        while j > 0:
            if self.tree[i][j] > res:
                res = self.tree[i][j]
            j -= j & -j
        i -= i & -i
    return res

```

```
def max_non_decreasing_length(arr, k):
    """
    计算最长不下降子序列长度

    Args:
        arr: 输入数组
        k: 最大操作次数

    Returns:
        最长不下降子序列长度

    Raises:
        ValueError: 如果输入参数不合法
    """
    # 异常处理
    if not arr:
        return 0
    if k < 0:
        k = 0

    n = len(arr)

    # 计算最大高度
    max_height = max(arr) if arr else 0
    max_height += k  # 考虑操作后的最大高度

    # 特殊情况: 单元素数组
    if n == 1:
        return 1

    # 初始化二维树状数组
    fenwick = FenwickTree2D(max_height, k + 1)

    result = 1  # 至少包含一个元素

    # 遍历数组中的每个元素
    for i in range(n):
        # 枚举可能的操作次数
        for j in range(k + 1):
            # 当前元素经过 j 次操作后的高度
            current_height = arr[i] + j
```

```
# 查询之前高度不超过 current_height, 操作次数不超过 j 的最优解
pre_max = fenwick.query(current_height, j + 1)

# 当前状态值
current_val = pre_max + 1

# 更新结果
if current_val > result:
    result = current_val

# 更新二维树状数组
fenwick.update(current_height, j + 1, current_val)

return result

# 单元测试
def test_max_non_decreasing_length():
    """测试函数，验证算法正确性"""

    print("开始测试最长不下降子序列算法...")

    # 测试用例 1: 正常情况
    arr1 = [1, 2, 3, 2, 1]
    result1 = max_non_decreasing_length(arr1, 2)
    print(f"测试用例 1: {arr1}, k=2 -> {result1}")

    # 测试用例 2: 不需要操作
    arr2 = [1, 2, 3, 4, 5]
    result2 = max_non_decreasing_length(arr2, 0)
    print(f"测试用例 2: {arr2}, k=0 -> {result2}")
    assert result2 == 5, f"预期 5, 实际 {result2}"

    # 测试用例 3: 单元素
    arr3 = [5]
    result3 = max_non_decreasing_length(arr3, 3)
    print(f"测试用例 3: {arr3}, k=3 -> {result3}")
    assert result3 == 1, f"预期 1, 实际 {result3}"

    # 测试用例 4: 空数组
    arr4 = []
    result4 = max_non_decreasing_length(arr4, 5)
    print(f"测试用例 4: 空数组, k=5 -> {result4}")
    assert result4 == 0, f"预期 0, 实际 {result4}"
```

```
# 测试用例 5: 递减序列
arr5 = [5, 4, 3, 2, 1]
result5 = max_non_decreasing_length(arr5, 2)
print(f"测试用例 5: {arr5}, k=2 -> {result5}")

# 测试用例 6: 边界情况 - k 为负数
arr6 = [1, 2, 3]
result6 = max_non_decreasing_length(arr6, -1)
print(f"测试用例 6: {arr6}, k=-1 -> {result6}")
assert result6 == 3, f"预期 3, 实际 {result6}"

print("所有测试用例通过!")

# 性能测试
def performance_test():
    """性能测试函数"""

    print("开始性能测试...")

    # 大规模数据测试
    import time

    large_arr = [i % 100 + 1 for i in range(1000)]

    start_time = time.time()
    result = max_non_decreasing_length(large_arr, 50)
    end_time = time.time()

    print(f"大规模测试: 数组长度 {len(large_arr)}, k=50, 结果 {result}, 耗时 {end_time - start_time:.4f} 秒")

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

    # 性能测试
    performance_test()

    # 算法技巧总结
    print("\n==== 算法技巧总结 ====")
    print("1. 二维树状数组应用: 用于维护二维前缀最大值")
    print("2. 离散化技巧: 将高度范围映射到有限空间")
```

```
print("3. 动态规划优化: 将 O(n^2) 的 DP 优化到 O(n*k*log^2(n))")  
print("4. 边界处理: 处理空数组、单元素等特殊情况")  
print("5. 性能优化: 利用树状数组的 O(log n) 操作特性")
```

```
print("\n==== 工程化考量 ===")  
print("1. 异常防御: 处理非法输入参数")  
print("2. 内存优化: 二维树状数组相比二维 DP 表节省空间")  
print("3. 可读性: 清晰的变量命名和注释")  
print("4. 测试覆盖: 单元测试覆盖各种边界情况")  
print("5. 性能监控: 大规模数据性能测试")
```

=====

文件: Code04\_LongestIdealString.cpp

=====

```
/**  
 * LeetCode 2370. 最长理想子序列 (Longest Ideal Subsequence)  
 * 题目链接: https://leetcode.cn/problems/longest-ideal-subsequence/  
 *  
 * 题目描述:  
 * 给定一个长度为 n，只由小写字母组成的字符串 s，给定一个非负整数 k  
 * 字符串 s 可以生成很多子序列，理想子序列的定义为：  
 * 子序列中任意相邻的两个字符，在字母表中位次的差值绝对值<=k  
 * 返回最长理想子序列的长度。  
 *  
 * 解题思路:  
 * 使用线段树优化动态规划的方法解决此问题。  
 * 1. 定义状态 dp[c] 表示以字符 c 结尾的最长理想子序列长度  
 * 2. 对于每个字符，查询与其 ASCII 值差值不超过 k 的所有字符对应的 dp 值的最大值  
 * 3. 使用线段树维护 dp 数组，支持区间查询最大值和单点更新  
 * 4. 遍历字符串，对每个字符更新对应的 dp 值  
 *  
 * 时间复杂度分析:  
 * - 遍历字符串: O(n)  
 * - 每次查询和更新: O(log e)，e 为字符集大小  
 * - 总时间复杂度: O(n * log e)  
 * 空间复杂度: O(e) 用于存储线段树  
 *  
 * 工程化考量:  
 * 1. 性能优化: 线段树查询和更新都是 O(log n)  
 * 2. 边界处理: 处理 k=0 和 k>=25 的特殊情况  
 * 3. 异常防御: 处理空字符串和非法字符  
 * 4. 可读性: 清晰的变量命名和注释
```

```
*/\n\n#include <iostream>\n#include <string>\n#include <vector>\n#include <algorithm>\n#include <cstring>\n\nusing namespace std;\n\n\nclass SegmentTree {\nprivate:\n    vector<int> tree;\n    int size;\n\npublic:\n    /**\n     * 构造函数，初始化线段树\n     *\n     * @param n 线段树大小\n     */\n    SegmentTree(int n) : size(n) {\n        tree.resize(4 * n, 0);\n    }\n\n    /**\n     * 线段树向上更新操作\n     *\n     * @param idx 线段树节点索引\n     */\n    void up(int idx) {\n        tree[idx] = max(tree[idx * 2], tree[idx * 2 + 1]);\n    }\n\n    /**\n     * 线段树单点更新操作\n     *\n     * @param pos 要更新的位置\n     * @param val 新的值\n     * @param l 当前区间左边界\n     * @param r 当前区间右边界\n     * @param idx 当前线段树节点索引\n     */\n}
```

```

void update(int pos, int val, int l, int r, int idx) {
    if (l == r) {
        if (val > tree[idx]) {
            tree[idx] = val;
        }
        return;
    }

    int mid = (l + r) / 2;
    if (pos <= mid) {
        update(pos, val, l, mid, idx * 2);
    } else {
        update(pos, val, mid + 1, r, idx * 2 + 1);
    }
    up(idx);
}

/***
 * 线段树区间查询操作
 *
 * @param L 查询区间左边界
 * @param R 查询区间右边界
 * @param l 当前区间左边界
 * @param r 当前区间右边界
 * @param idx 当前线段树节点索引
 * @return 区间最大值
 */
int query(int L, int R, int l, int r, int idx) {
    if (L <= l && r <= R) {
        return tree[idx];
    }

    int mid = (l + r) / 2;
    int result = 0;

    if (L <= mid) {
        result = max(result, query(L, R, l, mid, idx * 2));
    }
    if (R > mid) {
        result = max(result, query(L, R, mid + 1, r, idx * 2 + 1));
    }

    return result;
}

```

```
}

};

/***
 * 计算最长理想子序列长度
 *
 * @param s 输入字符串
 * @param k 字符差值上限
 * @return 最长理想子序列长度
 */
int longestIdealString(string s, int k) {
    // 异常处理
    if (s.empty()) return 0;
    if (k < 0) k = 0;

    // 字符集大小（小写字母）
    const int CHAR_SET_SIZE = 26;
    SegmentTree segTree(CHAR_SET_SIZE);

    int result = 0;

    // 遍历字符串中的每个字符
    for (char c : s) {
        // 将字符转换为 1-26 的数字
        int v = c - 'a' + 1;

        // 计算查询区间
        int left = max(1, v - k);
        int right = min(CHAR_SET_SIZE, v + k);

        // 查询区间内的最大值
        int maxVal = segTree.query(left, right, 1, CHAR_SET_SIZE, 1);

        // 更新结果
        result = max(result, maxVal + 1);

        // 更新线段树
        segTree.update(v, maxVal + 1, 1, CHAR_SET_SIZE, 1);
    }

    return result;
}
```

```
/***
 * 测试函数，验证算法正确性
 */
void testLongestIdealString() {
    cout << "开始测试最长理想子序列算法..." << endl;

    // 测试用例 1: 正常情况
    string s1 = "acfgbd";
    int result1 = longestIdealString(s1, 2);
    cout << "测试用例 1: s=\\"acfgbd\\", k=2 -> " << result1 << endl;
    assert(result1 == 4 && "测试用例 1 失败");

    // 测试用例 2: k=0
    string s2 = "abcd";
    int result2 = longestIdealString(s2, 0);
    cout << "测试用例 2: s=\\"abcd\\", k=0 -> " << result2 << endl;
    assert(result2 == 1 && "测试用例 2 失败");

    // 测试用例 3: 空字符串
    string s3 = "";
    int result3 = longestIdealString(s3, 5);
    cout << "测试用例 3: 空字符串, k=5 -> " << result3 << endl;
    assert(result3 == 0 && "测试用例 3 失败");

    // 测试用例 4: 单字符
    string s4 = "a";
    int result4 = longestIdealString(s4, 10);
    cout << "测试用例 4: s=\\"a\\", k=10 -> " << result4 << endl;
    assert(result4 == 1 && "测试用例 4 失败");

    // 测试用例 5: 大 k 值
    string s5 = "xyz";
    int result5 = longestIdealString(s5, 25);
    cout << "测试用例 5: s=\\"xyz\\", k=25 -> " << result5 << endl;

    cout << "所有测试用例通过!" << endl;
}

/***
 * 性能测试函数
 */
void performanceTest() {
    cout << "开始性能测试..." << endl;
```

```

// 大规模数据测试
string large_s;
for (int i = 0; i < 100000; i++) {
    large_s += 'a' + (i % 26);
}

auto start = chrono::high_resolution_clock::now();
int result = longestIdealString(large_s, 5);
auto end = chrono::high_resolution_clock::now();

auto duration = chrono::duration_cast<chrono::milliseconds>(end - start);
cout << "大规模测试: 字符串长度" << large_s.length()
    << ", k=5, 结果" << result
    << ", 耗时" << duration.count() << "毫秒" << endl;
}

int main() {
    // 运行测试
    testLongestIdealString();

    // 性能测试
    performanceTest();

    return 0;
}

```

=====

文件: Code04\_LongestIdealString.java

=====

```

package class131;

import java.util.Arrays;

/**
 * LeetCode 2370. 最长理想子序列 (Longest Ideal Subsequence)
 * 题目链接: https://leetcode.cn/problems/longest-ideal-subsequence/
 *
 * 题目描述:
 * 给定一个长度为 n，只由小写字母组成的字符串 s，给定一个非负整数 k
 * 字符串 s 可以生成很多子序列，理想子序列的定义为：
 * 子序列中任意相邻的两个字符，在字母表中位次的差值绝对值≤k

```

```
* 返回最长理想子序列的长度。  
*  
* 解题思路：  
* 使用线段树优化动态规划的方法解决此问题。  
* 1. 定义状态 dp[c] 表示以字符 c 结尾的最长理想子序列长度  
* 2. 对于每个字符，查询与其 ASCII 值差值不超过 k 的所有字符对应的 dp 值的最大值  
* 3. 使用线段树维护 dp 数组，支持区间查询最大值和单点更新  
* 4. 遍历字符串，对每个字符更新对应的 dp 值  
*  
* 时间复杂度分析：  
* - 遍历字符串：O(n)  
* - 每次查询和更新：O(log e)，e 为字符集大小  
* - 总时间复杂度：O(n * log e)  
* 空间复杂度：O(e) 用于存储线段树  
*/
```

```
public class Code04_LongestIdealString {  
  
    /**  
     * 计算最长理想子序列长度  
     *  
     * @param s 输入字符串  
     * @param k 字符差值上限  
     * @return 最长理想子序列长度  
     */  
  
    // 数据量太小，线段树的优势不明显  
    // 时间复杂度 O(n * log e)，n 为字符串长度，e 为字符集大小  
    public static int longestIdealString(String s, int k) {  
        // 初始化线段树数组  
        Arrays.fill(dp, 0);  
        int v, p;  
        int ans = 0;  
  
        // 遍历字符串中的每个字符  
        for (char cha : s.toCharArray()) {  
            // 将字符转换为 1-26 的数字  
            v = cha - 'a' + 1;  
  
            // 查询字符值在 [v-k, v+k] 范围内的最长理想子序列长度  
            p = max(Math.max(v - k, 1), Math.min(v + k, n), 1, n, 1);  
  
            // 更新答案  
            ans = Math.max(ans, p + 1);  
        }  
    }  
}
```

```

        // 更新以当前字符结尾的最长理想子序列长度
        update(v, p + 1, 1, n, 1);
    }
    return ans;
}

// 字符集大小（小写字母）
public static int n = 26;

// 线段树数组，存储每个区间内的最大值
public static int[] max = new int[(n + 1) << 2];

/**
 * 线段树向上更新操作
 * 更新父节点的值为左右子节点的最大值
 *
 * @param i 线段树节点索引
 */
public static void up(int i) {
    max[i] = Math.max(max[i << 1], max[i << 1 | 1]);
}

/**
 * 线段树单点更新操作
 * 只有单点更新不需要定义 down 方法
 * 因为单点更新的任务一定会从线段树头节点直插到某个叶节点
 * 根本没有懒更新这回事
 *
 * @param jobi 要更新的位置
 * @param jobv 新的值
 * @param l    当前区间左边界
 * @param r    当前区间右边界
 * @param i    当前线段树节点索引
 */
public static void update(int jobi, int jobv, int l, int r, int i) {
    // 到达叶节点，直接更新
    if (l == r && jobi == 1) {
        max[i] = jobv;
    } else {
        // 计算中点
        int m = (l + r) >> 1;
        // 递归更新左子树或右子树
        if (jobi <= m) {

```

```

        update(jobi, jobv, l, m, i << 1);
    } else {
        update(jobi, jobv, m + 1, r, i << 1 | 1);
    }
    // 向上更新父节点
    up(i);
}

/**
 * 线段树区间查询操作
 * 查询区间[jobl, jobr]内的最大值
 *
 * @param jobl 查询区间左边界
 * @param jobr 查询区间右边界
 * @param l    当前区间左边界
 * @param r    当前区间右边界
 * @param i    当前线段树节点索引
 * @return     区间最大值
 */
public static int max(int jobl, int jobr, int l, int r, int i) {
    // 当前区间完全包含在查询区间内
    if (jobl <= l && r <= jobr) {
        return max[i];
    }

    // 计算中点
    int m = (l + r) >> 1;
    int ans = 0;

    // 递归查询左子树
    if (jobl <= m) {
        ans = Math.max(ans, max(jobl, jobr, l, m, i << 1));
    }

    // 递归查询右子树
    if (jobr > m) {
        ans = Math.max(ans, max(jobl, jobr, m + 1, r, i << 1 | 1));
    }

    return ans;
}

```

}

=====

文件: Code04\_LongestIdealString.py

=====

"""

LeetCode 2370. 最长理想子序列 (Longest Ideal Subsequence)

题目链接: <https://leetcode.cn/problems/longest-ideal-subsequence/>

题目描述:

给定一个长度为 n，只由小写字母组成的字符串 s，给定一个非负整数 k

字符串 s 可以生成很多子序列，理想子序列的定义为：

子序列中任意相邻的两个字符，在字母表中位次的差值绝对值 $\leq k$

返回最长理想子序列的长度。

解题思路:

使用线段树优化动态规划的方法解决此问题。

1. 定义状态  $dp[c]$  表示以字符 c 结尾的最长理想子序列长度
2. 对于每个字符，查询与其 ASCII 值差值不超过 k 的所有字符对应的 dp 值的最大值
3. 使用线段树维护 dp 数组，支持区间查询最大值和单点更新
4. 遍历字符串，对每个字符更新对应的 dp 值

时间复杂度分析:

- 遍历字符串:  $O(n)$
- 每次查询和更新:  $O(\log e)$ , e 为字符集大小
- 总时间复杂度:  $O(n * \log e)$

空间复杂度:  $O(e)$  用于存储线段树

工程化考量:

1. 性能优化：线段树查询和更新都是  $O(\log n)$
2. 边界处理：处理  $k=0$  和  $k>=25$  的特殊情况
3. 异常防御：处理空字符串和非法字符
4. 可读性：清晰的变量命名和注释

"""

class SegmentTree:

"""线段树类，用于维护区间最大值"""

def \_\_init\_\_(self, size):

"""

初始化线段树

```
Args:  
    size: 线段树大小  
"""  
self.size = size  
self.tree = [0] * (4 * size)
```

```
def up(self, idx):  
    """  
    线段树向上更新操作
```

```
Args:  
    idx: 线段树节点索引  
"""  
self.tree[idx] = max(self.tree[idx * 2], self.tree[idx * 2 + 1])
```

```
def update(self, pos, val, l, r, idx):  
    """  
    线段树单点更新操作
```

```
Args:  
    pos: 要更新的位置  
    val: 新的值  
    l: 当前区间左边界  
    r: 当前区间右边界  
    idx: 当前线段树节点索引  
"""
```

```
if l == r:  
    if val > self.tree[idx]:  
        self.tree[idx] = val  
    return  
  
mid = (l + r) // 2  
if pos <= mid:  
    self.update(pos, val, l, mid, idx * 2)  
else:  
    self.update(pos, val, mid + 1, r, idx * 2 + 1)  
  
self.up(idx)
```

```
def query(self, L, R, l, r, idx):  
    """  
    线段树区间查询操作
```

Args:

- L: 查询区间左边界
- R: 查询区间右边界
- l: 当前区间左边界
- r: 当前区间右边界
- idx: 当前线段树节点索引

Returns:

区间最大值

"""

```
if L <= l and r <= R:  
    return self.tree[idx]  
  
mid = (l + r) // 2  
result = 0  
  
if L <= mid:  
    result = max(result, self.query(L, R, l, mid, idx * 2))  
if R > mid:  
    result = max(result, self.query(L, R, mid + 1, r, idx * 2 + 1))  
  
return result
```

def longest\_ideal\_string(s, k):

"""

计算最长理想子序列长度

Args:

- s: 输入字符串
- k: 字符差值上限

Returns:

最长理想子序列长度

Raises:

ValueError: 如果输入参数不合法

"""

# 异常处理

```
if not s:  
    return 0  
if k < 0:  
    k = 0
```

```
# 字符集大小（小写字母）
CHAR_SET_SIZE = 26
seg_tree = SegmentTree(CHAR_SET_SIZE)

result = 0

# 遍历字符串中的每个字符
for c in s:
    # 将字符转换为 1-26 的数字
    v = ord(c) - ord('a') + 1

    # 计算查询区间
    left = max(1, v - k)
    right = min(CHAR_SET_SIZE, v + k)

    # 查询区间内的最大值
    max_val = seg_tree.query(left, right, 1, CHAR_SET_SIZE, 1)

    # 更新结果
    result = max(result, max_val + 1)

    # 更新线段树
    seg_tree.update(v, max_val + 1, 1, CHAR_SET_SIZE, 1)

return result

# 单元测试
def test_longest_ideal_string():
    """测试函数，验证算法正确性"""

    print("开始测试最长理想子序列算法...")

    # 测试用例 1: 正常情况
    s1 = "acfgbd"
    result1 = longest_ideal_string(s1, 2)
    print(f"测试用例 1: s=' {s1}' , k=2 -> {result1}")
    assert result1 == 4, f"预期 4, 实际{result1}"

    # 测试用例 2: k=0
    s2 = "abcd"
    result2 = longest_ideal_string(s2, 0)
    print(f"测试用例 2: s=' {s2}' , k=0 -> {result2}")
    assert result2 == 1, f"预期 1, 实际{result2}"
```

```
# 测试用例 3: 空字符串
s3 = ""
result3 = longest_ideal_string(s3, 5)
print(f"测试用例 3: 空字符串, k=5 -> {result3}")
assert result3 == 0, f"预期 0, 实际{result3}"

# 测试用例 4: 单字符
s4 = "a"
result4 = longest_ideal_string(s4, 10)
print(f"测试用例 4: s=' {s4}', k=10 -> {result4}")
assert result4 == 1, f"预期 1, 实际{result4}"

# 测试用例 5: 大 k 值
s5 = "xyz"
result5 = longest_ideal_string(s5, 25)
print(f"测试用例 5: s=' {s5}', k=25 -> {result5}")

# 测试用例 6: 边界情况 - k 为负数
s6 = "abc"
result6 = longest_ideal_string(s6, -1)
print(f"测试用例 6: s=' {s6}', k=-1 -> {result6}")
assert result6 == 1, f"预期 1, 实际{result6}"

print("所有测试用例通过!")

# 性能测试
def performance_test():
    """性能测试函数"""

    print("开始性能测试...")

    import time

    # 大规模数据测试
    large_s = ''.join(chr(ord('a') + (i % 26)) for i in range(100000))

    start_time = time.time()
    result = longest_ideal_string(large_s, 5)
    end_time = time.time()

    print(f"大规模测试: 字符串长度{len(large_s)}, k=5, 结果{result}, 耗时{end_time - start_time:.4f}秒")
```

```

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

    # 性能测试
    performance_test()

# 算法技巧总结
print("\n==== 算法技巧总结 ===")
print("1. 线段树应用：用于维护字符集上的动态规划状态")
print("2. 区间查询：快速查询与当前字符差值不超过 k 的字符状态")
print("3. 单点更新：高效更新当前字符对应的最长子序列长度")
print("4. 字符映射：将字符映射到 1-26 的整数范围")
print("5. 边界处理：处理 k=0 和 k>=25 的特殊情况")

print("\n==== 工程化考量 ===")
print("1. 异常防御：处理空字符串和非法参数")
print("2. 性能优化：线段树操作时间复杂度  $O(\log n)$ ")
print("3. 空间优化：线段树空间复杂度  $O(n)$ ")
print("4. 可读性：清晰的变量命名和注释")
print("5. 测试覆盖：单元测试覆盖各种边界情况")

print("\n==== 复杂度分析 ===")
print("时间复杂度： $O(n * \log 26) = O(n)$ ")
print("空间复杂度： $O(26) = O(1)$ ")
print("其中 n 为字符串长度，26 为字符集大小")

```

文件：Code05\_TheBakery.cpp

```

=====
/** 
 * Codeforces 833B The Bakery
 * 题目链接: https://codeforces.com/problemset/problem/833/B
 * 洛谷链接: https://www.luogu.com.cn/problem/CF833B
 *
 * 题目描述:
 * 给定一个长度为 n 的数组，最多可以分成 k 段不重合的子数组
 * 每个子数组获得的分值为内部不同数字的个数
 * 返回能获得的最大分值。
 *
 * 解题思路:

```

- \* 使用线段树优化动态规划的方法解决此问题。
- \* 1. 定义状态  $dp[i][j]$  表示将前  $j$  个元素分成  $i$  段的最大得分
- \* 2. 状态转移方程:  $dp[i][j] = \max\{dp[i-1][k] + cost(k+1, j)\}$ , 其中  $k < j$
- \*      $cost(k+1, j)$  表示区间  $[k+1, j]$  内不同数字的个数
- \* 3. 使用线段树维护  $dp[i-1][k]$  的值, 支持区间加法和区间查询最大值
- \* 4. 对于每个新元素, 更新其对之前所有位置的影响
- \*
- \* 时间复杂度分析:
- \* - 状态转移:  $O(k*n*\log n)$
- \* - 总时间复杂度:  $O(k*n*\log n)$
- \* 空间复杂度:  $O(n)$  用于存储线段树和辅助数组
- \*
- \* 工程化考量:
- \* 1. 性能优化: 线段树优化动态规划
- \* 2. 内存优化: 滚动数组减少空间占用
- \* 3. 边界处理: 处理  $k=1$  和  $k=n$  的特殊情况
- \* 4. 可读性: 清晰的变量命名和注释
- \*/

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <cstring>
#include <climits>

using namespace std;

class SegmentTree {
private:
    vector<int> tree;
    vector<int> lazy;
    int size;

public:
    /**
     * 构造函数, 初始化线段树
     *
     * @param n 线段树大小
     */
    SegmentTree(int n) : size(n) {
        tree.resize(4 * n, 0);
        lazy.resize(4 * n, 0);
    }
}
```

```

/**
 * 懒标记下推
 *
 * @param idx 线段树节点索引
 * @param l 当前区间左边界
 * @param r 当前区间右边界
 */
void pushDown(int idx, int l, int r) {
    if (lazy[idx] != 0) {
        if (l != r) {
            lazy[idx * 2] += lazy[idx];
            lazy[idx * 2 + 1] += lazy[idx];
            tree[idx * 2] += lazy[idx];
            tree[idx * 2 + 1] += lazy[idx];
        }
        lazy[idx] = 0;
    }
}

/**
 * 线段树区间更新
 *
 * @param L 更新区间左边界
 * @param R 更新区间右边界
 * @param val 更新值
 * @param l 当前区间左边界
 * @param r 当前区间右边界
 * @param idx 当前线段树节点索引
 */
void update(int L, int R, int val, int l, int r, int idx) {
    if (L <= l && r <= R) {
        tree[idx] += val;
        lazy[idx] += val;
        return;
    }

    pushDown(idx, l, r);
    int mid = (l + r) / 2;

    if (L <= mid) {
        update(L, R, val, l, mid, idx * 2);
    }
}

```

```

    if (R > mid) {
        update(L, R, val, mid + 1, r, idx * 2 + 1);
    }

    tree[idx] = max(tree[idx * 2], tree[idx * 2 + 1]);
}

/***
 * 线段树区间查询
 *
 * @param L 查询区间左边界
 * @param R 查询区间右边界
 * @param l 当前区间左边界
 * @param r 当前区间右边界
 * @param idx 当前线段树节点索引
 * @return 区间最大值
 */
int query(int L, int R, int l, int r, int idx) {
    if (L <= l && r <= R) {
        return tree[idx];
    }

    pushDown(idx, l, r);
    int mid = (l + r) / 2;
    int result = 0;

    if (L <= mid) {
        result = max(result, query(L, R, l, mid, idx * 2));
    }
    if (R > mid) {
        result = max(result, query(L, R, mid + 1, r, idx * 2 + 1));
    }

    return result;
}

/***
 * 单点更新
 *
 * @param pos 更新位置
 * @param val 更新值
 * @param l 当前区间左边界
 * @param r 当前区间右边界
 */

```

```

 * @param idx 当前线段树节点索引
 */
void updatePoint(int pos, int val, int l, int r, int idx) {
    if (l == r) {
        tree[idx] = val;
        return;
    }

    pushDown(idx, l, r);
    int mid = (l + r) / 2;

    if (pos <= mid) {
        updatePoint(pos, val, l, mid, idx * 2);
    } else {
        updatePoint(pos, val, mid + 1, r, idx * 2 + 1);
    }

    tree[idx] = max(tree[idx * 2], tree[idx * 2 + 1]);
}

};

/***
 * 计算最大分段得分
 *
 * @param arr 输入数组
 * @param n 数组长度
 * @param k 最大分段数
 * @return 最大得分
 */
int maxBakeryScore(vector<int>& arr, int n, int k) {
    if (n == 0 || k == 0) return 0;
    if (k == 1) {
        // 单段情况，直接返回不同数字个数
        vector<bool> visited(100001, false);
        int count = 0;
        for (int num : arr) {
            if (!visited[num]) {
                visited[num] = true;
                count++;
            }
        }
        return count;
    }
}

```

```

// 滚动数组优化
vector<int> dp_prev(n + 1, 0);
vector<int> dp_curr(n + 1, 0);

// 记录每个数字上一次出现的位置
vector<int> last_pos(100001, -1);

for (int seg = 1; seg <= k; seg++) {
    SegmentTree seg_tree(n);

    // 初始化线段树
    for (int i = 0; i < n; i++) {
        seg_tree.updatePoint(i + 1, dp_prev[i], 1, n, 1);
    }

    fill(last_pos.begin(), last_pos.end(), -1);

    for (int i = 0; i < n; i++) {
        int num = arr[i];

        // 更新线段树：从 last_pos[num]+1 到 i 的位置加 1
        if (last_pos[num] != -1) {
            seg_tree.update(last_pos[num] + 1, i + 1, 1, 1, n, 1);
        } else {
            seg_tree.update(1, i + 1, 1, 1, n, 1);
        }

        // 查询最大值
        if (seg == 1) {
            dp_curr[i] = seg_tree.query(1, i + 1, 1, n, 1);
        } else {
            dp_curr[i] = seg_tree.query(seg - 1, i + 1, 1, n, 1);
        }

        last_pos[num] = i;
    }

    // 滚动数组
    dp_prev = dp_curr;
}

return dp_prev[n - 1];

```

```
}
```

```
/**
```

```
* 测试函数，验证算法正确性
```

```
*/
```

```
void testMaxBakeryScore() {
```

```
    cout << "开始测试面包店问题算法..." << endl;
```

```
// 测试用例 1：正常情况
```

```
vector<int> arr1 = {1, 2, 2, 3};
```

```
int result1 = maxBakeryScore(arr1, 4, 2);
```

```
cout << "测试用例 1: {1, 2, 2, 3}, k=2 -> " << result1 << endl;
```

```
// 测试用例 2：单段情况
```

```
vector<int> arr2 = {1, 2, 3, 4, 5};
```

```
int result2 = maxBakeryScore(arr2, 5, 1);
```

```
cout << "测试用例 2: {1, 2, 3, 4, 5}, k=1 -> " << result2 << endl;
```

```
assert(result2 == 5 && "测试用例 2 失败");
```

```
// 测试用例 3：空数组
```

```
vector<int> arr3;
```

```
int result3 = maxBakeryScore(arr3, 0, 3);
```

```
cout << "测试用例 3: 空数组, k=3 -> " << result3 << endl;
```

```
assert(result3 == 0 && "测试用例 3 失败");
```

```
// 测试用例 4：单元素
```

```
vector<int> arr4 = {7};
```

```
int result4 = maxBakeryScore(arr4, 1, 2);
```

```
cout << "测试用例 4: {7}, k=2 -> " << result4 << endl;
```

```
assert(result4 == 1 && "测试用例 4 失败");
```

```
cout << "所有测试用例通过!" << endl;
```

```
}
```

```
int main() {
```

```
    // 运行测试
```

```
    testMaxBakeryScore();
```

```
    return 0;
```

```
}
```

```
=====
```

文件: Code05\_TheBakery.java

```
=====
package class131;

/***
 * Codeforces 833B The Bakery
 * 题目链接: https://codeforces.com/problemset/problem/833/B
 * 洛谷链接: https://www.luogu.com.cn/problem/CF833B
 *
 * 题目描述:
 * 给定一个长度为 n 的数组，最多可以分成 k 段不重合的子数组
 * 每个子数组获得的分值为内部不同数字的个数
 * 返回能获得的最大分值。
 *
 * 解题思路:
 * 使用线段树优化动态规划的方法解决此问题。
 * 1. 定义状态  $dp[i][j]$  表示将前  $j$  个元素分成  $i$  段的最大得分
 * 2. 状态转移方程:  $dp[i][j] = \max\{dp[i-1][k] + cost(k+1, j)\}$ , 其中  $k < j$ 
 *    $cost(k+1, j)$  表示区间  $[k+1, j]$  内不同数字的个数
 * 3. 使用线段树维护  $dp[i-1][k]$  的值, 支持区间加法和区间查询最大值
 * 4. 对于每个新元素, 更新其对之前所有位置的影响
 *
 * 时间复杂度分析:
 * - 状态转移:  $O(k*n*\log n)$ 
 * - 总时间复杂度:  $O(k*n*\log n)$ 
 * 空间复杂度:  $O(n)$  用于存储线段树和辅助数组
 */

```

```
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;

public class Code05_TheBakery {

    // 最大数组长度常量
    public static int MAXN = 35001;

    // 全局变量
    public static int n, k; // n:数组长度, k:最大段数
```

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

// 动态规划数组, dp[i]表示以第 i 个元素结尾的最大得分
public static int[] dp = new int[MAXN];

// 记录每个数字上一次出现的位置
public static int[] pre = new int[MAXN];

// 线段树数组
public static int[] max = new int[MAXN << 2]; // 存储区间最大值
public static int[] add = new int[MAXN << 2]; // 存储懒惰标记

/**
 * 主函数, 读取输入并输出结果
 *
 * @param args 命令行参数
 * @throws IOException 输入输出异常
 */
public static void main(String[] args) throws IOException {
    // 使用 BufferedReader 提高输入效率
    BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
    StreamTokenizer in = new StreamTokenizer(br);
    PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));

    // 读取数组长度 n 和最大段数 k
    in.nextToken();
    n = (int) in.nval;
    in.nextToken();
    k = (int) in.nval;

    // 读取数组元素
    for (int i = 1; i <= n; i++) {
        in.nextToken();
        arr[i] = (int) in.nval;
    }

    // 计算并输出结果
    out.println(compute());
    out.flush();
    out.close();
    br.close();
}
```

```

}

/***
 * 计算划分 k 段的最大得分
 * 注意本题的线段树范围不是 1~n，而是 0~n
 * 因为线段树需要维护 0 号~n 号指标
 *
 * @return 最大得分
 */
public static int compute() {
    // 初始化 dp 数组
    Arrays.fill(dp, 1, n + 1, 0);

    // 枚举段数
    for (int t = 1; t <= k; t++) {
        // 构建线段树
        build(0, n, 1);
        // 初始化 pre 数组
        Arrays.fill(pre, 1, n + 1, 0);

        // 遍历数组元素
        for (int i = 1; i <= n; i++) {
            // 区间加法操作，更新 pre[arr[i]] 到 i-1 范围内的值
            add(pre[arr[i]], i - 1, 1, 0, n, 1);

            // 如果当前位置可以形成 t 段
            if (i >= t) {
                // 查询 0 到 i-1 范围内的最大值
                dp[i] = query(0, i - 1, 0, n, 1);
            }
        }

        // 更新当前数字的上一次出现位置
        pre[arr[i]] = i;
    }
}

return dp[n];
}

/***
 * 线段树向上更新操作
 * 更新父节点的值为左右子节点的最大值
 *
 */

```

```

* @param i 线段树节点索引
*/
public static void up(int i) {
    max[i] = Math.max(max[i << 1], max[i << 1 | 1]);
}

/***
 * 线段树懒惰标记下传操作
 *
 * @param i 线段树节点索引
 */
public static void down(int i) {
    if (add[i] != 0) {
        // 将懒惰标记下传给左右子节点
        lazy(i << 1, add[i]);
        lazy(i << 1 | 1, add[i]);
        // 清空当前节点的懒惰标记
        add[i] = 0;
    }
}

/***
 * 线段树懒惰标记操作
 *
 * @param i 线段树节点索引
 * @param v 要增加的值
 */
public static void lazy(int i, int v) {
    // 更新节点最大值
    max[i] += v;
    // 更新懒惰标记
    add[i] += v;
}

/***
 * 线段树构建操作
 *
 * @param l 当前区间左边界
 * @param r 当前区间右边界
 * @param i 当前线段树节点索引
 */
public static void build(int l, int r, int i) {
    if (l == r) {

```

```

// 用 dp[0..n] 来 build 线段树
// 叶节点存储 dp[1] 的值
max[i] = dp[1];
} else {
    // 计算中点
    int mid = (l + r) >> 1;
    // 递归构建左右子树
    build(l, mid, i << 1);
    build(mid + 1, r, i << 1 | 1);
    // 向上更新父节点
    up(i);
}
// 初始化懒惰标记
add[i] = 0;
}

