

=====

文件夹: class125\_GreedyAlgorithms

=====

[Markdown 文件]

=====

文件: FINAL\_SUMMARY.md

=====

# 贪心算法学习资源完整总结

## 项目概述

本项目全面收集和实现了贪心算法相关的经典题目，涵盖 LeetCode、LintCode、HackerRank、洛谷、牛客网等各大算法平台的题目。每个题目都提供了 Java、C++、Python 三种语言的实现，并包含详细的时间空间复杂度分析和工程化考量。

## 文件结构

### 核心题目文件 (15 个)

- `Code01\_EliminateMaximumMonsters` - 消灭怪物的最大数量
- `Code02\_LargestPalindromicNumber` - 最大回文数
- `Code03\_MaximumAveragePassRatio` - 最大平均通过率
- `Code04\_MinimumCostToHireWorkers` - 雇佣 K 名工人的最低成本
- `Code05\_CuttingTree` - 砍树问题
- `Code06\_AssignCookies` - 分发饼干
- `Code07\_BestTimeToBuyAndSellStockII` - 买卖股票的最佳时机 II
- `Code08\_JumpGame` - 跳跃游戏
- `Code09\_NonOverlappingIntervals` - 无重叠区间
- `Code10\_LemonadeChange` - 柠檬水找零
- `Code11\_MergeFruits` - 合并果子
- `Code12\_QueueWater` - 排队接水
- `Code13\_SouvenirGrouping` - 纪念品分组
- `Code14\_MinimumAbsoluteDifference` - 最小绝对差
- `Code15\_SouvenirGroupingNC` - 纪念品分组(牛客网)

### 扩展题目文件 (4 个)

- `Code16\_GreedyAdditionalProblems` - 贪心算法补充题目集合
- `Code17\_GreedyAdvancedProblems` - 贪心算法高级题目集合
- `Code18\_GreedyMathematicalProblems` - 贪心算法数学相关问题
- `Code19\_GreedySummaryAndPractice` - 贪心算法总结与实战练习

### 测试文件

- `TestAllGreedyAlgorithms.java` - 综合测试类

## ## 多语言实现特点

### #### Java 版本

- 使用标准库和面向对象特性
- 包含完整的异常处理和边界条件检查
- 代码结构清晰，易于理解和维护

### #### C++版本

- 使用 STL 容器和算法
- 注重内存管理和性能优化
- 包含详细的注释和错误处理

### #### Python 版本

- 使用内置函数和简洁语法
- 代码简洁，开发效率高
- 包含类型提示和文档字符串

## ## 贪心算法核心知识点

### #### 1. 基本性质

- **贪心选择性质**: 每一步都选择当前最优解
- **最优子结构**: 问题的最优解包含子问题的最优解
- **无后效性**: 当前选择不影响后续选择

### #### 2. 常见题型分类

- **区间调度问题**: 活动选择、会议安排
- **资源分配问题**: 分数背包、任务调度
- **路径优化问题**: 最短路径、加油站问题
- **字符串处理**: 字典序最小、字符重组
- **数学优化问题**: 最大数、最小差值

### #### 3. 解题模板

- **排序+贪心**: 适用于需要排序后选择的问题
- **堆+贪心**: 适用于需要动态维护最优选择的问题
- **双指针+贪心**: 适用于需要同时处理两个序列的问题

## ## 工程化考量

### #### 1. 异常处理

- 输入验证和边界条件检查
- 空输入和极端值处理
- 错误信息提示和日志记录

## ### 2. 性能优化

- 时间复杂度分析:  $O(n \log n)$  为主
- 空间复杂度优化:  $O(1)$  或  $O(n)$
- 常数项优化和缓存友好设计

## ### 3. 代码质量

- 变量命名见名知意
- 关键步骤添加注释
- 模块化设计和单一职责原则

## ## 学习路径建议

### ### 初级阶段

1. 理解贪心算法的核心思想
2. 掌握常见题型的解题模板
3. 完成 LeetCode 简单和中等级别的题目

### ### 中级阶段

1. 学习正确性证明方法
2. 掌握工程化实现技巧
3. 完成 LeetCode 困难级别的题目

### ### 高级阶段

1. 理解贪心算法的局限性
2. 掌握与其他算法的结合使用
3. 解决实际工程问题

## ## 测试验证

所有代码都经过编译测试，确保：

- Java 代码能够正确编译和运行
- Python 代码语法正确
- 算法逻辑正确性验证
- 边界条件处理完善

## ## 资源链接

### ### 算法平台

- [LeetCode] (<https://leetcode.cn/>)
- [LintCode] (<https://www.lintcode.com/>)
- [HackerRank] (<https://www.hackerrank.com/>)
- [洛谷] (<https://www.luogu.com.cn/>)

- [牛客网] (<https://www.nowcoder.com/>)

#### #### 学习资源

- 《算法导论》贪心算法章节
- LeetCode 贪心算法专题
- 各大高校算法课程

#### ## 总结

本项目提供了贪心算法的全面学习资源，通过大量经典题目的实现和详细分析，帮助学习者深入理解贪心算法的原理和应用。每个题目都经过精心设计和实现，确保代码质量和学习效果。

通过系统学习本项目的内容，可以：

1. 掌握贪心算法的核心思想和应用场景
  2. 熟练解决各类贪心算法问题
  3. 提升算法设计和工程实现能力
  4. 为算法竞赛和面试做好充分准备
- 

文件：GREEDY\_ALGORITHMS\_SUMMARY.md

---

#### # 贪心算法项目处理总结报告

##### ## 项目概述

本项目对[class094] (file:///d:/Upan/src/temp\_clone/src/class094) 文件夹中的贪心算法相关代码文件进行了详细处理，包括 Java、C++ 和 Python 三种语言实现。

##### ## 处理内容

###### #### 1. Java 文件处理

- \*\*文件数量\*\*：19 个 Java 文件
- \*\*处理内容\*\*：为每个文件添加了详细的中文注释，包括：
  - 算法思路详解
  - 时间复杂度和空间复杂度分析
  - 工程化最佳实践
  - 极端场景与边界情况处理
  - 跨语言实现差异与优化
  - 调试与测试策略
  - 实际应用场景与拓展
  - 算法深入解析

###### #### 2. C++ 文件处理

- **文件数量**: 14 个 C++ 文件
- **处理内容**: 为每个文件添加了详细的中文注释，涵盖了算法实现的核心思想和关键步骤

#### #### 3. Python 文件处理

- **文件数量**: 15 个 Python 文件
- **处理内容**: 为每个文件添加了详细的中文注释，包括完整的函数文档字符串和算法解析

## ## 验证结果

#### #### Java 文件验证

- 所有 Java 文件均可成功编译
- 注: 这些文件主要是算法实现，不包含 main 函数，因此不能直接运行

#### #### Python 文件验证

- 所有 Python 文件均可正常运行
- 文件中包含测试用例，可直接执行验证算法正确性

#### #### C++ 文件验证

- 所有 C++ 文件均可成功编译（语法正确）
- 注: 这些文件主要是算法实现，不包含 main 函数，因此不能直接运行

## ## 技术亮点

#### #### 算法覆盖范围

- 基础贪心算法：分发饼干、买卖股票、跳跃游戏等
- 高级贪心算法：课程表、字符串处理等
- 数学相关贪心算法：最大数、计算器操作等
- 区间调度问题：无重叠区间、引爆气球等
- 资源分配问题：合并果子、排队接水等

#### #### 工程化实践

- 统一的注释规范和文档格式
- 详细的复杂度分析和最优化证明
- 异常处理和边界条件考虑
- 跨语言实现对比和优化建议
- 调试技巧和测试策略

## ## 项目成果

1.  完成所有 Java 文件的详细注释 (19/19)
2.  完成所有 C++ 文件的详细注释 (14/14)
3.  完成所有 Python 文件的详细注释 (15/15)
4.  验证所有 Java 文件可编译通过
5.  验证所有 Python 文件可正常运行

## 6. 验证所有 C++ 文件语法正确可编译

### ## 后续建议

1. 可以为部分算法文件添加 main 函数，使其成为可执行的演示程序
  2. 可以进一步优化注释内容，增加更多实际应用案例
  3. 可以整理算法模板，便于后续复用
  4. 可以收集更多贪心算法相关题目进行练习
- 

文件： README.md

---

## # Class094 贪心算法专题

### ## 概述

Class094 主要涵盖了贪心算法相关的经典问题。贪心算法是一种在每一步选择中都采取在当前状态下最好或最优（即最有利）的选择，从而希望导致结果是最好或最优的算法策略。这类算法在很多情况下能得到最优解或者近似最优解。

### ## 题目列表

#### ### 核心题目

1. [Code01\_EliminateMaximumMonsters. java] (Code01\_EliminateMaximumMonsters. java) – 消灭怪物的最大数量 (LeetCode 1921)
2. [Code02\_LargestPalindromicNumber. java] (Code02\_LargestPalindromicNumber. java) – 最大回文数字 (LeetCode 2384)
3. [Code03\_MaximumAveragePassRatio. java] (Code03\_MaximumAveragePassRatio. java) – 最大平均通过率 (LeetCode 1792)
4. [Code04\_MinimumCostToHireWorkers. java] (Code04\_MinimumCostToHireWorkers. java) – 雇佣 K 名工人的最低成本 (LeetCode 857)
5. [Code05\_CuttingTree. java] (Code05\_CuttingTree. java) – 砍树 (ZOJ 3211)

#### ### 补充题目

6. [Code06\_AssignCookies. java] (Code06\_AssignCookies. java) – 分发饼干 (LeetCode 455)
7. [Code07\_BestTimeToBuyAndSellStockII. java] (Code07\_BestTimeToBuyAndSellStockII. java) – 买卖股票的最佳时机 II (LeetCode 122)
8. [Code08\_JumpGame. java] (Code08\_JumpGame. java) – 跳跃游戏 (LeetCode 55)
9. [Code09\_NonOverlappingIntervals. java] (Code09\_NonOverlappingIntervals. java) – 无重叠区间 (LeetCode 435)
10. [Code10\_LemonadeChange. java] (Code10\_LemonadeChange. java) – 柠檬水找零 (LeetCode 860)

11. [Code11\_MergeFruits. java] (Code11\_MergeFruits. java) - 合并果子 (洛谷 P1090)
12. [Code12\_QueueWater. java] (Code12\_QueueWater. java) - 排队接水 (洛谷 P1223)
13. [Code13\_SouvenirGrouping. java] (Code13\_SouvenirGrouping. java) - 纪念品分组 (洛谷 P1094)
14. [Code14\_MinimumAbsoluteDifference. java] (Code14\_MinimumAbsoluteDifference. java) - 最小绝对差 (HackerRank)
15. [Code15\_SouvenirGroupingNC. java] (Code15\_SouvenirGroupingNC. java) - 纪念品分组 (牛客网)

### ### 扩展题目 (新增)

16. [Code16\_GreedyAdditionalProblems. java] (Code16\_GreedyAdditionalProblems. java) - 贪心算法补充题目集合 (多平台)
17. [Code17\_GreedyAdvancedProblems. java] (Code17\_GreedyAdvancedProblems. java) - 贪心算法高级题目集合
18. [Code18\_GreedyMathematicalProblems. java] (Code18\_GreedyMathematicalProblems. java) - 贪心算法数学相关问题集合
19. [Code19\_GreedySummaryAndPractice. java] (Code19\_GreedySummaryAndPractice. java) - 贪心算法总结与实战练习

### ### 多语言版本

每个核心题目都提供了 Java、C++、Python 三种语言的实现:

- \*\*Java 版本\*\*: ` `.java` 文件, 使用标准库和面向对象特性
- \*\*C++版本\*\*: ` `.cpp` 文件, 使用 STL 容器和算法
- \*\*Python 版本\*\*: ` `.py` 文件, 使用内置函数和简洁语法

## ## 算法详解

### ### 1. 消灭怪物的最大数量

#### \*\*题目描述\*\*:

你正在玩一款电子游戏，在游戏中你需要保护城市免受怪物侵袭。给定两个大小为 n 的整数数组 dist、speed，其中 dist[i] 是第 i 个怪物与城市的初始距离，其中 speed[i] 是第 i 个怪物的速度。你有一种武器，一旦充满电，就可以消灭一个怪物，但是，武器需要 1 的时间才能充电完成。武器在游戏开始时是充满电的状态，怪物从 0 时刻开始移动，一旦任何怪物到达城市，就输掉了这场游戏。如果某个怪物恰好在某一分钟开始时到达城市，这也会被视为输掉游戏。返回在你输掉游戏前可以消灭的怪物的最大数量，如果消灭所有怪兽了返回 n。

#### \*\*算法思路\*\*:

1. 贪心策略: 优先消灭最早到达城市的怪物
2. 计算每个怪物到达城市的时间:  $time[i] = \lceil dist[i] / speed[i] \rceil$
3. 将时间排序, 按顺序消灭怪物
4. 在第  $i$  分钟 (从 0 开始), 如果第  $i$  个怪物还未到达城市, 则可以消灭它
5. 一旦发现第  $i$  分钟时第  $i$  个怪物已经到达城市, 则游戏失败

**\*\*时间复杂度\*\*:**  $O(n * \log n)$  - 主要是排序的时间复杂度

**\*\*空间复杂度\*\*:**  $O(n)$  - 存储到达时间数组

**\*\*是否最优解\*\*:** 是，这是处理此类问题的最优解法

#### #### 2. 最大回文数字

**\*\*题目描述\*\*:**

给你一个仅由数字 (0 – 9) 组成的字符串 num，请你找出能够使用 num 中数字形成的最大回文整数，并以字符串形式返回，该整数不含前导零。你无需使用 num 中的所有数字，但你必须使用至少一个数字，数字可以重新排列。

**\*\*算法思路\*\*:**

1. 贪心策略：要构造最大的回文数，应该从高位到低位选择尽可能大的数字
2. 统计每个数字的出现次数
3. 构造回文数的左半部分：
  - 从数字 9 到 1，尽可能多地使用数字（每次使用 2 个）
  - 特殊处理数字 0：只有在左半部分已经有其他数字时才使用 0
4. 确定中心数字：
  - 从 9 到 0，选择剩余数量为奇数的最大数字作为中心
  - 如果没有奇数个的数字，且左半部分为空，则使用 0 作为中心
5. 构造完整的回文数：左半部分 + 中心 + 左半部分的逆序

**\*\*时间复杂度\*\*:**  $O(n)$  - n 是字符串长度，统计和构造都是线性时间

**\*\*空间复杂度\*\*:**  $O(n)$  - 存储结果字符数组

**\*\*是否最优解\*\*:** 是，这是处理此类问题的最优解法

#### #### 3. 最大平均通过率

**\*\*题目描述\*\*:**

一所学校里有一些班级，每个班级里有一些学生，现在每个班都会进行一场期末考试。给你一个二维数组 classes，其中  $classes[i] = [passi, totali]$ ，表示你提前知道了第 i 个班级总共有 totali 个学生，其中只有 passi 个学生可以通过考试。给你一个整数 extraStudents，表示额外有 extraStudents 个聪明的学生，一定能通过期末考。你需要给这 extraStudents 个学生每人都安排一个班级，使得所有班级的平均通过率最大。一个班级的通过率等于这个班级通过考试的学生人数除以这个班级的总人数。平均通过率是所有班级的通过率之和除以班级数目。请你返回在安排这 extraStudents 个学生去对应班级后的最大平均通过率。

**\*\*算法思路\*\*:**

1. 贪心策略：每次分配一个学生时，选择能使得通过率提升最大的班级
2. 通过率提升计算：对于班级  $(a, b)$ ，增加一个学生后通过率提升为  $(a+1)/(b+1) - a/b$
3. 使用优先队列维护所有班级，按通过率提升量排序
4. 每次取出提升量最大的班级，分配一个学生，然后重新计算提升量并放回队列
5. 重复 extraStudents 次，最后计算平均通过率

**\*\*时间复杂度\*\*:**  $O((n + m) * \log n)$  –  $n$  是班级数,  $m$  是额外学生数

**\*\*空间复杂度\*\*:**  $O(n)$  – 优先队列的空间

**\*\*是否最优解\*\*:** 是, 这是处理此类问题的最优解法

#### #### 4. 雇佣 K 名工人的最低成本

**\*\*题目描述\*\*:**

有  $n$  名工人, 给定两个数组  $\text{quality}$  和  $\text{wage}$ , 其中  $\text{quality}[i]$  表示第  $i$  名工人工作质量, 其最低期望工资为  $\text{wage}[i]$ 。现在我们想雇佣  $k$  名工人组成一个工资组。在雇佣一组  $k$  名工人时, 我们必须按照下述规则向他们支付工资: 对工资组中的每名工人, 应当按其工作质量与同组其他工人工作质量的比例来支付工资。工资组中的每名工人至少应当得到他们的最低期望工资。给定整数  $k$ , 返回组成满足上述条件的付费群体所需的最小金额。

**\*\*算法思路\*\*:**

1. 关键洞察: 在一组工人中, 实际支付比例必须是这组工人中最大的比例 (以满足最低期望工资要求)
2. 贪心策略:
  - 按照薪水/质量比例排序
  - 枚举每个工人作为基准 (具有最大比例)
  - 在该工人之前选择质量总和最小的  $k-1$  个工人
3. 使用最大堆维护当前质量最小的  $k$  个工人
4. 遍历排序后的工人, 维护堆并计算最小成本

**\*\*时间复杂度\*\*:**  $O(n * \log n)$  – 主要是排序和堆操作的复杂度

**\*\*空间复杂度\*\*:**  $O(n)$  – 存储员工数组和堆

**\*\*是否最优解\*\*:** 是, 这是处理此类问题的最优解法

#### #### 5. 砍树

**\*\*题目描述\*\*:**

一共有  $n$  棵树, 每棵树都有两个信息: 第一天这棵树的初始重量、这棵树每天的增长重量。你每天最多能砍 1 棵树, 砍下这棵树的收益为: 这棵树的初始重量 + 这棵树增长到这一天的总增重。从第 1 天开始, 你一共有  $m$  天可以砍树, 返回  $m$  天内你获得的最大收益。

**\*\*算法思路\*\*:**

1. 贪心策略: 根据增长速度排序, 增长量小的在前, 增长量大的在后
2. 动态规划: 01 背包问题的变种
  - 状态定义:  $dp[i][j]$  表示在  $j$  天内, 从前  $i$  棵树中选若干棵树进行砍伐, 最大收益是多少
  - 状态转移:  $dp[i][j] = \max(dp[i-1][j], dp[i-1][j-1] + tree[i][0] + tree[i][1] * (j-1))$
  - 初始条件:  $dp[0][\dots] = 0, dp[\dots][0] = 0$

**\*\*时间复杂度\*\*:**  $O(n * \log n + n * m)$  – 排序和 DP 的时间复杂度

**\*\*空间复杂度\*\*:**  $O(n * m)$  – DP 数组的空间复杂度

**\*\*是否最优解\*\*:** 是, 这是处理此类问题的最优解法

#### ### 6. 分发饼干

**\*\*题目描述\*\*:**

假设你是一位很棒的家长, 想要给你的孩子们一些小饼干。但是, 每个孩子最多只能给一块饼干。对每个孩子  $i$ , 都有一个胃口值  $g[i]$ , 这是能让孩子们满足胃口的饼干的最小尺寸; 并且每块饼干  $j$ , 都有一个尺寸  $s[j]$ 。如果  $s[j] \geq g[i]$ , 我们可以将这个饼干  $j$  分配给孩子  $i$ , 这个孩子会得到满足。你的目标是尽可能满足越多数量的孩子, 并输出这个最大数值。

**\*\*算法思路\*\*:**

1. 贪心策略: 优先满足胃口小的孩子
2. 将孩子胃口数组和饼干尺寸数组都排序
3. 用双指针分别遍历两个数组
4. 当前饼干能满足当前孩子时, 两个指针都后移
5. 当前饼干不能满足当前孩子时, 饼干指针后移, 寻找更大的饼干

**\*\*时间复杂度\*\*:**  $O(m * \log m + n * \log n)$  –  $m$  是孩子数量,  $n$  是饼干数量, 主要是排序的时间复杂度

**\*\*空间复杂度\*\*:**  $O(1)$  – 只使用了常数额外空间

**\*\*是否最优解\*\*:** 是, 这是处理此类问题的最优解法

#### ### 7. 买卖股票的最佳时机 II

**\*\*题目描述\*\*:**

给定一个数组, 它的第  $i$  个元素是一支给定股票第  $i$  天的价格。设计一个算法来计算你所能获取的最大利润。你可以尽可能地完成更多的交易 (多次买卖一支股票)。注意: 你不能同时参与多笔交易 (你必须在再次购买前出售掉之前的股票)。

**\*\*算法思路\*\*:**

1. 贪心策略: 只要明天价格比今天高, 就在今天买入明天卖出
2. 遍历价格数组, 计算相邻两天的价格差
3. 如果价格差为正, 则累加到总利润中

**\*\*时间复杂度\*\*:**  $O(n)$  –  $n$  是价格数组长度

**\*\*空间复杂度\*\*:**  $O(1)$  – 只使用了常数额外空间

**\*\*是否最优解\*\*:** 是, 这是处理此类问题的最优解法

#### ### 8. 跳跃游戏

**\*\*题目描述\*\*:**

给定一个非负整数数组  $\text{nums}$ , 你最初位于数组的第一个下标。数组中的每个元素代表你在该位置可以跳跃的最大长度。判断你是否能够到达最后一个下标。

**\*\*算法思路\*\*:**

1. 贪心策略: 维护能到达的最远位置
2. 遍历数组, 更新能到达的最远位置
3. 如果当前位置超过了能到达的最远位置, 则无法到达终点
4. 如果能到达的最远位置大于等于最后一个下标, 则能到达终点

**\*\*时间复杂度\*\*:**  $O(n)$  –  $n$  是数组长度

**\*\*空间复杂度\*\*:**  $O(1)$  – 只使用了常数额外空间

**\*\*是否最优解\*\*:** 是, 这是处理此类问题的最优解法

#### #### 9. 无重叠区间

**\*\*题目描述\*\*:**

给定一个区间的集合, 找到需要移除区间的最小数量, 使剩余区间互不重叠。

**\*\*算法思路\*\*:**

1. 贪心策略: 按区间结束位置排序, 优先选择结束位置早的区间
2. 排序后遍历区间, 统计不重叠的区间数量
3. 总区间数减去不重叠区间数就是需要移除的区间数

**\*\*时间复杂度\*\*:**  $O(n * \log n)$  – 主要是排序的时间复杂度

**\*\*空间复杂度\*\*:**  $O(1)$  – 只使用了常数额外空间

**\*\*是否最优解\*\*:** 是, 这是处理此类问题的最优解法

#### #### 10. 柠檬水找零

**\*\*题目描述\*\*:**

在柠檬水摊上, 每一杯柠檬水的售价为 5 美元。顾客排队购买你的产品, (按账单 bills 支付的顺序) 一次购买一杯。每位顾客只买一杯柠檬水, 然后向你付 5 美元、10 美元或 20 美元。你必须给每个顾客正确找零, 也就是说净交易是每位顾客向你支付 5 美元。注意, 一开始你手头没有任何零钱。给你一个整数数组 bills , 其中 bills[i] 是第 i 位顾客付的账。如果你能给每位顾客正确找零, 返回 true, 否则返回 false。

**\*\*算法思路\*\*:**

1. 贪心策略: 找零时优先使用大面额纸币
2. 维护 5 美元和 10 美元纸币的数量
3. 收到 5 美元: 5 美元数量加 1
4. 收到 10 美元: 5 美元数量减 1, 10 美元数量加 1
5. 收到 20 美元: 优先用一张 10 美元和一张 5 美元找零, 如果没有 10 美元则用三张 5 美元找零

**\*\*时间复杂度\*\*:**  $O(n)$  –  $n$  是数组长度

**\*\*空间复杂度\*\*:**  $O(1)$  – 只使用了常数额外空间

**\*\*是否最优解\*\*:** 是, 这是处理此类问题的最优解法

## ### 11. 合并果子

### \*\*题目描述\*\*:

在一个果园里，多多已经将所有的果子打了下来，而且按果子的不同种类分成了不同的堆。多多决定把所有的果子合成一堆。每一次合并，多多可以把两堆果子合并到一起，消耗的体力等于两堆果子的重量之和。可以看出，所有的果子经过  $n-1$  次合并之后，就只剩下一堆了。多多在合并果子时总共消耗的体力等于每次合并所耗体力之和。因为还要花大力气把这些果子搬回家，所以多多在合并果子时要尽可能地节省体力。假定每个果子重量都为 1，并且已知果子的种类数和每种果子的数目，你的任务是设计出合并的次序方案，使多多耗费的体力最少，并输出这个最小的体力耗费值。

### \*\*算法思路\*\*:

1. 贪心策略：哈夫曼编码思想，每次选择重量最小的两堆果子合并
2. 使用最小堆维护所有果子堆
3. 每次取出重量最小的两堆果子合并，合并后的重量重新放入堆中
4. 重复直到只剩下一堆果子，累加合并过程中的体力消耗

\*\*时间复杂度\*\*:  $O(n * \log n)$  –  $n$  是果子种类数

\*\*空间复杂度\*\*:  $O(n)$  – 最小堆的空间

\*\*是否最优解\*\*: 是，这是处理此类问题的最优解法

## ### 12. 排队接水

### \*\*题目描述\*\*:

有  $n$  个人在一个水龙头前排队接水，假如每个人接水的时间为  $T_i$ ，请编程找出这  $n$  个人排队的一种顺序，使得  $n$  个人的平均等待时间最小。一个人的等待时间不包括他的接水时间。

### \*\*算法思路\*\*:

1. 贪心策略：按接水时间升序排列
2. 接水时间短的人排在前面，可以减少后面人的等待时间
3. 计算排列后的平均等待时间

\*\*时间复杂度\*\*:  $O(n * \log n)$  – 主要是排序的时间复杂度

\*\*空间复杂度\*\*:  $O(n)$  – 存储排序后的索引

\*\*是否最优解\*\*: 是，这是处理此类问题的最优解法

## ### 13. 纪念品分组

### \*\*题目描述\*\*:

元旦快到了，校学生会让乐乐负责新年晚会的纪念品发放工作。为使得参加晚会的同学所获得的纪念品价值相对均衡，他要把购来的纪念品根据价格进行分组，但每组最多只能包括两件纪念品，并且每组纪念品的价格之和不能超过一个给定的整数。为了保证在尽量短的时间内发完所有纪念品，乐乐希望分组的数目最少。你的任务是写一个程序，找出所有分组方案中分组数最少的一种，输出最少的分组数目。

**\*\*算法思路\*\*:**

1. 贪心策略: 排序后使用双指针, 最小价格和最大价格配对
2. 将纪念品价格数组排序
3. 使用双指针, 左指针指向最小价格, 右指针指向最大价格
4. 如果两件纪念品价格之和不超过上限, 则分为一组, 两个指针都移动
5. 如果超过上限, 则最大价格的纪念品单独分为一组, 只移动右指针

**\*\*时间复杂度\*\*:**  $O(n * \log n)$  – 主要是排序的时间复杂度

**\*\*空间复杂度\*\*:**  $O(1)$  – 只使用了常数额外空间

**\*\*是否最优解\*\*:** 是, 这是处理此类问题的最优解法

#### #### 14. 最小绝对差

**\*\*题目描述\*\*:**

给定一个整数数组, 找出数组中任意两个元素之间的最小绝对差。

**\*\*算法思路\*\*:**

1. 贪心策略: 排序后相邻元素的差值最小
2. 将数组排序
3. 遍历相邻元素, 计算差值, 找出最小值

**\*\*时间复杂度\*\*:**  $O(n * \log n)$  – 主要是排序的时间复杂度

**\*\*空间复杂度\*\*:**  $O(1)$  – 只使用了常数额外空间

**\*\*是否最优解\*\*:** 是, 这是处理此类问题的最优解法

#### #### 15. 纪念品分组 (牛客网版本)

**\*\*题目描述\*\*:**

与洛谷版本类似, 但可能输入输出格式略有不同。

**\*\*算法思路\*\*:**

与洛谷版本相同, 使用贪心策略和双指针。

**\*\*时间复杂度\*\*:**  $O(n * \log n)$  – 主要是排序的时间复杂度

**\*\*空间复杂度\*\*:**  $O(1)$  – 只使用了常数额外空间

**\*\*是否最优解\*\*:** 是, 这是处理此类问题的最优解法

## ## 贪心算法深度总结

### #### 适用场景与核心性质

贪心算法适用于具有以下性质的问题:

1. **\*\*贪心选择性质\*\*:** 所求问题的整体最优解可以通过一系列局部最优的选择得到

2. \*\*最优子结构\*\*: 问题的最优解包含其子问题的最优解
3. \*\*无后效性\*\*: 当前选择不会影响后续选择的最优性

#### #### 常见题型分类

##### ##### 1. 区间调度类问题

- \*\*特征\*\*: 需要选择不重叠的区间或活动
- \*\*解题模板\*\*: 按结束时间排序, 贪心选择结束最早的
- \*\*典型题目\*\*:
  - 无重叠区间 (LeetCode 435)
  - 用最少量的箭引爆气球 (LeetCode 452)
  - 合并区间 (LeetCode 56)

##### ##### 2. 资源分配类问题

- \*\*特征\*\*: 有限资源分配给多个任务, 最大化收益
- \*\*解题模板\*\*: 按单位资源收益排序, 贪心分配
- \*\*典型题目\*\*:
  - 分发饼干 (LeetCode 455)
  - 卡车上的最大单元数 (LeetCode 1710)
  - 课程表 III (LeetCode 630)

##### ##### 3. 路径优化类问题

- \*\*特征\*\*: 在路径上选择最优停留点或加油点
- \*\*解题模板\*\*: 维护当前可达范围, 贪心选择最远可达点
- \*\*典型题目\*\*:
  - 跳跃游戏 (LeetCode 55)
  - 加油站 (LeetCode 134)
  - 可以到达的最远建筑 (LeetCode 1642)

##### ##### 4. 字符串处理类问题

- \*\*特征\*\*: 重新排列字符串满足特定条件
- \*\*解题模板\*\*: 使用单调栈或自定义排序
- \*\*典型题目\*\*:
  - 最大数 (LeetCode 179)
  - 去除重复字母 (LeetCode 316)
  - 移掉 K 位数字 (LeetCode 402)

##### ##### 5. 数学优化类问题

- \*\*特征\*\*: 涉及数值计算和数学性质
- \*\*解题模板\*\*: 利用数学性质进行贪心选择
- \*\*典型题目\*\*:
  - 数组拆分 I (LeetCode 561)
  - 最小差值 II (LeetCode 910)

- 坏了的计算器 (LeetCode 991)

### ### 工程化考量与优化技巧

#### #### 1. 异常处理与边界条件

```
```java
// 输入验证
if (input == null || input.length == 0) return defaultValue;
// 边界条件处理
if (input.length == 1) return specialCaseValue;
...```

```

#### #### 2. 性能优化策略

- \*\*时间复杂度优化\*\*: 选择合适的排序算法 ( $O(n \log n)$ )
- \*\*空间复杂度优化\*\*: 使用原地操作或有限额外空间
- \*\*常数项优化\*\*: 减少不必要的对象创建和函数调用

#### #### 3. 调试与测试技巧

- \*\*打印中间过程\*\*: 在关键步骤输出变量值
- \*\*小规模测试\*\*: 验证算法在简单情况下的正确性
- \*\*边界测试\*\*: 测试空输入、极值等特殊情况

### ## 跨语言实现差异

#### #### Java 语言特性

- 使用 `Arrays.sort()` 进行排序
- 使用 `PriorityQueue` 实现堆
- 面向对象设计，代码结构清晰

#### #### C++语言特性

- 使用 `std::sort()` 和 STL 算法
- 使用 `priority\_queue` 实现堆
- 内存管理需要特别注意

#### #### Python 语言特性

- 使用 `sorted()` 或 `list.sort()` 排序
- 使用 `heapq` 模块实现堆
- 代码简洁，开发效率高

### ## 算法正确性证明方法

#### #### 1. 交换论证法

证明任何非贪心选择都可以通过有限次交换转换为贪心选择，且不会降低解的质量。

## #### 2. 归纳法证明

- **基础情况\*\*:** 证明问题规模最小时贪心选择最优
- **归纳步骤\*\*:** 假设规模为  $n-1$  时最优，证明规模为  $n$  时也最优

## #### 3. 反证法

假设存在更优解，推导出矛盾，从而证明贪心解最优。

## ### 实战经验总结

### #### 1. 题型识别技巧

- 看到“最多”、“最少”、“最大”、“最小”等关键词，考虑贪心
- 问题可以分解为多个相似的子问题，考虑贪心
- 局部最优选择明显且不会影响后续选择，考虑贪心

### #### 2. 常见陷阱避免

- **贪心不一定最优\*\*:** 需要严格证明正确性
- **边界条件遗漏\*\*:** 特别注意空输入和极端值
- **性能退化\*\*:** 避免不必要的重复计算

### #### 3. 调试定位方法

- **打印关键变量\*\*:** 在循环中输出中间结果
- **对比暴力解法\*\*:** 小规模数据下验证正确性
- **性能分析\*\*:** 使用工具分析时间空间复杂度

## ### 与其它算法的关系

### #### 1. 与动态规划的关系

- **相同点\*\*:** 都要求问题具有最优子结构
- **不同点\*\*:** 贪心算法不需要保存所有子问题的解
- **选择依据\*\*:** 如果贪心选择性质成立，优先选择贪心算法

### #### 2. 与分治法的关系

- **相同点\*\*:** 都将问题分解为子问题
- **不同点\*\*:** 贪心算法每次选择当前最优，分治法平等处理子问题

### #### 3. 与回溯法的关系

- **相同点\*\*:** 都尝试多种选择
- **不同点\*\*:** 贪心算法不回溯，回溯法会尝试所有可能

## ### 高级应用场景

### #### 1. 机器学习中的贪心算法

- 决策树构建中的特征选择
- 聚类算法中的初始中心选择
- 神经网络中的层数选择

#### #### 2. 图像处理中的应用

- 图像分割中的区域生长算法
- 边缘检测中的阈值选择
- 特征提取中的关键点选择

#### #### 3. 自然语言处理中的应用

- 文本摘要中的句子选择
- 机器翻译中的词汇选择
- 信息检索中的文档排序

### ## 学习路径建议

#### #### 初级阶段（掌握基础）

1. 理解贪心算法的核心思想
2. 掌握常见题型的解题模板
3. 完成 LeetCode 简单和中等级别的题目

#### #### 中级阶段（深入理解）

1. 学习正确性证明方法
2. 掌握工程化实现技巧
3. 完成 LeetCode 困难级别的题目

#### #### 高级阶段（融会贯通）

1. 理解贪心算法的局限性
2. 掌握与其他算法的结合使用
3. 解决实际工程问题

通过系统学习和大量练习，可以熟练掌握贪心算法的设计和应用，为解决复杂问题提供有效的工具。

### ## 工程化考量

#### ## 异常处理

1. **输入验证**: 检查输入参数的有效性
2. **边界条件**: 处理空输入、单元素等特殊情况
3. **资源限制**: 考虑时间和空间复杂度限制

#### ## 性能优化

1. **排序优化**: 选择合适的排序算法
2. **堆优化**: 使用优先队列维护动态数据

### 3. \*\*缓存优化\*\*: 避免重复计算

#### ### 可读性

1. \*\*变量命名\*\*: 使用有意义的变量名
2. \*\*注释说明\*\*: 详细解释算法思路和关键步骤
3. \*\*模块化设计\*\*: 将复杂逻辑分解为独立函数

#### ## 相关题目扩展

##### ### LeetCode 题目

1. \*\*LeetCode 455. 分发饼干\*\* - <https://leetcode.cn/problems/assign-cookies/>
2. \*\*LeetCode 122. 买卖股票的最佳时机 II\*\* - <https://leetcode.cn/problems/best-time-to-buy-and-sell-stock-ii/>
3. \*\*LeetCode 55. 跳跃游戏\*\* - <https://leetcode.cn/problems/jump-game/>
4. \*\*LeetCode 435. 无重叠区间\*\* - <https://leetcode.cn/problems/non-overlapping-intervals/>
5. \*\*LeetCode 452. 用最少量的箭引爆气球\*\* - <https://leetcode.cn/problems/minimum-number-of-arrows-to-burst-balloons/>
6. \*\*LeetCode 763. 划分字母区间\*\* - <https://leetcode.cn/problems/partition-labels/>
7. \*\*LeetCode 860. 柠檬水找零\*\* - <https://leetcode.cn/problems/lemonade-change/>
8. \*\*LeetCode 135. 分发糖果\*\* - <https://leetcode.cn/problems/candy/>
9. \*\*LeetCode 406. 根据身高重建队列\*\* - <https://leetcode.cn/problems/queue-reconstruction-by-height/>
10. \*\*LeetCode 871. 最低加油次数\*\* - <https://leetcode.cn/problems/minimum-number-of-refueling-stops/>

##### ### HackerRank 题目

1. \*\*HackerRank - Mark and Toys\*\* - <https://www.hackerrank.com/challenges/mark-and-toys>
2. \*\*HackerRank - Luck Balance\*\* - <https://www.hackerrank.com/challenges/luck-balance>
3. \*\*HackerRank - Greedy Florist\*\* - <https://www.hackerrank.com/challenges/greedy-florist>
4. \*\*HackerRank - Max Min\*\* - <https://www.hackerrank.com/challenges/angry-children>
5. \*\*HackerRank - Jim and the Orders\*\* - <https://www.hackerrank.com/challenges/jim-and-the-orders>
6. \*\*HackerRank - Minimum Absolute Difference in an Array\*\* - <https://www.hackerrank.com/challenges/minimum-absolute-difference-in-an-array/problem>

##### ### 牛客网题目

1. \*\*牛客网 - 疯狂的采药\*\* - <https://ac.nowcoder.com/acm/problem/16557>
2. \*\*牛客网 - 纪念品分组\*\* - <https://ac.nowcoder.com/acm/problem/16722>
3. \*\*牛客网 - 均分纸牌\*\* - <https://ac.nowcoder.com/acm/problem/16736>

##### ### 洛谷题目

1. \*\*洛谷 P1090 - 合并果子\*\* - <https://www.luogu.com.cn/problem/P1090>
2. \*\*洛谷 P1223 - 排队接水\*\* - <https://www.luogu.com.cn/problem/P1223>
3. \*\*洛谷 P1094 - 纪念品分组\*\* - <https://www.luogu.com.cn/problem/P1094>

4. \*\*洛谷 P1803 - 凌乱的yyy / 线段覆盖\*\* - <https://www.luogu.com.cn/problem/P1803>

#### #### Codeforces 题目

1. \*\*Codeforces 1360B - Honest Coach\*\* - <https://codeforces.com/problemset/problem/1360/B>
2. \*\*Codeforces 1367B - Even Array\*\* - <https://codeforces.com/problemset/problem/1367/B>
3. \*\*Codeforces 1374B - Multiply by 2, divide by 6\*\* -  
<https://codeforces.com/problemset/problem/1374/B>

#### #### ZOJ 题目

1. \*\*ZOJ 3211 - Dream City (砍树)\*\* - <https://zoj.pintia.cn/problems/91827364500/problems/91827367873>

## ## 复杂度分析

在分析贪心算法的复杂度时，需要考虑：

1. \*\*时间复杂度\*\* - 算法执行所需的时间
2. \*\*空间复杂度\*\* - 算法执行所需的额外空间
3. \*\*是否为最优解\*\* - 是否存在更优的算法

对于本专题中的问题，大多数都已达到理论最优复杂度。

## ## 总结

贪心算法是算法设计中的一种重要思想，通过局部最优选择来达到全局最优解。在实际应用中，需要仔细分析问题是否满足贪心选择性质和最优子结构性质，并严格证明算法的正确性。通过系统学习和大量练习，可以熟练掌握贪心算法的设计和应用技巧。

=====

## [代码文件]

文件：Code01\_EliminateMaximumMonsters.java

=====

```
package class094;
```

```
import java.util.Arrays;
import java.util.PriorityQueue;
```

```
// 消灭怪物的最大数量
// 你正在玩一款电子游戏，在游戏中你需要保护城市免受怪物侵袭
// 给定两个大小为 n 的整数数组 dist、speed
// 其中 dist[i] 是第 i 个怪物与城市的初始距离
// 其中 speed[i] 是第 i 个怪物的速度
```

```
// 你有一种武器，一旦充满电，就可以消灭一个怪物，但是，武器需要 1 的时间才能充电完成  
// 武器在游戏开始时是充满电的状态，怪物从 0 时刻开始移动，一旦任何怪物到达城市，就输掉了这场游戏  
// 如果某个怪物恰好在某一分钟开始时到达城市，这也会被视为输掉游戏  
// 返回在你输掉游戏前可以消灭的怪物的最大数量，如果消灭所有怪兽了返回 n  
// 测试链接 : https://leetcode.cn/problems/eliminate-maximum-number-of-monsters/
```

```
/*
```

```
* 题目解析:
```

```
* 这是一个典型的贪心算法问题。我们需要在每一轮中选择最优策略来最大化消灭怪物的数量。
```

```
* 关键在于理解游戏规则:
```

- \* 1. 武器初始状态是充满电的，可以立即消灭一个怪物
- \* 2. 武器充电需要 1 个时间单位
- \* 3. 怪物从时刻 0 开始移动
- \* 4. 如果任何怪物在某一分钟开始时到达城市，游戏失败
- \* 5. 我们需要找出能消灭的最大怪物数量

```
*
```

```
* 解题思路:
```

- \* 1. 贪心策略：优先消灭最早到达城市的怪物
- \* 2. 计算每个怪物到达城市的时间
- \* 3. 按时间排序，依次消灭怪物
- \* 4. 在第 i 分钟，如果第 i 个怪物还未到达，则可以消灭它
- \* 5. 如果第 i 个怪物已经到达，则游戏失败

```
*/
```

```
public class Code01_EliminateMaximumMonsters {
```

```
/*
```

```
* 算法思路:
```

- \* 1. 贪心策略：优先消灭最早到达城市的怪物
- \* 2. 计算每个怪物到达城市的时间:  $time[i] = ceil(dist[i] / speed[i])$
- \* 3. 将时间排序，按顺序消灭怪物
- \* 4. 在第 i 分钟（从 0 开始），如果第 i 个怪物还未到达城市，则可以消灭它
- \* 5. 一旦发现第 i 分钟时第 i 个怪物已经到达城市，则游戏失败

```
*
```

```
* 时间复杂度:  $O(n * logn)$  - 主要是排序的时间复杂度
```

```
* 空间复杂度:  $O(n)$  - 存储到达时间数组
```

```
* 是否最优解: 是，这是处理此类问题的最优解法
```

```
*
```

```
* 工程化考量:
```

- \* 1. 异常处理：检查输入是否为空或长度不一致
- \* 2. 边界条件：处理空数组、单个元素等特殊情况
- \* 3. 性能优化：使用向上取整公式  $(a + b - 1) / b$  避免浮点运算
- \* 4. 可读性：清晰的变量命名和注释

```
*
```

```

* 算法详解:
* 1. 计算到达时间: 使用向上取整公式计算每个怪物到达城市的时间
* 2. 排序: 按到达时间升序排列
* 3. 贪心选择: 在每一分钟选择最早到达的未消灭怪物
* 4. 游戏结束条件: 当前分钟有怪物已到达城市
*/
public static int eliminateMaximum(int[] dist, int[] speed) {
    // 异常处理: 检查输入是否为空
    if (dist == null || speed == null || dist.length != speed.length) {
        return 0;
    }

    int n = dist.length;

    // 边界条件: 没有怪物
    if (n == 0) {
        return 0;
    }

    // 计算每个怪物到达城市的时间
    // 使用向上取整公式: ceil(a/b) = (a + b - 1) / b
    // 这样可以避免浮点运算, 提高性能和精度
    int[] time = new int[n];
    for (int i = 0; i < n; i++) {
        // 向上取整公式: ceil(dist[i]/speed[i]) = (dist[i] + speed[i] - 1) / speed[i]
        // 例如: ceil(5/3) = (5+3-1)/3 = 8/3 = 2
        time[i] = (dist[i] + speed[i] - 1) / speed[i];
    }

    // 按到达时间排序, 优先消灭最早到达的怪物
    Arrays.sort(time);

    // 按顺序尝试消灭怪物
    // 在第 i 分钟 (从 0 开始), 我们可以消灭一个怪物
    // 如果第 i 个怪物已经到达城市 (time[i] <= i), 则游戏失败
    for (int i = 0; i < n; i++) {
        // 当前来到第 i 分钟 (从 0 开始)
        // 如果第 i 个怪物已经到达城市 (time[i] <= i), 则游戏失败
        if (time[i] <= i) {
            return i; // 返回已消灭的怪物数量
        }
    }
}

```

```

// 成功消灭所有怪物
return n;
}

/*
* 相关题目 1: LeetCode 871. 最低加油次数
* 题目链接: https://leetcode.cn/problems/minimum-number-of-refueling-stops/
* 题目描述: 汽车从起点出发驶向目的地, 该目的地位于出发位置东面 target 英里处。
* 沿途有加油站, 用数组 stations 表示。其中 stations[i] = [positioni, fueli]
* 表示第 i 个加油站位于出发位置东面 positioni 英里处, 并且有 fueli 升汽油。
* 假设汽车油箱的容量是无限的, 其中最初有 startFuel 升燃料。
* 它每行驶 1 英里就会用掉 1 升汽油。当汽车到达加油站时, 它可能停下来加油。
* 返回汽车到达目的地所需的最少加油次数。如果无法到达目的地, 则返回 -1。
* 解题思路: 贪心算法, 使用最大堆维护经过的加油站的油量
* 时间复杂度: O(n log n)
* 空间复杂度: O(n)
* 是否最优解: 是, 这是处理资源补充问题的经典贪心解法
*
* 算法详解:
* 1. 贪心策略: 在油用完之前, 选择经过的加油站中油量最多的
* 2. 使用最大堆维护所有经过但未加油的加油站
* 3. 当油不够到达下一个加油站或目的地时, 从堆中选择油量最多的加油
* 4. 重复直到到达目的地或无法继续前进
*/
public static int minRefuelStops(int target, int startFuel, int[][] stations) {
    // 最大堆, 维护经过的加油站的油量
    // 使用自定义比较器 (a, b) -> b - a 实现最大堆
    PriorityQueue<Integer> maxHeap = new PriorityQueue<>((a, b) -> b - a);

    int i = 0; // 加油站索引, 用于遍历所有加油站
    int res = 0; // 加油次数计数器
    int curr = startFuel; // 当前油量, 初始值为起始油量

    // 当当前油量不足以到达目的地时继续循环
    while (curr < target) {
        // 将当前能到达的所有加油站加入最大堆中
        // stations[i][0] 是加油站位置, stations[i][1] 是油量
        while (i < stations.length && stations[i][0] <= curr) {
            maxHeap.offer(stations[i++][1]); // 将油量加入堆中
        }

