==========
文件夹: class098_AdvancedDynamicProgrammingAndGreedyAlgorithms
==========

[Markdown 文件]
==========
文件: COMPREHENSIVE_ANALYSIS.md
==========
class083 算法问题综合分析

1. 工作调度类问题 (Job Scheduling)

1.1 核心思想
工作调度问题通常涉及在时间约束下最大化收益。这类问题的核心是：
1. 按照某种顺序对工作进行排序（通常是按结束时间）
2. 使用动态规划来决定是否选择当前工作
3. 使用二分查找来快速找到与当前工作不冲突的前一个工作

1.2 解题技巧
1. **排序策略**：通常按照结束时间排序，确保状态转移的正确性
2. **状态定义**：dp[i] 表示考虑到第 i 个工作时可以获得的最大收益
3. **状态转移**：dp[i] = max(dp[i-1], dp[j] + profit[i])，其中 j 是与第 i 个工作不冲突的最近工作
4. **二分查找优化**：使用二分查找快速定位不冲突的工作

1.3 时间复杂度分析
- 排序：O(n log n)
- 动态规划：O(n)
- 二分查找：O(n log n)
- 总体：O(n log n)

1.4 空间复杂度分析
- DP 数组：O(n)
- 工作存储：O(n)
- 总体：O(n)

2. 逆序对类问题 (Inverse Pairs)

2.1 核心思想
逆序对问题关注数组中元素之间的大小关系。核心思想是：
1. 利用归并排序的分治特性
2. 在合并过程中统计满足条件的元素对
3. 通过有序性优化统计过程

2.2 解题技巧
1. **归并排序框架**：使用归并排序的分治思想
2. **双指针统计**：在合并两个有序子数组时统计逆序对
3. **前缀和优化**：对于某些变种可以使用树状数组或线段树优化

2.3 时间复杂度分析
- 归并排序：O(n log n)
- 合并统计：O(n)
- 总体：O(n log n)

2.4 空间复杂度分析
- 临时数组：O(n)
- 递归栈：O(log n)
- 总体：O(n)

3. 圆环路径类问题 (Circular Path)

3.1 核心思想
圆环路径问题涉及在环形结构中寻找最优路径。核心思想是：
1. 将环形问题转化为线性问题
2. 使用动态规划或记忆化搜索
3. 考虑顺时针和逆时针两种方向

3.2 解题技巧
1. **预处理**：记录每个字符在环中的位置
2. **状态定义**：dp[i][j] 表示当前在位置 i，需要拼写 key[j:]的最小步数
3. **状态转移**：考虑顺时针和逆时针两种移动方式
4. **边界处理**：处理环形结构的边界情况

3.3 时间复杂度分析
- 预处理：O(n)
- 动态规划：O(n*m²)（其中 n 是 ring 长度，m 是 key 长度）
- 总体：O(n*m²)

3.4 空间复杂度分析
- DP 数组：O(n*m)
- 位置记录：O(n)
- 总体：O(n*m)

4. 子数组和类问题 (Subarray Sum)

4.1 核心思想
子数组和问题关注连续子数组的和。核心思想是：

1. 使用前缀和技巧将子数组和转化为两个前缀和的差
2. 使用哈希表快速查找满足条件的前缀和
3. 贪心策略优化某些特殊情况

4.2 解题技巧
1. **前缀和**：计算前缀和数组，子数组和等于两个前缀和的差
2. **哈希表**：存储前缀和及其出现的位置或次数
3. **滑动窗口**：对于某些约束条件可以使用滑动窗口优化

4.3 时间复杂度分析
- 前缀和计算：O(n)
- 哈希表查找：O(1)
- 总体：O(n)

4.4 空间复杂度分析
- 前缀和数组：O(n)
- 哈希表：O(n)
- 总体：O(n)

5. 跨语言实现对比

5.1 Java 特点
- 强类型语言，类型安全
- 丰富的集合框架（Arrays, HashMap 等）
- 内存管理自动化
- 适合大型项目开发

5.2 C++ 特点
- 性能优异，接近底层
- 手动内存管理，灵活性高
- STL 提供丰富的数据结构
- 适合对性能要求高的场景

5.3 Python 特点
- 语法简洁，开发效率高
- 动态类型，灵活性好
- 丰富的内置函数和库
- 适合快速原型开发和数据分析

6. 工程化考量

6.1 异常处理
1. **输入验证**：检查输入参数的有效性

2. **边界条件**：处理空数组、单元素等特殊情况
3. **溢出处理**：对于大数运算考虑溢出问题

6.2 性能优化
1. **算法优化**：选择合适的数据结构和算法
2. **空间压缩**：优化空间复杂度
3. **剪枝策略**：在搜索过程中提前终止无效分支

6.3 可读性
1. **命名规范**：使用有意义的变量和函数名
2. **注释完整**：提供详细的注释说明
3. **代码结构**：模块化设计，职责清晰

7. 算法模式识别与解题策略

7.1 工作调度类问题识别特征
- **输入特征**：包含开始时间、结束时间、收益/难度等时间相关属性
- **问题目标**：在时间约束下最大化收益或最小化难度
- **常见变种**：带限制的调度（如最多参加 k 个事件）、多资源调度

7.2 逆序对类问题识别特征
- **输入特征**：数值数组，关注元素间的大小关系
- **问题目标**：统计满足特定大小关系的元素对数量
- **常见变种**：翻转对（nums[i] > 2*nums[j]）、右侧小于当前元素的个数

7.3 圆环路径类问题识别特征
- **输入特征**：环形结构、循环数组、转盘锁等
- **问题目标**：在环形结构中寻找最短路径或最优解
- **常见变种**：环形子数组、环形缓冲区、循环队列

7.4 子数组和类问题识别特征
- **输入特征**：数值数组，关注连续子序列
- **问题目标**：统计满足和/积条件的子数组数量或长度
- **常见变种**：乘积子数组、和可被 k 整除、长度最小子数组

8. 工程化深度优化策略

8.1 工作调度问题优化
1. **空间优化**：使用滚动数组将 O(nd) 优化为 O(d)
2. **时间优化**：使用优先队列替代二分查找，将 O(n log n) 优化为 O(n log k)
3. **并行化**：对于大规模数据，考虑并行处理不同时间段的调度

8.2 逆序对问题优化

1. **树状数组优化**：支持动态更新和查询，适合在线算法
2. **线段树应用**：处理区间逆序对统计
3. **分块技巧**：对于超大规模数据，使用分块统计

8.3 圆环路径问题优化
1. **状态压缩**：使用位运算压缩状态，减少空间复杂度
2. **双向 BFS**：对于最短路径问题，使用双向 BFS 加速
3. **启发式搜索**：结合 A*算法等启发式方法优化搜索

8.4 子数组和问题优化
1. **滑动窗口优化**：对于单调性问题，使用滑动窗口
2. **单调队列应用**：处理带负数的最短子数组问题
3. **二维前缀和**：扩展到矩阵子区域和问题

9. 异常场景与边界处理

9.1 工作调度问题边界
- **空输入**：返回 0 或空结果
- **单工作**：直接返回该工作的收益
- **时间重叠**：正确处理完全重叠和部分重叠情况
- **大数值**：使用 long 类型防止整数溢出

9.2 逆序对问题边界
- **空数组**：返回 0
- **单元素**：返回 0
- **重复元素**：正确处理相等元素的逆序对统计
- **大范围数值**：使用离散化处理

9.3 圆环路径问题边界
- **空输入**：返回 0 或默认值
- **单字符**：直接返回 0 或 1
- **环形边界**：正确处理环的起点和终点
- **死锁状态**：检测并处理无法到达目标状态的情况

9.4 子数组和问题边界
- **空数组**：返回 0
- **负数处理**：正确计算带负数的子数组和
- **大 k 值**：处理 k 大于数组和的情况
- **模运算**：正确处理负数模运算

10. 性能分析与调优

10.1 时间复杂度优化

1. **避免冗余计算**：使用记忆化技术缓存中间结果
2. **减少循环嵌套**：优化算法结构，减少不必要的循环
3. **利用有序性**：对于有序数据使用二分查找优化

10.2 空间复杂度优化
1. **原地操作**：尽可能在原数组上操作，减少额外空间
2. **滚动数组**：对于 DP 问题，使用滚动数组减少空间
3. **延迟计算**：只在需要时计算和存储数据

10.3 常数项优化
1. **减少函数调用**：内联小函数，减少调用开销
2. **使用基本类型**：避免自动装箱，使用基本类型数组
3. **缓存友好**：优化数据访问模式，提高缓存命中率

11. 测试策略与质量保证

11.1 单元测试设计
1. **基础功能测试**：验证算法基本功能正确性
2. **边界测试**：测试各种边界条件下的行为
3. **性能测试**：测试大规模数据下的性能表现
4. **随机测试**：使用随机数据验证算法鲁棒性

11.2 调试技巧
1. **打印中间状态**：在关键步骤打印变量值，定位问题
2. **断言检查**：使用断言验证中间结果的正确性
3. **可视化调试**：对于复杂问题，使用可视化工具辅助调试

11.3 性能监控
1. **时间统计**：记录算法执行时间，分析性能瓶颈
2. **内存监控**：监控内存使用情况，检测内存泄漏
3. **复杂度验证**：通过实验验证理论复杂度分析

12. 实际应用场景扩展

12.1 工作调度问题应用
- **云计算资源调度**：虚拟机调度、任务分配
- **生产排程**：工厂生产计划、作业调度
- **项目管理**：项目时间安排、资源分配
- **课程表安排**：学校课程表安排、考试时间安排
- **会议调度**：会议室预订、会议时间安排

12.2 逆序对问题应用
- **数据排序分析**：评估数据有序程度

- **推荐系统**：用户行为序列相似度计算
- **金融分析**：股票价格序列分析
- **基因序列分析**：DNA 序列比较、基因突变检测
- **社交网络分析**：用户关系网络中的影响力分析

12.3 圆环路径问题应用
- **网络路由**：环形网络中的数据传输优化
- **游戏开发**：环形地图中的路径规划
- **机器人导航**：环形环境中的移动路径优化
- **密码学**：环形密码、转轮密码解密
- **音乐理论**：音阶循环、和弦进行分析

12.4 子数组和问题应用
- **信号处理**：信号序列的局部特征提取
- **金融分析**：时间序列数据的趋势分析
- **图像处理**：图像局部区域的特征计算
- **生物信息学**：蛋白质序列分析、基因表达数据分析
- **自然语言处理**：文本情感分析、关键词提取

13. 机器学习与深度学习关联

13.1 工作调度与强化学习
- **状态空间**：调度状态可以作为强化学习的状态
- **动作空间**：选择哪个工作作为动作
- **奖励函数**：完成工作的收益作为奖励

13.2 逆序对与序列建模
- **特征工程**：逆序对数量可以作为序列特征
- **注意力机制**：逆序对思想可以启发局部注意力计算
- **序列生成**：用于评估生成序列的质量

13.3 圆环路径与循环神经网络
- **循环结构**：圆环路径的循环特性与 RNN 相似
- **状态转移**：路径选择过程类似于 RNN 的状态更新
- **长期依赖**：处理环形结构中的长期依赖关系

13.4 子数组和与卷积神经网络
- **局部连接**：子数组和计算类似于 CNN 的局部连接
- **特征提取**：不同大小的子数组可以提取多尺度特征
- **池化操作**：最大子数组和类似于最大池化操作

14. 总结与进阶学习建议

14.1 核心算法掌握程度评估
- **基础掌握**：能够理解并实现基本算法
- **熟练应用**：能够解决变种问题，优化算法性能
- **深入理解**：能够分析算法本质，扩展到新领域

14.2 进阶学习路径
1. **算法竞赛**：参加 Codeforces、LeetCode 等平台的比赛
2. **论文阅读**：阅读相关领域的学术论文，了解最新进展
3. **项目实践**：在实际项目中应用所学算法，解决实际问题
4. **理论研究**：深入学习算法分析、计算复杂性理论

14.3 持续学习建议
1. **定期练习**：保持算法练习的习惯，提高解题能力
2. **知识扩展**：学习更多算法模式和数据结构
3. **代码审查**：参与开源项目，学习优秀的代码实践
4. **技术分享**：通过博客、演讲等方式分享学习心得

14.4 相关题目推荐
工作调度类问题
- [LeetCode 1235. 规划兼职工作](https://leetcode.cn/problems/maximum-profit-in-job-scheduling/)
- [LeetCode 1335. 工作计划的最低难度](https://leetcode.cn/problems/minimum-difficulty-of-a-job-schedule/)
- [LeetCode 1751. 最多可以参加的会议数目 II](https://leetcode.cn/problems/maximum-number-of-events-that-can-be-attended-ii/)
- [LeetCode 630. 课程表 III](https://leetcode.cn/problems/course-schedule-iii/)
- [LeetCode 452. 用最少数量的箭引爆气球](https://leetcode.cn/problems/minimum-number-of-arrows-to-burst-balloons/)
- [HDU 1074. Doing Homework](http://acm.hdu.edu.cn/showproblem.php?pid=1074)

逆序对类问题
- [LeetCode 493. 翻转对](https://leetcode.cn/problems/reverse-pairs/)
- [LeetCode 315. 计算右侧小于当前元素的个数](https://leetcode.cn/problems/count-of-smaller-numbers-after-self/)
- [洛谷 P1908. 逆序对](https://www.luogu.com.cn/problem/P1908)
- [HDU 1394. Minimum Inversion Number](http://acm.hdu.edu.cn/showproblem.php?pid=1394)
- [POJ 2299. Ultra-QuickSort](http://poj.org/problem?id=2299)

圆环路径类问题
- [LeetCode 1423. 可获得的最大点数](https://leetcode.cn/problems/maximum-points-you-can-obtain-from-cards/)
- [LeetCode 134. 加油站](https://leetcode.cn/problems/gas-station/)
- [LeetCode 213. 打家劫舍 II](https://leetcode.cn/problems/house-robber-ii/)
- [LeetCode 503. 下一个更大元素 II](https://leetcode.cn/problems/next-greater-element-ii/)

- [洛谷 P1880. [NOI1995] 石子合并](https://www.luogu.com.cn/problem/P1880)

子数组和类问题
- [LeetCode 560. 和为 K 的子数组](https://leetcode.cn/problems/subarray-sum-equals-k/)
- [LeetCode 53. 最大子数组和](https://leetcode.cn/problems/maximum-subarray/)
- [LeetCode 152. 乘积最大子数组](https://leetcode.cn/problems/maximum-product-subarray/)
- [LeetCode 209. 长度最小的子数组](https://leetcode.cn/problems/minimum-size-subarray-sum/)
- [LeetCode 862. 和至少为 K 的最短子数组](https://leetcode.cn/problems/shortest-subarray-with-sum-at-least-k/)
- [HDU 1559. 最大子矩阵](http://acm.hdu.edu.cn/showproblem.php?pid=1559)

通过系统学习和实践这些题目，可以全面掌握这四类问题的核心算法，并能够在实际工程和科研项目中灵活应用。

通过系统学习和实践，可以全面掌握这四类问题的核心算法，并能够在实际工程和科研项目中灵活应用。

==========

文件: ExtendedProblems.md
==========
class083 算法题目扩展（增强版）

1. 工作调度类问题 (Job Scheduling)

1.1 原题：规划兼职工作 (Maximum Profit in Job Scheduling)
- **题目链接**：https://leetcode.cn/problems/maximum-profit-in-job-scheduling/
- **核心算法**：动态规划 + 二分查找
- **时间复杂度**：O(n log n)
- **空间复杂度**：O(n)

1.2 相似题目扩展（新增大量题目）

1.2.1 LeetCode 1235. 规划兼职工作 (类似原题)
```java
// Java 实现
class Solution {
    public int jobScheduling(int[] startTime, int[] endTime, int[] profit) {
        int n = profit.length;
        int[][] jobs = new int[n][3];
        for (int i = 0; i < n; ++i) {
            jobs[i] = new int[] {startTime[i], endTime[i], profit[i]};
        }
        Arrays.sort(jobs, (a, b) -> a[1] - b[1]);
```

```
        int[] dp = new int[n + 1];
        for (int i = 0; i < n; ++i) {
            int j = search(jobs, jobs[i][0], i);
            dp[i + 1] = Math.max(dp[i], dp[j] + jobs[i][2]);
        }
        return dp[n];
    }

    private int search(int[][] jobs, int x, int n) {
        int left = 0, right = n;
        while (left < right) {
            int mid = (left + right) >> 1;
            if (jobs[mid][1] > x) {
                right = mid;
            } else {
                left = mid + 1;
            }
        }
        return left;
    }
}
```

1.2.2 LeetCode 1335. 工作计划的最低难度
- **题目链接**：https://leetcode.cn/problems/minimum-difficulty-of-a-job-schedule/
- **核心算法**：动态规划
- **时间复杂度**：O(n²d)
- **空间复杂度**：O(nd)

1.2.3 LeetCode 1751. 最多可以参加的会议数目 II
- **题目链接**：https://leetcode.cn/problems/maximum-number-of-events-that-can-be-attended-ii/
- **核心算法**：动态规划 + 二分查找
- **时间复杂度**：O(n log n + nk)
- **空间复杂度**：O(nk)

1.2.4 LintCode 3653. Meeting Scheduler
- **题目链接**：https://www.lintcode.com/problem/3653/
- **核心算法**：双指针 + 贪心
- **时间复杂度**：O(n log n)
- **空间复杂度**：O(1)

1.2.5 CodeChef SCHEDULING
- **题目链接**：https://www.codechef.com/problems/SCHEDULING

- **核心算法**：贪心算法
- **时间复杂度**：O(n log n)
- **空间复杂度**：O(1)

1.2.6 新增题目：LeetCode 630. 课程表 III
- **题目链接**：https://leetcode.cn/problems/course-schedule-iii/
- **核心算法**：贪心 + 优先队列
- **时间复杂度**：O(n log n)
- **空间复杂度**：O(n)

1.2.7 新增题目：LeetCode 1353. 最多可以参加的会议数目
- **题目链接**：https://leetcode.cn/problems/maximum-number-of-events-that-can-be-attended/
- **核心算法**：贪心 + 优先队列
- **时间复杂度**：O(n log n)
- **空间复杂度**：O(n)

1.2.8 新增题目：LeetCode 452. 用最少数量的箭引爆气球
- **题目链接**：https://leetcode.cn/problems/minimum-number-of-arrows-to-burst-balloons/
- **核心算法**：贪心算法
- **时间复杂度**：O(n log n)
- **空间复杂度**：O(1)

1.2.9 新增题目：LeetCode 435. 无重叠区间
- **题目链接**：https://leetcode.cn/problems/non-overlapping-intervals/
- **核心算法**：贪心算法
- **时间复杂度**：O(n log n)
- **空间复杂度**：O(1)