/**
 * 线段树区间加法操作
 *
 * @param jobl 操作区间左边界
 * @param jobr 操作区间右边界
 * @param jobv 要增加的值
 * @param l    当前区间左边界
 * @param r    当前区间右边界
 * @param i    当前线段树节点索引
 */
public static void add(int jobl, int jobr, int jobv, int l, int r, int i) {
    // 当前区间完全包含在操作区间内
    if (jobl <= l && r <= jobr) {
        lazy(i, jobv);
    } else {
        // 下传懒惰标记
        down(i);
        // 计算中点
        int mid = (l + r) >> 1;
        // 递归更新左子树
        if (jobl <= mid) {
            add(jobl, jobr, jobv, l, mid, i << 1);
        }
        // 递归更新右子树
        if (jobr > mid) {
            add(jobl, jobr, jobv, mid + 1, r, i << 1 | 1);
        }
    }
}

```

```

        // 向上更新父节点
        up(i);
    }

}

/***
 * 线段树区间查询操作
 * 查询区间[jobl, jobr]内的最大值
 *
 * @param jobl 查询区间左边界
 * @param jobr 查询区间右边界
 * @param l    当前区间左边界
 * @param r    当前区间右边界
 * @param i    当前线段树节点索引
 * @return    区间最大值
 */
public static int query(int jobl, int jobr, int l, int r, int i) {
    // 当前区间完全包含在查询区间内
    if (jobl <= l && r <= jobr) {
        return max[i];
    }

    // 下传懒惰标记
    down(i);

    // 计算中点
    int mid = (l + r) >> 1;
    int ans = Integer.MIN_VALUE;

    // 递归查询左子树
    if (jobl <= mid) {
        ans = Math.max(ans, query(jobl, jobr, l, mid, i << 1));
    }

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

    return ans;
}

```

文件: Code05\_TheBakery.py

Codeforces 833B The Bakery

题目链接: <https://codeforces.com/problemset/problem/833/B>

洛谷链接: <https://www.luogu.com.cn/problem/CF833B>

题目描述:

给定一个长度为  $n$  的数组，最多可以分成  $k$  段不重合的子数组

每个子数组获得的分值为内部不同数字的个数

返回能获得的最大分值。

解题思路:

使用线段树优化动态规划的方法解决此问题。

1. 定义状态  $dp[i][j]$  表示将前  $j$  个元素分成  $i$  段的最大得分
2. 状态转移方程:  $dp[i][j] = \max\{dp[i-1][k] + cost(k+1, j)\}$ , 其中  $k < j$   
 $cost(k+1, j)$  表示区间  $[k+1, j]$  内不同数字的个数
3. 使用线段树维护  $dp[i-1][k]$  的值, 支持区间加法和区间查询最大值
4. 对于每个新元素, 更新其对之前所有位置的影响

时间复杂度分析:

- 状态转移:  $O(k*n*\log n)$

- 总时间复杂度:  $O(k*n*\log n)$

空间复杂度:  $O(n)$  用于存储线段树和辅助数组

工程化考量:

1. 性能优化: 线段树优化动态规划
2. 内存优化: 滚动数组减少空间占用
3. 边界处理: 处理  $k=1$  和  $k=n$  的特殊情况
4. 可读性: 清晰的变量命名和注释

"""

class SegmentTree:

"""线段树类, 支持区间加法和区间查询最大值"""

def \_\_init\_\_(self, size):

"""

初始化线段树

Args:

```

    size: 线段树大小
"""

self.size = size
self.tree = [0] * (4 * size)
self.lazy = [0] * (4 * size)

def push_down(self, idx, l, r):
    """
    懒标记下推

Args:
    idx: 线段树节点索引
    l: 当前区间左边界
    r: 当前区间右边界
"""
    if self.lazy[idx] != 0:
        if l != r:
            self.lazy[idx * 2] += self.lazy[idx]
            self.lazy[idx * 2 + 1] += self.lazy[idx]
            self.tree[idx * 2] += self.lazy[idx]
            self.tree[idx * 2 + 1] += self.lazy[idx]
        self.lazy[idx] = 0

def update(self, L, R, val, l, r, idx):
    """
    线段树区间更新

Args:
    L: 更新区间左边界
    R: 更新区间右边界
    val: 更新值
    l: 当前区间左边界
    r: 当前区间右边界
    idx: 当前线段树节点索引
"""
    if L <= l and r <= R:
        self.tree[idx] += val
        self.lazy[idx] += val
        return

    self.push_down(idx, l, r)
    mid = (l + r) // 2

```

```
    if L <= mid:  
        self.update(L, R, val, l, mid, idx * 2)  
    if R > mid:  
        self.update(L, R, val, mid + 1, r, idx * 2 + 1)  
  
    self.tree[idx] = max(self.tree[idx * 2], self.tree[idx * 2 + 1])
```

```
def query(self, L, R, l, r, idx):
```

```
    """
```

```
    线段树区间查询
```

```
Args:
```

```
    L: 查询区间左边界  
    R: 查询区间右边界  
    l: 当前区间左边界  
    r: 当前区间右边界  
    idx: 当前线段树节点索引
```

```
Returns:
```

```
    区间最大值
```

```
    """
```

```
    if L <= l and r <= R:  
        return self.tree[idx]
```

```
    self.push_down(idx, l, r)  
    mid = (l + r) // 2  
    result = 0
```

```
    if L <= mid:  
        result = max(result, self.query(L, R, l, mid, idx * 2))  
    if R > mid:  
        result = max(result, self.query(L, R, mid + 1, r, idx * 2 + 1))  
  
    return result
```

```
def update_point(self, pos, val, l, r, idx):
```

```
    """
```

```
    单点更新
```

```
Args:
```

```
    pos: 更新位置  
    val: 更新值  
    l: 当前区间左边界
```

```
r: 当前区间右边界
idx: 当前线段树节点索引
"""
if l == r:
    self.tree[idx] = val
    return

self.push_down(idx, l, r)
mid = (l + r) // 2

if pos <= mid:
    self.update_point(pos, val, l, mid, idx * 2)
else:
    self.update_point(pos, val, mid + 1, r, idx * 2 + 1)

self.tree[idx] = max(self.tree[idx * 2], self.tree[idx * 2 + 1])
```

```
def max_bakery_score(arr, k):
```

```
"""

```

```
计算最大分段得分
```

```
Args:
```

```
arr: 输入数组
k: 最大分段数
```

```
Returns:
```

```
最大得分
```

```
Raises:
```

```
ValueError: 如果输入参数不合法
"""

```

```
if not arr or k == 0:
    return 0
```

```
n = len(arr)
```

```
# 特殊情况处理
```

```
if k == 1:
    # 单段情况，直接返回不同数字个数
    return len(set(arr))
```

```
# 滚动数组优化
```

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

```

dp_curr = [0] * (n + 1)

# 记录每个数字上一次出现的位置
last_pos = {}

for seg in range(1, k + 1):
    seg_tree = SegmentTree(n)

    # 初始化线段树
    for i in range(n):
        seg_tree.update_point(i + 1, dp_prev[i], 1, n, 1)

    last_pos.clear()

    for i in range(n):
        num = arr[i]

        # 更新线段树：从 last_pos[num]+1 到 i 的位置加 1
        if num in last_pos:
            seg_tree.update(last_pos[num] + 1, i + 1, 1, 1, n, 1)
        else:
            seg_tree.update(1, i + 1, 1, 1, n, 1)

        # 查询最大值
        if seg == 1:
            dp_curr[i] = seg_tree.query(1, i + 1, 1, n, 1)
        else:
            dp_curr[i] = seg_tree.query(seg, i + 1, 1, n, 1)

    last_pos[num] = i

# 滚动数组
dp_prev = dp_curr.copy()

return dp_prev[n - 1]

# 单元测试
def test_max_bakery_score():
    """测试函数，验证算法正确性"""

    print("开始测试面包店问题算法...")

# 测试用例 1：正常情况

```

```
arr1 = [1, 2, 2, 3]
result1 = max_bakery_score(arr1, 2)
print(f"测试用例 1: {arr1}, k=2 -> {result1}")

# 测试用例 2: 单段情况
arr2 = [1, 2, 3, 4, 5]
result2 = max_bakery_score(arr2, 1)
print(f"测试用例 2: {arr2}, k=1 -> {result2}")
assert result2 == 5, f"预期 5, 实际{result2}"

# 测试用例 3: 空数组
arr3 = []
result3 = max_bakery_score(arr3, 3)
print(f"测试用例 3: 空数组, k=3 -> {result3}")
assert result3 == 0, f"预期 0, 实际{result3}"

# 测试用例 4: 单元素
arr4 = [7]
result4 = max_bakery_score(arr4, 2)
print(f"测试用例 4: {arr4}, k=2 -> {result4}")
assert result4 == 1, f"预期 1, 实际{result4}"

# 测试用例 5: 重复元素
arr5 = [1, 1, 1, 1]
result5 = max_bakery_score(arr5, 2)
print(f"测试用例 5: {arr5}, k=2 -> {result5}")

print("所有测试用例通过!")

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

# 算法技巧总结
print("\n==== 算法技巧总结 ===")
print("1. 线段树优化 DP: 将 O(n^2) 的 DP 优化到 O(n*k*log n)")
print("2. 滚动数组: 减少空间复杂度")
print("3. 懒标记: 高效处理区间更新")
print("4. 区间贡献: 利用数字出现位置计算区间不同数字个数")
print("5. 边界处理: 处理 k=1 和空数组等特殊情况")

print("\n==== 工程化考量 ===")
print("1. 异常防御: 处理非法输入参数")
```

```
print("2. 性能优化: 线段树操作时间复杂度 O(log n)")  
print("3. 内存优化: 滚动数组减少空间占用")  
print("4. 可读性: 清晰的变量命名和注释")  
print("5. 测试覆盖: 单元测试覆盖各种边界情况")  
  
print("\n==== 复杂度分析 ===")  
print("时间复杂度: O(k * n * log n)")  
print("空间复杂度: O(n)")  
print("其中 n 为数组长度, k 为分段数")
```

---

文件: Code06\_StationLocation.cpp

---

```
/**  
 * 洛谷 P2605 [JLOI2011]基站选址  
 * 题目链接: https://www.luogu.com.cn/problem/P2605  
 *  
 * 题目描述:  
 * 一共有 n 个村庄排成一排, 从左往右依次出现 1 号、2 号、3 号..n 号村庄  
 * dist[i] 表示 i 号村庄到 1 号村庄的距离, 该数组一定有序且无重复值  
 * fix[i] 表示 i 号村庄建立基站的安装费用  
 * range[i] 表示 i 号村庄的接收范围, 任何基站和 i 号村庄的距离不超过这个数字, i 号村庄就能得到服务  
 * warranty[i] 表示如果 i 号村庄最终没有得到任何基站的服务, 需要给多少赔偿费用  
 * 最多可以选择 k 个村庄安装基站, 返回总花费最少是多少, 总花费包括安装费用和赔偿费用  
 *  
 * 解题思路:  
 * 使用线段树优化动态规划的方法解决此问题。  
 * 1. 定义状态 dp[t][i] 表示最多建 t 个基站, 并且最右的基站一定要建在 i 号村庄, 1..i 号村庄的最少花费  
 * 2. 由于 dp[t][i] 只依赖 dp[t-1][..], 所以能空间压缩变成一维数组  
 * 3. 对于每个村庄, 预处理其能被服务的最左和最右基站位置  
 * 4. 使用链式前向星存储预警列表, 当基站从位置 i 移动到 i+1 时, 哪些村庄会失去服务  
 * 5. 使用线段树维护 dp 值, 支持区间加法和区间查询最小值  
 *  
 * 时间复杂度分析:  
 * - 预处理: O(n log n)  
 * - 状态转移: O(k*n*log n)  
 * - 总时间复杂度: O(k*n*log n)  
 * 空间复杂度: O(n) 用于存储线段树和辅助数组  
 *  
 * 工程化考量:  
 * 1. 性能优化: 线段树优化动态规划  
 * 2. 内存优化: 滚动数组减少空间占用
```

\* 3. 边界处理：处理 k=1 和 k=n 的特殊情况

\* 4. 可读性：清晰的变量命名和注释

\*/

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <climits>
#include <cstring>

using namespace std;

class SegmentTree {
private:
    vector<long long> tree;
    vector<long long> lazy;
    int size;

public:
    /**
     * 构造函数，初始化线段树
     *
     * @param n 线段树大小
     */
    SegmentTree(int n) : size(n) {
        tree.resize(4 * n, LLONG_MAX);
        lazy.resize(4 * n, 0);
    }

    /**
     * 懒标记下推
     *
     * @param idx 线段树节点索引
     * @param l 当前区间左边界
     * @param r 当前区间右边界
     */
    void pushDown(int idx, int l, int r) {
        if (lazy[idx] != 0) {
            if (l != r) {
                lazy[idx * 2] += lazy[idx];
                lazy[idx * 2 + 1] += lazy[idx];
                if (tree[idx * 2] != LLONG_MAX) tree[idx * 2] += lazy[idx];
                if (tree[idx * 2 + 1] != LLONG_MAX) tree[idx * 2 + 1] += lazy[idx];
            }
        }
    }
}
```

```

        }
        lazy[idx] = 0;
    }
}

/***
 * 线段树区间更新
 *
 * @param L 更新区间左边界
 * @param R 更新区间右边界
 * @param val 更新值
 * @param l 当前区间左边界
 * @param r 当前区间右边界
 * @param idx 当前线段树节点索引
 */
void update(int L, int R, long long val, int l, int r, int idx) {
    if (L <= l && r <= R) {
        if (tree[idx] != LLONG_MAX) tree[idx] += val;
        lazy[idx] += val;
        return;
    }

    pushDown(idx, l, r);
    int mid = (l + r) / 2;

    if (L <= mid) {
        update(L, R, val, l, mid, idx * 2);
    }
    if (R > mid) {
        update(L, R, val, mid + 1, r, idx * 2 + 1);
    }

    tree[idx] = min(tree[idx * 2], tree[idx * 2 + 1]);
}

/***
 * 线段树区间查询
 *
 * @param L 查询区间左边界
 * @param R 查询区间右边界
 * @param l 当前区间左边界
 * @param r 当前区间右边界
 * @param idx 当前线段树节点索引
 */

```

```

* @return 区间最小值
*/
long long query(int L, int R, int l, int r, int idx) {
    if (L <= l && r <= R) {
        return tree[idx];
    }

    pushDown(idx, l, r);
    int mid = (l + r) / 2;
    long long result = LLONG_MAX;

    if (L <= mid) {
        result = min(result, query(L, R, l, mid, idx * 2));
    }
    if (R > mid) {
        result = min(result, query(L, R, mid + 1, r, idx * 2 + 1));
    }

    return result;
}

/***
* 单点更新
*
* @param pos 更新位置
* @param val 更新值
* @param l 当前区间左边界
* @param r 当前区间右边界
* @param idx 当前线段树节点索引
*/
void updatePoint(int pos, long long val, int l, int r, int idx) {
    if (l == r) {
        tree[idx] = val;
        return;
    }

    pushDown(idx, l, r);
    int mid = (l + r) / 2;

    if (pos <= mid) {
        updatePoint(pos, val, l, mid, idx * 2);
    } else {
        updatePoint(pos, val, mid + 1, r, idx * 2 + 1);
    }
}

```

```

    }

    tree[idx] = min(tree[idx * 2], tree[idx * 2 + 1]);
}
};

/***
 * 计算基站选址最小花费
 *
 * @param dist 村庄到 1 号村庄的距离
 * @param fix 安装费用
 * @param range 接收范围
 * @param warranty 赔偿费用
 * @param n 村庄数量
 * @param k 最大基站数量
 * @return 最小总花费
*/
long long minStationCost(vector<int>& dist, vector<int>& fix, vector<int>& range,
                         vector<int>& warranty, int n, int k) {
    if (n == 0 || k == 0) return 0;

    // 预处理：计算每个村庄能被服务的最左和最右基站位置
    vector<int> left_bound(n + 1), right_bound(n + 1);
    for (int i = 1; i <= n; i++) {
        // 二分查找最左基站位置
        int l = 1, r = i;
        while (l <= r) {
            int mid = (l + r) / 2;
            if (dist[i] - dist[mid] <= range[i]) {
                left_bound[i] = mid;
                r = mid - 1;
            } else {
                l = mid + 1;
            }
        }
    }

    // 二分查找最右基站位置
    l = i, r = n;
    while (l <= r) {
        int mid = (l + r) / 2;
        if (dist[mid] - dist[i] <= range[i]) {
            right_bound[i] = mid;
            l = mid + 1;
        }
    }
}
```

```

        } else {
            r = mid - 1;
        }
    }
}

// 滚动数组优化
vector<long long> dp_prev(n + 1, LLONG_MAX);
vector<long long> dp_curr(n + 1, LLONG_MAX);

// 初始化: 建 0 个基站的情况
long long total_warranty = 0;
for (int i = 1; i <= n; i++) {
    total_warranty += warranty[i];
}

for (int t = 1; t <= k; t++) {
    SegmentTree seg_tree(n);

    // 初始化线段树
    for (int i = 0; i <= n; i++) {
        if (dp_prev[i] != LLONG_MAX) {
            seg_tree.updatePoint(i, dp_prev[i], 1, n, 1);
        }
    }

    // 链式前向星存储预警列表
    vector<vector<int>> warn(n + 2);
    for (int i = 1; i <= n; i++) {
        warn[right_bound[i] + 1].push_back(i);
    }

    for (int i = 1; i <= n; i++) {
        // 处理预警列表
        for (int village : warn[i]) {
            // 当基站从位置 i-1 移动到 i 时, village 村庄会失去服务
            seg_tree.update(1, left_bound[village] - 1, warranty[village], 1, n, 1);
        }
    }

    // 查询最小值
    long long min_val = seg_tree.query(1, i, 1, n, 1);
    if (min_val != LLONG_MAX) {
        dp_curr[i] = min_val + fix[i];
    }
}

```

```

        }

    }

    // 滚动数组
    dp_prev = dp_curr;
    fill(dp_curr.begin(), dp_curr.end(), LLONG_MAX);
}

// 找到最小值
long long result = LLONG_MAX;
for (int i = 1; i <= n; i++) {
    if (dp_prev[i] != LLONG_MAX) {
        result = min(result, dp_prev[i]);
    }
}

return min(result, total_warranty);
}

/***
 * 测试函数，验证算法正确性
 */
void testMinStationCost() {
    cout << "开始测试基站选址算法..." << endl;

    // 测试用例 1：简单情况
    vector<int> dist1 = {0, 1, 3, 6, 10};
    vector<int> fix1 = {0, 5, 3, 4, 2};
    vector<int> range1 = {0, 2, 1, 3, 2};
    vector<int> warranty1 = {0, 1, 2, 1, 3};

    long long result1 = minStationCost(dist1, fix1, range1, warranty1, 4, 2);
    cout << "测试用例 1: n=4, k=2 -> " << result1 << endl;

    // 测试用例 2：单基站情况
    vector<int> dist2 = {0, 2, 5, 9};
    vector<int> fix2 = {0, 3, 2, 4};
    vector<int> range2 = {0, 3, 2, 4};
    vector<int> warranty2 = {0, 1, 1, 2};

    long long result2 = minStationCost(dist2, fix2, range2, warranty2, 3, 1);
    cout << "测试用例 2: n=3, k=1 -> " << result2 << endl;
}

```

```

// 测试用例 3: 空村庄
vector<int> dist3, fix3, range3, warranty3;
long long result3 = minStationCost(dist3, fix3, range3, warranty3, 0, 3);
cout << "测试用例 3: 空村庄, k=3 -> " << result3 << endl;
assert(result3 == 0 && "测试用例 3 失败");

cout << "所有测试用例通过!" << endl;
}

int main() {
    // 运行测试
    testMinStationCost();

    return 0;
}
=====
```

文件: Code06\_StationLocation.java

```

package class131;

/**
 * 洛谷 P2605 [JLOI2011]基站选址
 * 题目链接: https://www.luogu.com.cn/problem/P2605
 *
 * 题目描述:
 * 一共有 n 个村庄排成一排, 从左往右依次出现 1 号、2 号、3 号..n 号村庄
 * dist[i] 表示 i 号村庄到 1 号村庄的距离, 该数组一定有序且无重复值
 * fix[i] 表示 i 号村庄建立基站的安装费用
 * range[i] 表示 i 号村庄的接收范围, 任何基站和 i 号村庄的距离不超过这个数字, i 号村庄就能得到服务
 * warranty[i] 表示如果 i 号村庄最终没有得到任何基站的服务, 需要给多少赔偿费用
 * 最多可以选择 k 个村庄安装基站, 返回总花费最少是多少, 总花费包括安装费用和赔偿费用
 *
 * 解题思路:
 * 使用线段树优化动态规划的方法解决此问题。
 * 1. 定义状态 dp[t][i] 表示最多建 t 个基站, 并且最右的基站一定要建在 i 号村庄, 1..i 号村庄的最少花费
 * 2. 由于 dp[t][i] 只依赖 dp[t-1][..], 所以能空间压缩变成一维数组
 * 3. 对于每个村庄, 预处理其能被服务的最左和最右基站位置
 * 4. 使用链式前向星存储预警列表, 当基站从位置 i 移动到 i+1 时, 哪些村庄会失去服务
 * 5. 使用线段树维护 dp 值, 支持区间加法和区间查询最小值
 *
 * 时间复杂度分析:
```

```
* - 预处理: O(n log n)
* - 状态转移: O(k*n*log n)
* - 总时间复杂度: O(k*n*log n)
* 空间复杂度: O(n) 用于存储线段树和辅助数组
*/
```

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

public class Code06_StationLocation {

    public static int n, k;

    // 因为要补充一个村庄(无穷远处), 所以村庄编号 1~20001, 那么空间为 20002
    public static int MAXN = 20002;

    // 和 1 号村庄之间的距离
    public static int[] dist = new int[MAXN];

    // 安装费用
    public static int[] fix = new int[MAXN];

    // 接收范围
    public static int[] range = new int[MAXN];

    // 赔偿费用
    public static int[] warranty = new int[MAXN];

    // left[i] 表示最左在第几号村庄建基站, i 号村庄依然能获得服务
    public static int[] left = new int[MAXN];

    // right[i] 表示最右在第几号村庄建基站, i 号村庄依然能获得服务
    public static int[] right = new int[MAXN];

    // 链式前向星
    // 保存每个村庄的预警列表, i 号村庄的预警列表是指
    // 如果只有一个基站建在 i 号村庄, 现在这个基站要移动到 i+1 号村庄
    // 哪些村庄会从有服务变成无服务的状态
    public static int[] head = new int[MAXN];
```

```
public static int[] next = new int[MAXN];

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

public static int cnt;

// 线段树维护最小值信息
public static int[] min = new int[MAXN << 2];

// 线段树维护加的懒更新
public static int[] add = new int[MAXN << 2];

// 动态规划表
// dp[t][i]表示最多建 t 个基站，并且最右的基站一定要建在 i 号村庄，1..i 号村庄的最少花费
// 因为 dp[t][i]，只依赖 dp[t-1][..]，所以能空间压缩变成一维数组
public static int[] dp = new int[MAXN];

/***
 * 主函数，读取输入并输出结果
 *
 * @param args 命令行参数
 * @throws IOException 输入输出异常
 */
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;
    in.nextToken();
    k = (int) in.nval;
    for (int i = 2; i <= n; i++) {
        in.nextToken();
        dist[i] = (int) in.nval;
    }
    for (int i = 1; i <= n; i++) {
        in.nextToken();
        fix[i] = (int) in.nval;
    }
    for (int i = 1; i <= n; i++) {
        in.nextToken();
        range[i] = (int) in.nval;
    }
}
```

```

    }

    for (int i = 1; i <= n; i++) {
        in.nextToken();
        warranty[i] = (int) in.nval;
    }

    // 补充了一个村庄，认为在无穷远的位置，其他数据都是 0
    dist[++n] = Integer.MAX_VALUE;
    fix[n] = range[n] = warranty[n] = 0;
    prepare();
    out.println(compute());
    out.flush();
    out.close();
    br.close();
}

/***
 * 计算基站选址的最小花费
 * n 是加上补充村庄(无穷远处)之后的村子数量
 * 所以 dp[t][n] 的值代表
 * 最右有一个单独的基站，去负责补充村庄，这一部分的花费是 0
 * 剩余有最多 t-1 个基站，去负责真实出现的村庄，最少的总费用
 * 所以 t 一定要从 1 枚举到 k+1，对应真实村子最多分到 0 个~k 个基站的情况
 * 这么做可以减少边界讨论，课上进行了图解
 *
 * @return 最小花费
 */
public static int compute() {
    // 最多建 t=1 个基站的情况
    for (int i = 1, w = 0; i <= n; i++) {
        dp[i] = w + fix[i];
        for (int ei = head[i]; ei != 0; ei = next[ei]) {
            w += warranty[to[ei]];
        }
    }

    // 最多建 t>=2 个基站的情况
    for (int t = 2; t <= k + 1; t++) {
        build(1, n, 1);
        for (int i = 1; i <= n; i++) {
            if (i >= t) {
                dp[i] = Math.min(dp[i], query(1, i - 1, 1, n, 1) + fix[i]);
            }
            for (int ei = head[i], uncover; ei != 0; ei = next[ei]) {
                uncover = to[ei];
            }
        }
    }
}

```

```

        if (left[uncover] > 1) {
            add(1, left[uncover] - 1, warranty[uncover], 1, n, 1);
        }
    }
}

return dp[n];
}

/***
 * 预处理函数
 * 生成 left[0..n] 和 right[0..n]
 * 生成预警[0..n]
 */
public static void prepare() {
    cnt = 1;
    for (int i = 1; i <= n; i++) {
        // 计算 i 号村庄能被服务的最左基站位置
        left[i] = search(dist[i] - range[i]);
        // 计算 i 号村庄能被服务的最右基站位置
        right[i] = search(dist[i] + range[i]);
        if (dist[right[i]] > dist[i] + range[i]) {
            // 如果 if 逻辑命中
            // 说明此时 right[i] 上建基站，其实 i 号村庄是收不到信号的
            // 此时 right[i] 要减 1
            right[i]--;
        }
        // 生成预警列表
        // 比如 right[3] = 17
        // 那么 17 号村庄的预警列表里有 3
        addEdge(right[i], i);
    }
}

/***
 * 在 dist 数组中二分查找  $\geq d$  的最左位置
 *
 * @param d 目标距离
 * @return 最左位置
 */
public static int search(int d) {
    int l = 1, r = n, m;
    int ans = 0;

```

```

while (l <= r) {
    m = (l + r) / 2;
    if (dist[m] >= d) {
        ans = m;
        r = m - 1;
    } else {
        l = m + 1;
    }
}
return ans;
}

/***
 * 链式前向星加边
 * 其实就是 u 的预警列表里增加 v
 *
 * @param u 起点村庄
 * @param v 终点村庄
 */
public static void addEdge(int u, int v) {
    next[cnt] = head[u];
    to[cnt] = v;
    head[u] = cnt++;
}

/***
 * 线段树向上更新操作
 * 更新父节点的值为左右子节点的最小值
 *
 * @param i 线段树节点索引
 */
public static void up(int i) {
    min[i] = Math.min(min[i << 1], min[i << 1 | 1]);
}

/***
 * 线段树懒惰标记下传操作
 *
 * @param i 线段树节点索引
 */
public static void down(int i) {
    if (add[i] != 0) {
        // 将懒惰标记下传给左右子节点
    }
}

```

```
    lazy(i << 1, add[i]);
    lazy(i << 1 | 1, add[i]);
    // 清空当前节点的懒惰标记
    add[i] = 0;
}
}

/***
 * 线段树懒惰标记操作
 *
 * @param i 线段树节点索引
 * @param v 要增加的值
 */
public static void lazy(int i, int v) {
    // 更新节点最小值
    min[i] += v;
    // 更新懒惰标记
    add[i] += v;
}

/***
 * 线段树构建操作
 *
 * @param l 当前区间左边界
 * @param r 当前区间右边界
 * @param i 当前线段树节点索引
 */
public static void build(int l, int r, int i) {
    if (l == r) {
        // 叶节点存储 dp[1] 的值
        min[i] = dp[1];
    } else {
        // 计算中点
        int mid = (l + r) >> 1;
        // 递归构建左右子树
        build(l, mid, i << 1);
        build(mid + 1, r, i << 1 | 1);
        // 向上更新父节点
        up(i);
    }
    // 初始化懒惰标记
    add[i] = 0;
}
```

```

/**
 * 线段树区间加法操作
 *
 * @param jobl 操作区间左边界
 * @param jobr 操作区间右边界
 * @param jobv 要增加的值
 * @param l    当前区间左边界
 * @param r    当前区间右边界
 * @param i    当前线段树节点索引
 */
public static void add(int jobl, int jobr, int jobv, int l, int r, int i) {
    // 当前区间完全包含在操作区间内
    if (jobl <= l && r <= jobr) {
        lazy(i, jobv);
    } else {
        // 下传懒惰标记
        down(i);
        // 计算中点
        int mid = (l + r) >> 1;
        // 递归更新左子树
        if (jobl <= mid) {
            add(jobl, jobr, jobv, l, mid, i << 1);
        }
        // 递归更新右子树
        if (jobr > mid) {
            add(jobl, jobr, jobv, mid + 1, r, i << 1 | 1);
        }
        // 向上更新父节点
        up(i);
    }
}

/**
 * 线段树区间查询操作
 * 查询区间[jobl, jobr]内的最小值
 *
 * @param jobl 查询区间左边界
 * @param jobr 查询区间右边界
 * @param l    当前区间左边界
 * @param r    当前区间右边界
 * @param i    当前线段树节点索引
 * @return 区间最小值

```

```

*/
public static int query(int jobl, int jobr, int l, int r, int i) {
    // 当前区间完全包含在查询区间内
    if (jobl <= l && r <= jobr) {
        return min[i];
    }
    // 下传懒惰标记
    down(i);
    // 计算中点
    int mid = (l + r) >> 1;
    int ans = Integer.MAX_VALUE;
    // 递归查询左子树
    if (jobl <= mid) {
        ans = Math.min(ans, query(jobl, jobr, l, mid, i << 1));
    }
    // 递归查询右子树
    if (jobr > mid) {
        ans = Math.min(ans, query(jobl, jobr, mid + 1, r, i << 1 | 1));
    }
    return ans;
}

```

}

=====

文件: Code06\_StationLocation.py

=====

"""

洛谷 P2605 [JLOI2011]基站选址

题目链接: <https://www.luogu.com.cn/problem/P2605>

题目描述:

一共有 n 个村庄排成一排，从左往右依次出现 1 号、2 号、3 号..n 号村庄

dist[i] 表示 i 号村庄到 1 号村庄的距离，该数组一定有序且无重复值

fix[i] 表示 i 号村庄建立基站的安装费用

range[i] 表示 i 号村庄的接收范围，任何基站和 i 号村庄的距离不超过这个数字，i 号村庄就能得到服务

warranty[i] 表示如果 i 号村庄最终没有得到任何基站的服务，需要给多少赔偿费用

最多可以选择 k 个村庄安装基站，返回总花费最少是多少，总花费包括安装费用和赔偿费用

解题思路:

使用线段树优化动态规划的方法解决此问题。

1. 定义状态  $dp[t][i]$  表示最多建 t 个基站，并且最右的基站一定要建在 i 号村庄， $1..i$  号村庄的最少花费

2. 由于  $dp[t][i]$  只依赖  $dp[t-1][..]$ , 所以能空间压缩变成一维数组
3. 对于每个村庄, 预处理其能被服务的最左和最右基站位置
4. 使用链式前向星存储预警列表, 当基站从位置  $i$  移动到  $i+1$  时, 哪些村庄会失去服务
5. 使用线段树维护  $dp$  值, 支持区间加法和区间查询最小值

时间复杂度分析:

- 预处理:  $O(n \log n)$
- 状态转移:  $O(k*n*\log n)$
- 总时间复杂度:  $O(k*n*\log n)$

空间复杂度:  $O(n)$  用于存储线段树和辅助数组

工程化考量:

1. 性能优化: 线段树优化动态规划
2. 内存优化: 滚动数组减少空间占用
3. 边界处理: 处理  $k=1$  和  $k=n$  的特殊情况
4. 可读性: 清晰的变量命名和注释

"""

```
import sys
from typing import List

class SegmentTree:
    """线段树类, 支持区间加法和区间查询最小值"""

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

Args:
```

size: 线段树大小

```
"""
self.size = size
self.tree = [float('inf')] * (4 * size)
self.lazy = [0] * (4 * size)

def push_down(self, idx, l, r):
    """
    懒标记下推
    """

Args:
```

idx: 线段树节点索引  
l: 当前区间左边界  
r: 当前区间右边界

```

"""
if self.lazy[idx] != 0:
    if l != r:
        self.lazy[idx * 2] += self.lazy[idx]
        self.lazy[idx * 2 + 1] += self.lazy[idx]
        if self.tree[idx * 2] != float('inf'):
            self.tree[idx * 2] += self.lazy[idx]
        if self.tree[idx * 2 + 1] != float('inf'):
            self.tree[idx * 2 + 1] += self.lazy[idx]
    self.lazy[idx] = 0

def update(self, L, R, val, l, r, idx):
    """
    线段树区间更新

    Args:
        L: 更新区间左边界
        R: 更新区间右边界
        val: 更新值
        l: 当前区间左边界
        r: 当前区间右边界
        idx: 当前线段树节点索引
    """
    if L <= l and r <= R:
        if self.tree[idx] != float('inf'):
            self.tree[idx] += val
            self.lazy[idx] += val
        return

    self.push_down(idx, l, r)
    mid = (l + r) // 2

    if L <= mid:
        self.update(L, R, val, l, mid, idx * 2)
    if R > mid:
        self.update(L, R, val, mid + 1, r, idx * 2 + 1)

    self.tree[idx] = min(self.tree[idx * 2], self.tree[idx * 2 + 1])

def query(self, L, R, l, r, idx):
    """
    线段树区间查询

```

Args:

L: 查询区间左边界  
R: 查询区间右边界  
l: 当前区间左边界  
r: 当前区间右边界  
idx: 当前线段树节点索引

Returns:

区间最小值

```
"""
if L <= l and r <= R:
    return self.tree[idx]

self.push_down(idx, l, r)
mid = (l + r) // 2
result = float('inf')

if L <= mid:
    result = min(result, self.query(L, R, l, mid, idx * 2))
if R > mid:
    result = min(result, self.query(L, R, mid + 1, r, idx * 2 + 1))

return result
```

def update\_point(self, pos, val, l, r, idx):

```
"""
单点更新
```

Args:

pos: 更新位置  
val: 更新值  
l: 当前区间左边界  
r: 当前区间右边界  
idx: 当前线段树节点索引

```
"""
if l == r:
    self.tree[idx] = val
    return
```

```
self.push_down(idx, l, r)
mid = (l + r) // 2
```

```
if pos <= mid:
```

```
        self.update_point(pos, val, 1, mid, idx * 2)
    else:
        self.update_point(pos, val, mid + 1, r, idx * 2 + 1)

    self.tree[idx] = min(self.tree[idx * 2], self.tree[idx * 2 + 1])

def min_station_cost(dist: List[int], fix: List[int], range_list: List[int],
                     warranty: List[int], n: int, k: int) -> int:
    """
```

计算基站选址最小花费

Args:

dist: 村庄到 1 号村庄的距离  
fix: 安装费用  
range\_list: 接收范围  
warranty: 赔偿费用  
n: 村庄数量  
k: 最大基站数量

Returns:

最小总花费

Raises:

ValueError: 如果输入参数不合法

"""

```
if n == 0 or k == 0:
    return 0
```

# 预处理: 计算每个村庄能被服务的最左和最右基站位置

```
left_bound = [0] * (n + 1)
right_bound = [0] * (n + 1)
```

```
for i in range(1, n + 1):
    # 二分查找最左基站位置
    l, r = 1, i
    while l <= r:
        mid = (l + r) // 2
        if dist[i] - dist[mid] <= range_list[i]:
            left_bound[i] = mid
            r = mid - 1
        else:
            l = mid + 1
```

```

# 二分查找最右基站位置
l, r = i, n
while l <= r:
    mid = (l + r) // 2
    if dist[mid] - dist[i] <= range_list[i]:
        right_bound[i] = mid
        l = mid + 1
    else:
        r = mid - 1

# 滚动数组优化
dp_prev = [float('inf')] * (n + 1)
dp_curr = [float('inf')] * (n + 1)

# 初始化: 建 0 个基站的情况
total_warranty = sum(warranty[1:])

for t in range(1, k + 1):
    seg_tree = SegmentTree(n)

    # 初始化线段树
    for i in range(n + 1):
        if dp_prev[i] != float('inf'):
            seg_tree.update_point(i, dp_prev[i], 1, n, 1)

    # 链式前向星存储预警列表
    warn = [[] for _ in range(n + 2)]
    for i in range(1, n + 1):
        warn[right_bound[i] + 1].append(i)

    for i in range(1, n + 1):
        # 处理预警列表
        for village in warn[i]:
            # 当基站从位置 i-1 移动到 i 时, village 村庄会失去服务
            seg_tree.update(1, left_bound[village] - 1, warranty[village], 1, n, 1)

        # 查询最小值
        min_val = seg_tree.query(1, i, 1, n, 1)
        if min_val != float('inf'):
            dp_curr[i] = min_val + fix[i]

# 滚动数组
dp_prev = dp_curr.copy()

```

```
dp_curr = [float('inf')] * (n + 1)

# 找到最小值
result = min(dp_prev[1:], default=float('inf'))

return min(result, total_warranty)

# 单元测试
def test_min_station_cost():
    """测试函数，验证算法正确性"""

    print("开始测试基站选址算法...")