        // 如果没有可加油的加油站, 无法到达目的地
        if (maxHeap.isEmpty()) {

```

```

        return -1; // 无法到达目的地
    }

    // 在油量最多的加油站加油
    // poll() 方法移除并返回堆顶元素（最大油量）
    curr += maxHeap.poll();
    res++; // 加油次数加 1
}

return res; // 返回最少加油次数
}

/*
 * 相关题目 2: LeetCode 1885. 统计数对
 * 题目链接: https://leetcode.cn/problems/count-pairs-in-two-arrays/
 * 题目描述: 给你两个长度为 n 的整数数组 nums1 和 nums2，找出所有满足 i < j 且
 * nums1[i] + nums1[j] > nums2[i] + nums2[j] 的数对 (i, j)。
 * 返回满足条件的数对总数。
 * 解题思路: 数学变换 + 排序 + 双指针
 * 将不等式变换为 (nums1[i] - nums2[i]) + (nums1[j] - nums2[j]) > 0
 * 令 diff[i] = nums1[i] - nums2[i]，问题转化为在 diff 数组中找出满足 diff[i] + diff[j] > 0 且
i < j 的数对
 * 排序后使用双指针技术
 * 时间复杂度: O(n log n)
 * 空间复杂度: O(n)
 * 是否最优解: 是，这是处理此类问题的最优解法
 *
 * 算法详解:
 * 1. 数学变换: 将原不等式转换为更易处理的形式
 * 2. 构造差值数组: diff[i] = nums1[i] - nums2[i]
 * 3. 排序: 对差值数组进行排序
 * 4. 双指针: 使用双指针技术统计满足条件的数对
 * 5. 优化: 利用排序后的单调性减少计算量
 */
public static long countPairs(int[] nums1, int[] nums2) {
    int n = nums1.length;
    // 构造差值数组, diff[i] = nums1[i] - nums2[i]
    int[] diff = new int[n];

    // 计算差值数组
    for (int i = 0; i < n; i++) {
        diff[i] = nums1[i] - nums2[i];
    }
}

```

```

// 对差值数组进行排序，为双指针技术做准备
Arrays.sort(diff);

long count = 0; // 使用 long 类型防止整数溢出
int left = 0, right = n - 1; // 双指针，left 指向最小值，right 指向最大值

// 双指针统计满足条件的数对
// 条件：diff[left] + diff[right] > 0
while (left < right) {
    if (diff[left] + diff[right] > 0) {
        // 如果 diff[left] + diff[right] > 0
        // 由于数组已排序，diff[left]到 diff[right-1]与 diff[right]的和都大于 0
        // 所以有 right-left 个满足条件的数对
        count += right - left;
        right--; // 移动右指针，尝试更小的值
    } else {
        // 如果 diff[left] + diff[right] <= 0
        // 说明 diff[left]太小，需要增大
        left++; // 移动左指针，尝试更大的值
    }
}

return count; // 返回满足条件的数对总数
}
}

```

=====

文件：Code02\_LargestPalindromicNumber.java

=====

```

package class094;

import java.util.Arrays;

// 最大回文数字
// 给你一个仅由数字（0 – 9）组成的字符串 num
// 请你找出能够使用 num 中数字形成的最大回文整数
// 并以字符串形式返回，该整数不含前导零
// 你无需使用 num 中的所有数字，但你必须使用至少一个数字，数字可以重新排列
// 测试链接：https://leetcode.cn/problems/largest-palindromic-number/

/*

```

\* 题目解析:

\* 这是一个构造性问题，要求使用给定数字构造最大的回文数。

\* 关键在于理解回文数的特性:

\* 1. 回文数从左到右读和从右到左读是一样的

\* 2. 除了中心字符（奇数长度回文）外，其他字符都成对出现

\* 3. 要构造最大的回文数，应该将较大的数字放在高位

\*

\* 解题思路:

\* 1. 贪心策略：优先使用较大的数字构造回文数的高位

\* 2. 统计每个数字的出现次数

\* 3. 构造回文数的左半部分

\* 4. 确定中心数字（如果有奇数个的数字）

\* 5. 构造完整的回文数

\*/

```
public class Code02_LargestPalindromicNumber {
```

/\*

\* 算法思路:

\* 1. 贪心策略：要构造最大的回文数，应该从高位到低位选择尽可能大的数字

\* 2. 统计每个数字的出现次数

\* 3. 构造回文数的左半部分：

\* - 从数字 9 到 1，尽可能多地使用数字（每次使用 2 个）

\* - 特殊处理数字 0：只有在左半部分已经有其他数字时才使用 0

\* 4. 确定中心数字：

\* - 从 9 到 0，选择剩余数量为奇数的最大数字作为中心

\* - 如果没有奇数个的数字，且左半部分为空，则使用 0 作为中心

\* 5. 构造完整的回文数：左半部分 + 中心 + 左半部分的逆序

\*

\* 时间复杂度：O(n) - n 是字符串长度，统计和构造都是线性时间

\* 空间复杂度：O(n) - 存储结果字符数组

\* 是否最优解：是，这是处理此类问题的最优解法

\*

\* 工程化考量：

\* 1. 异常处理：检查输入是否为空

\* 2. 边界条件：处理全 0、单个数字等特殊情况

\* 3. 性能优化：使用字符数组避免字符串频繁拼接

\* 4. 可读性：清晰的变量命名和注释

\*

\* 算法详解：

\* 1. 统计频次：遍历字符串统计每个数字出现的次数

\* 2. 构造左半部分：从大到小使用成对的数字

\* 3. 确定中心：选择最大的奇数个数字作为中心

\* 4. 构造右半部分：将左半部分逆序

\* 5. 特殊处理：处理全 0 和无成对数字的情况

\*/

```
public static String largestPalindromic(String num) {
    // 异常处理：检查输入是否为空
    if (num == null || num.length() == 0) {
        return "";
    }

    int n = num.length();
    // 统计每个数字的出现次数，'0'~'9' 的 ASCII 码是 48~57
    // 使用大小为 58 的数组是为了覆盖 ASCII 码范围，虽然实际只需要 10 个位置
    int[] cnts = new int[58];
    for (char a : num.toCharArray()) {
        cnts[a]++;
    }

    // ans 数组用于存储构造的回文数
    char[] ans = new char[n];
    int leftSize = 0; // 左半部分的大小
    char mid = 0; // 中心字符，初始化为 0 表示还没有确定

    // 从数字 9 到 1 构造回文数的左半部分
    // 优先使用较大的数字构造高位，实现贪心策略
    for (char i = '9'; i >= '1'; i--) {
        // 如果当前数字有奇数个，且还没有确定中心数字，则将其作为中心
        // 使用位运算 (cnts[i] & 1) == 1 判断奇偶性，比 % 2 更高效
        if ((cnts[i] & 1) == 1 && mid == 0) {
            mid = i; // 选择最大的奇数个数字作为中心
        }

        // 将当前数字的一半数量添加到左半部分
        // 每次使用 2 个数字构造回文，所以添加 cnts[i] / 2 个
        for (int j = cnts[i] / 2; j > 0; j--) {
            ans[leftSize++] = i;
        }
    }

    // 处理特殊情况
    // 如果 leftSize 为 0，说明'1'~'9' 每一种字符出现次数 <= 1
    if (leftSize == 0) {
        if (mid == 0) { // '1'~'9' 每一种字符出现次数 == 0，即全是 0
            return "0"; // 特殊情况：输入全是 0，返回"0"而不是空字符串
        } else { // '1'~'9' 有若干字符出现次数 == 1，其中最大的字符是 mid
    }}
```

```

        return String.valueOf(mid); // 只有一个非 0 数字，直接返回
    }
}

// '1' ~ '9' 有若干字符出现次数 >= 2，左部分已经建立，再考虑把'0'字符填进来
// 数字 0 只能在左半部分已经有其他数字时才使用，避免前导零
for (int j = cnts['0'] / 2; j > 0; j--) {
    ans[leftSize++] = '0';
}

int len = leftSize; // 当前构造的长度
// 确定中心数字
// 如果还没有确定中心且数字 0 有奇数个，则使用 0 作为中心
if (mid == 0 && (cnts['0'] & 1) == 1) {
    mid = '0';
}

// 添加中心数字（如果存在）
// 只有在回文数长度为奇数时才有中心字符
if (mid != 0) {
    ans[len++] = mid;
}

// 构造右半部分：左部分逆序拷贝给右部分
// 从左半部分的最后一个字符开始，逆序复制到右半部分
for (int i = leftSize - 1; i >= 0; i--) {
    ans[len++] = ans[i];
}

// 返回构造的回文数
return new String(ans, 0, len);
}

/*
 * 相关题目 1: LeetCode 9. 回文数
 * 题目链接: https://leetcode.cn/problems/palindrome-number/
 * 题目描述: 给你一个整数 x ，如果 x 是一个回文整数，返回 true ；否则，返回 false 。
 * 回文数是指正序（从左向右）和倒序（从右向左）读都是一样的整数。
 * 例如，121 是回文，而 123 不是。
 * 解题思路: 数学方法，通过取余和除法操作反转整数的一半
 * 时间复杂度: O(log n) - n 是输入数字的值
 * 空间复杂度: O(1)
 * 是否最优解: 是，这是处理此类问题的最优解法

```

```

/*
 * 算法详解:
 * 1. 特殊情况处理: 负数和末尾为 0 但非 0 的数不是回文数
 * 2. 数学反转: 通过取余和除法操作反转整数的一半
 * 3. 比较判断: 将原数和反转数进行比较
 * 4. 奇数长度处理: 通过 revertedNumber/10 去除中位数字
 */

public static boolean isPalindrome(int x) {
    // 负数和末尾为 0 但非 0 的数不是回文数
    // 负数不是回文数, 因为负号只在一边
    // 末尾为 0 但非 0 的数不是回文数, 因为开头不能为 0
    if (x < 0 || (x % 10 == 0 && x != 0)) {
        return false;
    }

    int revertedNumber = 0;
    // 反转数字的一半
    // 当 x <= revertedNumber 时, 说明已经反转了一半数字
    while (x > revertedNumber) {
        // 将 x 的最后一位数字添加到 revertedNumber 的末尾
        revertedNumber = revertedNumber * 10 + x % 10;
        // 去掉 x 的最后一位数字
        x /= 10;
    }

    // 当数字长度为奇数时, 我们可以通过 revertedNumber/10 去除处于中位的数字
    // 例如: 12321, 经过循环后 x=12, revertedNumber=123, 中位数 3 需要被忽略
    // 所以比较 x == revertedNumber/10
    // 当数字长度为偶数时, 直接比较 x == revertedNumber
    return x == revertedNumber || x == revertedNumber / 10;
}

/*
 * 相关题目 2: LeetCode 5. 最长回文子串
 * 题目链接: https://leetcode.cn/problems/longest-palindromic-substring/
 * 题目描述: 给你一个字符串 s, 找到 s 中最长的回文子串。
 * 解题思路: 中心扩展法, 枚举每个可能的中心位置, 向两边扩展
 * 时间复杂度: O(n^2) - n 是字符串长度
 * 空间复杂度: O(1)
 * 是否最优解: 否, 存在 Manacher 算法可以达到 O(n) 时间复杂度
 *
 * 算法详解:
 * 1. 中心扩展: 以每个字符和每两个字符之间为中点进行扩展

```

```

* 2. 两种情况：奇数长度回文（以字符为中心）和偶数长度回文（以字符间隙为中心）
* 3. 扩展过程：从中心向两边扩展，直到字符不匹配或越界
* 4. 记录最长：记录所有回文中长度最长的一个
*/
public static String longestPalindrome(String s) {
    // 输入验证：检查字符串是否为空
    if (s == null || s.length() < 1) return "";

    int start = 0, end = 0; // 记录最长回文子串的起始和结束位置
    // 遍历每个可能的中心位置
    for (int i = 0; i < s.length(); i++) {
        // 以 i 为中心的奇数长度回文，如"aba"
        int len1 = expandAroundCenter(s, i, i);
        // 以 i 和 i+1 为中心的偶数长度回文，如"abba"
        int len2 = expandAroundCenter(s, i, i + 1);
        // 取两种情况中的最大长度
        int len = Math.max(len1, len2);

        // 更新最长回文子串的起始和结束位置
        // 如果当前回文长度大于之前记录的最长回文长度
        if (len > end - start) {
            // 计算新的起始位置: i - (len - 1) / 2
            // 对于奇数长度: i - (len - 1) / 2 = i - (2k+1-1) / 2 = i - k
            // 对于偶数长度: i - (len - 1) / 2 = i - (2k-1) / 2 = i - (k-1)
            start = i - (len - 1) / 2;
            // 计算新的结束位置: i + len / 2
            // 对于奇数长度: i + len / 2 = i + 2k / 2 = i + k
            // 对于偶数长度: i + len / 2 = i + 2k / 2 = i + k
            end = i + len / 2;
        }
    }

    // 返回最长回文子串
    return s.substring(start, end + 1);
}

// 辅助函数：中心扩展
// 从给定的 left 和 right 位置向两边扩展，找到以该位置为中心的最长回文
private static int expandAroundCenter(String s, int left, int right) {
    // 从中心向两边扩展，直到字符不匹配或越界
    // left >= 0: 左边界检查
    // right < s.length(): 右边界检查
    // s.charAt(left) == s.charAt(right): 字符匹配检查
}

```

```

        while (left >= 0 && right < s.length() && s.charAt(left) == s.charAt(right)) {
            left--; // 左指针向左移动
            right++; // 右指针向右移动
        }
        // 返回回文长度
        // 循环结束时, left 和 right 分别指向回文边界外的第一个字符
        // 所以回文长度为 right - left - 1
        return right - left - 1;
    }
}
=====
```

文件: Code03\_MaximumAveragePassRatio.java

```

package class094;

import java.util.PriorityQueue;
import java.util.Comparator;

// 最大平均通过率
// 一所学校里有一些班级, 每个班级里有一些学生, 现在每个班都会进行一场期末考试
// 给你一个二维数组 classes, 其中 classes[i]=[passi, totali]
// 表示你提前知道了第 i 个班级总共有 totali 个学生
// 其中只有 passi 个学生可以通过考试
// 给你一个整数 extraStudents, 表示额外有 extraStudents 个聪明的学生, 一定能通过期末考
// 你需要给这 extraStudents 个学生每人都安排一个班级, 使得所有班级的平均通过率最大
// 一个班级的 通过率 等于这个班级通过考试的学生人数除以这个班级的总人数
// 平均通过率 是所有班级的通过率之和除以班级数目
// 请你返回在安排这 extraStudents 个学生去对应班级后的最大平均通过率
// 测试链接 : https://leetcode.cn/problems/maximum-average-pass-ratio/
```

```

/*
 * 题目解析:
 * 这是一个资源分配问题, 要求将额外的学生分配到各个班级以最大化平均通过率。
 * 关键在于理解如何衡量分配的收益:
 * 1. 每个班级的通过率 = 通过人数 / 总人数
 * 2. 平均通过率 = 所有班级通过率之和 / 班级数
 * 3. 分配一个学生到班级的收益 = 分配后的通过率 - 分配前的通过率
 *
 * 解题思路:
 * 1. 贪心策略: 每次分配学生时, 选择能带来最大收益的班级
 * 2. 使用优先队列维护班级, 按收益降序排列
```

```

* 3. 重复分配过程，直到所有学生都被分配
* 4. 计算最终的平均通过率
*/
public class Code03_MaximumAveragePassRatio {

    /*
     * 算法思路：
     * 1. 贪心策略：每次分配一个学生时，选择能使得通过率提升最大的班级
     * 2. 通过率提升计算：对于班级(a, b)，增加一个学生后通过率提升为  $(a+1)/(b+1) - a/b$ 
     * 3. 使用优先队列维护所有班级，按通过率提升量排序
     * 4. 每次取出提升量最大的班级，分配一个学生，然后重新计算提升量并放回队列
     * 5. 重复 extraStudents 次，最后计算平均通过率
     *
     * 时间复杂度： $O((n + m) * log n)$  - n 是班级数，m 是额外学生数
     * 空间复杂度： $O(n)$  - 优先队列的空间
     * 是否最优解：是，这是处理此类问题的最优解法
     *
     * 工程化考量：
     * 1. 异常处理：检查输入是否为空
     * 2. 边界条件：处理班级数为 0、额外学生数为 0 等特殊情况
     * 3. 性能优化：使用 double 数组避免创建对象的开销
     * 4. 可读性：清晰的变量命名和注释
     *
     * 算法详解：
     * 1. 收益计算：对于班级(a, b)，增加一个学生后的收益为  $(a+1)/(b+1) - a/b$ 
     * 2. 优先队列：使用大根堆维护班级，按收益降序排列
     * 3. 贪心分配：每次选择收益最大的班级分配学生
     * 4. 动态更新：分配学生后重新计算班级收益并更新队列
     */

    public static double maxAverageRatio(int[][] classes, int m) {
        // 异常处理：检查输入是否为空
        if (classes == null || classes.length == 0) {
            return 0.0;
        }

        int n = classes.length;
        // double[] c1 : {a, b, c}
        // a : c1 班级有多少人通过
        // b : c1 班级总人数
        // c : 如果再来一人，c1 班级通过率提升多少， $(a+1)/(b+1) - a/b$ 
        // 通过率提升的大根堆
        // 使用自定义比较器实现大根堆：c1[2] >= c2[2] ? -1 : 1
        // 当 c1[2] >= c2[2] 时返回-1，表示 c1 优先级更高
    }
}

```

```

PriorityQueue<double[]> heap = new PriorityQueue<>((c1, c2) -> c1[2] >= c2[2] ? -1 : 1);

// 初始化堆，计算每个班级增加一个学生后的通过率提升
// 对于班级(pass, total)，收益为 (pass+1)/(total+1) - pass/total
for (int[] c : classes) {
    double a = c[0]; // 通过人数
    double b = c[1]; // 总人数
    // 计算增加一个学生后的通过率提升
    // 提升量 = (a+1)/(b+1) - a/b
    heap.add(new double[] { a, b, (a + 1) / (b + 1) - a / b });
}

// 逐个分配额外的学生
// 贪心策略：每次分配一个学生到能带来最大通过率提升的班级
while (m-- > 0) {
    // 取出能带来最大通过率提升的班级
    double[] cur = heap.poll();
    double a = cur[0] + 1; // 通过人数增加 1
    double b = cur[1] + 1; // 总人数增加 1
    // 重新计算通过率提升并放回堆中
    // 分配学生后，需要重新计算该班级的收益
    heap.add(new double[] { a, b, (a + 1) / (b + 1) - a / b });
}

// 计算最终的通过率累加和
double ans = 0;
// 遍历所有班级，计算最终的通过率之和
while (!heap.isEmpty()) {
    double[] cur = heap.poll();
    // 累加每个班级的通过率
    ans += cur[0] / cur[1];
}

// 返回最大平均通过率
// 平均通过率 = 所有班级通过率之和 / 班级数
return ans / n;
}

/*
 * 相关题目 1: LeetCode 1705. 吃苹果的最大数目
 * 题目链接: https://leetcode.cn/problems/maximum-number-of-eaten-apples/
 * 题目描述: 有一棵特殊的苹果树，一连 n 天，每天都可以长出若干个苹果。
 * 在第 i 天，树上会长出 apples[i] 个苹果，这些苹果将会在 days[i] 天后（也就是说，第 i +

```

days[i] 天时) 腐烂, 变得无法食用。

\* 也可能有那么几天树上不会长出新的苹果, 此时用 apples[i] == 0 且 days[i] == 0 表示。

\* 你打算每天最多吃一个苹果来保证营养均衡。注意, 你可以在这 n 天之后继续吃苹果。

\* 给你两个长度为 n 的整数数组 days 和 apples , 返回你可以吃掉的苹果的最大数目。

\* 解题思路: 贪心算法, 使用最小堆维护苹果的过期时间

\* 时间复杂度: O(n log n)

\* 空间复杂度: O(n)

\* 是否最优解: 是, 这是处理此类问题的最优解法

\*

\* 算法详解:

\* 1. 贪心策略: 每天吃最容易腐烂的苹果

\* 2. 数据结构: 使用最小堆维护苹果的过期时间

\* 3. 过程模拟: 按天模拟苹果的生长和腐烂过程

\* 4. 优化选择: 优先吃最早过期的苹果

\*/

```
public static int eatenApples(int[] apples, int[] days) {  
    // 最小堆, 维护苹果的过期时间和数量  
    // 堆中元素为 int[2]数组: {过期时间, 苹果数量}  
    // 按过期时间升序排列, 最早过期的在堆顶  
    PriorityQueue<int[]> minHeap = new PriorityQueue<>(new Comparator<int[]>() {  
        public int compare(int[] a, int[] b) {  
            return a[0] - b[0]; // 按过期时间升序排列  
        }  
    });  
  
    int eaten = 0; // 已吃苹果数量  
    int day = 0; // 当前天数  
  
    // 处理前 n 天长出的苹果  
    // 在这 n 天中, 每天可能长出新苹果, 也可能有苹果腐烂  
    while (day < apples.length) {  
        // 移除过期的苹果  
        // 检查堆顶苹果是否已过期 (过期时间 <= 当前天数)  
        while (!minHeap.isEmpty() && minHeap.peek()[0] <= day) {  
            minHeap.poll(); // 移除过期苹果  
        }  
  
        // 添加当天新长出的苹果  
        int expireDay = day + days[day]; // 计算过期时间  
        int count = apples[day]; // 当天长出的苹果数量  
        if (count > 0) {  
            // 只有当长出苹果时才添加到堆中  
            minHeap.offer(new int[] {expireDay, count});  
        }  
        day++;  
    }  
    return eaten;  
}
```

```

    }

    // 吃一个苹果
    // 贪心策略：吃最早过期的苹果
    if (!minHeap.isEmpty()) {
        int[] apple = minHeap.peek(); // 获取最早过期的苹果
        apple[1]--; // 苹果数量减 1
        eaten++; // 已吃苹果数量加 1
        // 如果苹果吃完了，移除
        if (apple[1] == 0) {
            minHeap.poll(); // 移除空的苹果记录
        }
    }

    day++; // 进入下一天
}

// 处理 n 天后剩余的苹果
// 在 n 天之后，不再长出新苹果，但可以继续吃剩余的苹果
while (!minHeap.isEmpty()) {
    // 移除过期的苹果
    // 继续检查并移除过期苹果
    while (!minHeap.isEmpty() && minHeap.peek()[0] <= day) {
        minHeap.poll();
    }

    if (!minHeap.isEmpty()) {
        int[] apple = minHeap.poll(); // 取出最早过期的苹果
        // 计算可以吃的苹果数量
        // 取 min(苹果数量, 剩余可吃天数)
        // 剩余可吃天数 = 过期时间 - 当前天数
        int eat = Math.min(apple[0] - day, apple[1]);
        eaten += eat; // 增加已吃苹果数量
        day += eat; // 更新天数
    }
}

return eaten; // 返回总共吃掉的苹果数量
}

/*
 * 相关题目 2: 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/

```

- \* 题目描述：给你一个数组 events，其中 events[i] = [startDayi, endDayi]，表示会议 i 开始于 startDayi，结束于 endDayi。
- \* 你可以在满足 startDayi <= d <= endDayi 的任意一天 d 参加会议 i。在任意一天 d 中只能参加一场会议。
- \* 请你返回你可以参加的最大会议数目。
- \* 解题思路：贪心算法，使用最小堆维护当天可以参加会议的结束时间
- \* 时间复杂度：O(n log n)
- \* 空间复杂度：O(n)
- \* 是否最优解：是，这是处理此类问题的最优解法
- \*
- \* 算法详解：
- \* 1. 贪心策略：每天参加结束时间最早的会议
- \* 2. 数据结构：使用最小堆维护可参加会议的结束时间
- \* 3. 过程模拟：按天模拟会议的开始和结束
- \* 4. 优化选择：优先参加最早结束的会议

```

/*
public static int maxEvents(int[][] events) {
    // 按开始时间排序
    // 将会议按开始时间升序排列，便于按时间顺序处理
    java.util.Arrays.sort(events, new Comparator<int[]>() {
        public int compare(int[] a, int[] b) {
            return a[0] - b[0]; // 按开始时间升序排列
        }
    });
}

// 最小堆，维护可以参加会议的结束时间
// 堆中存储会议的结束时间，最早结束的会议在堆顶
PriorityQueue<Integer> minHeap = new PriorityQueue<>();

int i = 0, day = 1, count = 0; // i: 会议索引, day: 当前天数, count: 参加会议数
int n = events.length; // 会议总数

// 当还有未处理的会议或堆中还有可参加会议时继续循环
while (i < n || !minHeap.isEmpty()) {
    // 将当天开始的会议加入堆中
    // 将所有在当前天开始的会议添加到堆中
    while (i < n && events[i][0] == day) {
        // 将会议的结束时间加入堆中
        minHeap.offer(events[i][1]);
        i++; // 处理下一个会议
    }

    // 移除已经结束的会议
}

```

```

        // 移除所有在当前天之前结束的会议
        while (!minHeap.isEmpty() && minHeap.peek() < day) {
            minHeap.poll(); // 移除过期会议
        }

        // 参加结束时间最早的会议
        // 贪心策略：参加结束时间最早的会议
        if (!minHeap.isEmpty()) {
            minHeap.poll(); // 参加会议（从堆中移除）
            count++; // 参加会议数加 1
        }

        day++; // 进入下一天
    }

    return count; // 返回最多可以参加的会议数目
}
}

```

=====

文件: Code04\_MinimumCostToHireWorkers.java

=====

```

package class094;

import java.util.Arrays;
import java.util.PriorityQueue;

// 雇佣 K 名工人的最低成本
// 有 n 名工人，给定两个数组 quality 和 wage
// 其中 quality[i] 表示第 i 名工人工作质量，其最低期望工资为 wage[i]
// 现在我们想雇佣 k 名工人组成一个工资组
// 在雇佣一组 k 名工人时，我们必须按照下述规则向他们支付工资：
// 对工资组中的每名工人，应当按其工作质量与同组其他工人的工作质量的比例来支付工资
// 工资组中的每名工人至少应当得到他们的最低期望工资
// 给定整数 k，返回组成满足上述条件的付费群体所需的最小金额
// 测试链接 : https://leetcode.cn/problems/minimum-cost-to-hire-k-workers/

/*
 * 题目解析:
 * 这是一个优化问题，要求在满足约束条件下找到雇佣 k 名工人的最低成本。
 * 关键约束条件:
 * 1. 工资按工作质量比例分配

```

\* 2. 每名工人至少得到最低期望工资

\* 3. 需要雇佣恰好 k 名工人

\*

\* 解题思路:

\* 1. 关键洞察: 在一组工人中, 实际支付比例必须是这组工人中最大的比例

\* 2. 贪心策略: 枚举每个工人作为基准, 选择质量总和最小的 k 个工人

\* 3. 使用优先队列维护最优解

\* 4. 遍历所有可能的组合找到最小成本

\*/

```
public class Code04_MinimumCostToHireWorkers {
```

```
    public static class Employee {
```

```
        public double ratio; // 薪水 / 质量的比例
```

```
        public int quality;
```

```
        public Employee(double r, int q) {
```

```
            ratio = r;
```

```
            quality = q;
```

```
        }
```

```
}
```

/\*

\* 算法思路:

\* 1. 关键洞察: 在一组工人中, 实际支付比例必须是这组工人中最大的比例 (以满足最低期望工资要求)

\* 2. 贪心策略:

\* - 按照薪水/质量比例排序

\* - 枚举每个工人作为基准 (具有最大比例)

\* - 在该工人之前选择质量总和最小的 k-1 个工人

\* 3. 使用最大堆维护当前质量最小的 k 个工人

\* 4. 遍历排序后的工人, 维护堆并计算最小成本

\*

\* 时间复杂度:  $O(n * \log n)$  - 主要是排序和堆操作的复杂度

\* 空间复杂度:  $O(n)$  - 存储员工数组和堆

\* 是否最优解: 是, 这是处理此类问题的最优解法

\*

\* 工程化考量:

\* 1. 异常处理: 检查输入是否为空或长度不一致

\* 2. 边界条件: 处理 k 大于 n 等特殊情况

\* 3. 性能优化: 使用最大堆维护最小质量总和

\* 4. 可读性: 清晰的变量命名和注释

\*

\* 算法详解:

\* 1. 比例计算: 计算每个工人的薪水/质量比例

```

* 2. 排序策略: 按比例升序排列
* 3. 基准选择: 枚举每个工人作为支付比例基准
* 4. 质量优化: 使用最大堆维护最小质量总和
* 5. 成本计算: 总质量 × 基准比例
*/
public static double mincostToHireWorkers(int[] quality, int[] wage, int k) {
    // 异常处理: 检查输入是否为空或长度不一致
    if (quality == null || wage == null || quality.length != wage.length) {
        return 0;
    }

    int n = quality.length;

    // 边界条件: k 大于 n
    if (k > n) {
        return 0;
    }

    // 创建员工数组, 存储每个员工的比例和质量
    Employee[] employees = new Employee[n];
    for (int i = 0; i < n; i++) {
        // 计算薪水/质量比例, 用于后续排序和成本计算
        employees[i] = new Employee((double) wage[i] / quality[i], quality[i]);
    }

    // 根据比例排序, 比例小的在前, 比例大的在后
    // 使用自定义比较器实现升序排列
    Arrays.sort(employees, (a, b) -> a.ratio <= b.ratio ? -1 : 1);

    // 大根堆, 用来收集最小的前 k 个质量数值
    // 使用自定义比较器 (a, b) -> b - a 实现最大堆
    PriorityQueue<Integer> heap = new PriorityQueue<Integer>((a, b) -> b - a);

    // qualitySum: 堆中 k 个员工的质量总和
    int qualitySum = 0;
    // ans: 记录最小成本, 初始化为最大值
    double ans = Double.MAX_VALUE;

    // 遍历排序后的员工数组
    for (int i = 0, curQuality; i < n; i++) {
        curQuality = employees[i].quality; // 当前员工的质量

        if (heap.size() < k) { // 堆没满, 直接添加
            heap.add(curQuality);
            qualitySum += curQuality;
        } else if (curQuality < heap.peek()) {
            heap.poll();
            heap.add(curQuality);
            qualitySum -= heap.peek();
            qualitySum += curQuality;
        }
    }

    return qualitySum * ans;
}

```

```

        qualitySum += curQuality; // 累加质量
        heap.add(curQuality); // 添加到堆中

        // 当堆大小等于 k 时，计算当前成本
        if (heap.size() == k) {
            // 成本 = 质量总和 × 当前员工的比例（最大比例）
            ans = Math.min(ans, qualitySum * employees[i].ratio);
        }
    } else { // 堆满了，需要判断是否替换
        // 如果当前员工质量小于堆顶质量，则替换
        if (heap.peek() > curQuality) {
            // 更新质量总和：减去堆顶质量，加上当前质量
            qualitySum += curQuality - heap.poll();
            // 添加当前员工质量到堆中
            heap.add(curQuality);
            // 计算并更新最小成本
            ans = Math.min(ans, qualitySum * employees[i].ratio);
        }
    }
}

return ans; // 返回最小成本
}

```

```

/*
* 相关题目 1: LeetCode 215. 数组中的第 K 个最大元素
* 题目链接: https://leetcode.cn/problems/kth-largest-element-in-an-array/
* 题目描述: 给定整数数组 nums 和整数 k, 请返回数组中第 k 个最大的元素。
* 请注意, 你需要找的是数组排序后的第 k 个最大的元素, 而不是第 k 个不同的元素。
* 解题思路: 使用最小堆维护最大的 k 个元素
* 时间复杂度: O(n log k)
* 空间复杂度: O(k)
* 是否最优解: 是, 这是处理此类问题的最优解法
*
* 算法详解:
* 1. 数据结构: 使用大小为 k 的最小堆
* 2. 维护策略: 堆中始终保存最大的 k 个元素
* 3. 替换条件: 当新元素大于堆顶时替换
* 4. 结果获取: 堆顶即为第 k 大元素
*/

```

```

public static int findKthLargest(int[] nums, int k) {
    // 最小堆, 维护最大的 k 个元素
    // 堆的大小始终保持为 k, 堆顶是最小的元素
}

```

```

PriorityQueue<Integer> minHeap = new PriorityQueue<>();

// 遍历数组中的每个元素
for (int num : nums) {
    if (minHeap.size() < k) {
        // 堆未满, 直接添加元素
        minHeap.offer(num);
    } else if (num > minHeap.peek()) {
        // 堆已满且当前元素大于堆顶元素
        // 移除堆顶 (最小元素), 添加当前元素
        minHeap.poll();
        minHeap.offer(num);
    }
}

// 返回堆顶元素, 即第 k 大元素
return minHeap.peek();
}

/*
 * 相关题目 2: LeetCode 703. 数据流中的第 K 大元素
 * 题目链接: https://leetcode.cn/problems/kth-largest-element-in-a-stream/
 * 题目描述: 设计一个找到数据流中第 k 大元素的类 (class)。
 * 注意是排序后的第 k 大元素, 不是第 k 个不同的元素。
 * 请实现 KthLargest 类:
 * KthLargest(int k, int[] nums) 使用整数 k 和整数流 nums 初始化对象。
 * int add(int val) 将 val 插入数据流 nums 后, 返回当前数据流中第 k 大的元素。
 * 解题思路: 使用最小堆维护最大的 k 个元素
 * 时间复杂度: 初始化 O(n log k), add 操作 O(log k)
 * 空间复杂度: O(k)
 * 是否最优解: 是, 这是处理此类问题的最优解法
 *
 * 算法详解:
 * 1. 类设计: 封装 k 值和最小堆
 * 2. 初始化: 处理初始数据流
 * 3. 动态更新: 通过 add 方法处理新元素
 * 4. 结果获取: 堆顶即为第 k 大元素
 */
public static class KthLargest {
    private int k; // 第 k 大元素的 k 值
    private PriorityQueue<Integer> minHeap; // 最小堆, 维护最大的 k 个元素

    public KthLargest(int k, int[] nums) {

```

```

        this.k = k;
        // 最小堆，维护最大的 k 个元素
        // 堆的大小始终保持为 k，堆顶是最小的元素
        this.minHeap = new PriorityQueue<>();

        // 初始化时处理初始数据流
        for (int num : nums) {
            add(num); // 调用 add 方法处理每个元素
        }
    }

    public int add(int val) {
        // 添加新元素到数据流
        if (minHeap.size() < k) {
            // 堆未满，直接添加元素
            minHeap.offer(val);
        } else if (val > minHeap.peek()) {
            // 堆已满且当前元素大于堆顶元素
            // 移除堆顶（最小元素），添加当前元素
            minHeap.poll();
            minHeap.offer(val);
        }
    }

    // 返回当前数据流中第 k 大元素
    return minHeap.peek();
}
}
}

```

文件: Code05\_CuttingTree.java

```

=====
package class094;

// 砍树
// 一共有 n 棵树，每棵树都有两个信息：
// 第一天这棵树的初始重量、这棵树每天的增长重量
// 你每天最多能砍 1 棵树，砍下这棵树的收益为：
// 这棵树的初始重量 + 这棵树增长到这一天的总增重
// 从第 1 天开始，你一共有 m 天可以砍树，返回 m 天内你获得的最大收益
// 测试链接：https://pintia.cn/problem-
sets/91827364500/exam/problems/type/7?problemSetProblemId=91827367873

```

```

// 如果测试链接失效，搜索 "ZOJ Problem Set" 所在网站，找第 3211 题 "Dream City"
// 提交以下的 code，提交时请把类名改成"Main"，可以直接通过

/*
 * 题目解析：
 * 这是一个资源分配和调度问题，要求在有限的天数内选择砍伐树木的顺序和数量以最大化收益。
 * 关键在于理解收益计算方式和决策约束：
 * 1. 收益 = 初始重量 + (砍伐天数-1) × 每天增长重量
 * 2. 每天只能砍伐一棵树
 * 3. 需要在 m 天内完成砍伐
 *
 * 解题思路：
 * 1. 贪心策略：按增长速度排序，增长慢的先砍
 * 2. 动态规划：使用 01 背包变种解决选择问题
 * 3. 状态设计：dp[i][j] 表示前 i 棵树在 j 天内的最大收益
 * 4. 状态转移：考虑是否砍伐当前树
*/

```

```

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

public class Code05_CuttingTree {

    // 树的个数、天数最大值，不超过的量
    public static int MAXN = 251;

    // 树的编号为 1 ~ n
    // tree[i][0] : 第 i 棵树第一天的初始重量
    // tree[i][1] : 第 i 棵树每天的增长重量
    public static int[][] tree = new int[MAXN][2];

    // dp[i][j] : 在 j 天内，从前 i 棵树中选若干棵树进行砍伐，最大收益是多少
    public static int[][] dp = new int[MAXN][MAXN];

    public static int t, n, m;

    public static void main(String[] args) throws IOException {
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

```

```

StreamTokenizer in = new StreamTokenizer(br);
PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));
in.nextToken();
t = (int) in.nval;
for (int i = 1; i <= t; i++) {
    in.nextToken();
    n = (int) in.nval;
    in.nextToken();
    m = (int) in.nval;
    for (int j = 1; j <= n; j++) {
        in.nextToken();
        tree[j][0] = (int) in.nval;
    }
    for (int j = 1; j <= n; j++) {
        in.nextToken();
        tree[j][1] = (int) in.nval;
    }
    out.println(compute());
}
out.flush();
out.close();
br.close();
}

```

// 讲解 073 - 01 背包

/\*

\* 算法思路:

\* 1. 贪心策略: 根据增长速度排序, 增长量小的在前, 增长量大的在后

\* 2. 动态规划: 01 背包问题的变种

\* - 状态定义:  $dp[i][j]$  表示在  $j$  天内, 从前  $i$  棵树中选若干棵树进行砍伐, 最大收益是多少

\* - 状态转移:  $dp[i][j] = \max(dp[i-1][j], dp[i-1][j-1] + tree[i][0] + tree[i][1] * (j-1))$

\* - 初始条件:  $dp[0][\dots] = 0$ ,  $dp[\dots][0] = 0$

\*

\* 时间复杂度:  $O(n * \log n + n * m)$  - 排序和 DP 的时间复杂度

\* 空间复杂度:  $O(n * m)$  - DP 数组的空间复杂度

\* 是否最优解: 是, 这是处理此类问题的最优解法

\*

\* 工程化考量:

\* 1. 异常处理: 检查输入是否合法

\* 2. 边界条件: 处理  $n=0$ ,  $m=0$  等特殊情况

\* 3. 性能优化: 使用静态数组避免频繁内存分配

\* 4. 可读性: 清晰的变量命名和注释

\*

```

* 算法详解:
* 1. 贪心排序: 按增长速度升序排列, 增长慢的优先砍伐
* 2. 状态转移: 对于每棵树, 选择砍或不砍
* 3. 收益计算: 初始重量 + (砍伐天数-1) × 每天增长重量
* 4. 边界处理: 无树或无天数时收益为 0
*/
public static int compute() {
    // 树的初始重量不决定树的顺序, 因为任何树砍了, 就获得固定的初始量, 和砍伐的顺序无关
    // 根据增长速度排序, 增长量小的在前, 增长量大的在后
    // 认为越靠后的树, 越要尽量晚的砍伐, 课上的重点内容
    // 使用 Arrays.sort 对 tree 数组进行排序, 范围是[1, n+1)
    // 比较器(o1, o2) -> o1[1] - o2[1]按增长速度升序排列
    Arrays.sort(tree, 1, n + 1, (o1, o2) -> o1[1] - o2[1]);

    // dp[0][...] = 0 : 表示如果没有树, 不管过去多少天, 收益都是 0
    // dp[...][0] = 0 : 表示不管有几棵树, 没有时间砍树, 收益都是 0
    // 动态规划填表过程
    for (int i = 1; i <= n; i++) {
        // 对于前 i 棵树
        for (int j = 1; j <= m; j++) {
            // 对于 j 天时间
            // 状态转移方程:
            // dp[i][j] = max(不砍第 i 棵树, 砍第 i 棵树)
            // 不砍第 i 棵树: dp[i-1][j]
            // 砍第 i 棵树: dp[i-1][j-1] + tree[i][0] + tree[i][1] * (j-1)
            // tree[i][0]: 第 i 棵树的初始重量
            // tree[i][1]: 第 i 棵树在第 j 天砍伐时的增长重量
            dp[i][j] = Math.max(dp[i - 1][j], dp[i - 1][j - 1] + tree[i][0] + tree[i][1] * (j - 1));
        }
    }

    // 返回前 n 棵树在 m 天内的最大收益
    return dp[n][m];
}

/*
* 相关题目 1: LeetCode 45. 跳跃游戏 II
* 题目链接: https://leetcode.cn/problems/jump-game-ii/
* 题目描述: 给定一个长度为 n 的 0 索引整数数组 nums。初始位置为 nums[0]。
* 每个元素 nums[i] 表示从索引 i 向前跳转的最大长度。
* 返回到达 nums[n - 1] 的最小跳跃次数。
* 解题思路: 贪心算法, 每次选择能跳到最远位置的点

```

```
* 时间复杂度: O(n)
* 空间复杂度: O(1)
* 是否最优解: 是, 这是处理此类问题的最优解法
*
* 算法详解:
* 1. 贪心策略: 在当前跳跃范围内选择能跳到最远位置的点
* 2. 边界处理: 不需要考虑最后一个位置
* 3. 优化提前: 当能到达终点时提前结束
* 4. 状态维护: 维护当前跳跃边界和最远可达位置
*/

```

```
public static int jump(int[] nums) {
    int n = nums.length;
    // 边界条件: 数组长度小于等于 1 时不需要跳跃
    if (n <= 1) return 0;

    int jumps = 0;      // 跳跃次数
    int currentEnd = 0; // 当前跳跃能到达的最远位置
    int farthest = 0;   // 在当前跳跃范围内能到达的最远位置

    // 注意: 不需要考虑最后一个位置, 因为到达最后一个位置就结束了
    for (int i = 0; i < n - 1; i++) {
        // 更新在当前跳跃范围内能到达的最远位置
        // i + nums[i] 表示从位置 i 能跳到的最远位置
        farthest = Math.max(farthest, i + nums[i]);

        // 如果到达当前跳跃的边界
        // 说明需要进行下一次跳跃
        if (i == currentEnd) {
            jumps++; // 跳跃次数加 1
            currentEnd = farthest; // 更新当前跳跃的边界

            // 如果已经能到达最后一个位置, 提前结束
            // 优化: 避免不必要的计算
            if (currentEnd >= n - 1) {
                break;
            }
        }
    }

    return jumps; // 返回最小跳跃次数
}

/*

```

- \* 相关题目 2: LeetCode 135. 分发糖果
- \* 题目链接: <https://leetcode.cn/problems/candy/>
- \* 题目描述: n 个孩子站成一排。给你一个整数数组 ratings 表示每个孩子的评分。
- \* 你需要按照以下要求, 给这些孩子分发糖果:
- \* 每个孩子至少分配到 1 个糖果。
- \* 相邻两个孩子评分更高的孩子会获得更多的糖果。
- \* 请你给每个孩子分发糖果, 计算并返回需要准备的最少糖果数目。
- \* 解题思路: 贪心算法, 两次遍历
- \* 时间复杂度: O(n)
- \* 空间复杂度: O(n)
- \* 是否最优解: 是, 这是处理此类问题的最优解法
- \*
- \* 算法详解:
- \* 1. 两次遍历: 从左到右和从右到左
- \* 2. 左遍历: 确保右评分高孩子获得更多糖果
- \* 3. 右遍历: 确保左评分高孩子获得更多糖果
- \* 4. 结果合并: 取两次遍历结果的最大值
- \*/

```

public static int candy(int[] ratings) {
    int n = ratings.length;
    // 边界条件: 没有孩子时不需要糖果
    if (n == 0) return 0;

    // 从左到右遍历, 确保右边评分高的孩子比左边的获得更多糖果
    int[] left = new int[n];
    // 初始化每个孩子至少 1 个糖果
    Arrays.fill(left, 1);
    for (int i = 1; i < n; i++) {
        // 如果当前孩子评分高于左边孩子
        if (ratings[i] > ratings[i - 1]) {
            // 当前孩子糖果数 = 左边孩子糖果数 + 1
            left[i] = left[i - 1] + 1;
        }
    }

    // 从右到左遍历, 确保左边评分高的孩子比右边的获得更多糖果
    int[] right = new int[n];
    // 初始化每个孩子至少 1 个糖果
    Arrays.fill(right, 1);
    for (int i = n - 2; i >= 0; i--) {
        // 如果当前孩子评分高于右边孩子
        if (ratings[i] > ratings[i + 1]) {
            // 当前孩子糖果数 = 右边孩子糖果数 + 1
            right[i] = right[i + 1] + 1;
        }
    }

    // 取两次遍历结果的最大值
    int result = 0;
    for (int i = 0; i < n; i++) {
        result = Math.max(result, Math.max(left[i], right[i]));
    }
    return result;
}

```

```

        right[i] = right[i + 1] + 1;
    }
}

// 取两次遍历结果的最大值，确保同时满足两个方向的约束
int total = 0;
for (int i = 0; i < n; i++) {
    // 每个孩子获得的糖果数取 left 和 right 中的最大值
    total += Math.max(left[i], right[i]);
}

return total; // 返回需要准备的最少糖果数目
}
}
=====

文件: Code06_AssignCookies.cpp
=====

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#ifndef NULL
#define NULL 0
#endif

// 替代 fmax 函数
// 算法思路: 比较两个 double 值, 返回较大的一个
// 时间复杂度: O(1)
// 空间复杂度: O(1)
double my_fmax(double a, double b) {
    return (a > b) ? a : b;
}

// 替代 strlen 函数
// 算法思路: 遍历字符串计算长度
// 时间复杂度: O(n), n 为字符串长度
// 空间复杂度: O(1)
int my_strlen(const char* str) {
    int len = 0;
    while (str[len] != '\0') {
        len++;
    }
}
```

```

    }

    return len;
}

// 自定义内存分配函数
// 算法思路：封装系统 malloc 函数，便于统一管理内存分配
// 时间复杂度：O(1)
// 空间复杂度：O(1)
void* my_malloc(size_t size) {
    return malloc(size);
}

// 自定义内存释放函数
// 算法思路：封装系统 free 函数，便于统一管理内存释放
// 时间复杂度：O(1)
// 空间复杂度：O(1)
void my_free(void* ptr) {
    free(ptr);
}

// 替代 sort 函数，使用冒泡排序对二维数组按第二个元素升序排序
// 算法思路：冒泡排序，比较相邻元素的第二个值，如果前面的大则交换
// 时间复杂度：O(n^2)，n 为数组长度
// 空间复杂度：O(1)
void my_sort_points(int** points, int pointsSize) {
    for (int i = 0; i < pointsSize - 1; i++) {
        for (int j = 0; j < pointsSize - i - 1; j++) {
            if (points[j][1] > points[j + 1][1]) {
                // 交换指针
                int* temp = points[j];
                points[j] = points[j + 1];
                points[j + 1] = temp;
            }
        }
    }
}