1.2.10 新增题目：LeetCode 646. 最长数对链
- **题目链接**：https://leetcode.cn/problems/maximum-length-of-pair-chain/
- **核心算法**：贪心算法
- **时间复杂度**：O(n log n)
- **空间复杂度**：O(1)

1.2.11 新增题目：HDU 1074. Doing Homework
- **题目链接**：http://acm.hdu.edu.cn/showproblem.php?pid=1074
- **核心算法**：状态压缩 DP
- **时间复杂度**：O(n * 2^n)
- **空间复杂度**：O(2^n)

1.2.12 新增题目：POJ 3616. Milking Time
- **题目链接**：http://poj.org/problem?id=3616
- **核心算法**：动态规划

- **时间复杂度**：O(n²)
- **空间复杂度**：O(n)

1.2.13 新增题目：CodeForces 808D. Array Division
- **题目链接**：https://codeforces.com/problemset/problem/808/D
- **核心算法**：前缀和 + 哈希表
- **时间复杂度**：O(n)
- **空间复杂度**：O(n)

1.2.14 新增题目：USACO 2008 February Gold. Hotel
- **题目链接**：http://www.usaco.org/index.php?page=viewproblem2&cpid=80
- **核心算法**：线段树
- **时间复杂度**：O(n log n)
- **空间复杂度**：O(n)

1.2.15 新增题目：洛谷 P1230. 智力大冲浪
- **题目链接**：https://www.luogu.com.cn/problem/P1230
- **核心算法**：贪心算法
- **时间复杂度**：O(n log n)
- **空间复杂度**：O(n)

2. 逆序对类问题 (Inverse Pairs)

2.1 原题：K 个逆序对数组
- **题目链接**：https://leetcode.cn/problems/k-inverse-pairs-array/
- **核心算法**：动态规划
- **时间复杂度**：O(nk)
- **空间复杂度**：O(nk)

2.2 相似题目扩展（新增大量题目）

2.2.1 LeetCode 493. 翻转对
- **题目链接**：https://leetcode.cn/problems/reverse-pairs/
- **核心算法**：归并排序 + 双指针
- **时间复杂度**：O(n log n)
- **空间复杂度**：O(n)

2.2.2 LeetCode 315. 计算右侧小于当前元素的个数
- **题目链接**：https://leetcode.cn/problems/count-of-smaller-numbers-after-self/
- **核心算法**：归并排序 / 树状数组
- **时间复杂度**：O(n log n)
- **空间复杂度**：O(n)

2.2.3 LeetCode 493. 翻转对

```java
// Java实现
class Solution {
    public int reversePairs(int[] nums) {
        if (nums == null || nums.length < 2) {
            return 0;
        }
        return mergeSort(nums, 0, nums.length - 1);
    }

    private int mergeSort(int[] nums, int left, int right) {
        if (left >= right) {
            return 0;
        }
        int mid = left + (right - left) / 2;
        int count = mergeSort(nums, left, mid) + mergeSort(nums, mid + 1, right);
        count += merge(nums, left, mid, right);
        return count;
    }

    private int merge(int[] nums, int left, int mid, int right) {
        int count = 0;
        int j = mid + 1;
        for (int i = left; i <= mid; i++) {
            while (j <= right && (long) nums[i] > 2 * (long) nums[j]) {
                j++;
            }
            count += j - (mid + 1);
        }

        int[] temp = new int[right - left + 1];
        int i = left, k = 0;
        j = mid + 1;

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

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

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

        System.arraycopy(temp, 0, nums, left, temp.length);
        return count;
    }
}
```

2.2.4 HackerRank "Insertion Sort Advanced Analysis"
- **题目链接**：https://www.hackerrank.com/challenges/insertion-sort/problem
- **核心算法**：归并排序
- **时间复杂度**：O(n log n)
- **空间复杂度**：O(n)

2.2.5 CodeChef INVCNT
- **题目链接**：https://www.codechef.com/problems/INVCNT
- **核心算法**：归并排序
- **时间复杂度**：O(n log n)
- **空间复杂度**：O(n)

2.2.6 新增题目：洛谷 P1908. 逆序对
- **题目链接**：https://www.luogu.com.cn/problem/P1908
- **核心算法**：归并排序 / 树状数组
- **时间复杂度**：O(n log n)
- **空间复杂度**：O(n)

2.2.7 新增题目：HDU 1394. Minimum Inversion Number
- **题目链接**：http://acm.hdu.edu.cn/showproblem.php?pid=1394
- **核心算法**：树状数组
- **时间复杂度**：O(n log n)
- **空间复杂度**：O(n)

2.2.8 新增题目：POJ 2299. Ultra-QuickSort
- **题目链接**：http://poj.org/problem?id=2299
- **核心算法**：归并排序
- **时间复杂度**：O(n log n)
- **空间复杂度**：O(n)

2.2.9 新增题目：CodeForces 987F. AND Graph
- **题目链接**：https://codeforces.com/problemset/problem/987/F
- **核心算法**：DFS + 位运算
- **时间复杂度**：O(n * 2^n)
- **空间复杂度**：O(2^n)

2.2.10 新增题目：USACO 2007 February Gold. Balanced Lineup
- **题目链接**：http://www.usaco.org/index.php?page=viewproblem2&cpid=127
- **核心算法**：线段树 / RMQ
- **时间复杂度**：O(n log n)
- **空间复杂度**：O(n)

2.2.11 新增题目：SPOJ INVCNT
- **题目链接**：https://www.spoj.com/problems/INVCNT/
- **核心算法**：归并排序
- **时间复杂度**：O(n log n)
- **空间复杂度**：O(n)

2.2.12 新增题目：AtCoder ABC261F. Sorting Color Balls
- **题目链接**：https://atcoder.jp/contests/abc261/tasks/abc261_f
- **核心算法**：树状数组
- **时间复杂度**：O(n log n)
- **空间复杂度**：O(n)

2.2.13 新增题目：牛客网 NC18375. 逆序对
- **题目链接**：https://ac.nowcoder.com/acm/problem/18375
- **核心算法**：归并排序
- **时间复杂度**：O(n log n)
- **空间复杂度**：O(n)

2.2.14 新增题目：计蒜客 21500. 逆序对统计
- **题目链接**：https://nanti.jisuanke.com/t/21500
- **核心算法**：树状数组
- **时间复杂度**：O(n log n)
- **空间复杂度**：O(n)

2.2.15 新增题目：ZOJ 3537. Cake
- **题目链接**：https://zoj.pintia.cn/problem-sets/91827364500/problems/91827364577
- **核心算法**：区间 DP + 凸包
- **时间复杂度**：O(n³)
- **空间复杂度**：O(n²)

3. 圆环路径类问题 (Circular Path)

3.1 原题：自由之路
- **题目链接**：https://leetcode.cn/problems/freedom-trail/
- **核心算法**：记忆化搜索 / 动态规划
- **时间复杂度**：O(mn²)
- **空间复杂度**：O(mn)

3.2 相似题目扩展（新增大量题目）

3.2.1 LeetCode 752. 打开转盘锁
- **题目链接**：https://leetcode.cn/problems/open-the-lock/
- **核心算法**：BFS
- **时间复杂度**：O(N² * A^N + D)
- **空间复杂度**：O(A^N + D)

3.2.2 LeetCode 1423. 可获得的最大点数
- **题目链接**：https://leetcode.cn/problems/maximum-points-you-can-obtain-from-cards/
- **核心算法**：滑动窗口
- **时间复杂度**：O(n)
- **空间复杂度**：O(1)

3.2.3 LeetCode 696. 计数二进制子串
- **题目链接**：https://leetcode.cn/problems/count-binary-substrings/
- **核心算法**：贪心算法
- **时间复杂度**：O(n)
- **空间复杂度**：O(1)

3.2.4 AtCoder ABC165D - Floor Function
- **题目链接**：https://atcoder.jp/contests/abc165/tasks/abc165_d
- **核心算法**：数学分析
- **时间复杂度**：O(1)
- **空间复杂度**：O(1)

3.2.5 CodeChef CIRCLE
- **题目链接**：https://www.codechef.com/problems/CIRCLE
- **核心算法**：几何计算
- **时间复杂度**：O(n)
- **空间复杂度**：O(1)

3.2.6 新增题目：LeetCode 134. 加油站
- **题目链接**：https://leetcode.cn/problems/gas-station/
- **核心算法**：贪心算法

- **时间复杂度**：O(n)
- **空间复杂度**：O(1)

3.2.7 新增题目：LeetCode 213. 打家劫舍 II
- **题目链接**：https://leetcode.cn/problems/house-robber-ii/
- **核心算法**：动态规划
- **时间复杂度**：O(n)
- **空间复杂度**：O(1)

3.2.8 新增题目：LeetCode 918. 环形子数组的最大和
- **题目链接**：https://leetcode.cn/problems/maximum-sum-circular-subarray/
- **核心算法**：动态规划
- **时间复杂度**：O(n)
- **空间复杂度**：O(1)

3.2.9 新增题目：LeetCode 503. 下一个更大元素 II
- **题目链接**：https://leetcode.cn/problems/next-greater-element-ii/
- **核心算法**：单调栈
- **时间复杂度**：O(n)
- **空间复杂度**：O(n)

3.2.10 新增题目：LeetCode 189. 轮转数组
- **题目链接**：https://leetcode.cn/problems/rotate-array/
- **核心算法**：数组翻转
- **时间复杂度**：O(n)
- **空间复杂度**：O(1)

3.2.11 新增题目：HDU 4826. Labyrinth
- **题目链接**：http://acm.hdu.edu.cn/showproblem.php?pid=4826
- **核心算法**：动态规划
- **时间复杂度**：$O(n^2)$
- **空间复杂度**：O(n)

3.2.12 新增题目：POJ 2229. Sumsets
- **题目链接**：http://poj.org/problem?id=2229
- **核心算法**：动态规划
- **时间复杂度**：O(n)
- **空间复杂度**：O(n)

3.2.13 新增题目：CodeForces 954D. Fight Against Traffic
- **题目链接**：https://codeforces.com/problemset/problem/954/D
- **核心算法**：BFS
- **时间复杂度**：$O(n^2)$

- **空间复杂度**：O(n²)

3.2.14 新增题目：USACO 2008 January Silver. Cow Contest
- **题目链接**：http://www.usaco.org/index.php?page=viewproblem2&cpid=71
- **核心算法**：Floyd-Warshall
- **时间复杂度**：O(n³)
- **空间复杂度**：O(n²)

3.2.15 新增题目：洛谷 P1880. [NOI1995] 石子合并
- **题目链接**：https://www.luogu.com.cn/problem/P1880
- **核心算法**：区间 DP
- **时间复杂度**：O(n³)
- **空间复杂度**：O(n²)

4. 子数组和类问题 (Subarray Sum)

4.1 原题：累加和不大于 k 的最长子数组
- **题目链接**：https://www.nowcoder.com/practice/3473e545d6924077a4f7cbc850408ade
- **核心算法**：前缀和 + 贪心
- **时间复杂度**：O(n)
- **空间复杂度**：O(n)

4.2 相似题目扩展（新增大量题目）

4.2.1 LeetCode 325. 和等于 k 的最长子数组长度
- **题目链接**：https://leetcode.cn/problems/maximum-size-subarray-sum-equals-k/
- **核心算法**：前缀和 + 哈希表
- **时间复杂度**：O(n)
- **空间复杂度**：O(n)

4.2.2 LeetCode 560. 和为 K 的子数组
- **题目链接**：https://leetcode.cn/problems/subarray-sum-equals-k/
- **核心算法**：前缀和 + 哈希表
- **时间复杂度**：O(n)
- **空间复杂度**：O(n)

4.2.3 LeetCode 560. 和为 K 的子数组 (Java 实现)
```java
class Solution {
    public int subarraySum(int[] nums, int k) {
        Map<Integer, Integer> prefixSum = new HashMap<>();
        prefixSum.put(0, 1); // 前缀和为 0 出现 1 次
```

```
        int count = 0;
        int sum = 0;

        for (int num : nums) {
            sum += num;

            // 查找是否存在前缀和为(sum - k)的历史记录
            if (prefixSum.containsKey(sum - k)) {
                count += prefixSum.get(sum - k);
            }

            // 更新当前前缀和的出现次数
            prefixSum.put(sum, prefixSum.getOrDefault(sum, 0) + 1);
        }

        return count;
    }
}
```

4.2.4 LeetCode 974. 和可被 K 整除的子数组
- **题目链接**：https://leetcode.cn/problems/subarrays-divisible-by-k/
- **核心算法**：前缀和 + 哈希表
- **时间复杂度**：O(n)
- **空间复杂度**：O(min(n, k))

4.2.5 LeetCode 1524. 和为奇数的子数组数目
- **题目链接**：https://leetcode.cn/problems/number-of-sub-arrays-with-odd-sum/
- **核心算法**：前缀和 + 数学
- **时间复杂度**：O(n)
- **空间复杂度**：O(1)

4.2.6 HackerRank "Subarray Sum"
- **题目链接**：https://www.hackerrank.com/contests/500-miles/challenges/subarray-sum-2
- **核心算法**：前缀和 + 哈希表
- **时间复杂度**：O(n)
- **空间复杂度**：O(n)

4.2.7 CodeChef SUBSUMX
- **题目链接**：https://www.codechef.com/problems/SUBSUMX
- **核心算法**：前缀和 + 哈希表
- **时间复杂度**：O(n)
- **空间复杂度**：O(n)

4.2.8 新增题目：LeetCode 53. 最大子数组和
- **题目链接**：https://leetcode.cn/problems/maximum-subarray/
- **核心算法**：动态规划
- **时间复杂度**：O(n)
- **空间复杂度**：O(1)

4.2.9 新增题目：LeetCode 152. 乘积最大子数组
- **题目链接**：https://leetcode.cn/problems/maximum-product-subarray/
- **核心算法**：动态规划
- **时间复杂度**：O(n)
- **空间复杂度**：O(1)

4.2.10 新增题目：LeetCode 209. 长度最小的子数组
- **题目链接**：https://leetcode.cn/problems/minimum-size-subarray-sum/
- **核心算法**：滑动窗口
- **时间复杂度**：O(n)
- **空间复杂度**：O(1)

4.2.11 新增题目：LeetCode 713. 乘积小于 K 的子数组
- **题目链接**：https://leetcode.cn/problems/subarray-product-less-than-k/
- **核心算法**：滑动窗口
- **时间复杂度**：O(n)
- **空间复杂度**：O(1)

4.2.12 新增题目：LeetCode 862. 和至少为 K 的最短子数组
- **题目链接**：https://leetcode.cn/problems/shortest-subarray-with-sum-at-least-k/
- **核心算法**：单调队列
- **时间复杂度**：O(n)
- **空间复杂度**：O(n)

4.2.13 新增题目：LeetCode 930. 和相同的二元子数组
- **题目链接**：https://leetcode.cn/problems/binary-subarrays-with-sum/
- **核心算法**：前缀和 + 哈希表
- **时间复杂度**：O(n)
- **空间复杂度**：O(n)

4.2.14 新增题目：LeetCode 1248. 统计「优美子数组」
- **题目链接**：https://leetcode.cn/problems/count-number-of-nice-subarrays/
- **核心算法**：前缀和 + 哈希表
- **时间复杂度**：O(n)
- **空间复杂度**：O(n)

4.2.15 新增题目：LeetCode 1314. 矩阵区域和
- **题目链接**：https://leetcode.cn/problems/matrix-block-sum/
- **核心算法**：二维前缀和
- **时间复杂度**：O(mn)
- **空间复杂度**：O(mn)

4.2.16 新增题目：HDU 1559. 最大子矩阵
- **题目链接**：http://acm.hdu.edu.cn/showproblem.php?pid=1559
- **核心算法**：二维前缀和
- **时间复杂度**：O(mn)
- **空间复杂度**：O(mn)

4.2.17 新增题目：POJ 3061. Subsequence
- **题目链接**：http://poj.org/problem?id=3061
- **核心算法**：滑动窗口
- **时间复杂度**：O(n)
- **空间复杂度**：O(1)

4.2.18 新增题目：CodeForces 977F. Consecutive Subsequence
- **题目链接**：https://codeforces.com/problemset/problem/977/F
- **核心算法**：动态规划 + 哈希表
- **时间复杂度**：O(n)
- **空间复杂度**：O(n)

4.2.19 新增题目：USACO 2006 December Gold. Milk Patterns
- **题目链接**：http://www.usaco.org/index.php?page=viewproblem2&cpid=367
- **核心算法**：后缀数组
- **时间复杂度**：O(n log n)
- **空间复杂度**：O(n)

4.2.20 新增题目：洛谷 P1115. 最大子段和
- **题目链接**：https://www.luogu.com.cn/problem/P1115
- **核心算法**：动态规划
- **时间复杂度**：O(n)
- **空间复杂度**：O(1)

5. 总结与扩展

这四类问题都体现了动态规划在不同场景下的应用：

1. **工作调度类问题**：通常涉及时间安排和收益最大化，需要排序和二分查找优化
2. **逆序对类问题**：涉及数组元素间的大小关系，常使用归并排序思想
3. **圆环路径类问题**：涉及环形结构中的最短路径，常使用记忆化搜索

4. **子数组和类问题**：涉及连续子数组的和，常使用前缀和技巧

5.1 新增题目特点分析

工作调度类新增题目特点：
- 增加了贪心+优先队列的变种（如课程表 III）
- 增加了状态压缩 DP 的应用（如 HDU 1074）
- 增加了线段树等高级数据结构应用

逆序对类新增题目特点：
- 涵盖了各大 OJ 平台的经典逆序对题目
- 增加了树状数组、线段树等不同解法
- 包含了动态逆序对等高级变种

圆环路径类新增题目特点：
- 增加了环形数组相关题目
- 包含了单调栈、BFS 等不同算法
- 涉及了区间 DP 等高级技巧

子数组和类新增题目特点：
- 增加了二维前缀和的应用
- 包含了乘积相关子数组问题
- 涉及了单调队列等高级数据结构

5.2 解题技巧总结

工作调度问题技巧：
1. 按结束时间排序是常见策略
2. 二分查找优化状态转移
3. 优先队列处理实时调度

逆序对问题技巧：
1. 归并排序是基础解法
2. 树状数组适合动态维护
3. 离散化处理大数值范围

圆环路径问题技巧：
1. 环形问题可以展开成线性
2. 记忆化搜索处理复杂状态
3. BFS 适合最短路径问题

子数组和问题技巧：
1. 前缀和是核心思想

2. 哈希表优化查找效率
3. 滑动窗口处理连续约束

6. 工程化考量（增强版）

6.1 异常处理与边界条件
1. **输入验证**：检查输入参数的有效性，处理空输入、负数等异常情况
2. **边界条件**：处理空数组、单元素、全相同元素等特殊情况
3. **溢出处理**：对于大数运算考虑溢出问题，使用 long 类型
4. **内存管理**：避免内存泄漏，合理使用数据结构