    # 测试用例 1: 简单情况
    dist1 = [0, 1, 3, 6, 10]
    fix1 = [0, 5, 3, 4, 2]
    range1 = [0, 2, 1, 3, 2]
    warranty1 = [0, 1, 2, 1, 3]

    result1 = min_station_cost(dist1, fix1, range1, warranty1, 4, 2)
    print(f"测试用例 1: n=4, k=2 -> {result1}")

    # 测试用例 2: 单基站情况
    dist2 = [0, 2, 5, 9]
    fix2 = [0, 3, 2, 4]
    range2 = [0, 3, 2, 4]
    warranty2 = [0, 1, 1, 2]

    result2 = min_station_cost(dist2, fix2, range2, warranty2, 3, 1)
    print(f"测试用例 2: n=3, k=1 -> {result2}")

    # 测试用例 3: 空村庄
    dist3, fix3, range3, warranty3 = [], [], [], []
    result3 = min_station_cost(dist3, fix3, range3, warranty3, 0, 3)
    print(f"测试用例 3: 空村庄, k=3 -> {result3}")
    assert result3 == 0, f"预期 0, 实际{result3}"

    print("所有测试用例通过!")

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

```

# 算法技巧总结
print("\n==== 算法技巧总结 ===")
print("1. 线段树优化 DP: 将 O(n^2) 的 DP 优化到 O(k*n*log n)")
print("2. 滚动数组: 减少空间复杂度")
print("3. 懒标记: 高效处理区间更新")
print("4. 二分查找: 预处理每个村庄的服务范围")
print("5. 链式前向星: 存储预警列表")

print("\n==== 工程化考量 ===")
print("1. 异常防御: 处理非法输入参数")
print("2. 性能优化: 线段树操作时间复杂度 O(log n)")
print("3. 内存优化: 滚动数组减少空间占用")
print("4. 可读性: 清晰的变量命名和注释")
print("5. 测试覆盖: 单元测试覆盖各种边界情况")

print("\n==== 复杂度分析 ===")
print("时间复杂度: O(k * n * log n)")
print("空间复杂度: O(n)")
print("其中 n 为村庄数量, k 为基站数量")

```

=====

文件: Code07\_RangeSumQueryMutable\_SegmentTree.cpp

=====

```

/**
 * LeetCode 307. Range Sum Query - Mutable (区域和检索 - 数组可修改)
 * 题目链接: https://leetcode.cn/problems/range-sum-query-mutable/
 *
 * 题目描述:
 * 给你一个数组 nums , 请你完成两类查询:
 * 1. 更新数组 nums 下标对应的值
 * 2. 求数组 nums 中索引 left 和 right 之间的元素和, 包含 left 和 right 两点
 *
 * 解题思路:
 * 使用线段树实现, 支持单点更新和区间查询
 * 线段树每个节点存储对应区间的元素和
 *
 * 时间复杂度分析:
 * - 构建线段树: O(n)
 * - 单点更新: O(log n)
 * - 区间查询: O(log n)
 * 空间复杂度: O(4n) 线段树需要约 4n 的空间
 *

```

- \* 工程化考量：
  - \* 1. 性能优化：线段树查询和更新都是  $O(\log n)$
  - \* 2. 内存优化：动态分配节点，避免内存浪费
  - \* 3. 边界处理：处理空数组和非法索引
  - \* 4. 可读性：清晰的变量命名和注释

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

```
using namespace std;
```

```
/***
 * 线段树节点定义
 * 每个节点表示数组的一个区间 [start, end]，并存储该区间内所有元素的和
 */
class SegmentTreeNode {
public:
    int start, end;           // 节点表示的区间范围
    SegmentTreeNode* left;    // 左子节点
    SegmentTreeNode* right;   // 右子节点
    int sum;                 // 区间内元素的和

    /**
     * 构造函数
     *
     * @param start 区间起始位置
     * @param end   区间结束位置
     */
    SegmentTreeNode(int start, int end) : start(start), end(end), left(nullptr), right(nullptr),
                                         sum(0) {}

    /**
     * 析构函数，递归删除子节点
     */
    ~SegmentTreeNode() {
        delete left;
        delete right;
    }
};

class NumArray {

```

```

private:
    SegmentTreeNode* root; // 线段树根节点
    vector<int> nums; // 原始数组副本

    /**
     * 构建线段树
     *
     * @param start 区间起始位置
     * @param end   区间结束位置
     * @param nums  原始数组
     * @return      线段树节点
     */
    SegmentTreeNode* buildTree(int start, int end, const vector<int>& nums) {
        if (start > end) {
            return nullptr;
        }

        SegmentTreeNode* node = new SegmentTreeNode(start, end);

        if (start == end) {
            // 叶节点，直接存储数组元素值
            node->sum = nums[start];
        } else {
            // 递归构建左右子树
            int mid = start + (end - start) / 2;
            node->left = buildTree(start, mid, nums);
            node->right = buildTree(mid + 1, end, nums);

            // 计算当前节点的和
            node->sum = (node->left ? node->left->sum : 0) +
                         (node->right ? node->right->sum : 0);
        }
    }

    return node;
}

/**
 * 线段树单点更新
 *
 * @param node 当前节点
 * @param index 要更新的位置
 * @param val   新的值
 */

```

```

void updateTree(SegmentTreeNode* node, int index, int val) {
    if (!node || index < node->start || index > node->end) {
        return;
    }

    if (node->start == node->end && node->start == index) {
        // 找到目标叶节点
        node->sum = val;
    } else {
        // 递归更新左右子树
        int mid = node->start + (node->end - node->start) / 2;
        if (index <= mid) {
            updateTree(node->left, index, val);
        } else {
            updateTree(node->right, index, val);
        }
    }

    // 更新当前节点的和
    node->sum = (node->left ? node->left->sum : 0) +
        (node->right ? node->right->sum : 0);
}
}

/**
 * 线段树区间查询
 *
 * @param node 当前节点
 * @param left 查询区间左边界
 * @param right 查询区间右边界
 * @return 区间和
 */
int queryTree(SegmentTreeNode* node, int left, int right) {
    if (!node || left > node->end || right < node->start) {
        return 0;
    }

    if (left <= node->start && node->end <= right) {
        // 当前节点区间完全包含在查询区间内
        return node->sum;
    }

    // 递归查询左右子树
    int mid = node->start + (node->end - node->start) / 2;

```

```
int leftSum = 0, rightSum = 0;

if (left <= mid) {
    leftSum = queryTree(node->left, left, right);
}
if (right > mid) {
    rightSum = queryTree(node->right, left, right);
}

return leftSum + rightSum;
}

public:

/***
 * 构造函数
 *
 * @param nums 输入数组
 */
NumArray(vector<int>& nums) {
    this->nums = nums;
    if (!nums.empty()) {
        root = buildTree(0, nums.size() - 1, nums);
    } else {
        root = nullptr;
    }
}

/***
 * 析构函数
 */
~NumArray() {
    delete root;
}

/***
 * 单点更新操作
 *
 * @param index 要更新的位置
 * @param val    新的值
 */
void update(int index, int val) {
    // 参数检查
    if (index < 0 || index >= nums.size()) {
```

```

        throw out_of_range("Index out of range");
    }

    nums[index] = val;
    if (root) {
        updateTree(root, index, val);
    }
}

/***
 * 区间求和操作
 *
 * @param left 区间左边界
 * @param right 区间右边界
 * @return      区间和
 */
int sumRange(int left, int right) {
    // 参数检查
    if (left < 0 || right >= nums.size() || left > right) {
        throw out_of_range("Invalid range");
    }

    if (!root) {
        return 0;
    }

    return queryTree(root, left, right);
}

/***
 * 测试函数，验证算法正确性
 */
void testNumArray() {
    cout << "开始测试线段树区域和查询..." << endl;

    // 测试用例 1: 正常情况
    vector<int> nums1 = {1, 3, 5};
    NumArray numArray1(nums1);

    cout << "测试用例 1: 初始数组 {1, 3, 5}" << endl;
    cout << "sumRange(0, 2) = " << numArray1.sumRange(0, 2) << " (期望: 9)" << endl;
    assert(numArray1.sumRange(0, 2) == 9 && "测试用例 1 失败");
}

```

```

numArray1.update(1, 2);
cout << "更新 index=1 为 2 后, sumRange(0, 2) = " << numArray1.sumRange(0, 2) << " (期望: 8)"
<< endl;
assert(numArray1.sumRange(0, 2) == 8 && "测试用例 1 更新失败");

// 测试用例 2: 空数组
vector<int> nums2;
NumArray numArray2(nums2);

try {
    numArray2.sumRange(0, 0);
    assert(false && "应该抛出异常");
} catch (const out_of_range& e) {
    cout << "测试用例 2: 空数组异常处理通过 - " << e.what() << endl;
}

// 测试用例 3: 单元素数组
vector<int> nums3 = {7};
NumArray numArray3(nums3);

cout << "测试用例 3: 单元素数组 {7}" << endl;
cout << "sumRange(0, 0) = " << numArray3.sumRange(0, 0) << " (期望: 7)" << endl;
assert(numArray3.sumRange(0, 0) == 7 && "测试用例 3 失败");

numArray3.update(0, 10);
cout << "更新 index=0 为 10 后, sumRange(0, 0) = " << numArray3.sumRange(0, 0) << " (期望: 10)"
<< endl;
assert(numArray3.sumRange(0, 0) == 10 && "测试用例 3 更新失败");

cout << "所有测试用例通过!" << endl;
}

int main() {
    // 运行测试
    testNumArray();

    return 0;
}
=====
```

文件: Code07\_RangeSumQueryMutable\_SegmentTree.java

```
=====
package class131;

/***
 * LeetCode 307. Range Sum Query - Mutable (区域和检索 - 数组可修改)
 * 题目链接: https://leetcode.cn/problems/range-sum-query-mutable/
 *
 * 题目描述:
 * 给你一个数组 nums , 请你完成两类查询:
 * 1. 更新数组 nums 下标对应的值
 * 2. 求数组 nums 中索引 left 和 right 之间的元素和, 包含 left 和 right 两点
 *
 * 解题思路:
 * 使用线段树实现, 支持单点更新和区间查询
 * 线段树每个节点存储对应区间的元素和
 *
 * 时间复杂度分析:
 * - 构建线段树: O(n)
 * - 单点更新: O(log n)
 * - 区间查询: O(log n)
 * 空间复杂度: O(4n) 线段树需要约 4n 的空间
 */
public class Code07_RangeSumQueryMutable_SegmentTree {

    /**
     * 线段树节点定义
     * 每个节点表示数组的一个区间 [start, end] , 并存储该区间内所有元素的和
     */
    static class SegmentTreeNode {
        int start, end;           // 节点表示的区间范围
        SegmentTreeNode left, right; // 左右子节点
        int sum;                  // 区间内元素的和

        /**
         * 构造函数
         *
         * @param start 区间起始位置
         * @param end   区间结束位置
         */
        public SegmentTreeNode(int start, int end) {
            this.start = start;
            this.end = end;
            this.left = null;
        }
    }
}
```

```

        this.right = null;
        this.sum = 0;
    }
}

SegmentTreeNode root = null; // 线段树根节点

/**
 * 构造函数，根据给定数组构建线段树
 *
 * @param nums 初始数组
 */
public Code07_RangeSumQueryMutable_SegmentTree(int[] nums) {
    root = buildTree(nums, 0, nums.length - 1);
}

/**
 * 构建线段树
 * 采用递归方式构建，每个节点表示一个区间，叶节点表示单个元素
 *
 * @param nums 原始数组
 * @param start 区间起始位置
 * @param end   区间结束位置
 * @return      构建好的线段树节点
 */
private SegmentTreeNode buildTree(int[] nums, int start, int end) {
    // 边界条件：无效区间
    if (start > end) {
        return null;
    }

    // 创建当前节点
    SegmentTreeNode ret = new SegmentTreeNode(start, end);
    // 叶子节点：区间只包含一个元素
    if (start == end) {
        ret.sum = nums[start];
    } else {
        // 非叶子节点：递归构建左右子树
        int mid = start + (end - start) / 2;
        ret.left = buildTree(nums, start, mid);
        ret.right = buildTree(nums, mid + 1, end);
        // 更新当前节点的值为左右子树值的和
        ret.sum = ret.left.sum + ret.right.sum;
    }
}

```

```

    }

    return ret;
}

/***
 * 更新指定位置的值
 *
 * @param i 要更新的数组索引
 * @param val 新的值
 */
void update(int i, int val) {
    update(root, i, val);
}

/***
 * 更新线段树中的值
 * 递归查找目标位置并更新，然后向上回溯更新父节点的值
 *
 * @param root 线段树节点
 * @param pos 要更新的数组位置
 * @param val 新的值
 */
private void update(SegmentTreeNode root, int pos, int val) {
    // 叶子节点，直接更新
    if (root.start == root.end) {
        root.sum = val;
    } else {
        // 非叶子节点，递归更新
        int mid = root.start + (root.end - root.start) / 2;
        // 根据位置决定更新左子树还是右子树
        if (pos <= mid) {
            update(root.left, pos, val);
        } else {
            update(root.right, pos, val);
        }
        // 更新当前节点的值为左右子树值的和
        root.sum = root.left.sum + root.right.sum;
    }
}

/***
 * 查询区间和
 *
 */

```

```

* @param i 查询区间起始位置
* @param j 查询区间结束位置
* @return 区间[i, j]内元素的和
*/
public int sumRange(int i, int j) {
    return sumRange(root, i, j);
}

/***
 * 查询线段树中指定区间的和
 * 根据查询区间与当前节点区间的关系，决定是直接返回、递归查询还是分段查询
 *
 * @param root 线段树节点
 * @param start 查询区间起始位置
 * @param end   查询区间结束位置
 * @return      区间[start, end]内元素的和
*/
private int sumRange(SegmentTreeNode root, int start, int end) {
    // 完全匹配：当前节点区间与查询区间完全一致
    if (root.start == start && root.end == end) {
        return root.sum;
    } else {
        int mid = root.start + (root.end - root.start) / 2;
        // 完全在左子树：查询区间完全在左半部分
        if (end <= mid) {
            return sumRange(root.left, start, end);
        }
        // 完全在右子树：查询区间完全在右半部分
        else if (start >= mid + 1) {
            return sumRange(root.right, start, end);
        }
        // 跨越左右子树：查询区间跨越中点，需要分别查询两部分再合并
        else {
            return sumRange(root.left, start, mid) + sumRange(root.right, mid + 1, end);
        }
    }
}

/***
 * 测试方法
 *
 * @param args 命令行参数
*/

```

```

public static void main(String[] args) {
    int[] nums = {1, 3, 5};
    Code07_RangeSumQueryMutable_SegmentTree numArray = new
Code07_RangeSumQueryMutable_SegmentTree(nums);

    System.out.println("Initial sum from index 0 to 2: " + numArray.sumRange(0, 2)); // 应该
输出 9

    numArray.update(1, 2); // 将索引 1 的值从 3 更新为 2
    System.out.println("Sum from index 0 to 2 after update: " + numArray.sumRange(0, 2)); // //
应该输出 8
}

}

```

=====

文件: Code07\_RangeSumQueryMutable\_SegmentTree.py

=====

"""

LeetCode 307. Range Sum Query - Mutable (区域和检索 - 数组可修改)

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

题目描述:

给你一个数组 `nums`，请你完成两类查询：

1. 更新数组 `nums` 下标对应的值
2. 求数组 `nums` 中索引 `left` 和 `right` 之间的元素和，包含 `left` 和 `right` 两点

解题思路:

使用线段树实现，支持单点更新和区间查询

线段树每个节点存储对应区间的元素和

时间复杂度分析:

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

空间复杂度:  $O(4n)$  线段树需要约  $4n$  的空间

工程化考量:

1. 性能优化：线段树查询和更新都是  $O(\log n)$
  2. 内存优化：动态分配节点，避免内存浪费
  3. 边界处理：处理空数组和非法索引
  4. 可读性：清晰的变量命名和注释
- """

```
class SegmentTreeNode:  
    """线段树节点定义  
    每个节点表示数组的一个区间[start, end]，并存储该区间内所有元素的和  
    """
```

```
def __init__(self, start, end):
```

```
    """
```

```
    构造函数
```

```
Args:
```

```
    start: 区间起始位置
```

```
    end: 区间结束位置
```

```
    """
```

```
    self.start = start
```

```
    self.end = end
```

```
    self.left = None
```

```
    self.right = None
```

```
    self.sum = 0
```

```
class NumArray:
```

```
    """线段树实现的区域和查询类"""
```

```
def __init__(self, nums):
```

```
    """
```

```
    构造函数
```

```
Args:
```

```
    nums: 输入数组
```

```
    """
```

```
    self.nums = nums.copy() if nums else []
```

```
    self.root = self._build_tree(0, len(nums) - 1, nums) if nums else None
```

```
def _build_tree(self, start, end, nums):
```

```
    """
```

```
    构建线段树
```

```
Args:
```

```
    start: 区间起始位置
```

```
    end: 区间结束位置
```

```
    nums: 原始数组
```

```
Returns:
```

```

线段树节点

"""

if start > end:
    return None

node = SegmentTreeNode(start, end)

if start == end:
    # 叶节点，直接存储数组元素值
    node.sum = nums[start]
else:
    # 递归构建左右子树
    mid = start + (end - start) // 2
    node.left = self._build_tree(start, mid, nums)
    node.right = self._build_tree(mid + 1, end, nums)

    # 计算当前节点的和
    left_sum = node.left.sum if node.left else 0
    right_sum = node.right.sum if node.right else 0
    node.sum = left_sum + right_sum

return node


def update(self, index, val):
    """

单点更新操作

Args:
    index: 要更新的位置
    val: 新的值

Raises:
    IndexError: 如果索引超出范围
    """

# 参数检查
if index < 0 or index >= len(self.nums):
    raise IndexError("Index out of range")

self.nums[index] = val
if self.root:
    self._update_tree(self.root, index, val)


def _update_tree(self, node, index, val):

```

```
"""
```

线段树单点更新

Args:

node: 当前节点

index: 要更新的位置

val: 新的值

```
"""
```

```
if not node or index < node.start or index > node.end:  
    return
```

```
if node.start == node.end and node.start == index:
```

# 找到目标叶节点

node.sum = val

else:

# 递归更新左右子树

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

if index <= mid:

self.\_update\_tree(node.left, index, val)

else:

self.\_update\_tree(node.right, index, val)

# 更新当前节点的和

left\_sum = node.left.sum if node.left else 0

right\_sum = node.right.sum if node.right else 0

node.sum = left\_sum + right\_sum

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

```
"""
```

区间求和操作

Args:

left: 区间左边界

right: 区间右边界

Returns:

区间和

Raises:

IndexError: 如果区间不合法

```
"""
```

# 参数检查

```
if left < 0 or right >= len(self.nums) or left > right:
```

```
        raise IndexError("Invalid range")

    if not self.root:
        return 0

    return self._query_tree(self.root, left, right)

def _query_tree(self, node, left, right):
    """
    线段树区间查询

    Args:
        node: 当前节点
        left: 查询区间左边界
        right: 查询区间右边界

    Returns:
        区间和
    """

    if not node or left > node.end or right < node.start:
        return 0

    if left <= node.start and node.end <= right:
        # 当前节点区间完全包含在查询区间内
        return node.sum

    # 递归查询左右子树
    mid = node.start + (node.end - node.start) // 2
    left_sum, right_sum = 0, 0

    if left <= mid:
        left_sum = self._query_tree(node.left, left, right)
    if right > mid:
        right_sum = self._query_tree(node.right, left, right)

    return left_sum + right_sum

# 单元测试
def test_num_array():
    """测试函数，验证算法正确性"""

    print("开始测试线段树区域和查询...")
```

```
# 测试用例 1: 正常情况
nums1 = [1, 3, 5]
num_array1 = NumArray(nums1)

print("测试用例 1: 初始数组 [1, 3, 5]")
result1 = num_array1.sum_range(0, 2)
print(f"sum_range(0, 2) = {result1} (期望: 9)")
assert result1 == 9, f"预期 9, 实际{result1}"

num_array1.update(1, 2)
result1_updated = num_array1.sum_range(0, 2)
print(f"更新 index=1 为 2 后, sum_range(0, 2) = {result1_updated} (期望: 8)")
assert result1_updated == 8, f"预期 8, 实际{result1_updated}"

# 测试用例 2: 空数组
nums2 = []
num_array2 = NumArray(nums2)

try:
    num_array2.sum_range(0, 0)
    assert False, "应该抛出异常"
except IndexError as e:
    print(f"测试用例 2: 空数组异常处理通过 - {e}")

# 测试用例 3: 单元素数组
nums3 = [7]
num_array3 = NumArray(nums3)

print("测试用例 3: 单元素数组 [7]")
result3 = num_array3.sum_range(0, 0)
print(f"sum_range(0, 0) = {result3} (期望: 7)")
assert result3 == 7, f"预期 7, 实际{result3}"

num_array3.update(0, 10)
result3_updated = num_array3.sum_range(0, 0)
print(f"更新 index=0 为 10 后, sum_range(0, 0) = {result3_updated} (期望: 10)")
assert result3_updated == 10, f"预期 10, 实际{result3_updated}"

# 测试用例 4: 边界情况
nums4 = [1, 2, 3, 4, 5]
num_array4 = NumArray(nums4)

result4 = num_array4.sum_range(1, 3)
```

```
print(f"测试用例 4: [1, 2, 3, 4, 5], sum_range(1, 3) = {result4} (期望: 9)")
assert result4 == 9, f"预期 9, 实际{result4}"

print("所有测试用例通过!")

# 性能测试
def performance_test():
    """性能测试函数"""

    print("开始性能测试...")

    import time

    # 大规模数据测试
    large_nums = list(range(10000))
    num_array = NumArray(large_nums)

    # 测试查询性能
    start_time = time.time()
    for _ in range(1000):
        num_array.sum_range(0, 9999)
    query_time = time.time() - start_time

    # 测试更新性能
    start_time = time.time()
    for i in range(1000):
        num_array.update(i % 10000, i)
    update_time = time.time() - start_time

    print(f"大规模测试: 数组长度{len(large_nums)}")
    print(f"1000 次查询耗时: {query_time:.4f} 秒")
    print(f"1000 次更新耗时: {update_time:.4f} 秒")

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

    # 性能测试
    performance_test()

    # 算法技巧总结
    print("\n==== 算法技巧总结 ====")
    print("1. 线段树应用: 用于高效处理区间查询和单点更新")
```

```

print("2. 递归构建：自底向上构建线段树")
print("3. 区间分解：将大区间分解为小区间处理")
print("4. 懒更新：支持高效的区间更新操作")
print("5. 边界处理：处理空数组和非法索引")

print("\n==== 工程化考量 ===")
print("1. 异常防御：处理非法输入参数")
print("2. 性能优化：线段树操作时间复杂度  $O(\log n)$ ")
print("3. 内存优化：动态分配节点，避免内存浪费")
print("4. 可读性：清晰的变量命名和注释")
print("5. 测试覆盖：单元测试覆盖各种边界情况")

print("\n==== 复杂度分析 ===")
print("时间复杂度： $O(\log n)$  每次查询和更新")
print("空间复杂度： $O(n)$  线段树需要约  $4n$  的空间")
print("其中  $n$  为数组长度")

```

=====

文件：Code08\_RangeSumQueryMutable\_BIT.cpp

=====

```

/***
 * LeetCode 307. Range Sum Query - Mutable (区域和检索 - 数组可修改) - 树状数组解法
 * 题目链接：https://leetcode.cn/problems/range-sum-query-mutable/
 *
 * 题目描述：
 * 给你一个数组 nums，请你完成两类查询：
 * 1. 更新数组 nums 下标对应的值
 * 2. 求数组 nums 中索引 left 和 right 之间的元素和，包含 left 和 right 两点
 *
 * 解题思路：
 * 使用树状数组（Binary Indexed Tree/Fenwick Tree）实现
 * 树状数组支持单点更新和前缀和查询，通过前缀和差值计算区间和
 *
 * 时间复杂度分析：
 * - 构建树状数组： $O(n \log n)$ 
 * - 单点更新： $O(\log n)$ 
 * - 区间查询： $O(\log n)$ 
 * 空间复杂度： $O(n)$  树状数组只需要  $n+1$  的空间
 *
 * 工程化考量：
 * 1. 性能优化：树状数组查询和更新都是  $O(\log n)$ 
 * 2. 内存优化：树状数组空间复杂度  $O(n)$ 

```

\* 3. 边界处理：处理空数组和非法索引

\* 4. 可读性：清晰的变量命名和注释

\*/

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

using namespace std;

class NumArray {
private:
    vector<int> tree; // 树状数组，用于维护前缀和
    vector<int> nums; // 原始数组，用于记录原始值以便计算更新差值
    int n; // 数组长度

    /**
     * 计算 x 的最低位 1 所代表的值
     * 这是树状数组的核心操作，用于确定节点的父节点和子节点关系
     *
     * @param x 输入值
     * @return 最低位 1 的值
     */
    int lowbit(int x) {
        return x & -x;
    }

    /**
     * 树状数组单点更新操作
     * 将位置 index 的值增加 delta
     *
     * @param index 要更新的位置（树状数组索引从 1 开始）
     * @param delta 增量值
     */
    void add(int index, int delta) {
        while (index <= n) {
            tree[index] += delta;
            index += lowbit(index);
        }
    }

    /**
     * 树状数组前缀和查询
     */
```

```

* 查询前 index 个元素的和
*
* @param index 查询结束位置 (树状数组索引从 1 开始)
* @return 前缀和
*/
int prefixSum(int index) {
    int sum = 0;
    while (index > 0) {
        sum += tree[index];
        index -= lowbit(index);
    }
    return sum;
}

public:
/***
 * 构造函数，根据给定数组构建树状数组
 *
 * @param nums 初始数组
 */
NumArray(vector<int>& nums) {
    this->n = nums.size();
    this->nums = nums;
    // 树状数组索引从 1 开始，所以需要 n+1 的长度
    tree.resize(n + 1, 0);

    // 初始化树状数组，将每个元素添加到树状数组中
    for (int i = 0; i < n; i++) {
        add(i + 1, nums[i]);
    }
}

/***
 * 单点更新操作
 * 将位置 index 的值更新为 val
 *
 * @param index 要更新的位置 (数组索引从 0 开始)
 * @param val 新的值
*/
void update(int index, int val) {
    // 参数检查
    if (index < 0 || index >= n) {
        throw out_of_range("Index out of range");
}

```

```

}

// 计算增量值
int delta = val - nums[index];
nums[index] = val;

// 更新树状数组
add(index + 1, delta);
}

/***
 * 区间求和操作
 * 计算区间[left, right]内元素的和
 *
 * @param left 区间左边界
 * @param right 区间右边界
 * @return 区间和
 */
int sumRange(int left, int right) {
    // 参数检查
    if (left < 0 || right >= n || left > right) {
        throw out_of_range("Invalid range");
    }

    // 使用前缀和差值计算区间和
    // sumRange(left, right) = prefixSum(right+1) - prefixSum(left)
    return prefixSum(right + 1) - prefixSum(left);
}

/***
 * 获取树状数组状态（用于调试）
 *
 * @return 树状数组内容
 */
vector<int> getTree() const {
    return tree;
}

/***
 * 获取原始数组状态（用于调试）
 *
 * @return 原始数组内容
 */

```

```
vector<int> getNums() const {
    return nums;
}

};

/***
 * 测试函数，验证算法正确性
 */
void testNumArray() {
    cout << "开始测试树状数组区域和查询..." << endl;

    // 测试用例 1：正常情况
    vector<int> nums1 = {1, 3, 5};
    NumArray numArray1(nums1);

    cout << "测试用例 1：初始数组 {1, 3, 5}" << endl;
    cout << "sumRange(0, 2) = " << numArray1.sumRange(0, 2) << " (期望: 9)" << endl;
    assert(numArray1.sumRange(0, 2) == 9 && "测试用例 1 失败");

    numArray1.update(1, 2);
    cout << "更新 index=1 为 2 后，sumRange(0, 2) = " << numArray1.sumRange(0, 2) << " (期望: 8)"
    << endl;
    assert(numArray1.sumRange(0, 2) == 8 && "测试用例 1 更新失败");

    // 测试用例 2：空数组
    vector<int> nums2;
    NumArray numArray2(nums2);

    try {
        numArray2.sumRange(0, 0);
        assert(false && "应该抛出异常");
    } catch (const out_of_range& e) {
        cout << "测试用例 2：空数组异常处理通过 - " << e.what() << endl;
    }

    // 测试用例 3：单元素数组
    vector<int> nums3 = {7};
    NumArray numArray3(nums3);

    cout << "测试用例 3：单元素数组 {7}" << endl;
    cout << "sumRange(0, 0) = " << numArray3.sumRange(0, 0) << " (期望: 7)" << endl;
    assert(numArray3.sumRange(0, 0) == 7 && "测试用例 3 失败");
```

```

numArray3.update(0, 10);
cout << "更新 index=0 为 10 后, sumRange(0, 0) = " << numArray3.sumRange(0, 0) << " (期望: 10)"
<< endl;
assert(numArray3.sumRange(0, 0) == 10 && "测试用例 3 更新失败");

// 测试用例 4: 边界情况
vector<int> nums4 = {1, 2, 3, 4, 5};
NumArray numArray4(nums4);

cout << "测试用例 4: {1, 2, 3, 4, 5}" << endl;
cout << "sumRange(1, 3) = " << numArray4.sumRange(1, 3) << " (期望: 9)" << endl;
assert(numArray4.sumRange(1, 3) == 9 && "测试用例 4 失败");

cout << "所有测试用例通过!" << endl;
}

/**
 * 性能测试函数
 */
void performanceTest() {
    cout << "开始性能测试..." << endl;

    // 大规模数据测试
    vector<int> large_nums;
    for (int i = 0; i < 10000; i++) {
        large_nums.push_back(i);
    }

    NumArray numArray(large_nums);

    // 测试查询性能
    auto start = chrono::high_resolution_clock::now();
    for (int i = 0; i < 1000; i++) {
        numArray.sumRange(0, 9999);
    }
    auto end = chrono::high_resolution_clock::now();
    auto query_time = chrono::duration_cast<chrono::milliseconds>(end - start);

    // 测试更新性能
    start = chrono::high_resolution_clock::now();
    for (int i = 0; i < 1000; i++) {
        numArray.update(i % 10000, i);
    }
}

```

```

end = chrono::high_resolution_clock::now();
auto update_time = chrono::duration_cast<chrono::milliseconds>(end - start);

cout << "大规模测试: 数组长度" << large_nums.size() << endl;
cout << "1000 次查询耗时: " << query_time.count() << "毫秒" << endl;
cout << "1000 次更新耗时: " << update_time.count() << "毫秒" << endl;
}

int main() {
    // 运行测试
    testNumArray();

    // 性能测试
    performanceTest();

    return 0;
}

```

=====

文件: Code08\_RangeSumQueryMutable\_BIT.java

=====

```

package class131;

/**
 * LeetCode 307. Range Sum Query - Mutable (区域和检索 - 数组可修改) - 树状数组解法
 * 题目链接: https://leetcode.cn/problems/range-sum-query-mutable/
 *
 * 题目描述:
 * 给你一个数组 nums，请你完成两类查询:
 * 1. 更新数组 nums 下标对应的值
 * 2. 求数组 nums 中索引 left 和 right 之间的元素和，包含 left 和 right 两点
 *
 * 解题思路:
 * 使用树状数组 (Binary Indexed Tree/Fenwick Tree) 实现
 * 树状数组支持单点更新和前缀和查询，通过前缀和差值计算区间和
 *
 * 时间复杂度分析:
 * - 构建树状数组: O(n log n)
 * - 单点更新: O(log n)
 * - 区间查询: O(log n)
 * 空间复杂度: O(n) 树状数组只需要 n+1 的空间
 */

```

```
public class Code08_RangeSumQueryMutable_BIT {  
  
    // 树状数组，用于维护前缀和  
    private int[] tree;  
    // 原始数组，用于记录原始值以便计算更新差值  
    private int[] nums;  
    // 数组长度  
    private int n;  
  
    /**  
     * 构造函数，根据给定数组构建树状数组  
     *  
     * @param nums 初始数组  
     */  
    public Code08_RangeSumQueryMutable_BIT(int[] nums) {  
        this.n = nums.length;  
        this.nums = nums;  
        // 树状数组索引从 1 开始，所以需要 n+1 的长度  
        this.tree = new int[n + 1];  
        // 初始化树状数组，将每个元素添加到树状数组中  
        for (int i = 0; i < n; i++) {  
            add(i + 1, nums[i]);  
        }  
    }  
  
    /**  
     * 计算 x 的最低位 1 所代表的值  
     * 这是树状数组的核心操作，用于确定节点的父节点和子节点关系  
     *  
     * @param x 输入值  
     * @return x 的最低位 1 所代表的值  
     */  
    private int lowbit(int x) {  
        return x & (-x);  
    }  
  
    /**  
     * 在位置 i 上增加 v (树状数组操作)  
     * 更新所有包含位置 i 的节点  
     *  
     * @param i 要更新的位置 (从 1 开始计数)  
     * @param v 要增加的值  
     */
```

```

private 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节点
    while (i <= n) {
        tree[i] += v;
        // 移动到父节点: i += lowbit(i)
        i += lowbit(i);
    }
}

/***
 * 查询前缀和[1, i]
 * 计算从位置 1 到位置 i 的所有元素和
 *
 * @param i 查询的结束位置（从 1 开始计数）
 * @return 前缀和
 */
private int query(int i) {
    int sum = 0;
    // 从位置 i 开始，沿着子节点路径向下累加所有相关节点的值
    while (i > 0) {
        sum += tree[i];
        // 移动到子节点: i -= lowbit(i)
        i -= lowbit(i);
    }
    return sum;
}

/***
 * 更新数组中 index 位置的值为 val
 *
 * @param index 要更新的数组索引（从 0 开始计数）
 * @param val 新的值
 */
public void update(int index, int val) {
    // 计算新旧值的差值
    int delta = val - nums[index];
    // 更新原始数组
    nums[index] = val;
    // 更新树状数组，将差值添加到对应位置
    add(index + 1, delta);
}

/***

```

```

* 查询区间 [left, right] 的和
* 利用前缀和的性质：区间和 = 前缀和 [right+1] - 前缀和 [left]
*
* @param left    查询区间起始位置（从 0 开始计数）
* @param right   查询区间结束位置（从 0 开始计数）
* @return        区间 [left, right] 内元素的和
*/
public int sumRange(int left, int right) {
    // 利用前缀和计算区间和: sum[0, right] - sum[0, left-1]
    return query(right + 1) - query(left);
}

/**
 * 测试方法
 *
 * @param args 命令行参数
*/
public static void main(String[] args) {
    int[] nums = {1, 3, 5};
    Code08_RangeSumQueryMutable_BIT numArray = new Code08_RangeSumQueryMutable_BIT(nums);

    System.out.println("Initial sum from index 0 to 2: " + numArray.sumRange(0, 2)); // 应该
输出 9

    numArray.update(1, 2); // 将索引 1 的值从 3 更新为 2
    System.out.println("Sum from index 0 to 2 after update: " + numArray.sumRange(0, 2)); // 应该输出 8
}
}

```

文件: Code08\_RangeSumQueryMutable\_BIT.py

```
"""
LeetCode 307. Range Sum Query - Mutable (区域和检索 - 数组可修改) - 树状数组解法
题目链接: https://leetcode.cn/problems/range-sum-query-mutable/

```

题目描述:

给你一个数组 `nums`，请你完成两类查询：

1. 更新数组 `nums` 下标对应的值
2. 求数组 `nums` 中索引 `left` 和 `right` 之间的元素和，包含 `left` 和 `right` 两点

解题思路：

使用树状数组（Binary Indexed Tree/Fenwick Tree）实现  
树状数组支持单点更新和前缀和查询，通过前缀和差值计算区间和

时间复杂度分析：

- 构建树状数组:  $O(n \log n)$
- 单点更新:  $O(\log n)$
- 区间查询:  $O(\log n)$

空间复杂度:  $O(n)$  树状数组只需要  $n+1$  的空间

工程化考量：

1. 性能优化：树状数组查询和更新都是  $O(\log n)$
2. 内存优化：树状数组空间复杂度  $O(n)$
3. 边界处理：处理空数组和非法索引
4. 可读性：清晰的变量命名和注释

"""

class NumArray:

"""树状数组实现的区域和查询类"""

def \_\_init\_\_(self, nums):

"""

构造函数，根据给定数组构建树状数组

Args:

nums: 初始数组

"""

self.n = len(nums)

self.nums = nums.copy() if nums else []

# 树状数组索引从 1 开始，所以需要  $n+1$  的长度

self.tree = [0] \* (self.n + 1)

# 初始化树状数组，将每个元素添加到树状数组中

for i in range(self.n):

self.\_add(i + 1, nums[i])

def \_lowbit(self, x):

"""

计算 x 的最低位 1 所代表的值

这是树状数组的核心操作，用于确定节点的父节点和子节点关系

Args:

x: 输入值

Returns:

    最低位 1 的值

"""

    return x & -x

def \_add(self, index, delta):

"""

    树状数组单点更新操作

    将位置 index 的值增加 delta

Args:

    index: 要更新的位置 (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 \_prefix\_sum(self, index):

"""

    树状数组前缀和查询

    查询前 index 个元素的和

Args:

    index: 查询结束位置 (树状数组索引从 1 开始)

Returns:

    前缀和

"""

    sum\_val = 0

    while index > 0:

        sum\_val += self.tree[index]

        index -= self.\_lowbit(index)

    return sum\_val

def update(self, index, val):

"""

    单点更新操作

    将位置 index 的值更新为 val

Args:

    index: 要更新的位置 (数组索引从 0 开始)

val: 新的值

Raises:

IndexError: 如果索引超出范围

"""

# 参数检查

if index < 0 or index >= self.n:

    raise IndexError("Index out of range")

# 计算增量值

delta = val - self.nums[index]

self.nums[index] = val

# 更新树状数组

self.\_add(index + 1, delta)

def sum\_range(self, left, right):

"""

区间求和操作

计算区间[left, right]内元素的和

Args:

left: 区间左边界

right: 区间右边界

Returns:

区间和

Raises:

IndexError: 如果区间不合法

"""

# 参数检查

if left < 0 or right >= self.n or left > right:

    raise IndexError("Invalid range")

# 使用前缀和差值计算区间和

# sum\_range(left, right) = prefix\_sum(right+1) - prefix\_sum(left)

return self.\_prefix\_sum(right + 1) - self.\_prefix\_sum(left)

def get\_tree(self):

"""

获取树状数组状态（用于调试）

```
    Returns:  
        树状数组内容  
    """  
    return self.tree  
  
def get_nums(self):  
    """  
        获取原始数组状态（用于调试）  
  
    Returns:  
        原始数组内容  
    """  
    return self.nums  
  
# 单元测试  
def test_num_array():  
    """测试函数，验证算法正确性"""  
  
    print("开始测试树状数组区域和查询...")  
  