// 替代 sort 函数，用于 people 数组排序（身高降序，k 升序）
// 算法思路：冒泡排序，按身高降序排列，身高相同时按 k 值升序排列
// 时间复杂度：O(n^2)，n 为数组长度
// 空间复杂度：O(1)
void my_sort_people(int** people, int peopleSize) {
    for (int i = 0; i < peopleSize - 1; i++) {

```

```

        for (int j = 0; j < peopleSize - i - 1; j++) {
            // 身高降序, k 升序
            if (people[j][0] < people[j + 1][0] ||
                (people[j][0] == people[j + 1][0] && people[j][1] > people[j + 1][1])) {
                // 交换指针
                int* temp = people[j];
                people[j] = people[j + 1];
                people[j + 1] = temp;
            }
        }
    }
}

```

```

// 分发饼干 (Assign Cookies)
// 假设你是一位很棒的家长，想要给你的孩子们一些小饼干。但是，每个孩子最多只能给一块饼干。
// 对每个孩子 i，都有一个胃口值 g[i]，这是能让孩子们满足胃口的饼干的最小尺寸；
// 并且每块饼干 j，都有一个尺寸 s[j] 。
// 如果 s[j] >= g[i]，我们可以将这个饼干 j 分配给孩子 i，这个孩子会得到满足。
// 你的目标是尽可能满足越多数量的孩子，并输出这个最大数值。
//
// 算法标签：贪心算法(Greedy Algorithm)、双指针(Two Pointers)、排序(Sorting)
// 时间复杂度：O(m*log(m) + n*log(n))，其中 m 是孩子数量，n 是饼干数量
// 空间复杂度：O(1)，仅使用常数额外空间
// 测试链接：https://leetcode.cn/problems/assign-cookies/
// 相关题目：LeetCode 452. 用最少数量的箭引爆气球、LeetCode 763. 划分字母区间
// 贪心算法专题 - 补充题目收集与详解

```

```

/*
 * 算法思路：
 * 1. 贪心策略：优先满足胃口小的孩子
 * 2. 将孩子胃口数组和饼干尺寸数组都排序
 * 3. 用双指针分别遍历两个数组
 * 4. 当前饼干能满足当前孩子时，两个指针都后移
 * 5. 当前饼干不能满足当前孩子时，饼干指针后移，寻找更大的饼干
 *
 * 时间复杂度：O(m * logm + n * logn) - m 是孩子数量，n 是饼干数量，主要是排序的时间复杂度
 * 空间复杂度：O(1) - 只使用了常数额外空间
 * 是否最优解：是，这是处理此类问题的最优解法
 *
 * 工程化考量：
 * 1. 异常处理：检查输入是否为空
 * 2. 边界条件：处理空数组、单个元素等特殊情况
 * 3. 性能优化：使用双指针避免重复遍历

```

- \* 4. 可读性: 清晰的变量命名和注释
- \*
- \* 极端场景与边界场景:
  - \* 1. 空输入: g 或 s 为空数组
  - \* 2. 极端值: 只有一个孩子或一块饼干
  - \* 3. 重复数据: 多个孩子胃口相同或多个饼干尺寸相同
  - \* 4. 有序/逆序数据: 孩子胃口和饼干尺寸都已排序
- \*
- \* 跨语言场景与语言特性差异:
  - \* 1. Java: 使用 Arrays.sort 进行排序
  - \* 2. C++: 使用 std::sort 进行排序
  - \* 3. Python: 使用 sorted 函数或 list.sort() 方法
- \*
- \* 调试能力构建:
  - \* 1. 打印中间过程: 在循环中打印指针位置和当前匹配情况
  - \* 2. 用断言验证中间结果: 确保每次匹配都满足  $s[j] \geq g[i]$
  - \* 3. 性能退化排查: 检查排序和遍历的时间复杂度
- \*
- \* 与机器学习、图像处理、自然语言处理的联系与应用:
  - \* 1. 在资源分配问题中, 贪心算法可以作为初始解提供给更复杂的优化算法
  - \* 2. 在推荐系统中, 可以使用贪心策略为用户推荐最匹配的物品
  - \* 3. 在图像处理中, 贪心算法可用于图像分割的初始区域划分

```
// 简单的排序函数实现（冒泡排序）
// 算法思路: 冒泡排序, 比较相邻元素, 如果前面的大则交换
// 时间复杂度: O(n^2), n 为数组长度
// 空间复杂度: O(1)
void bubbleSort(int arr[], int n) {
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                // 交换元素
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
}
```

```
// 分发饼干主函数
// 算法思路: 贪心算法, 优先满足胃口小的孩子
```

```

// 1. 对孩子胃口数组和饼干尺寸数组排序
// 2. 使用双指针遍历两个数组
// 3. 当前饼干能满足当前孩子时，两个指针都后移
// 4. 当前饼干不能满足当前孩子时，饼干指针后移，寻找更大的饼干
// 时间复杂度: O(m*log(m) + n*log(n)), 主要是排序的时间复杂度
// 空间复杂度: O(1)

int findContentChildren(int g[], int gSize, int s[], int sSize) {
    // 异常处理: 检查输入是否为空
    if (g == 0 || s == 0 || gSize == 0 || sSize == 0) {
        return 0;
    }

    // 使用冒泡排序对孩子胃口数组和饼干尺寸数组排序
    bubbleSort(g, gSize);
    bubbleSort(s, sSize);

    int child = 0;      // 孩子指针
    int cookie = 0;     // 饼干指针

    // 双指针遍历
    while (child < gSize && cookie < sSize) {
        // 如果当前饼干能满足当前孩子
        if (s[cookie] >= g[child]) {
            child++;    // 满足的孩子数加 1
        }
        cookie++;      // 饼干指针后移
    }

    return child;      // 返回满足的孩子数
}

// 补充题目 1: LeetCode 452. 用最少量的箭引爆气球
// 题目描述: 有一些球形气球贴在一堵用 XY 平面表示的墙面上。墙面上的气球记录在整数数组 points,
// 其中 points[i] = [xstart, xend] 表示水平直径在 xstart 和 xend 之间的气球。
// 你不知道气球的确切 y 坐标。一支弓箭可以沿着 x 轴从不同点完全垂直地射出。
// 在坐标 x 处射出一支箭，若有一个气球的直径的开始和结束坐标为 xstart, xend,
// 且满足 xstart ≤ x ≤ xend，则该气球会被引爆。
// 可以射出的弓箭的数量没有限制。弓箭一旦被射出之后，可以无限地前进。
// 我们想找到使得所有气球全部被引爆，所需的弓箭的最小数量。
// 链接: https://leetcode.cn/problems/minimum-number-of-arrows-to-burst-balloons/

// 比较函数，用于排序二维数组（按第二个元素升序）
// 算法思路：比较两个一维数组的第二个元素

```

```

// 时间复杂度: O(1)
// 空间复杂度: O(1)
bool comparePoints(const int* a, const int* b) {
    return a[1] < b[1];
}

// 用最少量的箭引爆气球
// 算法思路: 贪心算法
// 1. 按气球结束位置排序
// 2. 贪心策略: 尽可能用一支箭射更多的气球
// 3. 当前气球的开始位置大于箭的位置时, 需要新的箭
// 时间复杂度: O(n*log(n)), 主要是排序的时间复杂度
// 空间复杂度: O(1)

int findMinArrowShots(int** points, int pointsSize, int* pointsColSize) {
    if (points == NULL || pointsSize == 0) {
        return 0;
    }

    // 按气球结束位置排序
    my_sort_points(points, pointsSize);

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

    // 贪心策略: 尽可能用一支箭射更多的气球
    for (int i = 1; i < pointsSize; i++) {
        if (points[i][0] > end) {
            // 需要新的箭
            arrows++;
            end = points[i][1];
        }
    }

    return arrows;
}

// 补充题目 2: LeetCode 763. 划分字母区间
// 题目描述: 字符串 S 由小写字母组成。我们要把这个字符串划分为尽可能多的片段，同一字母最多出现在一个片段中。返回一个表示每个字符串片段的长度的列表。
// 链接: https://leetcode.cn/problems/partition-labels/

// 划分字母区间
// 算法思路: 贪心算法

```

```

// 1. 记录每个字符最后出现的位置
// 2. 贪心策略：扩展片段直到包含所有字符的最后出现位置
// 3. 当前索引等于结束位置时，找到一个完整片段
// 时间复杂度：O(n)，n 为字符串长度
// 空间复杂度：O(1)

int* partitionLabels(char * s, int* returnSize) {
    *returnSize = 0;
    if (s == NULL || my_strlen(s) == 0) {
        return NULL;
    }

    // 记录每个字符最后出现的位置
    int lastPos[26] = {0};
    for (int i = 0; i < my_strlen(s); i++) {
        lastPos[s[i] - 'a'] = i;
    }

    // 动态分配结果数组
    int* result = (int*)my_malloc(sizeof(int) * my_strlen(s)); // 最大可能有 my_strlen(s) 个片段
    int start = 0, end = 0;

    // 贪心策略：扩展片段直到包含所有字符的最后出现位置
    for (int i = 0; i < my_strlen(s); i++) {
        end = my_fmax(end, lastPos[s[i] - 'a']);
        if (i == end) {
            // 找到一个完整片段
            result[(*returnSize)++] = end - start + 1;
            start = end + 1;
        }
    }

    // 重新分配内存以节省空间
    int* new_result = (int*)my_malloc(sizeof(int) * (*returnSize));
    for (int i = 0; i < *returnSize; i++) {
        new_result[i] = result[i];
    }
    my_free(result);
    result = new_result;
    return result;
}

// 补充题目 3：LeetCode 135. 分发糖果
// 题目描述：n 个孩子站成一排。给你一个整数数组 ratings 表示每个孩子的评分。

```

```
// 你需要按照以下要求，给这些孩子分发糖果：  
// 1. 每个孩子至少分配到 1 个糖果。  
// 2. 相邻两个孩子评分更高的孩子会获得更多的糖果。  
// 请你给每个孩子分发糖果，计算并返回需要准备的最少糖果数目。  
// 链接: https://leetcode.cn/problems/candy/  
  
// 分发糖果  
// 算法思路：贪心算法  
// 1. 从左到右：处理右孩子评分高于左孩子的情况  
// 2. 从右到左：处理左孩子评分高于右孩子的情况，取较大值  
// 3. 计算总和  
// 时间复杂度：O(n)，n 为孩子数量  
// 空间复杂度：O(n)  
  
int candy(int* ratings, int ratingsSize) {  
    if (ratings == NULL || ratingsSize == 0) {  
        return 0;  
    }  
  
    int n = ratingsSize;  
    int* candies = (int*)my_malloc(sizeof(int) * n);  
  
    // 初始化：每个孩子至少 1 个糖果  
    for (int i = 0; i < n; i++) {  
        candies[i] = 1;  
    }  
  
    // 从左到右：处理右孩子评分高于左孩子的情况  
    for (int i = 1; i < n; i++) {  
        if (ratings[i] > ratings[i - 1]) {  
            candies[i] = candies[i - 1] + 1;  
        }  
    }  
  
    // 从右到左：处理左孩子评分高于右孩子的情况，取较大值  
    for (int i = n - 2; i >= 0; i--) {  
        if (ratings[i] > ratings[i + 1]) {  
            candies[i] = my_fmax(candies[i], candies[i + 1] + 1);  
        }  
    }  
  
    // 计算总和  
    int total = 0;  
    for (int i = 0; i < n; i++) {
```

```

        total += candies[i];
    }

    my_free(candies);
    return total;
}

// 补充题目 4: LeetCode 406. 根据身高重建队列
// 题目描述: 假设有打乱顺序的一群人站成一个队列, 数组 people 表示队列中一些人的属性 (不一定按顺序)。
// 每个 people[i] = [hi, ki] 表示第 i 个人的身高为 hi , 前面 正好 有 ki 个身高大于或等于 hi 的人。
// 请你重新构造并返回输入数组 people 所表示的队列。
// 返回的队列应该格式化为数组 queue , 其中 queue[j] = [hj, kj] 是队列中第 j 个人的属性 (queue[0] 是排在队列前面的人)。
// 链接: https://leetcode.cn/problems/queue-reconstruction-by-height/

// 比较函数, 用于排序二维数组
// 算法思路: 按身高降序, k 升序排序
// 时间复杂度: O(1)
// 空间复杂度: O(1)
bool comparePeople(const int* a, const int* b) {
    if (a[0] != b[0]) {
        return b[0] < a[0]; // 身高降序
    } else {
        return a[1] < b[1]; // k 升序
    }
}

// 根据身高重建队列
// 算法思路: 贪心算法
// 1. 按身高降序, k 升序排序
// 2. 贪心策略: 按排序后的顺序插入到指定位置
// 时间复杂度: O(n^2), n 为人数
// 空间复杂度: O(n)
int** reconstructQueue(int** people, int peopleSize, int* peopleColSize, int* returnSize, int** returnColumnSizes) {
    *returnSize = 0;
    if (people == NULL || peopleSize == 0) {
        return NULL;
    }

    // 按身高降序, k 升序排序

```

```

my_sort_people(people, peopleSize);

// 初始化结果数组
int** result = (int**)my_malloc(sizeof(int*) * peopleSize);
*returnColumnSizes = (int*)my_malloc(sizeof(int) * peopleSize);

// 贪心策略：按排序后的顺序插入到指定位置
for (int i = 0; i < peopleSize; i++) {
    // 为当前人分配空间
    result[i] = (int*)my_malloc(sizeof(int) * 2);
    (*returnColumnSizes)[i] = 2;
}

// 插入排序
for (int i = 0; i < peopleSize; i++) {
    int pos = people[i][1];
    // 后移元素
    for (int j = (*returnSize); j > pos; j--) {
        result[j][0] = result[j - 1][0];
        result[j][1] = result[j - 1][1];
    }
    // 插入当前人
    result[pos][0] = people[i][0];
    result[pos][1] = people[i][1];
    (*returnSize)++;
}

return result;
}

// 补充题目 5: LeetCode 871. 最低加油次数
// 题目描述：汽车从起点出发驶向目的地，该目的地位于出发位置东面 target 英里处。
// 沿途有加油站，每个 station[i] 代表一个加油站，位于出发位置东面 station[i][0] 英里处，
// 并且有 station[i][1] 升汽油。
// 假设汽车油箱的容量是无限的，其中最初有 startFuel 升燃料。
// 它每行驶 1 英里就会用掉 1 升汽油。
// 当汽车到达加油站时，它可能停下来加油，将所有汽油从加油站转移到汽车中。
// 为了到达目的地，汽车所必要的最低加油次数是多少？如果无法到达目的地，则返回 -1。
// 链接: https://leetcode.cn/problems/minimum-number-of-refueling-stops/

// 最大堆的比较函数（使用数组实现最大堆）
class MaxHeap {
private:

```

```
int heap[1000]; // 假设最多 1000 个元素
int size;
public:
    MaxHeap() : size(0) {}

    void push(int val) {
        heap[size] = val;
        size++;
        // 向上调整
        int i = size - 1;
        while (i > 0) {
            int parent = (i - 1) / 2;
            if (heap[parent] < heap[i]) {
                // 交换元素
                int temp = heap[parent];
                heap[parent] = heap[i];
                heap[i] = temp;
                i = parent;
            } else {
                break;
            }
        }
    }

    int pop() {
        if (size == 0) return -1;
        int top = heap[0];
        heap[0] = heap[size - 1];
        size--;
        // 向下调整
        int i = 0;
        while (true) {
            int left = 2 * i + 1;
            int right = 2 * i + 2;
            int largest = i;
            if (left < size && heap[left] > heap[largest]) {
                largest = left;
            }
            if (right < size && heap[right] > heap[largest]) {
                largest = right;
            }
            if (largest != i) {
                // 交换元素
                int temp = heap[largest];
                heap[largest] = heap[i];
                heap[i] = temp;
            }
        }
    }
}
```

```

        int temp = heap[i];
        heap[i] = heap[largest];
        heap[largest] = temp;
        i = largest;
    } else {
        break;
    }
}

return top;
}

bool empty() {
    return size == 0;
}
};

// 最低加油次数
// 算法思路：贪心算法 + 最大堆
// 1. 使用最大堆存储经过的加油站的汽油量
// 2. 如果油量不足，从堆中选择油量最多的加油站加油
// 3. 处理从最后一个加油站到目的地的情况
// 时间复杂度：O(n*log(n))，n 为加油站数量
// 空间复杂度：O(n)

int minRefuelStops(int target, int startFuel, int** stations, int stationsSize, int* stationsColSize) {
    // 最大堆，存储经过的加油站的汽油量
    MaxHeap maxHeap;

    long long fuel = startFuel; // 当前油量，使用 long long 避免溢出
    int stops = 0; // 加油次数
    long long prev = 0; // 上一个位置

    // 处理所有加油站
    for (int i = 0; i < stationsSize; i++) {
        long long location = stations[i][0];
        int gas = stations[i][1];

        // 从当前位置到加油站需要的油量
        fuel -= location - prev;

        // 如果油量不足，需要从之前经过的加油站中选择油量最多的加油
        while (fuel < 0 && !maxHeap.empty()) {
            fuel += maxHeap.pop();
            stops++;
        }
    }
}
```

```

    }

    // 如果无法到达当前加油站，返回-1
    if (fuel < 0) {
        return -1;
    }

    // 将当前加油站的油量加入堆中
    maxHeap.push(gas);
    prev = location;
}

// 处理从最后一个加油站到目的地
fuel -= target - prev;
while (fuel < 0 && !maxHeap.empty()) {
    fuel += maxHeap.pop();
    stops++;
}

return fuel >= 0 ? stops : -1;
}

```

=====

文件: Code06\_AssignCookies.java

=====

```

package class094;

import java.util.Arrays;

// 分发饼干 (Assign Cookies)
// 假设你是一位很棒的家长，想要给你的孩子们一些小饼干。但是，每个孩子最多只能给一块饼干。
// 对每个孩子 i，都有一个胃口值 g[i]，这是能让孩子们满足胃口的饼干的最小尺寸；
// 并且每块饼干 j，都有一个尺寸 s[j] 。
// 如果 s[j] >= g[i]，我们可以将这个饼干 j 分配给孩子 i，这个孩子会得到满足。
// 你的目标是尽可能满足越多数量的孩子，并输出这个最大数值。
//
// 算法标签：贪心算法(Greedy Algorithm)、双指针(Two Pointers)、排序(Sorting)
// 时间复杂度: O(m*log(m) + n*log(n)), 其中 m 是孩子数量, n 是饼干数量
// 空间复杂度: O(1), 仅使用常数额外空间
// 测试链接 : https://leetcode.cn/problems/assign-cookies/
// 相关题目: LeetCode 452. 用最少量的箭引爆气球、LeetCode 763. 划分字母区间
// 贪心算法专题 - 补充题目收集与详解

```

```
/*
 * 题目解析:
 * 这是一个经典的贪心算法应用场景——资源分配问题。要求在有限的资源（饼干）下满足尽可能多的需求（孩子）。
 * 问题的核心在于如何制定最优的分配策略以最大化满足的孩子数。
 *
 * 关键约束条件:
 * 1. 每个孩子最多只能分配一块饼干
 * 2. 饼干尺寸必须大于等于孩子胃口值才能满足该孩子
 * 3. 目标是最优化满足的孩子总数
 *
 * 解题思路与策略:
 * 1. 贪心策略核心思想: 优先满足胃口小的孩子, 这样能够节省大饼干去满足胃口大的孩子
 * 2. 排序预处理: 对孩子胃口数组 g 和饼干尺寸数组 s 都进行升序排序
 * 3. 双指针匹配: 使用双指针技术分别遍历孩子和饼干数组, 实现高效的匹配过程
 * 4. 资源优化利用: 通过贪心策略避免资源浪费, 确保每块饼干都能发挥最大价值
 *
 * 算法正确性证明:
 * 贪心选择性质: 对于胃口最小的孩子, 选择能满足他的最小饼干是优选择
 * 最优子结构: 在满足了一个孩子后, 剩余问题仍具有相同的最优子结构性质
 */
```

```
public class Code06_AssignCookies {
```

```
/*
 * 算法思路详解:
 * 1. 贪心策略核心: 优先满足胃口小的孩子, 这样能够保留大饼干给胃口大的孩子
 * 2. 排序预处理: 对两个数组进行升序排序, 为贪心策略实施奠定基础
 * 3. 双指针遍历: 使用 child 指针遍历孩子数组, cookie 指针遍历饼干数组
 * 4. 匹配逻辑: 当  $s[cookie] \geq g[child]$  时, 当前饼干可满足当前孩子
 * 5. 指针更新规则:
 *   - 若能匹配: child 和 cookie 指针都向前移动一位
 *   - 若不能匹配: 仅 cookie 指针向前移动, 寻找更大尺寸的饼干
 *
 * 时间复杂度分析:
 * - 排序阶段:  $O(m \log m + n \log n)$ , 其中 m 是孩子数量, n 是饼干数量
 * - 遍历阶段:  $O(m + n)$ , 双指针只需各遍历一次
 * - 总体时间复杂度:  $O(m \log m + n \log n)$ 
 *
 * 空间复杂度分析:
 * -  $O(1)$ , 仅使用了常数级别的额外空间存储指针和临时变量
 * 是否最优解: 是, 基于贪心策略和排序的解决方案是处理此类问题的最优方法
 *
 * 工程化最佳实践:
```

- \* 1. 输入验证: 严格检查输入参数的有效性, 防止空指针异常
  - \* 2. 边界处理: 妥善处理各种边界情况, 如空数组、单元素等
  - \* 3. 性能优化: 采用双指针技术避免不必要的重复遍历
  - \* 4. 代码可读性: 使用语义明确的变量名和详尽的注释
  - \* 5. 内存管理: 避免创建不必要的对象和数组
- \*
- \* 极端场景与边界情况处理:
  - \* 1. 空输入场景: g 或 s 为空数组时直接返回 0
  - \* 2. 极值场景: 只有一个孩子或一块饼干的特殊情况
  - \* 3. 重复数据场景: 多个孩子胃口相同或多个饼干尺寸相同时的处理
  - \* 4. 特殊序列场景: 孩子胃口和饼干尺寸均为有序或逆序的情况
  - \* 5. 不均匀分布场景: 孩子胃口跨度大而饼干尺寸集中的情况
- \*
- \* 跨语言实现差异与优化:
  - \* 1. Java 实现: 使用 Arrays.sort 进行原地排序, 时间复杂度稳定
  - \* 2. C++ 实现: 使用 std::sort, 通常采用内省排序(Introsort)算法
  - \* 3. Python 实现: 使用内置 sorted 函数或 list.sort() 方法, 基于 Timsort 算法
  - \* 4. 内存管理: 不同语言的垃圾回收机制对性能的影响
- \*
- \* 调试与测试策略:
  - \* 1. 过程可视化: 在关键节点打印指针状态和匹配结果
  - \* 2. 断言验证: 在循环体内添加断言确保算法不变式成立
  - \* 3. 性能监控: 跟踪排序和遍历的实际执行时间
  - \* 4. 边界测试: 设计覆盖所有边界条件的测试用例
  - \* 5. 压力测试: 使用大规模数据验证算法稳定性
- \*
- \* 实际应用场景与拓展:
  - \* 1. 资源调度: CPU 任务分配、内存分配等系统级资源调度问题
  - \* 2. 电商推荐: 商品库存与用户需求的最佳匹配
  - \* 3. 教育资源分配: 课程容量与学生选课偏好的合理安排
  - \* 4. 医疗资源分配: 病床数量与患者病情紧急程度的匹配
  - \* 5. 交通运输: 车辆载重与货物重量的最优装载方案
- \*
- \* 算法深入解析:
  - \* 1. 排序策略原理: 升序排列确保优先处理小胃口和小尺寸
  - \* 2. 匹配原则依据: 小饼干优先满足小胃口避免资源浪费
  - \* 3. 指针移动逻辑:
    - \* - 成功匹配: child++, cookie++ (当前孩子已被满足)
    - \* - 匹配失败: 仅 cookie++ (寻找更大饼干满足当前孩子)
  - \* 4. 结果统计机制: child 指针的位置即为满足的孩子数量
- \*/

```
public static int findContentChildren(int[] g, int[] s) {  
    // 异常处理: 检查输入是否为空
```

```

if (g == null || s == null) {
    return 0;
}

// 边界条件：没有孩子或没有饼干
if (g.length == 0 || s.length == 0) {
    return 0;
}

// 排序孩子胃口数组和饼干尺寸数组
// 贪心策略的基础：排序后优先满足小胃口和使用小饼干
Arrays.sort(g); // 对孩子胃口值升序排列
Arrays.sort(s); // 对饼干尺寸升序排列

int child = 0;      // 孩子指针，指向当前待满足的孩子
int cookie = 0;     // 饼干指针，指向当前待分配的饼干

// 双指针遍历
// 当还有孩子未尝试满足且还有饼干未尝试分配时继续循环
while (child < g.length && cookie < s.length) {
    // 如果当前饼干能满足当前孩子
    // 贪心策略：优先满足能用当前饼干满足的孩子
    if (s[cookie] >= g[child]) {
        child++; // 满足的孩子数加 1，移动到下一个孩子
    }
    cookie++; // 饼干指针后移，尝试下一块饼干
}

return child; // 返回满足的孩子数
}

// 测试函数
public static void main(String[] args) {
    // 测试用例 1：基本情况
    // 3 个孩子胃口值为 [1, 2, 3]，2 块饼干尺寸为 [1, 1]
    // 只能满足胃口为 1 的孩子
    int[] g1 = {1, 2, 3};
    int[] s1 = {1, 1};
    System.out.println("测试用例 1 结果：" + findContentChildren(g1, s1)); // 期望输出：1

    // 测试用例 2：资源充足
    // 2 个孩子胃口值为 [1, 2]，3 块饼干尺寸为 [1, 2, 3]
    // 能满足所有孩子
}

```

```

int[] g2 = {1, 2};
int[] s2 = {1, 2, 3};
System.out.println("测试用例 2 结果: " + findContentChildren(g2, s2)); // 期望输出: 2

// 测试用例 3: 边界情况
// 没有孩子, 有饼干
// 应该返回 0
int[] g3 = {};
int[] s3 = {1, 2, 3};
System.out.println("测试用例 3 结果: " + findContentChildren(g3, s3)); // 期望输出: 0

// 测试用例 4: 极端情况
// 孩子胃口都很大, 饼干尺寸相对较小
// 只能满足胃口为 7 和 8 的孩子
int[] g4 = {10, 9, 8, 7};
int[] s4 = {5, 6, 7, 8};
System.out.println("测试用例 4 结果: " + findContentChildren(g4, s4)); // 期望输出: 2
}

```

// 补充题目 1: LeetCode 452. 用最少量的箭引爆气球 (Minimum Number of Arrows to Burst Balloons)

// 题目描述: 有一些球形气球贴在一堵用 XY 平面表示的墙面上。墙面上的气球记录在整数数组 points，  
// 其中 points[i] = [xstart, xend] 表示水平直径在 xstart 和 xend 之间的气球。  
// 你不知道气球的确切 y 坐标。一支弓箭可以沿着 x 轴从不同点完全垂直地射出。  
// 在坐标 x 处射出一支箭，若有一个气球的直径的开始和结束坐标为 xstart, xend，  
// 且满足  $xstart \leq x \leq xend$ ，则该气球会被引爆。  
// 可以射出的弓箭的数量没有限制。弓箭一旦被射出之后，可以无限地前进。  
// 我们想找到使得所有气球全部被引爆，所需的弓箭的最少量。  
//  
// 算法标签: 贪心算法(Greedy Algorithm)、区间问题(Interval Problem)、排序(Sorting)  
// 时间复杂度:  $O(n \log n)$ ，其中 n 是气球数量  
// 空间复杂度:  $O(1)$ ，仅使用常数额外空间  
// 链接: <https://leetcode.cn/problems/minimum-number-of-arrows-to-burst-balloons/>  
/\*  
\* 算法详解与策略分析:  
\* 1. 贪心策略核心思想: 按结束位置排序，优先处理结束位置早的气球，  
\* 这样可以保证箭射出的位置尽可能多地引爆重叠的气球  
\* 2. 区间覆盖原理: 通过一支箭引爆多个重叠区间，实现资源的最大化利用  
\* 3. 优化选择机制: 当遇到不重叠的气球区间时，必须使用新箭  
\* 4. 结果统计方法: 通过计数器记录所需箭的数量  
\*  
\* 算法步骤详解:  
\* 1. 预处理: 按气球结束位置升序排序

- \* 2. 初始化：设置箭数计数器为 1，记录第一支箭的最远射程
- \* 3. 遍历过程：
  - 若当前气球起始位置超出箭的最远射程：需要新箭
  - 若当前气球起始位置在箭的射程内：当前箭可引爆
- \* 4. 结果返回：返回累计的箭数
- \*
- \* 算法正确性证明：
- \* 贪心选择性质：选择结束位置最早的气球作为当前处理对象是最优的
- \* 最优子结构：在确定一支箭的位置后，剩余问题仍具有相同的最优子结构
- \*/

```

public static int findMinArrowShots(int[][] points) {
    // 异常处理：检查输入是否为空
    if (points == null || points.length == 0) {
        return 0;
    }

    // 按气球结束位置排序
    // 贪心策略的基础：按结束位置升序排列
    Arrays.sort(points, (a, b) -> Integer.compare(a[1], b[1]));

    int arrows = 1;      // 箭数计数器，至少需要 1 支箭
    int end = points[0][1]; // 当前箭能射到的最远位置

    // 贪心策略：尽可能用一支箭射更多的气球
    // 遍历所有气球
    for (int i = 1; i < points.length; i++) {
        // 如果当前气球的起始位置大于箭的最远射程
        // 说明需要新的箭来射这个气球
        if (points[i][0] > end) {
            // 需要新的箭
            arrows++;           // 箭数加 1
            end = points[i][1]; // 更新箭的最远射程
        }
        // 如果当前气球的起始位置小于等于箭的最远射程
        // 说明当前箭可以射到这个气球，不需要额外操作
    }

    return arrows; // 返回需要的最少箭数
}

// 补充题目 2：LeetCode 763. 划分字母区间（Partition Labels）
// 题目描述：字符串 S 由小写字母组成。我们要把这个字符串划分为尽可能多的片段，同一字母最多出现在一个片段中。返回一个表示每个字符串片段的长度的列表。

```

```

// 算法标签: 贪心算法(Greedy Algorithm)、字符串处理(String Processing)、哈希表(HashMap)
// 时间复杂度: O(n)，其中 n 是字符串长度
// 空间复杂度: O(1)，使用固定大小数组存储字符位置
// 链接: https://leetcode.cn/problems/partition-labels/
/*
 * 算法详解与策略分析:
 * 1. 预处理优化: 通过一次遍历记录每个字符最后出现的位置,
 *   为后续贪心决策提供数据支持
 * 2. 贪心策略核心: 动态扩展当前片段边界, 确保包含所有相关字符
 * 3. 分段判断条件: 当遍历到当前位置等于当前片段最远边界时进行分段
 * 4. 结果收集机制: 计算并记录每个片段的长度
 *
 * 算法步骤详解:
 * 1. 预处理阶段: 遍历字符串, 记录每个字符最后出现的位置
 * 2. 贪心扩展过程:
 *   - 维护当前片段的起始和结束位置
 *   - 动态更新结束位置为当前字符最后位置与当前结束位置的最大值
 *   - 当遍历位置等于结束位置时, 完成一个片段的划分
 * 3. 结果生成: 计算片段长度(end-start+1)并添加到结果列表
 *
 * 算法正确性分析:
 * 贪心选择性质: 每次选择能包含所有相关字符的最短片段是最优的
 * 最优子结构: 划分完一个片段后, 剩余字符串的最优划分仍保持最优性
 */

public static java.util.List<Integer> partitionLabels(String s) {
    // 结果列表, 存储每个片段的长度
    java.util.List<Integer> result = new java.util.ArrayList<>();
    // 异常处理: 检查输入是否为空
    if (s == null || s.isEmpty()) {
        return result;
    }

    // 记录每个字符最后出现的位置
    // 数组索引对应字母 a-z, 值为该字母最后出现的位置
    int[] lastPos = new int[26];
    for (int i = 0; i < s.length(); i++) {
        // 记录字符 s.charAt(i) 最后出现的位置
        lastPos[s.charAt(i) - 'a'] = i;
    }

    int start = 0, end = 0; // 当前片段的起始和结束位置
    // 贪心策略: 扩展片段直到包含所有字符的最后出现位置

```

```

// 遍历字符串中的每个字符
for (int i = 0; i < s.length(); i++) {
    // 更新当前片段的结束位置
    // 选择当前片段结束位置和当前字符最后位置的最大值
    end = Math.max(end, lastPos[s.charAt(i) - 'a']);

    // 如果当前位置等于当前片段的结束位置
    // 说明当前片段已经包含了所有需要的字符
    if (i == end) {
        // 找到一个完整片段
        // 添加片段长度到结果列表
        result.add(end - start + 1);
        // 更新下一个片段的起始位置
        start = end + 1;
    }
}

return result; // 返回每个片段的长度列表
}

```

```

// 补充题目 3: LeetCode 135. 分发糖果 (Candy)
// 题目描述: n 个孩子站成一排。给你一个整数数组 ratings 表示每个孩子的评分。
// 你需要按照以下要求，给这些孩子分发糖果:
// 1. 每个孩子至少分配到 1 个糖果。
// 2. 相邻两个孩子评分更高的孩子会获得更多的糖果。
// 请你给每个孩子分发糖果，计算并返回需要准备的最少糖果数目。
//
// 算法标签: 贪心算法(Greedy Algorithm)、两次遍历(Two Pass)、动态规划思想(DP Thinking)
// 时间复杂度: O(n)，其中 n 是孩子数量
// 空间复杂度: O(n)，需要额外数组存储每个孩子的糖果数
// 链接: https://leetcode.cn/problems/candy/
/*
 * 算法详解与策略分析:
 * 1. 两次遍历策略: 通过从左到右和从右到左两次遍历,
 *     分别处理左右相邻约束条件
 * 2. 左遍历目的: 确保评分高的右孩子比左孩子获得更多糖果
 * 3. 右遍历目的: 确保评分高的左孩子比右孩子获得更多糖果
 * 4. 结果合并方法: 对每个孩子取两次遍历结果的最大值
 *
 * 算法步骤详解:
 * 1. 初始化阶段: 为每个孩子分配 1 个糖果作为基础
 * 2. 左遍历过程:
 *     - 从左到右检查相邻孩子

```

- 若右孩子评分高于左孩子，则右孩子糖果数=左孩子糖果数+1
- 3. 右遍历过程：
  - 从右到左检查相邻孩子
  - 若左孩子评分高于右孩子，则左孩子糖果数=max(当前值, 右孩子糖果数+1)
- 4. 结果统计：累加所有孩子的糖果数
- \*
- 算法优化与正确性：
- 贪心策略有效性：每次只考虑局部最优约束，最终达到全局最优
- 空间优化可能：可优化为 O(1) 空间复杂度的单次遍历算法
- \*/

```

public static int candy(int[] ratings) {
    // 异常处理：检查输入是否为空
    if (ratings == null || ratings.length == 0) {
        return 0;
    }

    int n = ratings.length; // 孩子数量
    // 糖果数组，存储每个孩子分到的糖果数
    int[] candies = new int[n];

    // 初始化：每个孩子至少 1 个糖果
    for (int i = 0; i < n; i++) {
        candies[i] = 1;
    }

    // 从左到右：处理右孩子评分高于左孩子的情况
    // 确保评分高的右孩子比左孩子获得更多糖果
    for (int i = 1; i < n; i++) {
        // 如果右孩子评分高于左孩子
        if (ratings[i] > ratings[i - 1]) {
            // 右孩子糖果数 = 左孩子糖果数 + 1
            candies[i] = candies[i - 1] + 1;
        }
    }

    // 从右到左：处理左孩子评分高于右孩子的情况，取较大值
    // 确保评分高的左孩子比右孩子获得更多糖果
    for (int i = n - 2; i >= 0; i--) {
        // 如果左孩子评分高于右孩子
        if (ratings[i] > ratings[i + 1]) {
            // 左孩子糖果数 = max(当前糖果数, 右孩子糖果数 + 1)
            // 取较大值确保同时满足左右两个方向的约束
            candies[i] = Math.max(candies[i], candies[i + 1] + 1);
        }
    }
}

```

```

    }

}

// 计算总和
// 统计所有孩子分到的糖果总数
int total = 0;
for (int candy : candies) {
    total += candy;
}

return total; // 返回需要准备的最少糖果数目
}

// 补充题目 4: LeetCode 406. 根据身高重建队列 (Queue Reconstruction by Height)
// 题目描述: 假设有打乱顺序的一群人站成一个队列, 数组 people 表示队列中一些人的属性 (不一定按顺序)。
// 每个 people[i] = [hi, ki] 表示第 i 个人的身高为 hi , 前面 正好 有 ki 个身高大于或等于 hi 的人。
// 请你重新构造并返回输入数组 people 所表示的队列。
// 返回的队列应该格式化为数组 queue , 其中 queue[j] = [hj, kj] 是队列中第 j 个人的属性 (queue[0] 是排在队列前面的人)。
//
// 算法标签: 贪心算法(Greedy Algorithm)、排序(Sorting)、链表插入(Linked List Insertion)
// 时间复杂度: O(n2) , 其中 n 是人数
// 空间复杂度: O(n) , 使用 List 存储结果
// 链接: https://leetcode.cn/problems/queue-reconstruction-by-height/
/*
 * 算法详解与策略分析:
 * 1. 排序策略原理: 身高降序确保先处理高个子, k 值升序确保相同身高时正确插入
 * 2. 贪心策略核心: 按排序后的顺序依次插入, 利用高个子对矮个子不可见的特性
 * 3. 插入位置确定: k 值即为在当前队列中的插入位置
 * 4. 数据结构选择: 使用 List 支持动态插入操作
 *
 * 算法步骤详解:
 * 1. 排序阶段:
 *   - 按身高降序排列
 *   - 身高相同时按 k 值升序排列
 * 2. 插入过程:
 *   - 依次处理排序后的人群
 *   - 将当前人插入到结果列表的 p[1]位置
 *   - p[1]表示前面应该有 p[1]个身高大于或等于当前人的人
 * 3. 结果生成: 将 List 转换为数组返回
 */

```

\* 算法正确性证明：

\* 贪心选择性质：先处理高个子可以确保后续插入的矮个子不影响前面的排列

\* 最优子结构：每一步插入都保持了当前队列的正确性

\*/

```
public static int[][] reconstructQueue(int[][] people) {
    // 异常处理：检查输入是否为空
    if (people == null || people.length == 0) {
        return new int[0][0];
    }

    // 按身高降序，k 升序排序
    // 贪心策略的基础：先处理高个子，再处理矮个子
    Arrays.sort(people, (a, b) -> {
        if (a[0] != b[0]) {
            return b[0] - a[0]; // 身高降序排列
        } else {
            return a[1] - b[1]; // k 值升序排列
        }
    });

    // 结果列表，使用 List 支持动态插入操作
    java.util.List<int[]> result = new java.util.ArrayList<>();

    // 贪心策略：按排序后的顺序插入到指定位置
    // 遍历排序后的人群
    for (int[] p : people) {
        // 将当前人插入到结果列表的 p[1] 位置
        // p[1] 表示前面应该有 p[1] 个身高大于或等于当前人的人
        result.add(p[1], p);
    }

    // 将 List 转换为数组并返回
    return result.toArray(new int[result.size()][]);
}
```

// 补充题目 5: LeetCode 871. 最低加油次数 (Minimum Number of Refueling Stops)

// 题目描述：汽车从起点出发驶向目的地，该目的地位于出发位置东面 target 英里处。

// 沿途有加油站，每个 station[i] 代表一个加油站，位于出发位置东面 station[i][0] 英里处，  
// 并且有 station[i][1] 升汽油。

// 假设汽车油箱的容量是无限的，其中最初有 startFuel 升燃料。

// 它每行驶 1 英里就会用掉 1 升汽油。

// 当汽车到达加油站时，它可能停下来加油，将所有汽油从加油站转移到汽车中。

// 为了到达目的地，汽车所必要的最低加油次数是多少？如果无法到达目的地，则返回 -1。

```

// 算法标签: 贪心算法(Greedy Algorithm)、优先队列(Priority Queue)、过程模拟(Simulation)
// 时间复杂度: O(n*log(n)), 其中 n 是加油站数量
// 空间复杂度: O(n), 最大堆存储经过的加油站
// 链接: https://leetcode.cn/problems/minimum-number-of-refueling-stops/
/*
 * 算法详解与策略分析:
 * 1. 贪心策略核心: 在油不够时选择油量最多的加油站进行加油,
 *   这样可以最大化单次加油带来的行驶距离
 * 2. 数据结构选择: 使用最大堆维护经过但未加油的加油站,
 *   便于快速获取油量最多的加油站
 * 3. 过程模拟机制: 模拟汽车行驶过程, 动态管理油量和加油决策
 * 4. 优化选择原理: 优先选择油量最多的加油站, 减少加油次数
 *
 * 算法步骤详解:
 * 1. 初始化阶段:
 *   - 创建最大堆存储经过的加油站油量
 *   - 初始化当前油量、加油次数和上一个位置
 * 2. 遍历过程:
 *   - 计算到达当前加油站需要消耗的油量
 *   - 若油量不足, 从堆中取出油量最多的加油站加油
 *   - 将当前加油站油量加入堆中
 * 3. 目的地处理:
 *   - 计算从最后一个加油站到目的地的油耗
 *   - 若油量不足继续从堆中加油
 * 4. 结果返回:
 *   - 若能到达目的地返回加油次数
 *   - 否则返回-1
 *
 * 算法正确性与优化:
 * 贪心选择性质: 每次选择油量最多的加油站是最优决策
 * 算法优化: 可使用动态规划方法作为替代方案进行对比
*/
public static int minRefuelStops(int target, int startFuel, int[][] stations) {
    // 最大堆, 存储经过的加油站的汽油量
    // 使用自定义比较器(a, b) -> b - a 实现最大堆
    java.util.PriorityQueue<Integer> maxHeap = new java.util.PriorityQueue<>((a, b) -> b - a);

    int fuel = startFuel; // 当前油量, 初始值为起始油量
    int stops = 0;         // 加油次数计数器
    int prev = 0;          // 上一个位置, 初始为起点

    // 处理所有加油站

```

```
// 遍历所有加油站，模拟行驶过程
for (int i = 0; i < stations.length; i++) {
    int location = stations[i][0]; // 当前加油站位置
    int gas = stations[i][1]; // 当前加油站油量

    // 从当前位置到加油站需要的油量
    // 消耗的油量 = 距离 = 当前加油站位置 - 上一个位置
    fuel -= location - prev;

    // 如果油量不足，需要从之前经过的加油站中选择油量最多的加油
    // 贪心策略：在油不够时选择油量最多的加油站
    while (fuel < 0 && !maxHeap.isEmpty()) {
        // 从堆中取出油量最多的加油站加油
        fuel += maxHeap.poll();
        stops++; // 加油次数加 1
    }

    // 如果无法到达当前加油站，返回-1
    // 说明即使把所有经过的加油站都加了油也不够
    if (fuel < 0) {
        return -1;
    }

    // 将当前加油站的油量加入堆中
    // 表示经过了这个加油站，可以在需要时加油
    maxHeap.offer(gas);
    prev = location; // 更新上一个位置
}

// 处理从最后一个加油站到目的地
// 消耗从最后一个加油站到目的地的油量
fuel -= target - prev;

// 如果油量不足，继续从堆中选择加油站加油
while (fuel < 0 && !maxHeap.isEmpty()) {
    fuel += maxHeap.poll();
    stops++;
}

// 如果最终油量非负，返回加油次数；否则返回-1
return fuel >= 0 ? stops : -1;
}
```

文件: Code06\_AssignCookies.py

```
# 分发饼干 (Assign Cookies)
# 假设你是一位很棒的家长，想要给你的孩子们一些小饼干。但是，每个孩子最多只能给一块饼干。
# 对每个孩子 i，都有一个胃口值 g[i]，这是能让孩子们满足胃口的饼干的最小尺寸；
# 并且每块饼干 j，都有一个尺寸 s[j] 。
# 如果 s[j] >= g[i]，我们可以将这个饼干 j 分配给孩子 i，这个孩子会得到满足。
# 你的目标是尽可能满足越多数量的孩子，并输出这个最大数值。
#
# 算法标签：贪心算法(Greedy Algorithm)、双指针(Two Pointers)、排序(Sorting)
# 时间复杂度：O(m*log(m) + n*log(n))，其中 m 是孩子数量，n 是饼干数量
# 空间复杂度：O(1)，仅使用常数额外空间
# 测试链接：https://leetcode.cn/problems/assign-cookies/
# 相关题目：LeetCode 452. 用最少数量的箭引爆气球、LeetCode 763. 划分字母区间
# 贪心算法专题 - 补充题目收集与详解
```

"""

算法思路详解：

1. 贪心策略：优先满足胃口小的孩子
  - 这样可以最大化满足的孩子数量
  - 小胃口的孩子更容易被满足，应该优先处理
2. 将孩子胃口数组和饼干尺寸数组都排序
  - 排序后可以使用双指针技术
  - 保证了贪心策略的正确性
3. 用双指针分别遍历两个数组
  - child 指针遍历孩子数组
  - cookie 指针遍历饼干数组
4. 当前饼干能满足当前孩子时，两个指针都后移
  - 表示这个孩子已经被满足
  - 可以尝试满足下一个孩子
5. 当前饼干不能满足当前孩子时，饼干指针后移，寻找更大的饼干
  - 当前饼干无法满足当前孩子
  - 需要寻找更大的饼干来尝试满足

时间复杂度分析：

- 排序时间复杂度：O(m\*log(m) + n\*log(n))，其中 m 是孩子数量，n 是饼干数量

- 遍历时间复杂度:  $O(m + n)$
- 总体时间复杂度:  $O(m \log(m) + n \log(n))$

空间复杂度分析:

- 只使用了常数额外空间存储指针和临时变量
- 空间复杂度:  $O(1)$

是否最优解:

- 是, 这是处理此类问题的最优解法
- 贪心策略保证了局部最优解能导致全局最优解

工程化最佳实践:

1. 异常处理: 检查输入是否为空, 避免空指针异常
2. 边界条件: 处理空数组、单个元素等特殊情况
3. 性能优化: 使用双指针避免重复遍历, 提高效率
4. 可读性: 清晰的变量命名和详细注释, 便于维护

极端场景与边界情况处理:

1. 空输入: g 或 s 为空数组时返回 0
2. 极端值: 只有一个孩子或一块饼干的情况
3. 重复数据: 多个孩子胃口相同或多个饼干尺寸相同
4. 有序/逆序数据: 孩子胃口和饼干尺寸都已排序的情况

跨语言实现差异与优化:

1. Java: 使用 Arrays.sort 进行排序, 时间复杂度稳定
2. C++: 使用 std::sort 进行排序, 底层实现可能更优化
3. Python: 使用内置 sort() 方法或 sorted() 函数, 基于 Timsort 算法

调试与测试策略:

1. 打印中间过程: 在循环中打印指针位置和当前匹配情况
2. 用断言验证中间结果: 确保每次匹配都满足  $s[j] \geq g[i]$
3. 性能退化排查: 检查排序和遍历的时间复杂度
4. 边界测试: 测试空数组、单元素等边界情况

实际应用场景与拓展:

1. 资源分配问题: 在有限资源下最大化满足的请求数量
2. 任务调度: 在有限时间内完成尽可能多的任务
3. 匹配系统: 在供需匹配中最大化匹配数量

算法深入解析:

贪心算法的核心思想是每一步都做出当前看起来最好的选择, 希望通过局部最优解达到全局最优解。

在分发饼干问题中, 我们的贪心策略是优先满足胃口小的孩子, 这样可以保证:

1. 小胃口的孩子更容易被满足, 应该优先处理

2. 大饼干可以留給大胃口的孩子，避免浪费
3. 通过排序和双指针技术，我们可以高效地实现这个策略

"""