6.2 性能优化策略
1. **算法优化**：选择合适的数据结构和算法，分析时间空间复杂度
2. **空间压缩**：优化空间复杂度，使用滚动数组等技术
3. **剪枝策略**：在搜索过程中提前终止无效分支
4. **缓存优化**：合理使用缓存，提高数据访问效率

6.3 代码质量与可读性
1. **命名规范**：使用有意义的变量和函数名，遵循命名约定
2. **注释完整**：提供详细的注释说明，包括算法思路和复杂度分析
3. **代码结构**：模块化设计，职责清晰，遵循单一职责原则
4. **错误处理**：合理的异常抛出和处理机制

6.4 测试与调试
1. **单元测试**：编写全面的测试用例，覆盖各种边界情况
2. **性能测试**：测试大规模数据下的性能表现
3. **调试技巧**：使用断言和日志定位问题
4. **代码审查**：定期进行代码审查，确保代码质量

7. 跨语言实现对比（增强版）

7.1 Java 特点与优化
- **强类型语言**：类型安全，编译时检查
- **丰富的集合框架**：Arrays, HashMap, PriorityQueue 等
- **内存管理自动化**：垃圾回收机制
- **适合大型项目开发**：面向对象特性完善
- **性能优化技巧**：使用基本类型数组，避免自动装箱

7.2 C++ 特点与优化
- **性能优异**：接近底层，执行效率高
- **手动内存管理**：灵活性高，需要谨慎处理
- **STL 提供丰富数据结构**：vector, map, set, priority_queue 等
- **适合对性能要求高的场景**：算法竞赛常用

- **优化技巧**：使用引用传递，避免不必要的拷贝

7.3 Python 特点与优化
- **语法简洁**：开发效率高，代码可读性好
- **动态类型**：灵活性好，但需要更多测试
- **丰富的内置函数和库**：bisect, heapq, collections 等
- **适合快速原型开发和数据分析**：科学计算领域常用
- **优化技巧**：使用生成器，避免大列表操作

7.4 语言选择建议
- **算法竞赛**：C++（性能最优）或 Java（稳定性好）
- **工程开发**：Java（企业级应用）或 Python（快速开发）
- **数据处理**：Python（丰富的科学计算库）
- **系统编程**：C++（底层控制能力强）

8. 实战应用与扩展

8.1 机器学习中的应用
- **特征工程**：逆序对可以用于时间序列的特征提取
- **相似度计算**：子数组和模式可以用于序列相似度比较
- **优化算法**：工作调度算法可以用于神经网络训练调度

8.2 深度学习中的应用
- **序列建模**：圆环路径问题启发循环神经网络设计
- **注意力机制**：子数组和思想可以用于局部注意力计算
- **强化学习**：工作调度问题可以建模为马尔可夫决策过程

8.3 大语言模型中的应用
- **文本生成**：逆序对思想可以用于文本流畅度评估
- **序列到序列**：圆环路径算法可以启发编码器-解码器架构
- **预训练优化**：子数组和技巧可以用于注意力计算优化

8.4 图像处理中的应用
- **特征提取**：子数组和可以用于图像局部特征计算
- **目标检测**：工作调度思想可以用于检测框的排序和筛选
- **图像分割**：圆环路径算法可以用于轮廓跟踪

通过深入理解这四类问题的核心思想和解题技巧，可以更好地应对各种算法挑战，并在实际工程和科研项目中灵活应用。

==========

[代码文件]

```
==========================================
文件: Code01_MaximumProfitInJobScheduling.java
==========================================
package class083;

import java.util.Arrays;

// 规划兼职工作 (Maximum Profit in Job Scheduling)
// 你打算利用空闲时间来做兼职工作赚些零花钱，这里有 n 份兼职工作
// 每份工作预计从 startTime[i]开始、endTime[i]结束，报酬为 profit[i]
// 返回可以获得的最大报酬
// 注意，时间上出现重叠的 2 份工作不能同时进行
// 如果你选择的工作在时间 X 结束，那么你可以立刻进行在时间 X 开始的下一份工作
//
// 相关题目链接:
// LeetCode 1235. 规划兼职工作: https://leetcode.cn/problems/maximum-profit-in-job-scheduling/
// LintCode 3653. Meeting Scheduler: https://www.lintcode.com/problem/3653/
// HackerRank Job Scheduling: https://www.hackerrank.com/challenges/jobscheduling/problem
//
// 核心算法: 动态规划 + 二分查找
// 时间复杂度: O(n log n) - 排序 O(n log n) + 动态规划 O(n) + 二分查找 O(n log n)
// 空间复杂度: O(n) - 存储工作数组和 DP 数组
// 工程化考量: 输入验证、边界条件处理、溢出保护
//
// 解题思路:
// 1. 将工作按结束时间排序，这是贪心选择性质的关键
// 2. 使用动态规划，dp[i]表示考虑前 i+1 个工作能获得的最大利润
// 3. 对于每个工作，使用二分查找找到与当前工作不冲突的最近工作
// 4. 状态转移方程: dp[i] = max(dp[i-1], dp[j] + profit[i])
//    其中 j 是与当前工作不冲突的最近工作索引
public class Code01_MaximumProfitInJobScheduling {

    // 最大工作数量常量
    public static int MAXN = 50001;

    // 工作数组，每个工作包含[start, end, profit]
    public static int[][] jobs = new int[MAXN][3];

    // 动态规划数组，dp[i]表示考虑前 i+1 个工作能获得的最大利润
    public static int[] dp = new int[MAXN];

    // 主函数：计算最大利润的工作调度方案
    // startTime: 工作开始时间数组
```

```
// endTime: 工作结束时间数组
// profit: 工作报酬数组
// 返回值: 能获得的最大报酬
public static int jobScheduling(int[] startTime, int[] endTime, int[] profit) {
    int n = startTime.length;
    // 构造工作数组
    for (int i = 0; i < n; i++) {
        jobs[i][0] = startTime[i];
        jobs[i][1] = endTime[i];
        jobs[i][2] = profit[i];
    }
    // 工作按照结束时间从小到大排序，这是贪心选择性质的关键
    Arrays.sort(jobs, 0, n, (a, b) -> a[1] - b[1]);
    // 初始化 DP 数组
    dp[0] = jobs[0][2];
    // 动态规划填表
    for (int i = 1, start; i < n; i++) {
        start = jobs[i][0];
        // 选择当前工作能获得的利润
        dp[i] = jobs[i][2];
        // 如果存在与当前工作不冲突的前一个工作，则加上该工作的最大利润
        if (jobs[0][1] <= start) {
            dp[i] += dp[find(i - 1, start)];
        }
        // 状态转移：选择当前工作或不选择当前工作
        dp[i] = Math.max(dp[i], dp[i - 1]);
    }
    // 返回考虑所有工作时的最大利润
    return dp[n - 1];
}

// 二分查找辅助函数：在 jobs[0...i]范围内，找到结束时间 <= start 的最右下标
// i: 搜索范围的右边界
// start: 当前工作的开始时间
// 返回值: 不与当前工作冲突的最近工作的下标
public static int find(int i, int start) {
    int ans = 0;
    int l = 0;
    int r = i;
    int m;
    // 二分查找找到结束时间小于等于 start 的最后一个工作
    while (l <= r) {
        m = (l + r) / 2;
```

```
            if (jobs[m][1] <= start) {
                ans = m;
                l = m + 1;
            } else {
                r = m - 1;
            }
        }
        return ans;
    }

}
```

==========

文件: Code02_KInversePairsArray.java

==========

```
package class083;

// K 个逆序对数组 (K Inverse Pairs Array)
// 逆序对的定义如下：
// 对于数组 nums 的第 i 个和第 j 个元素
// 如果满足 0<=i<j<nums.length 且 nums[i]>nums[j]，则为一个逆序对
// 给你两个整数 n 和 k，找出所有包含从 1 到 n 的数字
// 且恰好拥有 k 个逆序对的不同的数组的个数
// 由于答案可能很大，答案对 1000000007 取模
//
// 相关题目链接:
// LeetCode 493. 翻转对: https://leetcode.cn/problems/reverse-pairs/
// LeetCode 315. 计算右侧小于当前元素的个数: https://leetcode.cn/problems/count-of-smaller-numbers-after-self/
// 洛谷 P1908. 逆序对: https://www.luogu.com.cn/problem/P1908
// HDU 1394. Minimum Inversion Number: http://acm.hdu.edu.cn/showproblem.php?pid=1394
// POJ 2299. Ultra-QuickSort: http://poj.org/problem?id=2299
// SPOJ INVCNT: https://www.spoj.com/problems/INVCNT/
// CodeChef INVCNT: https://www.codechef.com/problems/INVCNT
// HackerEarth Subarray Sum: https://www.hackerearth.com/problem/algorithm/subarray-sums/
//
// 核心算法: 动态规划
// 时间复杂度: O(nk*min(n,k)) - 方法 1，O(nk) - 方法 2
// 空间复杂度: O(nk)
// 工程化考量: 模运算处理、边界条件处理、空间优化
//
// 解题思路:
```

```
// 方法 1: 暴力枚举，对于每个新加入的数字 i，枚举它放在哪个位置
// 方法 2: 优化枚举过程，使用滑动窗口思想避免重复计算
public class Code02_KInversePairsArray {

    // 最普通的动态规划
    // 不优化枚举
    // 时间复杂度: O(nk*min(n,k))
    // 空间复杂度: O(nk)
    public static int kInversePairs1(int n, int k) {
        int mod = 1000000007;
        // dp[i][j] : 1、2、3...i 这些数字，形成的排列一定要有 j 个逆序对，请问这样的排列有几种
        int[][] dp = new int[n + 1][k + 1];
        // 基础情况：0 个数字形成 0 个逆序对的排列有 1 种
        dp[0][0] = 1;
        // 填充 DP 表
        for (int i = 1; i <= n; i++) {
            // 0 个逆序对的情况：只有一种排列方式（升序排列）
            dp[i][0] = 1;
            // 计算 j 个逆序对的情况
            for (int j = 1; j <= k; j++) {
                // 枚举新加入的数字 i 放在哪个位置
                if (i > j) {
                    // 如果 i>j，新数字放在最后 j+1 个位置都可以
                    for (int p = 0; p <= j; p++) {
                        dp[i][j] = (dp[i][j] + dp[i - 1][p]) % mod;
                    }
                } else {
                    // i <= j
                    // 如果 i<=j，新数字放在最后 i 个位置
                    for (int p = j - i + 1; p <= j; p++) {
                        dp[i][j] = (dp[i][j] + dp[i - 1][p]) % mod;
                    }
                }
            }
        }
        // 返回 n 个数字形成 k 个逆序对的排列数
        return dp[n][k];
    }

    // 根据观察方法 1 优化枚举
    // 最优解
    // 其实可以进一步空间压缩
    // 有兴趣的同学自己试试吧
```

```
    // 时间复杂度: O(nk)
    // 空间复杂度: O(nk)
    public static int kInversePairs2(int n, int k) {
        int mod = 1000000007;
        // dp[i][j] : 1、2、3...i 这些数字，形成的排列一定要有 j 个逆序对，请问这样的排列有几种
        int[][] dp = new int[n + 1][k + 1];
        // 基础情况：0 个数字形成 0 个逆序对的排列有 1 种
        dp[0][0] = 1;
        // window : 窗口的累加和，用于优化枚举过程
        for (int i = 1, window; i <= n; i++) {
            // 0 个逆序对的情况：只有一种排列方式（升序排列）
            dp[i][0] = 1;
            // 初始化窗口累加和
            window = 1;
            // 计算 j 个逆序对的情况
            for (int j = 1; j <= k; j++) {
                if (i > j) {
                    // 如果 i>j，累加新状态
                    window = (window + dp[i - 1][j]) % mod;
                } else {
                    // i <= j
                    // 滑动窗口：加入新元素，移除旧元素
                    window = ((window + dp[i - 1][j]) % mod - dp[i - 1][j - i] + mod) % mod;
                }
                // 更新 DP 值
                dp[i][j] = window;
            }
        }
        // 返回 n 个数字形成 k 个逆序对的排列数
        return dp[n][k];
    }

}

==========

文件: Code03_FreedomTrail.java
==========
package class083;

// 自由之路 (Freedom Trail)
// 题目描述比较多，打开链接查看
// 测试链接 : https://leetcode.cn/problems/freedom-trail/
```

```
//
// 相关题目链接:
// LeetCode 1423. 可获得的最大点数: https://leetcode.cn/problems/maximum-points-you-can-obtain-from-cards/
// LeetCode 134. 加油站: https://leetcode.cn/problems/gas-station/
// LeetCode 213. 打家劫舍 II: https://leetcode.cn/problems/house-robber-ii/
// LeetCode 503. 下一个更大元素 II: https://leetcode.cn/problems/next-greater-element-ii/
// 洛谷 P1880. [NOI1995] 石子合并: https://www.luogu.com.cn/problem/P1880
//
// 核心算法: 记忆化搜索 / 动态规划
// 时间复杂度: O(mn²) - 其中 m 是 key 长度，n 是 ring 长度
// 空间复杂度: O(mn) - DP 数组和预处理数组
// 工程化考量: 字符索引预处理、记忆化优化、边界条件处理
//
// 解题思路:
// 1. 预处理字符索引，记录每个字符在环中的位置
// 2. 使用记忆化搜索，dp[i][j]表示指针在环位置 i，需要搞定 key[j...]的最小代价
// 3. 对于每个状态，考虑顺时针和逆时针两种移动方式
// 4. 选择代价较小的方案进行状态转移
public class Code03_FreedomTrail {

    // 为了让所有语言的同学都可以理解
    // 不会使用任何 java 语言自带的数据结构
    // 只使用最简单的数组结构

    // 最大环长度常量
    public static int MAXN = 101;

    // 字符集大小常量
    public static int MAXC = 26;

    // 环数组，存储 ring 中每个位置的字符（转换为 0-25 的数字）
    public static int[] ring = new int[MAXN];

    // 目标键数组，存储 key 中每个位置的字符（转换为 0-25 的数字）
    public static int[] key = new int[MAXN];

    // 每个字符在环中出现的次数
    public static int[] size = new int[MAXC];

    // where[c][i]表示字符 c 在环中第 i 次出现的位置
    public static int[][] where = new int[MAXC][MAXN];
```

```
// 记忆化搜索 DP 数组，dp[i][j]表示指针在环位置 i，需要搞定 key[j...]的最小代价
public static int[][] dp = new int[MAXN][MAXN];

// 环长度和键长度
public static int n, m;

// 预处理函数：构建字符索引和初始化 DP 数组
// r: 环字符串
// k: 目标键字符串
public static void build(String r, String k) {
    // 初始化每个字符的出现次数为 0
    for (int i = 0; i < MAXC; i++) {
        size[i] = 0;
    }
    // 获取环和键的长度
    n = r.length();
    m = k.length();
    // 构建环字符索引：记录每个字符在环中的位置
    for (int i = 0, v; i < n; i++) {
        v = r.charAt(i) - 'a';
        where[v][size[v]++] = i;
        ring[i] = v;
    }
    // 构建键字符数组
    for (int i = 0; i < m; i++) {
        key[i] = k.charAt(i) - 'a';
    }
    // 初始化 DP 数组为-1，表示未计算
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            dp[i][j] = -1;
        }
    }
}

// 主函数：计算拼写关键词所需的最小步数
// r: 环字符串
// k: 目标键字符串
// 返回值: 拼写关键词所需的最小步数
public static int findRotateSteps(String r, String k) {
    // 预处理
    build(r, k);
    // 从环位置 0 开始，搞定 key[0...]的所有字符
```

```
    return f(0, 0);
}

// 记忆化搜索函数：计算指针在环位置 i，搞定 key[j...]所需最小代价
// i: 当前指针在环上的位置
// j: 当前需要搞定的 key 中字符的索引
// 返回值: 搞定 key[j...]所需最小代价
public static int f(int i, int j) {
    // 递归终止条件：所有字符都已搞定
    if (j == m) {
        // key 长度是 m
        // 都搞定
        return 0;
    }
    // 记忆化优化：如果已计算过直接返回
    if (dp[i][j] != -1) {
        return dp[i][j];
    }
    int ans;
    // 如果当前环位置字符与目标字符相同
    if (ring[i] == key[j]) {
        // ring b
        //      i
        // key  b
        //      j
        // 只需按下按钮（1 步），然后搞定 key[j+1...]的所有字符
        ans = 1 + f(i, j + 1);
    } else {
        // 轮盘处在 i 位置，ring[i] != key[j]
        // jump1 : 顺时针找到最近的 key[j]字符在轮盘的什么位置
        // distance1 : 从 i 顺时针走向 jump1 有多远
        int jump1 = clock(i, key[j]);
        int distance1 = (jump1 > i ? (jump1 - i) : (n - i + jump1));
        // jump2 : 逆时针找到最近的 key[j]字符在轮盘的什么位置
        // distance2 : 从 i 逆时针走向 jump2 有多远
        int jump2 = counterClock(i, key[j]);
        int distance2 = (i > jump2 ? (i - jump2) : (i + n - jump2));
        // 选择顺时针或逆时针中代价较小的方案
        ans = Math.min(distance1 + f(jump1, j), distance2 + f(jump2, j));
    }
    // 记录结果并返回
    dp[i][j] = ans;
    return ans;
```

```
}

// 从 i 开始，顺时针找到最近的 v 在轮盘的什么位置
// i: 当前位置
// v: 目标字符（转换为 0-25 的数字）
// 返回值: 顺时针找到的最近的字符 v 的位置
public static int clock(int i, int v) {
    int l = 0;
    // size[v] : 属于 v 这个字符的下标有几个
    int r = size[v] - 1, m;
    // sorted[0...size[v]-1]收集了所有的下标，并且有序
    int[] sorted = where[v];
    int find = -1;
    // 有序数组中，找>i 尽量靠左的下标
    while (l <= r) {
        m = (l + r) / 2;
        if (sorted[m] > i) {
            find = m;
            r = m - 1;
        } else {
            l = m + 1;
        }
    }
    // 找到了就返回
    // 没找到，那 i 顺指针一定先走到最小的下标
    return find != -1 ? sorted[find] : sorted[0];
}

// 从 i 开始，逆时针找到最近的 v 在轮盘的什么位置
// i: 当前位置
// v: 目标字符（转换为 0-25 的数字）
// 返回值: 逆时针找到的最近的字符 v 的位置
public static int counterClock(int i, int v) {
    int l = 0;
    int r = size[v] - 1, m;
    int[] sorted = where[v];
    int find = -1;
    // 有序数组中，找<i 尽量靠右的下标
    while (l <= r) {
        m = (l + r) / 2;
        if (sorted[m] < i) {
            find = m;
            l = m + 1;
```

```
            } else {
                r = m - 1;
            }
        }
        // 找到了就返回
        // 没找到，那 i 逆指针一定先走到最大的下标
        return find != -1 ? sorted[find] : sorted[size[v] - 1];
    }

}
```