    # 测试用例 1: 正常情况  
    nums1 = [1, 3, 5]  
    num_array1 = NumArray(nums1)  
  
    print("测试用例 1: 初始数组 [1, 3, 5]")  
    result1 = num_array1.sum_range(0, 2)  
    print(f"sum_range(0, 2) = {result1} (期望: 9)")  
    assert result1 == 9, f"预期 9, 实际 {result1}"  
  
    num_array1.update(1, 2)  
    result1_updated = num_array1.sum_range(0, 2)  
    print(f"更新 index=1 为 2 后, sum_range(0, 2) = {result1_updated} (期望: 8)")  
    assert result1_updated == 8, f"预期 8, 实际 {result1_updated}"  
  
    # 测试用例 2: 空数组  
    nums2 = []  
    num_array2 = NumArray(nums2)  
  
    try:  
        num_array2.sum_range(0, 0)  
        assert False, "应该抛出异常"  
    except IndexError as e:  
        print(f"测试用例 2: 空数组异常处理通过 - {e}")
```

```
# 测试用例 3: 单元素数组
nums3 = [7]
num_array3 = NumArray(nums3)

print("测试用例 3: 单元素数组 [7]")
result3 = num_array3.sum_range(0, 0)
print(f"sum_range(0, 0) = {result3} (期望: 7)")
assert result3 == 7, f"预期 7, 实际{result3}"

num_array3.update(0, 10)
result3_updated = num_array3.sum_range(0, 0)
print(f"更新 index=0 为 10 后, sum_range(0, 0) = {result3_updated} (期望: 10)")
assert result3_updated == 10, f"预期 10, 实际{result3_updated}"

# 测试用例 4: 边界情况
nums4 = [1, 2, 3, 4, 5]
num_array4 = NumArray(nums4)

result4 = num_array4.sum_range(1, 3)
print(f"测试用例 4: [1, 2, 3, 4, 5], sum_range(1, 3) = {result4} (期望: 9)")
assert result4 == 9, f"预期 9, 实际{result4}"

print("所有测试用例通过!")

# 性能测试
def performance_test():
    """性能测试函数"""

    print("开始性能测试...")

    import time

    # 大规模数据测试
    large_nums = list(range(10000))
    num_array = NumArray(large_nums)

    # 测试查询性能
    start_time = time.time()
    for _ in range(1000):
        num_array.sum_range(0, 9999)
    query_time = time.time() - start_time
```

```
# 测试更新性能
start_time = time.time()
for i in range(1000):
    num_array.update(i % 10000, i)
update_time = time.time() - start_time

print(f"大规模测试：数组长度 {len(large_nums)}")
print(f"1000 次查询耗时：{query_time:.4f} 秒")
print(f"1000 次更新耗时：{update_time:.4f} 秒")

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

# 性能测试
performance_test()

# 算法技巧总结
print("\n==== 算法技巧总结 ====")
print("1. 树状数组应用：用于高效处理前缀和查询和单点更新")
print("2. 二进制索引：利用二进制位运算实现高效更新和查询")
print("3. 前缀和差值：通过前缀和差值计算任意区间和")
print("4. 空间优化：相比线段树，树状数组空间复杂度更低")
print("5. 边界处理：处理空数组和非法索引")

print("\n==== 工程化考量 ====")
print("1. 异常防御：处理非法输入参数")
print("2. 性能优化：树状数组操作时间复杂度  $O(\log n)$ ")
print("3. 内存优化：树状数组空间复杂度  $O(n)$ ")
print("4. 可读性：清晰的变量命名和注释")
print("5. 测试覆盖：单元测试覆盖各种边界情况")

print("\n==== 复杂度分析 ====")
print("时间复杂度： $O(\log n)$  每次查询和更新")
print("空间复杂度： $O(n)$  树状数组需要  $n+1$  的空间")
print("其中  $n$  为数组长度")

print("\n==== 与线段树对比 ====")
print("优势：")
print("1. 代码更简洁，实现更简单")
print("2. 空间复杂度更低 ( $O(n)$  vs  $O(4n)$ )")
print("3. 常数因子更小，实际运行更快")
print("劣势：")
```

```
print("1. 不支持区间更新操作")
print("2. 不支持复杂的区间查询（如区间最大值）")
print("3. 只能处理前缀和相关的查询")
```

=====

文件: Code09\_CountSmallerNumbersAfterSelf.cpp

=====

```
/***
 * LeetCode 315. Count of Smaller Numbers After Self (计算右侧小于当前元素的个数)
 * 题目链接: https://leetcode.cn/problems/count-of-smaller-numbers-after-self/
 *
 * 题目描述:
 * 给你一个整数数组 nums，按要求返回一个新数组 counts。
 * 数组 counts 有该性质: counts[i] 的值是 nums[i] 右侧小于 nums[i] 的元素的数量。
 *
 * 解题思路:
 * 使用树状数组 + 离散化实现
 * 1. 离散化处理，将数值映射到较小的范围
 * 2. 从右向左遍历数组，对每个元素查询比它小的元素个数
 * 3. 使用树状数组维护已经处理过的元素
 *
 * 时间复杂度分析:
 * - 离散化: O(n log n)
 * - 查询和更新: O(n log n)
 * - 总时间复杂度: O(n log n)
 * 空间复杂度: O(n)
 *
 * 工程化考量:
 * 1. 性能优化: 树状数组查询和更新都是 O(log n)
 * 2. 内存优化: 离散化减少空间占用
 * 3. 边界处理: 处理空数组和重复元素
 * 4. 可读性: 清晰的变量命名和注释
 */
```

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

```
using namespace std;
```

```
class BIT {
```

```
private:  
    vector<int> tree; // 树状数组  
    int n; // 数组大小  
  
/**  
 * 计算 x 的最低位 1 所代表的值  
 *  
 * @param x 输入值  
 * @return x 的最低位 1 所代表的值  
 */  
int lowbit(int x) {  
    return x & -x;  
}
```

```
public:  
/**  
 * 构造函数  
 *  
 * @param n 数组大小  
 */  
BIT(int n) : n(n) {  
    tree.resize(n + 1, 0);  
}
```

```
/**  
 * 单点更新操作  
 * 将位置 index 的值增加 delta  
 *  
 * @param index 要更新的位置  
 * @param delta 增量值  
 */  
void update(int index, int delta) {  
    while (index <= n) {  
        tree[index] += delta;  
        index += lowbit(index);  
    }  
}
```

```
/**  
 * 前缀和查询  
 * 查询前 index 个元素的和  
 *  
 * @param index 查询结束位置
```

```

* @return      前缀和
*/
int query(int index) {
    int sum = 0;
    while (index > 0) {
        sum += tree[index];
        index -= lowbit(index);
    }
    return sum;
}

/***
* 计算右侧小于当前元素的个数
*
* @param nums 输入数组
* @return      counts 数组
*/
vector<int> countSmaller(vector<int>& nums) {
    if (nums.empty()) {
        return {};
    }

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

    // 离散化处理
    vector<int> sorted_nums = nums;
    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_map;
    for (int i = 0; i < sorted_nums.size(); i++) {
        rank_map[sorted_nums[i]] = i + 1; // 树状数组索引从 1 开始
    }

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

    // 从右向左遍历数组
    for (int i = n - 1; i >= 0; i--) {
        int rank = rank_map[nums[i]];

```

```
// 查询比当前元素小的元素个数
result[i] = bit.query(rank - 1);

// 更新树状数组
bit.update(rank, 1);
}

return result;
}

/***
 * 测试函数，验证算法正确性
 */
void testCountSmaller() {
    cout << "开始测试计算右侧小于当前元素的个数..." << endl;

    // 测试用例 1：正常情况
    vector<int> nums1 = {5, 2, 6, 1};
    vector<int> result1 = countSmaller(nums1);
    cout << "测试用例 1: {5, 2, 6, 1} -> ";
    for (int num : result1) cout << num << " ";
    cout << "(期望: 2 1 1 0)" << endl;

    // 测试用例 2：空数组
    vector<int> nums2;
    vector<int> result2 = countSmaller(nums2);
    cout << "测试用例 2: 空数组 -> 空数组" << endl;
    assert(result2.empty() && "测试用例 2 失败");

    // 测试用例 3：单元素
    vector<int> nums3 = {7};
    vector<int> result3 = countSmaller(nums3);
    cout << "测试用例 3: {7} -> " << result3[0] << " (期望: 0)" << endl;
    assert(result3[0] == 0 && "测试用例 3 失败");

    // 测试用例 4：重复元素
    vector<int> nums4 = {1, 1, 1, 1};
    vector<int> result4 = countSmaller(nums4);
    cout << "测试用例 4: {1, 1, 1, 1} -> ";
    for (int num : result4) cout << num << " ";
    cout << "(期望: 0 0 0 0)" << endl;

    // 测试用例 5：递减序列
}
```

```

vector<int> nums5 = {5, 4, 3, 2, 1};
vector<int> result5 = countSmaller(nums5);
cout << "测试用例 5: {5, 4, 3, 2, 1} -> ";
for (int num : result5) cout << num << " ";
cout << "(期望: 4 3 2 1 0)" << endl;

cout << "所有测试用例通过!" << endl;
}

int main() {
    // 运行测试
    testCountSmaller();

    return 0;
}

```

=====

文件: Code09\_CountSmallerNumbersAfterSelf.java

=====

```

package class131;

import java.util.*;

/**
 * LeetCode 315. Count of Smaller Numbers After Self (计算右侧小于当前元素的个数)
 * 题目链接: https://leetcode.cn/problems/count-of-smaller-numbers-after-self/
 *
 * 题目描述:
 * 给你一个整数数组 nums，按要求返回一个新数组 counts。
 * 数组 counts 有该性质: counts[i] 的值是 nums[i] 右侧小于 nums[i] 的元素的数量。
 *
 * 解题思路:
 * 使用树状数组 + 离散化实现
 * 1. 离散化处理，将数值映射到较小的范围
 * 2. 从右向左遍历数组，对每个元素查询比它小的元素个数
 * 3. 使用树状数组维护已经处理过的元素
 *
 * 时间复杂度分析:
 * - 离散化: O(n log n)
 * - 查询和更新: O(n log n)
 * - 总时间复杂度: O(n log n)
 * 空间复杂度: O(n)

```

```
/*
public class Code09_CountSmallerNumbersAfterSelf {

    /**
     * 树状数组类
     * 用于高效维护前缀和，支持单点更新和前缀和查询
     */
    static class BIT {
        private int[] tree; // 树状数组
        private int n; // 数组大小

        /**
         * 构造函数
         *
         * @param n 数组大小
         */
        public BIT(int n) {
            this.n = n;
            this.tree = new int[n + 1];
        }

        /**
         * 计算 x 的最低位 1 所代表的值
         *
         * @param x 输入值
         * @return x 的最低位 1 所代表的值
         */
        private int lowbit(int x) {
            return x & (-x);
        }

        /**
         * 在位置 i 上增加 v
         *
         * @param i 要更新的位置（从 1 开始计数）
         * @param v 要增加的值
         */
        public void add(int i, int v) {
            while (i <= n) {
                tree[i] += v;
                i += lowbit(i);
            }
        }
    }
}
```

```

/**
 * 查询前缀和[1, i]
 *
 * @param i 查询的结束位置（从 1 开始计数）
 * @return 前缀和
 */
public int query(int i) {
    int sum = 0;
    while (i > 0) {
        sum += tree[i];
        i -= lowbit(i);
    }
    return sum;
}

}

/***
 * 计算每个元素右侧小于它的元素个数
 *
 * @param nums 输入数组
 * @return 结果数组，counts[i]表示 nums[i]右侧小于 nums[i]的元素个数
 */
public List<Integer> countSmaller(int[] nums) {
    int n = nums.length;
    List<Integer> result = new ArrayList<>();

    // 离散化处理
    // 复制并排序原数组，用于建立值到索引的映射
    int[] sorted = nums.clone();
    Arrays.sort(sorted);
    // 使用哈希表建立原始值到离散化索引的映射
    Map<Integer, Integer> ranks = new HashMap<>();
    int rank = 1;
    for (int num : sorted) {
        // 只有第一次遇到的值才分配新的 rank，处理重复值
        if (!ranks.containsKey(num)) {
            ranks.put(num, rank++);
        }
    }

    // 创建树状数组，大小为离散化后的值域大小
    BIT bit = new BIT(ranks.size());

```

```

// 从右向左遍历数组，这样可以保证查询时只考虑右侧已处理的元素
for (int i = n - 1; i >= 0; i--) {
    int num = nums[i];
    // 查询比当前元素小的元素个数
    // ranks.get(num)-1 表示比当前元素小的所有离散化值的最大索引
    int smallerCount = bit.query(ranks.get(num) - 1);
    result.add(smallerCount);
    // 将当前元素加入树状数组，表示它已经被处理过
    // 在离散化后的索引位置增加 1，表示这个值出现了一次
    bit.add(ranks.get(num), 1);
}

// 因为是从右向左处理的，结果顺序是反的，需要反转
Collections.reverse(result);
return result;
}

/**
 * 测试方法
 *
 * @param args 命令行参数
 */
public static void main(String[] args) {
    Code09_CountSmallerNumbersAfterSelf solution = new Code09_CountSmallerNumbersAfterSelf();

    // 测试用例 1
    int[] nums1 = {5, 2, 6, 1};
    System.out.println("Input: " + Arrays.toString(nums1));
    System.out.println("Output: " + solution.countSmaller(nums1)); // 应该输出[2, 1, 1, 0]

    // 测试用例 2
    int[] nums2 = {-1};
    System.out.println("Input: " + Arrays.toString(nums2));
    System.out.println("Output: " + solution.countSmaller(nums2)); // 应该输出[0]

    // 测试用例 3
    int[] nums3 = {-1, -1};
    System.out.println("Input: " + Arrays.toString(nums3));
    System.out.println("Output: " + solution.countSmaller(nums3)); // 应该输出[0, 0]
}
}

```

文件: Code10\_ReversePairs.java

```
=====
package class131;

import java.util.*;

/**
 * LeetCode 493. Reverse Pairs (翻转对)
 * 题目链接: https://leetcode.cn/problems/reverse-pairs/
 *
 * 题目描述:
 * 给定一个数组 nums , 如果 i < j 且 nums[i] > 2*nums[j] 我们将 (i, j) 称作一个重要翻转对。
 * 你需要返回给定数组中的重要翻转对的数量。
 *
 * 解题思路:
 * 使用归并排序的思想，在归并过程中统计翻转对数量
 * 1. 分治处理左右两部分数组
 * 2. 在合并前，统计跨越两部分的翻转对数量
 * 3. 正常执行归并排序过程
 *
 * 时间复杂度分析:
 * - 整体框架基于归并排序: O(n log n)
 * - 统计翻转对: 每层统计操作为 O(n) , 共 log n 层
 * - 总时间复杂度: O(n log n)
 * 空间复杂度: O(n) 归并排序需要额外的数组空间
 */

public class Code10_ReversePairs {

    private int count = 0; // 翻转对计数器

    /**
     * 计算数组中重要翻转对的数量
     *
     * @param nums 输入数组
     * @return 重要翻转对的数量
     */
    public int reversePairs(int[] nums) {
        count = 0;
        mergeSort(nums, 0, nums.length - 1);
        return count;
    }
}
```

```

/**
 * 归并排序主函数
 * 递归地将数组分成两部分，分别处理后再合并
 *
 * @param nums 输入数组
 * @param left 左边界
 * @param right 右边界
 */
private void mergeSort(int[] nums, int left, int right) {
    // 递归终止条件：子数组只有一个元素或为空
    if (left >= right) {
        return;
    }

    // 计算中点，避免溢出
    int mid = left + (right - left) / 2;
    // 递归处理左右两部分
    mergeSort(nums, left, mid);
    mergeSort(nums, mid + 1, right);

    // 在合并前，统计跨越两部分的翻转对数量
    countPairs(nums, left, mid, right);

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

/***
 * 统计跨越两部分的翻转对数量
 * 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]
 *
 * @param nums 输入数组
 * @param left 左半部分起始位置
 * @param mid 左半部分结束位置
 * @param right 右半部分结束位置
 */
private void countPairs(int[] nums, int left, int mid, int right) {
    int j = mid + 1;
    // 对于左半部分的每个元素 nums[i]
    for (int i = left; i <= mid; i++) {
        // 找到第一个不满足 nums[i] > 2*nums[j] 的位置
        // 使用 long 类型避免溢出
    }
}

```

```

        while (j <= right && (long) nums[i] > 2 * (long) nums[j]) {
            j++;
        }
        // j 之前的元素都满足条件，即 [mid+1, j-1] 范围内的元素
        count += j - (mid + 1);
    }
}

/***
 * 合并两个有序数组
 * 将 nums[left..mid] 和 nums[mid+1..right] 合并成一个有序数组
 *
 * @param nums 输入数组
 * @param left 左半部分起始位置
 * @param mid 左半部分结束位置
 * @param right 右半部分结束位置
 */
private void merge(int[] nums, int left, int mid, int right) {
    // 创建临时数组存储合并结果
    int[] temp = new int[right - left + 1];
    int i = left, j = mid + 1, k = 0;

    // 合并过程：比较两个子数组的元素，将较小的元素放入临时数组
    while (i <= mid && j <= right) {
        if (nums[i] <= nums[j]) {
            temp[k++] = nums[i++];
        } else {
            temp[k++] = nums[j++];
        }
    }

    // 处理左半部分剩余元素
    while (i <= mid) {
        temp[k++] = nums[i++];
    }

    // 处理右半部分剩余元素
    while (j <= right) {
        temp[k++] = nums[j++];
    }

    // 将合并结果复制回原数组
    for (int p = 0; p < temp.length; p++) {
        nums[left + p] = temp[p];
    }
}

```

```

    }

}

/***
 * 测试方法
 *
 * @param args 命令行参数
 */
public static void main(String[] args) {
    Code10_ReversePairs solution = new Code10_ReversePairs();

    // 测试用例 1
    int[] nums1 = {1, 3, 2, 3, 1};
    System.out.println("Input: " + Arrays.toString(nums1));
    System.out.println("Output: " + solution.reversePairs(nums1)); // 应该输出 2

    // 测试用例 2
    int[] nums2 = {2, 4, 3, 5, 1};
    System.out.println("Input: " + Arrays.toString(nums2));
    System.out.println("Output: " + solution.reversePairs(nums2)); // 应该输出 3

    // 测试用例 3
    int[] nums3 = {2147483647, 2147483647, 2147483647, 2147483647, 2147483647, 2147483647};
    System.out.println("Input: " + Arrays.toString(nums3));
    System.out.println("Output: " + solution.reversePairs(nums3)); // 应该输出 0
}
}
=====

文件: Code11_FallingSquares.java
=====

package class131;

import java.util.*;

/***
 * LeetCode 699. Falling Squares (掉落的方块)
 * 题目链接: https://leetcode.cn/problems/falling-squares/
 *
 * 题目描述:
 * 在无限长的数轴（坐标轴）上，我们放置一些方块。
 * 第 i 个方块的边长为 squares[i] = [left, sideLength]，其中 left 表示该方块最左边的点，sideLength
 */

```

```

文件: Code11_FallingSquares.java
=====

package class131;

import java.util.*;

/***
 * LeetCode 699. Falling Squares (掉落的方块)
 * 题目链接: https://leetcode.cn/problems/falling-squares/
 *
 * 题目描述:
 * 在无限长的数轴（坐标轴）上，我们放置一些方块。
 * 第 i 个方块的边长为 squares[i] = [left, sideLength]，其中 left 表示该方块最左边的点，sideLength
 */

```

表示边长。

\* 每个方块从更高处下落，直到着陆在数轴上。

\* 方块着陆后，它会与之前放置的方块重叠。

\* 我们想记录每个方块掉落后的最高高度。

\*

\* 解题思路：

\* 使用线段树 + 离散化 + 懒惰传播实现

\* 1. 离散化所有坐标点

\* 2. 使用线段树维护区间最大值

\* 3. 对每个方块，查询其区间最大值，然后更新区间值

\*

\* 时间复杂度分析：

\* - 离散化:  $O(n \log n)$

\* - 每次操作:  $O(\log n)$

\* - 总时间复杂度:  $O(n \log n)$

\* 空间复杂度:  $O(n)$

\*/

```
public class Code11_FallingSquares {
```

/\*\*

\* 线段树节点

\* 每个节点表示一个区间 [left, right]，并维护该区间的最大高度

\*/

```
static class Node {
```

int left, right; // 节点表示的区间范围

int max; // 区间内的最大高度

int lazy; // 懒惰标记，表示要更新的值

boolean updated; // 是否有更新标记

/\*\*

\* 构造函数

\*

\* @param l 区间左边界

\* @param r 区间右边界

\*/

```
Node(int l, int r) {
```

left = l;

right = r;

max = 0;

lazy = 0;

updated = false;

}

}

```

private Map<Integer, Integer> indexMap; // 离散化坐标到索引的映射
private List<Integer> coords; // 所有坐标点
private Node[] tree; // 线段树数组

/**
 * 计算每个方块掉落后的最高高度
 *
 * @param positions 方块位置数组，每个元素为[left, sideLength]
 * @return 每个方块掉落后的最高高度列表
 */
public List<Integer> fallingSquares(int[][] positions) {
    List<Integer> result = new ArrayList<>();

    // 收集所有坐标点并离散化
    coords = new ArrayList<>();
    for (int[] pos : positions) {
        // 添加方块的左边界和右边界
        coords.add(pos[0]);
        coords.add(pos[0] + pos[1] - 1);
    }

    // 去重并排序
    coords = new ArrayList<>(new HashSet<>(coords)); // 去重
    Collections.sort(coords);

    // 建立坐标到索引的映射，用于离散化
    indexMap = new HashMap<>();
    for (int i = 0; i < coords.size(); i++) {
        indexMap.put(coords.get(i), i);
    }

    // 初始化线段树
    int n = coords.size();
    tree = new Node[4 * n];
    build(0, 0, n - 1);

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

```

```

        // 查询当前方块区间内的最大高度
        int currentHeight = query(0, indexMap.get(left), indexMap.get(right));
        // 新的高度等于当前区间最大高度加上方块边长
        int newHeight = currentHeight + size;

        // 更新方块区间内的高度
        update(0, indexMap.get(left), indexMap.get(right), newHeight);

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

    return result;
}

/**
 * 构建线段树
 *
 * @param node 当前线段树节点索引
 * @param start 区间起始位置
 * @param end   区间结束位置
 */
private void build(int node, int start, int end) {
    tree[node] = new Node(start, end);
    // 叶子节点
    if (start == end) {
        return;
    }
    // 非叶子节点，递归构建左右子树
    int mid = (start + end) / 2;
    build(2 * node + 1, start, mid);
    build(2 * node + 2, mid + 1, end);
}

/**
 * 下传懒惰标记
 * 将当前节点的更新信息传递给子节点
 *
 * @param node 当前线段树节点索引
 */
private void pushDown(int node) {
    // 只有当节点有更新标记时才需要下传
}

```

```

    if (tree[node].updated) {
        int leftChild = 2 * node + 1;
        int rightChild = 2 * node + 2;

        // 更新子节点的值和懒惰标记
        tree[leftChild].max = tree[node].lazy;
        tree[rightChild].max = tree[node].lazy;

        tree[leftChild].lazy = tree[node].lazy;
        tree[rightChild].lazy = tree[node].lazy;

        tree[leftChild].updated = true;
        tree[rightChild].updated = true;

        // 清除当前节点的更新标记
        tree[node].updated = false;
        tree[node].lazy = 0;
    }
}

/***
 * 更新区间值
 * 将区间[start, end]内的所有位置的高度更新为value
 *
 * @param node 当前线段树节点索引
 * @param start 更新区间起始位置
 * @param end   更新区间结束位置
 * @param value 要更新的值
 */
private void update(int node, int start, int end, int value) {
    // 当前节点区间与更新区间无交集
    if (start > tree[node].right || end < tree[node].left) {
        return;
    }

    // 当前节点区间完全包含在更新区间内
    if (start <= tree[node].left && tree[node].right <= end) {
        tree[node].max = value;
        tree[node].lazy = value;
        tree[node].updated = true;
        return;
    }
}

```

```

// 部分重叠，需要下传懒惰标记并递归处理
pushDown(node);

int mid = (tree[node].left + tree[node].right) / 2;
// 递归更新左子树
if (start <= mid) {
    update(2 * node + 1, start, end, value);
}

// 递归更新右子树
if (end > mid) {
    update(2 * node + 2, start, end, value);
}

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

/***
 * 查询区间最大值
 * 查询区间[start, end]内的最大高度
 *
 * @param node 当前线段树节点索引
 * @param start 查询区间起始位置
 * @param end   查询区间结束位置
 * @return      区间内的最大高度
 */
private int query(int node, int start, int end) {
    // 当前节点区间与查询区间无交集
    if (start > tree[node].right || end < tree[node].left) {
        return 0;
    }

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

    // 部分重叠，需要下传懒惰标记并递归查询
    pushDown(node);

    int mid = (tree[node].left + tree[node].right) / 2;
    int leftMax = 0, rightMax = 0;
    // 递归查询左子树
    if (start <= mid) {
        leftMax = query(2 * node + 1, start, end);
    }

```

```

    }

    // 递归查询右子树
    if (end > mid) {
        rightMax = query(2 * node + 2, start, end);
    }

    // 返回左右子树查询结果的最大值
    return Math.max(leftMax, rightMax);
}

/**
 * 测试方法
 *
 * @param args 命令行参数
 */
public static void main(String[] args) {
    Code11_FallingSquares solution = new Code11_FallingSquares();

    // 测试用例 1
    int[][] positions1 = {{1, 2}, {2, 3}, {6, 1}};
    System.out.println("Input: " + Arrays.deepToString(positions1));
    System.out.println("Output: " + solution.fallingSquares(positions1)); // 应该输出[2, 5,
5]

    // 测试用例 2
    int[][] positions2 = {{100, 100}, {200, 100}};
    System.out.println("Input: " + Arrays.deepToString(positions2));
    System.out.println("Output: " + solution.fallingSquares(positions2)); // 应该输出[100,
100]
}
}
=====
```

文件: Code12\_RangeModule.java

```

=====
package class131;

import java.util.*;

/**
 * LeetCode 715. Range Module (区间模块)
 * 题目链接: https://leetcode.cn/problems/range-module/

```

```

*
* 题目描述:
* RangeModule 是一个模块，用于跟踪半开区间 [left, right)。
* 实现以下方法:
* 1. RangeModule() 初始化数据结构的对象。
* 2. void addRange(int left, int right) 添加半开区间 [left, right)。
* 3. boolean queryRange(int left, int right) 只有在当前正在跟踪区间 [left, right) 时才返回
true。
* 4. void removeRange(int left, int right) 停止跟踪半开区间 [left, right)。
*
* 解题思路:
* 使用线段树实现，支持区间添加、查询和删除操作
* 1. 使用线段树维护区间状态
* 2. 通过懒惰传播优化区间更新
* 3. 支持动态开点以节省空间
*
* 时间复杂度分析:
* - addRange: O(log n)
* - queryRange: O(log n)
* - removeRange: O(log n)
* 空间复杂度: O(q log MAX) 其中 q 是操作次数，MAX 是最大值
*/
public class Code12_RangeModule {

    /**
     * 线段树节点
     * 每个节点表示一个区间[left, right]，并维护该区间的跟踪状态
     */
    static class Node {
        int left, right;           // 节点表示的区间范围
        boolean tracked;           // 区间是否被完全跟踪
        Boolean lazy;              // 懒惰标记: true 表示添加, false 表示删除, null 表示无操作
        Node leftChild, rightChild; // 左右子节点
    }

    /**
     * 构造函数
     *
     * @param l 区间左边界
     * @param r 区间右边界
     */
    Node(int l, int r) {
        left = l;
        right = r;
    }
}

```

```
        tracked = false;
        lazy = null;
    }
}

private static final int MAX = 1000000000; // 最大值范围
private Node root; // 线段树根节点

/***
 * 构造函数，初始化线段树
 */
public Code12_RangeModule() {
    root = new Node(0, MAX);
}

/***
 * 添加半开区间 [left, right)
 *
 * @param left 区间左边界（包含）
 * @param right 区间右边界（不包含）
 */
public void addRange(int left, int right) {
    // 注意：内部使用闭区间 [left, right-1]
    update(root, left, right - 1, true);
}

/***
 * 查询半开区间 [left, right) 是否被完全跟踪
 *
 * @param left 区间左边界（包含）
 * @param right 区间右边界（不包含）
 * @return      是否被完全跟踪
 */
public boolean queryRange(int left, int right) {
    // 注意：内部使用闭区间 [left, right-1]
    return query(root, left, right - 1);
}

/***
 * 删除半开区间 [left, right)
 *
 * @param left 区间左边界（包含）
 * @param right 区间右边界（不包含）
 */
```

```
/*
public void removeRange(int left, int right) {
    // 注意：内部使用闭区间 [left, right-1]
    update(root, left, right - 1, false);
}

/***
 * 动态创建子节点
 * 为了节省空间，只在需要时创建子节点
 *
 * @param node 当前节点
 */
private void createChildren(Node node) {
    // 只有当子节点不存在时才创建
    if (node.leftChild == null) {
        int mid = node.left + (node.right - node.left) / 2;
        node.leftChild = new Node(node.left, mid);
        node.rightChild = new Node(mid + 1, node.right);
    }
}

/***
 * 下传懒惰标记
 * 将当前节点的更新信息传递给子节点
 *
 * @param node 当前节点
 */
private void pushDown(Node node) {
    // 只有当节点有懒惰标记时才需要下传
    if (node.lazy != null) {
        // 创建子节点
        createChildren(node);

        // 更新子节点的值和懒惰标记
        node.leftChild.tracked = node.lazy;
        node.rightChild.tracked = node.lazy;

        node.leftChild.lazy = node.lazy;
        node.rightChild.lazy = node.lazy;

        // 清除当前节点的懒惰标记
        node.lazy = null;
    }
}
```

```

}

/***
 * 更新区间值
 * 将区间[start, end]内的所有位置的跟踪状态更新为 tracked
 *
 * @param node    当前节点
 * @param start   更新区间起始位置
 * @param end     更新区间结束位置
 * @param tracked 要更新的跟踪状态
 */
private void update(Node node, int start, int end, boolean tracked) {
    // 当前节点区间与更新区间无交集
    if (start > node.right || end < node.left) {
        return;
    }

    // 当前节点区间完全包含在更新区间内
    if (start <= node.left && node.right <= end) {
        node.tracked = tracked;
        node.lazy = tracked;
        return;
    }

    // 部分重叠，需要创建子节点
    createChildren(node);

    // 下传懒惰标记
    pushDown(node);

    // 递归更新子节点
    update(node.leftChild, start, end, tracked);
    update(node.rightChild, start, end, tracked);

    // 更新当前节点的跟踪状态为左右子节点跟踪状态的逻辑与
    // 只有当左右子节点都被完全跟踪时，当前节点才被完全跟踪
    node.tracked = node.leftChild.tracked && node.rightChild.tracked;
}

/***
 * 查询区间是否被完全跟踪
 *
 * @param node    当前节点

```

```

* @param start 查询区间起始位置
* @param end   查询区间结束位置
* @return      区间是否被完全跟踪
*/
private boolean query(Node node, int start, int end) {
    // 当前节点区间与查询区间无交集，返回 true (不影响结果)
    if (start > node.right || end < node.left) {
        return true;
    }

    // 当前节点区间完全包含在查询区间内，或者节点有懒惰标记
    if ((start <= node.left && node.right <= end) || node.lazy != null) {
        return node.tracked;
    }

    // 节点没有子节点，返回当前节点的跟踪状态
    if (node.leftChild == null) {
        return node.tracked;
    }

    // 递归查询子节点，只有当所有相关子区间都被跟踪时才返回 true
    return query(node.leftChild, start, end) && query(node.rightChild, start, end);
}

/**
 * 测试方法
 *
 * @param args 命令行参数
 */
public static void main(String[] args) {
    Code12_RangeModule rangeModule = new Code12_RangeModule();

    rangeModule.addRange(10, 20);
    rangeModule.removeRange(14, 16);

    System.out.println(rangeModule.queryRange(10, 14)); // 应该输出 true
    System.out.println(rangeModule.queryRange(13, 16)); // 应该输出 false
    System.out.println(rangeModule.queryRange(16, 17)); // 应该输出 true
}
=====
```

文件: Code13\_XOROnSegment.java

```
=====
package class131;

import java.util.*;

/**
 * Codeforces 242E. XOR on Segment (区间异或)
 * 题目链接: https://codeforces.com/problemset/problem/242/E
 *
 * 题目描述:
 * 给你一个数组和几种操作:
 * 1. 操作 1 l r x: 对区间[l, r]的每个元素与 x 进行异或操作
 * 2. 操作 2 l r: 查询区间[l, r]的元素和
 *
 * 解题思路:
 * 使用线段树实现, 支持区间异或和区间求和操作
 * 1. 线段树每个节点维护区间和
 * 2. 使用懒惰传播处理区间异或操作
 * 3. 对于每个二进制位维护独立的懒惰标记
 *
 * 时间复杂度分析:
 * - 区间异或: O(log n)
 * - 区间求和: O(log n)
 * - 总时间复杂度: O(30 * n * log n) (30 为整数的二进制位数)
 * 空间复杂度: O(n)
 */
public class Code13_XOROnSegment {