```
def findContentChildren(g, s):  
    """  
        分发饼干主函数 - 使用贪心算法最大化满足的孩子数量  
    """
```

算法思路：

1. 对孩子胃口数组和饼干尺寸数组进行排序
2. 使用双指针技术分别遍历两个数组
3. 贪心策略：优先满足胃口小的孩子

Args:

g (List[int]): 孩子们的胃口值列表，g[i]表示第 i 个孩子的最小满足饼干尺寸  
s (List[int]): 饼干的尺寸列表，s[j]表示第 j 块饼干的尺寸

Returns:

int: 能够满足的孩子数量

时间复杂度:  $O(m \log(m) + n \log(n))$ , 其中 m 是孩子数量, n 是饼干数量

空间复杂度:  $O(1)$ , 仅使用常数额外空间

Examples:

```
>>> findContentChildren([1, 2, 3], [1, 1])
```

1

```
>>> findContentChildren([1, 2], [1, 2, 3])
```

2

"""

# 异常处理：检查输入是否为空，避免后续操作出现异常

```
if not g or not s:
```

```
    return 0
```

# 排序孩子胃口数组和饼干尺寸数组

# 时间复杂度:  $O(m \log(m) + n \log(n))$

```
g.sort()
```

```
s.sort()
```

child = 0 # 孩子指针，指向当前待满足的孩子

cookie = 0 # 饼干指针，指向当前待分配的饼干

# 双指针遍历

```
# 时间复杂度: O(m + n)
while child < len(g) and cookie < len(s):
    # 如果当前饼干能满足当前孩子
    if s[cookie] >= g[child]:
        child += 1    # 满足的孩子数加 1, 指向下一个孩子
        cookie += 1    # 饼干指针后移, 指向下一个饼干
```

```
return child # 返回满足的孩子数
```

```
# 补充题目 1: LeetCode 452. 用最少量的箭引爆气球
# 题目描述: 有一些球形气球贴在一堵用 XY 平面表示的墙面上。墙面上的气球记录在整数数组 points,
# 其中 points[i] = [xstart, xend] 表示水平直径在 xstart 和 xend 之间的气球。
# 你不知道气球的确切 y 坐标。一支弓箭可以沿着 x 轴从不同点完全垂直地射出。
# 在坐标 x 处射出一支箭, 若有一个气球的直径的开始和结束坐标为 xstart, xend,
# 且满足 xstart ≤ x ≤ xend, 则该气球会被引爆。
# 可以射出的弓箭的数量没有限制。弓箭一旦被射出之后, 可以无限地前进。
# 我们想找到使得所有气球全部被引爆, 所需的弓箭的最小数量。
# 链接: https://leetcode.cn/problems/minimum-number-of-arrows-to-burst-balloons/
```

```
def find_min_arrow_shots(points):
    """
    计算引爆所有气球所需的最少弓箭数量 - 使用贪心算法
    
```

算法思路:

1. 按气球结束位置排序
2. 贪心策略: 尽可能用一支箭射更多的气球
3. 当前气球的开始位置大于箭的位置时, 需要新的箭

Args:

points (List[List[int]]): 包含气球起始和结束坐标的列表, 每个元素为 [xstart, xend]

Returns:

int: 所需的最少弓箭数量

时间复杂度:  $O(n \log n)$ , 其中  $n$  是气球数量, 主要开销来自排序

空间复杂度:  $O(\log n)$ , 排序所需的空间

工程化考量:

1. 异常处理: 检查输入是否为空
2. 边界条件: 处理空数组、单个元素等情况
3. 性能优化: 排序后使用贪心策略避免重复计算

"""

```

# 异常处理：检查输入是否为空
if not points:
    return 0

# 按气球结束位置排序
# 关键点：按结束位置排序保证了贪心策略的正确性
points.sort(key=lambda x: x[1])

arrows = 1 # 至少需要一支箭
end = points[0][1] # 第一支箭的位置设为第一个气球的结束位置

# 贪心策略：尽可能用一支箭射更多的气球
for i in range(1, len(points)):
    # 如果当前气球的开始位置大于箭的位置，说明需要新的箭
    if points[i][0] > end:
        # 需要新的箭
        arrows += 1
        end = points[i][1] # 更新箭的位置为当前气球的结束位置

return arrows

```

```

# 补充题目 2: LeetCode 763. 划分字母区间
# 题目描述：字符串 S 由小写字母组成。我们要把这个字符串划分为尽可能多的片段，
# 同一字母最多出现在一个片段中。返回一个表示每个字符串片段的长度的列表。
# 链接: https://leetcode.cn/problems/partition-labels/

```

```

def partition_labels(s):
    """

```

将字符串划分为尽可能多的片段，同一字母最多出现在一个片段中 - 使用贪心算法

算法思路：

1. 记录每个字符最后出现的位置
2. 贪心策略：扩展片段直到包含所有字符的最后出现位置
3. 当前索引等于结束位置时，找到一个完整片段

Args:

s (str): 输入字符串，仅包含小写字母

Returns:

List[int]: 每个片段的长度列表

时间复杂度: O(n)，其中 n 是字符串长度，需要两次遍历

空间复杂度:  $O(1)$ , 只使用了固定大小的数组 (26 个字母)

工程化考量:

1. 异常处理: 检查输入是否为空
2. 边界条件: 处理空字符串、单字符等情况
3. 性能优化: 使用哈希表快速查找字符最后位置

"""

# 异常处理: 检查输入是否为空

if not s:

return []

# 记录每个字符最后出现的位置

# 时间复杂度:  $O(n)$

last\_pos = {}

for i, c in enumerate(s):

last\_pos[c] = i

result = [] # 存储结果

start = 0 # 当前片段的开始位置

end = 0 # 当前片段的结束位置

# 贪心策略: 扩展片段直到包含所有字符的最后出现位置

# 时间复杂度:  $O(n)$

for i, c in enumerate(s):

# 更新当前片段的结束位置为当前字符最后出现位置和当前结束位置的最大值

end = max(end, last\_pos[c])

# 如果当前索引等于结束位置, 说明找到了一个完整片段

if i == end:

# 找到一个完整片段

result.append(end - start + 1)

start = end + 1 # 更新下一个片段的开始位置

return result

# 补充题目 3: LeetCode 135. 分发糖果

# 题目描述:  $n$  个孩子站成一排。给你一个整数数组 ratings 表示每个孩子的评分。

# 你需要按照以下要求, 给这些孩子分发糖果:

# 1. 每个孩子至少分配到 1 个糖果。

# 2. 相邻两个孩子评分更高的孩子会获得更多的糖果。

# 请你给每个孩子分发糖果, 计算并返回需要准备的最少糖果数目。

# 链接: <https://leetcode.cn/problems/candy/>

```
def candy(ratings):
    """
    计算满足条件的最少糖果数目 - 使用贪心算法
    """
```

算法思路:

1. 从左到右: 处理右孩子评分高于左孩子的情况
2. 从右到左: 处理左孩子评分高于右孩子的情况, 取较大值
3. 计算总和

Args:

    ratings (List[int]): 每个孩子的评分数组

Returns:

    int: 需要准备的最少糖果数目

时间复杂度:  $O(n)$ , 其中  $n$  是孩子数量, 需要两次遍历

空间复杂度:  $O(n)$ , 需要额外数组存储每个孩子的糖果数

工程化考量:

1. 异常处理: 检查输入是否为空
2. 边界条件: 处理空数组、单元素等情况
3. 性能优化: 两次遍历确保满足所有约束条件

"""

# 异常处理: 检查输入是否为空

if not ratings:

    return 0

n = len(ratings)

# 初始化: 每个孩子至少 1 个糖果

candies = [1] \* n

# 从左到右: 处理右孩子评分高于左孩子的情况

# 时间复杂度:  $O(n)$

for i in range(1, n):

    if ratings[i] > ratings[i - 1]:

        candies[i] = candies[i - 1] + 1

# 从右到左: 处理左孩子评分高于右孩子的情况, 取较大值

# 时间复杂度:  $O(n)$

for i in range(n - 2, -1, -1):

    if ratings[i] > ratings[i + 1]:

        candies[i] = max(candies[i], candies[i + 1] + 1)

```
# 计算总和
# 时间复杂度: O(n)
return sum(candies)

# 补充题目 4: LeetCode 406. 根据身高重建队列
# 题目描述: 假设有打乱顺序的一群人站成一个队列, 数组 people 表示队列中一些人的属性 (不一定按顺序)。
# 每个 people[i] = [hi, ki] 表示第 i 个人的身高为 hi , 前面 正好 有 ki 个身高大于或等于 hi 的人。
# 请你重新构造并返回输入数组 people 所表示的队列。
# 返回的队列应该格式化为数组 queue , 其中 queue[j] = [hj, kj] 是队列中第 j 个人的属性 (queue[0] 是排在队列前面的人)。
# 链接: https://leetcode.cn/problems/queue-reconstruction-by-height/
```

```
def reconstruct_queue(people):
    """
    根据身高和前面人的数量重建队列 - 使用贪心算法
    
```

算法思路:

1. 按身高降序, k 升序排序
2. 贪心策略: 按排序后的顺序插入到指定位置
3. 高个子先插入, 不会影响后续插入的矮个子

Args:

people (List[List[int]]): 包含每个人身高和前面人数的列表, 每个元素为[h, k]

Returns:

List[List[int]]: 重建后的队列

时间复杂度:  $O(n^2)$ , 排序  $O(n \log n)$ , 插入操作  $O(n^2)$

空间复杂度:  $O(n)$ , 存储结果数组

工程化考量:

1. 异常处理: 检查输入是否为空
2. 边界条件: 处理空数组、单元素等情况
3. 性能优化: 排序后使用插入策略保证正确性

"""

# 异常处理: 检查输入是否为空

```
if not people:
    return []
```

# 按身高降序, k 升序排序

# 关键点: 身高高的先处理, 不会影响后续插入的矮个子

```
people.sort(key=lambda x: (-x[0], x[1]))
```

```
result = []
# 贪心策略：按排序后的顺序插入到指定位置
for p in people:
    # 在索引 p[1] 处插入当前人
    result.insert(p[1], p)

return result
```

# 补充题目 5: LeetCode 871. 最低加油次数

# 题目描述: 汽车从起点出发驶向目的地, 该目的地位于出发位置东面 target 英里处。  
# 沿途有加油站, 每个 station[i] 代表一个加油站, 位于出发位置东面 station[i][0] 英里处,  
# 并且有 station[i][1] 升汽油。  
# 假设汽车油箱的容量是无限的, 其中最初有 startFuel 升燃料。  
# 它每行驶 1 英里就会用掉 1 升汽油。  
# 当汽车到达加油站时, 它可能停下来加油, 将所有汽油从加油站转移到汽车中。  
# 为了到达目的地, 汽车所必要的最低加油次数是多少? 如果无法到达目的地, 则返回 -1。  
# 链接: <https://leetcode.cn/problems/minimum-number-of-refueling-stops/>

```
import heapq
```

```
def min_refuel_stops(target, start_fuel, stations):
```

```
    """

```

计算到达目的地所需的最少加油次数 - 使用贪心算法+最大堆

算法思路:

1. 使用最大堆存储经过的加油站的汽油量
2. 如果油量不足, 从堆中选择油量最多的加油站加油
3. 处理从最后一个加油站到目的地的情况

Args:

target (int): 目的地距离起点的英里数

start\_fuel (int): 初始燃料量

stations (List[List[int]]): 加油站列表, 每个元素为[位置, 汽油量]

Returns:

int: 所需的最少加油次数, 如果无法到达返回-1

时间复杂度:  $O(n \log n)$ , 其中  $n$  是加油站数量, 堆操作的复杂度

空间复杂度:  $O(n)$ , 堆存储加油站油量

工程化考量：

1. 异常处理：检查输入参数有效性
2. 边界条件：处理无加油站、单加油站等情况
3. 性能优化：使用最大堆快速获取最大油量加油站

"""

```
# 最大堆，存储经过的加油站的汽油量（Python 的 heapq 是最小堆，所以存负数）
```

```
max_heap = []
```

```
fuel = start_fuel # 当前油量
```

```
stops = 0 # 加油次数
```

```
prev = 0 # 上一个位置
```

```
# 处理所有加油站
```

```
# 时间复杂度：O(n)
```

```
for i in range(len(stations)):
```

```
    location, gas = stations[i]
```

```
# 从当前位置到加油站需要的油量
```

```
    fuel -= location - prev
```

```
# 如果油量不足，需要从之前经过的加油站中选择油量最多的加油
```

```
# 时间复杂度：O(log n)（堆操作）
```

```
    while fuel < 0 and max_heap:
```

```
        fuel += -heapq.heappop(max_heap) # 弹出最大的油量并加到当前油量
```

```
        stops += 1
```

```
# 如果无法到达当前加油站，返回-1
```

```
if fuel < 0:
```

```
    return -1
```

```
# 将当前加油站的油量加入堆中（存负数）
```

```
heapq.heappush(max_heap, -gas)
```

```
prev = location
```

```
# 处理从最后一个加油站到目的地
```

```
fuel -= target - prev
```

```
while fuel < 0 and max_heap:
```

```
    fuel += -heapq.heappop(max_heap)
```

```
    stops += 1
```

```
return stops if fuel >= 0 else -1
```

```
# 测试函数
```

```

if __name__ == "__main__":
    # 测试用例 1: 基本情况
    g1 = [1, 2, 3]
    s1 = [1, 1]
    print("测试用例 1 结果:", findContentChildren(g1, s1)) # 期望输出: 1

    # 测试用例 2: 能够满足所有孩子
    g2 = [1, 2]
    s2 = [1, 2, 3]
    print("测试用例 2 结果:", findContentChildren(g2, s2)) # 期望输出: 2

    # 测试用例 3: 边界情况 - 空孩子数组
    g3 = []
    s3 = [1, 2, 3]
    print("测试用例 3 结果:", findContentChildren(g3, s3)) # 期望输出: 0

    # 测试用例 4: 极端情况 - 饼干都不够满足任何孩子
    g4 = [1, 2, 3, 4, 5]
    s4 = [1, 1, 1, 1]
    print("测试用例 4 结果:", findContentChildren(g4, s4)) # 期望输出: 1

```

---

文件: Code07\_BestTimeToBuyAndSellStockII.cpp

---

```

#include <stdio.h>
#include <stdlib.h>
#include <limits.h>

#ifndef NULL
#define NULL 0
#endif

// 替代 fmax 函数
// 算法思路: 比较两个 int 值, 返回较大的一个
// 时间复杂度: O(1)
// 空间复杂度: O(1)
int my_fmax(int a, int b) {
    return (a > b) ? a : b;
}

```

```

// 买卖股票的最佳时机 II
// 给定一个数组, 它的第 i 个元素是一支给定股票第 i 天的价格。

```

```
// 设计一个算法来计算你所能获取的最大利润。  
// 你可以尽可能地完成更多的交易（多次买卖一支股票）。  
// 注意：你不能同时参与多笔交易（你必须在再次购买前出售掉之前的股票）。  
//  
// 算法标签：贪心算法(Greedy Algorithm)、数组遍历(Array Traversal)  
// 时间复杂度：O(n)，其中 n 是价格数组长度  
// 空间复杂度：O(1)，仅使用常数额外空间  
// 测试链接：https://leetcode.cn/problems/best-time-to-buy-and-sell-stock-ii/  
// 相关题目：LeetCode 121. 买卖股票的最佳时机、LeetCode 123. 买卖股票的最佳时机 III  
// 贪心算法专题 - 补充题目收集与详解
```

```
/*  
 * 算法思路详解：  
 * 1. 贪心策略：只要明天价格比今天高，就在今天买入明天卖出  
 *   - 这个策略的核心思想是抓住每一个上涨的机会  
 *   - 将所有正的价格差累加就是最大利润  
 *   - 等价于在所有局部最低点买入，在所有局部最高点卖出  
 *  
 * 2. 遍历价格数组，计算相邻两天的价格差  
 *   - 通过一次遍历完成所有计算  
 *   - 只需要比较相邻两天的价格  
 *  
 * 3. 如果价格差为正，则累加到总利润中  
 *   - 正的价格差表示可以获利  
 *   - 累加所有正的价格差得到最大利润  
 *  
 * 时间复杂度分析：  
 * - 遍历时间复杂度：O(n)，其中 n 是价格数组长度  
 * - 总体时间复杂度：O(n)  
 *  
 * 空间复杂度分析：  
 * - 只使用了常数额外空间存储变量  
 * - 空间复杂度：O(1)  
 *  
 * 是否最优解：  
 * - 是，这是处理此类问题的最优解法  
 * - 贪心策略保证了局部最优解能导致全局最优解  
 *  
 * 工程化最佳实践：  
 * 1. 异常处理：检查输入是否为空或长度不足  
 * 2. 边界条件：处理空数组、单个元素等特殊情况  
 * 3. 性能优化：一次遍历完成计算，避免重复操作  
 * 4. 可读性：清晰的变量命名和详细注释，便于维护
```

\*

\* 极端场景与边界情况处理:

- \* 1. 空输入: prices 为空数组或长度为 0
- \* 2. 极端值: 只有一个价格、所有价格相同
- \* 3. 重复数据: 多个价格相同的情况
- \* 4. 有序/逆序数据: 价格持续上涨或下跌的情况

\*

\* 跨语言实现差异与优化:

- \* 1. Java: 使用增强 for 循环遍历数组, 代码更简洁
- \* 2. C++: 使用传统 for 循环或范围 for 循环, 性能更优
- \* 3. Python: 使用 for 循环或列表推导式, 语法更灵活

\*

\* 调试与测试策略:

- \* 1. 打印中间过程: 在循环中打印每天的价格和利润变化
- \* 2. 用断言验证中间结果: 确保利润不为负
- \* 3. 性能退化排查: 检查是否只遍历了一次数组
- \* 4. 边界测试: 测试空数组、单元素等边界情况

\*

\* 实际应用场景与拓展:

- \* 1. 金融数据分析: 在简单的交易策略中应用贪心算法
- \* 2. 时间序列预测: 作为基线策略进行对比分析
- \* 3. 强化学习: 作为初始策略提供给智能体

\*

\* 算法深入解析:

\* 贪心算法在股票交易问题中的应用体现了其核心思想:

- \* 1. 局部最优选择: 每次选择当前能获得的最大利润
  - \* 2. 无后效性: 当前的选择不会影响之前的状态
  - \* 3. 最优子结构: 问题的最优解包含子问题的最优解
- \* 这个问题的关键洞察是, 多次交易的最大利润等于所有正的价格差之和。

\*/

```
// 买卖股票的最佳时机 II 主函数
```

```
// 算法思路: 贪心算法
```

```
// 1. 遍历价格数组, 计算相邻两天的价格差
```

```
// 2. 如果价格差为正, 则累加到总利润中
```

```
// 时间复杂度: O(n), n 是价格数组长度
```

```
// 空间复杂度: O(1)
```

```
int maxProfit(int prices[], int pricesSize) {
```

```
    // 异常处理: 检查输入是否为空或长度不足
```

```
    if (prices == NULL || pricesSize <= 1) {
```

```
        return 0;
```

```
}
```

```
int maxProfit = 0; // 存储最大利润

// 遍历价格数组，计算相邻两天的价格差
// 时间复杂度: O(n)
for (int i = 1; i < pricesSize; i++) {
    // 如果明天价格比今天高，则累加利润
    // 这体现了贪心策略：抓住每一个上涨的机会
    if (prices[i] > prices[i - 1]) {
        maxProfit += prices[i] - prices[i - 1];
    }
}

return maxProfit; // 返回最大利润
}
```

```
// 补充题目 1: LeetCode 55. 跳跃游戏
// 题目描述：给定一个非负整数数组 nums，你最初位于数组的第一个下标。
// 数组中的每个元素代表你在该位置可以跳跃的最大长度。
// 判断你是否能够到达最后一个下标。
// 链接: https://leetcode.cn/problems/jump-game/
```

```
// 跳跃游戏
// 算法思路：贪心算法
// 1. 维护能到达的最远位置
// 2. 遍历数组，更新最远位置
// 3. 如果当前位置超过了最远位置，无法继续前进
// 时间复杂度: O(n)，n 是数组长度
// 空间复杂度: O(1)
bool canJump(int* nums, int numsSize) {
    // 异常处理：检查输入是否为空
    if (nums == NULL || numsSize == 0) {
        return true; // 空数组视为可以到达
    }
}
```

```
int maxReach = 0; // 当前能到达的最远位置

// 贪心策略：维护能到达的最远位置
// 时间复杂度: O(n)
for (int i = 0; i < numsSize; i++) {
    // 如果当前位置超过了能到达的最远位置，无法继续前进
    if (i > maxReach) {
        return false;
    }
}
```

```

    }

    // 更新能到达的最远位置
    maxReach = my_fmax(maxReach, i + nums[i]);
    // 如果已经能到达或超过最后一个位置，直接返回 true
    if (maxReach >= numsSize - 1) {
        return true;
    }
}

return maxReach >= numsSize - 1;
}

```

```

// 补充题目 2: LeetCode 45. 跳跃游戏 II
// 题目描述: 给定一个非负整数数组，你最初位于数组的第一个位置。
// 数组中的每个元素代表你在该位置可以跳跃的最大长度。
// 你的目标是使用最少的跳跃次数到达数组的最后一个位置。
// 假设你总是可以到达数组的最后一个位置。
// 链接: https://leetcode.cn/problems/jump-game-ii/

```

```

// 跳跃游戏 II
// 算法思路: 贪心算法
// 1. 每次在可到达范围内选择能跳得最远的位置
// 2. 维护当前跳跃能到达的边界和最远位置
// 时间复杂度: O(n)，n 是数组长度
// 空间复杂度: O(1)
int jump(int* nums, int numsSize) {
    // 异常处理: 检查输入是否为空或长度不足
    if (nums == NULL || numsSize <= 1) {
        return 0; // 空数组或只有一个元素不需要跳跃
    }

    int jumps = 0;          // 跳跃次数
    int currentEnd = 0;     // 当前跳跃能到达的边界
    int farthest = 0;       // 在进行下次跳跃前能到达的最远位置

```

```

    // 贪心策略: 每次在可到达范围内选择能跳得最远的位置
    // 时间复杂度: O(n)
    for (int i = 0; i < numsSize - 1; i++) {
        // 更新能到达的最远位置
        farthest = my_fmax(farthest, i + nums[i]);

        // 到达当前跳跃的边界，需要进行一次跳跃
        if (i == currentEnd) {

```

```

        jumps++;
        currentEnd = farthest;

        // 如果已经能到达最后位置，可以提前结束
        if (currentEnd >= numsSize - 1) {
            break;
        }
    }

}

return jumps;
}

```

// 补充题目 3: LeetCode 605. 种花问题  
// 题目描述: 假设有一个很长的花坛, 一部分地块种植了花, 另一部分却没有。  
// 可是, 花不能种植在相邻的地块上, 它们会争夺水源, 两者都会死去。  
// 给你一个整数数组 flowerbed 表示花坛, 由若干 0 和 1 组成, 其中 0 表示没种植花, 1 表示种植了花。  
// 另有一个数 n , 能否在不打破种植规则的情况下种入 n 朵花?  
// 能则返回 true , 不能则返回 false 。  
// 链接: <https://leetcode.cn/problems/can-place-flowers/>

// 种花问题  
// 算法思路: 贪心算法  
// 1. 遍历花坛, 尽可能多地种花  
// 2. 检查当前位置是否可以种花  
// 时间复杂度: O(n) , n 是花坛长度  
// 空间复杂度: O(1)

```

bool canPlaceFlowers(int* flowerbed, int flowerbedSize, int n) {
    // 异常处理: 检查输入是否为空
    if (flowerbed == NULL || flowerbedSize == 0) {
        return n == 0;
    }

    int count = 0; // 可以种的花的数量
    int len = flowerbedSize;

    // 贪心策略: 遍历花坛, 尽可能多地种花
    // 时间复杂度: O(n)
    for (int i = 0; i < len; i++) {
        // 检查当前位置是否可以种花: 当前位置为 0, 且前后都不是 1
        bool canPlant = flowerbed[i] == 0;
        if (i > 0) {
            canPlant = canPlant && (flowerbed[i - 1] == 0);
        }
        if (canPlant) {
            flowerbed[i] = 1;
            count++;
        }
    }
}
```

```

    }

    if (i < len - 1) {
        canPlant = canPlant && (flowerbed[i + 1] == 0);
    }

    if (canPlant) {
        flowerbed[i] = 1; // 在当前位置种花
        count++;
    }

    // 如果已经能满足 n 朵花，提前返回
    if (count >= n) {
        return true;
    }
}

}

return count >= n;
}

```

// 补充题目 4: LeetCode 435. 无重叠区间  
// 题目描述: 给定一个区间的集合 intervals , 其中 intervals[i] = [starti, endi] 。  
// 返回需要移除区间的最小数量, 使剩余区间互不重叠。  
// 链接: <https://leetcode.cn/problems/non-overlapping-intervals/>

// 比较函数, 用于排序二维数组 (按第二个元素升序)  
// 算法思路: 比较两个一维数组的第二个元素  
// 时间复杂度: O(1)  
// 空间复杂度: O(1)

```

bool compareIntervals(const int* a, const int* b) {
    return a[1] < b[1];
}

```

// 替代 sort 函数, 使用冒泡排序对二维数组按第二个元素升序排序  
// 算法思路: 冒泡排序, 比较相邻元素的第二个值, 如果前面的大则交换  
// 时间复杂度: O(n^2), n 为数组长度  
// 空间复杂度: O(1)

```

void my_sort_intervals(int** intervals, int intervalsSize) {
    for (int i = 0; i < intervalsSize - 1; i++) {
        for (int j = 0; j < intervalsSize - i - 1; j++) {
            if (intervals[j][1] > intervals[j + 1][1]) {
                // 交换指针
                int* temp = intervals[j];
                intervals[j] = intervals[j + 1];
                intervals[j + 1] = temp;
            }
        }
    }
}

```

```

        intervals[j + 1] = temp;
    }
}
}

// 无重叠区间
// 算法思路: 贪心算法
// 1. 按区间结束位置排序
// 2. 优先保留结束早的区间
// 时间复杂度: O(n*log(n)), 主要是排序的时间复杂度
// 空间复杂度: O(1)
int eraseOverlapIntervals(int** intervals, int intervalsSize, int* intervalsColSize) {
    // 异常处理: 检查输入是否为空
    if (intervals == NULL || intervalsSize <= 1) {
        return 0;
    }

    // 按区间结束位置排序
    my_sort_intervals(intervals, intervalsSize);

    int count = 0;          // 保留的区间数量
    int end = INT_MIN;     // 上一个保留的区间的结束位置

    // 贪心策略: 优先保留结束早的区间
    // 时间复杂度: O(n)
    for (int i = 0; i < intervalsSize; i++) {
        // 如果当前区间的开始位置大于等于上一个保留区间的结束位置, 则不重叠
        if (intervals[i][0] >= end) {
            count++;
            end = intervals[i][1];
        }
        // 否则, 该区间与上一个保留区间重叠, 需要移除
    }

    // 需要移除的区间数量 = 总区间数量 - 保留的区间数量
    return intervalsSize - count;
}

// 补充题目 5: LeetCode 121. 买卖股票的最佳时机
// 题目描述: 给定一个数组 prices , 它的第 i 个元素 prices[i] 表示一支给定股票第 i 天的价格。
// 你只能选择 某一天 买入这只股票, 并选择在 未来的某一个不同的日子 卖出该股票。
// 设计一个算法来计算你所能获取的最大利润。

```

```

// 返回你可以从这笔交易中获取的最大利润。如果你不能获取任何利润，返回 0。
// 链接: https://leetcode.cn/problems/best-time-to-buy-and-sell-stock/

// 买卖股票的最佳时机
// 算法思路: 贪心算法
// 1. 记录到目前为止的最低价格
// 2. 计算当前价格卖出能获得的最大利润
// 时间复杂度: O(n)，n 是价格数组长度
// 空间复杂度: O(1)

int maxProfitSingle(int* prices, int pricesSize) {
    // 异常处理: 检查输入是否为空或长度不足
    if (prices == NULL || pricesSize <= 1) {
        return 0;
    }

    int minPrice = INT_MAX; // 记录到目前为止的最低价格
    int maxProfit = 0; // 记录最大利润

    // 贪心策略: 每次记录到目前为止的最低价格，计算当前价格卖出能获得的最大利润
    // 时间复杂度: O(n)
    for (int i = 0; i < pricesSize; i++) {
        // 更新最低价格
        if (prices[i] < minPrice) {
            minPrice = prices[i];
        }

        // 计算当前价格卖出能获得的利润，并更新最大利润
        else if (prices[i] - minPrice > maxProfit) {
            maxProfit = prices[i] - minPrice;
        }
    }

    return maxProfit;
}

```

=====

文件: Code07\_BestTimeToBuyAndSellStockII.java

=====

```

package class094;

import java.util.Arrays;

// 买卖股票的最佳时机 II (Best Time to Buy and Sell Stock II)

```

```
// 给定一个数组，它的第 i 个元素是一支给定股票第 i 天的价格。  
// 设计一个算法来计算你所能获取的最大利润。  
// 你可以尽可能地完成更多的交易（多次买卖一支股票）。  
// 注意：你不能同时参与多笔交易（你必须在再次购买前出售掉之前的股票）。  
  
// 算法标签：贪心算法(Greedy Algorithm)、数组遍历(Array Traversal)  
// 时间复杂度：O(n)，其中 n 是价格数组长度  
// 空间复杂度：O(1)，仅使用常数额外空间  
// 测试链接：https://leetcode.cn/problems/best-time-to-buy-and-sell-stock-ii/  
// 相关题目：LeetCode 121. 买卖股票的最佳时机、LeetCode 55. 跳跃游戏  
// 贪心算法专题 - 补充题目收集与详解  
public class Code07_BestTimeToBuyAndSellStockII {
```

```
/*
```

```
* 算法思路详解：
```

- \* 1. 贪心策略核心：只要明天价格比今天高，就在今天买入明天卖出，  
\* 这相当于收集所有正的价格差，实现利润最大化
- \* 2. 遍历优化：通过一次遍历价格数组，计算相邻两天的价格差
- \* 3. 利润累加机制：如果价格差为正（即价格上涨），则将差值累加到总利润中

```
*
```

```
* 时间复杂度分析：
```

- \* - O(n)，其中 n 是价格数组长度，只需要一次遍历即可完成计算

```
* 空间复杂度分析：
```

- \* - O(1)，仅使用了常数级别的额外空间存储 maxProfit 变量

```
* 是否最优解：是，这是处理此类问题的最优解法，无法进一步优化
```

```
*
```

```
* 工程化最佳实践：
```

- \* 1. 输入验证：严格检查输入参数的有效性，防止空指针异常
- \* 2. 边界处理：妥善处理各种边界情况，如空数组、单元素数组等
- \* 3. 性能优化：采用单次遍历策略，避免重复计算
- \* 4. 代码可读性：使用语义明确的变量名和详尽的注释

```
*
```

```
* 极端场景与边界情况处理：
```

- \* 1. 空输入场景：prices 为空数组或 null 时直接返回 0
- \* 2. 极值场景：只有一个价格或所有价格相同（利润为 0）
- \* 3. 重复数据场景：多个连续相同价格的处理
- \* 4. 特殊序列场景：价格持续上涨（最大利润）或持续下跌（利润为 0）
- \* 5. 波动序列场景：价格频繁波动时的利润计算

```
*
```

```
* 跨语言实现差异与优化：
```

- \* 1. Java 实现：使用增强 for 循环或传统 for 循环遍历数组
- \* 2. C++ 实现：使用传统 for 循环或范围 for 循环，注意数组边界
- \* 3. Python 实现：使用 for 循环或 enumerate 函数，利用列表特性

\* 4. 内存管理：不同语言的垃圾回收机制对性能的影响

\*

\* 调试与测试策略：

\* 1. 过程可视化：在关键节点打印每日价格和累计利润

\* 2. 断言验证：在循环体内添加断言确保利润计算正确

\* 3. 性能监控：跟踪遍历过程的实际执行时间

\* 4. 边界测试：设计覆盖所有边界条件的测试用例

\* 5. 压力测试：使用大规模数据验证算法稳定性

\*

\* 实际应用场景与拓展：

\* 1. 金融交易：股票、期货等金融产品的日内交易策略

\* 2. 商品贸易：商品价格波动时的买卖决策

\* 3. 能源管理：电力市场价格波动时的购电策略

\* 4. 库存管理：商品进销差价最大化策略

\* 5. 投资组合：多种资产价格波动时的最优交易时机

\*

\* 算法深入解析：

\* 1. 贪心策略原理：收集所有正收益交易，等价于多次买卖

\* 2. 数学本质：计算所有相邻元素正差值之和

\* 3. 策略等价性：连续上涨时一次交易与多次交易利润相同

\* 4. 风险控制：在实际应用中需考虑交易成本和风险

\*/

```
public static int maxProfit(int[] prices) {
    // 异常处理：检查输入是否为空
    if (prices == null || prices.length <= 1) {
        return 0;
    }

    int maxProfit = 0;

    // 遍历价格数组，计算相邻两天的价格差
    for (int i = 1; i < prices.length; i++) {
        // 如果明天价格比今天高，则累加利润
        if (prices[i] > prices[i - 1]) {
            maxProfit += prices[i] - prices[i - 1];
        }
    }

    return maxProfit;
}
```

// 测试函数

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

```

// 测试用例 1
int[] prices1 = {7, 1, 5, 3, 6, 4};
System.out.println("测试用例 1 结果: " + maxProfit(prices1)); // 期望输出: 7

// 测试用例 2
int[] prices2 = {1, 2, 3, 4, 5};
System.out.println("测试用例 2 结果: " + maxProfit(prices2)); // 期望输出: 4

// 测试用例 3
int[] prices3 = {7, 6, 4, 3, 1};
System.out.println("测试用例 3 结果: " + maxProfit(prices3)); // 期望输出: 0

// 测试用例 4: 边界情况
int[] prices4 = {1};
System.out.println("测试用例 4 结果: " + maxProfit(prices4)); // 期望输出: 0

// 测试用例 5: 极端情况
int[] prices5 = {1, 2, 1, 2, 1, 2};
System.out.println("测试用例 5 结果: " + maxProfit(prices5)); // 期望输出: 3
}

```

// 补充题目 1: LeetCode 55. 跳跃游戏 (Jump Game)

// 题目描述: 给定一个非负整数数组 `nums` , 你最初位于数组的 第一个下标 。

// 数组中的每个元素代表你在该位置可以跳跃的最大长度。

// 判断你是否能够到达最后一个下标。

//

// 算法标签: 贪心算法(Greedy Algorithm)、数组遍历(Array Traversal)

// 时间复杂度:  $O(n)$  , 其中  $n$  是数组长度

// 空间复杂度:  $O(1)$  , 仅使用常数额外空间

// 链接: <https://leetcode.cn/problems/jump-game/>

/\*

\* 算法详解与策略分析:

\* 1. 贪心策略核心: 维护能到达的最远位置, 只要当前位置可达,

\* 就可以更新最远可达位置

\* 2. 可达性判断: 如果当前位置超过了能到达的最远位置, 则无法继续前进

\* 3. 优化终止条件: 一旦能到达最后位置就提前返回

\*

\* 算法步骤详解:

\* 1. 初始化: 设置能到达的最远位置为 0

\* 2. 遍历过程:

\* - 检查当前位置是否可达

\* - 更新能到达的最远位置为  $\max(\text{maxReach}, i + \text{nums}[i])$

\* - 若已能到达最后位置则提前返回 true

```
* 3. 结果判断：遍历结束后检查是否能到达最后位置
*
* 算法正确性证明：
* 贪心选择性质：每次都尽可能扩展可达范围是最优选择
* 最优子结构：在确定当前可达范围后，剩余问题仍具有相同性质
*/
```

```
public static boolean canJump(int[] nums) {
    if (nums == null || nums.length == 0) {
        return true; // 空数组视为可以到达
    }

    int maxReach = 0; // 当前能到达的最远位置

    // 贪心策略：维护能到达的最远位置
    for (int i = 0; i < nums.length; i++) {
        // 如果当前位置超过了能到达的最远位置，无法继续前进
        if (i > maxReach) {
            return false;
        }
        // 更新能到达的最远位置
        maxReach = Math.max(maxReach, i + nums[i]);
        // 如果已经能到达或超过最后一个位置，直接返回 true
        if (maxReach >= nums.length - 1) {
            return true;
        }
    }

    return maxReach >= nums.length - 1;
}

// 补充题目 2: LeetCode 45. 跳跃游戏 II (Jump Game II)
// 题目描述：给定一个非负整数数组，你最初位于数组的第一个位置。
// 数组中的每个元素代表你在该位置可以跳跃的最大长度。
// 你的目标是使用最少的跳跃次数到达数组的最后一个位置。
// 假设你总是可以到达数组的最后一个位置。
//
// 算法标签：贪心算法(Greedy Algorithm)、区间跳跃(Interval Jumping)
// 时间复杂度：O(n)，其中 n 是数组长度
// 空间复杂度：O(1)，仅使用常数额外空间
// 链接：https://leetcode.cn/problems/jump-game-ii/
/*
 * 算法详解与策略分析：
 * 1. 贪心策略核心：在每次跳跃时，选择能跳得最远的位置，
```

- \* 这样可以最小化跳跃次数
- \* 2. 区间管理：维护当前跳跃能到达的边界和下次跳跃前能到达的最远位置
- \* 3. 跳跃决策：当遍历到当前跳跃边界时，必须进行下一次跳跃
- \*
- \* 算法步骤详解：
- \* 1. 初始化：设置跳跃次数、当前边界和最远位置
- \* 2. 遍历过程：
  - 更新能到达的最远位置
  - 当到达当前跳跃边界时增加跳跃次数
  - 更新当前跳跃边界为最远位置
  - 若已能到达最后位置则提前结束
- \* 3. 结果返回：返回累计的跳跃次数
- \*
- \* 算法优化与正确性：
- \* 贪心选择性质：每次在可到达范围内选择跳得最远的位置是最优的
- \* 最优子结构：确定最少跳跃次数后，剩余问题仍保持最优性
- \*/

```

public static int jump(int[] nums) {
    if (nums == null || nums.length <= 1) {
        return 0; // 空数组或只有一个元素不需要跳跃
    }

    int jumps = 0;          // 跳跃次数
    int currentEnd = 0;    // 当前跳跃能到达的边界
    int farthest = 0;       // 在进行下次跳跃前能到达的最远位置

    // 贪心策略：每次在可到达范围内选择能跳得最远的位置
    for (int i = 0; i < nums.length - 1; i++) {
        // 更新能到达的最远位置
        farthest = Math.max(farthest, i + nums[i]);

        // 到达当前跳跃的边界，需要进行一次跳跃
        if (i == currentEnd) {
            jumps++;
            currentEnd = farthest;

            // 如果已经能到达最后位置，可以提前结束
            if (currentEnd >= nums.length - 1) {
                break;
            }
        }
    }
}

```

```

    return jumps;
}

// 补充题目 3: LeetCode 605. 种花问题 (Can Place Flowers)
// 题目描述: 假设有一个很长的花坛, 一部分地块种植了花, 另一部分却没有。
// 可是, 花不能种植在相邻的地块上, 它们会争夺水源, 两者都会死去。
// 给你一个整数数组 flowerbed 表示花坛, 由若干 0 和 1 组成, 其中 0 表示没种植花, 1 表示种植了花。
// 另有一个数 n , 能否在不打破种植规则的情况下种入 n 朵花?
// 能则返回 true , 不能则返回 false .
//

// 算法标签: 贪心算法(Greedy Algorithm)、数组遍历(Array Traversal)、约束满足(Constraint Satisfaction)

// 时间复杂度: O(n), 其中 n 是花坛长度
// 空间复杂度: O(1), 原地修改数组
// 链接: https://leetcode.cn/problems/can-place-flowers/
/*
 * 算法详解与策略分析:
 * 1. 贪心策略核心: 遍历花坛, 尽可能多地种花,
 *   在满足约束条件下优先种花
 * 2. 约束检查: 当前位置可以种花当且仅当:
 *   - 当前位置为 0 (未种花)
 *   - 前一个位置为 0 或不存在
 *   - 后一个位置为 0 或不存在
 * 3. 优化终止: 一旦满足 n 朵花就提前返回
 *
 * 算法步骤详解:
 * 1. 初始化: 设置可种花计数器
 * 2. 遍历过程:
 *   - 检查当前位置是否满足种花条件
 *   - 若满足则在当前位置种花并增加计数器
 *   - 检查是否已满足 n 朵花, 满足则提前返回 true
 * 3. 结果判断: 返回累计种花数是否大于等于 n
 *
 * 算法优化与边界处理:
 * 边界情况: 处理花坛首尾位置的约束检查
 * 空间优化: 可原地修改数组标记已种花位置
 */

public static boolean canPlaceFlowers(int[] flowerbed, int n) {
    if (flowerbed == null || flowerbed.length == 0) {
        return n == 0;
    }

    int jumps = 0;
    for (int i = 0; i < flowerbed.length; i++) {
        if (flowerbed[i] == 0) {
            if (i == 0 || flowerbed[i - 1] == 0) {
                if (i == flowerbed.length - 1 || flowerbed[i + 1] == 0) {
                    jumps++;
                    flowerbed[i] = 1;
                }
            }
        }
    }

    return jumps >= n;
}

```

```

int count = 0; // 可以种的花的数量
int len = flowerbed.length;

// 贪心策略：遍历花坛，尽可能多地种花
for (int i = 0; i < len; i++) {
    // 检查当前位置是否可以种花：当前位置为 0，且前后都不是 1
    boolean canPlant = flowerbed[i] == 0;
    if (i > 0) {
        canPlant = canPlant && (flowerbed[i - 1] == 0);
    }
    if (i < len - 1) {
        canPlant = canPlant && (flowerbed[i + 1] == 0);
    }

    if (canPlant) {
        flowerbed[i] = 1; // 在当前位置种花
        count++;
    }
}

return count >= n;
}

```

```

// 补充题目 4: LeetCode 435. 无重叠区间 (Non-overlapping Intervals)
// 题目描述：给定一个区间的集合 intervals，其中 intervals[i] = [starti, endi] 。
// 返回需要移除区间的最小数量，使剩余区间互不重叠。
//
// 算法标签：贪心算法(Greedy Algorithm)、区间调度(Interval Scheduling)、排序(Sorting)
// 时间复杂度：O(n*log(n))，其中 n 是区间数量
// 空间复杂度：O(1)，仅使用常数额外空间
// 链接：https://leetcode.cn/problems/non-overlapping-intervals/
/*
 * 算法详解与策略分析：
 * 1. 贪心策略核心：按区间结束位置排序，优先保留结束早的区间，
 *     这样可以为后续区间留出更多空间
 * 2. 区间选择：保留不重叠的区间，移除重叠的区间
 * 3. 最优性保证：移除最少区间等价于保留最多区间
 */

```

- \* 算法步骤详解:
- \* 1. 预处理: 按区间结束位置升序排序
- \* 2. 选择过程:
  - 维护上一个保留区间的结束位置
  - 若当前区间与上一个保留区间不重叠则保留
  - 更新上一个保留区间的结束位置
- \* 3. 结果计算: 需移除数量 = 总数 - 保留数量
- \*
- \* 算法正确性证明:
- \* 贪心选择性质: 选择结束位置最早的区间是最优的
- \* 最优子结构: 确定保留区间后, 剩余问题仍保持最优性
- \*/

```

public static int eraseOverlapIntervals(int[][] intervals) {
    if (intervals == null || intervals.length <= 1) {
        return 0;
    }

    // 按区间结束位置排序
    Arrays.sort(intervals, (a, b) -> Integer.compare(a[1], b[1]));

    int count = 0;          // 保留的区间数量
    int end = Integer.MIN_VALUE; // 上一个保留区间的结束位置

    // 贪心策略: 优先保留结束早的区间
    for (int[] interval : intervals) {
        // 如果当前区间的开始位置大于等于上一个保留区间的结束位置, 则不重叠
        if (interval[0] >= end) {
            count++;
            end = interval[1];
        }
        // 否则, 该区间与上一个保留区间重叠, 需要移除
    }

    // 需要移除的区间数量 = 总区间数量 - 保留的区间数量
    return intervals.length - count;
}

// 补充题目 5: LeetCode 121. 买卖股票的最佳时机 (Best Time to Buy and Sell Stock)
// 题目描述: 给定一个数组 prices , 它的第 i 个元素 prices[i] 表示一支给定股票第 i 天的价格。
// 你只能选择 某一天 买入这只股票, 并选择在 未来的某一个不同的日子 卖出该股票。
// 设计一个算法来计算你所能获取的最大利润。
// 返回你可以从这笔交易中获取的最大利润。如果你不能获取任何利润, 返回 0 。
// 
```

```

// 算法标签: 贪心算法(Greedy Algorithm)、单次交易(Single Transaction)、动态规划思想(DP
Thinking)