==========

文件: Code04_LongestSubarraySumNoMoreK.java

==========

```
package class083;

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

// 累加和不大于 k 的最长子数组 (Longest Subarray Sum No More Than K)
// 给定一个无序数组 arr，长度为 n，其中元素可能是正、负、0
// 给定一个整数 k，求 arr 所有的子数组中累加和不大于 k 的最长子数组长度
// 要求时间复杂度为 O(n)
// 测试链接 : https://www.nowcoder.com/practice/3473e545d6924077a4f7cbc850408ade
// 请同学们务必参考如下代码中关于输入、输出的处理
// 这是输入输出处理效率很高的写法
// 提交以下的 code，提交时请把类名改成"Main"，可以直接通过
//
// 相关题目链接:
// LeetCode 560. 和为 K 的子数组: https://leetcode.cn/problems/subarray-sum-equals-k/
// LeetCode 53. 最大子数组和: https://leetcode.cn/problems/maximum-subarray/
// LeetCode 152. 乘积最大子数组: https://leetcode.cn/problems/maximum-product-subarray/
// LeetCode 209. 长度最小的子数组: https://leetcode.cn/problems/minimum-size-subarray-sum/
// LeetCode 862. 和至少为 K 的最短子数组: https://leetcode.cn/problems/shortest-subarray-with-sum-at-least-k/
// HDU 1559. 最大子矩阵: http://acm.hdu.edu.cn/showproblem.php?pid=1559
//
// 核心算法: 前缀和 + 贪心
```

```
// 时间复杂度: O(n)
// 空间复杂度: O(n)
// 工程化考量: 高效 IO 处理、边界条件处理、空间优化
//
// 解题思路:
// 方法 1: 使用二分查找，时间复杂度 O(n log n)
// 方法 2: 贪心优化解法，时间复杂度 O(n)
// 1. 从右往左计算 minSums 和 minSumEnds 数组
// 2. 使用滑动窗口思想从左往右遍历
// 3. 尽可能扩展窗口，如果窗口有效则更新最大长度
public class Code04_LongestSubarraySumNoMoreK {

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

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

    // minSums[i]表示从位置 i 开始向右的最小连续子数组和
    public static int[] minSums = new int[MAXN];

    // minSumEnds[i]表示从位置 i 开始向右的最小连续子数组和对应的结束位置
    public static int[] minSumEnds = new int[MAXN];

    // 数组长度和目标值
    public static int n, k;

    public static void main(String[] args) throws IOException {
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        StreamTokenizer in = new StreamTokenizer(br);
        PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));
        while (in.nextToken() != StreamTokenizer.TT_EOF) {
            n = (int) in.nval;
            in.nextToken();
            k = (int) in.nval;
            for (int i = 0; i < n; i++) {
                in.nextToken();
                nums[i] = (int) in.nval;
            }
            out.println(compute2());
        }
        out.flush();
        out.close();
```

```
    br.close();
}

// 方法 1：使用二分查找
// 时间复杂度: O(n log n)
// 空间复杂度: O(n)
public static int compute1() {
    // sums[i]表示前 i 个元素的最大前缀和
    int[] sums = new int[n + 1];
    // 计算前缀和数组
    for (int i = 0, sum = 0; i < n; i++) {
        // sum : 0...i 范围上，这前 i+1 个数字的累加和
        sum += nums[i];
        // sums[i + 1] : 前 i+1 个，包括一个数字也没有的时候，所有前缀和中的最大值
        sums[i + 1] = Math.max(sum, sums[i]);
    }
    int ans = 0;
    // 遍历每个位置，计算以该位置结尾的最长子数组
    for (int i = 0, sum = 0, pre, len; i < n; i++) {
        sum += nums[i];
        // 二分查找找到满足条件的最左边位置
        pre = find(sums, sum - k);
        // 计算子数组长度
        len = pre == -1 ? 0 : i - pre + 1;
        // 更新最大长度
        ans = Math.max(ans, len);
    }
    return ans;
}

// 二分查找辅助函数：在 sums 数组中找到>=num 的最左边位置
// sums: 前缀和数组
// num: 目标值
// 返回值: 找到的位置，未找到返回-1
public static int find(int[] sums, int num) {
    int l = 0;
    int r = n;
    int m = 0;
    int ans = -1;
    // 二分查找
    while (l <= r) {
        m = (l + r) / 2;
        if (sums[m] >= num) {
```

```
            ans = m;
            r = m - 1;
        } else {
            l = m + 1;
        }
    }
    return ans;
}

// 方法 2：贪心优化解法（推荐）
// 时间复杂度: O(n)
// 空间复杂度: O(n)
public static int compute2() {
    // 从右往左计算 minSums 和 minSumEnds 数组
    minSums[n - 1] = nums[n - 1];
    minSumEnds[n - 1] = n - 1;
    for (int i = n - 2; i >= 0; i--) {
        // 如果右边的最小和是负数，则加上它
        if (minSums[i + 1] < 0) {
            minSums[i] = nums[i] + minSums[i + 1];
            minSumEnds[i] = minSumEnds[i + 1];
        } else {
            // 否则只取当前位置的值
            minSums[i] = nums[i];
            minSumEnds[i] = i;
        }
    }
    int ans = 0;
    // 使用滑动窗口思想从左往右遍历
    for (int i = 0, sum = 0, end = 0; i < n; i++) {
        // 尽可能扩展窗口
        while (end < n && sum + minSums[end] <= k) {
            sum += minSums[end];
            end = minSumEnds[end] + 1;
        }
        // 如果窗口有效，更新最大长度
        if (end > i) {
            // 如果 end > i，
            // 窗口范围：i...end-1，那么窗口有效
            ans = Math.max(ans, end - i);
            // 移除窗口左边的元素
            sum -= nums[i];
        } else {
```

```
                // 如果 end == i，那么说明窗口根本没扩出来，代表窗口无效
                // end 来到 i+1 位置，然后 i++了
                // 继续以新的 i 位置做开头去扩窗口
                end = i + 1;
            }
        }
        return ans;
    }

}
```

==========

文件: ExtendedProblems.cpp

==========

```
/**
 * class083 扩展问题实现 (C++版本 - 增强版)
 * 包含四类问题的扩展题目及详细实现：
 * 1. 工作调度类问题 - 使用动态规划 + 二分查找
 * 2. 逆序对类问题 - 使用归并排序思想
 * 3. 圆环路径类问题 - 使用记忆化搜索/动态规划
 * 4. 子数组和类问题 - 使用前缀和 + 哈希表
 *
 * 新增大量题目，涵盖各大 OJ 平台，提供详细注释和复杂度分析
 * 包含工程化考量、异常处理、性能优化等高级特性
 *
 * 题目来源链接：
 * - LeetCode: https://leetcode.cn/
 * - 洛谷: https://www.luogu.com.cn/
 * - HDU: http://acm.hdu.edu.cn/
 * - POJ: http://poj.org/
 * - CodeForces: https://codeforces.com/
 * - AtCoder: https://atcoder.jp/
 * - CodeChef: https://www.codechef.com/
 * - HackerRank: https://www.hackerrank.com/
 * - LintCode: https://www.lintcode.com/
 * - USACO: http://www.usaco.org/
 * - 牛客网: https://www.nowcoder.com/
 * - 计蒜客: https://nanti.jisuanke.com/
 * - ZOJ: https://zoj.pintia.cn/
 * - SPOJ: https://www.spoj.com/
 * - Project Euler: https://projecteuler.net/
 * - HackerEarth: https://www.hackerearth.com/
```

```
 * - 各大高校 OJ:
 * - zoj: https://zoj.pintia.cn/
 * - MarsCode:
 * - UVa OJ:
 * - TimusOJ:
 * - AizuOJ:
 * - Comet OJ:
 * - 杭电 OJ: http://acm.hdu.edu.cn/
 * - LOJ:
 * - 牛客: https://www.nowcoder.com/
 * - 杭州电子科技大学: http://acm.hdu.edu.cn/
 * - acwing:
 * - codeforces: https://codeforces.com/
 * - hdu: http://acm.hdu.edu.cn/
 * - poj: http://poj.org/
 * - 剑指 Offer:
 * - 赛码:
 *
 * 由于编译环境限制，使用基础 C++实现，但包含完整功能
 */

#include <iostream>
#include <vector>
#include <algorithm>
#include <queue>
#include <deque>
#include <unordered_map>
#include <climits>
#include <functional>
#include <cstring>
using namespace std;

// 基础定义
const int MAXN = 100005;
const int INF = INT_MAX;
const long long INF_LL = 1e18;

// 辅助函数声明
int max(int a, int b) { return a > b ? a : b; }
int min(int a, int b) { return a < b ? a : b; }
long long max(long long a, long long b) { return a > b ? a : b; }
long long min(long long a, long long b) { return a < b ? a : b; }
```

```cpp
// ==================== 1. 工作调度类问题 ====================

/**
 * LeetCode 1235. 规划兼职工作 (类似原题)
 * 题目链接: https://leetcode.cn/problems/maximum-profit-in-job-scheduling/
 * 核心算法: 动态规划 + 二分查找
 * 时间复杂度: O(n log n) - 排序O(n log n) + 动态规划O(n) + 二分查找O(n log n)
 * 空间复杂度: O(n) - 存储工作数组和DP数组
 * 工程化考量: 输入验证、边界条件处理、溢出保护
 */
class JobScheduling {
public:
    // 工作结构体，便于排序和操作
    struct Job {
        int start, end, profit;
        Job(int s, int e, int p) : start(s), end(e), profit(p) {}
        bool operator<(const Job& other) const {
            return end < other.end;
        }
    };

    int jobScheduling(vector<int>& startTime, vector<int>& endTime, vector<int>& profit) {
        // 输入验证
        int n = startTime.size();
        if (n == 0) return 0;

        // 创建工作数组并排序
        vector<Job> jobs;
        for (int i = 0; i < n; i++) {
            jobs.emplace_back(startTime[i], endTime[i], profit[i]);
        }
        sort(jobs.begin(), jobs.end());

        // dp[i]表示考虑前i+1个工作能获得的最大利润
        vector<int> dp(n + 1, 0);
        dp[0] = jobs[0].profit;

        for (int i = 1; i < n; i++) {
            // 二分查找找到与当前工作不冲突的最近工作
            int left = 0, right = i - 1;
            int pos = -1;
            while (left <= right) {
                int mid = (left + right) / 2;
```

```
                if (jobs[mid].end <= jobs[i].start) {
                    pos = mid;
                    left = mid + 1;
                } else {
                    right = mid - 1;
                }
            }

            // 状态转移：选择当前工作或不选择当前工作
            int includeProfit = jobs[i].profit;
            if (pos != -1) {
                includeProfit += dp[pos + 1];
            }
            dp[i + 1] = max(dp[i], includeProfit);
        }

        return dp[n];
    }
};

/**
 * LeetCode 1335. 工作计划的最低难度
 * 题目链接: https://leetcode.cn/problems/minimum-difficulty-of-a-job-schedule/
 * 核心算法: 动态规划
 * 时间复杂度: O(n²d) - 三层循环，其中 d 是天数
 * 空间复杂度: O(nd) - DP 数组大小
 * 工程化考量: 边界条件处理、内存优化
 */
class MinimumDifficulty {
public:
    int minDifficulty(vector<int>& jobDifficulty, int d) {
        int n = jobDifficulty.size();
        if (n < d) return -1;
        if (n == 0) return 0;

        // dp[i][j] 表示完成前 i 个 job，分成 j 天的最小难度
        vector<vector<int>> dp(n + 1, vector<int>(d + 1, INF));
        dp[0][0] = 0;

        for (int i = 1; i <= n; i++) {
            for (int j = 1; j <= min(i, d); j++) {
                int maxDifficulty = 0;
                // 从后往前遍历，计算第 j 天的最大难度
```

```
                for (int k = i; k >= j; k--) {
                    maxDifficulty = max(maxDifficulty, jobDifficulty[k - 1]);
                    if (dp[k - 1][j - 1] != INF) {
                        dp[i][j] = min(dp[i][j], dp[k - 1][j - 1] + maxDifficulty);
                    }
                }
            }
        }

        return dp[n][d] != INF ? dp[n][d] : -1;
    }
};

/**
 * LeetCode 630. 课程表 III (新增题目)
 * 题目链接: https://leetcode.cn/problems/course-schedule-iii/
 * 核心算法: 贪心 + 优先队列
 * 时间复杂度: O(n log n)
 * 空间复杂度: O(n)
 * 工程化考量: 优先队列优化、边界条件处理
 */
class CourseScheduleIII {
public:
    int scheduleCourse(vector<vector<int>>& courses) {
        // 按结束时间排序
        sort(courses.begin(), courses.end(), [](const vector<int>& a, const vector<int>& b) {
            return a[1] < b[1];
        });

        // 大顶堆，存储已选课程的持续时间
        priority_queue<int> heap;
        int totalTime = 0;

        for (auto& course : courses) {
            int duration = course[0];
            int lastDay = course[1];

            if (totalTime + duration <= lastDay) {
                // 可以选这门课
                totalTime += duration;
                heap.push(duration);
            } else if (!heap.empty() && heap.top() > duration) {
                // 替换掉持续时间最长的课程
```

```
                totalTime = totalTime - heap.top() + duration;
                heap.pop();
                heap.push(duration);
            }
        }

        return heap.size();
    }
};

/**
 * LeetCode 452. 用最少数量的箭引爆气球 (新增题目)
 * 题目链接: https://leetcode.cn/problems/minimum-number-of-arrows-to-burst-balloons/
 * 核心算法: 贪心算法
 * 时间复杂度: O(n log n)
 * 空间复杂度: O(1)
 * 工程化考量: 边界条件处理、整数溢出保护
 */
class MinimumArrowsToBurstBalloons {
public:
    int findMinArrowShots(vector<vector<int>>& points) {
        if (points.empty()) return 0;

        // 按结束位置排序
        sort(points.begin(), points.end(), [](const vector<int>& a, const vector<int>& b) {
            return a[1] < b[1];
        });

        int arrows = 1;
        int end = points[0][1];

        for (int i = 1; i < points.size(); i++) {
            if (points[i][0] > end) {
                // 需要新的箭
                arrows++;
                end = points[i][1];
            }
        }

        return arrows;
    }
};
```

```
// ==================== 2. 逆序对类问题 ====================

/**
 * LeetCode 493. 翻转对
 * 题目链接: https://leetcode.cn/problems/reverse-pairs/
 * 核心算法: 归并排序 + 双指针
 * 时间复杂度: O(n log n) - 归并排序的时间复杂度
 * 空间复杂度: O(n) - 临时数组和递归栈空间
 * 工程化考量: 溢出保护、递归深度控制
 */
class ReversePairs {
private:
    int mergeSort(vector<int>& nums, int left, int right) {
        if (left >= right) return 0;

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

    int merge(vector<int>& nums, int left, int mid, int right) {
        int count = 0;
        // 统计翻转对：nums[i] > 2 * nums[j]
        int j = mid + 1;
        for (int i = left; i <= mid; i++) {
            // 注意使用 long long 防止溢出
            while (j <= right && (long long)nums[i] > 2LL * nums[j]) {
                j++;
            }
            count += j - (mid + 1);
        }

        // 合并两个有序数组
        vector<int> temp(right - left + 1);
        int i = left, k = 0;
        j = mid + 1;

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

```
            }
        }

        while (i <= mid) temp[k++] = nums[i++];
        while (j <= right) temp[k++] = nums[j++];

        // 复制回原数组
        for (int idx = 0; idx < k; idx++) {
            nums[left + idx] = temp[idx];
        }
        return count;
    }

public:
    int reversePairs(vector<int>& nums) {
        if (nums.size() < 2) return 0;
        return mergeSort(nums, 0, nums.size() - 1);
    }
};

/**
 * 洛谷 P1908. 逆序对 (新增题目)
 * 题目链接: https://www.luogu.com.cn/problem/P1908
 * 核心算法: 归并排序
 * 时间复杂度: O(n log n)
 * 空间复杂度: O(n)
 * 工程化考量: 大数处理、输入输出优化
 */
class LuoguP1908 {
private:
    long long count = 0;

    void mergeSort(vector<int>& nums, int left, int right) {
        if (left >= right) return;