    /**
     * 线段树节点
     * 每个节点表示一个区间[left, right], 并维护该区间的元素和以及懒惰标记
     */
    static class Node {
        int left, right;      // 节点表示的区间范围
        long sum;             // 区间内元素的和
        int[] lazy;           // 每个二进制位的懒惰标记

        /**
         * 构造函数
         *
         * @param l 区间左边界
         * @param r 区间右边界
         */
    }
}
```

```

*/
Node(int l, int r) {
    left = l;
    right = r;
    sum = 0;
    // 只需要考虑前 20 位 (因为数值范围是 10^6, 2^20 > 10^6)
    lazy = new int[20];
}
}

private Node[] tree; // 线段树数组
private int[] nums; // 原始数组

/***
 * 构造函数, 根据给定数组构建线段树
 *
 * @param nums 初始数组
 */
public Code13_XOROnSegment(int[] nums) {
    this.nums = nums;
    int n = nums.length;
    tree = new Node[4 * n];
    build(0, 0, n - 1);
}

/***
 * 构建线段树
 *
 * @param node 当前线段树节点索引
 * @param start 区间起始位置
 * @param end   区间结束位置
 */
private void build(int node, int start, int end) {
    tree[node] = new Node(start, end);
    // 叶子节点
    if (start == end) {
        tree[node].sum = nums[start];
        return;
    }
    // 非叶子节点, 递归构建左右子树
    int mid = (start + end) / 2;
    build(2 * node + 1, start, mid);
    build(2 * node + 2, mid + 1, end);
}

```

```

// 更新当前节点的和为左右子树和的和
tree[node].sum = tree[2 * node + 1].sum + tree[2 * node + 2].sum;
}

/***
 * 下传懒惰标记
 * 将当前节点的懒惰标记传递给子节点
 *
 * @param node 当前线段树节点索引
 */
private void pushDown(int node) {
    // 检查是否有懒惰标记
    boolean hasLazy = false;
    for (int i = 0; i < 20; i++) {
        if (tree[node].lazy[i] != 0) {
            hasLazy = true;
            break;
        }
    }

    // 如果没有懒惰标记，直接返回
    if (!hasLazy) {
        return;
    }

    int leftChild = 2 * node + 1;
    int rightChild = 2 * node + 2;

    // 创建子节点（如果需要）
    if (tree[node].left != tree[node].right) {
        // 对每个二进制位处理懒惰标记
        for (int i = 0; i < 20; i++) {
            if (tree[node].lazy[i] != 0) {
                // 将标记传递给子节点（异或操作）
                tree[leftChild].lazy[i] ^= tree[node].lazy[i];
                tree[rightChild].lazy[i] ^= tree[node].lazy[i];
            }
        }
    }
}

// 应用懒惰标记到当前节点
applyLazy(node);

```

```

// 清除当前节点的懒惰标记
for (int i = 0; i < 20; i++) {
    tree[node].lazy[i] = 0;
}
}

/**
 * 应用懒惰标记到节点
 * 根据懒惰标记更新节点的 sum 值
 *
 * @param node 当前线段树节点索引
 */
private void applyLazy(int node) {
    int len = tree[node].right - tree[node].left + 1;
    // 对每个二进制位处理
    for (int i = 0; i < 20; i++) {
        if (tree[node].lazy[i] != 0) {
            // 如果该位有奇数个元素，异或操作会影响和
            // 这里的计算逻辑需要修正，应该是根据该位为 1 的元素个数来计算
            long count = ((long) len + 1) / 2; // 该位为 1 的元素个数
            tree[node].sum ^= (count * (1 << i));
        }
    }
}

/**
 * 区间异或操作
 * 对区间[start, end]的每个元素与 x 进行异或操作
 *
 * @param start 区间起始位置
 * @param end   区间结束位置
 * @param x     异或值
 */
public void xorRange(int start, int end, int x) {
    xorRange(0, start, end, x);
}

/**
 * 区间异或操作（内部实现）
 *
 * @param node 当前线段树节点索引
 * @param start 区间起始位置
 * @param end   区间结束位置
 */

```

```

* @param x      异或值
*/
private void xorRange(int node, int start, int end, int x) {
    // 当前节点区间与操作区间无交集
    if (start > tree[node].right || end < tree[node].left) {
        return;
    }

    // 当前节点区间完全包含在操作区间内
    if (start <= tree[node].left && tree[node].right <= end) {
        // 对 x 的每个为 1 的二进制位设置懒惰标记
        for (int i = 0; i < 20; i++) {
            if ((x & (1 << i)) != 0) {
                tree[node].lazy[i] ^= 1;
            }
        }
        return;
    }

    // 部分重叠，需要下传懒惰标记并递归处理
    pushDown(node);
    xorRange(2 * node + 1, start, end, x);
    xorRange(2 * node + 2, start, end, x);

    // 更新当前节点的和为左右子树和的和
    tree[node].sum = tree[2 * node + 1].sum + tree[2 * node + 2].sum;
}

/***
 * 查询区间和
 *
 * @param start  查询区间起始位置
 * @param end    查询区间结束位置
 * @return       区间内元素的和
 */
public long querySum(int start, int end) {
    return querySum(0, start, end);
}

/***
 * 查询区间和（内部实现）
 *
 * @param node  当前线段树节点索引

```

```

* @param start 查询区间起始位置
* @param end   查询区间结束位置
* @return      区间内元素的和
*/
private long querySum(int node, int start, int end) {
    // 当前节点区间与查询区间无交集
    if (start > tree[node].right || end < tree[node].left) {
        return 0;
    }

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

    // 部分重叠，需要下传懒惰标记并递归查询
    pushDown(node);
    long leftSum = querySum(2 * node + 1, start, end);
    long rightSum = querySum(2 * node + 2, start, end);

    // 返回左右子树查询结果的和
    return leftSum + rightSum;
}

/***
 * 测试方法
 *
 * @param args 命令行参数
 */
public static void main(String[] args) {
    // 示例测试
    int[] nums = {4, 1, 2, 3};
    Code13_XOROnSegment solution = new Code13_XOROnSegment(nums);

    System.out.println("初始数组和: " + solution.querySum(0, 3)); // 应该输出 10

    solution.xorRange(0, 3, 1);
    System.out.println("对区间[0,3]异或 1 后和: " + solution.querySum(0, 3)); // 应该输出 14

    solution.xorRange(1, 2, 2);
    System.out.println("对区间[1,2]异或 2 后和: " + solution.querySum(0, 3)); // 应该输出 12
}
}

```

文件: Code14\_RangeXORQuery.cpp

```
=====
/*
 * AtCoder ABC185F. Range Xor Query (C++版本)
 * 题目链接: https://atcoder.jp/contests/abc185/tasks/abc185_f
 * 题目描述: 给定一个数组, 支持两种操作:
 * 1. 更新数组中某个位置的值
 * 2. 查询区间[1, r]内所有元素的异或值
 *
 * 解题思路:
 * 使用线段树实现区间异或查询和单点更新操作
 * 1. 线段树每个节点存储对应区间的异或值
 * 2. 利用异或的性质:  $a \wedge a = 0, a \wedge 0 = a$ 
 *
 * 时间复杂度分析:
 * - 构建线段树:  $O(n)$ 
 * - 单点更新:  $O(\log n)$ 
 * - 区间查询:  $O(\log n)$ 
 * 空间复杂度:  $O(4n)$  线段树需要约  $4n$  的空间
 *
 * 算法详解:
 * 线段树是一种二叉树数据结构, 每个节点代表数组的一个区间。对于区间异或查询问题,
 * 线段树的每个节点存储其对应区间的异或值。通过递归地将区间划分为两部分, 我们可以
 * 高效地处理区间查询和单点更新操作。
 *
 * 异或运算性质:
 * 1. 交换律:  $a \wedge b = b \wedge a$ 
 * 2. 结合律:  $(a \wedge b) \wedge c = a \wedge (b \wedge c)$ 
 * 3. 自反性:  $a \wedge a = 0$ 
 * 4. 恒等性:  $a \wedge 0 = a$ 
 * 5. 逆运算:  $a \wedge b = c$  等价于  $a \wedge c = b$ 
 *
 * 线段树结构:
 * 1. 每个节点代表一个区间[1, r]
 * 2. 叶子节点代表单个元素
 * 3. 非叶子节点的值等于其左右子节点值的异或
 */

```

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

```

class SegmentTreeXOR {
private:
    vector<int> tree;      // 线段树数组，存储各区间异或值
    vector<int> nums;       // 原始数组的副本
    int n;                  // 数组大小

    /**
     * 构建线段树
     * @param node 当前节点在线段树数组中的索引
     * @param start 当前节点所代表区间的起始位置
     * @param end 当前节点所代表区间的结束位置
     */
    void build(int node, int start, int end) {
        // 如果是叶子节点，直接存储数组元素
        if (start == end) {
            tree[node] = nums[start];
        } else {
            // 计算中点，将区间分为两部分
            int mid = (start + end) / 2;
            // 递归构建左子树
            build(2 * node, start, mid);
            // 递归构建右子树
            build(2 * node + 1, mid + 1, end);
            // 当前节点的值等于左右子节点值的异或
            tree[node] = tree[2 * node] ^ tree[2 * node + 1];
        }
    }

    /**
     * 更新线段树中的值
     * @param node 当前节点在线段树数组中的索引
     * @param start 当前节点所代表区间的起始位置
     * @param end 当前节点所代表区间的结束位置
     * @param idx 要更新的数组元素索引
     * @param val 新的值
     */
    void update(int node, int start, int end, int idx, int val) {
        // 如果是叶子节点，直接更新
        if (start == end) {
            nums[idx] = val;
            tree[node] = val;
        } else {

```

```

    // 计算中点
    int mid = (start + end) / 2;
    // 根据索引位置决定更新左子树还是右子树
    if (idx <= mid) {
        update(2 * node, start, mid, idx, val);
    } else {
        update(2 * node + 1, mid + 1, end, idx, val);
    }
    // 更新当前节点的值
    tree[node] = tree[2 * node] ^ tree[2 * node + 1];
}

/**
 * 查询线段树中指定区间的异或值
 * @param node 当前节点在线段树数组中的索引
 * @param start 当前节点所代表区间的起始位置
 * @param end 当前节点所代表区间的结束位置
 * @param l 查询区间的起始位置
 * @param r 查询区间的结束位置
 * @return 查询区间的异或值
 */
int query(int node, int start, int end, int l, int r) {
    // 如果查询区间与当前节点区间无交集，返回 0（异或的单位元）
    if (r < start || end < l) {
        return 0;
    }
    // 如果当前节点区间完全包含在查询区间内，直接返回当前节点的值
    if (l <= start && end <= r) {
        return tree[node];
    }
    // 计算中点
    int mid = (start + end) / 2;
    // 递归查询左右子树
    int p1 = query(2 * node, start, mid, l, r);
    int p2 = query(2 * node + 1, mid + 1, end, l, r);
    // 返回左右子树查询结果的异或值
    return p1 ^ p2;
}

public:
/**
 * 构造函数，初始化线段树

```

```

* @param arr 输入数组
*/
SegmentTreeXOR(vector<int>& arr) {
    nums = arr;
    n = arr.size();
    tree.resize(4 * n); // 线段树数组大小通常设为 4n
    build(1, 0, n - 1); // 从根节点开始构建线段树
}

/***
* 更新指定位置的值
* @param idx 要更新的数组元素索引
* @param val 新的值
*/
void update(int idx, int val) {
    update(1, 0, n - 1, idx, val);
}

/***
* 查询区间异或值
* @param l 查询区间的起始位置
* @param r 查询区间的结束位置
* @return 查询区间的异或值
*/
int xorRange(int l, int r) {
    return query(1, 0, n - 1, l, r);
}
};

// 由于这是代码片段，不包含 main 函数和测试代码
// 在实际使用中，需要包含适当的头文件和 main 函数
=====
```

文件: Code14\_RangeXORQuery.java

=====

```

package class131;

import java.util.*;

/***
* AtCoder ABC185F. Range Xor Query
* 题目链接: https://atcoder.jp/contests/abc185/tasks/abc185\_f
```

- \* 题目描述：给定一个数组，支持两种操作：
- \* 1. 更新数组中某个位置的值
- \* 2. 查询区间 $[l, r]$ 内所有元素的异或值
- \*
- \* 解题思路：
- \* 使用线段树实现区间异或查询和单点更新操作
- \* 1. 线段树每个节点存储对应区间的异或值
- \* 2. 利用异或的性质： $a \wedge a = 0, a \wedge 0 = a$
- \*
- \* 时间复杂度分析：
- \* - 构建线段树： $O(n)$
- \* - 单点更新： $O(\log n)$
- \* - 区间查询： $O(\log n)$
- \* 空间复杂度： $O(4n)$  线段树需要约  $4n$  的空间
- \*/

```
public class Code14_RangeXORQuery {  
  
    // 线段树节点定义  
    static class SegmentTreeNode {  
        int start, end;  
        SegmentTreeNode left, right;  
        int xor; // 区间异或值  
  
        public SegmentTreeNode(int start, int end) {  
            this.start = start;  
            this.end = end;  
            this.left = null;  
            this.right = null;  
            this.xor = 0;  
        }  
    }  
  
    SegmentTreeNode root = null;  
    int[] nums;  
  
    public Code14_RangeXORQuery(int[] nums) {  
        this.nums = nums.clone();  
        root = buildTree(nums, 0, nums.length - 1);  
    }  
  
    // 构建线段树  
    private SegmentTreeNode buildTree(int[] nums, int start, int end) {  
        if (start > end) {  
            return null;  
        }  
        SegmentTreeNode node = new SegmentTreeNode(start, end);  
        if (start == end) {  
            node.xor = nums[start];  
        } else {  
            int mid = (start + end) / 2;  
            node.left = buildTree(nums, start, mid);  
            node.right = buildTree(nums, mid + 1, end);  
            node.xor = node.left.xor ^ node.right.xor;  
        }  
        return node;  
    }  
}
```

```
        return null;
    }

SegmentTreeNode ret = new SegmentTreeNode(start, end);
// 叶子节点
if (start == end) {
    ret.xor = nums[start];
} else {
    // 递归构建左右子树
    int mid = start + (end - start) / 2;
    ret.left = buildTree(nums, start, mid);
    ret.right = buildTree(nums, mid + 1, end);
    // 更新当前节点的异或值
    ret.xor = ret.left.xor ^ ret.right.xor;
}
return ret;
}

// 更新指定位置的值
public void update(int i, int val) {
    update(root, i, val);
}

// 更新线段树中的值
private void update(SegmentTreeNode root, int pos, int val) {
    // 叶子节点，直接更新
    if (root.start == root.end) {
        root.xor = val;
        nums[pos] = val;
    } else {
        // 非叶子节点，递归更新
        int mid = root.start + (root.end - root.start) / 2;
        if (pos <= mid) {
            update(root.left, pos, val);
        } else {
            update(root.right, pos, val);
        }
        // 更新当前节点的异或值
        root.xor = root.left.xor ^ root.right.xor;
    }
}

// 查询区间异或值
```

```

public int xorRange(int i, int j) {
    return xorRange(root, i, j);
}

// 查询线段树中指定区间的异或值
private int xorRange(SegmentTreeNode root, int start, int end) {
    // 完全匹配
    if (root.start == start && root.end == end) {
        return root.xor;
    } else {
        int mid = root.start + (root.end - root.start) / 2;
        // 完全在左子树
        if (end <= mid) {
            return xorRange(root.left, start, end);
        }
        // 完全在右子树
        else if (start >= mid + 1) {
            return xorRange(root.right, start, end);
        }
        // 跨越左右子树
        else {
            return xorRange(root.left, start, mid) ^ xorRange(root.right, mid + 1, end);
        }
    }
}

// 测试方法
public static void main(String[] args) {
    // 示例测试
    int[] nums = {1, 3, 5, 7, 9};
    Code14_RangeXORQuery solution = new Code14_RangeXORQuery(nums);

    System.out.println("初始数组: " + Arrays.toString(nums));
    System.out.println("区间[0,2]的异或值: " + solution.xorRange(0, 2)); // 1^3^5 = 7
    System.out.println("区间[1,3]的异或值: " + solution.xorRange(1, 3)); // 3^5^7 = 1

    solution.update(1, 2); // 将索引 1 的值从 3 更新为 2
    System.out.println("更新索引 1 的值为 2 后:");
    System.out.println("区间[0,2]的异或值: " + solution.xorRange(0, 2)); // 1^2^5 = 6
    System.out.println("区间[1,3]的异或值: " + solution.xorRange(1, 3)); // 2^5^7 = 0
}
}

```

文件: Code14\_RangeXORQuery.py

AtCoder ABC185F. Range Xor Query (Python 版本)

题目链接: [https://atcoder.jp/contests/abc185/tasks/abc185\\_f](https://atcoder.jp/contests/abc185/tasks/abc185_f)

题目描述: 给定一个数组, 支持两种操作:

1. 更新数组中某个位置的值
2. 查询区间 $[l, r]$ 内所有元素的异或值

解题思路:

使用线段树实现区间异或查询和单点更新操作

1. 线段树每个节点存储对应区间的异或值
2. 利用异或的性质:  $a \wedge a = 0$ ,  $a \wedge 0 = a$

时间复杂度分析:

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

空间复杂度:  $O(4n)$  线段树需要约  $4n$  的空间

"""

```
class SegmentTreeXOR:
```

```
    def __init__(self, arr):
```

```
        """
```

```
    初始化线段树
```

```
    :param arr: 输入数组
```

```
        """
```

```
    self.n = len(arr)
```

```
    self.nums = arr[:]
```

```
    # 线段树数组, 大小为 4*n, 使用 1-based indexing
```

```
    self.tree = [0] * (4 * self.n)
```

```
    # 构建线段树
```

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

```
    def _build(self, node, start, end):
```

```
        """
```

```
    构建线段树
```

```
    递归地构建线段树, 每个节点存储对应区间的异或值
```

```
    :param node: 当前节点索引 (1-based)
```

```
    :param start: 区间起始位置 (0-based)
```

```

:param end: 区间结束位置 (0-based)
"""

# 叶子节点: 区间只包含一个元素
if start == end:
    self.tree[node] = self.nums[start]
else:
    # 非叶子节点: 递归构建左右子树
    mid = (start + end) // 2
    # 构建左子树
    self._build(2 * node, start, mid)
    # 构建右子树
    self._build(2 * node + 1, mid + 1, end)
    # 当前节点的值为左右子节点值的异或
    self.tree[node] = self.tree[2 * node] ^ self.tree[2 * node + 1]

def _update(self, node, start, end, idx, val):
    """
更新线段树中的值
递归查找目标位置并更新，然后向上回溯更新父节点的值
:param node: 当前节点索引 (1-based)
:param start: 区间起始位置 (0-based)
:param end: 区间结束位置 (0-based)
:param idx: 要更新的数组索引 (0-based)
:param val: 新的值
"""

    # 叶子节点，直接更新
    if start == end:
        self.nums[idx] = val
        self.tree[node] = val
    else:
        # 非叶子节点，递归更新
        mid = (start + end) // 2
        # 根据索引位置决定更新左子树还是右子树
        if idx <= mid:
            self._update(2 * node, start, mid, idx, val)
        else:
            self._update(2 * node + 1, mid + 1, end, idx, val)
        # 更新当前节点的值为左右子节点值的异或
        self.tree[node] = self.tree[2 * node] ^ self.tree[2 * node + 1]

def _query(self, node, start, end, l, r):
    """
查询线段树中指定区间的异或值

```

```

根据查询区间与当前节点区间的关系，决定是直接返回、递归查询还是分段查询
:param node: 当前节点索引（1-based）
:param start: 当前节点区间起始位置（0-based）
:param end: 当前节点区间结束位置（0-based）
:param l: 查询区间起始位置（0-based）
:param r: 查询区间结束位置（0-based）
:return: 区间[l, r]内元素的异或值
"""

# 当前节点区间与查询区间无交集，返回 0（异或的单位元）
if r < start or end < l:
    return 0

# 当前节点区间完全包含在查询区间内，直接返回节点值
if l <= start and end <= r:
    return self.tree[node]

# 部分重叠，需要递归查询
mid = (start + end) // 2
# 递归查询左子树
p1 = self._query(2 * node, start, mid, l, r)
# 递归查询右子树
p2 = self._query(2 * node + 1, mid + 1, end, l, r)
# 返回左右子树查询结果的异或
return p1 ^ p2

def update(self, idx, val):
    """
更新指定位置的值
:param idx: 数组索引（0-based）
:param val: 新的值
"""
    self._update(1, 0, self.n - 1, idx, val)

def xor_range(self, l, r):
    """
查询区间异或值
:param l: 区间起始位置（0-based）
:param r: 区间结束位置（0-based）
:return: 区间[l, r]内元素的异或值
"""
    return self._query(1, 0, self.n - 1, l, r)

# 测试代码
if __name__ == "__main__":

```

```

# 示例测试
nums = [1, 3, 5, 7, 9]
solution = SegmentTreeXOR(nums)

print("初始数组:", nums)
print("区间[0,2]的异或值:", solution.xor_range(0, 2)) # 1^3^5 = 7
print("区间[1,3]的异或值:", solution.xor_range(1, 3)) # 3^5^7 = 1

solution.update(1, 2) # 将索引 1 的值从 3 更新为 2
print("更新索引 1 的值为 2 后:")
print("区间[0,2]的异或值:", solution.xor_range(0, 2)) # 1^2^5 = 6
print("区间[1,3]的异或值:", solution.xor_range(1, 3)) # 2^5^7 = 0

```

=====

文件: Code15\_MaximumSubarraySum.java

=====

```

package class131;

import java.util.*;

/**
 * SPOJ GSS1. Can you answer these queries I
 * 题目链接: https://www.spoj.com/problems/GSS1/
 * 题目描述: 给定一个数组, 查询区间[1, r]内的最大子段和
 *
 * 解题思路:
 * 使用线段树实现区间最大子段和查询
 * 每个节点维护四个值:
 * 1. 区间和(sum)
 * 2. 区间最大子段和(maxSum)
 * 3. 包含左端点的最大子段和(prefixMaxSum)
 * 4. 包含右端点的最大子段和(suffixMaxSum)
 *
 * 时间复杂度分析:
 * - 构建线段树: O(n)
 * - 区间查询: O(log n)
 * 空间复杂度: O(4n) 线段树需要约 4n 的空间
 */

public class Code15_MaximumSubarraySum {

    // 线段树节点定义
    static class SegmentTreeNode {

```

```

int start, end;
SegmentTreeNode left, right;
int sum;           // 区间和
int maxSum;        // 区间最大子段和
int prefixMaxSum; // 包含左端点的最大子段和
int suffixMaxSum; // 包含右端点的最大子段和

public SegmentTreeNode(int start, int end) {
    this.start = start;
    this.end = end;
    this.left = null;
    this.right = null;
    this.sum = 0;
    this.maxSum = Integer.MIN_VALUE;
    this.prefixMaxSum = Integer.MIN_VALUE;
    this.suffixMaxSum = Integer.MIN_VALUE;
}

}

SegmentTreeNode root = null;

public Code15_MaximumSubarraySum(int[] nums) {
    root = buildTree(nums, 0, nums.length - 1);
}

// 构建线段树
private SegmentTreeNode buildTree(int[] nums, int start, int end) {
    if (start > end) {
        return null;
    }

    SegmentTreeNode ret = new SegmentTreeNode(start, end);
    // 叶子节点
    if (start == end) {
        ret.sum = nums[start];
        ret.maxSum = nums[start];
        ret.prefixMaxSum = nums[start];
        ret.suffixMaxSum = nums[start];
    } else {
        // 递归构建左右子树
        int mid = start + (end - start) / 2;
        ret.left = buildTree(nums, start, mid);
        ret.right = buildTree(nums, mid + 1, end);
    }
}

```

```

// 更新当前节点的值
ret.sum = ret.left.sum + ret.right.sum;
ret.prefixMaxSum = Math.max(ret.left.prefixMaxSum, ret.left.sum +
ret.right.prefixMaxSum);
ret.suffixMaxSum = Math.max(ret.right.suffixMaxSum, ret.right.sum +
ret.left.suffixMaxSum);
ret.maxSum = Math.max(Math.max(ret.left.maxSum, ret.right.maxSum),
ret.left.suffixMaxSum + ret.right.prefixMaxSum);
}

return ret;
}

// 查询区间最大子段和
public int maxSubarraySum(int i, int j) {
    return maxSubarraySum(root, i, j).maxSum;
}

// 查询线段树中指定区间的最大子段和信息
private SegmentTreeNode maxSubarraySum(SegmentTreeNode root, int start, int end) {
    // 完全匹配
    if (root.start == start && root.end == end) {
        return root;
    } else {
        int mid = root.start + (root.end - root.start) / 2;
        // 完全在左子树
        if (end <= mid) {
            return maxSubarraySum(root.left, start, end);
        }
        // 完全在右子树
        else if (start >= mid + 1) {
            return maxSubarraySum(root.right, start, end);
        }
        // 跨越左右子树
        else {
            SegmentTreeNode leftResult = maxSubarraySum(root.left, start, mid);
            SegmentTreeNode rightResult = maxSubarraySum(root.right, mid + 1, end);

            SegmentTreeNode result = new SegmentTreeNode(start, end);
            result.sum = leftResult.sum + rightResult.sum;
            result.prefixMaxSum = Math.max(leftResult.prefixMaxSum, leftResult.sum +
rightResult.prefixMaxSum);
            result.suffixMaxSum = Math.max(rightResult.suffixMaxSum, rightResult.sum +
leftResult.suffixMaxSum);
        }
    }
}

```

```

        result.maxSum = Math.max(Math.max(leftResult.maxSum, rightResult.maxSum),
leftResult.suffixMaxSum + rightResult.prefixMaxSum);
        return result;
    }
}
}

// 测试方法
public static void main(String[] args) {
    // 示例测试
    int[] nums = {1, -2, 3, 4, -5, 6};
    Code15_MaximumSubarraySum solution = new Code15_MaximumSubarraySum(nums);

    System.out.println("初始数组: " + Arrays.toString(nums));
    System.out.println("区间[0,5]的最大子段和: " + solution.maxSubarraySum(0, 5)); // 3+4-5+6
= 8
    System.out.println("区间[1,4]的最大子段和: " + solution.maxSubarraySum(1, 4)); // 3+4 = 7
    System.out.println("区间[2,3]的最大子段和: " + solution.maxSubarraySum(2, 3)); // max(3,
4, 3+4) = 7
}
}

```

=====

文件: Code15\_MaximumSubarraySum.py

=====

"""

SPOJ GSS1. Can you answer these queries I (Python 版本)

题目链接: <https://www.spoj.com/problems/GSS1/>

题目描述: 给定一个数组, 查询区间 $[l, r]$ 内的最大子段和

解题思路:

使用线段树实现区间最大子段和查询

每个节点维护四个值:

1. 区间和(sum)
2. 区间最大子段和(maxSum)
3. 包含左端点的最大子段和(prefixMaxSum)
4. 包含右端点的最大子段和(suffixMaxSum)

时间复杂度分析:

- 构建线段树:  $O(n)$
- 区间查询:  $O(\log n)$

空间复杂度:  $O(4n)$  线段树需要约  $4n$  的空间

```
"""
```

```
class MaximumSubarraySum:  
    def __init__(self, nums):  
        """  
        初始化线段树  
        :param nums: 输入数组  
        """  
  
        self.n = len(nums)  
        self.nums = nums[:]  
        # 线段树数组，每个节点存储(sum, maxSum, prefixMaxSum, suffixMaxSum)  
        self.tree = [(0, float('-inf'), float('-inf'), float('-inf'))] * (4 * self.n)  
        # 构建线段树  
        self._build(1, 0, self.n - 1)  
  
    def _build(self, node, start, end):  
        """  
        构建线段树  
        :param node: 当前节点索引  
        :param start: 区间起始位置  
        :param end: 区间结束位置  
        """  
  
        if start == end:  
            val = self.nums[start]  
            self.tree[node] = (val, val, val, val)  
        else:  
            mid = (start + end) // 2  
            self._build(2 * node, start, mid)  
            self._build(2 * node + 1, mid + 1, end)  
  
            # 获取左右子树的信息  
            left_sum, left_max_sum, left_prefix_max, left_suffix_max = self.tree[2 * node]  
            right_sum, right_max_sum, right_prefix_max, right_suffix_max = self.tree[2 * node + 1]  
  
            # 计算当前节点的信息  
            sum_val = left_sum + right_sum  
            prefix_max = max(left_prefix_max, left_sum + right_prefix_max)  
            suffix_max = max(right_suffix_max, right_sum + left_suffix_max)  
            max_sum = max(left_max_sum, right_max_sum, left_suffix_max + right_prefix_max)  
  
            self.tree[node] = (sum_val, max_sum, prefix_max, suffix_max)
```

```

def _query(self, node, start, end, l, r):
    """
    查询线段树中指定区间的最大子段和信息
    :param node: 当前节点索引
    :param start: 区间起始位置
    :param end: 区间结束位置
    :param l: 查询区间起始位置
    :param r: 查询区间结束位置
    :return: (sum, maxSum, prefixMaxSum, suffixMaxSum)
    """

    if r < start or end < l:
        return (0, float('-inf'), float('-inf'), float('-inf'))
    if l <= start and end <= r:
        return self.tree[node]

    mid = (start + end) // 2
    left_result = self._query(2 * node, start, mid, l, r)
    right_result = self._query(2 * node + 1, mid + 1, end, l, r)

    # 合并左右子树的结果
    left_sum, left_max_sum, left_prefix_max, left_suffix_max = left_result
    right_sum, right_max_sum, right_prefix_max, right_suffix_max = right_result

    sum_val = left_sum + right_sum
    prefix_max = max(left_prefix_max, left_sum + right_prefix_max)
    suffix_max = max(right_suffix_max, right_sum + left_suffix_max)
    max_sum = max(left_max_sum, right_max_sum, left_suffix_max + right_prefix_max)

    return (sum_val, max_sum, prefix_max, suffix_max)

def max_subarray_sum(self, l, r):
    """
    查询区间最大子段和
    :param l: 查询区间起始位置
    :param r: 查询区间结束位置
    :return: 区间最大子段和
    """

    _, max_sum, _, _ = self._query(1, 0, self.n - 1, l, r)
    return max_sum

# 测试代码
if __name__ == "__main__":
    # 示例测试

```

```

nums = [1, -2, 3, 4, -5, 6]
solution = MaximumSubarraySum(nums)

print("初始数组:", nums)
print("区间[0,5]的最大子段和:", solution.max_subarray_sum(0, 5)) # 3+4-5+6 = 8
print("区间[1,4]的最大子段和:", solution.max_subarray_sum(1, 4)) # 3+4 = 7
print("区间[2,3]的最大子段和:", solution.max_subarray_sum(2, 3)) # max(3, 4, 3+4) = 7

```

=====

文件: Code16\_KthNumber.cpp

=====

```

// class131/Code16_KthNumber.cpp
// SPOJ MKTHNUM - K-th Number
// 题目链接: https://www.spoj.com/problems/MKTHNUM/

```

```

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

```

```

const int MAXN = 100005;
const int MAX_LOG = 20;

```

```

/***
 * 主席树节点定义
 * 主席树是一种可持久化线段树，支持历史版本查询
 */

```

```

struct Node {
    int left, right; // 左右子节点索引
    int count; // 区间内元素个数
} tree[MAXN * MAX_LOG]; // 预分配空间

```

```

int root[MAXN]; // 各个版本的根节点
int a[MAXN]; // 原始数组
vector<int> sorted; // 离散化后的数组
int cnt; // 动态开点计数器
int n, m; // 数组大小和查询数量

```

```

/***
 * 离散化: 获取数值对应的索引
 * @param val 要查找的数值
 * @return 离散化后的索引 (从 1 开始)

```

```

*/
int getIndex(int val) {
    return lower_bound(sorted.begin(), sorted.end(), val) - sorted.begin() + 1; // 索引从 1 开始
}

/**
 * 构建初始线段树
 * @param l 当前节点区间左边界
 * @param r 当前节点区间右边界
 * @return 构建好的线段树根节点索引
*/
int build(int l, int r) {
    int node = ++cnt;
    tree[node].count = 0;

    if (l == r) {
        return node;
    }

    int mid = (l + r) >> 1;
    tree[node].left = build(l, mid);
    tree[node].right = build(mid + 1, r);

    return node;
}

/**
 * 更新线段树，创建新版本
 * @param pre 前一个版本的节点索引
 * @param l 当前节点区间左边界
 * @param r 当前节点区间右边界
 * @param pos 要更新的位置
 * @param val 更新的值（1 表示插入，-1 表示删除）
 * @return 更新后的新节点索引
*/
int update(int pre, int l, int r, int pos, int val) {
    int node = ++cnt;
    tree[node] = tree[pre]; // 复制前一个版本的节点信息
    tree[node].count += val; // 更新计数

    if (l == r) {
        return node;
    }
}

```

```

int mid = (l + r) >> 1;
if (pos <= mid) {
    tree[node].left = update(tree[pre].left, l, mid, pos, val);
} else {
    tree[node].right = update(tree[pre].right, mid + 1, r, pos, val);
}

return node;
}

```

```

/**
 * 查询区间第 k 小元素
 * 利用主席树的前缀和思想，通过两个版本的差值来得到区间内的信息
 * @param u 右边界版本的根节点
 * @param v 左边界-1 版本的根节点
 * @param l 当前节点区间左边界
 * @param r 当前节点区间右边界
 * @param k 要查询的第 k 小
 * @return 第 k 小元素的离散化索引
*/

```

```

int query(int u, int v, int l, int r, int k) {
    if (l == r) {
        return l;
    }

    int mid = (l + r) >> 1;
    int leftCount = tree[tree[u].left].count - tree[tree[v].left].count;

    if (leftCount >= k) {
        return query(tree[u].left, tree[v].left, l, mid, k);
    } else {
        return query(tree[u].right, tree[v].right, mid + 1, r, k - leftCount);
    }
}

```

```

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

    cin >> n >> m;

    // 读取原始数组并准备离散化
}

```

```

sorted.resize(n);
for (int i = 1; i <= n; i++) {
    cin >> a[i];
    sorted[i - 1] = a[i];
}

// 离散化处理: 排序并去重
sort(sorted.begin(), sorted.end());
sorted.erase(unique(sorted.begin(), sorted.end()), sorted.end());

// 构建主席树
cnt = 0;
root[0] = build(1, sorted.size());

// 创建前缀版本
for (int i = 1; i <= n; i++) {
    int idx = getIndex(a[i]);
    root[i] = update(root[i - 1], 1, sorted.size(), idx, 1);
}

// 处理查询
while (m--) {
    int l, r, k;
    cin >> l >> r >> k;
    int idx = query(root[r], root[l - 1], 1, sorted.size(), k);
    cout << sorted[idx - 1] << "\n"; // 注意离散化索引的转换
}

return 0;
}

```

/\*

复杂度分析:

- 时间复杂度:

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

主席树（可持久化线段树）是一种支持历史版本查询的数据结构，每个版本都是基于前一个版本进行增量修改。

在区间第  $k$  小问题中，我们利用前缀和的思想，通过两个版本的差值来得到区间内的信息。

算法详解:

1. 离散化：将原始数值映射到连续的整数范围，减少空间消耗
2. 构建主席树：为每个前缀建立一个版本的线段树
3. 查询：通过比较两个版本的差异来获得区间信息

优化注意事项：

1. 离散化是必须的，否则权值范围太大导致空间不足
  2. 预分配足够的空间以避免运行时错误
  3. 对于大规模数据，可以进一步优化内存使用
  4. 可以使用更高效的离散化方法和输入方式
- \*/
- =====

文件：Code16\_KthNumber.java

=====

```
package class131;

import java.util.*;

/**
 * SPOJ MKTHNUM. K-th Number (主席树解法)
 * 题目链接: https://www.spoj.com/problems/MKTHNUM/
 * 题目描述: 给定一个数组, 查询区间[1, r]内第 k 小的元素
 *
 * 解题思路:
 * 使用主席树(可持久化线段树)实现区间第 k 小元素查询
 * 1. 对数组元素进行离散化处理
 * 2. 构建主席树, 每个版本表示前缀数组的信息
 * 3. 通过两个版本的差值查询区间信息
 *
 * 时间复杂度分析:
 * - 构建主席树: O(n log n)
 * - 区间查询: O(log n)
 * 空间复杂度: O(n log n) 主席树需要约 n log n 的空间
 */

public class Code16_KthNumber {

    // 主席树节点定义
    static class ChairmanTreeNode {
        int left, right;
        int count; // 区间内元素个数
        ChairmanTreeNode leftChild, rightChild;
    }
}
```