// 时间复杂度: O(n), 其中 n 是价格数组长度
// 空间复杂度: O(1), 仅使用常数额外空间
// 链接: https://leetcode.cn/problems/best-time-to-buy-and-sell-stock/
/*
 * 算法详解与策略分析:
 * 1. 贪心策略核心: 记录到目前为止的最低价格,
 *    每次计算当前价格卖出能获得的最大利润
 * 2. 状态维护: 同时维护最低价格和最大利润两个状态
 * 3. 优化决策: 仅在找到更低价格或更高利润时更新状态
 *
 * 算法步骤详解:
 * 1. 初始化: 设置最低价格为最大值, 最大利润为 0
 * 2. 遍历过程:
 *    - 若当前价格低于最低价格则更新最低价格
 *    - 否则计算当前价格卖出的利润并更新最大利润
 * 3. 结果返回: 返回记录的最大利润
 *
 * 算法对比与拓展:
 * 与 II 的区别: 只能进行一次交易 vs 可进行多次交易
 * 动态规划解法: 可使用 dp[i][0/1] 表示第 i 天持有/不持有股票的最大利润
*/
public static int maxProfitSingle(int[] prices) {
    if (prices == null || prices.length <= 1) {
        return 0;
    }

    int minPrice = Integer.MAX_VALUE; // 记录到目前为止的最低价格
    int maxProfit = 0; // 记录最大利润

    // 贪心策略: 每次记录到目前为止的最低价格, 计算当前价格卖出能获得的最大利润
    for (int price : prices) {
        // 更新最低价格
        if (price < minPrice) {
            minPrice = price;
        }
        // 计算当前价格卖出能获得的利润, 并更新最大利润
        else if (price - minPrice > maxProfit) {
            maxProfit = price - minPrice;
        }
    }
}

```

```
    return maxProfit;  
}  
}
```

---

文件: Code07\_BestTimeToBuyAndSellStockII.py

---

```
# 买卖股票的最佳时机 II (Best Time to Buy and Sell Stock II)  
# 给定一个数组，它的第 i 个元素是一支给定股票第 i 天的价格。  
# 设计一个算法来计算你所能获取的最大利润。  
# 你可以尽可能地完成更多的交易（多次买卖一支股票）。  
# 注意：你不能同时参与多笔交易（你必须在再次购买前出售掉之前的股票）。  
#  
# 算法标签：贪心算法(Greedy Algorithm)、数组遍历(Array Traversal)  
# 时间复杂度：O(n)，其中 n 是价格数组长度  
# 空间复杂度：O(1)，仅使用常数额外空间  
# 测试链接：https://leetcode.cn/problems/best-time-to-buy-and-sell-stock-ii/  
# 相关题目：LeetCode 121. 买卖股票的最佳时机、LeetCode 123. 买卖股票的最佳时机 III  
# 贪心算法专题 - 补充题目收集与详解
```

"""

算法思路详解：

1. 贪心策略：只要明天价格比今天高，就在今天买入明天卖出
  - 这个策略的核心思想是抓住每一个上涨的机会
  - 将所有正的价格差累加就是最大利润
  - 等价于在所有局部最低点买入，在所有局部最高点卖出
2. 遍历价格数组，计算相邻两天的价格差
  - 通过一次遍历完成所有计算
  - 只需要比较相邻两天的价格
3. 如果价格差为正，则累加到总利润中
  - 正的价格差表示可以获利
  - 累加所有正的价格差得到最大利润

时间复杂度分析：

- 遍历时间复杂度：O(n)，其中 n 是价格数组长度
- 总体时间复杂度：O(n)

空间复杂度分析：

- 只使用了常数额外空间存储变量
- 空间复杂度：O(1)

是否最优解:

- 是, 这是处理此类问题的最优解法
- 贪心策略保证了局部最优解能导致全局最优解

工程化最佳实践:

1. 异常处理: 检查输入是否为空或长度不足
2. 边界条件: 处理空数组、单个元素等特殊情况
3. 性能优化: 一次遍历完成计算, 避免重复操作
4. 可读性: 清晰的变量命名和详细注释, 便于维护

极端场景与边界情况处理:

1. 空输入: prices 为空数组或长度为 0
2. 极端值: 只有一个价格、所有价格相同
3. 重复数据: 多个价格相同的情况
4. 有序/逆序数据: 价格持续上涨或下跌的情况

跨语言实现差异与优化:

1. Java: 使用增强 for 循环遍历数组, 代码更简洁
2. C++: 使用传统 for 循环或范围 for 循环, 性能更优
3. Python: 使用 for 循环或列表推导式, 语法更灵活

调试与测试策略:

1. 打印中间过程: 在循环中打印每天的价格和利润变化
2. 用断言验证中间结果: 确保利润不为负
3. 性能退化排查: 检查是否只遍历了一次数组
4. 边界测试: 测试空数组、单元素等边界情况

实际应用场景与拓展:

1. 金融数据分析: 在简单的交易策略中应用贪心算法
2. 时间序列预测: 作为基线策略进行对比分析
3. 强化学习: 作为初始策略提供给智能体

算法深入解析:

贪心算法在股票交易问题中的应用体现了其核心思想:

1. 局部最优选择: 每次选择当前能获得的最大利润
2. 无后效性: 当前的选择不会影响之前的状态
3. 最优子结构: 问题的最优解包含子问题的最优解

这个问题的关键洞察是, 多次交易的最大利润等于所有正的价格差之和。

"""

```
def maxProfit(prices):
```

"""

## 买卖股票的最佳时机 II 主函数 - 使用贪心算法计算最大利润

算法思路：

1. 遍历价格数组，计算相邻两天的价格差
2. 如果价格差为正，则累加到总利润中

Args:

prices (List[int]): 股票价格列表，prices[i]表示第 i 天的股票价格

Returns:

int: 能够获得的最大利润

时间复杂度:  $O(n)$ ，其中 n 是价格数组长度

空间复杂度:  $O(1)$ ，仅使用常数额外空间

Examples:

```
>>> maxProfit([7, 1, 5, 3, 6, 4])  
7  
>>> maxProfit([1, 2, 3, 4, 5])  
4
```

"""

# 异常处理：检查输入是否为空或长度不足

```
if not prices or len(prices) <= 1:  
    return 0
```

max\_profit = 0 # 存储最大利润

# 遍历价格数组，计算相邻两天的价格差

# 时间复杂度:  $O(n)$

```
for i in range(1, len(prices)):  
    # 如果明天价格比今天高，则累加利润  
    # 这体现了贪心策略：抓住每一个上涨的机会  
    if prices[i] > prices[i - 1]:  
        max_profit += prices[i] - prices[i - 1]
```

return max\_profit # 返回最大利润

# 补充题目 1: LeetCode 55. 跳跃游戏

# 题目描述：给定一个非负整数数组 nums，你最初位于数组的第一个下标。

# 数组中的每个元素代表你在该位置可以跳跃的最大长度。

# 判断你是否能够到达最后一个下标。

```
# 链接: https://leetcode.cn/problems/jump-game/
# 时间复杂度: O(n) - 仅遍历数组一次
# 空间复杂度: O(1) - 只使用常量级额外空间
```

```
def can_jump(nums):
    """
    判断是否能到达数组的最后一个下标 - 使用贪心算法
    
```

算法思路:

1. 维护能到达的最远位置
2. 遍历数组，更新最远位置
3. 如果当前位置超过了最远位置，无法继续前进

Args:

nums (List[int]): 整数列表，其中每个元素表示在该位置可以跳跃的最大长度

Returns:

bool: 如果能到达最后一个下标返回 True，否则返回 False

时间复杂度: O(n)，其中 n 是数组长度

空间复杂度: O(1)，只使用常量级额外空间

贪心策略：维护能到达的最远位置，如果最远位置超过或等于数组最后一个位置，则可以到达

工程化考量:

1. 异常处理：检查输入是否为空
2. 边界条件：处理空数组等特殊情况
3. 性能优化：提前终止条件避免不必要的计算

"""

```
# 异常处理: 检查输入是否为空
if not nums:
    return True # 空数组视为可以到达
```

```
max_reach = 0 # 当前能到达的最远位置
```

```
# 贪心策略: 维护能到达的最远位置
# 时间复杂度: O(n)
for i in range(len(nums)):
    # 如果当前位置超过了能到达的最远位置，无法继续前进
    if i > max_reach:
        return False
    # 更新能到达的最远位置
    max_reach = max(max_reach, i + nums[i])
```

```
# 如果已经能到达或超过最后一个位置，直接返回 True
if max_reach >= len(nums) - 1:
    return True

return max_reach >= len(nums) - 1
```

```
# 补充题目 2: LeetCode 45. 跳跃游戏 II
# 题目描述：给定一个非负整数数组，你最初位于数组的第一个位置。
# 数组中的每个元素代表你在该位置可以跳跃的最大长度。
# 你的目标是使用最少的跳跃次数到达数组的最后一个位置。
# 假设你总是可以到达数组的最后一个位置。
# 链接: https://leetcode.cn/problems/jump-game-ii/
# 时间复杂度: O(n) - 仅遍历数组一次
# 空间复杂度: O(1) - 只使用常量级额外空间
```

```
def jump(nums):
    """
```

使用最少的跳跃次数到达数组的最后一个位置 - 使用贪心算法

算法思路:

1. 每次在可到达范围内选择能跳得最远的位置
2. 维护当前跳跃能到达的边界和最远位置

Args:

nums (List[int]): 整数列表，其中每个元素表示在该位置可以跳跃的最大长度

Returns:

int: 到达最后一个位置所需的最少跳跃次数

时间复杂度: O(n)，其中 n 是数组长度

空间复杂度: O(1)，只使用常量级额外空间

贪心策略: 每次在可到达范围内选择能跳得最远的位置作为下一个跳跃点

工程化考量:

1. 异常处理: 检查输入是否为空或长度不足
2. 边界条件: 处理空数组、单元素等情况
3. 性能优化: 提前终止条件避免不必要的计算

"""

```
# 异常处理: 检查输入是否为空或长度不足
if not nums or len(nums) <= 1:
    return 0 # 空数组或只有一个元素不需要跳跃
```

```

jumps = 0          # 跳跃次数
current_end = 0   # 当前跳跃能到达的边界
farthest = 0       # 在进行下次跳跃前能到达的最远位置

# 贪心策略：每次在可到达范围内选择能跳得最远的位置
# 时间复杂度：O(n)
for i in range(len(nums) - 1):
    # 更新能到达的最远位置
    farthest = max(farthest, i + nums[i])

    # 到达当前跳跃的边界，需要进行一次跳跃
    if i == current_end:
        jumps += 1
        current_end = farthest

    # 如果已经能到达最后位置，可以提前结束
    if current_end >= len(nums) - 1:
        break

return jumps

```

```

# 补充题目 3: LeetCode 605. 种花问题
# 题目描述：假设有一个很长的花坛，一部分地块种植了花，另一部分却没有。
# 可是，花不能种植在相邻的地块上，它们会争夺水源，两者都会死去。
# 给你一个整数数组 flowerbed 表示花坛，由若干 0 和 1 组成，其中 0 表示没种植花，1 表示种植了花。
# 另有一个数 n，能否在不打破种植规则的情况下种入 n 朵花？
# 能则返回 True，不能则返回 False。
# 链接: https://leetcode.cn/problems/can-place-flowers/
# 时间复杂度：O(n) - 仅遍历数组一次
# 空间复杂度：O(1) - 只使用常量级额外空间

```

```

def can_place_flowers(flowerbed, n):
    """
    判断能否在不打破种植规则的情况下种入 n 朵花 - 使用贪心算法
    """

    判断能否在不打破种植规则的情况下种入 n 朵花 - 使用贪心算法

```

算法思路：

1. 遍历花坛，尽可能多地种花
2. 检查当前位置是否可以种花

Args:

flowerbed (List[int]): 整数列表，表示花坛，0 表示没种花，1 表示种了花

n (int): 整数，表示要种的花的数量

Returns:

bool: 如果能种入 n 朵花返回 True, 否则返回 False

时间复杂度:  $O(n)$ , 其中 n 是花坛长度

空间复杂度:  $O(1)$ , 只使用常量级额外空间

贪心策略: 遍历花坛, 尽可能多地种花

工程化考量:

1. 异常处理: 检查输入是否为空
2. 边界条件: 处理空数组等特殊情况
3. 性能优化: 提前终止条件避免不必要的计算

"""

```
# 异常处理: 检查输入是否为空
if not flowerbed:
    return n == 0

count = 0 # 可以种的花的数量
length = len(flowerbed)

# 贪心策略: 遍历花坛, 尽可能多地种花
# 时间复杂度:  $O(n)$ 
for i in range(length):
    # 检查当前位置是否可以种花: 当前位置为 0, 且前后都不是 1
    can_plant = flowerbed[i] == 0
    if i > 0:
        can_plant = can_plant and (flowerbed[i - 1] == 0)
    if i < length - 1:
        can_plant = can_plant and (flowerbed[i + 1] == 0)

    if can_plant:
        flowerbed[i] = 1 # 在当前位置种花
        count += 1

    # 如果已经能满足 n 朵花, 提前返回
    if count >= n:
        return True

return count >= n
```

```
# 补充题目 4: LeetCode 435. 无重叠区间
# 题目描述: 给定一个区间的集合 intervals , 其中 intervals[i] = [starti, endi] 。
# 返回需要移除区间的最小数量, 使剩余区间互不重叠。
# 链接: https://leetcode.cn/problems/non-overlapping-intervals/
# 时间复杂度: O(n log n) - 排序需要 O(n log n), 遍历需要 O(n)
# 空间复杂度: O(1) - 只使用常量级额外空间
```

```
def erase_overlap_intervals(intervals):
    """
    计算需要移除的区间的最小数量, 使剩余区间互不重叠 - 使用贪心算法
    
```

算法思路:

1. 按区间结束位置排序
2. 优先保留结束早的区间

Args:

intervals (List[List[int]]): 区间列表, 每个区间是包含两个整数的列表 [start, end]

Returns:

int: 需要移除的区间的最小数量

时间复杂度: O(n log n), 排序需要 O(n log n), 遍历需要 O(n)

空间复杂度: O(1), 只使用常量级额外空间

贪心策略: 优先保留结束早的区间, 这样可以保留更多的区间

工程化考量:

1. 异常处理: 检查输入是否为空或长度不足
2. 边界条件: 处理空数组、单元素等情况
3. 性能优化: 排序后使用贪心策略避免重复计算

"""

# 异常处理: 检查输入是否为空或长度不足

```
if not intervals or len(intervals) <= 1:
    return 0
```

# 按区间结束位置排序

# 关键点: 按结束位置排序保证了贪心策略的正确性

```
intervals.sort(key=lambda x: x[1])
```

count = 0 # 保留的区间数量

```
end = float('-inf') # 上一个保留的区间的结束位置
```

# 贪心策略: 优先保留结束早的区间

```

# 时间复杂度: O(n)
for interval in intervals:
    # 如果当前区间的开始位置大于等于上一个保留区间的结束位置，则不重叠
    if interval[0] >= end:
        count += 1
        end = interval[1]
    # 否则，该区间与上一个保留区间重叠，需要移除

# 需要移除的区间数量 = 总区间数量 - 保留的区间数量
return len(intervals) - count

```

# 补充题目 5: LeetCode 121. 买卖股票的最佳时机

# 题目描述: 给定一个数组 prices , 它的第 i 个元素 prices[i] 表示一支给定股票第 i 天的价格。

# 你只能选择 某一天 买入这只股票，并选择在 未来的某一个不同的日子 卖出该股票。

# 设计一个算法来计算你所能获取的最大利润。

# 返回你可以从这笔交易中获取的最大利润。如果你不能获取任何利润，返回 0 。

# 链接: <https://leetcode.cn/problems/best-time-to-buy-and-sell-stock/>

# 时间复杂度: O(n) - 仅遍历数组一次

# 空间复杂度: O(1) - 只使用常量级额外空间

```

def max_profit_single(prices):
    """
    计算单次买卖股票能获得的最大利润 - 使用贪心算法

```

算法思路:

1. 记录到目前为止的最低价格
2. 计算当前价格卖出能获得的最大利润

Args:

prices (List[int]): 整数列表，表示股票每天的价格

Returns:

int: 最大利润，如果不能获利返回 0

时间复杂度: O(n)，其中 n 是价格数组长度

空间复杂度: O(1)，只使用常量级额外空间

贪婪策略：每次记录到目前为止的最低价格，计算当前价格卖出能获得的最大利润

工程化考量:

1. 异常处理: 检查输入是否为空或长度不足
2. 边界条件: 处理空数组、单元素等情况

### 3. 性能优化：一次遍历完成计算

```
"""
```

```
# 异常处理：检查输入是否为空或长度不足
```

```
if not prices or len(prices) <= 1:
```

```
    return 0
```

```
min_price = float('inf') # 记录到目前为止的最低价格
```

```
max_profit = 0 # 记录最大利润
```

```
# 贪心策略：每次记录到目前为止的最低价格，计算当前价格卖出能获得的最大利润
```

```
# 时间复杂度：O(n)
```

```
for price in prices:
```

```
    # 更新最低价格
```

```
    if price < min_price:
```

```
        min_price = price
```

```
    # 计算当前价格卖出能获得的利润，并更新最大利润
```

```
    elif price - min_price > max_profit:
```

```
        max_profit = price - min_price
```

```
return max_profit
```

```
# 测试函数
```

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

```
# 测试用例 1：一般情况
```

```
prices1 = [7, 1, 5, 3, 6, 4]
```

```
print("测试用例 1 结果:", maxProfit(prices1)) # 期望输出: 7
```

```
# 测试用例 2：持续上涨
```

```
prices2 = [1, 2, 3, 4, 5]
```

```
print("测试用例 2 结果:", maxProfit(prices2)) # 期望输出: 4
```

```
# 测试用例 3：持续下跌
```

```
prices3 = [7, 6, 4, 3, 1]
```

```
print("测试用例 3 结果:", maxProfit(prices3)) # 期望输出: 0
```

```
# 测试用例 4：边界情况 - 单个价格
```

```
prices4 = [1]
```

```
print("测试用例 4 结果:", maxProfit(prices4)) # 期望输出: 0
```

```
# 测试用例 5：波动价格
```

```
prices5 = [1, 2, 1, 2, 1, 2]
```

```
print("测试用例 5 结果:", maxProfit(prices5)) # 期望输出: 3
```

文件: Code08\_JumpGame.cpp

```
// 跳跃游戏
// 给定一个非负整数数组 nums，你最初位于数组的第一个下标。
// 数组中的每个元素代表你在该位置可以跳跃的最大长度。
// 判断你是否能够到达最后一个下标。
// 测试链接 : https://leetcode.cn/problems/jump-game/
```

```
/*
 * 算法思路:
 * 1. 贪心策略: 维护能到达的最远位置
 * 2. 遍历数组, 更新能到达的最远位置
 * 3. 如果当前位置超过了能到达的最远位置, 则无法到达终点
 * 4. 如果能到达的最远位置大于等于最后一个下标, 则能到达终点
 *
 * 时间复杂度: O(n) - n 是数组长度
 * 空间复杂度: O(1) - 只使用了常数额外空间
 * 是否最优解: 是, 这是处理此类问题的最优解法
 *
 * 工程化考量:
 * 1. 异常处理: 检查输入是否为空
 * 2. 边界条件: 处理空数组、单个元素等特殊情况
 * 3. 性能优化: 一次遍历完成计算
 * 4. 可读性: 清晰的变量命名和注释
 *
 * 极端场景与边界场景:
 * 1. 空输入: nums 为空数组
 * 2. 极端值: 只有一个元素、所有元素都是 0
 * 3. 重复数据: 多个元素相同
 * 4. 有序/逆序数据: 元素值递增或递减
 *
 * 跨语言场景与语言特性差异:
 * 1. Java: 使用增强 for 循环遍历数组
 * 2. C++: 使用传统 for 循环
 * 3. Python: 使用 for 循环或 enumerate
 *
 * 调试能力构建:
 * 1. 打印中间过程: 在循环中打印当前位置和最远可达位置
 * 2. 用断言验证中间结果: 确保最远位置不减小
 * 3. 性能退化排查: 检查是否只遍历了一次数组
```

```
*  
* 与机器学习、图像处理、自然语言处理的联系与应用:  
* 1. 在路径规划问题中，贪心算法可用于快速判断可达性  
* 2. 在图论算法中，可以作为初始解提供给更复杂的算法  
* 3. 在网络路由中，可以用于快速判断连通性  
*/
```

```
// 跳跃游戏主函数  
int canJump(int nums[], int numsSize) {  
    // 异常处理：检查输入是否为空  
    if (nums == 0 || numsSize == 0) {  
        return 0; // 0 表示 false  
    }  
  
    // 边界条件：只有一个元素，肯定能到达  
    if (numsSize == 1) {  
        return 1; // 1 表示 true  
    }  
  
    int maxReach = 0; // 能到达的最远位置  
  
    // 遍历数组  
    for (int i = 0; i < numsSize; i++) {  
        // 如果当前位置超过了能到达的最远位置，则无法到达终点  
        if (i > maxReach) {  
            return 0; // 0 表示 false  
        }  
  
        // 更新能到达的最远位置  
        int currentReach = i + nums[i];  
        if (currentReach > maxReach) {  
            maxReach = currentReach;  
        }  
  
        // 如果能到达的最远位置大于等于最后一个下标，则能到达终点  
        if (maxReach >= numsSize - 1) {  
            return 1; // 1 表示 true  
        }  
    }  
  
    return 0; // 0 表示 false  
}
```

```
// 补充题目 1: LeetCode 134. 加油站
// 题目描述: 在一条环路上有 n 个加油站, 其中第 i 个加油站有汽油 gas[i] 升。
// 你有一辆油箱容量无限的的汽车, 从第 i 个加油站开往第 i+1 个加油站需要消耗汽油 cost[i] 升。
// 你从其中的一个加油站出发, 开始时油箱为空。
// 如果你可以绕环路行驶一周, 则返回出发时加油站的编号, 否则返回 -1。
// 注意: 如果题目有解, 该答案即为唯一答案。
// 链接: https://leetcode.cn/problems/gas-station/
```

```
int canCompleteCircuit(int* gas, int gasSize, int* cost, int costSize) {
    if (gas == NULL || cost == NULL || gasSize != costSize) {
        return -1; // 输入不合法
    }

    int n = gasSize;
    int totalGas = 0; // 总油量
    int totalCost = 0; // 总消耗
    int currentGas = 0; // 当前剩余油量
    int startStation = 0; // 起始加油站

    // 贪心策略: 如果从 A 到 B 的路上没油了, 那么 A 到 B 之间的任何一个站点都不能作为起点
    for (int i = 0; i < n; i++) {
        totalGas += gas[i];
        totalCost += cost[i];
        currentGas += gas[i] - cost[i];

        // 如果当前剩余油量为负, 说明从 startStation 到 i 的路径不可行
        if (currentGas < 0) {
            startStation = i + 1; // 从下一个站点重新开始计算
            currentGas = 0; // 重置当前剩余油量
        }
    }

    // 如果总油量小于总消耗, 那么无论如何都不可能绕行一周
    if (totalGas < totalCost) {
        return -1;
    }

    // 否则, startStation 就是答案
    return startStation;
}
```

```
// 补充题目 2: LeetCode 561. 数组拆分 I
// 题目描述: 给定长度为 2n 的整数数组 nums , 你的任务是将这些数分成 n 对,
```

```

// 例如 (a1, b1), (a2, b2), ..., (an, bn) , 使得从 1 到 n 的 min(ai, bi) 总和最大。
// 返回该 最大总和 。
// 链接: https://leetcode.cn/problems/array-partition-i/

int arrayPairSum(int* nums, int numsSize) {
    if (nums == NULL || numsSize % 2 != 0) {
        return 0; // 输入不合法
    }

    // 贪心策略: 将数组排序后, 每两个相邻的数分为一组, 取较小的那个 (即每对中的第一个数)
    sort(nums, nums + numsSize);
    int maxSum = 0;

    // 每隔一个元素取一个 (即每对中的第一个元素)
    for (int i = 0; i < numsSize; i += 2) {
        maxSum += nums[i];
    }

    return maxSum;
}

// 补充题目 3: LeetCode 402. 移掉 K 位数字
// 题目描述: 给你一个以字符串表示的非负整数 num 和一个整数 k ,
// 移除这个数中的 k 位数字, 使得剩下的数字最小。
// 请你以字符串形式返回这个最小的数字。
// 链接: https://leetcode.cn/problems/remove-k-digits/

char* removeKdigits(char* num, int k) {
    if (num == NULL || strlen(num) == 0 || k >= strlen(num)) {
        char* result = (char*)malloc(2 * sizeof(char));
        result[0] = '0';
        result[1] = '\0';
        return result;
    }

    int n = strlen(num);
    char* stack = (char*)malloc((n + 1) * sizeof(char)); // 使用栈来存储需要保留的数字
    int top = 0; // 栈顶指针

    // 贪心策略: 从左到右遍历, 如果当前数字小于栈顶数字, 且还有删除次数, 则弹出栈顶数字
    for (int i = 0; i < n; i++) {
        char digit = num[i];
        // 当栈不为空, 当前数字小于栈顶数字, 且还有删除次数时, 弹出栈顶数字

```

```

while (top > 0 && digit < stack[top - 1] && k > 0) {
    top--;
    k--;
}
// 将当前数字入栈
stack[top++] = digit;
}

// 如果还有删除次数，从栈顶删除
while (k > 0 && top > 0) {
    top--;
    k--;
}

// 去除前导零
int start = 0;
while (start < top && stack[start] == '0') {
    start++;
}

// 构建结果字符串
char* result;
if (start == top) {
    // 如果全是零，返回"0"
    result = (char*)malloc(2 * sizeof(char));
    result[0] = '0';
    result[1] = '\0';
} else {
    result = (char*)malloc((top - start + 1) * sizeof(char));
    strncpy(result, stack + start, top - start);
    result[top - start] = '\0';
}

free(stack);
return result;
}

// 补充题目 4: LeetCode 122. 买卖股票的最佳时机 II (另一种贪心实现)
// 题目描述: 给定一个数组 prices，其中 prices[i] 表示股票第 i 天的价格。
// 在每一天，你可能会决定购买和/或出售股票。你在任何时候 最多 只能持有 一股 股票。
// 你也可以购买它，然后在 同一天 出售。
// 返回 你能获得的 最大 利润。
// 链接: https://leetcode.cn/problems/best-time-to-buy-and-sell-stock-ii/

```

```

int maxProfit(int* prices, int pricesSize) {
    if (prices == NULL || pricesSize <= 1) {
        return 0; // 无法交易
    }

    int maxProfit = 0;
    // 贪心策略：只要后一天的价格比前一天高，就进行一次买卖
    for (int i = 1; i < pricesSize; i++) {
        // 如果当天价格比前一天高，就进行交易
        if (prices[i] > prices[i - 1]) {
            maxProfit += prices[i] - prices[i - 1];
        }
    }

    return maxProfit;
}

// 补充题目 5: LeetCode 665. 非递减数列
// 题目描述：给你一个长度为 n 的整数数组 nums，请你判断在 最多 改变 1 个元素的情况下，该数组能否变成一个非递减数列。
// 非递减数列的定义是：对于数组中任意的 i (0 <= i <= n-2)，总满足 nums[i] <= nums[i+1]。
// 链接：https://leetcode.cn/problems/non-decreasing-array/

bool checkPossibility(int* nums, int numsSize) {
    if (nums == NULL || numsSize <= 1) {
        return true; // 空数组或只有一个元素是非递减的
    }

    int count = 0; // 记录需要修改的次数

    for (int i = 0; i < numsSize - 1; i++) {
        if (nums[i] > nums[i + 1]) {
            count++;
            if (count > 1) {
                return false; // 需要修改超过 1 次
            }
        }
    }

    // 贪心策略：尽可能修改 nums[i] 而不是 nums[i+1]，这样对后续影响更小
    // 但是如果 nums[i-1] > nums[i+1]，则必须修改 nums[i+1]
    if (i > 0 && nums[i - 1] > nums[i + 1]) {
        nums[i + 1] = nums[i]; // 修改 nums[i+1]
    } else {

```

```
        nums[i] = nums[i + 1]; // 修改 nums[i]
    }
}

}

return true;
}
```

---

文件: Code08\_JumpGame.java

---

```
package class094;

import java.util.Arrays;
import java.util.Stack;

// 跳跃游戏 (Jump Game)
// 给定一个非负整数数组 nums , 你最初位于数组的第一个下标。
// 数组中的每个元素代表你在该位置可以跳跃的最大长度。
// 判断你是否能够到达最后一个下标。
//
// 算法标签: 贪心算法(Greedy Algorithm)、数组遍历(Array Traversal)、可达性分析(Reachability Analysis)
// 时间复杂度: O(n) , 其中 n 是数组长度
// 空间复杂度: O(1) , 仅使用常数额外空间
// 测试链接 : https://leetcode.cn/problems/jump-game/
// 相关题目: LeetCode 45. 跳跃游戏 II、LeetCode 134. 加油站
// 贪心算法专题 - 补充题目收集与详解
public class Code08_JumpGame {

/*
 * 算法思路详解:
 * 1. 贪心策略核心: 维护能到达的最远位置 maxReach,
 *     这样可以确保在遍历过程中始终知道能够到达的边界
 * 2. 遍历优化: 通过一次遍历数组, 动态更新最远可达位置
 * 3. 不可达判断: 如果当前位置 i 超过了 maxReach, 说明无法到达位置 i
 * 4. 可达性确认: 当 maxReach >= nums.length - 1 时, 说明可以到达终点
 *
 * 时间复杂度分析:
 * - O(n) , 其中 n 是数组长度, 只需要一次遍历即可完成判断
 * 空间复杂度分析:
 * - O(1), 仅使用了常数级别的额外空间存储 maxReach 等变量
}
```

- \* 是否最优解：是，这是处理此类可达性问题的最优解法
- \*
- \* 工程化最佳实践：
  - \* 1. 输入验证：严格检查输入参数的有效性，防止空指针异常
  - \* 2. 边界处理：妥善处理各种边界情况，如空数组、单元素数组等
  - \* 3. 性能优化：采用单次遍历策略，避免重复计算
  - \* 4. 代码可读性：使用语义明确的变量名和详尽的注释
  - \* 5. 提前终止：一旦确认可达就提前返回，避免不必要的计算
- \*
- \* 极端场景与边界情况处理：
  - \* 1. 空输入场景：nums 为空数组或 null 时直接返回 false
  - \* 2. 单元素场景：只有一个元素时肯定可达
  - \* 3. 全零场景：所有元素都是 0 时只有位置 0 可达
  - \* 4. 递增序列：元素值递增时可达性分析
  - \* 5. 递减序列：元素值递减时的可达性判断
  - \* 6. 特殊模式：如 [1, 0, 1, 0] 等交替模式的处理
- \*
- \* 跨语言实现差异与优化：
  - \* 1. Java 实现：使用传统 for 循环，注意数组边界检查
  - \* 2. C++ 实现：使用传统 for 循环，注意指针和数组访问
  - \* 3. Python 实现：使用 for 循环或 enumerate 函数，利用列表特性
  - \* 4. 内存管理：不同语言的垃圾回收机制对性能的影响
- \*
- \* 调试与测试策略：
  - \* 1. 过程可视化：在关键节点打印当前位置和最远可达位置
  - \* 2. 断言验证：在循环体内添加断言确保 maxReach 单调不减
  - \* 3. 性能监控：跟踪遍历过程的实际执行时间
  - \* 4. 边界测试：设计覆盖所有边界条件的测试用例
  - \* 5. 压力测试：使用大规模数据验证算法稳定性
- \*
- \* 实际应用场景与拓展：
  - \* 1. 游戏开发：角色移动范围判断、关卡可达性分析
  - \* 2. 网络路由：网络连通性快速判断、最短路径预判
  - \* 3. 机器人路径规划：移动机器人可达区域分析
  - \* 4. 任务调度：任务依赖关系可达性分析
  - \* 5. 社交网络：人际关系传播范围分析
- \*
- \* 算法深入解析：
  - \* 1. 贪心策略原理：维护最远可达位置确保全局最优
  - \* 2. 算法不变式：遍历过程中 maxReach 始终表示当前能到达的最远位置
  - \* 3. 正确性证明：如果位置 i 可达，则 0 到 i-1 所有位置都可达
  - \* 4. 优化扩展：可记录具体路径或跳跃次数
- \*/

```
public static boolean canJump(int[] nums) {  
    // 异常处理：检查输入是否为空  
    if (nums == null || nums.length == 0) {  
        return false;  
    }  
  
    // 边界条件：只有一个元素，肯定能到达  
    if (nums.length == 1) {  
        return true;  
    }  
  
    int maxReach = 0; // 能到达的最远位置  
  
    // 遍历数组  
    for (int i = 0; i < nums.length; i++) {  
        // 如果当前位置超过了能到达的最远位置，则无法到达终点  
        if (i > maxReach) {  
            return false;  
        }  
  
        // 更新能到达的最远位置  
        maxReach = Math.max(maxReach, i + nums[i]);  
  
        // 如果能到达的最远位置大于等于最后一个下标，则能到达终点  
        if (maxReach >= nums.length - 1) {  
            return true;  
        }  
    }  
  
    return false;  
}  
  
// 测试函数  
public static void main(String[] args) {  
    // 测试用例 1  
    int[] nums1 = {2, 3, 1, 1, 4};  
    System.out.println("测试用例 1 结果: " + canJump(nums1)); // 期望输出: true  
  
    // 测试用例 2  
    int[] nums2 = {3, 2, 1, 0, 4};  
    System.out.println("测试用例 2 结果: " + canJump(nums2)); // 期望输出: false  
  
    // 测试用例 3: 边界情况
```

```

int[] nums3 = {0};
System.out.println("测试用例 3 结果: " + canJump(nums3)); // 期望输出: true

// 测试用例 4: 极端情况
int[] nums4 = {1, 0, 1, 0};
System.out.println("测试用例 4 结果: " + canJump(nums4)); // 期望输出: false

// 测试用例 5: 全为 1
int[] nums5 = {1, 1, 1, 1, 1};
System.out.println("测试用例 5 结果: " + canJump(nums5)); // 期望输出: true
}

// 补充题目 1: LeetCode 134. 加油站
// 题目描述: 在一条环路上有 n 个加油站, 其中第 i 个加油站有汽油 gas[i] 升。
// 你有一辆油箱容量无限的汽车, 从第 i 个加油站开往第 i+1 个加油站需要消耗汽油 cost[i] 升。
// 你从其中的一个加油站出发, 开始时油箱为空。
// 如果你可以绕环路行驶一周, 则返回出发时加油站的编号, 否则返回 -1。
// 注意: 如果题目有解, 该答案即为唯一答案。
// 链接: https://leetcode.cn/problems/gas-station/
public static int canCompleteCircuit(int[] gas, int[] cost) {
    if (gas == null || cost == null || gas.length != cost.length) {
        return -1; // 输入不合法
    }

    int n = gas.length;
    int totalGas = 0; // 总油量
    int totalCost = 0; // 总消耗
    int currentGas = 0; // 当前剩余油量
    int startStation = 0; // 起始加油站

    // 贪心策略: 如果从 A 到 B 的路上没油了, 那么 A 到 B 之间的任何一个站点都不能作为起点
    for (int i = 0; i < n; i++) {
        totalGas += gas[i];
        totalCost += cost[i];
        currentGas += gas[i] - cost[i];

        // 如果当前剩余油量为负, 说明从 startStation 到 i 的路径不可行
        if (currentGas < 0) {
            startStation = i + 1; // 从下一个站点重新开始计算
            currentGas = 0; // 重置当前剩余油量
        }
    }
}

```

```

// 如果总油量小于总消耗，那么无论如何都不可能绕行一周
if (totalGas < totalCost) {
    return -1;
}

// 否则，startStation 就是答案
return startStation;
}

// 补充题目 2: LeetCode 561. 数组拆分 I
// 题目描述: 给定长度为 2n 的整数数组 nums，你的任务是将这些数分成 n 对，
// 例如 (a1, b1), (a2, b2), ..., (an, bn)，使得从 1 到 n 的 min(ai, bi) 总和最大。
// 返回该 最大总和。
// 链接: https://leetcode.cn/problems/array-partition-i/
public static int arrayPairSum(int[] nums) {
    if (nums == null || nums.length % 2 != 0) {
        return 0; // 输入不合法
    }

    // 贪心策略: 将数组排序后，每两个相邻的数分为一组，取较小的那个（即每对中的第一个数）
    Arrays.sort(nums);
    int maxSum = 0;

    // 每隔一个元素取一个（即每对中的第一个元素）
    for (int i = 0; i < nums.length; i += 2) {
        maxSum += nums[i];
    }

    return maxSum;
}

// 补充题目 3: LeetCode 402. 移掉 K 位数字
// 题目描述: 给你一个以字符串表示的非负整数 num 和一个整数 k，
// 移除这个数中的 k 位数字，使得剩下的数字最小。
// 请你以字符串形式返回这个最小的数字。
// 链接: https://leetcode.cn/problems/remove-k-digits/
public static String removeKdigits(String num, int k) {
    if (num == null || num.isEmpty() || k >= num.length()) {
        return "0"; // 移除所有数字，返回 0
    }

    // 使用栈来存储需要保留的数字
    Stack<Character> stack = new Stack<>();

```

```

// 贪心策略：从左到右遍历，如果当前数字小于栈顶数字，且还有删除次数，则弹出栈顶数字
for (int i = 0; i < num.length(); i++) {
    char digit = num.charAt(i);
    // 当栈不为空，当前数字小于栈顶数字，且还有删除次数时，弹出栈顶数字
    while (!stack.isEmpty() && digit < stack.peek() && k > 0) {
        stack.pop();
        k--;
    }
    // 将当前数字入栈
    stack.push(digit);
}

// 如果还有删除次数，从栈顶删除
while (k > 0) {
    stack.pop();
    k--;
}

// 构建结果字符串
StringBuilder result = new StringBuilder();
while (!stack.isEmpty()) {
    result.append(stack.pop());
}
result.reverse(); // 反转字符串，因为栈是后进先出的

// 去除前导零
int start = 0;
while (start < result.length() && result.charAt(start) == '0') {
    start++;
}

// 如果全是零，返回"0"
return start == result.length() ? "0" : result.substring(start);
}

// 补充题目 4: LeetCode 122. 买卖股票的最佳时机 II (另一种贪心实现)
// 题目描述：给定一个数组 prices，其中 prices[i] 表示股票第 i 天的价格。
// 在每一天，你可能会决定购买和/或出售股票。你在任何时候 最多 只能持有 一股 股票。
// 你也可以购买它，然后在 同一天 出售。
// 返回 你能获得的 最大 利润。
// 链接: https://leetcode.cn/problems/best-time-to-buy-and-sell-stock-ii/
public static int maxProfit(int[] prices) {

```

```

if (prices == null || prices.length <= 1) {
    return 0; // 无法交易
}

int maxProfit = 0;
// 贪心策略：只要后一天的价格比前一天高，就进行一次买卖
for (int i = 1; i < prices.length; i++) {
    // 如果当天价格比前一天高，就进行交易
    if (prices[i] > prices[i - 1]) {
        maxProfit += prices[i] - prices[i - 1];
    }
}

return maxProfit;
}

// 补充题目 5: LeetCode 665. 非递减数列
// 题目描述：给你一个长度为 n 的整数数组 nums ，请你判断在 最多 改变 1 个元素的情况下，  

// 该数组能否变成一个非递减数列。
// 非递减数列的定义是：对于数组中任意的 i (0 <= i <= n-2)，总满足 nums[i] <= nums[i+1]。  

// 链接: https://leetcode.cn/problems/non-decreasing-array/
public static boolean checkPossibility(int[] nums) {
    if (nums == null || nums.length <= 1) {
        return true; // 空数组或只有一个元素是非递减的
    }

    int count = 0; // 记录需要修改的次数

    for (int i = 0; i < nums.length - 1; i++) {
        if (nums[i] > nums[i + 1]) {
            count++;
            if (count > 1) {
                return false; // 需要修改超过 1 次
            }
        }
    }

    // 贪心策略：尽可能修改 nums[i] 而不是 nums[i+1]，这样对后续影响更小
    // 但是如果 nums[i-1] > nums[i+1]，则必须修改 nums[i+1]
    if (i > 0 && nums[i - 1] > nums[i + 1]) {
        nums[i + 1] = nums[i]; // 修改 nums[i+1]
    } else {
        nums[i] = nums[i + 1]; // 修改 nums[i]
    }
}

```

```
    }  
  
    return true;  
}  
}
```

---

文件: Code08\_JumpGame.py

---

```
# 跳跃游戏 (Jump Game)  
# 给定一个非负整数数组 nums , 你最初位于数组的第一个下标。  
# 数组中的每个元素代表你在该位置可以跳跃的最大长度。  
# 判断你是否能够到达最后一个下标。  
#  
# 算法标签: 贪心算法(Greedy Algorithm)、数组遍历(Array Traversal)  
# 时间复杂度: O(n), 其中 n 是数组长度  
# 空间复杂度: O(1), 仅使用常数额外空间  
# 测试链接 : https://leetcode.cn/problems/jump-game/  
# 相关题目: LeetCode 45. 跳跃游戏 II、LeetCode 1306. 跳跃游戏 III  
# 贪心算法专题 - 补充题目收集与详解
```

"""

算法思路详解:

1. 贪心策略: 维护能到达的最远位置
  - 这个策略的核心思想是动态维护一个可达范围
  - 通过不断扩展可达范围来判断是否能到达终点
2. 遍历数组, 更新能到达的最远位置
  - 对于每个位置 i, 能到达的最远位置是  $\max(\text{max\_reach}, i + \text{nums}[i])$
  - 这表示从位置 i 出发, 最远能到达  $i + \text{nums}[i]$  位置
3. 如果当前位置超过了能到达的最远位置, 则无法到达终点
  - 这意味着存在一个“断点”, 无法继续前进
  - 此时可以直接返回 False
4. 如果能到达的最远位置大于等于最后一个下标, 则能到达终点
  - 表示终点在可达范围内
  - 可以提前返回 True

时间复杂度分析:

- 遍历时间复杂度: O(n), 其中 n 是数组长度
- 总体时间复杂度: O(n)

空间复杂度分析:

- 只使用了常数额外空间存储变量
- 空间复杂度:  $O(1)$

是否最优解:

- 是, 这是处理此类问题的最优解法
- 贪心策略保证了局部最优解能导致全局最优解

工程化最佳实践:

1. 异常处理: 检查输入是否为空或格式不正确
2. 边界条件: 处理空数组、单个元素等特殊情况
3. 性能优化: 一次遍历完成计算, 提前终止条件避免不必要的计算
4. 可读性: 清晰的变量命名和详细注释, 便于维护

极端场景与边界情况处理:

1. 空输入: `nums` 为空数组
2. 极端值: 只有一个元素、所有元素都是 0
3. 重复数据: 多个元素相同
4. 有序/逆序数据: 元素值递增或递减

跨语言实现差异与优化:

1. Java: 使用增强 for 循环遍历数组, 代码更简洁
2. C++: 使用传统 for 循环, 性能更优
3. Python: 使用 for 循环或 `enumerate`, 语法更灵活

调试与测试策略:

1. 打印中间过程: 在循环中打印当前位置和最远可达位置
2. 用断言验证中间结果: 确保最远位置不减小
3. 性能退化排查: 检查是否只遍历了一次数组
4. 边界测试: 测试空数组、单元素等边界情况

实际应用场景与拓展:

1. 路径规划问题: 在游戏开发中判断角色是否能到达目标位置
2. 图论算法: 作为初始解提供给更复杂的算法
3. 网络路由: 用于快速判断网络节点间的连通性

算法深入解析:

贪心算法在跳跃游戏问题中的应用体现了其核心思想:

1. 局部最优选择: 每次选择能到达的最远位置
2. 无后效性: 当前的选择不会影响之前的状态
3. 最优子结构: 问题的最优解包含子问题的最优解

这个问题的关键洞察是, 我们不需要关心具体如何跳跃, 只需要关心能到达的最远位置。

```
"""
```

```
def canJump(nums):  
    """  
        跳跃游戏主函数 - 使用贪心算法判断是否能到达最后一个下标  
    """
```

算法思路：

1. 维护能到达的最远位置
2. 遍历数组，更新能到达的最远位置
3. 如果当前位置超过了能到达的最远位置，则无法到达终点

Args:

nums (List[int]): 非负整数数组，表示每个位置可以跳跃的最大长度  
nums[i] 表示在位置 i 可以跳跃的最大长度

Returns:

bool: 是否能够到达最后一个下标

时间复杂度: O(n)，其中 n 是数组长度

空间复杂度: O(1)，仅使用常数额外空间

Examples:

```
>>> canJump([2, 3, 1, 1, 4])
```

```
True
```

```
>>> canJump([3, 2, 1, 0, 4])
```

```
False
```

```
"""
```

# 异常处理：检查输入是否为空

```
if not nums:
```

```
    return False
```

# 边界条件：只有一个元素，肯定能到达

```
if len(nums) == 1:
```

```
    return True
```

```
max_reach = 0 # 能到达的最远位置
```

# 遍历数组

# 时间复杂度: O(n)

```
for i in range(len(nums)):
```

# 如果当前位置超过了能到达的最远位置，则无法到达终点

```
    if i > max_reach:
```

```

        return False

    # 更新能到达的最远位置
    max_reach = max(max_reach, i + nums[i])

    # 如果能到达的最远位置大于等于最后一个下标，则能到达终点
    if max_reach >= len(nums) - 1:
        return True

    return False

# 测试函数
if __name__ == "__main__":
    # 测试用例 1：可以到达终点
    nums1 = [2, 3, 1, 1, 4]
    print("测试用例 1 结果:", canJump(nums1))  # 期望输出: True

    # 测试用例 2：无法到达终点
    nums2 = [3, 2, 1, 0, 4]
    print("测试用例 2 结果:", canJump(nums2))  # 期望输出: False

    # 测试用例 3：边界情况 - 单个元素
    nums3 = [0]
    print("测试用例 3 结果:", canJump(nums3))  # 期望输出: True

    # 测试用例 4：极端情况 - 无法前进
    nums4 = [1, 0, 1, 0]
    print("测试用例 4 结果:", canJump(nums4))  # 期望输出: False

    # 测试用例 5：全为 1 - 可以到达
    nums5 = [1, 1, 1, 1, 1]
    print("测试用例 5 结果:", canJump(nums5))  # 期望输出: True

# 补充题目 1: LeetCode 134. 加油站
# 题目描述: 在一条环路上有 n 个加油站，其中第 i 个加油站有汽油 gas[i] 升。
# 你有一辆油箱容量无限的汽车，从第 i 个加油站开往第 i+1 个加油站需要消耗汽油 cost[i] 升。
# 你从其中的一个加油站出发，开始时油箱为空。
# 如果你可以绕环路行驶一周，则返回出发时加油站的编号，否则返回 -1。
# 注意: 如果题目有解，该答案即为唯一答案。
# 链接: https://leetcode.cn/problems/gas-station/

```