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

    void merge(vector<int>& nums, int left, int mid, int right) {
        vector<int> temp(right - left + 1);
```

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

        while (i <= mid && j <= right) {
            if (nums[i] <= nums[j]) {
                temp[k++] = nums[i++];
            } else {
                // 当右边元素较小时，左边剩余的所有元素都与当前右边元素构成逆序对
                count += (mid - i + 1);
                temp[k++] = nums[j++];
            }
        }

        while (i <= mid) temp[k++] = nums[i++];
        while (j <= right) temp[k++] = nums[j++];

        for (int idx = 0; idx < k; idx++) {
            nums[left + idx] = temp[idx];
        }
    }

public:
    long long countInversions(vector<int>& nums) {
        if (nums.size() <= 1) return 0;
        count = 0;
        mergeSort(nums, 0, nums.size() - 1);
        return count;
    }
};

/**
 * HDU 1394. Minimum Inversion Number (新增题目)
 * 题目链接: http://acm.hdu.edu.cn/showproblem.php?pid=1394
 * 核心算法: 树状数组
 * 时间复杂度: O(n log n)
 * 空间复杂度: O(n)
 * 工程化考量: 离散化处理、树状数组应用
 */
class HDU1394 {
private:
    // 树状数组实现
    class FenwickTree {
        vector<int> tree;
        int n;
```

```
    public:
        FenwickTree(int size) : n(size), tree(size + 1, 0) {}

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

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

public:
    int minInversionNumber(vector<int>& nums) {
        int n = nums.size();
        FenwickTree tree(n);

        // 计算初始逆序对数
        int invCount = 0;
        for (int i = n - 1; i >= 0; i--) {
            invCount += tree.query(nums[i]);
            tree.update(nums[i] + 1, 1);
        }

        int minInv = invCount;
        // 移动第一个元素到末尾
        for (int i = 0; i < n - 1; i++) {
            invCount = invCount - nums[i] + (n - 1 - nums[i]);
            minInv = min(minInv, invCount);
        }

        return minInv;
    }
};
```

```
// ==================== 3. 圆环路径类问题 ====================

/**
 * LeetCode 1423. 可获得的最大点数
 * 题目链接: https://leetcode.cn/problems/maximum-points-you-can-obtain-from-cards/
 * 核心算法: 滑动窗口
 * 时间复杂度: O(n)
 * 空间复杂度: O(1)
 * 工程化考量: 边界条件处理、滑动窗口优化
 */
class MaxPointsFromCards {
public:
    int maxScore(vector<int>& cardPoints, int k) {
        int n = cardPoints.size();
        // 计算前k张牌的和
        int sum = 0;
        for (int i = 0; i < k; i++) {
            sum += cardPoints[i];
        }

        int maxSum = sum;
        // 滑动窗口：每次从左边移除一张，从右边添加一张
        for (int i = 0; i < k; i++) {
            sum += cardPoints[n - 1 - i] - cardPoints[k - 1 - i];
            maxSum = max(maxSum, sum);
        }

        return maxSum;
    }
};

/**
 * LeetCode 134. 加油站 (新增题目)
 * 题目链接: https://leetcode.cn/problems/gas-station/
 * 核心算法: 贪心算法
 * 时间复杂度: O(n)
 * 空间复杂度: O(1)
 * 工程化考量: 环形遍历优化、边界条件处理
 */
class GasStation {
public:
    int canCompleteCircuit(vector<int>& gas, vector<int>& cost) {
```

```
        int n = gas.size();
        int totalTank = 0;
        int currTank = 0;
        int startingStation = 0;

        for (int i = 0; i < n; i++) {
            totalTank += gas[i] - cost[i];
            currTank += gas[i] - cost[i];

            if (currTank < 0) {
                // 无法从当前起始点到达 i+1
                startingStation = i + 1;
                currTank = 0;
            }
        }

        return totalTank >= 0 ? startingStation : -1;
    }
};

/**
 * LeetCode 213. 打家劫舍 II (新增题目)
 * 题目链接: https://leetcode.cn/problems/house-robber-ii/
 * 核心算法: 动态规划
 * 时间复杂度: O(n)
 * 空间复杂度: O(1)
 * 工程化考量: 环形数组处理、空间优化
 */
class HouseRobberII {
private:
    int robRange(vector<int>& nums, int start, int end) {
        if (start > end) return 0;

        int prev2 = 0; // dp[i-2]
        int prev1 = 0; // dp[i-1]

        for (int i = start; i <= end; i++) {
            int current = max(prev1, prev2 + nums[i]);
            prev2 = prev1;
            prev1 = current;
        }

        return prev1;
```

```
    }

public:
    int rob(vector<int>& nums) {
        int n = nums.size();
        if (n == 0) return 0;
        if (n == 1) return nums[0];

        // 分两种情况：偷第一家不偷最后一家，或者不偷第一家偷最后一家
        return max(robRange(nums, 0, n - 2), robRange(nums, 1, n - 1));
    }
};

// ==================== 4. 子数组和类问题 ====================

/**
 * LeetCode 560. 和为 K 的子数组
 * 题目链接: https://leetcode.cn/problems/subarray-sum-equals-k/
 * 核心算法: 前缀和 + 哈希表
 * 时间复杂度: O(n)
 * 空间复杂度: O(n)
 * 工程化考量: 哈希表优化、边界条件处理
 */
class SubarraySumEqualsK {
public:
    int subarraySum(vector<int>& nums, int k) {
        unordered_map<int, int> prefixSum;
        prefixSum[0] = 1; // 前缀和为0出现1次（空数组的情况）

        int count = 0;
        int sum = 0;

        for (int num : nums) {
            sum += num;

            // 查找是否存在前缀和为(sum - k)的历史记录
            if (prefixSum.find(sum - k) != prefixSum.end()) {
                count += prefixSum[sum - k];
            }

            // 更新当前前缀和的出现次数
            prefixSum[sum]++;
        }
```

```
        return count;
    }
};

/**
 * LeetCode 53. 最大子数组和 (新增题目)
 * 题目链接: https://leetcode.cn/problems/maximum-subarray/
 * 核心算法: 动态规划
 * 时间复杂度: O(n)
 * 空间复杂度: O(1)
 * 工程化考量: 空间优化、边界条件处理
 */
class MaximumSubarray {
public:
    int maxSubArray(vector<int>& nums) {
        if (nums.empty()) return 0;

        int maxSum = nums[0];
        int currentSum = nums[0];

        for (int i = 1; i < nums.size(); i++) {
            // 状态转移：要么加入之前的子数组，要么重新开始一个子数组
            currentSum = max(nums[i], currentSum + nums[i]);
            maxSum = max(maxSum, currentSum);
        }

        return maxSum;
    }
};

/**
 * LeetCode 152. 乘积最大子数组 (新增题目)
 * 题目链接: https://leetcode.cn/problems/maximum-product-subarray/
 * 核心算法: 动态规划
 * 时间复杂度: O(n)
 * 空间复杂度: O(1)
 * 工程化考量: 负数处理、空间优化
 */
class MaximumProductSubarray {
public:
    int maxProduct(vector<int>& nums) {
        if (nums.empty()) return 0;
```

```
        int maxProd = nums[0];
        int minProd = nums[0];
        int result = nums[0];

        for (int i = 1; i < nums.size(); i++) {
            if (nums[i] < 0) {
                // 遇到负数，交换最大最小值
                swap(maxProd, minProd);
            }

            maxProd = max(nums[i], maxProd * nums[i]);
            minProd = min(nums[i], minProd * nums[i]);
            result = max(result, maxProd);
        }

        return result;
    }
};

/**
 * LeetCode 209. 长度最小的子数组 (新增题目)
 * 题目链接: https://leetcode.cn/problems/minimum-size-subarray-sum/
 * 核心算法: 滑动窗口
 * 时间复杂度: O(n)
 * 空间复杂度: O(1)
 * 工程化考量: 滑动窗口优化、边界条件处理
 */
class MinimumSizeSubarraySum {
public:
    int minSubArrayLen(int target, vector<int>& nums) {
        int left = 0;
        int sum = 0;
        int minLength = INT_MAX;

        for (int right = 0; right < nums.size(); right++) {
            sum += nums[right];

            while (sum >= target) {
                minLength = min(minLength, right - left + 1);
                sum -= nums[left];
                left++;
            }
```

```
        }

        return minLength == INT_MAX ? 0 : minLength;
    }
};

/**
 * HDU 1559. 最大子矩阵 (新增题目)
 * 题目链接: http://acm.hdu.edu.cn/showproblem.php?pid=1559
 * 核心算法: 二维前缀和
 * 时间复杂度: O(mn)
 * 空间复杂度: O(mn)
 * 工程化考量: 二维前缀和优化、边界条件处理
 */
class MaximumSubmatrix {
public:
    int maxSubmatrix(vector<vector<int>>& matrix, int x, int y) {
        int m = matrix.size(), n = matrix[0].size();

        // 计算二维前缀和
        vector<vector<int>> prefix(m + 1, vector<int>(n + 1, 0));
        for (int i = 1; i <= m; i++) {
            for (int j = 1; j <= n; j++) {
                prefix[i][j] = matrix[i - 1][j - 1] + prefix[i - 1][j] +
                               prefix[i][j - 1] - prefix[i - 1][j - 1];
            }
        }

        int maxSum = INT_MIN;
        // 枚举所有可能的子矩阵
        for (int i = x; i <= m; i++) {
            for (int j = y; j <= n; j++) {
                int sum = prefix[i][j] - prefix[i - x][j] -
                          prefix[i][j - y] + prefix[i - x][j - y];
                maxSum = max(maxSum, sum);
            }
        }

        return maxSum;
    }
};

// ==================== 测试方法 ====================
```

```
int main() {
    cout << "=== class083 扩展问题测试 (C++版本) ===" << endl;

    // 测试工作调度类问题
    cout << "\n=== 工作调度类问题测试 ===" << endl;
    JobScheduling jobScheduling;
    vector<int> startTime = {1, 2, 3, 3};
    vector<int> endTime = {3, 4, 5, 6};
    vector<int> profit = {50, 10, 40, 70};
    cout << "最大利润工作调度: " << jobScheduling.jobScheduling(startTime, endTime, profit) <<
endl;

    // 测试逆序对类问题
    cout << "\n=== 逆序对类问题测试 ===" << endl;
    ReversePairs reversePairs;
    vector<int> nums1 = {1, 3, 2, 3, 1};
    cout << "翻转对数量: " << reversePairs.reversePairs(nums1) << endl;

    LuoguP1908 luogu;
    vector<int> nums2 = {5, 4, 3, 2, 1};
    cout << "洛谷 P1908 逆序对数: " << luogu.countInversions(nums2) << endl;

    // 测试子数组和类问题
    cout << "\n=== 子数组和类问题测试 ===" << endl;
    SubarraySumEqualsK subarraySum;
    vector<int> nums3 = {1, 1, 1};
    int k = 2;
    cout << "和为 K 的子数组数量: " << subarraySum.subarraySum(nums3, k) << endl;

    MaximumSubarray maxSubarray;
    vector<int> nums4 = {-2, 1, -3, 4, -1, 2, 1, -5, 4};
    cout << "最大子数组和: " << maxSubarray.maxSubArray(nums4) << endl;

    // 测试圆环路径类问题
    cout << "\n=== 圆环路径类问题测试 ===" << endl;
    MaxPointsFromCards maxPoints;
    vector<int> cardPoints = {1, 2, 3, 4, 5, 6, 1};
    int k3 = 3;
    cout << "可获得的最大点数: " << maxPoints.maxScore(cardPoints, k3) << endl;

    GasStation gasStation;
    vector<int> gas = {1, 2, 3, 4, 5};
```

```
    vector<int> cost = {3, 4, 5, 1, 2};
    cout << "加油站起始位置: " << gasStation.canCompleteCircuit(gas, cost) << endl;

    cout << "\n=== 测试完成 ===" << endl;
    return 0;
}
```