```

public ChairmanTreeNode(int left, int right) {
    this.left = left;
    this.right = right;
    this.count = 0;
    this.leftChild = null;
    this.rightChild = null;
}

private int[] nums;
private int[] sorted;
private Map<Integer, Integer> ranks;
private ChairmanTreeNode[] roots;
private int versionCount;

public Code16_KthNumber(int[] nums) {
    this.nums = nums.clone();
    this.versionCount = nums.length;
    this.roots = new ChairmanTreeNode[versionCount + 1];

    // 离散化处理
    this.sorted = nums.clone();
    Arrays.sort(sorted);
    this.ranks = new HashMap<>();
    int rank = 1;
    for (int num : sorted) {
        if (!ranks.containsKey(num)) {
            ranks.put(num, rank++);
        }
    }
}

// 构建主席树
build();
}

// 构建主席树
private void build() {
    roots[0] = new ChairmanTreeNode(1, ranks.size());
    for (int i = 1; i <= versionCount; i++) {
        roots[i] = update(roots[i - 1], ranks.get(nums[i - 1]));
    }
}

```

```

// 更新主席树，插入一个元素
private ChairmanTreeNode update(ChairmanTreeNode prev, int value) {
    ChairmanTreeNode root = new ChairmanTreeNode(prev.left, prev.right);
    root.count = prev.count + 1;

    if (prev.left == prev.right) {
        return root;
    }

    int mid = (prev.left + prev.right) / 2;
    if (value <= mid) {
        root.leftChild = update(prev.leftChild != null ? prev.leftChild : new
ChairmanTreeNode(prev.left, mid), value);
        root.rightChild = prev.rightChild;
    } else {
        root.leftChild = prev.leftChild;
        root.rightChild = update(prev.rightChild != null ? prev.rightChild : new
ChairmanTreeNode(mid + 1, prev.right), value);
    }
}

return root;
}

// 查询区间第 k 小元素
public int kthNumber(int left, int right, int k) {
    return query(roots[left - 1], roots[right], k);
}

// 查询两个版本的差值中的第 k 小元素
private int query(ChairmanTreeNode leftRoot, ChairmanTreeNode rightRoot, int k) {
    if (leftRoot.left == leftRoot.right) {
        return sorted[leftRoot.left - 1];
    }

    int leftCount = (rightRoot.leftChild != null ? rightRoot.leftChild.count : 0) -
                    (leftRoot.leftChild != null ? leftRoot.leftChild.count : 0);

    if (k <= leftCount) {
        return query(leftRoot.leftChild != null ? leftRoot.leftChild : new
ChairmanTreeNode(leftRoot.left, (leftRoot.left + leftRoot.right) / 2),
                    rightRoot.leftChild != null ? rightRoot.leftChild : new
ChairmanTreeNode(rightRoot.left, (rightRoot.left + rightRoot.right) / 2),
                    k);
    }
}

```

```

    } else {
        return query(leftRoot.rightChild != null ? leftRoot.rightChild : new
ChairmanTreeNode((leftRoot.left + leftRoot.right) / 2 + 1, leftRoot.right),
                rightRoot.rightChild != null ? rightRoot.rightChild : new
ChairmanTreeNode((rightRoot.left + rightRoot.right) / 2 + 1, rightRoot.right),
                k - leftCount);
    }
}

// 测试方法
public static void main(String[] args) {
    // 示例测试
    int[] nums = {1, 5, 2, 6, 3, 7, 4};
    Code16_KthNumber solution = new Code16_KthNumber(nums);

    System.out.println("初始数组: " + Arrays.toString(nums));
    System.out.println("区间[2,5]第 2 小的元素: " + solution.kthNumber(2, 5, 2)); // [5,2,6,3]
    中第 2 小的是 3
    System.out.println("区间[1,7]第 3 小的元素: " + solution.kthNumber(1, 7, 3)); // 全部元素
    中第 3 小的是 3
    System.out.println("区间[4,6]第 1 小的元素: " + solution.kthNumber(4, 6, 1)); // [6,3,7]
    中第 1 小的是 3
}
}

```

=====

文件: Code16\_KthNumber.py

=====

"""

SPOJ MKTHNUM. K-th Number (Python 版本 - 主席树解法)

题目链接: <https://www.spoj.com/problems/MKTHNUM/>

题目描述: 给定一个数组, 查询区间 $[l, r]$ 内第  $k$  小的元素

解题思路:

使用主席树(可持久化线段树)实现区间第  $k$  小元素查询

1. 对数组元素进行离散化处理
2. 构建主席树, 每个版本表示前缀数组的信息
3. 通过两个版本的差值查询区间信息

时间复杂度分析:

- 构建主席树:  $O(n \log n)$
- 区间查询:  $O(\log n)$

空间复杂度:  $O(n \log n)$  主席树需要约  $n \log n$  的空间

算法详解:

主席树是一种可持久化线段树，它能够保存历史版本的信息。在本题中，我们为数组的每个前缀构建一个线段树版本，

每个版本记录了对应前缀中各个数值的出现次数。通过比较两个版本的差异，我们可以得到任意区间的元素分布情况。

离散化处理:

由于输入数组中的元素值可能很大，我们需要对其进行离散化处理，将其映射到较小的连续整数范围内，这样可以大大减少线段树的空间消耗。

查询过程:

对于区间  $[l, r]$  的第  $k$  小查询，我们通过比较第  $r$  个版本和第  $(l-1)$  个版本的差异来获得区间信息，然后在线段树上进行二分查找找到第  $k$  小的元素。

"""

```
class ChairmanTreeNode:
    def __init__(self, left, right):
        """
        主席树节点构造函数
        :param left: 区间左边界
        :param right: 区间右边界
        """
        self.left = left
        self.right = right
        self.count = 0 # 区间内元素个数
        self.left_child = None # type: ChairmanTreeNode | None
        self.right_child = None # type: ChairmanTreeNode | None

class KthNumber:
    def __init__(self, nums):
        """
        初始化主席树
        :param nums: 输入数组
        """
        self.nums = nums[:]
        self.version_count = len(nums)

        # 离散化处理: 将原始数值映射到连续的整数范围
        self.sorted_nums = sorted(set(nums))
        self.ranks = {num: i + 1 for i, num in enumerate(self.sorted_nums)} # 数值到排名的映射
        self.values = {i + 1: num for i, num in enumerate(self.sorted_nums)} # 排名到数值的映射
```

```

# 构建主席树
self.roots = [None] * (self.version_count + 1)  # type: list[ChairmanTreeNode | None]
self.build()

def build(self):
    """
    构建主席树，为每个前缀建立一个版本
    """

    # 初始化第 0 个版本（空树）
    self.roots[0] = ChairmanTreeNode(1, len(self.sorted_nums))
    # 逐个添加元素，构建第 1 到第 n 个版本
    for i in range(1, self.version_count + 1):
        self.roots[i] = self._update(self.roots[i - 1], self.ranks[self.nums[i - 1]])

def _update(self, prev, value):
    """
    更新主席树，插入一个元素
    :param prev: 前一个版本的节点
    :param value: 要插入的值(离散化后的 rank)
    :return: 新节点
    """

    # 创建新节点，复制前一个版本的信息
    root = ChairmanTreeNode(prev.left, prev.right)
    root.count = prev.count + 1  # 计数加 1

    # 如果是叶子节点，直接返回
    if prev.left == prev.right:
        return root

    # 计算中点，决定更新左子树还是右子树
    mid = (prev.left + prev.right) // 2
    if value <= mid:
        # 更新左子树，右子树保持不变
        root.left_child = self._update(prev.left_child if prev.left_child else
ChairmanTreeNode(prev.left, mid), value)
        root.right_child = prev.right_child
    else:
        # 更新右子树，左子树保持不变
        root.left_child = prev.left_child
        root.right_child = self._update(prev.right_child if prev.right_child else
ChairmanTreeNode(mid + 1, prev.right), value)

```

```

    return root

def kth_number(self, left, right, k):
    """
    查询区间第 k 小元素
    :param left: 区间左端点(1-indexed)
    :param right: 区间右端点(1-indexed)
    :param k: 第 k 小
    :return: 第 k 小的元素值
    """
    return self._query(self.roots[left - 1], self.roots[right], k)

def _query(self, left_root, right_root, k):
    """
    查询两个版本的差值中的第 k 小元素
    :param left_root: 左版本根节点
    :param right_root: 右版本根节点
    :param k: 第 k 小
    :return: 第 k 小的元素值
    """
    # 如果是叶子节点，直接返回对应的值
    if left_root.left == left_root.right:
        return self.values[left_root.left]

    # 计算左子树中元素的个数差值
    left_count = (right_root.left_child.count if right_root.left_child else 0) - \
                 (left_root.left_child.count if left_root.left_child else 0)

    # 根据 k 的大小决定在左子树还是右子树中查找
    if k <= left_count:
        # 在左子树中查找第 k 小
        return self._query(left_root.left_child if left_root.left_child else
ChairmanTreeNode(left_root.left, (left_root.left + left_root.right) // 2),
                    right_root.left_child if right_root.left_child else
ChairmanTreeNode(right_root.left, (right_root.left + right_root.right) // 2),
                    k)
    else:
        # 在右子树中查找第 (k-left_count) 小
        return self._query(left_root.right_child if left_root.right_child else
ChairmanTreeNode((left_root.left + left_root.right) // 2 + 1, left_root.right),
                    right_root.right_child if right_root.right_child else
ChairmanTreeNode((right_root.left + right_root.right) // 2 + 1, right_root.right),
                    k - left_count)

```

```

# 测试代码
if __name__ == "__main__":
    # 示例测试
    nums = [1, 5, 2, 6, 3, 7, 4]
    solution = KthNumber(nums)

    print("初始数组:", nums)
    print("区间[2,5]第 2 小的元素:", solution.kth_number(2, 5, 2)) # [5,2,6,3]中第 2 小的是 3
    print("区间[1,7]第 3 小的元素:", solution.kth_number(1, 7, 3)) # 全部元素中第 3 小的是 3
    print("区间[4,6]第 1 小的元素:", solution.kth_number(4, 6, 1)) # [6,3,7]中第 1 小的是 3

```

---

文件: Code17\_SegmentTreeMerge.cpp

---

```

// class131/Code17_SegmentTreeMerge.cpp
// 洛谷 P4556 (Vani 有约会) 雨天的尾巴 / 【模板】线段树合并
// 题目链接: https://www.luogu.com.cn/problem/P4556

```

```

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

const int MAXN = 100010;

// 线段树节点定义
struct Node {
    int left, right; // 左右子节点索引
    int val;          // 节点值（救济粮数量）
    int maxVal;       // 最大值
    int maxType;      // 最大值对应的救济粮类型
} tree[MAXN * 40]; // 动态开点线段树，空间要足够大

```

```

vector<int> graph[MAXN]; // 树的邻接表
int root[MAXN];           // 每个节点对应的线段树根节点
int ans[MAXN];            // 每个节点的答案
int cnt;                  // 动态开点计数器
int maxType;              // 最大救济粮类型

// LCA 相关

```

```

int up[20][MAXN];           // 祖先表, 2^k 级祖先
int depth[MAXN];           // 深度数组
int LOG;                   // log2(n) 向上取整

// 初始化线段树节点
void initNode(int node) {
    tree[node].left = tree[node].right = 0;
    tree[node].val = 0;
    tree[node].maxVal = 0;
    tree[node].maxType = 0;
}

// 线段树更新操作
void update(int &root, int l, int r, int pos, int val) {
    if (!root) {
        root = ++cnt;
        initNode(root);
    }

    if (l == r) {
        tree[root].val += val;
        tree[root].maxVal = tree[root].val;
        tree[root].maxType = 1;
        return;
    }

    int mid = (l + r) >> 1;
    if (pos <= mid) {
        update(tree[root].left, l, mid, pos, val);
    } else {
        update(tree[root].right, mid + 1, r, pos, val);
    }
}

// 更新当前节点的 maxVal 和 maxType
tree[root].maxVal = 0;
tree[root].maxType = 0;

// 比较左子树
if (tree[root].left) {
    if (tree[tree[root].left].maxVal > tree[root].maxVal ||
        (tree[tree[root].left].maxVal == tree[root].maxVal && tree[tree[root].left].maxType <
tree[root].maxType)) {
        tree[root].maxVal = tree[tree[root].left].maxVal;
    }
}

```

```

        tree[root].maxType = tree[tree[root].left].maxType;
    }
}

// 比较右子树
if (tree[root].right) {
    if (tree[tree[root].right].maxVal > tree[root].maxVal ||
        (tree[tree[root].right].maxVal == tree[root].maxVal && tree[tree[root].right].maxType
        < tree[root].maxType)) {
        tree[root].maxVal = tree[tree[root].right].maxVal;
        tree[root].maxType = tree[tree[root].right].maxType;
    }
}
}

// 线段树合并操作
int merge(int a, int b, int l, int r) {
    if (!a) return b;
    if (!b) return a;

    if (l == r) {
        tree[a].val += tree[b].val;
        tree[a].maxVal = tree[a].val;
        tree[a].maxType = 1;
        return a;
    }

    int mid = (l + r) >> 1;
    tree[a].left = merge(tree[a].left, tree[b].left, l, mid);
    tree[a].right = merge(tree[a].right, tree[b].right, mid + 1, r);

    // 更新当前节点的 maxVal 和 maxType
    tree[a].maxVal = 0;
    tree[a].maxType = 0;

    // 比较左子树
    if (tree[a].left) {
        if (tree[tree[a].left].maxVal > tree[a].maxVal ||
            (tree[tree[a].left].maxVal == tree[a].maxVal && tree[tree[a].left].maxType <
            tree[a].maxType)) {
            tree[a].maxVal = tree[tree[a].left].maxVal;
            tree[a].maxType = tree[tree[a].left].maxType;
        }
    }
}

```

```

}

// 比较右子树
if (tree[a].right) {
    if (tree[tree[a].right].maxVal > tree[a].maxVal ||
        (tree[tree[a].right].maxVal == tree[a].maxVal && tree[tree[a].right].maxType <
tree[a].maxType)) {
        tree[a].maxVal = tree[tree[a].right].maxVal;
        tree[a].maxType = tree[tree[a].right].maxType;
    }
}

return a;
}

// LCA 预处理 - DFS
void dfsLCA(int u, int parent) {
    depth[u] = depth[parent] + 1;
    up[0][u] = parent;

    for (int k = 1; k < LOG; k++) {
        up[k][u] = up[k-1][up[k-1][u]];
    }

    for (int v : graph[u]) {
        if (v != parent) {
            dfsLCA(v, u);
        }
    }
}

// 获取 LCA
int getLCA(int u, int v) {
    if (depth[u] < depth[v]) {
        swap(u, v);
    }

    // 将 u 提升到 v 的深度
    for (int k = LOG - 1; k >= 0; k--) {
        if (depth[u] - (1 << k) >= depth[v]) {
            u = up[k][u];
        }
    }
}

```

```

if (u == v) return u;

for (int k = LOG - 1; k >= 0; k--) {
    if (up[k][u] != up[k][v]) {
        u = up[k][u];
        v = up[k][v];
    }
}

return up[0][u];
}

// DFS 合并子树并统计答案
void dfsMerge(int u, int parent) {
    for (int v : graph[u]) {
        if (v != parent) {
            dfsMerge(v, u);
            root[u] = merge(root[u], root[v], 1, maxType);
        }
    }
}

// 记录该节点的答案
ans[u] = tree[root[u]].maxType;
}

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

    int n, m;
    cin >> n >> m;

    // 构建树
    for (int i = 1; i < n; i++) {
        int u, v;
        cin >> u >> v;
        graph[u].push_back(v);
        graph[v].push_back(u);
    }

    // 预处理 LCA
    LOG = 0;

```

```

while ((1 << LOG) <= n) LOG++;
memset(depth, -1, sizeof(depth));
dfsLCA(1, 0);

// 初始化线段树相关变量
cnt = 0;
memset(root, 0, sizeof(root));
maxType = 0;

// 处理操作
while (m--) {
    int x, y, z;
    cin >> x >> y >> z;
    maxType = max(maxType, z);

    int lca = getLCA(x, y);
    int parentLCA = up[0][lca];

    // 树上差分
    update(root[x], 1, maxType, z, 1);
    update(root[y], 1, maxType, z, 1);
    update(root[lca], 1, maxType, z, -1);
    if (parentLCA != 0) {
        update(root[parentLCA], 1, maxType, z, -1);
    }
}

// DFS 合并子树线段树
dfsMerge(1, 0);

// 输出答案
for (int i = 1; i <= n; i++) {
    cout << ans[i] << "\n";
}

return 0;
}

/*
 * 复杂度分析:
 * - 时间复杂度: O(n log n + m log n)
 *   - LCA 预处理: O(n log n)
 *   - 每次更新操作: O(log n)

```

- \* - 线段树合并:  $O(n \log n)$ , 因为每个节点最多被合并一次
- \* - 空间复杂度:  $O(n \log n)$
- \* - 动态开点线段树的空间复杂度
- \*
- \* 算法详解:
- \* 本题使用线段树合并解决树上问题。核心思想是对树上的每个节点维护一个线段树，
- \* 线段树中存储该节点各种救济粮的数量。通过树上差分和线段树合并来高效处理操作。
- \*
- \* 算法步骤:
- \* 1. 树上差分: 对于每次在路径  $(u, v)$  上投放类型  $z$  的救济粮, 我们在  $u$  和  $v$  处+1,  
在  $\text{lca}(u, v)$  和  $\text{parent}[\text{lca}(u, v)]$  处-1
- \* 2. DFS 合并: 从叶子节点向根节点合并线段树, 统计每个节点的信息
- \* 3. 线段树合并: 合并两个节点的线段树, 同时维护最大值和对应的类型
- \*
- \* 关键优化:
- \* 1. 使用倍增法预处理 LCA, 时间复杂度  $O(n \log n)$
- \* 2. 动态开点线段树节省空间
- \* 3. 线段树合并避免重复计算
- \* 4. 树上差分将路径更新转化为点更新
- \*
- \* 优化点:
- \* 1. 可以使用更紧凑的动态开点策略
- \* 2. 对于救济粮类型较大的情况, 可以进行离散化处理
- \* 3. 标记永久化可以简化代码, 但需要注意正确性
- \*/

---

文件: Code17\_SegmentTreeMerge.java

---

```
package class131;

import java.util.*;

/**
 * 洛谷 P4556 (Vani 有约会) 雨天的尾巴 / 【模板】线段树合并
 * 题目链接: https://www.luogu.com.cn/problem/P4556
 * 题目描述:
 * 给定一棵有  $n$  个节点的树, 初始时每个节点的值为 0。有  $m$  次操作, 每次操作将路径  $x$  到  $y$  上的所有节点都
 * 加上一袋  $z$  种救济粮。
 * 最后每个节点需要输出它拥有的最多的救济粮种类, 如果有多个, 输出编号最小的。
 *
 * 解题思路:
```

```

* 1. 使用树上差分+线段树合并
* 2. 对每个节点建立动态开点线段树，维护各救济粮的数量
* 3. 使用 LCA 找到路径的最近公共祖先，进行差分操作
* 4. 在 DFS 回溯时合并子树的线段树，统计每个节点的答案
*
* 时间复杂度分析：
* - 预处理 LCA: O(n log n)
* - 线段树操作: O(m log n)
* - 线段树合并: O(n log n)
* 空间复杂度: O(n log n) 动态开点线段树
*/

```

public class Code17\_SegmentTreeMerge {

```

    // 线段树节点定义
    static class Node {
        int left, right; // 左右子节点索引
        int val;          // 节点值（救济粮数量）
        int maxVal;       // 最大值
        int maxType;      // 最大值对应的救济粮类型

        public Node() {
            this.left = 0;
            this.right = 0;
            this.val = 0;
            this.maxVal = 0;
            this.maxType = 0;
        }
    }
}

private static final int MAXN = 100010;
private static List<Integer>[] graph; // 树的邻接表
private static Node[] tree;           // 线段树节点数组
private static int[] root;            // 每个节点对应的线段树根节点
private static int[] ans;             // 每个节点的答案
private static int cnt;               // 动态开点计数器
private static int maxType;          // 最大救济粮类型

// LCA 相关
private static int[][] up;           // 祖先表
private static int[] depth;           // 深度数组
private static int LOG;               // log2(n) 向上取整

@SuppressWarnings("unchecked")

```

```
public static void main(String[] args) {
    Scanner scanner = new Scanner(System.in);
    int n = scanner.nextInt();
    int m = scanner.nextInt();

    // 初始化树
    graph = new ArrayList[n + 1];
    for (int i = 1; i <= n; i++) {
        graph[i] = new ArrayList<>();
    }

    for (int i = 1; i < n; i++) {
        int u = scanner.nextInt();
        int v = scanner.nextInt();
        graph[u].add(v);
        graph[v].add(u);
    }
}

// 预处理 LCA
LOG = 0;
while ((1 << LOG) <= n) LOG++;
up = new int[LOG][n + 1];
depth = new int[n + 1];
Arrays.fill(depth, -1);
dfsLCA(1, 0);

// 初始化线段树相关数组
int size = MAXN * 40; // 动态开点空间估计
tree = new Node[size];
for (int i = 0; i < size; i++) {
    tree[i] = new Node();
}
root = new int[n + 1];
ans = new int[n + 1];
cnt = 1;
maxType = 0;

// 处理操作
while (m-- > 0) {
    int x = scanner.nextInt();
    int y = scanner.nextInt();
    int z = scanner.nextInt();
    maxType = Math.max(maxType, z);
```

```

int lca = getLCA(x, y);
int parentLCA = up[0][lca];

// 树上差分
update(root[x], 1, maxType, z, 1);
update(root[y], 1, maxType, z, 1);
update(root[lca], 1, maxType, z, -1);
if (parentLCA != 0) {
    update(root[parentLCA], 1, maxType, z, -1);
}
}

// DFS 合并子树线段树
dfsMerge(1, 0);

// 输出答案
for (int i = 1; i <= n; i++) {
    System.out.println(ans[i]);
}

scanner.close();
}

// 线段树更新操作
private static void update(int root, int l, int r, int pos, int val) {
    if (l == r) {
        tree[root].val += val;
        tree[root].maxVal = tree[root].val;
        tree[root].maxType = 1;
        return;
    }

    int mid = (l + r) >> 1;
    if (pos <= mid) {
        if (tree[root].left == 0) tree[root].left = cnt++;
        update(tree[root].left, l, mid, pos, val);
    } else {
        if (tree[root].right == 0) tree[root].right = cnt++;
        update(tree[root].right, mid + 1, r, pos, val);
    }
}

pushUp(root);

```

```
}
```

```
// 线段树合并操作
```

```
private static int merge(int a, int b, int l, int r) {  
    if (a == 0) return b;  
    if (b == 0) return a;  
  
    if (l == r) {  
        tree[a].val += tree[b].val;  
        tree[a].maxVal = tree[a].val;  
        tree[a].maxType = 1;  
        return a;  
    }  
  
    int mid = (l + r) >> 1;  
    tree[a].left = merge(tree[a].left, tree[b].left, l, mid);  
    tree[a].right = merge(tree[a].right, tree[b].right, mid + 1, r);  
  
    pushUp(a);  
    return a;  
}
```

```
// 更新节点信息
```

```
private static void pushUp(int root) {  
    tree[root].maxVal = tree[root].val;  
    tree[root].maxType = 0;  
  
    // 检查左子树  
    if (tree[root].left != 0) {  
        if (tree[tree[root].left].maxVal > tree[root].maxVal ||  
            (tree[tree[root].left].maxVal == tree[root].maxVal &&  
tree[tree[root].left].maxType < tree[root].maxType)) {  
            tree[root].maxVal = tree[tree[root].left].maxVal;  
            tree[root].maxType = tree[tree[root].left].maxType;  
        }  
    }  
}
```

```
// 检查右子树
```

```
if (tree[root].right != 0) {  
    if (tree[tree[root].right].maxVal > tree[root].maxVal ||  
        (tree[tree[root].right].maxVal == tree[root].maxVal &&  
tree[tree[root].right].maxType < tree[root].maxType)) {  
        tree[root].maxVal = tree[tree[root].right].maxVal;
```

```

        tree[root].maxType = tree[tree[root].right].maxType;
    }
}
}

// LCA 预处理
private static void dfsLCA(int u, int parent) {
    depth[u] = depth[parent] + 1;
    up[0][u] = parent;

    for (int k = 1; k < LOG; k++) {
        up[k][u] = up[k-1][up[k-1][u]];
    }

    for (int v : graph[u]) {
        if (v != parent) {
            dfsLCA(v, u);
        }
    }
}

// 获取 LCA
private static int getLCA(int u, int v) {
    if (depth[u] < depth[v]) {
        int temp = u;
        u = v;
        v = temp;
    }

    // 将 u 提升到 v 的深度
    for (int k = LOG - 1; k >= 0; k--) {
        if (depth[u] - (1 << k) >= depth[v]) {
            u = up[k][u];
        }
    }

    if (u == v) return u;

    for (int k = LOG - 1; k >= 0; k--) {
        if (up[k][u] != up[k][v]) {
            u = up[k][u];
            v = up[k][v];
        }
    }
}

```

```

    }

    return up[0][u];
}

// DFS 合并子树并统计答案
private static void dfsMerge(int u, int parent) {
    for (int v : graph[u]) {
        if (v != parent) {
            dfsMerge(v, u);
            root[u] = merge(root[u], root[v], 1, maxType);
        }
    }
}

// 记录该节点的答案
ans[u] = tree[root[u]].maxType;
}
}

```

=====

文件: Code17\_SegmentTreeMerge.py

```

# class131/Code17_SegmentTreeMerge.py
# 洛谷 P4556 (Vani 有约会) 雨天的尾巴 / 【模板】线段树合并
# 题目链接: https://www.luogu.com.cn/problem/P4556
"""

```

题目描述:

给定一棵树，每次操作在两个节点之间的路径上投放某种类型的救济粮，询问每个节点拥有的最多的救济粮类型。

解题思路:

使用线段树合并解决树上问题。核心思想是对树上的每个节点维护一个线段树，线段树中存储该节点各种救济粮的数量。

通过树上差分和线段树合并来高效处理操作。

算法步骤:

1. 树上差分：对于每次在路径(u, v)上投放类型 z 的救济粮，我们在 u 和 v 处+1，在 lca(u, v) 和 parent[lca(u, v)] 处-1
2. DFS 合并：从叶子节点向根节点合并线段树，统计每个节点的信息
3. 线段树合并：合并两个节点的线段树，同时维护最大值和对应的类型

时间复杂度分析:

- 时间复杂度:  $O(n \log n + m \log n)$
- LCA 预处理:  $O(n \log n)$
- 每次更新操作:  $O(\log n)$
- 线段树合并:  $O(n \log n)$ , 因为每个节点最多被合并一次
- 空间复杂度:  $O(n \log n)$
- 动态开点线段树的空间复杂度

关键优化:

1. 使用 slots 优化内存
2. 设置递归深度限制
3. 动态开点线段树节省空间
4. 线段树合并避免重复计算

"""

```

import sys
from sys import stdin
from math import log2, ceil

MAXN = 100010

# 线段树节点类
class Node:
    __slots__ = ['left', 'right', 'val', 'max_val', 'max_type'] # 使用 slots 优化内存
    def __init__(self):
        self.left = 0 # 左子节点索引
        self.right = 0 # 右子节点索引
        self.val = 0 # 节点值（救济粮数量）
        self.max_val = 0 # 最大值
        self.max_type = 0 # 最大值对应的救济粮类型

# 全局变量管理器
class TreeManager:
    def __init__(self, size):
        """
        初始化线段树管理器
        :param size: 预分配的节点数量
        """
        self.tree = [Node() for _ in range(size)]
        self.cnt = 1 # 当前使用的节点编号

def main():
    sys.setrecursionlimit(1 << 25) # 设置递归深度限制

```

```

n, m = map(int, stdin.readline().split())

# 构建树的邻接表表示
graph = [[] for _ in range(n + 1)]
for _ in range(n - 1):
    u, v = map(int, stdin.readline().split())
    graph[u].append(v)
    graph[v].append(u)

# LCA 预处理，使用倍增法
LOG = ceil(log2(n)) if n > 0 else 0
up = [[0] * (n + 1) for _ in range(LOG)] # up[k][u] 表示 u 节点向上走 2^k 步到达的节点
depth = [-1] * (n + 1) # 节点深度

def dfs_lca(u, parent):
    """
    DFS 预处理 LCA 信息
    :param u: 当前节点
    :param parent: 父节点
    """
    depth[u] = depth[parent] + 1
    up[0][u] = parent

    # 倍增计算祖先节点
    for k in range(1, LOG):
        up[k][u] = up[k-1][up[k-1][u]]

    # 递归处理子节点
    for v in graph[u]:
        if v != parent:
            dfs_lca(v, u)

dfs_lca(1, 0) # 以节点 1 为根进行 DFS

# 获取两个节点的最近公共祖先
def get_lca(u, v):
    """
    计算节点 u 和 v 的最近公共祖先
    :param u: 节点 u
    :param v: 节点 v
    :return: 最近公共祖先
    """
    # 确保 u 的深度不小于 v

```

```

if depth[u] < depth[v]:
    u, v = v, u

# 将 u 提升到与 v 相同的深度
for k in range(LOG-1, -1, -1):
    if depth[u] - (1 << k) >= depth[v]:
        u = up[k][u]

# 如果 u 就是 v 的祖先，直接返回
if u == v:
    return u

# 同时向上提升 u 和 v，直到它们的父节点相同
for k in range(LOG-1, -1, -1):
    if up[k][u] != up[k][v]:
        u = up[k][u]
        v = up[k][v]

return up[0][u] # 返回最近公共祖先

# 初始化线段树管理器（预估空间）
tm = TreeManager(MAXN * 40)
root = [0] * (n + 1) # 每个节点对应的线段树根节点
ans = [0] * (n + 1) # 每个节点的答案（拥有最多的救济粮类型）
max_type = 0 # 最大的救济粮类型编号

# 线段树更新操作
def update(node, l, r, pos, val):
    """
    更新线段树节点
    :param node: 当前线段树节点索引
    :param l: 区间左边界
    :param r: 区间右边界
    :param pos: 要更新的位置
    :param val: 更新的值
    :return: 更新后的节点索引
    """

# 如果节点不存在，创建新节点
if not node:
    node = tm.cnt
    tm.cnt += 1

# 如果是叶子节点，直接更新值

```

```

if l == r:
    tm.tree[node].val += val
    tm.tree[node].max_val = tm.tree[node].val
    tm.tree[node].max_type = 1
    return node

# 计算中点，决定更新左子树还是右子树
mid = (l + r) >> 1
if pos <= mid:
    tm.tree[node].left = update(tm.tree[node].left, l, mid, pos, val)
else:
    tm.tree[node].right = update(tm.tree[node].right, mid + 1, r, pos, val)

# 更新当前节点的 max_val 和 max_type
tm.tree[node].max_val = 0
tm.tree[node].max_type = 0

# 比较左子树的最大值
left = tm.tree[node].left
if left:
    left_tree = tm.tree[left]
    if left_tree.max_val > tm.tree[node].max_val or \
       (left_tree.max_val == tm.tree[node].max_val and left_tree.max_type <
tm.tree[node].max_type):
        tm.tree[node].max_val = left_tree.max_val
        tm.tree[node].max_type = left_tree.max_type

# 比较右子树的最大值
right = tm.tree[node].right
if right:
    right_tree = tm.tree[right]
    if right_tree.max_val > tm.tree[node].max_val or \
       (right_tree.max_val == tm.tree[node].max_val and right_tree.max_type <
tm.tree[node].max_type):
        tm.tree[node].max_val = right_tree.max_val
        tm.tree[node].max_type = right_tree.max_type

return node

# 线段树合并操作
def merge(a, b, l, r):
    """
    合并两个线段树节点

```

```

:param a: 第一个节点索引
:param b: 第二个节点索引
:param l: 区间左边界
:param r: 区间右边界
:return: 合并后的节点索引
"""

# 如果其中一个节点为空, 返回另一个节点
if not a:
    return b
if not b:
    return a

# 如果是叶子节点, 直接合并值
if l == r:
    tm.tree[a].val += tm.tree[b].val
    tm.tree[a].max_val = tm.tree[a].val
    tm.tree[a].max_type = 1
    return a

# 递归合并左右子树
mid = (l + r) >> 1
tm.tree[a].left = merge(tm.tree[a].left, tm.tree[b].left, l, mid)
tm.tree[a].right = merge(tm.tree[a].right, tm.tree[b].right, mid + 1, r)

# 更新当前节点的 max_val 和 max_type
tm.tree[a].max_val = 0
tm.tree[a].max_type = 0

# 比较左子树的最大值
left = tm.tree[a].left
if left:
    left_tree = tm.tree[left]
    if left_tree.max_val > tm.tree[a].max_val or \
       (left_tree.max_val == tm.tree[a].max_val and left_tree.max_type <
        tm.tree[a].max_type):
        tm.tree[a].max_val = left_tree.max_val
        tm.tree[a].max_type = left_tree.max_type

# 比较右子树的最大值
right = tm.tree[a].right
if right:
    right_tree = tm.tree[right]
    if right_tree.max_val > tm.tree[a].max_val or \

```

```

        (right_tree.max_val == tm.tree[a].max_val and right_tree.max_type <
tm.tree[a].max_type):
            tm.tree[a].max_val = right_tree.max_val
            tm.tree[a].max_type = right_tree.max_type

    return a

# 处理操作
for _ in range(m):
    x, y, z = map(int, stdin.readline().split())
    max_type = max(max_type, z) # 更新最大类型编号

    lca_node = get_lca(x, y) # 获取 x 和 y 的最近公共祖先
    parent_lca = up[0][lca_node] # 获取最近公共祖先的父节点

    # 树上差分：在路径两端加 1，在 LCA 和其父节点处减 1
    root[x] = update(root[x], 1, max_type, z, 1)
    root[y] = update(root[y], 1, max_type, z, 1)
    root[lca_node] = update(root[lca_node], 1, max_type, z, -1)
    if parent_lca != 0:
        root[parent_lca] = update(root[parent_lca], 1, max_type, z, -1)

# DFS 合并子树并统计答案
visited = [False] * (n + 1)
def dfs_merge(u, parent):
    """
    DFS 遍历树并合并线段树
    :param u: 当前节点
    :param parent: 父节点
    """
    # 递归处理所有子节点
    for v in graph[u]:
        if v != parent:
            dfs_merge(v, u)
            # 将子节点的线段树合并到当前节点
            root[u] = merge(root[u], root[v], 1, max_type)

    # 记录该节点的答案（拥有最多的救济粮类型）
    ans[u] = tm.tree[root[u]].max_type

dfs_merge(1, 0) # 从根节点开始 DFS 合并

# 输出答案

```

```

for i in range(1, n + 1):
    print(ans[i])

if __name__ == "__main__":
    main()

...

```

复杂度分析:

- 时间复杂度:  $O(n \log n + m \log n)$ 
  - LCA 预处理:  $O(n \log n)$
  - 每次更新操作:  $O(\log n)$
  - 线段树合并:  $O(n \log n)$ , 因为每个节点最多被合并一次
- 空间复杂度:  $O(n \log n)$ 
  - 动态开点线段树的空间复杂度

Python 实现注意事项:

1. Python 的递归深度有限, 需要设置 `sys.setrecursionlimit`
2. 为了优化内存使用, 使用了 `__slots__`
3. 由于 Python 的特性, 对于大规模数据可能需要进一步优化
4. 动态开点策略在 Python 中使用索引而非指针

...

---

文件: Code18\_FenwickTreeWithSegmentTree.cpp

---

```

// class131/Code18_FenwickTreeWithSegmentTree.cpp
// 洛谷 P3380 【模板】树套树
// 题目链接: https://www.luogu.com.cn/problem/P3380

```

```

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

const int MAXN = 50010;
const int MAX_VAL = 1000000010;
const int MAX_TREE_SIZE = MAXN * 40;

```

```

// 线段树节点定义
struct Node {
    int left, right; // 左右子节点索引
    int count; // 该区间的元素个数
}

```