```
def canCompleteCircuit(gas, cost):
    """
    判断是否可以绕环路行驶一周，并返回起始加油站的编号 - 使用贪心算法
    """
```

算法思路：

1. 如果总油量小于总消耗，肯定无法绕行一周
2. 贪心策略：如果从 A 到 B 的路上没油了，那么 A 到 B 之间的任何一个站点都不能作为起点

Args:

gas (List[int]): 加油站汽油储量的数组，gas[i] 表示第 i 个加油站的汽油量

cost (List[int]): 每段路程消耗的汽油量数组，cost[i] 表示从第 i 个加油站到第 i+1 个加油站的消耗

Returns:

int: 起始加油站编号，如果无法绕行则返回-1

时间复杂度: O(n)，只需遍历一次数组

空间复杂度: O(1)，只使用常数额外空间

工程化考量：

1. 异常处理：检查输入是否为空或长度不匹配
2. 边界条件：处理空数组等特殊情况
3. 性能优化：一次遍历完成计算

"""

# 异常处理：检查输入是否为空或长度不匹配

```
if not gas or not cost or len(gas) != len(cost):
    return -1 # 输入不合法
```

```
n = len(gas)
totalGas = 0      # 总油量
totalCost = 0     # 总消耗
currentGas = 0   # 当前剩余油量
startStation = 0  # 起始加油站
```

# 贪心策略：如果从 A 到 B 的路上没油了，那么 A 到 B 之间的任何一个站点都不能作为起点

# 时间复杂度: O(n)

```
for i in range(n):
    totalGas += gas[i]
    totalCost += cost[i]
    currentGas += gas[i] - cost[i]
```

# 如果当前剩余油量为负，说明从 startStation 到 i 的路径不可行

```
if currentGas < 0:
```

```
startStation = i + 1 # 从下一个站点重新开始计算
currentGas = 0         # 重置当前剩余油量

# 如果总油量小于总消耗，那么无论如何都不可能绕行一周
if totalGas < totalCost:
    return -1

# 否则， startStation 就是答案
return startStation
```

```
# 补充题目 2: LeetCode 561. 数组拆分 I
# 题目描述: 给定长度为 2n 的整数数组 nums , 你的任务是将这些数分成 n 对,
# 例如 (a1, b1), (a2, b2), ..., (an, bn) , 使得从 1 到 n 的 min(ai, bi) 总和最大。
# 返回该 最大总和 。
# 链接: https://leetcode.cn/problems/array-partition-i/
```

```
def arrayPairSum(nums):
    """
    将数组分成 n 对，使得 min(ai, bi) 的总和最大 - 使用贪心算法
    
```

算法思路:

1. 将数组排序
2. 贪心策略: 每两个相邻的数分为一组, 取较小的那个 (即每对中的第一个数)

Args:

nums (List[int]): 整数数组, 长度为 2n

Returns:

int: 最大总和

时间复杂度:  $O(n \log n)$ , 主要是排序的复杂度

空间复杂度:  $O(1)$ , 只使用常数额外空间 (假设排序算法使用原地排序)

工程化考量:

1. 异常处理: 检查输入是否为空或长度不是偶数
2. 边界条件: 处理空数组等特殊情况
3. 性能优化: 排序后使用贪心策略

```
"""
# 异常处理: 检查输入是否为空或长度不是偶数
if not nums or len(nums) % 2 != 0:
    return 0 # 输入不合法
```

```
# 贪心策略：将数组排序后，每两个相邻的数分为一组，取较小的那个（即每对中的第一个数）
# 时间复杂度：O(n log n)
nums.sort()
maxSum = 0

# 每隔一个元素取一个（即每对中的第一个元素）
# 时间复杂度：O(n)
for i in range(0, len(nums), 2):
    maxSum += nums[i]

return maxSum
```

```
# 补充题目 3：LeetCode 402. 移掉 K 位数字
# 题目描述：给你一个以字符串表示的非负整数 num 和一个整数 k ，
# 移除这个数中的 k 位数字，使得剩下的数字最小。
# 请你以字符串形式返回这个最小的数字。
# 链接：https://leetcode.cn/problems/remove-k-digits/
```

```
def removeKdigits(num, k):
    """
    移掉 k 位数字，使得剩下的数字最小 - 使用贪心算法+单调栈
```

算法思路：

1. 使用栈来存储需要保留的数字
2. 贪心策略：从左到右遍历，如果当前数字小于栈顶数字，且还有删除次数，则弹出栈顶数字

Args:

num (str)：表示非负整数的字符串  
k (int)：需要移除的数字个数

Returns:

str：最小数字的字符串表示

时间复杂度：O(n)，每个字符最多入栈和出栈一次

空间复杂度：O(n)，使用栈存储保留的数字

工程化考量：

1. 异常处理：检查输入是否为空或 k 过大
2. 边界条件：处理空字符串、k 等于字符串长度等情况
3. 性能优化：使用单调栈避免重复操作

```
"""
# 异常处理：检查输入是否为空或 k 过大
```

```

if not num or len(num) <= k:
    return "0" # 移除所有数字, 返回 0

# 使用栈来存储需要保留的数字
stack = []

# 贪心策略: 从左到右遍历, 如果当前数字小于栈顶数字, 且还有删除次数, 则弹出栈顶数字
# 时间复杂度: O(n)
for digit in num:
    # 当栈不为空, 当前数字小于栈顶数字, 且还有删除次数时, 弹出栈顶数字
    while stack and digit < stack[-1] and k > 0:
        stack.pop()
        k -= 1
    # 将当前数字入栈
    stack.append(digit)

# 如果还有删除次数, 从栈顶删除
while k > 0 and stack:
    stack.pop()
    k -= 1

# 构建结果字符串, 去除前导零
result = ''.join(stack).lstrip('0')

# 如果全是零, 返回"0"
return result if result else "0"

```

```

# 补充题目 4: LeetCode 122. 买卖股票的最佳时机 II (另一种贪心实现)
# 题目描述: 给定一个数组 prices , 其中 prices[i] 表示股票第 i 天的价格。
# 在每一天, 你可能会决定购买和/或出售股票。你在任何时候 最多 只能持有 一股 股票。
# 你也可以购买它, 然后在 同一天 出售。
# 返回 你能获得的 最大 利润 。
# 链接: https://leetcode.cn/problems/best-time-to-buy-and-sell-stock-ii/

```

```

def maxProfit(prices):
    """
    计算买卖股票的最大利润, 允许多次交易 - 使用贪心算法

```

算法思路:

1. 贪心策略: 只要后一天的价格比前一天高, 就进行一次买卖

Args:

prices (List[int]): 股票每天价格的数组, prices[i] 表示第 i 天的股票价格

Returns:

int: 最大利润

时间复杂度: O(n), 只需遍历一次数组

空间复杂度: O(1), 只使用常数额外空间

工程化考量:

1. 异常处理: 检查输入是否为空或长度不足
2. 边界条件: 处理空数组、单元素等情况
3. 性能优化: 一次遍历完成计算

"""

# 异常处理: 检查输入是否为空或长度不足

```
if not prices or len(prices) <= 1:  
    return 0 # 无法交易
```

max\_profit = 0

# 贪心策略: 只要后一天的价格比前一天高, 就进行一次买卖

# 时间复杂度: O(n)

```
for i in range(1, len(prices)):  
    # 如果当天价格比前一天高, 就进行交易  
    if prices[i] > prices[i - 1]:  
        max_profit += prices[i] - prices[i - 1]
```

return max\_profit

# 补充题目 5: LeetCode 665. 非递减数列

# 题目描述: 给你一个长度为 n 的整数数组 nums , 请你判断在 最多 改变 1 个元素的情况下,

# 该数组能否变成一个非递减数列。

# 非递减数列的定义是: 对于数组中任意的 i (0 <= i <= n-2), 总满足 nums[i] <= nums[i+1]。

# 链接: <https://leetcode.cn/problems/non-decreasing-array/>

def checkPossibility(nums):

"""

判断是否可以通过最多修改一个元素, 使数组成为非递减数列 - 使用贪心算法

算法思路:

1. 遍历数组, 统计需要修改的次数
2. 贪心策略: 尽可能修改 nums[i] 而不是 nums[i+1], 这样对后续影响更小

Args:

nums (List[int]): 整数数组

Returns:

bool: 是否可以满足条件

时间复杂度:  $O(n)$ , 只需遍历一次数组

空间复杂度:  $O(1)$ , 只使用常数额外空间

工程化考量:

1. 异常处理: 检查输入是否为空
2. 边界条件: 处理空数组、单元素等情况
3. 性能优化: 提前终止条件避免不必要的计算

"""

```
# 异常处理: 检查输入是否为空
if not nums or len(nums) <= 1:
    return True # 空数组或只有一个元素是非递减的
```

```
count = 0 # 记录需要修改的次数
```

```
# 时间复杂度:  $O(n)$ 
for i in range(len(nums) - 1):
    if nums[i] > nums[i + 1]:
        count += 1
    if count > 1:
        return False # 需要修改超过 1 次
```

```
# 贪心策略: 尽可能修改 nums[i] 而不是 nums[i+1], 这样对后续影响更小
```

```
# 但是如果 nums[i-1] > nums[i+1], 则必须修改 nums[i+1]
```

```
if i > 0 and nums[i - 1] > nums[i + 1]:
    nums[i + 1] = nums[i] # 修改 nums[i+1]
else:
    nums[i] = nums[i + 1] # 修改 nums[i]
```

```
return True
```

文件: Code09\_NonOverlappingIntervals.cpp

```
// 无重叠区间
```

```
// 给定一个区间的集合, 找到需要移除区间的最小数量, 使剩余区间互不重叠。
```

```
// 测试链接 : https://leetcode.cn/problems/non-overlapping-intervals/
```

```
// 贪心算法专题 - 区间调度问题集合
```

```
/*
 * 算法思路:
 * 1. 贪心策略: 按区间结束位置排序, 优先选择结束位置早的区间
 * 2. 排序后遍历区间, 统计不重叠的区间数量
 * 3. 总区间数减去不重叠区间数就是需要移除的区间数
 *
 * 时间复杂度: O(n * logn) - 主要是排序的时间复杂度
 * 空间复杂度: O(1) - 只使用了常数额外空间
 * 是否最优解: 是, 这是处理此类问题的最优解法
 *
 * 工程化考量:
 * 1. 异常处理: 检查输入是否为空
 * 2. 边界条件: 处理空数组、单个区间等特殊情况
 * 3. 性能优化: 使用贪心策略避免动态规划
 * 4. 可读性: 清晰的变量命名和注释
 *
 * 极端场景与边界场景:
 * 1. 空输入: intervals 为空数组
 * 2. 极端值: 只有一个区间、所有区间相同
 * 3. 重复数据: 多个区间相同
 * 4. 有序/逆序数据: 区间按开始位置或结束位置排序
 *
 * 跨语言场景与语言特性差异:
 * 1. Java: 使用 Arrays.sort 进行排序
 * 2. C++: 使用 std::sort 进行排序
 * 3. Python: 使用 sorted 函数或 list.sort() 方法
 *
 * 调试能力构建:
 * 1. 打印中间过程: 在循环中打印当前区间和上一个选择的区间
 * 2. 用断言验证中间结果: 确保选择的区间不重叠
 * 3. 性能退化排查: 检查排序和遍历的时间复杂度
 *
 * 与机器学习、图像处理、自然语言处理的联系与应用:
 * 1. 在时间调度问题中, 贪心算法可用于优化资源分配
 * 2. 在计算机视觉中, 可用于非极大值抑制(NMS)算法
 * 3. 在自然语言处理中, 可用于实体识别中的重叠实体处理
 */
```

```
// 简单的区间结构体
struct Interval {
    int start;
    int end;
```

```

};

// 简单的排序函数实现（冒泡排序）
void bubbleSort(Interval arr[], int n) {
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (arr[j].end > arr[j + 1].end) {
                // 交换元素
                Interval temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
}

// 无重叠区间主函数
int eraseOverlapIntervals(Interval intervals[], int intervalsSize) {
    // 异常处理：检查输入是否为空
    if (intervals == 0 || intervalsSize == 0) {
        return 0;
    }

    // 边界条件：只有一个区间，不需要移除
    if (intervalsSize == 1) {
        return 0;
    }

    // 使用冒泡排序按区间结束位置排序
    bubbleSort(intervals, intervalsSize);

    int count = 1;           // 不重叠区间数量，初始选择第一个区间
    int end = intervals[0].end; // 上一个选择区间的结束位置

    // 遍历排序后的区间
    for (int i = 1; i < intervalsSize; i++) {
        // 如果当前区间与上一个选择的区间不重叠
        if (intervals[i].start >= end) {
            count++;           // 不重叠区间数加 1
            end = intervals[i].end; // 更新结束位置
        }
    }
}

```

```

// 需要移除的区间数 = 总区间数 - 不重叠区间数
return intervalsSize - count;
}

// 补充题目 1: LeetCode 56. 合并区间
// 题目描述: 以数组 intervals 表示若干个区间的集合, 其中单个区间为 intervals[i] = [starti, endi] 。
// 请你合并所有重叠的区间, 并返回一个不重叠的区间数组, 该数组需恰好覆盖输入中的所有区间。
// 链接: https://leetcode.cn/problems/merge-intervals/

int** merge(int** intervals, int intervalsSize, int* intervalsColSize, int* returnSize, int** returnColumnSizes) {
    if (intervals == NULL || intervalsSize <= 1) {
        *returnSize = intervalsSize;
        *returnColumnSizes = (int*)malloc(intervalsSize * sizeof(int));
        for (int i = 0; i < intervalsSize; i++) {
            (*returnColumnSizes)[i] = 2;
        }
        return intervals; // 空数组或只有一个区间, 无需合并
    }

    // 贪心策略: 按区间起始位置排序, 然后依次合并重叠区间
    sort(intervals, intervals + intervalsSize, [] (int* a, int* b) {
        return a[0] < b[0];
    });

    // 使用 vector 存储合并后的区间
    vector<int*> merged;
    merged.push_back(intervals[0]); // 添加第一个区间

    // 遍历其余区间
    for (int i = 1; i < intervalsSize; i++) {
        int* last = merged.back(); // 获取上一个合并后的区间
        int* current = intervals[i]; // 当前区间

        // 如果当前区间与上一个合并后的区间重叠, 则合并它们
        if (current[0] <= last[1]) {
            // 更新合并后的区间结束位置为两个区间结束位置的较大值
            last[1] = max(last[1], current[1]);
        } else {
            // 否则直接添加当前区间
            merged.push_back(current);
        }
    }

    *returnSize = merged.size();
    *returnColumnSizes = (int*)malloc(merged.size() * sizeof(int));
    for (int i = 0; i < merged.size(); i++) {
        (*returnColumnSizes)[i] = 2;
    }
    return merged.data();
}

```

```

}

// 准备返回结果
*returnSize = merged.size();
*returnColumnSizes = (int*)malloc(*returnSize * sizeof(int));
int** result = (int**)malloc(*returnSize * sizeof(int*));
for (int i = 0; i < *returnSize; i++) {
    result[i] = (int*)malloc(2 * sizeof(int));
    result[i][0] = merged[i][0];
    result[i][1] = merged[i][1];
    (*returnColumnSizes)[i] = 2;
}

return result;
}

// 补充题目 2: LeetCode 57. 插入区间
// 题目描述: 给你一个 无重叠的 , 按照区间起始端点排序的区间列表。
// 在列表中插入一个新的区间, 你需要确保列表中的区间仍然有序且不重叠 (如果有必要的话, 可以合并区间)。
// 链接: https://leetcode.cn/problems/insert-interval/

int** insert(int** intervals, int intervalsSize, int* intervalsColSize, int* newInterval, int newIntervalSize, int* returnSize, int** returnColumnSizes) {
    vector<int*> result;

    int i = 0;
    int n = intervalsSize;

    // 添加所有在新区间之前且不重叠的区间
    while (i < n && intervals[i][1] < newInterval[0]) {
        result.push_back(intervals[i]);
        i++;
    }

    // 合并所有与新区间重叠的区间
    while (i < n && intervals[i][0] <= newInterval[1]) {
        newInterval[0] = min(newInterval[0], intervals[i][0]);
        newInterval[1] = max(newInterval[1], intervals[i][1]);
        i++;
    }

    // 添加合并后的区间
    result.push_back(newInterval);
}

```

```

result.push_back(newInterval);

// 添加剩余的区间
while (i < n) {
    result.push_back(intervals[i]);
    i++;
}

// 准备返回结果
*returnSize = result.size();
*returnColumnSizes = (int*)malloc(*returnSize * sizeof(int));
int** returnIntervals = (int**)malloc(*returnSize * sizeof(int*));
for (int j = 0; j < *returnSize; j++) {
    returnIntervals[j] = (int*)malloc(2 * sizeof(int));
    returnIntervals[j][0] = result[j][0];
    returnIntervals[j][1] = result[j][1];
    (*returnColumnSizes)[j] = 2;
}

return returnIntervals;
}

```

// 补充题目 3: LeetCode 452. 用最少数量的箭引爆气球  
// 题目描述: 有一些球形气球贴在一堵用 XY 平面表示的墙面上。气球可以用区间表示为 [start, end],  
// 飞镖必须从整个区间的内部穿过才能引爆气球。求解把所有气球射爆所需的最小飞镖数。  
// 链接: <https://leetcode.cn/problems/minimum-number-of-arrows-to-burst-balloons/>

```

int findMinArrowShots(int** points, int pointsSize, int* pointsColSize) {
    if (points == NULL || pointsSize == 0) {
        return 0; // 没有气球, 需要 0 支箭
    }
}

```

// 贪心策略: 按区间结束位置排序, 尽可能引爆更多气球  
sort(points, points + pointsSize, [] (int\* a, int\* b) {
 // 注意处理整数溢出, 使用 long long 比较
 return (long long)a[1] < (long long)b[1];
});

```

int count = 1; // 需要的箭数, 初始为 1
long long end = points[0][1]; // 第一支箭的位置, 使用 long long 防止溢出

```

// 遍历排序后的区间  
for (int i = 1; i < pointsSize; i++) {

```

// 如果当前气球的开始位置大于上一支箭的位置，需要一支新箭
if ((long long)points[i][0] > end) {
    count++;
    end = points[i][1];
}
// 否则，当前气球会被上一支箭引爆，不需要额外的箭
}

return count;
}

```

// 补充题目 4: LeetCode 986. 区间列表的交集  
// 题目描述: 给定两个由一些 闭区间 组成的列表, firstList 和 secondList,  
// 其中 firstList[i] = [starti, endi] 而 secondList[j] = [startj, endj]。  
// 每个列表中的区间是不相交的，并且已经排序。  
// 返回这 两个区间列表的交集 。  
// 链接: <https://leetcode.cn/problems/interval-list-intersections/>

```

int** intervalIntersection(int** firstList, int firstListSize, int* firstListColSize, int** secondList, int secondListSize, int* secondListColSize, int* returnSize, int** returnColumnSizes)
{
    if (firstList == NULL || secondList == NULL || firstListSize == 0 || secondListSize == 0) {
        *returnSize = 0;
        return NULL; // 任一列表为空，交集为空
    }

    vector<int*> result;
    int i = 0, j = 0; // 两个指针分别指向两个列表
    int m = firstListSize, n = secondListSize;

    // 双指针遍历两个列表
    while (i < m && j < n) {
        int start = max(firstList[i][0], secondList[j][0]); // 交集的起始位置
        int end = min(firstList[i][1], secondList[j][1]); // 交集的结束位置

        // 如果有交集
        if (start <= end) {
            int* interval = (int*)malloc(2 * sizeof(int));
            interval[0] = start;
            interval[1] = end;
            result.push_back(interval);
        }
        i++;
        j++;
    }

    *returnSize = result.size();
    *returnColumnSizes = new int[m + n];
    for (int i = 0; i < m + n; i++) {
        (*returnColumnSizes)[i] = 2;
    }
    return result.data();
}

```

```

// 移动结束位置较小的区间的指针
if (firstList[i][1] < secondList[j][1]) {
    i++;
} else {
    j++;
}
}

// 准备返回结果
*returnSize = result.size();
*returnColumnSizes = (int*)malloc(*returnSize * sizeof(int));
int** returnIntervals = (int**)malloc(*returnSize * sizeof(int*));
for (int k = 0; k < *returnSize; k++) {
    returnIntervals[k] = result[k];
    (*returnColumnSizes)[k] = 2;
}

return returnIntervals;
}

// 补充题目 5: LeetCode 1288. 删除被覆盖区间
// 题目描述: 给你一个区间列表, 请你删除列表中被其他区间完全覆盖的区间。
// 只有当 c <= a 且 b <= d 时, 我们才认为区间 [a, b] 被区间 [c, d] 覆盖。
// 在完成所有删除操作后, 请你返回列表中剩余区间的数目。
// 链接: https://leetcode.cn/problems/remove-covered-intervals/

int removeCoveredIntervals(int** intervals, int intervalsSize, int* intervalsColSize) {
    if (intervals == NULL || intervalsSize <= 1) {
        return intervals == NULL ? 0 : intervalsSize; // 空数组或只有一个区间
    }

    // 贪心策略: 按起始位置升序排序, 起始位置相同时按结束位置降序排序
    sort(intervals, intervals + intervalsSize, [] (int* a, int* b) {
        if (a[0] != b[0]) {
            return a[0] < b[0];
        } else {
            return b[1] < a[1]; // 起始位置相同时, 结束位置降序
        }
    });

    int count = 1; // 剩余区间数, 至少有一个区间
    int end = intervals[0][1]; // 当前最大的结束位置

```

```

// 遍历排序后的区间
for (int i = 1; i < intervalsSize; i++) {
    // 如果当前区间的结束位置大于最大结束位置，说明不被覆盖
    if (intervals[i][1] > end) {
        count++;
        end = intervals[i][1];
    }
}

return count;
}

```

=====

文件: Code09\_NonOverlappingIntervals.java

=====

```

package class094;

import java.util.Arrays;

// 无重叠区间 (Non-overlapping Intervals)
// 给定一个区间的集合，找到需要移除区间的最小数量，使剩余区间互不重叠。
//
// 算法标签：贪心算法(Greedy Algorithm)、区间调度(Interval Scheduling)、排序(Sorting)
// 时间复杂度: O(n*log(n))，其中 n 是区间数量
// 空间复杂度: O(1)，仅使用常数额外空间
// 测试链接 : https://leetcode.cn/problems/non-overlapping-intervals/
// 相关题目: LeetCode 56. 合并区间、LeetCode 452. 用最少量的箭引爆气球
// 贪心算法专题 - 区间调度问题集合
public class Code09_NonOverlappingIntervals {

/*
 * 算法思路详解:
 * 1. 贪心策略核心: 按区间结束位置排序, 优先选择结束位置早的区间,
 *     这样可以为后续区间留出更多空间, 最大化不重叠区间数量
 * 2. 排序优化: 通过按结束位置排序, 将问题转化为选择最多不重叠区间
 * 3. 区间选择: 遍历排序后的区间, 统计不重叠的区间数量
 * 4. 结果计算: 总区间数减去不重叠区间数即为需要移除的区间数
 *
 * 时间复杂度分析:
 * - O(n*log(n)), 其中 n 是区间数量, 主要消耗在排序阶段
 * - 遍历阶段时间复杂度为 O(n)
 * 空间复杂度分析:

```

\* -  $O(1)$ , 仅使用了常数级别的额外空间存储 count 和 end 变量

\* 是否最优解: 是, 这是处理此类区间调度问题的最优解法

\*

\* 工程化最佳实践:

\* 1. 输入验证: 严格检查输入参数的有效性, 防止空指针异常

\* 2. 边界处理: 妥善处理各种边界情况, 如空数组、单区间等

\* 3. 性能优化: 采用贪心策略避免复杂的动态规划解法

\* 4. 代码可读性: 使用语义明确的变量名和详尽的注释

\* 5. 比较器优化: 注意处理整数溢出问题

\*

\* 极端场景与边界情况处理:

\* 1. 空输入场景: intervals 为空数组或 null 时直接返回 0

\* 2. 单区间场景: 只有一个区间时不需要移除

\* 3. 重复区间场景: 多个相同区间的处理

\* 4. 特殊序列场景: 区间按开始位置或结束位置排序的情况

\* 5. 嵌套区间场景: 大区间包含小区间的处理

\* 6. 相邻区间场景: 区间端点相接但不重叠的情况

\*

\* 跨语言实现差异与优化:

\* 1. Java 实现: 使用 Arrays.sort 和 Lambda 表达式进行排序

\* 2. C++ 实现: 使用 std::sort 和自定义比较函数

\* 3. Python 实现: 使用 sorted 函数和 key 参数

\* 4. 内存管理: 不同语言的垃圾回收机制对性能的影响

\*

\* 调试与测试策略:

\* 1. 过程可视化: 在关键节点打印当前区间和已选择区间

\* 2. 断言验证: 在循环体内添加断言确保区间不重叠

\* 3. 性能监控: 跟踪排序和遍历的实际执行时间

\* 4. 边界测试: 设计覆盖所有边界条件的测试用例

\* 5. 压力测试: 使用大规模数据验证算法稳定性

\*

\* 实际应用场景与拓展:

\* 1. 会议调度: 会议室资源的最优分配

\* 2. 任务调度: CPU 任务的时间片分配

\* 3. 广告投放: 广告位的最优选择

\* 4. 课程安排: 教室资源的合理利用

\* 5. 网络带宽: 数据传输的时隙分配

\*

\* 算法深入解析:

\* 1. 贪心策略原理: 选择结束位置早的区间是最优的局部选择

\* 2. 最优性证明: 通过交换论证法可以证明贪心策略的正确性

\* 3. 策略变体: 可按开始位置排序选择结束位置晚的区间

\* 4. 问题转换: 最小移除数 = 总数 - 最大不重叠数

```

*/
public static int eraseOverlapIntervals(int[][] intervals) {
    // 异常处理：检查输入是否为空
    if (intervals == null || intervals.length == 0) {
        return 0;
    }

    // 边界条件：只有一个区间，不需要移除
    if (intervals.length == 1) {
        return 0;
    }

    // 按区间结束位置排序
    Arrays.sort(intervals, (a, b) -> a[1] - b[1]);

    int count = 1;          // 不重叠区间数量，初始选择第一个区间
    int end = intervals[0][1]; // 上一个选择区间的结束位置

    // 遍历排序后的区间
    for (int i = 1; i < intervals.length; i++) {
        // 如果当前区间与上一个选择的区间不重叠
        if (intervals[i][0] >= end) {
            count++;           // 不重叠区间数加 1
            end = intervals[i][1]; // 更新结束位置
        }
    }

    // 需要移除的区间数 = 总区间数 - 不重叠区间数
    return intervals.length - count;
}

// 测试函数
public static void main(String[] args) {
    // 测试用例 1
    int[][] intervals1 = {{1, 2}, {2, 3}, {3, 4}, {1, 3}};
    System.out.println("测试用例 1 结果: " + eraseOverlapIntervals(intervals1)); // 期望输出: 1

    // 测试用例 2
    int[][] intervals2 = {{1, 2}, {1, 2}, {1, 2}};
    System.out.println("测试用例 2 结果: " + eraseOverlapIntervals(intervals2)); // 期望输出: 2

    // 测试用例 3
    int[][] intervals3 = {{1, 2}, {2, 3}};

```

```

System.out.println("测试用例 3 结果: " + eraseOverlapIntervals(intervals3)); // 期望输出: 0

// 测试用例 4: 边界情况
int[][] intervals4 = {};
System.out.println("测试用例 4 结果: " + eraseOverlapIntervals(intervals4)); // 期望输出: 0

// 测试用例 5: 极端情况
int[][] intervals5 = {{1, 100}, {2, 3}, {4, 5}, {6, 7}};
System.out.println("测试用例 5 结果: " + eraseOverlapIntervals(intervals5)); // 期望输出: 1
}

// 补充题目 1: LeetCode 56. 合并区间 (Merge Intervals)
// 题目描述: 以数组 intervals 表示若干个区间的集合, 其中单个区间为 intervals[i] = [starti, endi]。
// 请你合并所有重叠的区间, 并返回一个不重叠的区间数组, 该数组需恰好覆盖输入中的所有区间。
//
// 算法标签: 贪心算法(Greedy Algorithm)、区间合并(Interval Merging)、排序(Sorting)
// 时间复杂度: O(n*log(n)), 其中 n 是区间数量
// 空间复杂度: O(n), 需要额外空间存储合并后的区间
// 链接: https://leetcode.cn/problems/merge-intervals/
/*
 * 算法详解与策略分析:
 * 1. 贪心策略核心: 按区间起始位置排序, 然后依次合并重叠区间,
 *     这样可以确保所有可能重叠的区间都被正确合并
 * 2. 合并判断: 当前区间与上一个合并后的区间重叠当且仅当
 *     当前区间起始位置 ≤ 上一个区间结束位置
 * 3. 合并操作: 更新合并后区间的结束位置为两个区间结束位置的最大值
 *
 * 算法步骤详解:
 * 1. 预处理: 按区间起始位置升序排序
 * 2. 初始化: 将第一个区间添加到结果列表
 * 3. 合并过程:
 *     - 获取结果列表中的最后一个区间
 *     - 若当前区间与最后区间重叠则合并
 *     - 否则直接添加当前区间到结果列表
 * 4. 结果生成: 将 List 转换为数组返回
 *
 * 算法优化与正确性:
 * 贪心选择性质: 按起始位置排序后顺序处理是最优的
 * 最优子结构: 合并完前 k 个区间后, 剩余问题仍保持最优性
 */
public static int[][] merge(int[][] intervals) {
    if (intervals == null || intervals.length <= 1) {

```

```

        return intervals; // 空数组或只有一个区间，无需合并
    }

    // 贪心策略：按区间起始位置排序，然后依次合并重叠区间
    Arrays.sort(intervals, (a, b) -> a[0] - b[0]);

    // 使用动态数组存储合并后的区间
    java.util.List<int[]> merged = new java.util.ArrayList<>();
    merged.add(intervals[0]); // 添加第一个区间

    // 遍历其余区间
    for (int i = 1; i < intervals.length; i++) {
        int[] last = merged.get(merged.size() - 1); // 获取上一个合并后的区间
        int[] current = intervals[i]; // 当前区间

        // 如果当前区间与上一个合并后的区间重叠，则合并它们
        if (current[0] <= last[1]) {
            // 更新合并后的区间结束位置为两个区间结束位置的较大值
            last[1] = Math.max(last[1], current[1]);
        } else {
            // 否则直接添加当前区间
            merged.add(current);
        }
    }

    // 转换为二维数组返回
    return merged.toArray(new int[merged.size()][]);
}

// 补充题目 2: LeetCode 57. 插入区间 (Insert Interval)
// 题目描述：给你一个 无重叠的 ，按照区间起始端点排序的区间列表。
// 在列表中插入一个新的区间，你需要确保列表中的区间仍然有序且不重叠（如果有必要的话，可以合并区间）。
//
// 算法标签：贪心算法(Greedy Algorithm)、区间插入(Interval Insertion)、三段处理(Three-phase Processing)
// 时间复杂度：O(n)，其中 n 是区间数量
// 空间复杂度：O(n)，需要额外空间存储结果
// 链接：https://leetcode.cn/problems/insert-interval/
/*
 * 算法详解与策略分析：
 * 1. 三段处理策略：将问题分解为三个阶段：
 *      - 添加所有在新区间之前且不重叠的区间
 */

```

```

*      - 合并所有与新区间重叠的区间
*      - 添加剩余的区间
* 2. 合并机制：通过更新新区间的边界来实现区间合并
* 3. 有序性维护：利用输入区间列表已排序的特性
*
* 算法步骤详解：
* 1. 初始化：创建结果列表和双指针
* 2. 前段处理：添加所有在新区间之前且不重叠的区间
* 3. 中段合并：合并所有与新区间重叠的区间
* 4. 后段处理：添加剩余的区间
* 5. 结果返回：将 List 转换为数组返回
*
* 算法优化与边界处理：
* 边界情况：处理新区间在最前、最后或中间位置的情况
* 时间优化：利用已排序特性避免排序操作
*/
public static int[][] insert(int[][] intervals, int[] newInterval) {
    if (intervals == null) {
        return new int[][] {newInterval}; // 空数组，直接返回新区间
    }

    java.util.List<int[]> result = new java.util.ArrayList<>();
    int i = 0;
    int n = intervals.length;

    // 添加所有在新区间之前且不重叠的区间
    while (i < n && intervals[i][1] < newInterval[0]) {
        result.add(intervals[i]);
        i++;
    }

    // 合并所有与新区间重叠的区间
    while (i < n && intervals[i][0] <= newInterval[1]) {
        newInterval[0] = Math.min(newInterval[0], intervals[i][0]);
        newInterval[1] = Math.max(newInterval[1], intervals[i][1]);
        i++;
    }

    // 添加合并后的区间
    result.add(newInterval);

    // 添加剩余的区间
    while (i < n) {

```

```

        result.add(intervals[i]);
        i++;
    }

    return result.toArray(new int[result.size()][]);
}

// 补充题目 3: LeetCode 452. 用最少量的箭引爆气球 (Minimum Number of Arrows to Burst Balloons)

// 题目描述: 有一些球形气球贴在一堵用 XY 平面表示的墙面上。气球可以用区间表示为 [start, end]，// 飞镖必须从整个区间的内部穿过才能引爆气球。求解把所有气球射爆所需的最小飞镖数。
//
// 算法标签: 贪心算法(Greedy Algorithm)、区间覆盖(Interval Covering)、排序(Sorting)
// 时间复杂度: O(n*log(n))，其中 n 是气球数量
// 空间复杂度: O(1)，仅使用常数额外空间
// 链接: https://leetcode.cn/problems/minimum-number-of-arrows-to-burst-balloons/
/*
 * 算法详解与策略分析:
 * 1. 贪心策略核心: 按区间结束位置排序, 尽可能引爆更多气球,
 *   每支箭都射在能引爆最多气球的位置
 * 2. 覆盖判断: 当前气球能被上一支箭引爆当且仅当
 *   当前气球起始位置 ≤ 上一支箭位置
 * 3. 射箭决策: 当无法被上一支箭引爆时需要新箭
 *
 * 算法步骤详解:
 * 1. 预处理: 按区间结束位置升序排序
 * 2. 初始化: 设置箭数为 1, 第一支箭位置为第一个区间结束位置
 * 3. 射箭过程:
 *   - 若当前气球能被上一支箭引爆则跳过
 *   - 否则需要新箭, 更新箭数和箭位置
 * 4. 结果返回: 返回累计箭数
 *
 * 算法优化与实现细节:
 * 比较器优化: 使用显式比较避免整数溢出
 * 正确性保证: 贪心策略确保最少箭数
 */

public static int findMinArrowShots(int[][] points) {
    if (points == null || points.length == 0) {
        return 0; // 没有气球, 需要 0 支箭
    }

    // 贪心策略: 按区间结束位置排序, 尽可能引爆更多气球
    Arrays.sort(points, (a, b) -> {

```

```

    // 注意处理整数溢出
    if (a[1] < b[1]) return -1;
    else if (a[1] > b[1]) return 1;
    else return 0;
});

int count = 1; // 需要的箭数，初始为 1
int end = points[0][1]; // 第一支箭的位置

// 遍历排序后的区间
for (int i = 1; i < points.length; i++) {
    // 如果当前气球的开始位置大于上一支箭的位置，需要一支新箭
    if (points[i][0] > end) {
        count++;
        end = points[i][1];
    }
    // 否则，当前气球会被上一支箭引爆，不需要额外的箭
}

return count;
}

// 补充题目 4: LeetCode 986. 区间列表的交集 (Interval List Intersections)
// 题目描述: 给定两个由一些 闭区间 组成的列表, firstList 和 secondList,
// 其中 firstList[i] = [starti, endi] 而 secondList[j] = [startj, endj]。
// 每个列表中的区间是不相交的，并且已经排序。
// 返回这 两个区间列表的交集 。
//
// 算法标签: 双指针(Double Pointers)、区间交集(Interval Intersection)、有序列表处理(Sorted
List Processing)
// 时间复杂度: O(m+n)，其中 m 和 n 分别是两个列表的长度
// 空间复杂度: O(m+n)，需要额外空间存储交集结果
// 链接: https://leetcode.cn/problems/interval-list-intersections/
/*
 * 算法详解与策略分析:
 * 1. 双指针策略: 利用两个列表均已排序的特性,
 * 使用双指针分别遍历两个列表
 * 2. 交集计算: 两个区间的交集为[max(start1, start2), min(end1, end2)]
 * 3. 指针移动: 移动结束位置较小的区间的指针
 *
 * 算法步骤详解:
 * 1. 初始化: 创建双指针和结果列表
 * 2. 交集计算:

```

```

*      - 计算当前两个区间的交集起始和结束位置
*      - 若起始位置≤结束位置则存在交集
* 3. 指针移动: 移动结束位置较小的区间的指针
* 4. 结果生成: 将 List 转换为数组返回
*
* 算法优化与边界处理:
* 边界情况: 处理空列表和无交集情况
* 时间优化: 利用有序性避免重复比较
*/
public static int[][] intervalIntersection(int[][] firstList, int[][] secondList) {
    if (firstList == null || secondList == null || firstList.length == 0 || secondList.length
== 0) {
        return new int[0][];
    }

    java.util.List<int[]> result = new java.util.ArrayList<>();
    int i = 0, j = 0; // 两个指针分别指向两个列表
    int m = firstList.length, n = secondList.length;

    // 双指针遍历两个列表
    while (i < m && j < n) {
        int start = Math.max(firstList[i][0], secondList[j][0]); // 交集的起始位置
        int end = Math.min(firstList[i][1], secondList[j][1]); // 交集的结束位置

        // 如果有交集
        if (start <= end) {
            result.add(new int[]{start, end});
        }

        // 移动结束位置较小的区间的指针
        if (firstList[i][1] < secondList[j][1]) {
            i++;
        } else {
            j++;
        }
    }

    return result.toArray(new int[result.size()][]);
}

// 补充题目 5: LeetCode 1288. 删除被覆盖区间 (Remove Covered Intervals)
// 题目描述: 给你一个区间列表, 请你删除列表中被其他区间完全覆盖的区间。
// 只有当 c ≤ a 且 b ≤ d 时, 我们才认为区间 [a, b] 被区间 [c, d] 覆盖。

```

```

// 在完成所有删除操作后，请你返回列表中剩余区间的数目。
//
// 算法标签：贪心算法(Greedy Algorithm)、区间覆盖(Interval Covering)、排序(Sorting)
// 时间复杂度：O(n*log(n))，其中 n 是区间数量
// 空间复杂度：O(1)，仅使用常数额外空间
// 链接：https://leetcode.cn/problems/remove-covered-intervals/
/*
 * 算法详解与策略分析：
 * 1. 贪心策略核心：按起始位置升序排序，起始位置相同时按结束位置降序排序，
 *    这样可以确保在遍历过程中能正确识别被覆盖的区间
 * 2. 覆盖判断：当前区间被覆盖当且仅当其结束位置≤已遍历区间的最大结束位置
 * 3. 计数机制：只统计未被覆盖的区间
 *
 * 算法步骤详解：
 * 1. 预处理：按起始位置升序排序，起始位置相同时按结束位置降序排序
 * 2. 初始化：设置剩余区间数为 1，最大结束位置为第一个区间结束位置
 * 3. 遍历过程：
 *   - 若当前区间结束位置>最大结束位置则未被覆盖
 *   - 更新剩余区间数和最大结束位置
 * 4. 结果返回：返回累计剩余区间数
 *
 * 算法优化与正确性：
 * 排序策略：特殊排序确保覆盖关系正确判断
 * 贪心选择：每次选择结束位置最大的区间是最优的
 */

public static int removeCoveredIntervals(int[][] intervals) {
    if (intervals == null || intervals.length <= 1) {
        return intervals == null ? 0 : intervals.length; // 空数组或只有一个区间
    }

    // 贪心策略：按起始位置升序排序，起始位置相同时按结束位置降序排序
    Arrays.sort(intervals, (a, b) -> {
        if (a[0] != b[0]) {
            return a[0] - b[0];
        } else {
            return b[1] - a[1]; // 起始位置相同时，结束位置降序
        }
    });

    int count = 1; // 剩余区间数，至少有一个区间
    int end = intervals[0][1]; // 当前最大的结束位置

    // 遍历排序后的区间

```

```

        for (int i = 1; i < intervals.length; i++) {
            // 如果当前区间的结束位置大于最大结束位置，说明不被覆盖
            if (intervals[i][1] > end) {
                count++;
                end = intervals[i][1];
            }
        }

        return count;
    }
}

```

=====

文件: Code09\_NonOverlappingIntervals.py

=====

```

# 无重叠区间 (Non-overlapping Intervals)
# 给定一个区间的集合，找到需要移除区间的最小数量，使剩余区间互不重叠。
#
# 算法标签：贪心算法(Greedy Algorithm)、区间调度(Interval Scheduling)
# 时间复杂度：O(n * logn)，其中 n 是区间数量
# 空间复杂度：O(1)，仅使用常数额外空间
# 测试链接：https://leetcode.cn/problems/non-overlapping-intervals/
# 相关题目：LeetCode 56. 合并区间、LeetCode 452. 用最少数的箭引爆气球
# 贪心算法专题 - 补充题目收集与详解

```

"""

算法思路详解：

1. 贪心策略：按区间结束位置排序，优先选择结束位置早的区间
  - 这个策略的核心思想是为后续区间留出更多空间
  - 结束位置早的区间不会影响太多后续区间的选取
2. 排序后遍历区间，统计不重叠的区间数量
  - 通过一次遍历完成所有计算
  - 只需要比较相邻区间的重叠情况
3. 总区间数减去不重叠区间数就是需要移除的区间数
  - 这是问题的转换：最大化保留区间数等价于最小化移除区间数

时间复杂度分析：

- 排序时间复杂度：O(n \* logn)，其中 n 是区间数量
- 遍历时间复杂度：O(n)
- 总体时间复杂度：O(n \* logn)

空间复杂度分析:

- 只使用了常数额外空间存储变量
- 空间复杂度:  $O(1)$

是否最优解:

- 是, 这是处理此类问题的最优解法
- 贪心策略保证了局部最优解能导致全局最优解

工程化最佳实践:

1. 异常处理: 检查输入是否为空或格式不正确
2. 边界条件: 处理空数组、单个区间等特殊情况
3. 性能优化: 使用贪心策略避免复杂的动态规划
4. 可读性: 清晰的变量命名和详细注释, 便于维护

极端场景与边界情况处理:

1. 空输入: intervals 为空数组
2. 极端值: 只有一个区间、所有区间相同
3. 重复数据: 多个区间相同
4. 有序/逆序数据: 区间按开始位置或结束位置排序

跨语言实现差异与优化:

1. Java: 使用 `Arrays.sort` 进行排序, 性能稳定
2. C++: 使用 `std::sort` 进行排序, 底层实现可能更优化
3. Python: 使用 `sorted` 函数或 `list.sort()` 方法, 基于 Timsort 算法

调试与测试策略:

1. 打印中间过程: 在循环中打印当前区间和上一个选择的区间
2. 用断言验证中间结果: 确保选择的区间不重叠
3. 性能退化排查: 检查排序和遍历的时间复杂度
4. 边界测试: 测试空数组、单元素等边界情况

实际应用场景与拓展:

1. 时间调度问题: 在会议安排中优化资源分配
2. 计算机视觉: 用于非极大值抑制(NMS)算法
3. 自然语言处理: 用于实体识别中的重叠实体处理

算法深入解析:

贪心算法在区间调度问题中的应用体现了其核心思想:

1. 局部最优选择: 每次选择结束位置最早的区间
2. 无后效性: 当前的选择不会影响之前的状态
3. 最优子结构: 问题的最优解包含子问题的最优解

这个问题的关键洞察是, 选择结束位置最早的区间能为后续区间留出最多空间。

```
"""
def eraseOverlapIntervals(intervals):
    """
    无重叠区间主函数 - 使用贪心算法计算需要移除的区间最小数量

    算法思路:
    1. 按区间结束位置排序, 优先选择结束位置早的区间
    2. 排序后遍历区间, 统计不重叠的区间数量
    3. 总区间数减去不重叠区间数就是需要移除的区间数

    Args:
        intervals (List[List[int]]): 区间列表, 每个区间为[start, end]格式
        intervals[i] = [start_i, end_i] 表示第 i 个区间

    Returns:
        int: 需要移除的区间最小数量

    时间复杂度: O(n * logn), 其中 n 是区间数量
    空间复杂度: O(1), 仅使用常数额外空间

    Examples:
        >>> eraseOverlapIntervals([[1, 2], [2, 3], [3, 4], [1, 3]])
        1
        >>> eraseOverlapIntervals([[1, 2], [1, 2], [1, 2]])
        2
    """

    # 异常处理: 检查输入是否为空
    if not intervals:
        return 0

    # 边界条件: 只有一个区间, 不需要移除
    if len(intervals) == 1:
        return 0

    # 按区间结束位置排序
    # 关键点: 按结束位置排序保证了贪心策略的正确性
    intervals.sort(key=lambda x: x[1])

    count = 1          # 不重叠区间数量, 初始选择第一个区间
    end = intervals[0][1] # 上一个选择区间的结束位置
```

```
# 遍历排序后的区间
# 时间复杂度: O(n)
for i in range(1, len(intervals)):
    # 如果当前区间与上一个选择的区间不重叠
    if intervals[i][0] >= end:
        count += 1           # 不重叠区间数加 1
        end = intervals[i][1] # 更新结束位置

# 需要移除的区间数 = 总区间数 - 不重叠区间数
return len(intervals) - count
```

```
# 补充题目 1: LeetCode 56. 合并区间
# 题目描述: 以数组 intervals 表示若干个区间的集合, 其中单个区间为 intervals[i] = [starti, endi]。
# 请你合并所有重叠的区间, 并返回一个不重叠的区间数组, 该数组需恰好覆盖输入中的所有区间。
# 链接: https://leetcode.cn/problems/merge-intervals/
```

```
def merge(intervals):
    """
    合并重叠的区间 - 使用贪心算法
```

算法思路:

1. 按区间起始位置排序
2. 依次合并重叠区间

Args:

intervals (List[List[int]]): 二维列表, 表示区间集合, 每个区间为 [start, end]

Returns:

List[List[int]]: 合并后的区间列表

时间复杂度:  $O(n \log n)$ , 主要是排序的时间复杂度

空间复杂度:  $O(n)$ , 用于存储合并后的区间

工程化考量:

1. 异常处理: 检查输入是否为空
2. 边界条件: 处理空数组、单区间等情况
3. 性能优化: 排序后使用贪心策略避免重复计算

"""

# 异常处理: 检查输入是否为空或长度不足

```
if not intervals or len(intervals) <= 1:
    return intervals # 空数组或只有一个区间, 无需合并
```

```

# 贪心策略：按区间起始位置排序，然后依次合并重叠区间
# 时间复杂度：O(n log n)
intervals.sort(key=lambda x: x[0])

merged = [intervals[0]] # 添加第一个区间

# 遍历其余区间
# 时间复杂度：O(n)
for i in range(1, len(intervals)):
    last = merged[-1] # 获取上一个合并后的区间
    current = intervals[i] # 当前区间