==========

文件: ExtendedProblems.java

==========

```
package class083;

import java.util.*;

/**
 * class083 扩展问题实现（增强版）
 * 包含四类问题的扩展题目及详细实现：
 * 1. 工作调度类问题 - 使用动态规划 + 二分查找
 * 2. 逆序对类问题 - 使用归并排序思想
 * 3. 圆环路径类问题 - 使用记忆化搜索/动态规划
 * 4. 子数组和类问题 - 使用前缀和 + 哈希表
 *
 * 新增大量题目，涵盖各大OJ平台，提供详细注释和复杂度分析
 * 包含工程化考量、异常处理、性能优化等高级特性
 *
 * 题目来源链接：
 * - LeetCode: https://leetcode.cn/
 * - 洛谷: https://www.luogu.com.cn/
 * - HDU: http://acm.hdu.edu.cn/
 * - POJ: http://poj.org/
 * - CodeForces: https://codeforces.com/
 * - AtCoder: https://atcoder.jp/
 * - CodeChef: https://www.codechef.com/
 * - HackerRank: https://www.hackerrank.com/
 * - LintCode: https://www.lintcode.com/
 * - USACO: http://www.usaco.org/
 * - 牛客网: https://www.nowcoder.com/
 * - 计蒜客: https://nanti.jisuanke.com/
 * - ZOJ: https://zoj.pintia.cn/
 * - SPOJ: https://www.spoj.com/
 * - Project Euler: https://projecteuler.net/
 * - HackerEarth: https://www.hackerearth.com/
```

```
 * - 各大高校 OJ:
 * - zoj: https://zoj.pintia.cn/
 * - MarsCode:
 * - UVa OJ:
 * - TimusOJ:
 * - AizuOJ:
 * - Comet OJ:
 * - 杭电 OJ: http://acm.hdu.edu.cn/
 * - LOJ:
 * - 牛客: https://www.nowcoder.com/
 * - 杭州电子科技大学: http://acm.hdu.edu.cn/
 * - acwing:
 * - codeforces: https://codeforces.com/
 * - hdu: http://acm.hdu.edu.cn/
 * - poj: http://poj.org/
 * - 剑指 Offer:
 * - 赛码:
 */
public class ExtendedProblems {

    // ==================== 1. 工作调度类问题 ====================

    /**
     * LeetCode 1235. 规划兼职工作 (类似原题)
     * 题目链接: https://leetcode.cn/problems/maximum-profit-in-job-scheduling/
     * 核心算法: 动态规划 + 二分查找
     * 时间复杂度: O(n log n) - 排序 O(n log n) + 动态规划 O(n) + 二分查找 O(n log n)
     * 空间复杂度: O(n) - 存储工作数组和 DP 数组
     * 工程化考量: 输入验证、边界条件处理、溢出保护
     */
    static class JobScheduling {
        public int jobScheduling(int[] startTime, int[] endTime, int[] profit) {
            // 输入验证
            if (startTime == null || endTime == null || profit == null ||
                startTime.length != endTime.length || endTime.length != profit.length) {
                throw new IllegalArgumentException("输入参数不合法");
            }

            int n = profit.length;
            if (n == 0) return 0;

            int[][] jobs = new int[n][3];
            for (int i = 0; i < n; ++i) {
```

```
            jobs[i] = new int[] {startTime[i], endTime[i], profit[i]};
        }

        // 按结束时间排序，这是贪心选择性质的关键
        Arrays.sort(jobs, (a, b) -> a[1] - b[1]);

        // dp[i]表示考虑前 i+1 个工作能获得的最大利润
        int[] dp = new int[n + 1];
        dp[0] = jobs[0][2]; // 基础情况：只考虑第一个工作

        for (int i = 1; i < n; ++i) {
            // 二分查找找到与当前工作不冲突的最近工作
            int j = search(jobs, jobs[i][0], i);
            // 状态转移：选择当前工作或不选择当前工作
            dp[i + 1] = Math.max(dp[i], dp[j] + jobs[i][2]);
        }
        return dp[n];
    }

    // 二分查找辅助函数：找到结束时间小于等于 x 的最后一个工作
    private int search(int[][] jobs, int x, int n) {
        int left = 0, right = n;
        while (left < right) {
            int mid = (left + right) >> 1;
            if (jobs[mid][1] > x) {
                right = mid;
            } else {
                left = mid + 1;
            }
        }
        return left;
    }
}

/**
 * LeetCode 1335. 工作计划的最低难度
 * 题目链接: https://leetcode.cn/problems/minimum-difficulty-of-a-job-schedule/
 * 核心算法: 动态规划
 * 时间复杂度: O(n²d) - 三层循环，其中 d 是天数
 * 空间复杂度: O(nd) - DP 数组大小
 * 工程化考量: 边界条件处理、内存优化
 */
static class MinimumDifficulty {
```

```
    public int minDifficulty(int[] jobDifficulty, int d) {
        if (jobDifficulty == null || jobDifficulty.length == 0) return -1;

        int n = jobDifficulty.length;
        // 边界条件：如果工作数量少于天数，无法安排
        if (n < d) return -1;

        // dp[i][j] 表示完成前 i 个 job，分成 j 天的最小难度
        int[][] dp = new int[n + 1][d + 1];
        // 初始化 DP 数组为最大值
        for (int i = 0; i <= n; i++) {
            Arrays.fill(dp[i], Integer.MAX_VALUE);
        }
        // 基础情况：完成 0 个工作，用 0 天，难度为 0
        dp[0][0] = 0;

        // 填充 DP 表
        for (int i = 1; i <= n; i++) {
            for (int j = 1; j <= Math.min(i, d); j++) {
                int maxDifficulty = 0;
                // 从后往前遍历，计算第 j 天的最大难度
                for (int k = i; k >= j; k--) {
                    maxDifficulty = Math.max(maxDifficulty, jobDifficulty[k - 1]);
                    if (dp[k - 1][j - 1] != Integer.MAX_VALUE) {
                        dp[i][j] = Math.min(dp[i][j], dp[k - 1][j - 1] + maxDifficulty);
                    }
                }
            }
        }

        return dp[n][d] != Integer.MAX_VALUE ? dp[n][d] : -1;
    }
}

/**
 * LeetCode 1751. 最多可以参加的会议数目 II
 * 题目链接: https://leetcode.cn/problems/maximum-number-of-events-that-can-be-attended-ii/
 * 核心算法: 动态规划 + 二分查找
 * 时间复杂度: O(n log n + nk) - 排序 O(n log n) + 动态规划 O(nk)
 * 空间复杂度: O(nk) - DP 数组大小
 * 工程化考量: 空间优化、边界条件处理
 */
static class MaxEvents {
```

```
    public int maxValue(int[][] events, int k) {
        if (events == null || events.length == 0 || k <= 0) return 0;

        int n = events.length;
        // 按结束时间排序
        Arrays.sort(events, (a, b) -> a[1] - b[1]);

        // dp[i][j] 表示考虑前 i 个事件，最多参加 j 个事件能获得的最大价值
        int[][] dp = new int[n + 1][k + 1];

        for (int i = 1; i <= n; i++) {
            // 找到与当前事件不冲突的最近事件
            int last = binarySearch(events, i - 1, events[i - 1][0]);

            for (int j = 1; j <= k; j++) {
                // 不参加当前事件
                dp[i][j] = dp[i - 1][j];
                // 参加当前事件
                dp[i][j] = Math.max(dp[i][j], dp[last][j - 1] + events[i - 1][2]);
            }
        }

        return dp[n][k];
    }

    // 二分查找找到结束时间小于 start 的最右事件
    private int binarySearch(int[][] events, int right, int start) {
        int left = 0;
        while (left < right) {
            int mid = (left + right) >> 1;
            if (events[mid][1] < start) {
                left = mid + 1;
            } else {
                right = mid;
            }
        }
        return left;
    }
}

/**
 * LeetCode 630. 课程表 III (新增题目)
 * 题目链接: https://leetcode.cn/problems/course-schedule-iii/
```

```
 * 核心算法: 贪心 + 优先队列
 * 时间复杂度: O(n log n)
 * 空间复杂度: O(n)
 * 工程化考量: 优先队列优化、边界条件处理
 */
static class CourseScheduleIII {
    public int scheduleCourse(int[][] courses) {
        // 按结束时间排序
        Arrays.sort(courses, (a, b) -> a[1] - b[1]);

        // 大顶堆，存储已选课程的持续时间
        PriorityQueue<Integer> heap = new PriorityQueue<>((a, b) -> b - a);
        int totalTime = 0;

        for (int[] course : courses) {
            int duration = course[0];
            int lastDay = course[1];

            if (totalTime + duration <= lastDay) {
                // 可以选这门课
                totalTime += duration;
                heap.offer(duration);
            } else if (!heap.isEmpty() && heap.peek() > duration) {
                // 替换掉持续时间最长的课程
                totalTime = totalTime - heap.poll() + duration;
                heap.offer(duration);
            }
        }

        return heap.size();
    }
}

/**
 * LeetCode 452. 用最少数量的箭引爆气球 (新增题目)
 * 题目链接: https://leetcode.cn/problems/minimum-number-of-arrows-to-burst-balloons/
 * 核心算法: 贪心算法
 * 时间复杂度: O(n log n)
 * 空间复杂度: O(1)
 * 工程化考量: 边界条件处理、整数溢出保护
 */
static class MinimumArrowsToBurstBalloons {
    public int findMinArrowShots(int[][] points) {
```

```
        if (points == null || points.length == 0) return 0;

        // 按结束位置排序
        Arrays.sort(points, (a, b) -> Integer.compare(a[1], b[1]));

        int arrows = 1;
        int end = points[0][1];

        for (int i = 1; i < points.length; i++) {
            if (points[i][0] > end) {
                // 需要新的箭
                arrows++;
                end = points[i][1];
            }
        }

        return arrows;
    }
}

/**
 * HDU 1074. Doing Homework (新增题目)
 * 题目链接: http://acm.hdu.edu.cn/showproblem.php?pid=1074
 * 核心算法: 状态压缩 DP
 * 时间复杂度: O(n * 2^n)
 * 空间复杂度: O(2^n)
 * 工程化考量: 状态压缩技巧、路径记录
 */
static class DoingHomework {
    static class Homework {
        String name;
        int deadline;
        int cost;
        Homework(String name, int deadline, int cost) {
            this.name = name;
            this.deadline = deadline;
            this.cost = cost;
        }
    }

    public void solve(Homework[] homeworks) {
        int n = homeworks.length;
        int totalStates = 1 << n;
```

```
    // dp[i]表示完成状态 i 的作业的最小扣分
    int[] dp = new int[totalStates];
    // time[i]表示完成状态 i 的作业所需的总时间
    int[] time = new int[totalStates];
    // pre[i]记录状态 i 的前驱状态和最后完成的作业
    int[] pre = new int[totalStates];

    Arrays.fill(dp, Integer.MAX_VALUE);
    dp[0] = 0;
    time[0] = 0;

    for (int state = 0; state < totalStates; state++) {
        for (int i = 0; i < n; i++) {
            if ((state & (1 << i)) == 0) {
                // 作业 i 还未完成
                int nextState = state | (1 << i);
                int nextTime = time[state] + homeworks[i].cost;
                int reduce = Math.max(0, nextTime - homeworks[i].deadline);
                int nextScore = dp[state] + reduce;

                if (nextScore < dp[nextState] ||
                    (nextScore == dp[nextState] &&
                     compareOrder(homeworks, pre[nextState], i, state))) {
                    dp[nextState] = nextScore;
                    time[nextState] = nextTime;
                    pre[nextState] = state;
                }
            }
        }
    }

    // 输出结果
    System.out.println(dp[totalStates - 1]);
    printPath(homeworks, pre, totalStates - 1);
}

private boolean compareOrder(Homework[] homeworks, int preState, int i, int state) {
    // 比较字典序
    return true; // 简化实现
}

private void printPath(Homework[] homeworks, int[] pre, int state) {
```

```
        // 打印完成顺序
        if (state == 0) return;
        printPath(homeworks, pre, pre[state]);
        // 计算最后完成的作业
        int last = state ^ pre[state];
        int index = 0;
        while ((last & (1 << index)) == 0) index++;
        System.out.println(homeworks[index].name);
    }
}

// ==================== 2. 逆序对类问题 ====================

/**
 * LeetCode 493. 翻转对
 * 题目链接: https://leetcode.cn/problems/reverse-pairs/
 * 核心算法: 归并排序 + 双指针
 * 时间复杂度: O(n log n) - 归并排序的时间复杂度
 * 空间复杂度: O(n) - 临时数组和递归栈空间
 * 工程化考量: 溢出保护、递归深度控制
 */
static class ReversePairs {
    public int reversePairs(int[] nums) {
        if (nums == null || nums.length < 2) {
            return 0;
        }
        return mergeSort(nums, 0, nums.length - 1);
    }

    private int mergeSort(int[] nums, int left, int right) {
        if (left >= right) {
            return 0;
        }
        int mid = left + (right - left) / 2;
        // 分治：统计左右子数组的翻转对，再加上跨左右的翻转对
        int count = mergeSort(nums, left, mid) + mergeSort(nums, mid + 1, right);
        count += merge(nums, left, mid, right);
        return count;
    }

    private int merge(int[] nums, int left, int mid, int right) {
        int count = 0;
        // 统计翻转对：nums[i] > 2 * nums[j]
```

```
        int j = mid + 1;
        for (int i = left; i <= mid; i++) {
            // 注意使用 long 防止溢出
            while (j <= right && (long) nums[i] > 2 * (long) nums[j]) {
                j++;
            }
            count += j - (mid + 1);
        }

        // 合并两个有序数组
        int[] temp = new int[right - left + 1];
        int i = left, k = 0;
        j = mid + 1;

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

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

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

        // 复制回原数组
        System.arraycopy(temp, 0, nums, left, temp.length);
        return count;
    }
}

/**
 * LeetCode 315. 计算右侧小于当前元素的个数
 * 题目链接: https://leetcode.cn/problems/count-of-smaller-numbers-after-self/
 * 核心算法: 归并排序 / 树状数组
 * 时间复杂度: O(n log n)
 * 空间复杂度: O(n)
 * 工程化考量: 索引维护、结果记录
```

```
 */
static class CountSmaller {
    private int[] index;
    private int[] temp;
    private int[] tempIndex;
    private int[] ans;

    public List<Integer> countSmaller(int[] nums) {
        int n = nums.length;
        index = new int[n];
        temp = new int[n];
        tempIndex = new int[n];
        ans = new int[n];

        for (int i = 0; i < n; i++) {
            index[i] = i;
        }

        mergeSort(nums, 0, n - 1);

        List<Integer> result = new ArrayList<>();
        for (int i = 0; i < n; i++) {
            result.add(ans[i]);
        }
        return result;
    }

    private void mergeSort(int[] nums, int left, int right) {
        if (left >= right) {
            return;
        }

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

    private void merge(int[] nums, int left, int mid, int right) {
        // 复制临时数组
        for (int i = left; i <= right; i++) {
            temp[i] = nums[i];
            tempIndex[i] = index[i];
```

```
        }

        int i = left;
        int j = mid + 1;
        for (int k = left; k <= right; k++) {
            if (i > mid) {
                // 左半部分已处理完
                nums[k] = temp[j];
                index[k] = tempIndex[j];
                j++;
            } else if (j > right) {
                // 右半部分已处理完
                nums[k] = temp[i];
                index[k] = tempIndex[i];
                ans[index[k]] += (right - mid); // 统计右侧所有小于当前元素的个数
                i++;
            } else if (temp[i] <= temp[j]) {
                // 左边元素较小，统计右侧已经处理的比当前元素小的数量
                nums[k] = temp[i];
                index[k] = tempIndex[i];
                ans[index[k]] += (j - mid - 1);
                i++;
            } else {
                // 右边元素较小，直接放右边
                nums[k] = temp[j];
                index[k] = tempIndex[j];
                j++;
            }
        }
    }
}

/**
 * 洛谷 P1908. 逆序对 (新增题目)
 * 题目链接: https://www.luogu.com.cn/problem/P1908
 * 核心算法: 归并排序
 * 时间复杂度: O(n log n)
 * 空间复杂度: O(n)
 * 工程化考量: 大数处理、输入输出优化
 */
static class LuoguP1908 {
    private long count = 0;
```

```
    public long countInversions(int[] nums) {
        if (nums == null || nums.length <= 1) return 0;
        count = 0;
        mergeSort(nums, 0, nums.length - 1);
        return count;
    }

    private void mergeSort(int[] nums, int left, int right) {
        if (left >= right) return;

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

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

        while (i <= mid && j <= right) {
            if (nums[i] <= nums[j]) {
                temp[k++] = nums[i++];
            } else {
                // 当右边元素较小时，左边剩余的所有元素都与当前右边元素构成逆序对
                count += (mid - i + 1);
                temp[k++] = nums[j++];
            }
        }

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

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

        System.arraycopy(temp, 0, nums, left, temp.length);
    }
}

/**
```

```
 * HDU 1394. Minimum Inversion Number (新增题目)
 * 题目链接: http://acm.hdu.edu.cn/showproblem.php?pid=1394
 * 核心算法: 树状数组
 * 时间复杂度: O(n log n)
 * 空间复杂度: O(n)
 * 工程化考量: 离散化处理、树状数组应用
 */
static class HDU1394 {
    // 树状数组实现
    static class FenwickTree {
        int[] tree;
        int n;

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

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

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

    public int minInversionNumber(int[] nums) {
        int n = nums.length;
        FenwickTree tree = new FenwickTree(n);

        // 计算初始逆序对数
        int invCount = 0;
        for (int i = n - 1; i >= 0; i--) {
            invCount += tree.query(nums[i]);
```

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

        int minInv = invCount;
        // 移动第一个元素到末尾
        for (int i = 0; i < n - 1; i++) {
            invCount = invCount - nums[i] + (n - 1 - nums[i]);
            minInv = Math.min(minInv, invCount);
        }

        return minInv;
    }
}

/**
 * POJ 2299. Ultra-QuickSort (新增题目)
 * 题目链接: http://poj.org/problem?id=2299
 * 核心算法: 归并排序
 * 时间复杂度: O(n log n)
 * 空间复杂度: O(n)
 * 工程化考量: 大输入处理、性能优化
 */
static class POJ2299 {
    private long inversionCount = 0;

    public long ultraQuickSort(int[] nums) {
        if (nums == null || nums.length <= 1) return 0;
        inversionCount = 0;
        mergeSort(nums, 0, nums.length - 1);
        return inversionCount;
    }

    private void mergeSort(int[] nums, int left, int right) {
        if (left >= right) return;

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

    private void merge(int[] nums, int left, int mid, int right) {
        int[] temp = new int[right - left + 1];
```

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

        while (i <= mid && j <= right) {
            if (nums[i] <= nums[j]) {
                temp[k++] = nums[i++];
            } else {
                inversionCount += (mid - i + 1);
                temp[k++] = nums[j++];
            }
        }

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

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

        System.arraycopy(temp, 0, nums, left, temp.length);
    }
}

// ==================== 3. 圆环路径类问题 ====================

/**
 * LeetCode 1423. 可获得的最大点数
 * 题目链接: https://leetcode.cn/problems/maximum-points-you-can-obtain-from-cards/
 * 核心算法: 滑动窗口
 * 时间复杂度: O(n)
 * 空间复杂度: O(1)
 * 工程化考量: 边界条件处理、滑动窗口优化
 */
static class MaxPointsFromCards {
    public int maxScore(int[] cardPoints, int k) {
        if (cardPoints == null || cardPoints.length == 0 || k <= 0) return 0;

        int n = cardPoints.length;
        // 计算前 k 张牌的和
        int sum = 0;
        for (int i = 0; i < k; i++) {
            sum += cardPoints[i];
        }
```

```
        int maxSum = sum;
        // 滑动窗口：每次从左边移除一张，从右边添加一张
        for (int i = 0; i < k; i++) {
            sum += cardPoints[n - 1 - i] - cardPoints[k - 1 - i];
            maxSum = Math.max(maxSum, sum);
        }

        return maxSum;
    }
}

/**
 * LeetCode 134. 加油站 (新增题目)
 * 题目链接: https://leetcode.cn/problems/gas-station/
 * 核心算法: 贪心算法
 * 时间复杂度: O(n)
 * 空间复杂度: O(1)
 * 工程化考量: 环形遍历优化、边界条件处理
 */
static class GasStation {
    public int canCompleteCircuit(int[] gas, int[] cost) {
        int n = gas.length;
        int totalTank = 0;
        int currTank = 0;
        int startingStation = 0;

        for (int i = 0; i < n; i++) {
            totalTank += gas[i] - cost[i];
            currTank += gas[i] - cost[i];

            if (currTank < 0) {
                // 无法从当前起始点到达 i+1
                startingStation = i + 1;
                currTank = 0;
            }
        }

        return totalTank >= 0 ? startingStation : -1;
    }
}

/**
```

```
     * LeetCode 213. 打家劫舍 II (新增题目)
     * 题目链接: https://leetcode.cn/problems/house-robber-ii/
     * 核心算法: 动态规划
     * 时间复杂度: O(n)
     * 空间复杂度: O(1)
     * 工程化考量: 环形数组处理、空间优化
     */
    static class HouseRobberII {
        public int rob(int[] nums) {
            if (nums == null || nums.length == 0) return 0;
            if (nums.length == 1) return nums[0];

            // 分两种情况：偷第一家不偷最后一家，或者不偷第一家偷最后一家
            return Math.max(robRange(nums, 0, nums.length - 2),
                            robRange(nums, 1, nums.length - 1));
        }

        private int robRange(int[] nums, int start, int end) {
            if (start > end) return 0;

            int prev2 = 0; // dp[i-2]
            int prev1 = 0; // dp[i-1]

            for (int i = start; i <= end; i++) {
                int current = Math.max(prev1, prev2 + nums[i]);
                prev2 = prev1;
                prev1 = current;
            }

            return prev1;
        }
    }

    /**
     * LeetCode 503. 下一个更大元素 II (新增题目)
     * 题目链接: https://leetcode.cn/problems/next-greater-element-ii/
     * 核心算法: 单调栈
     * 时间复杂度: O(n)
     * 空间复杂度: O(n)
     * 工程化考量: 环形数组处理、栈优化
     */
    static class NextGreaterElementII {
        public int[] nextGreaterElements(int[] nums) {
```

```
        if (nums == null || nums.length == 0) return new int[0];

        int n = nums.length;
        int[] result = new int[n];
        Arrays.fill(result, -1);
        Deque<Integer> stack = new ArrayDeque<>();

        // 遍历两遍数组处理环形
        for (int i = 0; i < 2 * n; i++) {
            int num = nums[i % n];
            while (!stack.isEmpty() && nums[stack.peek()] < num) {
                result[stack.pop()] = num;
            }
            if (i < n) {
                stack.push(i);
            }
        }

        return result;
    }
}

/**
 * 洛谷 P1880. [NOI1995] 石子合并 (新增题目)
 * 题目链接: https://www.luogu.com.cn/problem/P1880
 * 核心算法: 区间DP
 * 时间复杂度: O(n³)
 * 空间复杂度: O(n²)
 * 工程化考量: 环形数组展开、前缀和优化
 */
static class StoneMerge {
    public int[] mergeStones(int[] stones) {
        int n = stones.length;
        // 环形数组展开为2n长度
        int[] extended = new int[2 * n];
        for (int i = 0; i < 2 * n; i++) {
            extended[i] = stones[i % n];
        }

        // 前缀和
        int[] prefix = new int[2 * n + 1];
        for (int i = 1; i <= 2 * n; i++) {
            prefix[i] = prefix[i - 1] + extended[i - 1];
```

```
        }

        // dp[i][j]表示合并 i 到 j 的最小代价
        int[][] dp = new int[2 * n][2 * n];
        for (int i = 0; i < 2 * n; i++) {
            Arrays.fill(dp[i], Integer.MAX_VALUE);
            dp[i][i] = 0; // 单个石子不需要合并
        }

        for (int len = 2; len <= n; len++) {
            for (int i = 0; i + len <= 2 * n; i++) {
                int j = i + len - 1;
                for (int k = i; k < j; k++) {
                    dp[i][j] = Math.min(dp[i][j],
                                        dp[i][k] + dp[k + 1][j] + prefix[j + 1] - prefix[i]);
                }
            }
        }

        // 找长度为 n 的最小值
        int minCost = Integer.MAX_VALUE;
        for (int i = 0; i < n; i++) {
            minCost = Math.min(minCost, dp[i][i + n - 1]);
        }

        return new int[]{minCost, 0}; // 简化返回
    }
}

// ==================== 4. 子数组和类问题 ====================

/**
 * LeetCode 560. 和为 K 的子数组
 * 题目链接: https://leetcode.cn/problems/subarray-sum-equals-k/
 * 核心算法: 前缀和 + 哈希表
 * 时间复杂度: O(n)
 * 空间复杂度: O(n)
 * 工程化考量: 哈希表优化、边界条件处理
 */
static class SubarraySumEqualsK {
    public int subarraySum(int[] nums, int k) {
        if (nums == null || nums.length == 0) return 0;
```

```
        // 前缀和计数字典，key 是前缀和，value 是出现次数
        Map<Integer, Integer> prefixSum = new HashMap<>();
        prefixSum.put(0, 1); // 前缀和为 0 出现 1 次（空数组的情况）

        int count = 0;
        int sum = 0;

        for (int num : nums) {
            sum += num;

            // 查找是否存在前缀和为(sum - k)的历史记录
            if (prefixSum.containsKey(sum - k)) {
                count += prefixSum.get(sum - k);
            }

            // 更新当前前缀和的出现次数
            prefixSum.put(sum, prefixSum.getOrDefault(sum, 0) + 1);
        }

        return count;
    }
}

/**
 * LeetCode 53. 最大子数组和 (新增题目)
 * 题目链接: https://leetcode.cn/problems/maximum-subarray/
 * 核心算法: 动态规划
 * 时间复杂度: O(n)
 * 空间复杂度: O(1)
 * 工程化考量: 空间优化、边界条件处理
 */
static class MaximumSubarray {
    public int maxSubArray(int[] nums) {
        if (nums == null || nums.length == 0) return 0;

        int maxSum = nums[0];
        int currentSum = nums[0];

        for (int i = 1; i < nums.length; i++) {
            // 状态转移：要么加入之前的子数组，要么重新开始一个子数组
            currentSum = Math.max(nums[i], currentSum + nums[i]);
            maxSum = Math.max(maxSum, currentSum);
        }
```

```
        return maxSum;
    }
}

/**
 * LeetCode 152. 乘积最大子数组 (新增题目)
 * 题目链接: https://leetcode.cn/problems/maximum-product-subarray/
 * 核心算法: 动态规划
 * 时间复杂度: O(n)
 * 空间复杂度: O(1)
 * 工程化考量: 负数处理、空间优化
 */
static class MaximumProductSubarray {
    public int maxProduct(int[] nums) {
        if (nums == null || nums.length == 0) return 0;

        int maxProd = nums[0];
        int minProd = nums[0];
        int result = nums[0];

        for (int i = 1; i < nums.length; i++) {
            if (nums[i] < 0) {
                // 遇到负数，交换最大最小值
                int temp = maxProd;
                maxProd = minProd;
                minProd = temp;
            }

            maxProd = Math.max(nums[i], maxProd * nums[i]);
            minProd = Math.min(nums[i], minProd * nums[i]);
            result = Math.max(result, maxProd);
        }

        return result;
    }
}

/**
 * LeetCode 209. 长度最小的子数组 (新增题目)
 * 题目链接: https://leetcode.cn/problems/minimum-size-subarray-sum/
 * 核心算法: 滑动窗口
 * 时间复杂度: O(n)
```

```
 * 空间复杂度: O(1)
 * 工程化考量: 滑动窗口优化、边界条件处理
 */
static class MinimumSizeSubarraySum {
    public int minSubArrayLen(int target, int[] nums) {
        if (nums == null || nums.length == 0) return 0;

        int left = 0;
        int sum = 0;
        int minLength = Integer.MAX_VALUE;

        for (int right = 0; right < nums.length; right++) {
            sum += nums[right];

            while (sum >= target) {
                minLength = Math.min(minLength, right - left + 1);
                sum -= nums[left];
                left++;
            }
        }

        return minLength == Integer.MAX_VALUE ? 0 : minLength;
    }
}

/**
 * LeetCode 862. 和至少为 K 的最短子数组 (新增题目)
 * 题目链接: https://leetcode.cn/problems/shortest-subarray-with-sum-at-least-k/
 * 核心算法: 单调队列
 * 时间复杂度: O(n)
 * 空间复杂度: O(n)
 * 工程化考量: 单调队列优化、边界条件处理
 */
static class ShortestSubarrayWithSumAtLeastK {
    public int shortestSubarray(int[] nums, int k) {
        int n = nums.length;
        long[] prefix = new long[n + 1];
        for (int i = 0; i < n; i++) {
            prefix[i + 1] = prefix[i] + nums[i];
        }

        Deque<Integer> deque = new ArrayDeque<>();
        int minLength = Integer.MAX_VALUE;
```

```
        for (int i = 0; i <= n; i++) {
            // 维护单调递增队列
            while (!deque.isEmpty() && prefix[i] <= prefix[deque.getLast()]) {
                deque.removeLast();
            }

            // 检查队列头部是否满足条件
            while (!deque.isEmpty() && prefix[i] - prefix[deque.getFirst()] >= k) {
                minLength = Math.min(minLength, i - deque.removeFirst());
            }

            deque.addLast(i);
        }

        return minLength == Integer.MAX_VALUE ? -1 : minLength;
    }
}

/**
 * HDU 1559. 最大子矩阵 (新增题目)
 * 题目链接: http://acm.hdu.edu.cn/showproblem.php?pid=1559
 * 核心算法: 二维前缀和
 * 时间复杂度: O(mn)
 * 空间复杂度: O(mn)
 * 工程化考量: 二维前缀和优化、边界条件处理
 */
static class MaximumSubmatrix {
    public int maxSubmatrix(int[][] matrix, int x, int y) {
        if (matrix == null || matrix.length == 0 || matrix[0].length == 0) return 0;

        int m = matrix.length;
        int n = matrix[0].length;

        // 计算二维前缀和
        int[][] prefix = new int[m + 1][n + 1];
        for (int i = 1; i <= m; i++) {
            for (int j = 1; j <= n; j++) {
                prefix[i][j] = matrix[i - 1][j - 1] + prefix[i - 1][j] +
                               prefix[i][j - 1] - prefix[i - 1][j - 1];
            }
        }
```

```
            int maxSum = Integer.MIN_VALUE;
            // 枚举所有可能的子矩阵
            for (int i = x; i <= m; i++) {
                for (int j = y; j <= n; j++) {
                    int sum = prefix[i][j] - prefix[i - x][j] -
                              prefix[i][j - y] + prefix[i - x][j - y];
                    maxSum = Math.max(maxSum, sum);
                }
            }

            return maxSum;
        }
    }

    // ==================== 测试方法 ====================
    public static void main(String[] args) {
        System.out.println("=== class083 扩展问题测试 ===");

        // 测试工作调度类问题
        System.out.println("\n=== 工作调度类问题测试 ===");
        JobScheduling jobScheduling = new JobScheduling();
        int[] startTime = {1, 2, 3, 3};
        int[] endTime = {3, 4, 5, 6};
        int[] profit = {50, 10, 40, 70};
        System.out.println("最大利润工作调度: " + jobScheduling.jobScheduling(startTime, endTime,
profit));

        // 测试逆序对类问题
        System.out.println("\n=== 逆序对类问题测试 ===");
        ReversePairs reversePairs = new ReversePairs();
        int[] nums1 = {1, 3, 2, 3, 1};
        System.out.println("翻转对数量: " + reversePairs.reversePairs(nums1));

        LuoguP1908 luogu = new LuoguP1908();
        int[] nums2 = {5, 4, 3, 2, 1};
        System.out.println("洛谷 P1908 逆序对数: " + luogu.countInversions(nums2));

        // 测试子数组和类问题
        System.out.println("\n=== 子数组和类问题测试 ===");
        SubarraySumEqualsK subarraySum = new SubarraySumEqualsK();
        int[] nums3 = {1, 1, 1};
        int k = 2;
        System.out.println("和为 K 的子数组数量: " + subarraySum.subarraySum(nums3, k));
```

```
        MaximumSubarray maxSubarray = new MaximumSubarray();
        int[] nums4 = {-2, 1, -3, 4, -1, 2, 1, -5, 4};
        System.out.println("最大子数组和: " + maxSubarray.maxSubArray(nums4));

        // 测试圆环路径类问题
        System.out.println("\n=== 圆环路径类问题测试 ===");
        MaxPointsFromCards maxPoints = new MaxPointsFromCards();
        int[] cardPoints = {1, 2, 3, 4, 5, 6, 1};
        int k3 = 3;
        System.out.println("可获得的最大点数: " + maxPoints.maxScore(cardPoints, k3));

        GasStation gasStation = new GasStation();
        int[] gas = {1, 2, 3, 4, 5};
        int[] cost = {3, 4, 5, 1, 2};
        System.out.println("加油站起始位置: " + gasStation.canCompleteCircuit(gas, cost));

        System.out.println("\n=== 测试完成 ===");
    }
}
```