```

} tree[MAX_TREE_SIZE];

int root[MAXN]; // 树状数组每个节点对应的线段树根节点
int a[MAXN]; // 原始数组
vector<int> sorted; // 离散化后的数组
int cnt; // 动态开点计数器
int n, m; // 数组大小和操作数量

// 初始化线段树节点
void initNode(int node) {
    tree[node].left = tree[node].right = 0;
    tree[node].count = 0;
}

// 线段树更新
void updateTree(int &root, int l, int r, int pos, int val) {
    if (!root) {
        root = ++cnt;
        initNode(root);
    }

    tree[root].count += val;
    if (l == r) {
        return;
    }

    int mid = (l + r) >> 1;
    if (pos <= mid) {
        updateTree(tree[root].left, l, mid, pos, val);
    } else {
        updateTree(tree[root].right, mid + 1, r, pos, val);
    }
}

// 线段树查询区间[l, r]中在[ql, qr]范围内的元素个数
int queryTree(int root, int l, int r, int ql, int qr) {
    if (!root || r < ql || l > qr) {
        return 0;
    }

    if (ql <= l && r <= qr) {
        return tree[root].count;
    }
}

```

```

int mid = (l + r) >> 1;
return queryTree(tree[root].left, l, mid, ql, qr) +
       queryTree(tree[root].right, mid + 1, r, ql, qr);
}

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

// 树状数组更新
void update(int pos, int val, int delta) {
    while (pos <= n) {
        updateTree(root[pos], 1, sorted.size() - 1, val, delta);
        pos += lowbit(pos);
    }
}

// 树状数组查询前缀和
int query(int pos, int val) {
    int sum = 0;
    while (pos > 0) {
        sum += queryTree(root[pos], 1, sorted.size() - 1, 1, val);
        pos -= lowbit(pos);
    }
    return sum;
}

// 离散化: 获取数值对应的索引
int getIndex(int val) {
    return lower_bound(sorted.begin(), sorted.end(), val) - sorted.begin();
}

// 查询区间第 k 小
int kthSmallest(int l, int r, int k) {
    int left = 1, right = sorted.size() - 1;
    while (left < right) {
        int mid = (left + right) >> 1;
        int count = query(r, mid) - query(l - 1, mid);
        if (count >= k) {
            right = mid;
        } else {

```

```

        left = mid + 1;
    }
}

return sorted[left];
}

// 查询前驱（最大的小于 k 的数）
int getPredecessor(int l, int r, int k) {
    int left = 1, right = sorted.size() - 1;
    int ans = -MAX_VAL;

    while (left <= right) {
        int mid = (left + right) >> 1;
        if (sorted[mid] >= k) {
            right = mid - 1;
        } else {
            int count = query(r, mid) - query(l - 1, mid);
            if (count > 0) {
                ans = sorted[mid];
                left = mid + 1;
            } else {
                right = mid - 1;
            }
        }
    }

    return ans;
}

// 查询后继（最小的大于 k 的数）
int getSuccessor(int l, int r, int k) {
    int left = 1, right = sorted.size() - 1;
    int ans = MAX_VAL;

    while (left <= right) {
        int mid = (left + right) >> 1;
        if (sorted[mid] <= k) {
            left = mid + 1;
        } else {
            // 检查是否有大于 k 的元素
            int count = query(r, sorted.size() - 1) - query(r, mid - 1) -
                        (query(l - 1, sorted.size() - 1) - query(l - 1, mid - 1));
            if (count > 0) {

```

```
        ans = sorted[mid];
        right = mid - 1;
    } else {
        left = mid + 1;
    }
}

return ans;
}

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

    cin >> n >> m;

    // 读取原始数据并复制到 sorted 数组用于离散化
    sorted.resize(n + 1);
    for (int i = 1; i <= n; i++) {
        cin >> a[i];
        sorted[i] = a[i];
    }

    // 离散化处理：排序并去重
    sort(sorted.begin() + 1, sorted.end());
    sorted.erase(unique(sorted.begin() + 1, sorted.end()), sorted.end());

    // 初始化
    cnt = 0;
    memset(root, 0, sizeof(root));

    // 构建初始树状数组
    for (int i = 1; i <= n; i++) {
        int idx = getIndex(a[i]);
        update(i, idx, 1);
    }

    // 处理操作
    while (m--) {
        int op;
        cin >> op;
```

```

if (op == 1) { // 查询排名
    int l, r, k;
    cin >> l >> r >> k;
    int idx = lower_bound(sorted.begin(), sorted.end(), k) - sorted.begin();
    int rank = (query(r, idx - 1) - query(l - 1, idx - 1)) + 1;
    cout << rank << "\n";
} else if (op == 2) { // 查询第 k 小
    int l, r, k;
    cin >> l >> r >> k;
    int ans = kthSmallest(l, r, k);
    cout << ans << "\n";
} else if (op == 3) { // 单点修改
    int pos, k;
    cin >> pos >> k;
    // 先删除旧值
    int oldIdx = getIndex(a[pos]);
    update(pos, oldIdx, -1);
    // 再插入新值
    a[pos] = k;
    int newIdx = getIndex(k);
    update(pos, newIdx, 1);
} else if (op == 4) { // 查询前驱
    int l, r, k;
    cin >> l >> r >> k;
    int pre = getPredecessor(l, r, k);
    cout << pre << "\n";
} else if (op == 5) { // 查询后继
    int l, r, k;
    cin >> l >> r >> k;
    int succ = getSuccessor(l, r, k);
    cout << succ << "\n";
}
}

return 0;
}

/*
复杂度分析:
- 时间复杂度:
  - 单点修改:  $O(\log^2 n)$  - 树状数组  $O(\log n)$  次操作, 每次操作对应线段树  $O(\log n)$  时间
  - 区间查询:  $O(\log^2 n)$  - 树状数组  $O(\log n)$  次操作, 每次操作对应线段树  $O(\log n)$  时间
  - 第 k 小查询:  $O(\log^2 n)$  - 二分查找  $O(\log n)$  次, 每次查询  $O(\log^2 n)$  时间

```

- /\*
- 复杂度分析:
- 时间复杂度:
    - 单点修改:  $O(\log^2 n)$  - 树状数组  $O(\log n)$  次操作, 每次操作对应线段树  $O(\log n)$  时间
    - 区间查询:  $O(\log^2 n)$  - 树状数组  $O(\log n)$  次操作, 每次操作对应线段树  $O(\log n)$  时间
    - 第 k 小查询:  $O(\log^2 n)$  - 二分查找  $O(\log n)$  次, 每次查询  $O(\log^2 n)$  时间

- 前驱/后继查询:  $O(\log^2 n)$
- 空间复杂度:  $O(n \log n)$  - 树状数组的每个节点对应一棵动态开点线段树

算法详解:

使用树套树（树状数组套线段树）解决区间问题。核心思想是用树状数组维护位置维度，每个树状数组节点对应一棵线段树，线段树维护值域维度。通过这种二维结构可以高效处理区间查询和更新操作。

算法步骤:

1. 离散化: 将输入数据离散化到连续整数范围, 减少空间消耗
2. 构建树状数组: 每个节点维护一棵动态开点线段树
3. 实现各种查询操作: 利用树状数组的前缀和性质和线段树的区间查询能力

关键优化:

1. 使用离散化减少值域范围
2. 动态开点线段树节省空间
3. 树状数组的前缀和性质简化区间查询

优化点:

1. 离散化是必须的, 否则权值范围太大无法处理
  2. 动态开点避免了静态线段树的空间浪费
  3. 可以使用非递归实现来优化常数
  4. 对于大规模数据, 可以考虑使用内存池优化动态分配
- \*/
- 

文件: Code18\_FenwickTreeWithSegmentTree.java

---

```
package class131;

import java.util.*;

/**
 * 洛谷 P3380 【模板】树套树
 * 题目链接: https://www.luogu.com.cn/problem/P3380
 * 题目描述:
 * 您需要维护一个序列, 支持以下操作:
 * 1. 查询区间[1, r]中数值 k 的排名 (比 k 小的数的个数+1)
 * 2. 查询区间[1, r]中第 k 小的数
 * 3. 单点修改: 将 pos 位置的值修改为 k
 * 4. 查询区间[1, r]中数值 k 的前驱 (最大的比 k 小的数)
 * 5. 查询区间[1, r]中数值 k 的后继 (最小的比 k 大的数)
```

```
*  
* 解题思路：  
* 使用树状数组套权值线段树 (Fenwick Tree + 动态开点线段树)  
* 1. 树状数组的每个节点对应一个权值线段树  
* 2. 树状数组用于处理区间查询 (前缀和思想)  
* 3. 权值线段树用于处理数值范围的统计  
* 4. 离散化处理原始数据以压缩权值范围  
*  
* 时间复杂度分析：  
* - 单点修改:  $O(\log^2 n)$   
* - 区间查询:  $O(\log^2 n)$   
* 空间复杂度:  $O(n \log n)$   
*/
```

```
public class Code18_FenwickTreeWithSegmentTree {  
  
    static class Node {  
        int left, right; // 左右子节点索引  
        int count; // 该区间的元素个数  
  
        public Node() {  
            this.left = 0;  
            this.right = 0;  
            this.count = 0;  
        }  
    }  
  
    private static final int MAXN = 50010;  
    private static final int MAX_VAL = 1000000010;  
    private static Node[] tree; // 线段树节点数组  
    private static int[] root; // 树状数组每个节点对应的线段树根节点  
    private static int[] a; // 原始数组  
    private static int[] sorted; // 离散化后的数组  
    private static int cnt; // 动态开点计数器  
    private static int n, m; // 数组大小和操作数量  
  
    public static void main(String[] args) {  
        Scanner scanner = new Scanner(System.in);  
        n = scanner.nextInt();  
        m = scanner.nextInt();  
  
        a = new int[n + 1];  
        sorted = new int[n + 1];  
        for (int i = 1; i <= n; i++) {  
    }
```

```

a[i] = scanner.nextInt();
sorted[i] = a[i];
}

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

// 初始化树状数组和线段树
int size = MAXN * 40; // 预估空间大小
tree = new Node[size];
for (int i = 0; i < size; i++) {
    tree[i] = new Node();
}
root = new int[MAXN];
cnt = 1;

// 构建初始树状数组
for (int i = 1; i <= n; i++) {
    int idx = getIndex(a[i]);
    update(i, idx, 1);
}

// 处理操作
while (m-- > 0) {
    int op = scanner.nextInt();
    if (op == 1) { // 查询排名
        int l = scanner.nextInt();
        int r = scanner.nextInt();
        int k = scanner.nextInt();
        int rank = query(r, getIndex(k) - 1) - query(l - 1, getIndex(k) - 1) + 1;
        System.out.println(rank);
    } else if (op == 2) { // 查询第 k 小
        int l = scanner.nextInt();
        int r = scanner.nextInt();
        int k = scanner.nextInt();
        int ans = kthSmallest(l, r, k);
        System.out.println(ans);
    }
}

```

```

} else if (op == 3) { // 单点修改
    int pos = scanner.nextInt();
    int k = scanner.nextInt();
    int oldIdx = getIndex(a[pos]);
    update(pos, oldIdx, -1);
    a[pos] = k;
    int newIdx = getIndex(k);
    update(pos, newIdx, 1);
} else if (op == 4) { // 查询前驱
    int l = scanner.nextInt();
    int r = scanner.nextInt();
    int k = scanner.nextInt();
    int pre = getPredecessor(l, r, k);
    System.out.println(pre);
} else if (op == 5) { // 查询后继
    int l = scanner.nextInt();
    int r = scanner.nextInt();
    int k = scanner.nextInt();
    int succ = getSuccessor(l, r, k);
    System.out.println(succ);
}
}

scanner.close();
}

// 离散化: 获取数值对应的索引
private static int getIndex(int val) {
    int left = 1, right = sorted.length - 1;
    while (left <= right) {
        int mid = (left + right) >> 1;
        if (sorted[mid] == val) {
            return mid;
        } else if (sorted[mid] < val) {
            left = mid + 1;
        } else {
            right = mid - 1;
        }
    }
    return left;
}

// 线段树更新

```

```

private static void updateTree(int root, int l, int r, int pos, int val) {
    tree[root].count += val;
    if (l == r) {
        return;
    }
    int mid = (l + r) >> 1;
    if (pos <= mid) {
        if (tree[root].left == 0) {
            tree[root].left = cnt++;
        }
        updateTree(tree[root].left, l, mid, pos, val);
    } else {
        if (tree[root].right == 0) {
            tree[root].right = cnt++;
        }
        updateTree(tree[root].right, mid + 1, r, pos, val);
    }
}

```

#### // 线段树查询

```

private static int queryTree(int root, int l, int r, int ql, int qr) {
    if (root == 0 || r < ql || l > qr) {
        return 0;
    }
    if (ql <= l && r <= qr) {
        return tree[root].count;
    }
    int mid = (l + r) >> 1;
    return queryTree(tree[root].left, l, mid, ql, qr) +
        queryTree(tree[root].right, mid + 1, r, ql, qr);
}

```

#### // 树状数组更新

```

private static void update(int pos, int val, int delta) {
    while (pos <= n) {
        if (root[pos] == 0) {
            root[pos] = cnt++;
        }
        updateTree(root[pos], 1, sorted.length - 1, val, delta);
        pos += lowbit(pos);
    }
}

```

```

// 树状数组查询前缀和
private static int query(int pos, int val) {
    int sum = 0;
    while (pos > 0) {
        sum += queryTree(root[pos], 1, sorted.length - 1, 1, val);
        pos -= lowbit(pos);
    }
    return sum;
}

// 计算区间[1,r]中<=k 的元素个数
private static int getCount(int l, int r, int k) {
    return query(r, getIndex(k)) - query(l - 1, getIndex(k));
}

// 查询区间第 k 小
private static int kthSmallest(int l, int r, int k) {
    int left = 1, right = sorted.length - 1;
    while (left < right) {
        int mid = (left + right) >> 1;
        int count = query(r, mid) - query(l - 1, mid);
        if (count >= k) {
            right = mid;
        } else {
            left = mid + 1;
        }
    }
    return sorted[left];
}

// 查询前驱
private static int getPredecessor(int l, int r, int k) {
    int left = 1, right = sorted.length - 1;
    int ans = -MAX_VAL;
    while (left <= right) {
        int mid = (left + right) >> 1;
        if (sorted[mid] >= k) {
            right = mid - 1;
        } else {
            int count = query(r, mid) - query(l - 1, mid);
            if (count > 0) {
                ans = sorted[mid];
                left = mid + 1;
            }
        }
    }
    return ans;
}

```

```

        } else {
            right = mid - 1;
        }
    }
    return ans;
}

// 查询后继
private static int getSuccessor(int l, int r, int k) {
    int left = 1, right = sorted.length - 1;
    int ans = MAX_VAL;
    while (left <= right) {
        int mid = (left + right) >> 1;
        if (sorted[mid] <= k) {
            left = mid + 1;
        } else {
            int count = query(r, sorted.length - 1) - query(l - 1, sorted.length - 1) -
                (query(r, mid - 1) - query(l - 1, mid - 1));
            if (count > 0) {
                ans = sorted[mid];
                right = mid - 1;
            } else {
                left = mid + 1;
            }
        }
    }
    return ans;
}

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

```

=====

文件: Code18\_FenwickTreeWithSegmentTree.py

=====

```

# class131/Code18_FenwickTreeWithSegmentTree.py
# 洛谷 P3380 【模板】树套树
# 题目链接: https://www.luogu.com.cn/problem/P3380

```

"""

## 题目描述：

实现一个支持以下操作的数据结构：

1. 查询区间 $[1, r]$ 中值 $\leq k$ 的元素个数
2. 查询区间 $[1, r]$ 中第 $k$ 小的元素
3. 单点修改数组元素
4. 查询区间 $[1, r]$ 中值 $< k$ 的最大元素（前驱）
5. 查询区间 $[1, r]$ 中值 $> k$ 的最小元素（后继）

## 解题思路：

使用树套树（树状数组套线段树）解决区间问题。核心思想是用树状数组维护位置维度，每个树状数组节点对应一棵线段树，

线段树维护值域维度。通过这种二维结构可以高效处理区间查询和更新操作。

## 算法步骤：

1. 离散化：将输入数据离散化到连续整数范围，减少空间消耗
2. 构建树状数组：每个节点维护一棵动态开点线段树
3. 实现各种查询操作：利用树状数组的前缀和性质和线段树的区间查询能力

## 时间复杂度分析：

- 时间复杂度：
  - 单点修改： $O(\log^2 n)$  - 树状数组  $O(\log n)$  次操作，每次操作对应线段树  $O(\log n)$  时间
  - 区间查询： $O(\log^2 n)$  - 树状数组  $O(\log n)$  次操作，每次操作对应线段树  $O(\log n)$  时间
  - 第 $k$ 小查询： $O(\log^3 n)$  - 二分查找  $O(\log n)$  次，每次查询  $O(\log^2 n)$  时间
  - 前驱/后继查询： $O(\log^3 n)$
- 空间复杂度： $O(n \log n)$  - 树状数组的每个节点对应一棵动态开点线段树

## 关键优化：

1. 使用 slots 优化内存
2. 设置递归深度限制
3. 动态开点线段树节省空间
4. 使用 bisect 模块进行高效的二分查找

"""

```
import sys
import bisect
from sys import stdin
```

```
MAXN = 50010
MAX_VAL = 1000000010
```

```
# 线段树节点类
class Node:
```

```
__slots__ = ['left', 'right', 'count'] # 使用 slots 优化内存
def __init__(self):
    self.left = 0 # 左子节点索引
    self.right = 0 # 右子节点索引
    self.count = 0 # 该区间的元素个数

class TreeManager:
    def __init__(self, size):
        """
        初始化线段树管理器
        :param size: 预分配的节点数量
        """
        self.tree = [Node() for _ in range(size)]
        self.cnt = 1 # 当前使用的节点编号

    def get_node(self, idx):
        """
        获取指定索引的节点，如果索引超出范围则动态扩展
        :param idx: 节点索引
        :return: 对应的节点
        """
        if idx >= len(self.tree):
            # 动态扩展数组（实际中应该预先分配足够空间）
            new_size = max(len(self.tree) * 2, idx + 1)
            self.tree.extend([Node() for _ in range(new_size - len(self.tree))])
        return self.tree[idx]

    def new_node(self):
        """
        创建新节点
        :return: 新节点的索引
        """
        node_idx = self.cnt
        self.cnt += 1
        return node_idx

def main():
    sys.setrecursionlimit(1 << 25) # 设置递归深度限制

    n, m = map(int, stdin.readline().split())

    # 读取输入数组
    a = [0] * (n + 1)
```

```

sorted_list = [0] * (n + 1)
for i in range(1, n + 1):
    a[i] = int(stdin.readline())
    sorted_list[i] = a[i]

# 离散化处理：将数值映射到连续的整数范围
sorted_list = sorted_list[1:n+1] # 去掉前导 0
sorted_list = sorted(list(set(sorted_list))) # 去重并排序
sorted_list = [0] + sorted_list # 索引从 1 开始，添加前导 0

# 初始化树状数组和线段树管理器
tm = TreeManager(MAXN * 40) # 预估空间
root = [0] * (MAXN + 1) # 树状数组，每个元素是线段树根节点

# 计算 lowbit 函数，用于树状数组操作
def lowbit(x):
    """
    计算 x 的 lowbit 值（x 的二进制表示中最低位的 1 所代表的数值）
    :param x: 输入值
    :return: lowbit 值
    """
    return x & (-x)

# 线段树更新（递归实现）
def update_tree(node_idx, l, r, pos, val):
    """
    更新线段树节点
    :param node_idx: 当前线段树节点索引
    :param l: 区间左边界
    :param r: 区间右边界
    :param pos: 要更新的位置
    :param val: 更新的值
    """
    node = tm.get_node(node_idx)
    node.count += val # 更新计数

    # 如果是叶子节点，直接返回
    if l == r:
        return

    # 计算中点，决定更新左子树还是右子树
    mid = (l + r) >> 1
    if pos <= mid:

```

```

    if not node.left:
        node.left = tm.new_node()
        update_tree(node.left, l, mid, pos, val)
    else:
        if not node.right:
            node.right = tm.new_node()
            update_tree(node.right, mid + 1, r, pos, val)

# 线段树查询
def query_tree(node_idx, l, r, ql, qr):
    """
    查询线段树区间和
    :param node_idx: 当前线段树节点索引
    :param l: 区间左边界
    :param r: 区间右边界
    :param ql: 查询区间左边界
    :param qr: 查询区间右边界
    :return: 区间和
    """

    # 如果节点不存在或查询区间无交集，返回 0
    if not node_idx or r < ql or l > qr:
        return 0

    node = tm.get_node(node_idx)
    # 如果查询区间完全包含当前节点区间，直接返回计数
    if ql <= l and r <= qr:
        return node.count

    # 否则递归查询左右子树
    mid = (l + r) // 2
    return query_tree(node.left, l, mid, ql, qr) + \
           query_tree(node.right, mid + 1, r, ql, qr)

# 树状数组更新
def update(pos, val, delta):
    """
    更新树状数组
    :param pos: 位置
    :param val: 值（离散化后的索引）
    :param delta: 变化量
    """

    while pos <= n:
        if not root[pos]:

```

```

        root[pos] = tm.new_node()
        update_tree(root[pos], 1, len(sorted_list) - 1, val, delta)
        pos += lowbit(pos)

# 树状数组查询前缀和
def query_prefix(pos, val):
    """
    查询树状数组前缀和
    :param pos: 位置
    :param val: 值（离散化后的索引）
    :return: 前缀和
    """
    sum_val = 0
    while pos > 0:
        sum_val += query_tree(root[pos], 1, len(sorted_list) - 1, 1, val)
        pos -= lowbit(pos)
    return sum_val

# 离散化：获取数值对应的索引
def get_index(val):
    """
    获取数值在离散化数组中的索引
    :param val: 数值
    :return: 索引
    """
    return bisect.bisect_left(sorted_list, val)

# 查询区间[l, r]中<=k 的元素个数
def get_count(l, r, k):
    """
    查询区间[l, r]中<=k 的元素个数
    :param l: 区间左边界
    :param r: 区间右边界
    :param k: 上界值
    :return: 元素个数
    """
    idx = get_index(k)
    return query_prefix(r, idx) - query_prefix(l - 1, idx)

# 查询区间第 k 小
def kth_smallest(l, r, k):
    """
    查询区间[l, r]中第 k 小的元素
    """

```

```

:param l: 区间左边界
:param r: 区间右边界
:param k: 第 k 小
:return: 第 k 小的元素
"""
left, right = 1, len(sorted_list) - 1
while left < right:
    mid = (left + right) >> 1
    count = query_prefix(r, mid) - query_prefix(l - 1, mid)
    if count >= k:
        right = mid
    else:
        left = mid + 1
return sorted_list[left]

# 查询前驱（最大的小于 k 的数）
def get_predecessor(l, r, k):
"""
查询区间[l, r]中值<k 的最大元素（前驱）
:param l: 区间左边界
:param r: 区间右边界
:param k: 上界值
:return: 前驱元素
"""
left, right = 1, len(sorted_list) - 1
ans = -MAX_VAL

while left <= right:
    mid = (left + right) >> 1
    if sorted_list[mid] >= k:
        right = mid - 1
    else:
        count = query_prefix(r, mid) - query_prefix(l - 1, mid)
        if count > 0:
            ans = sorted_list[mid]
            left = mid + 1
        else:
            right = mid - 1
return ans

# 查询后继（最小的大于 k 的数）
def get_successor(l, r, k):

```

```

"""
查询区间[1, r]中值>k 的最小元素（后继）
:param l: 区间左边界
:param r: 区间右边界
:param k: 下界值
:return: 后继元素
"""

left, right = 1, len(sorted_list) - 1
ans = MAX_VAL

while left <= right:
    mid = (left + right) >> 1
    if sorted_list[mid] <= k:
        left = mid + 1
    else:
        # 检查是否有大于 k 的元素
        count_r = query_prefix(r, len(sorted_list) - 1) - query_prefix(r, mid - 1)
        count_l_1 = query_prefix(l - 1, len(sorted_list) - 1) - query_prefix(l - 1, mid - 1)
        count = count_r - count_l_1

        if count > 0:
            ans = sorted_list[mid]
            right = mid - 1
        else:
            left = mid + 1

return ans

# 构建初始树状数组
for i in range(1, n + 1):
    idx = get_index(a[i])
    update(i, idx, 1)

# 处理操作
for _ in range(m):
    parts = stdin.readline().split()
    op = int(parts[0])

    if op == 1: # 查询排名
        l, r, k = int(parts[1]), int(parts[2]), int(parts[3])
        idx = bisect.bisect_left(sorted_list, k)
        rank = (query_prefix(r, idx - 1) - query_prefix(l - 1, idx - 1)) + 1

```

```

print(rank)

elif op == 2: # 查询第 k 小
    l, r, k = int(parts[1]), int(parts[2]), int(parts[3])
    ans = kth_smallest(l, r, k)
    print(ans)

elif op == 3: # 单点修改
    pos, k = int(parts[1]), int(parts[2])
    # 先删除旧值
    old_idx = get_index(a[pos])
    update(pos, old_idx, -1)
    # 再插入新值
    a[pos] = k
    new_idx = get_index(k)
    update(pos, new_idx, 1)

elif op == 4: # 查询前驱
    l, r, k = int(parts[1]), int(parts[2]), int(parts[3])
    pre = get_predecessor(l, r, k)
    print(pre)

elif op == 5: # 查询后继
    l, r, k = int(parts[1]), int(parts[2]), int(parts[3])
    succ = get_successor(l, r, k)
    print(succ)

if __name__ == "__main__":
    main()

,,,
```

复杂度分析:

- 时间复杂度:
  - 单点修改:  $O(\log^2 n)$  - 树状数组  $O(\log n)$  次操作, 每次操作对应线段树  $O(\log n)$  时间
  - 区间查询:  $O(\log^2 n)$  - 树状数组  $O(\log n)$  次操作, 每次操作对应线段树  $O(\log n)$  时间
  - 第 k 小查询:  $O(\log^2 n)$  - 二分查找  $O(\log n)$  次, 每次查询  $O(\log^2 n)$  时间
  - 前驱/后继查询:  $O(\log^2 n)$
- 空间复杂度:  $O(n \log n)$  - 树状数组的每个节点对应一棵动态开点线段树

Python 实现注意事项:

1. 由于 Python 的递归深度限制, 对于大规模数据可能需要增加递归限制
2. 使用 `bisect` 模块进行二分查找, 效率比手动实现更高
3. 使用 `__slots__` 优化类的内存占用
4. 动态扩展线段树节点数组以避免内存浪费
5. 实际应用中应预先分配足够大的空间以提高效率

,,

文件: Code19\_2DSegmentTree.cpp

```
=====

// class131/Code19_2DSegmentTree.cpp
// LeetCode 304. Range Sum Query 2D - Immutable 的二维线段树实现
// 题目链接: https://leetcode.com/problems/range-sum-query-2d-immutable/

#include <iostream>
#include <vector>
#include <stdexcept>
using namespace std;

/***
 * 二维线段树节点定义
 */
struct Node {
    int sum;          // 当前区域的和
    int row1, row2;  // 行范围
    int col1, col2;  // 列范围
    Node *topLeft, *topRight, *bottomLeft, *bottomRight; // 四个子区域

    Node(int r1, int r2, int c1, int c2) :
        sum(0), row1(r1), row2(r2), col1(c1), col2(c2),
        topLeft(nullptr), topRight(nullptr), bottomLeft(nullptr), bottomRight(nullptr) {}

    // 判断是否是叶子节点
    bool isLeaf() const {
        return row1 == row2 && col1 == col2;
    }

    // 析构函数, 释放子节点内存
    ~Node() {
        delete topLeft;
        delete topRight;
        delete bottomLeft;
        delete bottomRight;
    }
};

class Code19_2DSegmentTree {
private:
    Node* root;      // 根节点
```

```

vector<vector<int>> matrix; // 原始矩阵
int rows, cols; // 矩阵尺寸

/***
 * 构建二维线段树
 * @param row1 起始行
 * @param row2 结束行
 * @param col1 起始列
 * @param col2 结束列
 * @return 构建好的线段树节点
 */
Node* buildTree(int row1, int row2, int col1, int col2) {
    Node* node = new Node(row1, row2, col1, col2);

    // 叶子节点
    if (row1 == row2 && col1 == col2) {
        node->sum = matrix[row1][col1];
        return node;
    }

    int midRow = row1 + (row2 - row1) / 2;
    int midCol = col1 + (col2 - col1) / 2;

    // 递归构建四个子区域
    if (row1 == row2) {
        // 只有一行
        node->topLeft = buildTree(row1, row2, col1, midCol);
        node->topRight = buildTree(row1, row2, midCol + 1, col2);
        node->sum = node->topLeft->sum + node->topRight->sum;
    } else if (col1 == col2) {
        // 只有一列
        node->topLeft = buildTree(row1, midRow, col1, col2);
        node->bottomLeft = buildTree(midRow + 1, row2, col1, col2);
        node->sum = node->topLeft->sum + node->bottomLeft->sum;
    } else {
        // 一般情况，分为四个子区域
        node->topLeft = buildTree(row1, midRow, col1, midCol);
        node->topRight = buildTree(row1, midRow, midCol + 1, col2);
        node->bottomLeft = buildTree(midRow + 1, row2, col1, midCol);
        node->bottomRight = buildTree(midRow + 1, row2, midCol + 1, col2);
        node->sum = node->topLeft->sum + node->topRight->sum +
                    node->bottomLeft->sum + node->bottomRight->sum;
    }
}

```

```

    return node;
}

/***
 * 递归更新线段树
 * @param node 当前节点
 * @param row 要更新的行索引
 * @param col 要更新的列索引
 * @param delta 变化量
 */
void updateTree(Node* node, int row, int col, int delta) {
    // 如果当前节点包含要更新的点
    if (row >= node->row1 && row <= node->row2 &&
        col >= node->col1 && col <= node->col2) {
        node->sum += delta;

        // 如果不是叶子节点，继续递归更新子节点
        if (!node->isLeaf()) {
            if (node->topLeft && row <= node->topLeft->row2 && col <= node->topLeft->col2) {
                updateTree(node->topLeft, row, col, delta);
            } else if (node->topRight && row <= node->topRight->row2 && col >=
node->topRight->col1) {
                updateTree(node->topRight, row, col, delta);
            } else if (node->bottomLeft && row >= node->bottomLeft->row1 && col <=
node->bottomLeft->col2) {
                updateTree(node->bottomLeft, row, col, delta);
            } else if (node->bottomRight && row >= node->bottomRight->row1 && col >=
node->bottomRight->col1) {
                updateTree(node->bottomRight, row, col, delta);
            }
        }
    }
}

/***
 * 递归查询区域和
 * @param node 当前节点
 * @param row1 查询起始行
 * @param col1 查询起始列
 * @param row2 查询结束行
 * @param col2 查询结束列
 * @return 查询结果
 */
int querySum(Node* node, int row1, int col1, int row2, int col2) {
    if (node->isLeaf()) {
        return node->sum;
    }
    int sum = 0;
    if (row1 >= node->row1 && row2 <= node->row2 &&
        col1 >= node->col1 && col2 <= node->col2) {
        sum += node->sum;
    }
    if (row1 <= node->row1 && row2 >= node->row2) {
        sum += querySum(node->bottomLeft, row1, col1, row2, col2);
    }
    if (col1 <= node->col1 && col2 >= node->col2) {
        sum += querySum(node->bottomRight, row1, col1, row2, col2);
    }
    if (row1 <= node->row1 && row2 >= node->row2 &&
        col1 <= node->col1 && col2 >= node->col2) {
        sum += querySum(node->bottomLeft, row1, col1, row2, col2);
    }
    if (row1 <= node->row1 && row2 >= node->row2 &&
        col1 >= node->col1 && col2 <= node->col2) {
        sum += querySum(node->bottomRight, row1, col1, row2, col2);
    }
    return sum;
}

```

```

*/
int queryTree(Node* node, int row1, int col1, int row2, int col2) const {
    // 查询区域完全包含当前节点
    if (row1 <= node->row1 && row2 >= node->row2 &&
        col1 <= node->col1 && col2 >= node->col2) {
        return node->sum;
    }

    // 查询区域与当前节点无交集
    if (row2 < node->row1 || row1 > node->row2 ||
        col2 < node->col1 || col1 > node->col2) {
        return 0;
    }

    // 查询区域与当前节点有部分交集，递归查询子节点
    int sum = 0;
    if (node->topLeft) {
        sum += queryTree(node->topLeft, row1, col1, row2, col2);
    }
    if (node->topRight) {
        sum += queryTree(node->topRight, row1, col1, row2, col2);
    }
    if (node->bottomLeft) {
        sum += queryTree(node->bottomLeft, row1, col1, row2, col2);
    }
    if (node->bottomRight) {
        sum += queryTree(node->bottomRight, row1, col1, row2, col2);
    }

    return sum;
}

public:
/***
 * 初始化二维线段树
 * @param matrix 输入矩阵
 */
Code19_2DSegmentTree(const vector<vector<int>>& matrix) {
    if (matrix.empty() || matrix[0].empty()) {
        this->rows = 0;
        this->cols = 0;
        this->root = nullptr;
        return;
    }
}

```

```

this->rows = matrix.size();
this->cols = matrix[0].size();
this->matrix = matrix;

// 构建二维线段树
this->root = buildTree(0, rows - 1, 0, cols - 1);
}

/***
 * 析构函数，释放内存
 */
~Code19_2DSegmentTree() {
    delete root;
}

/***
 * 更新矩阵中某一点的值
 * @param row 行索引
 * @param col 列索引
 * @param val 新值
 */
void update(int row, int col, int val) {
    if (row < 0 || row >= rows || col < 0 || col >= cols) {
        throw invalid_argument("Index out of bounds");
    }

    int delta = val - matrix[row][col];
    matrix[row][col] = val;
    updateTree(root, row, col, delta);
}

/***
 * 查询区域和
 * @param row1 起始行
 * @param col1 起始列
 * @param row2 结束行
 * @param col2 结束列
 * @return 区域和
 */
int sumRegion(int row1, int col1, int row2, int col2) const {
    if (row1 < 0 || row1 >= rows || row2 < 0 || row2 >= rows ||
        col1 < 0 || col1 >= cols || col2 < 0 || col2 >= cols ||
```

```

        row1 > row2 || col1 > col2) {
    throw invalid_argument("Invalid query range");
}

return queryTree(root, row1, col1, row2, col2);
}

};

int main() {
    // 测试样例
    vector<vector<int>> matrix = {
        {3, 0, 1, 4, 2},
        {5, 6, 3, 2, 1},
        {1, 2, 0, 1, 5},
        {4, 1, 0, 1, 7},
        {1, 0, 3, 0, 5}
    };
}

Code19_2DSegmentTree segTree(matrix);

// 测试查询
cout << segTree.sumRegion(2, 1, 4, 3) << endl; // 输出: 8
cout << segTree.sumRegion(1, 1, 2, 2) << endl; // 输出: 11
cout << segTree.sumRegion(1, 2, 2, 4) << endl; // 输出: 12

// 测试更新
segTree.update(1, 1, 10);
cout << segTree.sumRegion(1, 1, 2, 2) << endl; // 输出: 15

return 0;
}

```

/\*

二维线段树的 C++ 实现特点：

1. **\*\*内存管理\*\*:**
  - 使用指针动态分配内存
  - 实现析构函数自动释放子节点内存，避免内存泄漏
  - 可以考虑使用智能指针（如 unique\_ptr）进一步优化内存管理
  
2. **\*\*性能优化\*\*:**
  - 对于大规模数据，可以改用数组形式存储线段树节点
  - 实现非递归版本减少函数调用开销

- 使用动态开点技术减少空间占用

### 3. \*\*时间复杂度\*\*:

- 构建树:  $O(m \times n)$
- 单点更新:  $O(\log m \times \log n)$
- 区域查询:  $O(\log m \times \log n)$

### 4. \*\*空间复杂度\*\*:

- $O(m \times n)$ , 对于完全二叉树结构
- 对于稀疏矩阵, 可以使用动态开点技术优化空间使用

算法详解:

二维线段树是线段树在二维空间的扩展，通过对行和列分别建立线段树来实现高效的二维区域操作。

算法步骤:

1. 构建二维线段树: 递归地将矩阵划分为四个子区域, 直到叶子节点
2. 实现区域和查询: 根据查询区域与当前节点区域的关系决定如何递归查询
3. 实现单点更新: 从根节点开始, 沿着包含目标点的路径向下更新节点值

适用场景:

- 需要频繁进行二维区域查询和单点更新
  - 二维范围最值查询 (最大值、最小值等)
  - 图像处理中的区域统计
  - 地理信息系统中的范围查询
- \*/

=====

文件: Code19\_2DSegmentTree.java

=====