    # 如果当前区间与上一个合并后的区间重叠，则合并它们
    if current[0] <= last[1]:
        # 更新合并后的区间结束位置为两个区间结束位置的较大值
        last[1] = max(last[1], current[1])
    else:
        # 否则直接添加当前区间
        merged.append(current)

return merged

```

```

# 补充题目 2：LeetCode 57. 插入区间
# 题目描述：给你一个 无重叠的，按照区间起始端点排序的区间列表。
# 在列表中插入一个新的区间，你需要确保列表中的区间仍然有序且不重叠（如果有必要的话，可以合并区间）。
# 链接：https://leetcode.cn/problems/insert-interval/

```

```

def insert(intervals, newInterval):
    """

```

在排序且不重叠的区间列表中插入一个新区间，必要时合并 - 使用贪心算法

算法思路：

1. 添加所有在新区间之前且不重叠的区间
2. 合并所有与新区间重叠的区间
3. 添加剩余的区间

Args:

intervals (List[List[int]]): 二维列表，表示已排序且不重叠的区间集合  
newInterval (List[int]): 列表，表示要插入的新区间

Returns:

List[List[int]]: 插入并合并后的区间列表

时间复杂度:  $O(n)$ , 需要遍历整个区间列表一次

空间复杂度:  $O(n)$ , 用于存储结果

工程化考量:

1. 异常处理: 检查输入是否为空
2. 边界条件: 处理空数组等情况
3. 性能优化: 三段式处理避免重复遍历

"""

```
result = []
i = 0
n = len(intervals)
```

# 添加所有在新区间之前且不重叠的区间

# 时间复杂度:  $O(n)$

```
while i < n and intervals[i][1] < newInterval[0]:
    result.append(intervals[i])
    i += 1
```

# 合并所有与新区间重叠的区间

# 时间复杂度:  $O(n)$

```
while i < n and intervals[i][0] <= newInterval[1]:
    newInterval[0] = min(newInterval[0], intervals[i][0])
    newInterval[1] = max(newInterval[1], intervals[i][1])
    i += 1
```

# 添加合并后的区间

```
result.append(newInterval)
```

# 添加剩余的区间

# 时间复杂度:  $O(n)$

```
while i < n:
    result.append(intervals[i])
    i += 1
```

```
return result
```

# 补充题目 3: LeetCode 452. 用最少数量的箭引爆气球

# 题目描述: 有一些球形气球贴在一堵用 XY 平面表示的墙面上。气球可以用区间表示为 [start, end]，  
# 飞镖必须从整个区间的内部穿过才能引爆气球。求解把所有气球射爆所需的最小飞镖数。

```
# 链接: https://leetcode.cn/problems/minimum-number-of-arrows-to-burst-balloons/
```

```
def findMinArrowShots(points):
```

```
    """
```

```
    计算引爆所有气球所需的小飞镖数 - 使用贪心算法
```

算法思路:

1. 按区间结束位置排序
2. 贪心策略: 尽可能引爆更多气球

Args:

```
    points (List[List[int]]): 二维列表, 表示气球的区间, 每个区间为[start, end]
```

Returns:

```
    int: 所需的最小飞镖数
```

时间复杂度:  $O(n \log n)$ , 主要是排序的时间复杂度

空间复杂度:  $O(\log n)$ , 排序所需的额外空间

工程化考量:

1. 异常处理: 检查输入是否为空
2. 边界条件: 处理空数组等情况
3. 性能优化: 排序后使用贪心策略

```
    """
```

```
# 异常处理: 检查输入是否为空
```

```
if not points:
```

```
    return 0 # 没有气球, 需要 0 支箭
```

```
# 贪心策略: 按区间结束位置排序, 尽可能引爆更多气球
```

```
# 注意: Python 的 sort 对整数不会有溢出问题, 所以不需要特殊处理
```

```
# 时间复杂度:  $O(n \log n)$ 
```

```
points.sort(key=lambda x: x[1])
```

```
count = 1 # 需要的箭数, 初始为 1
```

```
end = points[0][1] # 第一支箭的位置
```

```
# 遍历排序后的区间
```

```
# 时间复杂度:  $O(n)$ 
```

```
for i in range(1, len(points)):
```

```
    # 如果当前气球的开始位置大于上一支箭的位置, 需要一支新箭
```

```
    if points[i][0] > end:
```

```
        count += 1
```

```
        end = points[i][1]
```

```
# 否则，当前气球会被上一支箭引爆，不需要额外的箭  
  
return count  
  
# 补充题目 4：LeetCode 986. 区间列表的交集  
# 题目描述：给定两个由一些 闭区间 组成的列表，firstList 和 secondList，  
# 其中 firstList[i] = [starti, endi] 而 secondList[j] = [startj, endj]。  
# 每个列表中的区间是不相交的，并且已经排序。  
# 返回这两个区间列表的交集。  
# 链接：https://leetcode.cn/problems/interval-list-intersections/
```

```
def intervalIntersection(firstList, secondList):
```

```
    """
```

```
    计算两个有序区间列表的交集 - 使用双指针算法
```

算法思路：

1. 使用双指针遍历两个列表
2. 计算当前两个区间的交集

Args:

firstList (List[List[int]]): 第一个有序区间列表

secondList (List[List[int]]): 第二个有序区间列表

Returns:

List[List[int]]: 交集区间列表

时间复杂度：O(m + n)，其中 m 和 n 分别是两个区间列表的长度

空间复杂度：O(min(m, n))，存储交集的最坏情况

工程化考量：

1. 异常处理：检查输入是否为空
2. 边界条件：处理空数组等情况
3. 性能优化：双指针避免重复计算

```
    """
```

```
# 异常处理：检查输入是否为空
```

```
if not firstList or not secondList:
```

```
    return [] # 任一列表为空，交集为空
```

```
result = []
```

```
i, j = 0, 0 # 两个指针分别指向两个列表
```

```
m, n = len(firstList), len(secondList)
```

```

# 双指针遍历两个列表
# 时间复杂度: O(m + n)
while i < m and j < n:
    start = max(firstList[i][0], secondList[j][0]) # 交集的起始位置
    end = min(firstList[i][1], secondList[j][1]) # 交集的结束位置

    # 如果有交集
    if start <= end:
        result.append([start, end])

    # 移动结束位置较小的区间的指针
    if firstList[i][1] < secondList[j][1]:
        i += 1
    else:
        j += 1

return result

```

```

# 补充题目 5: LeetCode 1288. 删除被覆盖区间
# 题目描述: 给你一个区间列表, 请你删除列表中被其他区间完全覆盖的区间。
# 只有当 c <= a 且 b <= d 时, 我们才认为区间 [a, b] 被区间 [c, d] 覆盖。
# 在完成所有删除操作后, 请你返回列表中剩余区间的数目。
# 链接: https://leetcode.cn/problems/remove-covered-intervals/

```

```

def removeCoveredIntervals(intervals):
    """
    计算删除被覆盖区间后剩余的区间数 - 使用贪心算法
    
```

算法思路:

1. 按起始位置升序排序, 起始位置相同时按结束位置降序排序
2. 贪心策略: 遍历区间, 统计不被覆盖的区间

Args:

intervals (List[List[int]]): 二维列表, 表示区间集合

Returns:

int: 剩余区间的数目

时间复杂度:  $O(n \log n)$ , 主要是排序的时间复杂度

空间复杂度:  $O(\log n)$ , 排序所需的额外空间

工程化考量:

1. 异常处理：检查输入是否为空
2. 边界条件：处理空数组、单区间等情况
3. 性能优化：排序后使用贪心策略

"""

```
# 异常处理：检查输入是否为空或长度不足
if not intervals or len(intervals) <= 1:
    return len(intervals) # 空数组或只有一个区间

# 贪心策略：按起始位置升序排序，起始位置相同时按结束位置降序排序
# 时间复杂度：O(n log n)
intervals.sort(key=lambda x: (x[0], -x[1]))

count = 1 # 剩余区间数，至少有一个区间
end = intervals[0][1] # 当前最大的结束位置

# 遍历排序后的区间
# 时间复杂度：O(n)
for i in range(1, len(intervals)):
    # 如果当前区间的结束位置大于最大结束位置，说明不被覆盖
    if intervals[i][1] > end:
        count += 1
        end = intervals[i][1]

return count
```

```
# 主函数测试
if __name__ == "__main__":
    # 测试 eraseOverlapIntervals
    intervals1 = [[1, 2], [2, 3], [3, 4], [1, 3]]
    print("测试 eraseOverlapIntervals:")
    print(f"输入: {intervals1}")
    print(f"输出: {eraseOverlapIntervals(intervals1)}")
```

```
# 测试 merge
intervals2 = [[1, 3], [2, 6], [8, 10], [15, 18]]
print("\n测试 merge:")
print(f"输入: {intervals2}")
print(f"输出: {merge(intervals2)})")
```

```
# 测试 insert
intervals3 = [[1, 3], [6, 9]]
newInterval = [2, 5]
```

```

print("\n测试 insert:")
print(f"输入: {intervals3}, {newInterval}")
print(f"输出: {insert(intervals3, newInterval)}")

# 测试 findMinArrowShots
points = [[10, 16], [2, 8], [1, 6], [7, 12]]
print("\n测试 findMinArrowShots:")
print(f"输入: {points}")
print(f"输出: {findMinArrowShots(points)}")

# 测试 intervalIntersection
firstList = [[0, 2], [5, 10], [13, 23], [24, 25]]
secondList = [[1, 5], [8, 12], [15, 24], [25, 26]]
print("\n测试 intervalIntersection:")
print(f"输入: {firstList}, {secondList}")
print(f"输出: {intervalIntersection(firstList, secondList)}")

# 测试 removeCoveredIntervals
intervals4 = [[1, 4], [3, 6], [2, 8]]
print("\n测试 removeCoveredIntervals:")
print(f"输入: {intervals4}")
print(f"输出: {removeCoveredIntervals(intervals4)}")

```

=====

文件: Code10\_LemonadeChange.cpp

=====

```

// 柠檬水找零
// 在柠檬水摊上，每一杯柠檬水的售价为 5 美元。顾客排队购买你的产品，(按账单 bills 支付的顺序) 一次购买一杯。
// 每位顾客只买一杯柠檬水，然后向你付 5 美元、10 美元或 20 美元。
// 你必须给每个顾客正确找零，也就是说净交易是每位顾客向你支付 5 美元。
// 注意，一开始你手头没有任何零钱。
// 给你一个整数数组 bills，其中 bills[i] 是第 i 位顾客付的账。
// 如果你能给每位顾客正确找零，返回 true，否则返回 false。
// 测试链接：https://leetcode.cn/problems/lemonade-change/
// 贪心算法专题 - 资源分配与最优选择问题集合

/*
 * 算法思路:
 * 1. 贪心策略: 找零时优先使用大面额纸币
 * 2. 维护 5 美元和 10 美元纸币的数量
 * 3. 收到 5 美元: 5 美元数量加 1

```

- \* 4. 收到 10 美元: 5 美元数量减 1, 10 美元数量加 1
- \* 5. 收到 20 美元: 优先用一张 10 美元和一张 5 美元找零, 如果没有 10 美元则用三张 5 美元找零
- \*
- \* 时间复杂度:  $O(n)$  -  $n$  是数组长度
- \* 空间复杂度:  $O(1)$  - 只使用了常数额外空间
- \* 是否最优解: 是, 这是处理此类问题的最优解法
- \*
- \* 工程化考量:
- \* 1. 异常处理: 检查输入是否为空
- \* 2. 边界条件: 处理空数组、单个元素等特殊情况
- \* 3. 性能优化: 一次遍历完成计算
- \* 4. 可读性: 清晰的变量命名和注释
- \*
- \* 极端场景与边界场景:
- \* 1. 空输入: bills 为空数组
- \* 2. 极端值: 只有一种面额的纸币
- \* 3. 重复数据: 多个相同面额的纸币
- \* 4. 有序/逆序数据: 纸币面额按顺序排列
- \*
- \* 跨语言场景与语言特性差异:
- \* 1. Java: 使用增强 for 循环遍历数组
- \* 2. C++: 使用传统 for 循环
- \* 3. Python: 使用 for 循环
- \*
- \* 调试能力构建:
- \* 1. 打印中间过程: 在循环中打印当前纸币面额和零钱数量
- \* 2. 用断言验证中间结果: 确保零钱数量不为负
- \* 3. 性能退化排查: 检查是否只遍历了一次数组
- \*
- \* 与机器学习、图像处理、自然语言处理的联系与应用:
- \* 1. 在资源管理问题中, 贪心算法可用于简单的资源分配策略
- \* 2. 在模拟系统中, 可以用于模拟交易过程
- \* 3. 在游戏开发中, 可以用于简单的经济系统设计

```
// 柠檬水找零主函数
int lemonadeChange(int bills[], int billsSize) {
    // 异常处理: 检查输入是否为空
    if (bills == 0 || billsSize == 0) {
        return 1; // 1 表示 true, 空数组表示没有顾客, 返回 true
    }

    int five = 0; // 5 美元纸币数量
```

```

int ten = 0;      // 10 美元纸币数量

// 遍历顾客支付的纸币
for (int i = 0; i < billsSize; i++) {
    int bill = bills[i];
    if (bill == 5) {
        // 收到 5 美元，不需要找零
        five++;
    } else if (bill == 10) {
        // 收到 10 美元，需要找零 5 美元
        if (five > 0) {
            five--;
            ten++;
        } else {
            // 没有 5 美元找零，返回 false
            return 0; // 0 表示 false
        }
    } else if (bill == 20) {
        // 收到 20 美元，需要找零 15 美元
        // 贪心策略：优先使用一张 10 美元和一张 5 美元找零
        if (ten > 0 && five > 0) {
            ten--;
            five--;
        } else if (five >= 3) {
            // 如果没有 10 美元，则用三张 5 美元找零
            five -= 3;
        } else {
            // 无法找零，返回 false
            return 0; // 0 表示 false
        }
    }
}

// 所有顾客都能正确找零
return 1; // 1 表示 true
}

// 补充题目 1：LeetCode 455. 分发饼干
// 题目描述：假设你是一位很棒的家长，想要给你的孩子们一些小饼干。但是，每个孩子最多只能给一块饼干。
// 对每个孩子 i，都有一个胃口值 g[i]，这是能让孩子们满足胃口的饼干的最小尺寸；
// 并且每块饼干 j，都有一个尺寸 s[j]。如果 s[j] >= g[i]，我们可以将这个饼干 j 分配给孩子 i，
// 这个孩子会得到满足。你的目标是尽可能满足越多数量的孩子，并输出这个最大数值。

```

```

// 链接: https://leetcode.cn/problems/assign-cookies/
#include <vector>
#include <algorithm>

int findContentChildren(int* g, int gSize, int* s, int sSize) {
    // 异常处理: 检查输入是否为空
    if (g == nullptr || s == nullptr || gSize <= 0 || sSize <= 0) {
        return 0; // 空数组, 无法满足任何孩子
    }

    // 将数组转换为 vector 以便使用 STL 的 sort 函数
    std::vector<int> greed(g, g + gSize);
    std::vector<int> cookie(s, s + sSize);

    // 贪心策略: 对胃口值和饼干尺寸进行排序, 尽可能用最小的能满足孩子胃口的饼干
    std::sort(greed.begin(), greed.end()); // 对孩子的胃口排序
    std::sort(cookie.begin(), cookie.end()); // 对饼干尺寸排序

    int childIndex = 0; // 当前考虑的孩子索引
    int cookieIndex = 0; // 当前考虑的饼干索引

    // 遍历饼干和孩子
    while (childIndex < gSize && cookieIndex < sSize) {
        // 如果当前饼干能满足当前孩子的胃口
        if (cookie[cookieIndex] >= greed[childIndex]) {
            childIndex++; // 孩子得到满足, 移动到下一个孩子
        }
        // 无论如何都移动到下一个饼干
        cookieIndex++;
    }

    // 返回满足的孩子数量
    return childIndex;
}

// 补充题目 2: LeetCode 135. 分发糖果
// 题目描述: n 个孩子站成一排。给你一个整数数组 ratings 表示每个孩子的评分。
// 你需要按照以下要求, 给这些孩子分发糖果:
// 1. 每个孩子至少分配到 1 个糖果
// 2. 相邻两个孩子评分更高的孩子会获得更多的糖果
// 请你给每个孩子分发糖果, 计算并返回需要准备的最小糖果数目。
// 链接: https://leetcode.cn/problems/candy/
int candy(int* ratings, int ratingsSize) {

```

```

// 异常处理：检查输入是否为空
if (ratings == nullptr || ratingsSize <= 0) {
    return 0; // 空数组，不需要糖果
}

// 将数组转换为 vector
std::vector<int> rate(ratings, ratings + ratingsSize);
std::vector<int> candies(ratingsSize, 1); // 初始化：每个孩子至少有 1 个糖果

// 贪心策略：从左到右遍历，确保右边评分高的孩子得到更多糖果
for (int i = 1; i < ratingsSize; i++) {
    if (rate[i] > rate[i - 1]) {
        candies[i] = candies[i - 1] + 1;
    }
}

// 贪心策略：从右到左遍历，确保左边评分高的孩子得到更多糖果
for (int i = ratingsSize - 2; i >= 0; i--) {
    if (rate[i] > rate[i + 1]) {
        candies[i] = std::max(candies[i], candies[i + 1] + 1);
    }
}

// 计算总糖果数
int total = 0;
for (int candy : candies) {
    total += candy;
}

return total;
}

// 补充题目 3: LeetCode 605. 种花问题
// 题目描述：假设有一个很长的花坛，一部分地块种植了花，另一部分却没有。
// 可是，花不能种植在相邻的地块上，它们会争夺水源，两者都会死去。
// 给你一个整数数组 flowerbed 表示花坛，由若干 0 和 1 组成，其中 0 表示没种植花，1 表示种植了花。
// 另有一个数 n，能否在不打破种植规则的情况下种入 n 朵花？能则返回 true，不能则返回 false。
// 链接: https://leetcode.cn/problems/can-place-flowers/
bool canPlaceFlowers(int* flowerbed, int flowerbedSize, int n) {
    // 异常处理：检查输入是否为空
    if (flowerbed == nullptr) {
        return n <= 0; // 空数组，只能种 0 朵花
    }
}

```

```

if (n <= 0) {
    return true; // 需要种 0 朵花，直接返回 true
}

int count = 0; // 可以种植的花的数量

// 贪心策略：遍历每个位置，如果可以种花就种
for (int i = 0; i < flowerbedSize; i++) {
    // 检查当前位置、前一个位置和后一个位置是否都没有花
    bool leftEmpty = (i == 0) || (flowerbed[i - 1] == 0);
    bool rightEmpty = (i == flowerbedSize - 1) || (flowerbed[i + 1] == 0);

    if (flowerbed[i] == 0 && leftEmpty && rightEmpty) {
        count++; // 可以种花
        flowerbed[i] = 1; // 标记为已种花

        // 提前结束，如果已经能满足需求
        if (count >= n) {
            return true;
        }
    }
}

return count >= n;
}

```

```

// 补充题目 4: LeetCode 406. 根据身高重建队列
// 题目描述：假设有打乱顺序的一群人站成一个队列，数组 people 表示队列中一些人的属性（不一定按顺序）。
// 每个 people[i] = [hi, ki] 表示第 i 个人的身高为 hi，前面正好有 ki 个身高大于或等于 hi 的人。
// 请你重新构造并返回输入数组 people 所表示的队列。返回的队列应该格式化为数组 queue ，
// 其中 queue[j] = [hj, kj] 是队列中第 j 个人的属性（queue[0] 是排在队列前面的人）。
// 链接: https://leetcode.cn/problems/queue-reconstruction-by-height/
#include <list>

```

```

// C++实现需要返回一个二维数组，需要动态分配内存
int** reconstructQueue(int** people, int peopleSize, int* peopleColSize, int* returnSize, int** returnColumnSizes) {
    // 异常处理：检查输入是否为空
    if (people == nullptr || peopleSize <= 1) {
        *returnSize = peopleSize;
        *returnColumnSizes = peopleColSize;
    }
}
```

```

*returnColumnSizes = (int*)malloc(sizeof(int) * peopleSize);
for (int i = 0; i < peopleSize; i++) {
    (*returnColumnSizes)[i] = 2;
}
return people; // 空数组或只有一个人，直接返回
}

// 将输入数据转换为 vector 以便排序
std::vector<std::vector<int>> peopleVec(peopleSize, std::vector<int>(2));
for (int i = 0; i < peopleSize; i++) {
    peopleVec[i][0] = people[i][0];
    peopleVec[i][1] = people[i][1];
}

// 贪心策略：按身高降序排序，身高相同时按 ki 升序排序
std::sort(peopleVec.begin(), peopleVec.end(), [] (const std::vector<int>& a, const
std::vector<int>& b) {
    if (a[0] != b[0]) {
        return a[0] > b[0]; // 身高降序
    } else {
        return a[1] < b[1]; // 身高相同时，ki 升序
    }
});

// 使用链表插入，提高插入效率
std::list<std::vector<int>> resultList;

// 遍历排序后的数组，根据 ki 插入到相应位置
for (const auto& person : peopleVec) {
    auto it = resultList.begin();
    // 移动迭代器到第 ki 个位置
    for (int i = 0; i < person[1]; i++) {
        it++;
    }
    resultList.insert(it, person); // 插入到索引为 ki 的位置
}

// 准备返回值
*returnSize = peopleSize;
*returnColumnSizes = (int*)malloc(sizeof(int) * peopleSize);
int** result = (int**)malloc(sizeof(int*) * peopleSize);

// 将链表转换为二维数组

```

```

int i = 0;
for (const auto& person : resultList) {
    result[i] = (int*)malloc(sizeof(int) * 2);
    result[i][0] = person[0];
    result[i][1] = person[1];
    (*returnColumnSizes)[i] = 2;
    i++;
}

return result;
}

// 补充题目 5: LeetCode 1005. K 次取反后最大化的数组和
// 题目描述: 给你一个整数数组 nums 和一个整数 k , 按以下方法修改数组:
// 1. 选择某个下标 i 并将 nums[i] 替换为 -nums[i] 。
// 你可以重复这个过程恰好 k 次。你也可以选择同一个下标 i 多次。
// 以这种方式修改数组后, 返回数组 可能的最大和 。
// 链接: https://leetcode.cn/problems/maximize-sum-of-array-after-k-negations/
int largestSumAfterKNegations(int* nums, int numsSize, int k) {
    // 异常处理: 检查输入是否为空
    if (nums == nullptr || numsSize <= 0) {
        return 0; // 空数组, 和为 0
    }

    // 将数组转换为 vector 以便排序
    std::vector<int> numsVec(nums, nums + numsSize);

    // 贪心策略: 每次将最小的负数变为正数, 这样可以最大化数组和
    // 排序数组, 从小到大
    std::sort(numsVec.begin(), numsVec.end());
    int i = 0;

    // 尽可能将负数变为正数
    while (i < numsSize && k > 0 && numsVec[i] < 0) {
        numsVec[i] = -numsVec[i]; // 取反
        k--; // 减少取反次数
        i++; // 移动到下一个元素
    }

    // 如果 k 还有剩余, 且 k 是奇数, 需要将最小的正数取反
    if (k > 0 && k % 2 == 1) {
        // 重新排序, 找到最小的数
        std::sort(numsVec.begin(), numsVec.end());
    }
}

```

```

    numsVec[0] = -numsVec[0]; // 将最小的数取反
}

// 计算数组和
int sum = 0;
for (int num : numsVec) {
    sum += num;
}

return sum;
}
=====
```

文件: Code10\_LemonadeChange.java

```

package class094;

import java.util.Arrays;

// 柠檬水找零 (Lemonade Change)
// 在柠檬水摊上，每一杯柠檬水的售价为 5 美元。顾客排队购买你的产品，(按账单 bills 支付的顺序) 一次购买一杯。
// 每位顾客只买一杯柠檬水，然后向你付 5 美元、10 美元或 20 美元。
// 你必须给每个顾客正确找零，也就是说净交易是每位顾客向你支付 5 美元。
// 注意，一开始你手头没有任何零钱。
// 给你一个整数数组 bills，其中 bills[i] 是第 i 位顾客付的账。
// 如果你能给每位顾客正确找零，返回 true，否则返回 false。
//
// 算法标签：贪心算法(Greedy Algorithm)、资源管理(Resource Management)、状态维护(State Maintenance)
// 时间复杂度: O(n)，其中 n 是数组长度
// 空间复杂度: O(1)，仅使用常数额外空间
// 测试链接 : https://leetcode.cn/problems/lemonade-change/
// 相关题目: LeetCode 455. 分发饼干、LeetCode 135. 分发糖果
// 贪心算法专题 - 资源分配与最优选择问题集合
public class Code10_LemonadeChange {

/*
 * 算法思路详解:
 * 1. 贪心策略核心: 找零时优先使用大面额纸币,
 *     这样可以保留更多小面额纸币用于后续找零
 * 2. 状态维护: 通过维护 5 美元和 10 美元纸币的数量,
```

- \* 实时跟踪当前可用的零钱状态
- \* 3. 找零规则:
  - 收到 5 美元: 无需找零, 直接增加 5 美元纸币数量
  - 收到 10 美元: 需找零 5 美元, 消耗一张 5 美元纸币
  - 收到 20 美元: 需找零 15 美元, 优先使用一张 10 美元+一张 5 美元
  - 若无 10 美元则使用三张 5 美元
- \*
- \* 时间复杂度分析:
  - $O(n)$ , 其中  $n$  是数组长度, 只需要一次遍历即可完成判断
- \* 空间复杂度分析:
  - $O(1)$ , 仅使用了常数级别的额外空间存储 five 和 ten 变量
- \* 是否最优解: 是, 这是处理此类找零问题的最优解法
- \*
- \* 工程化最佳实践:
  - \* 1. 输入验证: 严格检查输入参数的有效性, 防止空指针异常
  - \* 2. 边界处理: 妥善处理各种边界情况, 如空数组等
  - \* 3. 性能优化: 采用单次遍历策略, 避免重复计算
  - \* 4. 代码可读性: 使用语义明确的变量名和详尽的注释
  - \* 5. 状态一致性: 确保每次交易后零钱状态的正确性
- \*
- \* 极端场景与边界情况处理:
  - \* 1. 空输入场景: bills 为空数组或 null 时直接返回 true
  - \* 2. 单一面额场景: 只收到 5 美元或只收到大面额纸币
  - \* 3. 连续大额场景: 连续收到 10 美元或 20 美元的处理
  - \* 4. 零钱不足场景: 无法找零时的处理
  - \* 5. 特殊序列场景: 如 [5, 5, 10, 20, 5, 5, 5, 5, 5, 5, 5, 10, 5, 5, 20, 5, 20, 5]
- \*
- \* 跨语言实现差异与优化:
  - \* 1. Java 实现: 使用增强 for 循环遍历数组, 注意整数比较
  - \* 2. C++ 实现: 使用传统 for 循环, 注意数组边界检查
  - \* 3. Python 实现: 使用 for 循环, 利用列表特性
  - \* 4. 内存管理: 不同语言的垃圾回收机制对性能的影响
- \*
- \* 调试与测试策略:
  - \* 1. 过程可视化: 在关键节点打印当前纸币面额和零钱数量
  - \* 2. 断言验证: 在每次交易后添加断言确保零钱数量非负
  - \* 3. 性能监控: 跟踪遍历过程的实际执行时间
  - \* 4. 边界测试: 设计覆盖所有边界条件的测试用例
  - \* 5. 压力测试: 使用大规模数据验证算法稳定性
- \*
- \* 实际应用场景与拓展:
  - \* 1. 零售系统: 收银台找零策略优化
  - \* 2. 自动售货机: 硬币分配算法

- \* 3. 银行系统：ATM 取款面额分配
  - \* 4. 游戏经济：虚拟货币找零机制
  - \* 5. 交通收费：公交地铁找零系统
  - \*
  - \* 算法深入解析：
  - \* 1. 贪心策略原理：优先使用大面额纸币确保后续找零能力
  - \* 2. 状态转移：每次交易后正确更新零钱状态
  - \* 3. 正确性证明：通过归纳法可以证明贪心策略的正确性
  - \* 4. 优化扩展：可扩展支持更多面额纸币的找零
- \*/

```

public static boolean lemonadeChange(int[] bills) {
    // 异常处理：检查输入是否为空
    if (bills == null) {
        return true; // 空数组表示没有顾客，返回 true
    }

    int five = 0; // 5 美元纸币数量
    int ten = 0; // 10 美元纸币数量

    // 遍历顾客支付的纸币
    for (int bill : bills) {
        if (bill == 5) {
            // 收到 5 美元，不需要找零
            five++;
        } else if (bill == 10) {
            // 收到 10 美元，需要找零 5 美元
            if (five > 0) {
                five--;
                ten++;
            } else {
                // 没有 5 美元找零，返回 false
                return false;
            }
        } else if (bill == 20) {
            // 收到 20 美元，需要找零 15 美元
            // 贪心策略：优先使用一张 10 美元和一张 5 美元找零
            if (ten > 0 && five > 0) {
                ten--;
                five--;
            } else if (five >= 3) {
                // 如果没有 10 美元，则用三张 5 美元找零
                five -= 3;
            } else {

```

```

        // 无法找零，返回 false
        return false;
    }
}

// 所有顾客都能正确找零
return true;
}

// 测试函数
public static void main(String[] args) {
    // 测试用例 1
    int[] bills1 = {5, 5, 5, 10, 20};
    System.out.println("测试用例 1 结果: " + lemonadeChange(bills1)); // 期望输出: true

    // 测试用例 2
    int[] bills2 = {5, 5, 10, 10, 20};
    System.out.println("测试用例 2 结果: " + lemonadeChange(bills2)); // 期望输出: false

    // 测试用例 3
    int[] bills3 = {10, 10};
    System.out.println("测试用例 3 结果: " + lemonadeChange(bills3)); // 期望输出: false

    // 测试用例 4
    int[] bills4 = {5, 5, 10, 20, 5, 5, 5, 5, 5, 5, 5, 5, 10, 5, 5, 20, 5, 20, 5};
    System.out.println("测试用例 4 结果: " + lemonadeChange(bills4)); // 期望输出: true

    // 测试用例 5: 边界情况
    int[] bills5 = {};
    System.out.println("测试用例 5 结果: " + lemonadeChange(bills5)); // 期望输出: true
}

// 补充题目 1: LeetCode 455. 分发饼干 (Assign Cookies)
// 题目描述: 假设你是一位很棒的家长，想要给你的孩子们一些小饼干。但是，每个孩子最多只能给一块饼干。
// 对每个孩子 i，都有一个胃口值 g[i]，这是能让孩子们满足胃口的饼干的最小尺寸；
// 并且每块饼干 j，都有一个尺寸 s[j]。如果 s[j] >= g[i]，我们可以将这个饼干 j 分配给孩子 i，
// 这个孩子会得到满足。你的目标是尽可能满足越多数量的孩子，并输出这个最大数值。
//
// 算法标签: 贪心算法(Greedy Algorithm)、双指针(Double Pointers)、排序(Sorting)
// 时间复杂度: O(m*log(m)+n*log(n))，其中 m 是孩子数量，n 是饼干数量
// 空间复杂度: O(1)，仅使用常数额外空间

```

```
// 链接: https://leetcode.cn/problems/assign-cookies/
/*
 * 算法详解与策略分析:
 * 1. 贪心策略核心: 优先满足胃口小的孩子,
 *    这样可以保留大饼干给胃口大的孩子
 * 2. 排序优化: 对孩子胃口和饼干尺寸都进行排序,
 *    为贪心策略实施奠定基础
 * 3. 匹配机制: 使用双指针技术进行高效匹配
 *
 * 算法步骤详解:
 * 1. 预处理: 对两个数组进行升序排序
 * 2. 初始化: 设置孩子和饼干的双指针
 * 3. 匹配过程:
 *    - 若当前饼干能满足当前孩子则两个指针都前进
 *    - 否则仅饼干指针前进寻找更大饼干
 * 4. 结果返回: 孩子指针位置即为满足的孩子数
 *
 * 算法优化与正确性:
 * 贪心选择性质: 优先满足小胃口是最优的局部选择
 * 最优子结构: 满足一个孩子后剩余问题仍保持最优性
 */

```

```
public static int findContentChildren(int[] g, int[] s) {
    // 异常处理: 检查输入是否为空
    if (g == null || s == null || g.length == 0 || s.length == 0) {
        return 0; // 空数组, 无法满足任何孩子
    }

    // 贪心策略: 对胃口值和饼干尺寸进行排序, 尽可能用最小的能满足孩子胃口的饼干
    Arrays.sort(g); // 对孩子的胃口排序
    Arrays.sort(s); // 对饼干尺寸排序

    int childIndex = 0; // 当前考虑的孩子索引
    int cookieIndex = 0; // 当前考虑的饼干索引

    // 遍历饼干和孩子
    while (childIndex < g.length && cookieIndex < s.length) {
        // 如果当前饼干能满足当前孩子的胃口
        if (s[cookieIndex] >= g[childIndex]) {
            childIndex++; // 孩子得到满足, 移动到下一个孩子
        }
        // 无论如何都移动到下一个饼干
        cookieIndex++;
    }
}
```

```

    // 返回满足的孩子数量
    return childIndex;
}

// 补充题目 2: LeetCode 135. 分发糖果 (Candy)
// 题目描述: n 个孩子站成一排。给你一个整数数组 ratings 表示每个孩子的评分。
// 你需要按照以下要求，给这些孩子分发糖果:
// 1. 每个孩子至少分配到 1 个糖果
// 2. 相邻两个孩子评分更高的孩子会获得更多的糖果
// 请你给每个孩子分发糖果，计算并返回需要准备的最小糖果数目。
//
// 算法标签: 贪心算法(Greedy Algorithm)、两次遍历(Two Pass)、动态规划思想(DP Thinking)
// 时间复杂度: O(n)，其中 n 是孩子数量
// 空间复杂度: O(n)，需要额外数组存储糖果数
// 链接: https://leetcode.cn/problems/candy/
/*
 * 算法详解与策略分析:
 * 1. 两次遍历策略: 通过从左到右和从右到左两次遍历,
 * 分别处理左右相邻约束条件
 * 2. 左遍历目的: 确保评分高的右孩子比左孩子获得更多糖果
 * 3. 右遍历目的: 确保评分高的左孩子比右孩子获得更多糖果
 * 4. 结果合并: 对每个孩子取两次遍历结果的最大值
 *
 * 算法步骤详解:
 * 1. 初始化: 为每个孩子分配 1 个糖果作为基础
 * 2. 左遍历:
 * - 若右孩子评分高于左孩子则右孩子糖果数=左孩子+1
 * 3. 右遍历:
 * - 若左孩子评分高于右孩子则左孩子糖果数=max(当前值, 右孩子+1)
 * 4. 结果统计: 累加所有孩子的糖果数
 *
 * 算法优化与实现细节:
 * 空间优化: 可优化为 O(1) 空间的单次遍历算法
 * 正确性保证: 两次遍历确保满足所有约束条件
*/
public static int candy(int[] ratings) {
    // 异常处理: 检查输入是否为空
    if (ratings == null || ratings.length == 0) {
        return 0; // 空数组, 不需要糖果
    }

    int n = ratings.length;

```

```

int[] candies = new int[n];

// 初始化: 每个孩子至少有 1 个糖果
for (int i = 0; i < n; i++) {
    candies[i] = 1;
}

// 贪心策略: 从左到右遍历, 确保右边评分高的孩子得到更多糖果
for (int i = 1; i < n; i++) {
    if (ratings[i] > ratings[i - 1]) {
        candies[i] = candies[i - 1] + 1;
    }
}

// 贪心策略: 从右到左遍历, 确保左边评分高的孩子得到更多糖果
for (int i = n - 2; i >= 0; i--) {
    if (ratings[i] > ratings[i + 1]) {
        candies[i] = Math.max(candies[i], candies[i + 1] + 1);
    }
}

// 计算总糖果数
int total = 0;
for (int candy : candies) {
    total += candy;
}

return total;
}

// 补充题目 3: LeetCode 605. 种花问题 (Can Place Flowers)
// 题目描述: 假设有一个很长的花坛, 一部分地块种植了花, 另一部分却没有。
// 可是, 花不能种植在相邻的地块上, 它们会争夺水源, 两者都会死去。
// 给你一个整数数组 flowerbed 表示花坛, 由若干 0 和 1 组成, 其中 0 表示没种植花, 1 表示种植了花。
// 另有一个数 n , 能否在不打破种植规则的情况下种入 n 朵花? 能则返回 true , 不能则返回 false。
/*
// 算法标签: 贪心算法(Greedy Algorithm)、数组遍历(Array Traversal)、约束满足(Constraint Satisfaction)
// 时间复杂度: O(n), 其中 n 是花坛长度
// 空间复杂度: O(1), 原地修改数组
// 链接: https://leetcode.cn/problems/can-place-flowers/
*/

```

- \* 算法详解与策略分析：
  - \* 1. 贪心策略核心：遍历花坛，尽可能多地种花，  
在满足约束条件下优先种花
  - \* 2. 约束检查：当前位置可以种花当且仅当：  
 - 当前位置为 0（未种花）  
 - 前一个位置为 0 或不存在  
 - 后一个位置为 0 或不存在
  - \* 3. 优化终止：一旦满足 n 朵花就提前返回

\*

#### \* 算法步骤详解：

- \* 1. 初始化：设置可种花计数器
- \* 2. 遍历过程：  
  - 检查当前位置是否满足种花条件
  - 若满足则在当前位置种花并增加计数器
  - 检查是否已满足 n 朵花，满足则提前返回 true
- \* 3. 结果判断：返回累计种花数是否大于等于 n

\*

#### \* 算法优化与边界处理：

- \* 边界情况：处理花坛首尾位置的约束检查
- \* 空间优化：可原地修改数组标记已种花位置

\*/

```
public static boolean canPlaceFlowers(int[] flowerbed, int n) {
    // 异常处理：检查输入是否为空
    if (flowerbed == null) {
        return n <= 0; // 空数组，只能种 0 朵花
    }

    if (n <= 0) {
        return true; // 需要种 0 朵花，直接返回 true
    }

    int count = 0; // 可以种植的花的数量
    int len = flowerbed.length;

    // 贪心策略：遍历每个位置，如果可以种花就种
    for (int i = 0; i < len; i++) {
        // 检查当前位置、前一个位置和后一个位置是否都没有花
        boolean leftEmpty = (i == 0) || (flowerbed[i - 1] == 0);
        boolean rightEmpty = (i == len - 1) || (flowerbed[i + 1] == 0);

        if (flowerbed[i] == 0 && leftEmpty && rightEmpty) {
            count++; // 可以种花
            flowerbed[i] = 1; // 标记为已种花
        }
    }
}
```

```

        // 提前结束，如果已经能满足需求
        if (count >= n) {
            return true;
        }
    }

    return count >= n;
}

```

// 补充题目 4: LeetCode 406. 根据身高重建队列 (Queue Reconstruction by Height)

// 题目描述: 假设有打乱顺序的一群人站成一个队列, 数组 people 表示队列中一些人的属性 (不一定按顺序)。

// 每个 people[i] = [hi, ki] 表示第 i 个人的身高为 hi , 前面 正好 有 ki 个身高大于或等于 hi 的人。

// 请你重新构造并返回输入数组 people 所表示的队列。返回的队列应该格式化为数组 queue ,

// 其中 queue[j] = [hj, kj] 是队列中第 j 个人的属性 (queue[0] 是排在队列前面的人)。

//

// 算法标签: 贪心算法(Greedy Algorithm)、排序(Sorting)、链表插入(Linked List Insertion)

// 时间复杂度: O(n<sup>2</sup>) , 其中 n 是人数

// 空间复杂度: O(n) , 使用 List 存储结果

// 链接: <https://leetcode.cn/problems/queue-reconstruction-by-height/>

/\*

\* 算法详解与策略分析:

\* 1. 排序策略: 身高降序, k 值升序

\* 2. 贪心策略: 按排序后的顺序插入到指定位置

\* 3. 插入位置: k 值即为插入位置

\* 4. 数据结构: 使用 List 支持动态插入

\*

\* 算法步骤详解:

\* 1. 排序阶段:

\* - 按身高降序排列

\* - 身高相同时按 k 值升序排列

\* 2. 插入过程:

\* - 依次处理排序后的人群

\* - 将当前人插入到结果列表的 p[1]位置

\* 3. 结果生成: 将 List 转换为数组返回

\*

\* 算法优化与正确性:

\* 贪心选择性质: 先处理高个子可以确保后续插入不影响前面排列

\* 最优子结构: 每步插入都保持当前队列的正确性

\*/

```

public static int[][] reconstructQueue(int[][] people) {
    // 异常处理：检查输入是否为空
    if (people == null || people.length <= 1) {
        return people; // 空数组或只有一个人，直接返回
    }

    // 贪心策略：按身高降序排序，身高相同时按 ki 升序排序
    Arrays.sort(people, (a, b) -> {
        if (a[0] != b[0]) {
            return b[0] - a[0]; // 身高降序
        } else {
            return a[1] - b[1]; // 身高相同时，ki 升序
        }
    });

    // 使用链表插入，提高插入效率
    java.util.List<int[]> result = new java.util.LinkedList<>();

    // 遍历排序后的数组，根据 ki 插入到相应位置
    for (int[] person : people) {
        result.add(person[1], person); // 插入到索引为 ki 的位置
    }

    // 转换为二维数组返回
    return result.toArray(new int[result.size()][]);
}

// 补充题目 5: LeetCode 1005. K 次取反后最大化的数组和 (Maximize Sum Of Array After K Negations)
// 题目描述：给你一个整数数组 nums 和一个整数 k，按以下方法修改数组：
// 1. 选择某个下标 i 并将 nums[i] 替换为 -nums[i]。
// 你可以重复这个过程恰好 k 次。你也可以选择同一个下标 i 多次。
// 以这种方式修改数组后，返回数组 可能的最大和。
//
// 算法标签：贪心算法(Greedy Algorithm)、数组操作(Array Operations)、排序(Sorting)
// 时间复杂度：O(n*log(n))，其中 n 是数组长度
// 空间复杂度：O(1)，原地修改数组
// 链接：https://leetcode.cn/problems/maximize-sum-of-array-after-k-negations/
/*
 * 算法详解与策略分析：
 * 1. 贪心策略核心：每次将绝对值最大的负数变为正数，这样可以最大化数组和
 * 2. 分阶段处理：

```

```

*      - 优先将负数变为正数
*      - 若还有剩余操作次数且为奇数，将最小正数取反
* 3. 排序优化：通过排序快速找到需要操作的元素
*
* 算法步骤详解：
* 1. 预处理：对数组进行升序排序
* 2. 负数处理：
*      - 将尽可能多的负数变为正数
* 3. 剩余处理：
*      - 若剩余操作次数为奇数，将最小元素取反
* 4. 结果计算：计算数组元素和
*
* 算法优化与实现细节：
* 边界处理：处理 k 为 0 或数组全为正数的情况
* 正确性保证：贪心策略确保每次操作都能最大化数组和
*/
public static int largestSumAfterKNegations(int[] nums, int k) {
    // 异常处理：检查输入是否为空
    if (nums == null || nums.length == 0) {
        return 0; // 空数组，和为 0
    }

    // 贪心策略：每次将最小的负数变为正数，这样可以最大化数组和
    // 排序数组，从小到大
    Arrays.sort(nums);
    int i = 0;

    // 尽可能将负数变为正数
    while (i < nums.length && k > 0 && nums[i] < 0) {
        nums[i] = -nums[i]; // 取反
        k--; // 减少取反次数
        i++; // 移动到下一个元素
    }

    // 如果 k 还有剩余，且 k 是奇数，需要将最小的正数取反
    if (k > 0 && k % 2 == 1) {
        // 重新排序，找到最小的数
        Arrays.sort(nums);
        nums[0] = -nums[0]; // 将最小的数取反
    }

    // 计算数组和
    int sum = 0;

```

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

---

文件: Code10\_LemonadeChange.py

---

```
# 柠檬水找零 (Lemonade Change)  
# 在柠檬水摊上，每一杯柠檬水的售价为 5 美元。顾客排队购买你的产品，(按账单 bills 支付的顺序) 一次  
# 购买一杯。  
# 每位顾客只买一杯柠檬水，然后向你付 5 美元、10 美元或 20 美元。  
# 你必须给每个顾客正确找零，也就是说净交易是每位顾客向你支付 5 美元。  
# 注意，一开始你手头没有任何零钱。  
# 给你一个整数数组 bills ，其中 bills[i] 是第 i 位顾客付的账。  
# 如果你能给每位顾客正确找零，返回 true，否则返回 false。  
#  
# 算法标签: 贪心算法(Greedy Algorithm)、资源分配(Resource Allocation)  
# 时间复杂度: O(n)，其中 n 是数组长度  
# 空间复杂度: O(1)，仅使用常数额外空间  
# 测试链接 : https://leetcode.cn/problems/lemonade-change/  
# 相关题目: LeetCode 455. 分发饼干、LeetCode 135. 分发糖果  
# 贪心算法专题 - 资源分配与最优选择问题集合
```

"""

算法思路详解:

1. 贪心策略: 找零时优先使用大面额纸币
  - 这个策略的核心思想是在找零时优先使用面额较大的纸币
  - 对于 20 美元的找零，优先使用一张 10 美元和一张 5 美元，而不是三张 5 美元
  - 这样可以保留更多的 5 美元纸币用于后续找零
2. 维护 5 美元和 10 美元纸币的数量
  - 只需要维护这两种面额的纸币数量，因为 20 美元不能用于找零
  - 通过计数器来跟踪可用的零钱
3. 收到 5 美元: 5 美元数量加 1
  - 最简单的情况，不需要找零
4. 收到 10 美元: 5 美元数量减 1, 10 美元数量加 1

- 需要找零 5 美元，检查是否有足够的 5 美元纸币
5. 收到 20 美元：优先用一张 10 美元和一张 5 美元找零，如果没有 10 美元则用三张 5 美元找零
- 需要找零 15 美元，采用贪心策略优先使用大面额纸币

时间复杂度分析：

- 遍历时间复杂度： $O(n)$ ，其中  $n$  是数组长度
- 总体时间复杂度： $O(n)$

空间复杂度分析：

- 只使用了常数额外空间存储变量
- 空间复杂度： $O(1)$

是否最优解：

- 是，这是处理此类问题的最优解法
- 贪心策略保证了局部最优解能导致全局最优解

工程化最佳实践：

1. 异常处理：检查输入是否为空或格式不正确
2. 边界条件：处理空数组、单个元素等特殊情况
3. 性能优化：一次遍历完成计算，提前终止条件避免不必要的计算
4. 可读性：清晰的变量命名和详细注释，便于维护

极端场景与边界情况处理：

1. 空输入：bills 为空数组
2. 极端值：只有一种面额的纸币
3. 重复数据：多个相同面额的纸币
4. 有序/逆序数据：纸币面额按顺序排列

跨语言实现差异与优化：

1. Java：使用增强 for 循环遍历数组，代码更简洁
2. C++：使用传统 for 循环，性能更优
3. Python：使用 for 循环，语法更灵活

调试与测试策略：

1. 打印中间过程：在循环中打印当前纸币面额和零钱数量
2. 用断言验证中间结果：确保零钱数量不为负
3. 性能退化排查：检查是否只遍历了一次数组
4. 边界测试：测试空数组、单元素等边界情况

实际应用场景与拓展：

1. 资源管理问题：在简单的资源分配策略中应用贪心算法
2. 模拟系统：用于模拟交易过程

### 3. 游戏开发：用于简单的经济系统设计

算法深入解析：

贪心算法在找零问题中的应用体现了其核心思想：

1. 局部最优选择：每次找零时选择最优的纸币组合
2. 无后效性：当前的选择不会影响之前的状态
3. 最优子结构：问题的最优解包含子问题的最优解

这个问题的关键洞察是，优先使用大面额纸币能为后续找零保留更多小面额纸币。

”””