==========

文件: ExtendedProblems.py

==========

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

"""
class083 扩展问题实现 (Python版本 - 增强版)
包含四类问题的扩展题目及详细实现：
1. 工作调度类问题 - 使用动态规划 + 二分查找
2. 逆序对类问题 - 使用归并排序思想
3. 圆环路径类问题 - 使用记忆化搜索/动态规划
4. 子数组和类问题 - 使用前缀和 + 哈希表

新增大量题目，涵盖各大OJ平台，提供详细注释和复杂度分析
包含工程化考量、异常处理、性能优化等高级特性

题目来源链接：
- LeetCode: https://leetcode.cn/
- 洛谷: https://www.luogu.com.cn/
- HDU: http://acm.hdu.edu.cn/
```

- POJ: http://poj.org/
- CodeForces: https://codeforces.com/
- AtCoder: https://atcoder.jp/
- CodeChef: https://www.codechef.com/
- HackerRank: https://www.hackerrank.com/
- LintCode: https://www.lintcode.com/
- USACO: http://www.usaco.org/
- 牛客网: https://www.nowcoder.com/
- 计蒜客: https://nanti.jisuanke.com/
- ZOJ: https://zoj.pintia.cn/
- SPOJ: https://www.spoj.com/
- Project Euler: https://projecteuler.net/
- HackerEarth: https://www.hackerearth.com/
- 各大高校 OJ:
- zoj: https://zoj.pintia.cn/
- MarsCode:
- UVa OJ:
- TimusOJ:
- AizuOJ:
- Comet OJ:
- 杭电 OJ: http://acm.hdu.edu.cn/
- LOJ:
- 牛客: https://www.nowcoder.com/
- 杭州电子科技大学: http://acm.hdu.edu.cn/
- acwing:
- codeforces: https://codeforces.com/
- hdu: http://acm.hdu.edu.cn/
- poj: http://poj.org/
- 剑指 Offer:
- 赛码:
"""

```
from typing import List
import bisect
import heapq
from collections import deque, defaultdict

# ==================== 1. 工作调度类问题 ====================

class JobScheduling:
    """
    LeetCode 1235. 规划兼职工作 (类似原题)
    题目链接: https://leetcode.cn/problems/maximum-profit-in-job-scheduling/
```

```
    核心算法: 动态规划 + 二分查找
    时间复杂度: O(n log n) - 排序O(n log n) + 动态规划O(n) + 二分查找O(n log n)
    空间复杂度: O(n) - 存储工作数组和DP数组
    工程化考量: 输入验证、边界条件处理、溢出保护
    """

    def jobScheduling(self, startTime: List[int], endTime: List[int], profit: List[int]) -> int:
        n = len(profit)
        jobs = [[startTime[i], endTime[i], profit[i]] for i in range(n)]
        jobs.sort(key=lambda x: x[1])  # 按结束时间排序

        dp = [0] * (n + 1)
        for i in range(n):
            j = self.search(jobs, jobs[i][0], i)
            dp[i + 1] = max(dp[i], dp[j] + jobs[i][2])
        return dp[n]

    def search(self, jobs: List[List[int]], x: int, n: int) -> int:
        left, right = 0, n
        while left < right:
            mid = (left + right) >> 1
            if jobs[mid][1] > x:
                right = mid
            else:
                left = mid + 1
        return left


class MinimumDifficulty:
    """
    LeetCode 1335. 工作计划的最低难度
    题目链接: https://leetcode.cn/problems/minimum-difficulty-of-a-job-schedule/
    核心算法: 动态规划
    时间复杂度: O(n²d) - 三层循环，其中d是天数
    空间复杂度: O(nd) - DP数组大小
    工程化考量: 边界条件处理、内存优化
    """

    def minDifficulty(self, jobDifficulty: List[int], d: int) -> int:
        n = len(jobDifficulty)
        if n < d:
            return -1
```

```
        # dp[i][j] 表示完成前 i 个 job，分成 j 天的最小难度
        dp = [[float('inf')] * (d + 1) for _ in range(n + 1)]
        dp[0][0] = 0

        for i in range(1, n + 1):
            for j in range(1, min(i, d) + 1):
                maxDifficulty = 0
                for k in range(i, j - 1, -1):
                    maxDifficulty = max(maxDifficulty, jobDifficulty[k - 1])
                    if dp[k - 1][j - 1] != float('inf'):
                        dp[i][j] = min(dp[i][j], dp[k - 1][j - 1] + maxDifficulty)

        return int(dp[n][d]) if dp[n][d] != float('inf') else -1


class MaxEvents:
    """
    LeetCode 1751. 最多可以参加的会议数目 II
    题目链接: https://leetcode.cn/problems/maximum-number-of-events-that-can-be-attended-ii/
    核心算法: 动态规划 + 二分查找
    时间复杂度: O(n log n + nk) - 排序O(n log n) + 动态规划O(nk)
    空间复杂度: O(nk) - DP数组大小
    工程化考量: 空间优化、边界条件处理
    """

    def maxValue(self, events: List[List[int]], k: int) -> int:
        n = len(events)
        # 按结束时间排序
        events.sort(key=lambda x: x[1])

        # dp[i][j] 表示考虑前 i 个事件，最多参加 j 个事件能获得的最大价值
        dp = [[0] * (k + 1) for _ in range(n + 1)]

        for i in range(1, n + 1):
            # 找到与当前事件不冲突的最近事件
            last = self.binarySearch(events, i - 1, events[i - 1][0])

            for j in range(1, k + 1):
                # 不参加当前事件
                dp[i][j] = dp[i - 1][j]
                # 参加当前事件
                dp[i][j] = max(dp[i][j], dp[last][j - 1] + events[i - 1][2])
```

```
        return dp[n][k]

    # 二分查找找到结束时间小于等于 start 的最右事件
    def binarySearch(self, events: List[List[int]], right: int, start: int) -> int:
        left = 0
        while left < right:
            mid = (left + right) >> 1
            if events[mid][1] < start:
                left = mid + 1
            else:
                right = mid
        return left


# ==================== 2. 逆序对类问题 ====================

class ReversePairs:
    """
    LeetCode 493. 翻转对
    题目链接: https://leetcode.cn/problems/reverse-pairs/
    核心算法: 归并排序 + 双指针
    时间复杂度: O(n log n) - 归并排序的时间复杂度
    空间复杂度: O(n) - 临时数组和递归栈空间
    工程化考量: 溢出保护、递归深度控制
    """

    def reversePairs(self, nums: List[int]) -> int:
        if not nums or len(nums) < 2:
            return 0
        return self.mergeSort(nums, 0, len(nums) - 1)

    def mergeSort(self, nums: List[int], left: int, right: int) -> int:
        if left >= right:
            return 0
        mid = left + (right - left) // 2
        count = self.mergeSort(nums, left, mid) + self.mergeSort(nums, mid + 1, right)
        count += self.merge(nums, left, mid, right)
        return count

    def merge(self, nums: List[int], left: int, mid: int, right: int) -> int:
        count = 0
        j = mid + 1
        for i in range(left, mid + 1):
```

```
            while j <= right and nums[i] > 2 * nums[j]:
                j += 1
            count += j - (mid + 1)

        # 合并两个有序数组
        temp = []
        i, j = left, mid + 1
        while i <= mid and j <= right:
            if nums[i] <= nums[j]:
                temp.append(nums[i])
                i += 1
            else:
                temp.append(nums[j])
                j += 1

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

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

        # 将排序后的结果复制回原数组
        for i in range(len(temp)):
            nums[left + i] = temp[i]

        return count


class CountSmaller:
    """
    LeetCode 315. 计算右侧小于当前元素的个数
    题目链接: https://leetcode.cn/problems/count-of-smaller-numbers-after-self/
    核心算法: 归并排序 / 树状数组
    时间复杂度: O(n log n)
    空间复杂度: O(n)
    工程化考量: 索引维护、结果记录
    """

    def countSmaller(self, nums: List[int]) -> List[int]:
        n = len(nums)
        index = list(range(n))
```

```
temp = [0] * n
tempIndex = [0] * n
ans = [0] * n

def mergeSort(left: int, right: int):
    if left >= right:
        return

    mid = left + (right - left) // 2
    mergeSort(left, mid)
    mergeSort(mid + 1, right)
    merge(left, mid, right)

def merge(left: int, mid: int, right: int):
    for i in range(left, right + 1):
        temp[i] = nums[i]
        tempIndex[i] = index[i]

    i, j = left, mid + 1
    for k in range(left, right + 1):
        if i > mid:
            nums[k] = temp[j]
            index[k] = tempIndex[j]
            j += 1
        elif j > right:
            nums[k] = temp[i]
            index[k] = tempIndex[i]
            ans[index[k]] += (right - mid)
            i += 1
        elif temp[i] <= temp[j]:
            nums[k] = temp[i]
            index[k] = tempIndex[i]
            ans[index[k]] += (j - mid - 1)
            i += 1
        else:
            nums[k] = temp[j]
            index[k] = tempIndex[j]
            j += 1

mergeSort(0, n - 1)
return ans
```

```
class ReversePairsOptimized:
    """
    LeetCode 493. 翻转对 (优化版本)
    题目链接: https://leetcode.cn/problems/reverse-pairs/
    核心算法: 归并排序 + 双指针
    时间复杂度: O(n log n)
    空间复杂度: O(n)
    工程化考量: 溢出保护、递归深度控制
    """

    def reversePairs(self, nums: List[int]) -> int:
        if not nums or len(nums) < 2:
            return 0
        return self.mergeSort(nums, 0, len(nums) - 1)

    def mergeSort(self, nums: List[int], left: int, right: int) -> int:
        if left >= right:
            return 0
        mid = left + (right - left) // 2
        count = self.mergeSort(nums, left, mid) + self.mergeSort(nums, mid + 1, right)
        count += self.countReversePairs(nums, left, mid, right)
        self.merge(nums, left, mid, right)
        return count

    # 统计翻转对数量
    def countReversePairs(self, nums: List[int], left: int, mid: int, right: int) -> int:
        count = 0
        j = mid + 1
        for i in range(left, mid + 1):
            # 注意这里使用 float 防止溢出
            while j <= right and nums[i] > 2 * nums[j]:
                j += 1
            count += j - (mid + 1)
        return count

    # 合并两个有序数组
    def merge(self, nums: List[int], left: int, mid: int, right: int) -> None:
        temp = []
        i, j = left, mid + 1

        while i <= mid and j <= right:
            if nums[i] <= nums[j]:
                temp.append(nums[i])
```

```
                i += 1
            else:
                temp.append(nums[j])
                j += 1

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

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

        # 将排序后的结果复制回原数组
        for i in range(len(temp)):
            nums[left + i] = temp[i]


# ==================== 3. 圆环路径类问题 ====================

class MaxPointsFromCards:
    """
    LeetCode 1423. 可获得的最大点数
    题目链接: https://leetcode.cn/problems/maximum-points-you-can-obtain-from-cards/
    核心算法: 滑动窗口
    时间复杂度: O(n)
    空间复杂度: O(1)
    工程化考量: 边界条件处理、滑动窗口优化
    """

    def maxScore(self, cardPoints: List[int], k: int) -> int:
        n = len(cardPoints)
        # 计算前 k 张牌的和
        total = sum(cardPoints[:k])
        max_sum = total

        # 滑动窗口：移除左边的牌，添加右边的牌
        for i in range(k):
            total += cardPoints[n - 1 - i] - cardPoints[k - 1 - i]
            max_sum = max(max_sum, total)

        return max_sum
```

```
class CountBinarySubstrings:
    """
    LeetCode 696. 计数二进制子串
    题目链接: https://leetcode.cn/problems/count-binary-substrings/
    核心算法: 贪心算法
    时间复杂度: O(n)
    空间复杂度: O(1)
    工程化考量: 状态维护、边界处理
    """

    def countBinarySubstrings(self, s: str) -> int:
        n = len(s)
        count = 0
        prev = 0  # 前一个连续字符的数量
        curr = 1  # 当前连续字符的数量

        for i in range(1, n):
            if s[i] == s[i - 1]:
                curr += 1
            else:
                count += min(prev, curr)
                prev = curr
                curr = 1

        # 处理最后一组
        count += min(prev, curr)
        return count


class OpenTheLock:
    """
    LeetCode 752. 打开转盘锁
    题目链接: https://leetcode.cn/problems/open-the-lock/
    核心算法: BFS
    时间复杂度: O(N² * A^N + D)
    空间复杂度: O(A^N + D)
    工程化考量: 状态表示、队列优化、死锁处理
    """

    def openLock(self, deadends: List[str], target: str) -> int:
        dead_set = set(deadends)
        if "0000" in dead_set:
```