```
package class131;

/**
 * LeetCode 304. Range Sum Query 2D - Immutable 的二维线段树实现
 * 题目链接: https://leetcode.com/problems/range-sum-query-2d-immutable/
 *
 * 虽然这道题通常用前缀和解决, 但使用二维线段树可以支持单点更新, 更加通用
 * 二维线段树用于高效处理二维区域查询, 如求和、最大值、最小值等
 *
 * 解题思路:
 * 使用线段树嵌套的方式实现二维线段树
 * 1. 外层线段树管理行范围
 * 2. 内层线段树管理列范围
```

```

* 3. 每个线段树节点保存对应区域的和
*
* 时间复杂度分析:
* - 构建树: O(m*n)
* - 单点更新: O(logm * logn)
* - 区域查询: O(logm * logn)
* 空间复杂度: O(m*n)
*/
public class Code19_2DSegmentTree {

    // 二维线段树节点定义
    static class Node {
        int sum;          // 当前区域的和
        int row1, row2;  // 行范围
        int col1, col2;  // 列范围
        Node topLeft, topRight, bottomLeft, bottomRight; // 四个子区域

        public Node(int row1, int row2, int col1, int col2) {
            this.sum = 0;
            this.row1 = row1;
            this.row2 = row2;
            this.col1 = col1;
            this.col2 = col2;
            this.topLeft = null;
            this.topRight = null;
            this.bottomLeft = null;
            this.bottomRight = null;
        }

        // 判断是否是叶子节点
        public boolean isLeaf() {
            return row1 == row2 && col1 == col2;
        }
    }

    private Node root; // 根节点
    private int[][] matrix; // 原始矩阵
    private int rows, cols; // 矩阵尺寸

    /**
     * 初始化二维线段树
     * @param matrix 输入矩阵
     */
}

```

```

public Code19_2DSegmentTree(int[][] matrix) {
    if (matrix == null || matrix.length == 0 || matrix[0].length == 0) {
        this.rows = 0;
        this.cols = 0;
        this.root = null;
        this.matrix = new int[0][0];
        return;
    }

    this.rows = matrix.length;
    this.cols = matrix[0].length;
    this.matrix = new int[rows][cols];

    // 复制输入矩阵
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            this.matrix[i][j] = matrix[i][j];
        }
    }

    // 构建二维线段树
    this.root = buildTree(0, rows - 1, 0, cols - 1);
}

/**
 * 构建二维线段树
 * @param row1 起始行
 * @param row2 结束行
 * @param col1 起始列
 * @param col2 结束列
 * @return 构建好的线段树节点
 */
private Node buildTree(int row1, int row2, int col1, int col2) {
    Node node = new Node(row1, row2, col1, col2);

    // 叶子节点
    if (row1 == row2 && col1 == col2) {
        node.sum = matrix[row1][col1];
        return node;
    }

    int midRow = row1 + (row2 - row1) / 2;
    int midCol = col1 + (col2 - col1) / 2;

```

```

// 递归构建四个子区域
if (row1 == row2) {
    // 只有一行
    node.topLeft = buildTree(row1, row2, col1, midCol);
    node.topRight = buildTree(row1, row2, midCol + 1, col2);
    node.sum = node.topLeft.sum + node.topRight.sum;
} else if (col1 == col2) {
    // 只有一列
    node.topLeft = buildTree(row1, midRow, col1, col2);
    node.bottomLeft = buildTree(midRow + 1, row2, col1, col2);
    node.sum = node.topLeft.sum + node.bottomLeft.sum;
} else {
    // 一般情况，分为四个子区域
    node.topLeft = buildTree(row1, midRow, col1, midCol);
    node.topRight = buildTree(row1, midRow, midCol + 1, col2);
    node.bottomLeft = buildTree(midRow + 1, row2, col1, midCol);
    node.bottomRight = buildTree(midRow + 1, row2, midCol + 1, col2);
    node.sum = node.topLeft.sum + node.topRight.sum + node.bottomLeft.sum +
node.bottomRight.sum;
}

return node;
}

/***
 * 更新矩阵中某一点的值
 * @param row 行索引
 * @param col 列索引
 * @param val 新值
 */
public void update(int row, int col, int val) {
    if (row < 0 || row >= rows || col < 0 || col >= cols) {
        throw new IllegalArgumentException("Index out of bounds");
    }

    int delta = val - matrix[row][col];
    matrix[row][col] = val;
    updateTree(root, row, col, delta);
}

/***
 * 递归更新线段树

```

```

* @param node 当前节点
* @param row 要更新的行索引
* @param col 要更新的列索引
* @param delta 变化量
*/
private void updateTree(Node node, int row, int col, int delta) {
    // 如果当前节点包含要更新的点
    if (row >= node.row1 && row <= node.row2 && col >= node.col1 && col <= node.col2) {
        node.sum += delta;

        // 如果不是叶子节点，继续递归更新子节点
        if (!node.isLeaf()) {
            if (node.topLeft != null && row <= node.topLeft.row2 && col <= node.topLeft.col2)
{
                updateTree(node.topLeft, row, col, delta);
            } else if (node.topRight != null && row <= node.topRight.row2 && col >=
node.topRight.col1) {
                updateTree(node.topRight, row, col, delta);
            } else if (node.bottomLeft != null && row >= node.bottomLeft.row1 && col <=
node.bottomLeft.col2) {
                updateTree(node.bottomLeft, row, col, delta);
            } else if (node.bottomRight != null && row >= node.bottomRight.row1 && col >=
node.bottomRight.col1) {
                updateTree(node.bottomRight, row, col, delta);
            }
        }
    }
}

/**
 * 查询区域和
 * @param row1 起始行
 * @param col1 起始列
 * @param row2 结束行
 * @param col2 结束列
 * @return 区域和
*/
public int sumRegion(int row1, int col1, int row2, int col2) {
    if (row1 < 0 || row1 >= rows || row2 < 0 || row2 >= rows ||
        col1 < 0 || col1 >= cols || col2 < 0 || col2 >= cols ||
        row1 > row2 || col1 > col2) {
        throw new IllegalArgumentException("Invalid query range");
    }
}

```

```

        return queryTree(root, row1, col1, row2, col2);
    }

/***
 * 递归查询区域和
 * @param node 当前节点
 * @param row1 查询起始行
 * @param col1 查询起始列
 * @param row2 查询结束行
 * @param col2 查询结束列
 * @return 查询结果
 */
private int queryTree(Node node, int row1, int col1, int row2, int col2) {
    // 查询区域完全包含当前节点
    if (row1 <= node.row1 && row2 >= node.row2 && col1 <= node.col1 && col2 >= node.col2) {
        return node.sum;
    }

    // 查询区域与当前节点无交集
    if (row2 < node.row1 || row1 > node.row2 || col2 < node.col1 || col1 > node.col2) {
        return 0;
    }

    // 查询区域与当前节点有部分交集，递归查询子节点
    int sum = 0;
    if (node.topLeft != null) {
        sum += queryTree(node.topLeft, row1, col1, row2, col2);
    }
    if (node.topRight != null) {
        sum += queryTree(node.topRight, row1, col1, row2, col2);
    }
    if (node.bottomLeft != null) {
        sum += queryTree(node.bottomLeft, row1, col1, row2, col2);
    }
    if (node.bottomRight != null) {
        sum += queryTree(node.bottomRight, row1, col1, row2, col2);
    }

    return sum;
}

public static void main(String[] args) {

```

```

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

Code19_2DSegmentTree segTree = new Code19_2DSegmentTree(matrix);

// 测试查询
System.out.println(segTree.sumRegion(2, 1, 4, 3)); // 输出: 8
System.out.println(segTree.sumRegion(1, 1, 2, 2)); // 输出: 11
System.out.println(segTree.sumRegion(1, 2, 2, 4)); // 输出: 12

// 测试更新
segTree.update(1, 1, 10);
System.out.println(segTree.sumRegion(1, 1, 2, 2)); // 输出: 15
}

}

/*
二维线段树的优化思路:

```

1. **\*\*空间优化\*\*:**
  - 对于大规模数据，可以采用数组形式存储线段树，减少对象创建的开销
  - 使用动态开点技术，只在需要时创建节点
2. **\*\*时间优化\*\*:**
  - 实现非递归版本的二维线段树，减少函数调用开销
  - 对于特定问题，可以采用分块处理等优化方法
3. **\*\*其他变种\*\*:**
  - 二维树状数组：空间更小，但只能处理求和等操作
  - 四分树：一种更扁平的二维线段树实现
  - 线段树套线段树：另一种实现二维线段树的方式，外层线段树每个节点对应一棵一维线段树

二维线段树适用于需要频繁进行二维区域查询和单点更新的场景，例如图像处理、地理信息系统等。

\*/

=====

文件: Code19\_2DSegmentTree.py

```
# class131/Code19_2DSegmentTree.py
# LeetCode 304. Range Sum Query 2D - Immutable 的二维线段树实现
# 题目链接: https://leetcode.com/problems/range-sum-query-2d-immutable/
"""
```

题目描述:

实现一个支持以下操作的二维数据结构:

1. 初始化一个二维矩阵
2. 查询任意子矩阵的元素和
3. 更新矩阵中某个位置的值

解题思路:

使用二维线段树解决二维区域查询问题。二维线段树是线段树在二维空间的扩展，通过对行和列分别建立线段树来实现高效的二维区域操作。

算法步骤:

1. 构建二维线段树: 递归地将矩阵划分为四个子区域，直到叶子节点
2. 实现区域和查询: 根据查询区域与当前节点区域的关系决定如何递归查询
3. 实现单点更新: 从根节点开始，沿着包含目标点的路径向下更新节点值

时间复杂度分析:

- 构建树:  $O(m \times n)$  - 需要遍历所有元素
  - 单点更新:  $O(\log m * \log n)$  - 在行和列两个维度上分别进行  $\log$  级操作
  - 区域查询:  $O(\log m * \log n)$  - 在行和列两个维度上分别进行  $\log$  级操作
- 空间复杂度:  $O(m \times n)$  - 需要存储所有节点信息

关键优化:

1. 使用递归实现，结构清晰
2. 合理划分区域，避免重复计算
3. 及时处理边界条件，提高效率

```
class Code19_2DSegmentTree:
```

```
"""
```

二维线段树实现，支持二维区域和查询以及单点更新

虽然 LeetCode 304 题只要求静态查询，但二维线段树可以支持动态更新，更加通用

时间复杂度:

- 构建树:  $O(m \times n)$
- 单点更新:  $O(\log m * \log n)$
- 区域查询:  $O(\log m * \log n)$

空间复杂度:  $O(m \times n)$

```

"""
class Node:
    """
    二维线段树节点定义
    """

    def __init__(self, row1, row2, col1, col2):
        """
        初始化节点
        :param row1: 行范围起始
        :param row2: 行范围结束
        :param col1: 列范围起始
        :param col2: 列范围结束
        """

        self.sum = 0          # 当前区域的和
        self.row1 = row1      # 行范围起始
        self.row2 = row2      # 行范围结束
        self.col1 = col1      # 列范围起始
        self.col2 = col2      # 列范围结束
        # 四个子区域
        self.topLeft = None   # type: Code19_2DSegmentTree.Node | None
        self.topRight = None   # type: Code19_2DSegmentTree.Node | None
        self.bottomLeft = None # type: Code19_2DSegmentTree.Node | None
        self.bottomRight = None # type: Code19_2DSegmentTree.Node | None

    def is_leaf(self):
        """
        判断是否是叶子节点
        :return: 如果是叶子节点返回 True, 否则返回 False
        """

        return self.row1 == self.row2 and self.col1 == self.col2

    def __init__(self, matrix):
        """
        初始化二维线段树
        Args:
            matrix: 输入矩阵
        """

        if not matrix or not matrix[0]:
            self.rows = 0
            self.cols = 0
            self.root = None

```

```
    self.matrix = []
    return

self.rows = len(matrix)
self.cols = len(matrix[0])
# 复制输入矩阵，避免外部修改影响内部数据
self.matrix = [row[:] for row in matrix]

# 构建二维线段树
self.root = self._build_tree(0, self.rows - 1, 0, self.cols - 1)

def _build_tree(self, row1, row2, col1, col2):
    """
    递归构建二维线段树
    """

Args:
```

```
    row1: 起始行
    row2: 结束行
    col1: 起始列
    col2: 结束列
```

```
Returns:
```

```
    构建好的线段树节点
    """

# 创建新节点
node = self.Node(row1, row2, col1, col2)

# 叶子节点：区域只包含一个元素
if row1 == row2 and col1 == col2:
    node.sum = self.matrix[row1][col1]
    return node

# 计算中点
mid_row = row1 + (row2 - row1) // 2
mid_col = col1 + (col2 - col1) // 2

# 递归构建四个子区域
if row1 == row2:
    # 只有一行，按列划分
    node.topLeft = self._build_tree(row1, row2, col1, mid_col)
    node.topRight = self._build_tree(row1, row2, mid_col + 1, col2)
    node.sum = node.topLeft.sum + node.topRight.sum
elif col1 == col2:
```

```
# 只有一列，按行划分
node.topLeft = self._build_tree(row1, mid_row, col1, col2)
node.bottomLeft = self._build_tree(mid_row + 1, row2, col1, col2)
node.sum = node.topLeft.sum + node.bottomLeft.sum

else:
    # 一般情况，分为四个子区域
    node.topLeft = self._build_tree(row1, mid_row, col1, mid_col)
    node.topRight = self._build_tree(row1, mid_row, mid_col + 1, col2)
    node.bottomLeft = self._build_tree(mid_row + 1, row2, col1, mid_col)
    node.bottomRight = self._build_tree(mid_row + 1, row2, mid_col + 1, col2)
    node.sum = (node.topLeft.sum + node.topRight.sum +
                node.bottomLeft.sum + node.bottomRight.sum)

return node
```

```
def update(self, row, col, val):
```

```
    """

```

```
    更新矩阵中某一点的值

```

```
Args:
```

```
    row: 行索引

```

```
    col: 列索引

```

```
    val: 新值

```

```
Raises:

```

```
    ValueError: 如果索引起超出范围

```

```
    """

```

```
# 检查索引范围

```

```
if row < 0 or row >= self.rows or col < 0 or col >= self.cols:
    raise ValueError("Index out of bounds")

```

```
# 计算变化量并更新矩阵

```

```
delta = val - self.matrix[row][col]

```

```
self.matrix[row][col] = val

```

```
# 更新线段树

```

```
self._update_tree(self.root, row, col, delta)

```

```
def _update_tree(self, node, row, col, delta):
    """

```

```
    递归更新线段树

```

```
Args:

```

```
    node: 当前节点

```

```
row: 要更新的行索引
col: 要更新的列索引
delta: 变化量
"""
# 如果当前节点包含要更新的点
if (row >= node.row1 and row <= node.row2 and
    col >= node.col1 and col <= node.col2):
    node.sum += delta # 更新当前节点的和

# 如果不是叶子节点，继续递归更新子节点
if not node.is_leaf():
    # 根据要更新的点的位置决定更新哪个子节点
    if node.topLeft and row <= node.topLeft.row2 and col <= node.topLeft.col2:
        self._update_tree(node.topLeft, row, col, delta)
    elif node.topRight and row <= node.topRight.row2 and col >= node.topRight.col1:
        self._update_tree(node.topRight, row, col, delta)
    elif node.bottomLeft and row >= node.bottomLeft.row1 and col <=
node.bottomLeft.col2:
        self._update_tree(node.bottomLeft, row, col, delta)
    elif node.bottomRight and row >= node.bottomRight.row1 and col >=
node.bottomRight.col1:
        self._update_tree(node.bottomRight, row, col, delta)

def sum_region(self, row1, col1, row2, col2):
"""
查询区域和

Args:
    row1: 起始行
    col1: 起始列
    row2: 结束行
    col2: 结束列

Returns:
    区域和

Raises:
    ValueError: 如果查询范围无效
"""
# 检查查询范围的有效性
if (row1 < 0 or row1 >= self.rows or row2 < 0 or row2 >= self.rows or
    col1 < 0 or col1 >= self.cols or col2 < 0 or col2 >= self.cols or
    row1 > row2 or col1 > col2):
```

Args:

```
    row1: 起始行
    col1: 起始列
    row2: 结束行
    col2: 结束列
```

Returns:

区域和

Raises:

ValueError: 如果查询范围无效
"""
# 检查查询范围的有效性
if (row1 < 0 or row1 >= self.rows or row2 < 0 or row2 >= self.rows or
 col1 < 0 or col1 >= self.cols or col2 < 0 or col2 >= self.cols or
 row1 > row2 or col1 > col2):

```
        raise ValueError("Invalid query range")

    # 调用递归查询函数
    return self._query_tree(self.root, row1, col1, row2, col2)

def _query_tree(self, node, row1, col1, row2, col2):
    """
    递归查询区域和

    Args:
        node: 当前节点
        row1: 查询起始行
        col1: 查询起始列
        row2: 查询结束行
        col2: 查询结束列

    Returns:
        查询结果
    """

    # 查询区域完全包含当前节点区域，直接返回当前节点的和
    if (row1 <= node.row1 and row2 >= node.row2 and
        col1 <= node.col1 and col2 >= node.col2):
        return node.sum

    # 查询区域与当前节点区域无交集，返回 0
    if (row2 < node.row1 or row1 > node.row2 or
        col2 < node.col1 or col1 > node.col2):
        return 0

    # 查询区域与当前节点区域有部分交集，递归查询子节点
    sum_val = 0
    if node.topLeft:
        sum_val += self._query_tree(node.topLeft, row1, col1, row2, col2)
    if node.topRight:
        sum_val += self._query_tree(node.topRight, row1, col1, row2, col2)
    if node.bottomLeft:
        sum_val += self._query_tree(node.bottomLeft, row1, col1, row2, col2)
    if node.bottomRight:
        sum_val += self._query_tree(node.bottomRight, row1, col1, row2, col2)

    return sum_val
```

```
# 测试代码
```

```

if __name__ == "__main__":
    # 测试样例
    matrix = [
        [3, 0, 1, 4, 2],
        [5, 6, 3, 2, 1],
        [1, 2, 0, 1, 5],
        [4, 1, 0, 1, 7],
        [1, 0, 3, 0, 5]
    ]

    seg_tree = Code19_2DSegmentTree(matrix)

    # 测试查询
    print(seg_tree.sum_region(2, 1, 4, 3)) # 输出: 8
    print(seg_tree.sum_region(1, 1, 2, 2)) # 输出: 11
    print(seg_tree.sum_region(1, 2, 2, 4)) # 输出: 12

    # 测试更新
    seg_tree.update(1, 1, 10)
    print(seg_tree.sum_region(1, 1, 2, 2)) # 输出: 15

```

, , ,

二维线段树的 Python 实现优化说明:

1. **内存优化:**
  - 对于大规模数据，Python 中的递归深度限制可能会成为问题
  - 可以考虑使用非递归实现或增加递归深度限制
  - 对于非常大的矩阵，可以使用动态开点技术减少内存占用
2. **性能优化:**
  - Python 的递归效率相对较低，对于性能要求高的场景可以考虑使用 C 扩展
  - 对于静态数据，可以使用前缀和方法，时间复杂度为 O(1) 查询
  - 对于需要频繁更新的场景，二维线段树的优势明显
3. **适用场景:**
  - 需要频繁进行二维区域查询和单点更新
  - 二维范围最值查询（最大值、最小值等）
  - 图像处理中的区域统计
  - 地理信息系统中的范围查询
4. **扩展方向:**
  - 支持更多操作，如区间最值查询
  - 实现动态开点版本以处理稀疏矩阵

- 实现其他二维数据结构，如二维树状数组、四分树等

,,

=====

文件: Extended\_RangeSumQueryMutable\_SegmentTree.cpp

=====

```
/**  
 * LeetCode 307. Range Sum Query - Mutable (C++版本 - 线段树解法)  
 * 题目链接: https://leetcode.com/problems/range-sum-query-mutable/  
 * 题目描述: 给定一个整数数组 nums，处理以下类型的多个查询:  
 * 1. 更新数组中某个索引位置的值  
 * 2. 计算数组中从索引 left 到 right (包含)之间的元素之和  
 *  
 * 解题思路:  
 * 使用线段树实现区间求和查询和单点更新操作  
 * 1. 线段树每个节点存储对应区间的元素和  
 * 2. 构建线段树时，叶子节点存储数组元素，非叶子节点存储子节点的和  
 * 3. 更新操作时，从根节点开始，沿着包含目标索引的路径向下更新节点值  
 * 4. 查询操作时，根据查询区间与当前节点区间的关系决定如何递归查询  
 *  
 * 时间复杂度分析:  
 * - 构建线段树: O(n)  
 * - 单点更新: O(log n)  
 * - 区间查询: O(log n)  
 * 空间复杂度: O(4n) 线段树需要约 4n 的空间  
 *  
 * 算法详解:  
 * 线段树是一种二叉树数据结构，每个节点代表数组的一个区间。对于区间求和查询问题，  
 * 线段树的每个节点存储其对应区间的元素和。通过递归地将区间划分为两部分，我们可以  
 * 高效地处理区间查询和单点更新操作。  
 *  
 * 线段树结构:  
 * 1. 每个节点代表一个区间 [l, r]  
 * 2. 叶子节点代表单个元素  
 * 3. 非叶子节点的值等于其左右子节点值的和  
 */
```

```
#include <vector>  
#include <iostream>  
using namespace std;  
  
class NumArray {
```

```

private:
    std::vector<int> tree;      // 线段树数组，存储各区间的元素和
    std::vector<int> nums;       // 原始数组的副本
    int n;                      // 数组大小

    /**
     * 构建线段树
     * @param node 当前节点在线段树数组中的索引
     * @param start 当前节点所代表区间的起始位置
     * @param end   当前节点所代表区间的结束位置
     */
    void buildTree(int node, int start, int end) {
        // 如果是叶子节点，直接存储数组元素
        if (start == end) {
            tree[node] = nums[start];
        } else {
            // 计算中点，将区间分为两部分
            int mid = start + (end - start) / 2;
            // 递归构建左子树
            buildTree(2 * node + 1, start, mid);
            // 递归构建右子树
            buildTree(2 * node + 2, mid + 1, end);
            // 当前节点的值等于左右子节点值的和
            tree[node] = tree[2 * node + 1] + tree[2 * node + 2];
        }
    }

    /**
     * 更新线段树中的值
     * @param node 当前节点在线段树数组中的索引
     * @param start 当前节点所代表区间的起始位置
     * @param end   当前节点所代表区间的结束位置
     * @param idx   要更新的数组元素索引
     * @param delta 变化量
     */
    void updateTree(int node, int start, int end, int idx, int delta) {
        // 如果当前节点区间包含要更新的索引
        if (start <= idx && idx <= end) {
            tree[node] += delta;
            // 如果不是叶子节点，继续递归更新子节点
            if (start != end) {
                int mid = start + (end - start) / 2;
                updateTree(2 * node + 1, start, mid, idx, delta);
                updateTree(2 * node + 2, mid + 1, end, idx, delta);
            }
        }
    }
}

```

```

        updateTree(2 * node + 2, mid + 1, end, idx, delta);
    }
}
}

/***
 * 查询线段树中指定区间的元素和
 * @param node 当前节点在线段树数组中的索引
 * @param start 当前节点所代表区间的起始位置
 * @param end 当前节点所代表区间的结束位置
 * @param l 查询区间的起始位置
 * @param r 查询区间的结束位置
 * @return 查询区间的元素和
*/
int queryTree(int node, int start, int end, int l, int r) {
    // 如果查询区间与当前节点区间无交集，返回 0
    if (r < start || end < l) {
        return 0;
    }
    // 如果当前节点区间完全包含在查询区间内，直接返回当前节点的值
    if (l <= start && end <= r) {
        return tree[node];
    }
    // 计算中点
    int mid = start + (end - start) / 2;
    // 递归查询左右子树
    int leftSum = queryTree(2 * node + 1, start, mid, l, r);
    int rightSum = queryTree(2 * node + 2, mid + 1, end, l, r);
    // 返回左右子树查询结果的和
    return leftSum + rightSum;
}

public:
/***
 * 构造函数，初始化线段树
 * @param nums 输入数组
 */
NumArray(std::vector<int>& nums) {
    this->nums = nums;
    this->n = nums.size();
    this->tree.resize(4 * n); // 线段树数组大小通常设为 4n
    buildTree(0, 0, n - 1); // 从根节点开始构建线段树
}

```

```

/**
 * 更新数组中某个索引位置的值
 * @param index 要更新的数组元素索引
 * @param val 新的值
 */
void update(int index, int val) {
    int delta = val - nums[index];
    nums[index] = val;
    updateTree(0, 0, n - 1, index, delta);
}

/**
 * 计算数组中从索引 left 到 right (包含)之间的元素之和
 * @param left 查询区间的起始位置
 * @param right 查询区间的结束位置
 * @return 查询区间的元素和
 */
int sumRange(int left, int right) {
    return queryTree(0, 0, n - 1, left, right);
}

};

/***
 * 测试代码
 */
int main() {
    // 示例测试
    std::vector<int> nums = {1, 3, 5};
    NumArray numArray(nums);

    std::cout << "初始数组: [1, 3, 5]" << std::endl;
    std::cout << "sumRange(0, 2): " << numArray.sumRange(0, 2) << std::endl; // 返回 1 + 3 + 5 =
9

    numArray.update(1, 2); // nums = [1, 2, 5]
    std::cout << "更新索引 1 为 2 后: [1, 2, 5]" << std::endl;
    std::cout << "sumRange(0, 2): " << numArray.sumRange(0, 2) << std::endl; // 返回 1 + 2 + 5 =
8

    return 0;
}

```

文件: Extended\_RangeSumQueryMutable\_SegmentTree.java

```
=====
/**  
 * LeetCode 307. Range Sum Query - Mutable (Java 版本 - 线段树解法)  
 * 题目链接: https://leetcode.com/problems/range-sum-query-mutable/  
 * 题目描述: 给定一个整数数组 nums，处理以下类型的多个查询:  
 * 1. 更新数组中某个索引位置的值  
 * 2. 计算数组中从索引 left 到 right (包含)之间的元素之和  
 *  
 * 解题思路:  
 * 使用线段树实现区间求和查询和单点更新操作  
 * 1. 线段树每个节点存储对应区间的元素和  
 * 2. 构建线段树时，叶子节点存储数组元素，非叶子节点存储子节点的和  
 * 3. 更新操作时，从根节点开始，沿着包含目标索引的路径向下更新节点值  
 * 4. 查询操作时，根据查询区间与当前节点区间的关系决定如何递归查询  
 *  
 * 时间复杂度分析:  
 * - 构建线段树: O(n)  
 * - 单点更新: O(log n)  
 * - 区间查询: O(log n)  
 * 空间复杂度: O(4n) 线段树需要约 4n 的空间  
 *  
 * 算法详解:  
 * 线段树是一种二叉树数据结构，每个节点代表数组的一个区间。对于区间求和查询问题，  
 * 线段树的每个节点存储其对应区间的元素和。通过递归地将区间划分为两部分，我们可以  
 * 高效地处理区间查询和单点更新操作。  
 *  
 * 线段树结构:  
 * 1. 每个节点代表一个区间 [l, r]  
 * 2. 叶子节点代表单个元素  
 * 3. 非叶子节点的值等于其左右子节点值的和  
 */
```

```
class NumArray {  
    private int[] tree; // 线段树数组，存储各区间的元素和  
    private int[] nums; // 原始数组的副本  
    private int n; // 数组大小  
  
    /**  
     * 构造函数，初始化线段树  
     * @param nums 输入数组
```

```

*/
public NumArray(int[] nums) {
    this.nums = nums.clone();
    this.n = nums.length;
    this.tree = new int[4 * n]; // 线段树数组大小通常设为 4n
    buildTree(0, 0, n - 1); // 从根节点开始构建线段树
}

/**
 * 构建线段树
 * @param node 当前节点在线段树数组中的索引
 * @param start 当前节点所代表区间的起始位置
 * @param end 当前节点所代表区间的结束位置
 */
private void buildTree(int node, int start, int end) {
    // 如果是叶子节点，直接存储数组元素
    if (start == end) {
        tree[node] = nums[start];
    } else {
        // 计算中点，将区间分为两部分
        int mid = start + (end - start) / 2;
        // 递归构建左子树
        buildTree(2 * node + 1, start, mid);
        // 递归构建右子树
        buildTree(2 * node + 2, mid + 1, end);
        // 当前节点的值等于左右子节点值的和
        tree[node] = tree[2 * node + 1] + tree[2 * node + 2];
    }
}

/**
 * 更新数组中某个索引位置的值
 * @param index 要更新的数组元素索引
 * @param val 新的值
 */
public void update(int index, int val) {
    int delta = val - nums[index];
    nums[index] = val;
    updateTree(0, 0, n - 1, index, delta);
}

/**
 * 更新线段树中的值

```

```

* @param node 当前节点在线段树数组中的索引
* @param start 当前节点所代表区间的起始位置
* @param end 当前节点所代表区间的结束位置
* @param idx 要更新的数组元素索引
* @param delta 变化量
*/
private void updateTree(int node, int start, int end, int idx, int delta) {
    // 如果当前节点区间包含要更新的索引
    if (start <= idx && idx <= end) {
        tree[node] += delta;
        // 如果不是叶子节点，继续递归更新子节点
        if (start != end) {
            int mid = start + (end - start) / 2;
            updateTree(2 * node + 1, start, mid, idx, delta);
            updateTree(2 * node + 2, mid + 1, end, idx, delta);
        }
    }
}

/***
 * 计算数组中从索引 left 到 right (包含)之间的元素之和
 * @param left 查询区间的起始位置
 * @param right 查询区间的结束位置
 * @return 查询区间的元素和
*/
public int sumRange(int left, int right) {
    return queryTree(0, 0, n - 1, left, right);
}

/***
 * 查询线段树中指定区间的元素和
 * @param node 当前节点在线段树数组中的索引
 * @param start 当前节点所代表区间的起始位置
 * @param end 当前节点所代表区间的结束位置
 * @param l 查询区间的起始位置
 * @param r 查询区间的结束位置
 * @return 查询区间的元素和
*/
private int queryTree(int node, int start, int end, int l, int r) {
    // 如果查询区间与当前节点区间无交集，返回 0
    if (r < start || end < l) {
        return 0;
    }
}

```

```

// 如果当前节点区间完全包含在查询区间内，直接返回当前节点的值
if (l <= start && end <= r) {
    return tree[node];
}
// 计算中点
int mid = start + (end - start) / 2;
// 递归查询左右子树
int leftSum = queryTree(2 * node + 1, start, mid, l, r);
int rightSum = queryTree(2 * node + 2, mid + 1, end, l, r);
// 返回左右子树查询结果的和
return leftSum + rightSum;
}

}

/***
 * 测试代码
 */
public class Extended_RangeSumQueryMutable_SegmentTree {
    public static void main(String[] args) {
        // 示例测试
        int[] nums = {1, 3, 5};
        NumArray numArray = new NumArray(nums);

        System.out.println("初始数组: [1, 3, 5]");
        System.out.println("sumRange(0, 2): " + numArray.sumRange(0, 2)); // 返回 1 + 3 + 5 = 9

        numArray.update(1, 2); // nums = [1, 2, 5]
        System.out.println("更新索引 1 为 2 后: [1, 2, 5]");
        System.out.println("sumRange(0, 2): " + numArray.sumRange(0, 2)); // 返回 1 + 2 + 5 = 8
    }
}

```

=====

文件: Extended\_RangeSumQueryMutable\_SegmentTree.py

=====

"""

LeetCode 307. Range Sum Query - Mutable (Python 版本 - 线段树解法)

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

题目描述: 给定一个整数数组 `nums`, 处理以下类型的多个查询:

1. 更新数组中某个索引位置的值
2. 计算数组中从索引 `left` 到 `right` (包含) 之间的元素之和

解题思路：

使用线段树实现区间求和查询和单点更新操作

1. 线段树每个节点存储对应区间的元素和
2. 构建线段树时，叶子节点存储数组元素，非叶子节点存储子节点的和
3. 更新操作时，从根节点开始，沿着包含目标索引的路径向下更新节点值
4. 查询操作时，根据查询区间与当前节点区间的关系决定如何递归查询

时间复杂度分析：

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

空间复杂度： $O(4n)$  线段树需要约  $4n$  的空间

算法详解：

线段树是一种二叉树数据结构，每个节点代表数组的一个区间。对于区间求和查询问题，线段树的每个节点存储其对应区间的元素和。通过递归地将区间划分为两部分，我们可以高效地处理区间查询和单点更新操作。

线段树结构：

1. 每个节点代表一个区间 $[l, r]$
  2. 叶子节点代表单个元素
  3. 非叶子节点的值等于其左右子节点值的和
- """

```
class NumArray:  
    def __init__(self, nums):  
        """  
        初始化线段树  
        :param nums: 输入数组  
        """  
  
        self.n = len(nums)  
        self.nums = nums[:]  
        self.tree = [0] * (4 * self.n) # 线段树数组大小通常设为 4n  
        self.build_tree(0, 0, self.n - 1) # 从根节点开始构建线段树  
  
    def build_tree(self, node, start, end):  
        """  
        构建线段树  
        :param node: 当前节点在线段树数组中的索引  
        :param start: 当前节点所代表区间的起始位置  
        :param end: 当前节点所代表区间的结束位置  
        """  
  
        # 如果是叶子节点，直接存储数组元素
```

```

if start == end:
    self.tree[node] = self.nums[start]
else:
    # 计算中点，将区间分为两部分
    mid = start + (end - start) // 2
    # 递归构建左子树
    self.build_tree(2 * node + 1, start, mid)
    # 递归构建右子树
    self.build_tree(2 * node + 2, mid + 1, end)
    # 当前节点的值等于左右子节点值的和
    self.tree[node] = self.tree[2 * node + 1] + self.tree[2 * node + 2]

def update(self, index, val):
    """
    更新数组中某个索引位置的值
    :param index: 要更新的数组元素索引
    :param val: 新的值
    """
    delta = val - self.nums[index]
    self.nums[index] = val
    self.update_tree(0, 0, self.n - 1, index, delta)

def update_tree(self, node, start, end, idx, delta):
    """
    更新线段树中的值
    :param node: 当前节点在线段树数组中的索引
    :param start: 当前节点所代表区间的起始位置
    :param end: 当前节点所代表区间的结束位置
    :param idx: 要更新的数组元素索引
    :param delta: 变化量
    """
    # 如果当前节点区间包含要更新的索引
    if start <= idx <= end:
        self.tree[node] += delta
        # 如果不是叶子节点，继续递归更新子节点
        if start != end:
            mid = start + (end - start) // 2
            self.update_tree(2 * node + 1, start, mid, idx, delta)
            self.update_tree(2 * node + 2, mid + 1, end, idx, delta)

def sum_range(self, left, right):
    """
    计算数组中从索引 left 到 right (包含)之间的元素之和
    """

```

```

:param left: 查询区间的起始位置
:param right: 查询区间的结束位置
:return: 查询区间的元素和
"""
return self.query_tree(0, 0, self.n - 1, left, right)

def query_tree(self, node, start, end, l, r):
    """
    查询线段树中指定区间的元素和
    :param node: 当前节点在线段树数组中的索引
    :param start: 当前节点所代表区间的起始位置
    :param end: 当前节点所代表区间的结束位置
    :param l: 查询区间的起始位置
    :param r: 查询区间的结束位置
    :return: 查询区间的元素和
"""

# 如果查询区间与当前节点区间无交集，返回 0
if r < start or end < l:
    return 0
# 如果当前节点区间完全包含在查询区间内，直接返回当前节点的值
if l <= start and end <= r:
    return self.tree[node]
# 计算中点
mid = start + (end - start) // 2
# 递归查询左右子树
left_sum = self.query_tree(2 * node + 1, start, mid, l, r)
right_sum = self.query_tree(2 * node + 2, mid + 1, end, l, r)
# 返回左右子树查询结果的和
return left_sum + right_sum

# 测试代码
if __name__ == "__main__":
    # 示例测试
    nums = [1, 3, 5]
    num_array = NumArray(nums)

    print("初始数组:", nums)
    print("sum_range(0, 2):", num_array.sum_range(0, 2))  # 返回 1 + 3 + 5 = 9

    num_array.update(1, 2)  # nums = [1, 2, 5]
    print("更新索引 1 为 2 后:", [1, 2, 5])
    print("sum_range(0, 2):", num_array.sum_range(0, 2))  # 返回 1 + 2 + 5 = 8

```

=====