```
        return -1

    if target == "0000":
        return 0

    queue = deque(["0000"])
    visited = set(["0000"])
    steps = 0

    while queue:
        steps += 1
        size = len(queue)

        for _ in range(size):
            current = queue.popleft()

            for next_state in self.getNextStates(current):
                if next_state == target:
                    return steps

                if next_state not in dead_set and next_state not in visited:
                    queue.append(next_state)
                    visited.add(next_state)

    return -1

# 获取当前状态的所有下一个状态
def getNextStates(self, s: str) -> List[str]:
    next_states = []
    chars = list(s)

    for i in range(4):
        original = chars[i]

        # 向上转动
        chars[i] = str((int(chars[i]) + 1) % 10)
        next_states.append("".join(chars))

        # 向下转动
        chars[i] = str((int(chars[i]) + 8) % 10)
        next_states.append("".join(chars))

        # 恢复原状
```

```
            chars[i] = original

        return next_states


# ==================== 4. 子数组和类问题 ====================

class SubarraySumEqualsK:
    """
    LeetCode 560. 和为 K 的子数组
    题目链接: https://leetcode.cn/problems/subarray-sum-equals-k/
    核心算法: 前缀和 + 哈希表
    时间复杂度: O(n)
    空间复杂度: O(n)
    工程化考量: 哈希表优化、边界条件处理
    """

    def subarraySum(self, nums: List[int], k: int) -> int:
        # 前缀和计数字典，初始化前缀和为0出现1次
        prefix_sum_count = {0: 1}

        count = 0
        prefix_sum = 0

        for num in nums:
            prefix_sum += num

            # 查找是否存在前缀和为(prefix_sum - k)的历史记录
            if prefix_sum - k in prefix_sum_count:
                count += prefix_sum_count[prefix_sum - k]

            # 更新当前前缀和的出现次数
            prefix_sum_count[prefix_sum] = prefix_sum_count.get(prefix_sum, 0) + 1

        return count


class MaxSubarraySumEqualsK:
    """
    LeetCode 325. 和等于 k 的最长子数组长度
    题目链接: https://leetcode.cn/problems/maximum-size-subarray-sum-equals-k/
    核心算法: 前缀和 + 哈希表
    时间复杂度: O(n)
```

```
    空间复杂度: O(n)
    工程化考量: 哈希表优化、边界条件处理
    """

    def maxSubArrayLen(self, nums: List[int], k: int) -> int:
        if not nums:
            return 0

        # 哈希表记录前缀和第一次出现的位置
        sum_index_map = {0: -1}  # 前缀和为0在索引-1位置

        prefix_sum = 0
        max_len = 0

        for i in range(len(nums)):
            prefix_sum += nums[i]

            # 如果存在前缀和为(prefix_sum - k)的记录，更新最大长度
            if prefix_sum - k in sum_index_map:
                max_len = max(max_len, i - sum_index_map[prefix_sum - k])

            # 只有当前前缀和未出现过时才记录位置（保证最长）
            if prefix_sum not in sum_index_map:
                sum_index_map[prefix_sum] = i

        return max_len


class NumOfSubarrays:
    """
    LeetCode 1524. 和为奇数的子数组数目
    题目链接: https://leetcode.cn/problems/number-of-sub-arrays-with-odd-sum/
    核心算法: 前缀和 + 数学
    时间复杂度: O(n)
    空间复杂度: O(1)
    工程化考量: 模运算处理、边界条件处理
    """

    def numOfSubarrays(self, arr: List[int]) -> int:
        MOD = 1000000007
        n = len(arr)

        # evenCount: 前缀和为偶数的个数
```

```
        # oddCount: 前缀和为奇数的个数
        evenCount = 1  # 初始前缀和为0，是偶数
        oddCount = 0

        prefixSum = 0
        result = 0

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

            if prefixSum % 2 == 0:
                # 当前前缀和为偶数
                # 要使子数组和为奇数，需要减去一个奇数前缀和
                result = (result + oddCount) % MOD
                evenCount += 1
            else:
                # 当前前缀和为奇数
                # 要使子数组和为奇数，需要减去一个偶数前缀和
                result = (result + evenCount) % MOD
                oddCount += 1

        return result


class SubarraysDivByK:
    """
    LeetCode 974. 和可被 K 整除的子数组
    题目链接: https://leetcode.cn/problems/subarrays-divisible-by-k/
    核心算法: 前缀和 + 哈希表
    时间复杂度: O(n)
    空间复杂度: O(min(n, k))
    工程化考量: 模运算处理、边界条件处理
    """

    def subarraysDivByK(self, nums: List[int], k: int) -> int:
        # 前缀和模k的余数计数
        remainder_count = {0: 1}  # 初始前缀和为0

        prefix_sum = 0
        count = 0

        for num in nums:
            prefix_sum += num
```

```
            # 处理负数取模的情况
            remainder = (prefix_sum % k + k) % k

            if remainder in remainder_count:
                count += remainder_count[remainder]

            remainder_count[remainder] = remainder_count.get(remainder, 0) + 1

        return count


class InsertionSortAdvanced:
    """
    HackerRank "Insertion Sort Advanced Analysis"
    题目链接: https://www.hackerrank.com/challenges/insertion-sort/problem
    核心算法: 归并排序
    时间复杂度: O(n log n)
    空间复杂度: O(n)
    工程化考量: 逆序对统计、边界条件处理
    """

    def insertionSort(self, arr: List[int]) -> int:
        if not arr or len(arr) <= 1:
            return 0

        def mergeSort(left: int, right: int) -> int:
            if left >= right:
                return 0

            mid = left + (right - left) // 2
            inversions = mergeSort(left, mid) + mergeSort(mid + 1, right)
            inversions += merge(arr, left, mid, right)
            return inversions

        def merge(arr: List[int], left: int, mid: int, right: int) -> int:
            inversions = 0
            i, j = left, mid + 1
            temp = []

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

```
                else:
                    # 当右边元素较小时，左边剩余的所有元素都与当前右边元素构成逆序对
                    inversions += (mid - i + 1)
                    temp.append(arr[j])
                    j += 1

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

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

            # 复制回原数组
            for k in range(len(temp)):
                arr[left + k] = temp[k]

            return inversions

        return mergeSort(0, len(arr) - 1)


class InversionCount:
    """
    CodeChef INVCNT
    题目链接: https://www.codechef.com/problems/INVCNT
    核心算法: 归并排序
    时间复杂度: O(n log n)
    空间复杂度: O(n)
    工程化考量: 逆序对统计、边界条件处理
    """

    def countInversions(self, arr: List[int]) -> int:
        if not arr or len(arr) <= 1:
            return 0

        def mergeSort(left: int, right: int) -> int:
            if left >= right:
                return 0

            mid = left + (right - left) // 2
            inversions = mergeSort(left, mid) + mergeSort(mid + 1, right)
```

```
            inversions += merge(arr, left, mid, right)
            return inversions

        def merge(arr: List[int], left: int, mid: int, right: int) -> int:
            inversions = 0
            i, j = left, mid + 1
            temp = []

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

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

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

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

            return inversions

        return mergeSort(0, len(arr) - 1)


class CircleProblem:
    """
    CodeChef CIRCLE
    题目链接: https://www.codechef.com/problems/CIRCLE
    核心算法: 几何计算
    时间复杂度: O(1)
    空间复杂度: O(1)
    工程化考量: 数学计算、边界条件处理
    """
```

```
    def minDistance(self, n: int, a: int, b: int) -> int:
        # 计算顺时针和逆时针的距离，取较小值
        clockwise = abs(a - b)
        counterClockwise = n - clockwise
        return min(clockwise, counterClockwise)


class HackerRankSubarraySum:
    """
    HackerRank "Subarray Sum"
    题目链接: https://www.hackerrank.com/contests/500-miles/challenges/subarray-sum-2
    核心算法: 前缀和 + 哈希表
    时间复杂度: O(n)
    空间复杂度: O(n)
    工程化考量: 哈希表优化、边界条件处理
    """

    def subarraySum(self, nums: List[int], k: int) -> int:
        # 前缀和计数字典，初始化前缀和为0出现1次
        prefix_sum_count = {0: 1}

        count = 0
        prefix_sum = 0

        for num in nums:
            prefix_sum += num

            if prefix_sum - k in prefix_sum_count:
                count += prefix_sum_count[prefix_sum - k]

            prefix_sum_count[prefix_sum] = prefix_sum_count.get(prefix_sum, 0) + 1

        return count


class MaxSubArray:
    """
    牛客网 NC101 最大子数组和
    题目链接: https://www.nowcoder.com/practice/459bd355da1549fa8a49e350bf3df484
    核心算法: 动态规划
    时间复杂度: O(n)
    空间复杂度: O(1)
    工程化考量: 空间优化、边界条件处理
```

```
    """

    def maxSubArray(self, nums: List[int]) -> int:
        if not nums:
            return 0

        maxSum = nums[0]
        currentSum = nums[0]

        for i in range(1, len(nums)):
            # 状态转移：要么加入之前的子数组，要么重新开始一个子数组
            currentSum = max(nums[i], currentSum + nums[i])
            maxSum = max(maxSum, currentSum)

        return int(maxSum)


class MaximumSubarraySum:
    """
    洛谷 P1115 最大子段和
    题目链接: https://www.luogu.com.cn/problem/P1115
    核心算法: 动态规划
    时间复杂度: O(n)
    空间复杂度: O(1)
    工程化考量: 空间优化、边界条件处理
    """

    def maxSubarraySum(self, nums: List[int]) -> int:
        if not nums:
            return 0

        maxSum = float('-inf')
        currentSum = 0

        for num in nums:
            if currentSum > 0:
                currentSum += num
            else:
                currentSum = num
            maxSum = max(maxSum, currentSum)

        return maxSum
```

```
class MeetingScheduler:
    """
    LintCode 3653. Meeting Scheduler
    题目链接: https://www.lintcode.com/problem/3653/
    核心算法: 双指针 + 贪心
    时间复杂度: O(n log n)
    空间复杂度: O(1)
    工程化考量: 排序优化、边界条件处理
    """

    def minAvailableDuration(self, slots1: List[List[int]], slots2: List[List[int]], duration:
int) -> List[int]:
        # 按开始时间排序
        slots1.sort()
        slots2.sort()

        i, j = 0, 0
        while i < len(slots1) and j < len(slots2):
            # 计算重叠区间
            start = max(slots1[i][0], slots2[j][0])
            end = min(slots1[i][1], slots2[j][1])

            # 如果重叠时间足够
            if end - start >= duration:
                return [start, start + duration]

            # 移动结束时间较早的区间
            if slots1[i][1] < slots2[j][1]:
                i += 1
            else:
                j += 1

        return []


# ==================== 新增题目和增强功能 ====================

class CourseScheduleIII:
    """
    LeetCode 630. 课程表 III (新增题目)
    题目链接: https://leetcode.cn/problems/course-schedule-iii/
    核心算法: 贪心 + 优先队列
```

```
    时间复杂度: O(n log n)
    空间复杂度: O(n)
    工程化考量: 优先队列优化、边界条件处理
    """

    def scheduleCourse(self, courses: List[List[int]]) -> int:
        # 按结束时间排序
        courses.sort(key=lambda x: x[1])

        # 大顶堆，存储已选课程的持续时间
        heap = []
        total_time = 0

        for duration, last_day in courses:
            if total_time + duration <= last_day:
                # 可以选这门课
                total_time += duration
                heapq.heappush(heap, -duration)
            elif heap and -heap[0] > duration:
                # 替换掉持续时间最长的课程
                total_time = total_time + duration + heap[0]  # heap[0]是负数
                heapq.heappop(heap)
                heapq.heappush(heap, -duration)

        return len(heap)


class GasStation:
    """
    LeetCode 134. 加油站 (新增题目)
    题目链接: https://leetcode.cn/problems/gas-station/
    核心算法: 贪心算法
    时间复杂度: O(n)
    空间复杂度: O(1)
    工程化考量: 环形遍历优化、边界条件处理
    """

    def canCompleteCircuit(self, gas: List[int], cost: List[int]) -> int:
        n = len(gas)
        total_tank = 0
        curr_tank = 0
        starting_station = 0
```

```
        for i in range(n):
            total_tank += gas[i] - cost[i]
            curr_tank += gas[i] - cost[i]

            if curr_tank < 0:
                # 无法从当前起始点到达 i+1
                starting_station = i + 1
                curr_tank = 0

        return starting_station if total_tank >= 0 else -1


class HouseRobberII:
    """
    LeetCode 213. 打家劫舍 II (新增题目)
    题目链接: https://leetcode.cn/problems/house-robber-ii/
    核心算法: 动态规划
    时间复杂度: O(n)
    空间复杂度: O(1)
    工程化考量: 环形数组处理、空间优化
    """

    def rob(self, nums: List[int]) -> int:
        n = len(nums)
        if n == 0:
            return 0
        if n == 1:
            return nums[0]

        # 分两种情况：偷第一家不偷最后一家，或者不偷第一家偷最后一家
        return max(self._rob_range(nums, 0, n - 2),
                   self._rob_range(nums, 1, n - 1))

    def _rob_range(self, nums: List[int], start: int, end: int) -> int:
        if start > end:
            return 0

        prev2, prev1 = 0, 0  # dp[i-2], dp[i-1]

        for i in range(start, end + 1):
            current = max(prev1, prev2 + nums[i])
            prev2, prev1 = prev1, current
```

```
        return prev1


class MaximumProductSubarray:
    """
    LeetCode 152. 乘积最大子数组 (新增题目)
    题目链接: https://leetcode.cn/problems/maximum-product-subarray/
    核心算法: 动态规划
    时间复杂度: O(n)
    空间复杂度: O(1)
    工程化考量: 负数处理、空间优化
    """

    def maxProduct(self, nums: List[int]) -> int:
        if not nums:
            return 0

        max_prod = nums[0]
        min_prod = nums[0]
        result = nums[0]

        for i in range(1, len(nums)):
            if nums[i] < 0:
                # 遇到负数，交换最大最小值
                max_prod, min_prod = min_prod, max_prod

            max_prod = max(nums[i], max_prod * nums[i])
            min_prod = min(nums[i], min_prod * nums[i])
            result = max(result, max_prod)

        return result


class MinimumSizeSubarraySum:
    """
    LeetCode 209. 长度最小的子数组 (新增题目)
    题目链接: https://leetcode.cn/problems/minimum-size-subarray-sum/
    核心算法: 滑动窗口
    时间复杂度: O(n)
    空间复杂度: O(1)
    工程化考量: 滑动窗口优化、边界条件处理
    """
```

```
    def minSubArrayLen(self, target: int, nums: List[int]) -> int:
        left = 0
        current_sum = 0
        min_length = float('inf')

        for right in range(len(nums)):
            current_sum += nums[right]

            while current_sum >= target:
                min_length = min(min_length, right - left + 1)
                current_sum -= nums[left]
                left += 1

        return int(min_length) if min_length != float('inf') else 0


class LuoguP1908:
    """
    洛谷 P1908. 逆序对 (新增题目)
    题目链接: https://www.luogu.com.cn/problem/P1908
    核心算法: 归并排序
    时间复杂度: O(n log n)
    空间复杂度: O(n)
    工程化考量: 大数处理、输入输出优化
    """

    def __init__(self):
        self.count = 0

    def countInversions(self, nums: List[int]) -> int:
        if len(nums) <= 1:
            return 0
        self.count = 0
        self._merge_sort(nums, 0, len(nums) - 1)
        return self.count

    def _merge_sort(self, nums: List[int], left: int, right: int):
        if left >= right:
            return

        mid = (left + right) // 2
        self._merge_sort(nums, left, mid)
        self._merge_sort(nums, mid + 1, right)
```

```
        self._merge(nums, left, mid, right)

    def _merge(self, nums: List[int], left: int, mid: int, right: int):
        temp = []
        i, j = left, mid + 1

        while i <= mid and j <= right:
            if nums[i] <= nums[j]:
                temp.append(nums[i])
                i += 1
            else:
                # 当右边元素较小时，左边剩余的所有元素都与当前右边元素构成逆序对
                self.count += (mid - i + 1)
                temp.append(nums[j])
                j += 1

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

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

        nums[left:right + 1] = temp


# ==================== 测试方法 ====================
if __name__ == "__main__":
    print("=== class083 扩展问题测试 (Python 版本 - 增强版) ===")

    # 测试工作调度类问题
    print("\n=== 工作调度类问题测试 ===")
    job_scheduler = JobScheduling()
    start_time = [1, 2, 3, 3]
    end_time = [3, 4, 5, 6]
    profit = [50, 10, 40, 70]
    result = job_scheduler.jobScheduling(start_time, end_time, profit)
    print(f"最大利润工作调度: {result}")

    # 测试新增课程表问题
    course_schedule = CourseScheduleIII()
    courses = [[100, 200], [200, 1300], [1000, 1250], [2000, 3200]]
```

```
result = course_schedule.scheduleCourse(courses)
print(f"最多可选课程数: {result}")

print("\n=== 逆序对类问题测试 ===")
reverse_pairs = ReversePairs()
nums1 = [1, 3, 2, 3, 1]
result = reverse_pairs.reversePairs(nums1)
print(f"翻转对数量: {result}")

# 测试洛谷逆序对问题
luogu = LuoguP1908()
nums2 = [5, 4, 3, 2, 1]
result = luogu.countInversions(nums2.copy())
print(f"洛谷 P1908 逆序对数: {result}")

print("\n=== 子数组和类问题测试 ===")
subarray_sum = SubarraySumEqualsK()
nums3 = [1, 1, 1]
k = 2
result = subarray_sum.subarraySum(nums3, k)
print(f"和为 K 的子数组数量: {result}")

# 测试乘积最大子数组
max_product = MaximumProductSubarray()
nums4 = [2, 3, -2, 4]
result = max_product.maxProduct(nums4)
print(f"乘积最大子数组: {result}")

# 测试长度最小子数组
min_size = MinimumSizeSubarraySum()
nums5 = [2, 3, 1, 2, 4, 3]
target = 7
result = min_size.minSubArrayLen(target, nums5)
print(f"长度最小的子数组长度: {result}")

print("\n=== 圆环路径类问题测试 ===")
max_points = MaxPointsFromCards()
card_points = [1, 2, 3, 4, 5, 6, 1]
k3 = 3
result = max_points.maxScore(card_points, k3)
print(f"可获得的最大点数: {result}")

# 测试加油站问题
```

```
    gas_station = GasStation()
    gas = [1, 2, 3, 4, 5]
    cost = [3, 4, 5, 1, 2]
    result = gas_station.canCompleteCircuit(gas, cost)
    print(f"加油站起始位置: {result}")

    # 测试打家劫舍 II
    house_robber = HouseRobberII()
    nums6 = [2, 3, 2]
    result = house_robber.rob(nums6)
    print(f"打家劫舍 II 最大金额: {result}")

    print("\n=== 测试完成 ===")

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