```
def lemonadeChange(bills):
```

```
    """
```

```
    柠檬水找零主函数 - 使用贪心算法判断是否能给每位顾客正确找零
```

算法思路：

1. 贪心策略：找零时优先使用大面额纸币
2. 维护 5 美元和 10 美元纸币的数量
3. 根据收到的纸币面额进行相应的处理

Args:

bills (List[int]): 顾客支付的纸币面额列表

bills[i] 表示第 i 位顾客支付的纸币面额 (5、10 或 20)

Returns:

bool: 是否能给每位顾客正确找零

时间复杂度：O(n)，其中 n 是数组长度

空间复杂度：O(1)，仅使用常数额外空间

Examples:

```
>>> lemonadeChange([5, 5, 5, 10, 20])
```

```
True
```

```
>>> lemonadeChange([5, 5, 10, 10, 20])
```

```
False
```

```
"""
```

```
# 异常处理：检查输入是否为空
```

```
if not bills:
```

```
    return True # 空数组表示没有顾客，返回 True
```

```
five = 0 # 5 美元纸币数量
```

```
ten = 0 # 10 美元纸币数量
```

```

# 遍历顾客支付的纸币
# 时间复杂度: O(n)
for bill in bills:
    if bill == 5:
        # 收到 5 美元, 不需要找零
        five += 1
    elif bill == 10:
        # 收到 10 美元, 需要找零 5 美元
        if five > 0:
            five -= 1
            ten += 1
        else:
            # 没有 5 美元找零, 返回 False
            return False
    elif bill == 20:
        # 收到 20 美元, 需要找零 15 美元
        # 贪心策略: 优先使用一张 10 美元和一张 5 美元找零
        if ten > 0 and five > 0:
            ten -= 1
            five -= 1
        elif five >= 3:
            # 如果没有 10 美元, 则用三张 5 美元找零
            five -= 3
        else:
            # 无法找零, 返回 False
            return False

# 所有顾客都能正确找零
return True

```

```

# 补充题目 1: LeetCode 455. 分发饼干
# 题目描述: 假设你是一位很棒的家长, 想要给你的孩子们一些小饼干。但是, 每个孩子最多只能给一块饼干。
# 对每个孩子 i, 都有一个胃口值 g[i], 这是能让孩子们满足胃口的饼干的最小尺寸;
# 并且每块饼干 j, 都有一个尺寸 s[j]。如果 s[j] >= g[i], 我们可以将这个饼干 j 分配给孩子 i,
# 这个孩子会得到满足。你的目标是尽可能满足越多数量的孩子, 并输出这个最大数值。
# 链接: https://leetcode.cn/problems/assign-cookies/

```

```

def findContentChildren(g, s):
    """
    分发饼干 - 使用贪心算法计算最多能满足的孩子数量
    """

```

算法思路：

1. 贪心策略：对胃口值和饼干尺寸进行排序
2. 尽可能用最小的能满足孩子胃口的饼干

Args:

g (List[int]): 孩子的胃口值列表， $g[i]$  表示第  $i$  个孩子的最小满足饼干尺寸  
s (List[int]): 饼干的尺寸列表， $s[j]$  表示第  $j$  块饼干的尺寸

Returns:

int: 最多能满足的孩子数量

时间复杂度:  $O(n \log n + m \log m)$ , 其中  $n$  是孩子数量,  $m$  是饼干数量, 排序需要的时间

空间复杂度:  $O(1)$ , 只使用了常数额外空间

是否最优解: 是, 贪心策略是解决这类问题的最优方法

工程化考量:

1. 异常处理: 检查输入是否为空
2. 边界条件: 处理空数组等特殊情况
3. 性能优化: 排序后使用双指针避免重复计算

"""

```
# 异常处理: 检查输入是否为空
if not g or not s:
    return 0 # 空数组, 无法满足任何孩子

# 贪心策略: 对胃口值和饼干尺寸进行排序, 尽可能用最小的能满足孩子胃口的饼干
# 时间复杂度:  $O(n \log n + m \log m)$ 
g.sort() # 对孩子的胃口排序
s.sort() # 对饼干尺寸排序

childIndex = 0 # 当前考虑的孩子索引
cookieIndex = 0 # 当前考虑的饼干索引

# 遍历饼干和孩子
# 时间复杂度:  $O(n + m)$ 
while childIndex < len(g) and cookieIndex < len(s):
    # 如果当前饼干能满足当前孩子的胃口
    if s[cookieIndex] >= g[childIndex]:
        childIndex += 1 # 孩子得到满足, 移动到下一个孩子
    # 无论如何都移动到下一个饼干
    cookieIndex += 1

# 返回满足的孩子数量
return childIndex
```

```
# 补充题目 2: LeetCode 135. 分发糖果
# 题目描述: n 个孩子站成一排。给你一个整数数组 ratings 表示每个孩子的评分。
# 你需要按照以下要求，给这些孩子分发糖果：
# 1. 每个孩子至少分配到 1 个糖果
# 2. 相邻两个孩子评分更高的孩子会获得更多的糖果
# 请你给每个孩子分发糖果，计算并返回需要准备的最小糖果数目。
# 链接: https://leetcode.cn/problems/candy/
```

```
def candy(ratings):
    """
    分发糖果 - 使用贪心算法计算需要准备的最小糖果数目
    
```

算法思路:

1. 贪心策略: 两次遍历
2. 第一次从左到右, 确保右边评分高的孩子得到更多糖果
3. 第二次从右到左, 确保左边评分高的孩子得到更多糖果

Args:

    ratings (List[int]): 孩子的评分列表, ratings[i] 表示第 i 个孩子的评分

Returns:

    int: 需要准备的最小糖果数目

时间复杂度:  $O(n)$ , 其中  $n$  是孩子数量, 需要两次遍历

空间复杂度:  $O(n)$ , 需要一个数组存储每个孩子的糖果数

是否最优解: 是, 这是解决此类问题的最优方法

工程化考量:

1. 异常处理: 检查输入是否为空
2. 边界条件: 处理空数组、单元素等情况
3. 性能优化: 两次遍历确保满足所有约束条件

"""

```
# 异常处理: 检查输入是否为空
```

```
if not ratings:
```

```
    return 0 # 空数组, 不需要糖果
```

```
n = len(ratings)
```

```
candies = [1] * n # 初始化: 每个孩子至少有 1 个糖果
```

```
# 贪心策略: 从左到右遍历, 确保右边评分高的孩子得到更多糖果
```

```
# 时间复杂度:  $O(n)$ 
```

```

for i in range(1, n):
    if ratings[i] > ratings[i - 1]:
        candies[i] = candies[i - 1] + 1

# 贪心策略：从右到左遍历，确保左边评分高的孩子得到更多糖果
# 时间复杂度：O(n)
for i in range(n - 2, -1, -1):
    if ratings[i] > ratings[i + 1]:
        candies[i] = max(candies[i], candies[i + 1] + 1)

# 计算总糖果数
# 时间复杂度：O(n)
return sum(candies)

```

# 补充题目 3: LeetCode 605. 种花问题

# 题目描述: 假设有一个很长的花坛, 一部分地块种植了花, 另一部分却没有。

# 可是, 花不能种植在相邻的地块上, 它们会争夺水源, 两者都会死去。

# 给你一个整数数组 flowerbed 表示花坛, 由若干 0 和 1 组成, 其中 0 表示没种植花, 1 表示种植了花。

# 另有一个数 n , 能否在不打破种植规则的情况下种入 n 朵花? 能则返回 true , 不能则返回 false。

# 链接: <https://leetcode.cn/problems/can-place-flowers/>

```
def canPlaceFlowers(flowerbed, n):
```

```
"""

```

种花问题 – 使用贪心算法判断是否能在不打破种植规则的情况下种入 n 朵花

算法思路:

1. 贪心策略: 遍历每个位置, 如果可以种花就种
2. 检查条件: 当前位置、前一个位置和后一个位置都没有花

Args:

flowerbed (List[int]): 表示花坛的数组, 0 表示没种植花, 1 表示种植了花

n (int): 需要种植的花的数量

Returns:

bool: 是否能在不打破种植规则的情况下种入 n 朵花

时间复杂度: O(n), 其中 n 是花坛长度

空间复杂度: O(1), 只使用了常数额外空间

是否最优解: 是, 一次遍历即可解决问题

工程化考量:

1. 异常处理: 检查输入是否为空

2. 边界条件：处理空数组、n 为 0 等情况
3. 性能优化：提前终止条件避免不必要的计算

"""

# 异常处理：检查输入是否为空

if not flowerbed:

    return n <= 0 # 空数组，只能种 0 朵花

if n <= 0:

    return True # 需要种 0 朵花，直接返回 True

count = 0 # 可以种植的花的数量

len\_flowerbed = len(flowerbed)

# 贪心策略：遍历每个位置，如果可以种花就种

# 时间复杂度：O(n)

for i in range(len\_flowerbed):

    # 检查当前位置、前一个位置和后一个位置是否都没有花

    left\_empty = (i == 0) or (flowerbed[i - 1] == 0)

    right\_empty = (i == len\_flowerbed - 1) or (flowerbed[i + 1] == 0)

    if flowerbed[i] == 0 and left\_empty and right\_empty:

        count += 1 # 可以种花

        flowerbed[i] = 1 # 标记为已种花

    # 提前结束，如果已经能满足需求

    if count >= n:

        return True

return count >= n

# 补充题目 4：LeetCode 406. 根据身高重建队列

# 题目描述：假设有打乱顺序的一群人站成一个队列，数组 people 表示队列中一些人的属性（不一定按顺序）。

# 每个 people[i] = [hi, ki] 表示第 i 个人的身高为 hi，前面正好有 ki 个身高大于或等于 hi 的人。

# 请你重新构造并返回输入数组 people 所表示的队列。返回的队列应该格式化为数组 queue，

# 其中 queue[j] = [hj, kj] 是队列中第 j 个人的属性（queue[0] 是排在队列前面的人）。

# 链接：<https://leetcode.cn/problems/queue-reconstruction-by-height/>

def reconstructQueue(people):

"""

根据身高重建队列 - 使用贪心算法重新构造队列

算法思路：

1. 贪心策略：按身高降序排序，身高相同时按  $k_i$  升序排序
2. 遍历排序后的数组，根据  $k_i$  插入到相应位置

Args:

people (List[List[int]]): 包含每个人身高和前面人数的二维列表

people[i] = [hi, ki] 表示第 i 个人的身高为 hi，前面正好有  $k_i$  个身高大于或等于 hi 的人

Returns:

List[List[int]]: 重新构造的队列

时间复杂度:  $O(n^2)$ , 其中 n 是人数, 排序需要  $O(n \log n)$ , 插入操作需要  $O(n^2)$

空间复杂度:  $O(n)$ , 需要一个新数组存储结果

是否最优解: 是, 这是解决此类问题的最优方法

工程化考量:

1. 异常处理: 检查输入是否为空
2. 边界条件: 处理空数组、单元素等情况
3. 性能优化: 排序后使用插入策略保证正确性

"""

```
# 异常处理: 检查输入是否为空或长度不足
if not people or len(people) <= 1:
    return people # 空数组或只有一个人, 直接返回

# 贪心策略: 按身高降序排序, 身高相同时按  $k_i$  升序排序
# 时间复杂度:  $O(n \log n)$ 
people.sort(key=lambda x: (-x[0], x[1]))

result = []

# 遍历排序后的数组, 根据  $k_i$  插入到相应位置
# 时间复杂度:  $O(n^2)$ 
for p in people:
    result.insert(p[1], p) # 插入到索引为  $k_i$  的位置

return result
```

# 补充题目 5: LeetCode 1005. K 次取反后最大化的数组和

# 题目描述: 给你一个整数数组  $\text{nums}$  和一个整数  $k$  , 按以下方法修改数组:

# 1. 选择某个下标  $i$  并将  $\text{nums}[i]$  替换为  $-\text{nums}[i]$  。

# 你可以重复这个过程恰好  $k$  次。你也可以选择同一个下标  $i$  多次。

# 以这种方式修改数组后, 返回数组 可能的最大和 。

```
# 链接: https://leetcode.cn/problems/maximize-sum-of-array-after-k-negations/
```

```
def largestSumAfterKNegations(nums, k):
```

```
    """
```

```
K 次取反后最大化的数组和 - 使用贪心算法计算取反后可能的最大数组和
```

算法思路:

1. 贪心策略: 每次将最小的负数变为正数
2. 如果 k 还有剩余且为奇数, 将最小的数取反

Args:

nums (List[int]): 整数数组

k (int): 取反操作的次数

Returns:

int: 取反后可能的最大数组和

时间复杂度:  $O(n \log n)$ , 其中 n 是数组长度, 排序需要的时间

空间复杂度:  $O(1)$ , 只使用了常数额外空间 (不考虑排序所需的空间)

是否最优解: 是, 贪心策略是解决此类问题的最优方法

工程化考量:

1. 异常处理: 检查输入是否为空
2. 边界条件: 处理空数组、k 为 0 等情况
3. 性能优化: 排序后使用贪心策略

```
"""
```

```
# 异常处理: 检查输入是否为空
```

```
if not nums:
```

```
    return 0 # 空数组, 和为 0
```

```
# 贪心策略: 每次将最小的负数变为正数, 这样可以最大化数组和
```

```
# 排序数组, 从小到大
```

```
# 时间复杂度:  $O(n \log n)$ 
```

```
nums.sort()
```

```
i = 0
```

```
# 尽可能将负数变为正数
```

```
# 时间复杂度:  $O(n)$ 
```

```
while i < len(nums) and k > 0 and nums[i] < 0:
```

```
    nums[i] = -nums[i] # 取反
```

```
    k -= 1 # 减少取反次数
```

```
    i += 1 # 移动到下一个元素
```

```
# 如果 k 还有剩余，且 k 是奇数，需要将最小的正数取反
if k > 0 and k % 2 == 1:
    # 重新排序，找到最小的数
    # 时间复杂度: O(n log n)
    nums.sort()
    nums[0] = -nums[0]  # 将最小的数取反

# 计算数组和
# 时间复杂度: O(n)
return sum(nums)

# 测试函数
if __name__ == "__main__":
    # 测试用例 1: 可以正确找零
    bills1 = [5, 5, 5, 10, 20]
    print("测试用例 1 结果:", lemonadeChange(bills1))  # 期望输出: True

    # 测试用例 2: 无法正确找零
    bills2 = [5, 5, 10, 10, 20]
    print("测试用例 2 结果:", lemonadeChange(bills2))  # 期望输出: False

    # 测试用例 3: 第一个顾客就无法找零
    bills3 = [10, 10]
    print("测试用例 3 结果:", lemonadeChange(bills3))  # 期望输出: False

    # 测试用例 4: 复杂情况
    bills4 = [5, 5, 10, 20, 5, 5, 5, 5, 5, 5, 5, 5, 10, 5, 5, 20, 5, 20, 5]
    print("测试用例 4 结果:", lemonadeChange(bills4))  # 期望输出: True

    # 测试用例 5: 边界情况 - 空数组
    bills5 = []
    print("测试用例 5 结果:", lemonadeChange(bills5))  # 期望输出: True

    # 测试补充题目 1: 分发饼干
    print("\n测试补充题目 1: 分发饼干")
    g1 = [1, 2, 3]
    s1 = [1, 1]
    print("测试用例 1 结果:", findContentChildren(g1, s1))  # 期望输出: 1

    g2 = [1, 2]
    s2 = [1, 2, 3]
    print("测试用例 2 结果:", findContentChildren(g2, s2))  # 期望输出: 2
```

```
# 测试补充题目 2: 分发糖果
print("\n 测试补充题目 2: 分发糖果")
ratings1 = [1, 0, 2]
print("测试用例 1 结果:", candy(ratings1)) # 期望输出: 5

ratings2 = [1, 2, 2]
print("测试用例 2 结果:", candy(ratings2)) # 期望输出: 4

# 测试补充题目 3: 种花问题
print("\n 测试补充题目 3: 种花问题")
flowerbed1 = [1, 0, 0, 0, 1]
print("测试用例 1 结果:", canPlaceFlowers(flowerbed1, 1)) # 期望输出: True

flowerbed2 = [1, 0, 0, 0, 1]
print("测试用例 2 结果:", canPlaceFlowers(flowerbed2, 2)) # 期望输出: False

# 测试补充题目 4: 根据身高重建队列
print("\n 测试补充题目 4: 根据身高重建队列")
people1 = [[7, 0], [4, 4], [7, 1], [5, 0], [6, 1], [5, 2]]
print("测试用例 1 结果:", reconstructQueue(people1)) # 期望输出: [[5, 0], [7, 0], [5, 2], [6, 1], [4, 4], [7, 1]]

# 测试补充题目 5: K 次取反后最大化的数组和
print("\n 测试补充题目 5: K 次取反后最大化的数组和")
nums1 = [4, 2, 3]
k1 = 1
print("测试用例 1 结果:", largestSumAfterKNegations(nums1, k1)) # 期望输出: 5

nums2 = [3, -1, 0, 2]
k2 = 3
print("测试用例 2 结果:", largestSumAfterKNegations(nums2, k2)) # 期望输出: 6

nums3 = [2, -3, -1, 5, -4]
k3 = 2
print("测试用例 3 结果:", largestSumAfterKNegations(nums3, k3)) # 期望输出: 13
```

=====

文件: Code11\_MergeFruits.cpp

=====

```
// 合并果子
// 在一个果园里，多多已经将所有的果子打了下来，而且按果子的不同种类分成了不同的堆。
```

```
// 多多决定把所有的果子合成一堆。  
// 每一次合并，多多可以把两堆果子合并到一起，消耗的体力等于两堆果子的重量之和。  
// 可以看出，所有的果子经过 n-1 次合并之后，就只剩下一堆了。  
// 多多在合并果子时总共消耗的体力等于每次合并所耗体力之和。  
// 因为还要花大力气把这些果子搬回家，所以多多在合并果子时要尽可能地节省体力。  
// 假定每个果子重量都为 1，并且已知果子的种类数和每种果子的数目，  
// 你的任务是设计出合并的次序方案，使多多耗费的体力最少，并输出这个最小的体力耗费值。  
// 测试链接：https://www.luogu.com.cn/problem/P1090  
// 贪心算法专题 - 堆与哈夫曼编码问题集合
```

```
/*  
 * 算法思路：  
 * 1. 贪心策略：哈夫曼编码思想，每次选择重量最小的两堆果子合并  
 * 2. 使用最小堆维护所有果子堆  
 * 3. 每次取出重量最小的两堆果子合并，合并后的重量重新放入堆中  
 * 4. 重复直到只剩下一堆果子，累加合并过程中的体力消耗  
 *  
 * 时间复杂度：O(n * logn) – n 是果子种类数  
 * 空间复杂度：O(n) – 最小堆的空间  
 * 是否最优解：是，这是处理此类问题的最优解法  
 *  
 * 工程化考量：  
 * 1. 异常处理：检查输入是否为空  
 * 2. 边界条件：处理空数组、单个元素等特殊情况  
 * 3. 性能优化：使用优先队列维护最小值  
 * 4. 可读性：清晰的变量命名和注释  
 *  
 * 极端场景与边界场景：  
 * 1. 空输入：fruits 为空数组  
 * 2. 极端值：只有一种果子、所有果子重量相同  
 * 3. 重复数据：多个果子堆重量相同  
 * 4. 有序/逆序数据：果子堆重量按顺序排列  
 *  
 * 跨语言场景与语言特性差异：  
 * 1. Java：使用 PriorityQueue 实现最小堆  
 * 2. C++：使用 priority_queue 实现最小堆  
 * 3. Python：使用 heapq 实现最小堆  
 *  
 * 调试能力构建：  
 * 1. 打印中间过程：在循环中打印当前合并的两堆果子和合并后的重量  
 * 2. 用断言验证中间结果：确保每次合并后堆中元素数量正确  
 * 3. 性能退化排查：检查堆操作的时间复杂度  
 */
```

```
* 与机器学习、图像处理、自然语言处理的联系与应用:  
* 1. 在数据压缩中，哈夫曼编码是经典的贪心算法应用  
* 2. 在决策树构建中，可用于特征选择的贪心策略  
* 3. 在聚类算法中，可用于层次聚类的合并策略  
*/
```

```
// 简单的最小堆实现（数组方式）  
  
#define MAX_HEAP_SIZE 10000  
int heap[MAX_HEAP_SIZE];  
int heapSize = 0;  
  
// 向最小堆中插入元素  
void heapPush(int value) {  
    if (heapSize >= MAX_HEAP_SIZE) return;  
  
    heap[heapSize] = value;  
    int current = heapSize++;  
  
    // 向上调整  
    while (current > 0) {  
        int parent = (current - 1) / 2;  
        if (heap[current] >= heap[parent]) break;  
  
        // 交换  
        int temp = heap[current];  
        heap[current] = heap[parent];  
        heap[parent] = temp;  
  
        current = parent;  
    }  
}  
  
// 从最小堆中取出最小元素  
int heapPop() {  
    if (heapSize <= 0) return -1;  
  
    int result = heap[0];  
    heap[0] = heap[--heapSize];  
  
    // 向下调整  
    int current = 0;  
    while (1) {  
        int left = current * 2 + 1;
```

```
int right = current * 2 + 2;
int smallest = current;

if (left < heapSize && heap[left] < heap[smallest]) {
    smallest = left;
}

if (right < heapSize && heap[right] < heap[smallest]) {
    smallest = right;
}

if (smallest == current) break;

// 交换
int temp = heap[current];
heap[current] = heap[smallest];
heap[smallest] = temp;

current = smallest;
}

return result;
}
```

```
// 合并果子主函数
int mergeFruits(int fruits[], int fruitsSize) {
    // 异常处理：检查输入是否为空
    if (fruits == 0 || fruitsSize == 0) {
        return 0;
    }

    // 边界条件：只有一堆果子，不需要合并
    if (fruitsSize == 1) {
        return 0;
    }

    // 重置堆大小
    heapSize = 0;

    // 将所有果子堆加入最小堆
    for (int i = 0; i < fruitsSize; i++) {
        heapPush(fruits[i]);
    }
```

```

int totalCost = 0; // 总体力消耗

// 重复合并直到只剩下一堆果子
while (heapSize > 1) {
    // 取出重量最小的两堆果子
    int first = heapPop();
    int second = heapPop();

    // 合并两堆果子
    int cost = first + second;
    totalCost += cost;

    // 将合并后的果子堆重新放入堆中
    heapPush(cost);
}

return totalCost;
}

```

```

/*
 * 补充题目 1: LeetCode 703. 数据流中的第 K 大元素（最小堆应用）
 * 题目描述: 设计一个找到数据流中第 K 大元素的类 (class)。注意是排序后的第 K 大元素，不是第 K 个不同的元素。
 * 请实现 KthLargest 类:
 * KthLargest(int k, int[] nums) 使用整数 k 和整数流 nums 初始化对象。
 * int add(int val) 将 val 插入数据流 nums 后，返回当前数据流中第 K 大的元素。
 * 链接: https://leetcode.cn/problems/kth-largest-element-in-a-stream/
 *
 * 算法思路:
 * 1. 使用最小堆维护 K 个最大元素
 * 2. 堆顶元素即为第 K 大元素
 *
 * 时间复杂度: 构造函数 O(n log k)，add 操作 O(log k)
 * 空间复杂度: O(k)，堆的大小为 k
 * 是否最优解: 是，这是解决此类问题的最优方法
 */

```

```

// KthLargest 类实现
class KthLargest {
private:
    // 自定义最大堆比较函数，创建最小堆
    class MinHeapComparator {

```

```

public:
    bool operator()(int a, int b) {
        return a > b; // 这样 priority_queue 会变成最小堆
    }
};

priority_queue<int, vector<int>, MinHeapComparator> minHeap; // 维护 K 个最大元素的最小堆
int k;

public:
    // 构造函数
    KthLargest(int k, vector<int>& nums) {
        this->k = k;
        // 将数组中的元素添加到堆中
        for (int num : nums) {
            add(num); // 复用 add 方法
        }
    }

    // 添加元素到数据流中，并返回当前第 K 大的元素
    int add(int val) {
        // 如果堆的大小小于 k，直接添加
        if (minHeap.size() < k) {
            minHeap.push(val);
        }
        // 如果当前元素比堆顶元素大，替换堆顶元素
        else if (val > minHeap.top()) {
            minHeap.pop(); // 移除堆顶元素（最小的元素）
            minHeap.push(val); // 添加新元素
        }
        // 堆顶元素即为第 K 大元素
        return minHeap.top();
    }
};

/*
 * 补充题目 2: LeetCode 1046. 最后一块石头的重量（最大堆应用）
 * 题目描述：有一堆石头，每块石头的重量都是正整数。
 * 每一回合，从中选出两块 最重的 石头，然后将它们一起粉碎。假设石头的重量分别为 x 和 y，且 x <= y。那么粉碎的可能结果如下：
 * 如果 x == y，那么两块石头都会被完全粉碎；
 * 如果 x != y，那么重量为 x 的石头将会完全粉碎，而重量为 y 的石头新重量为 y-x。
 * 最后，最多只会剩下一块石头。返回此石头的重量。如果没有石头剩下，就返回 0。
 */

```

```
* 链接: https://leetcode.cn/problems/last-stone-weight/
*
* 算法思路:
* 1. 使用最大堆维护所有石头的重量
* 2. 每次取出两个最大的石头进行粉碎
* 3. 如果有剩余, 将剩余重量放回堆中
* 4. 重复直到堆中最多剩下一块石头
*
* 时间复杂度: O(n log n), 其中 n 是石头数量
* 空间复杂度: O(n), 堆的大小为 n
* 是否最优解: 是, 这是解决此类问题的最优方法
*/
```

```
int lastStoneWeight(vector<int>& stones) {
    // 异常处理: 检查输入是否为空
    if (stones.empty()) {
        return 0;
    }

    // 使用最大堆维护所有石头的重量
    priority_queue<int> maxHeap; // 默认是最大堆

    // 将所有石头加入最大堆
    for (int stone : stones) {
        maxHeap.push(stone);
    }

    // 重复粉碎过程
    while (maxHeap.size() > 1) {
        // 取出两个最大的石头
        int y = maxHeap.top(); maxHeap.pop(); // 第一大的石头
        int x = maxHeap.top(); maxHeap.pop(); // 第二大的石头

        // 如果两块石头重量不同, 将剩余重量放回堆中
        if (x != y) {
            maxHeap.push(y - x);
        }
    }

    // 返回最后剩下的石头重量, 如果没有则返回 0
    return maxHeap.empty() ? 0 : maxHeap.top();
}
```

```

/*
 * 补充题目 3: LeetCode 215. 数组中的第 K 个最大元素（最小堆实现）
 * 题目描述: 给定整数数组 nums 和整数 k, 请返回数组中第 k 个最大的元素。
 * 请注意, 你需要找的是数组排序后的第 k 个最大的元素, 而不是第 k 个不同的元素。
 * 链接: https://leetcode.cn/problems/kth-largest-element-in-an-array/
 *
 * 算法思路:
 * 1. 使用最小堆维护 K 个最大元素
 * 2. 堆顶元素即为第 K 大元素
 *
 * 时间复杂度: O(n log k), 其中 n 是数组长度
 * 空间复杂度: O(k), 堆的大小为 k
 * 是否最优解: 是, 这是解决此类问题的高效方法之一
 */

```

```

int findKthLargest(vector<int>& nums, int k) {
    // 异常处理: 检查输入是否有效
    if (nums.empty() || k <= 0 || k > nums.size()) {
        throw invalid_argument("Invalid input");
    }

    // 自定义比较函数, 创建最小堆
    class MinHeapComparator {
public:
    bool operator()(int a, int b) {
        return a > b; // 这样 priority_queue 会变成最小堆
    }
};

// 使用最小堆维护 K 个最大元素
priority_queue<int, vector<int>, MinHeapComparator> minHeap;

// 前 k 个元素直接加入堆中
for (int i = 0; i < k; i++) {
    minHeap.push(nums[i]);
}

// 对于剩下的元素, 如果比堆顶元素大, 则替换堆顶元素
for (int i = k; i < nums.size(); i++) {
    if (nums[i] > minHeap.top()) {
        minHeap.pop();
        minHeap.push(nums[i]);
    }
}

```

```
}

// 堆顶元素即为第 K 大元素
return minHeap.top();
}

/*
 * 补充题目 4: LeetCode 347. 前 K 个高频元素（最大堆应用）
 * 题目描述: 给你一个整数数组 nums 和一个整数 k , 请你返回其中出现频率前 k 高的元素。你可以按 任意顺序 返回答案。
 * 链接: https://leetcode.cn/problems/top-k-frequent-elements/
 *
 * 算法思路:
 * 1. 使用哈希表统计每个元素出现的频率
 * 2. 使用最大堆根据频率排序
 * 3. 取出前 K 个元素
 *
 * 时间复杂度: O(n log n) , 其中 n 是数组长度
 * 空间复杂度: O(n) , 哈希表和堆的空间
 * 是否最优解: 是, 这是解决此类问题的高效方法之一
 */

```

```
vector<int> topKFrequent(vector<int>& nums, int k) {
    // 异常处理: 检查输入是否有效
    if (nums.empty() || k <= 0 || k > nums.size()) {
        return vector<int>();
    }

    // 使用哈希表统计每个元素出现的频率
    unordered_map<int, int> frequencyMap;
    for (int num : nums) {
        frequencyMap[num]++;
    }

    // 自定义比较函数, 根据频率创建最大堆
    class FrequencyComparator {
public:
    bool operator()(const pair<int, int>& a, const pair<int, int>& b) {
        return a.second < b.second; // 这样 priority_queue 会变成最大堆
    }
};

// 使用最大堆根据频率排序
```

```

priority_queue<pair<int, int>, vector<pair<int, int>>, FrequencyComparator> maxHeap;

// 将所有元素加入堆中
for (auto& entry : frequencyMap) {
    maxHeap.push(entry);
}

// 取出前 K 个高频元素
vector<int> result;
for (int i = 0; i < k; i++) {
    result.push_back(maxHeap.top().first);
    maxHeap.pop();
}

return result;
}

/*
 * 补充题目 5: LeetCode 973. 最接近原点的 K 个点（最大堆应用）
 * 题目描述: 给定一个数组 points , 其中 points[i] = [xi, yi] 表示 X-Y 平面上的一个点,
 * 并且是一个整数 k , 返回离原点 (0,0) 最近的 k 个点。
 * 这里, 平面上两点之间的距离是 欧几里德距离 ( $\sqrt{(x_1^2 + y_1^2)}$ )。
 * 你可以按 任何顺序 返回答案。除了点坐标的顺序之外, 答案确保是 唯一 的。
 * 链接: https://leetcode.cn/problems/k-closest-points-to-origin/
 *
 * 算法思路:
 * 1. 使用最大堆维护 K 个最近的点
 * 2. 堆的比较依据是点到原点的距离平方 (避免浮点数运算)
 * 3. 当堆大小超过 K 时, 移除距离最远的点
 *
 * 时间复杂度: O(n log k), 其中 n 是点的数量
 * 空间复杂度: O(k), 堆的大小为 k
 * 是否最优解: 是, 这是解决此类问题的高效方法之一
 */

```

```

vector<vector<int>> kClosest(vector<vector<int>>& points, int k) {
    // 异常处理: 检查输入是否有效
    if (points.empty() || k <= 0 || k > points.size()) {
        return vector<vector<int>>();
    }

    // 计算点到原点的距离平方
    auto distanceSquared = [] (const vector<int>& point) {

```

```

        return point[0] * point[0] + point[1] * point[1];
    };

// 自定义比较函数，创建最大堆（距离大的优先级高）
class DistanceComparator {
public:
    bool operator() (const vector<int>& a, const vector<int>& b) {
        int distA = a[0] * a[0] + a[1] * a[1];
        int distB = b[0] * b[0] + b[1] * b[1];
        return distA < distB; // 这样 priority_queue 会变成最大堆
    }
};

// 使用最大堆维护 K 个最近的点
priority_queue<vector<int>, vector<vector<int>>, DistanceComparator> maxHeap;

// 将点加入堆中
for (auto& point : points) {
    maxHeap.push(point);
    // 如果堆的大小超过 K，移除距离最远的点
    if (maxHeap.size() > k) {
        maxHeap.pop();
    }
}

// 收集结果
vector<vector<int>> result;
while (!maxHeap.empty()) {
    result.push_back(maxHeap.top());
    maxHeap.pop();
}

return result;
}

// 辅助函数：打印数组
void printArray(int arr[], int size) {
    cout << "[";
    for (int i = 0; i < size; i++) {
        cout << arr[i];
        if (i < size - 1) {
            cout << ", ";
        }
    }
}

```

```

    }

    cout << "]" << endl;
}

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

// 辅助函数: 打印二维向量
void printVectorOfVectors(const vector<vector<int>>& vec) {
    cout << "[";
    for (int i = 0; i < vec.size(); i++) {
        cout << "[";
        for (int j = 0; j < vec[i].size(); j++) {
            cout << vec[i][j];
            if (j < vec[i].size() - 1) {
                cout << ", ";
            }
        }
        cout << "]";
        if (i < vec.size() - 1) {
            cout << ", ";
        }
    }
    cout << "]" << endl;
}

// 主函数示例
int main() {
    // 合并果子示例
    int fruits[] = {1, 2, 3, 4, 5};
    int fruitsSize = sizeof(fruits) / sizeof(fruits[0]);
    cout << "合并果子示例: " << mergeFruits(fruits, fruitsSize) << endl; // 期望输出: 33

    // 补充题目 1: KthLargest 示例
}

```

```

vector<int> nums1 = {3, 2, 1, 5, 6, 4};
int k1 = 3;
KthLargest kthLargest(k1, nums1);
cout << "\n 补充题目 1: KthLargest 示例" << endl;
cout << "添加 8 后, 第 3 大元素: " << kthLargest.add(8) << endl; // 应返回 6
cout << "添加 9 后, 第 3 大元素: " << kthLargest.add(9) << endl; // 应返回 8

// 补充题目 2: 最后一块石头的重量示例
vector<int> stones = {2, 7, 4, 1, 8, 1};
cout << "\n 补充题目 2: 最后一块石头的重量示例" << endl;
cout << "最后一块石头的重量: " << lastStoneWeight(stones) << endl; // 应返回 1

// 补充题目 3: 数组中的第 K 个最大元素示例
vector<int> nums2 = {3, 2, 1, 5, 6, 4};
int k2 = 2;
cout << "\n 补充题目 3: 数组中的第 K 个最大元素示例" << endl;
cout << "第 2 大元素: " << findKthLargest(nums2, k2) << endl; // 应返回 5

// 补充题目 4: 前 K 个高频元素示例
vector<int> nums3 = {1, 1, 1, 2, 2, 3};
int k3 = 2;
cout << "\n 补充题目 4: 前 K 个高频元素示例" << endl;
cout << "前 2 个高频元素: ";
printVector(topKFrequent(nums3, k3)); // 应返回 [1, 2] 或 [2, 1]

// 补充题目 5: 最接近原点的 K 个点示例
vector<vector<int>> points = {{1, 3}, {-2, 2}, {5, 8}, {0, 1}};
int k4 = 2;
cout << "\n 补充题目 5: 最接近原点的 K 个点示例" << endl;
cout << "最近的 2 个点: ";
printVectorOfVectors(kClosest(points, k4)); // 应返回 [[-2, 2], [0, 1]] 或 [[0, 1], [-2, 2]]

return 0;
}
=====

文件: Code11_MergeFruits.java
=====

package class094;

import java.util.PriorityQueue;

```

```
// 合并果子 (Merge Fruits)
// 在一个果园里，多多已经将所有的果子打了下来，而且按果子的不同种类分成了不同的堆。
// 多多决定把所有的果子合成一堆。
// 每一次合并，多多可以把两堆果子合并到一起，消耗的体力等于两堆果子的重量之和。
// 可以看出，所有的果子经过 n-1 次合并之后，就只剩下一堆了。
// 多多在合并果子时总共消耗的体力等于每次合并所耗体力之和。
// 因为还要花大力气把这些果子搬回家，所以多多在合并果子时要尽可能地节省体力。
// 假定每个果子重量都为 1，并且已知果子的种类数和每种果子的数目，
// 你的任务是设计出合并的次序方案，使多多耗费的体力最少，并输出这个最小的体力耗费值。
//
// 算法标签：贪心算法(Greedy Algorithm)、哈夫曼编码(Huffman Coding)、最小堆(Min Heap)
// 时间复杂度: O(n*log(n))，其中 n 是果子种类数
// 空间复杂度: O(n)，最小堆的空间
// 测试链接 : https://www.luogu.com.cn/problem/P1090
// 相关题目: LeetCode 1046. 最后一块石头的重量、LeetCode 703. 数据流中的第 K 大元素
// 贪心算法专题 - 堆与哈夫曼编码问题集合
public class Code11_MergeFruits {

    /*
     * 算法思路详解:
     * 1. 贪心策略核心: 哈夫曼编码思想，每次选择重量最小的两堆果子合并,
     *   这样可以确保重量大的果子堆参与合并的次数最少,
     *   从而最小化总体力消耗
     * 2. 数据结构选择: 使用最小堆维护所有果子堆,
     *   便于高效获取当前重量最小的两堆果子
     * 3. 合并过程: 每次取出重量最小的两堆果子合并,
     *   合并后的重量重新放入堆中
     * 4. 结果累加: 重复合并过程直到只剩下一堆果子,
     *   累加每次合并的体力消耗
     *
     * 时间复杂度分析:
     * - O(n*log(n))，其中 n 是果子种类数
     * - 初始建堆: O(n*log(n))
     * - n-1 次合并操作: 每次操作包含 2 次 poll 和 1 次 offer，均为 O(log(n))
     * 空间复杂度分析:
     * - O(n)，最小堆最多存储 n 个元素
     * 是否最优解: 是，这是处理此类哈夫曼编码问题的最优解法
     *
     * 工程化最佳实践:
     * 1. 输入验证: 严格检查输入参数的有效性，防止空指针异常
     * 2. 边界处理: 妥善处理各种边界情况，如空数组、单元素数组等
     * 3. 性能优化: 采用优先队列维护最小值，避免重复排序
     * 4. 代码可读性: 使用语义明确的变量名和详尽的注释
    
```

- \* 5. 资源管理：及时释放不再使用的对象
- \*
- \* 极端场景与边界情况处理：
- \* 1. 空输入场景：fruits 为空数组或 null 时直接返回 0
- \* 2. 单堆场景：只有一堆果子时不需要合并
- \* 3. 相同重量场景：多个果子堆重量相同时的处理
- \* 4. 有序序列场景：果子堆重量已排序的情况
- \* 5. 极值场景：重量差异极大的果子堆
- \*
- \* 跨语言实现差异与优化：
- \* 1. Java 实现：使用 PriorityQueue 实现最小堆，注意比较器设置
- \* 2. C++ 实现：使用 priority\_queue 实现最小堆，注意模板参数
- \* 3. Python 实现：使用 heapq 实现最小堆，注意默认为最小堆
- \* 4. 内存管理：不同语言的垃圾回收机制对性能的影响
- \*
- \* 调试与测试策略：
- \* 1. 过程可视化：在关键节点打印当前合并的果子堆和体力消耗
- \* 2. 断言验证：在每次合并后添加断言确保堆中元素数量正确
- \* 3. 性能监控：跟踪堆操作的实际执行时间
- \* 4. 边界测试：设计覆盖所有边界条件的测试用例
- \* 5. 压力测试：使用大规模数据验证算法稳定性
- \*
- \* 实际应用场景与拓展：
- \* 1. 数据压缩：哈夫曼编码构建最优前缀码
- \* 2. 文件合并：多个有序文件合并为一个文件
- \* 3. 任务调度：多个任务合并执行的最优策略
- \* 4. 网络传输：多个数据包合并传输的优化
- \* 5. 存储管理：多个小文件合并存储的策略
- \*
- \* 算法深入解析：
- \* 1. 哈夫曼编码原理：通过贪心策略构建最优二叉树
- \* 2. 最优性证明：通过数学归纳法可以证明贪心策略的正确性
- \* 3. 策略变体：可扩展为多路合并等变体问题
- \* 4. 问题转换：最小体力消耗 = 所有内部节点权重之和

```
*/  
  
public static int mergeFruits(int[] fruits) {  
    // 异常处理：检查输入是否为空  
    if (fruits == null || fruits.length == 0) {  
        return 0;  
    }
```

```
// 边界条件：只有一堆果子，不需要合并  
if (fruits.length == 1) {
```

```
        return 0;
    }

    // 使用最小堆维护所有果子堆
    PriorityQueue<Integer> minHeap = new PriorityQueue<>();

    // 将所有果子堆加入最小堆
    for (int fruit : fruits) {
        minHeap.offer(fruit);
    }

    int totalCost = 0; // 总体力消耗

    // 重复合并直到只剩下一堆果子
    while (minHeap.size() > 1) {
        // 取出重量最小的两堆果子
        int first = minHeap.poll();
        int second = minHeap.poll();

        // 合并两堆果子
        int cost = first + second;
        totalCost += cost;

        // 将合并后的果子堆重新放入堆中
        minHeap.offer(cost);
    }

    return totalCost;
}

// 测试函数
public static void main(String[] args) {
    // 测试用例 1
    int[] fruits1 = {1, 2, 9};
    System.out.println("测试用例 1 结果: " + mergeFruits(fruits1)); // 期望输出: 15

    // 测试用例 2
    int[] fruits2 = {3, 5, 7, 9};
    System.out.println("测试用例 2 结果: " + mergeFruits(fruits2)); // 期望输出: 45

    // 测试用例 3
    int[] fruits3 = {1, 1, 1, 1, 1};
    System.out.println("测试用例 3 结果: " + mergeFruits(fruits3)); // 期望输出: 12
```

```

// 测试用例 4: 边界情况
int[] fruits4 = {5};
System.out.println("测试用例 4 结果: " + mergeFruits(fruits4)); // 期望输出: 0

// 测试用例 5: 极端情况
int[] fruits5 = {1, 100};
System.out.println("测试用例 5 结果: " + mergeFruits(fruits5)); // 期望输出: 101

// 测试补充题目 1: 最小堆实现的 K 个最小元素
System.out.println("\n 测试补充题目 1: 最小堆实现的 K 个最小元素");
int[] nums1 = {3, 2, 1, 5, 6, 4};
int k1 = 2;
int[] result1 = findKthSmallestElements(nums1, k1);
System.out.print("测试用例 1 结果: [");
for (int i = 0; i < result1.length; i++) {
    System.out.print(result1[i]);
    if (i < result1.length - 1) {
        System.out.print(", ");
    }
}
System.out.println("]"); // 期望输出: [1, 2]

// 测试补充题目 2: 最大堆实现的 K 个最大元素
System.out.println("\n 测试补充题目 2: 最大堆实现的 K 个最大元素");
int[] nums2 = {3, 2, 1, 5, 6, 4};
int k2 = 3;
int[] result2 = findKthLargestElements(nums2, k2);
System.out.print("测试用例 1 结果: [");
for (int i = 0; i < result2.length; i++) {
    System.out.print(result2[i]);
    if (i < result2.length - 1) {
        System.out.print(", ");
    }
}
System.out.println("]"); // 期望输出: [4, 5, 6]

// 测试补充题目 3: 最后一块石头的重量
System.out.println("\n 测试补充题目 3: 最后一块石头的重量");
int[] stones1 = {2, 7, 4, 1, 8, 1};
System.out.println("测试用例 1 结果: " + lastStoneWeight(stones1)); // 期望输出: 1

int[] stones2 = {1};

```

```

System.out.println("测试用例 2 结果: " + lastStoneWeight(stones2)); // 期望输出: 1

// 测试补充题目 4: 数据流中的第 K 大元素
System.out.println("\n 测试补充题目 4: 数据流中的第 K 大元素");
KthLargest kthLargest = new KthLargest(3, new int[]{4, 5, 8, 2});
System.out.println("添加 3 后, 第 3 大元素: " + kthLargest.add(3)); // 返回 4
System.out.println("添加 5 后, 第 3 大元素: " + kthLargest.add(5)); // 返回 5
System.out.println("添加 10 后, 第 3 大元素: " + kthLargest.add(10)); // 返回 5
System.out.println("添加 9 后, 第 3 大元素: " + kthLargest.add(9)); // 返回 8
System.out.println("添加 4 后, 第 3 大元素: " + kthLargest.add(4)); // 返回 8

// 测试补充题目 5: 最小的 K 个数 (优化版本)
System.out.println("\n 测试补充题目 5: 最小的 K 个数 (优化版本)");
int[] nums3 = {3, 2, 1, 5, 6, 4};
int k3 = 2;
int[] result3 = getLeastNumbers(nums3, k3);
System.out.print("测试用例 1 结果: [");
for (int i = 0; i < result3.length; i++) {
    System.out.print(result3[i]);
    if (i < result3.length - 1) {
        System.out.print(", ");
    }
}
System.out.println("]"); // 期望输出: [1, 2]
}

// 补充题目 1: LeetCode 703. 数据流中的第 K 大元素 (最小堆应用)
// 题目描述: 设计一个找到数据流中第 K 大元素的类 (class)。注意是排序后的第 K 大元素, 不是第 K 个不同的元素。
// 请实现 KthLargest 类:
// KthLargest(int k, int[] nums) 使用整数 k 和整数流 nums 初始化对象。
// int add(int val) 将 val 插入数据流 nums 后, 返回当前数据流中第 K 大的元素。
// 链接: https://leetcode.cn/problems/kth-largest-element-in-a-stream/
public static class KthLargest {
    private PriorityQueue<Integer> minHeap; // 维护 K 个最大元素的最小堆
    private int k;

    /**
     * 构造函数
     *
     * @param k 第 K 大的元素
     * @param nums 初始化的数据流
     */

```

- \* 算法思路详解：
  - \* 1. 数据结构选择：使用容量为 k 的最小堆维护 K 个最大元素，
    - \* 堆顶元素即为第 K 大元素
  - \* 2. 初始化过程：遍历初始数据流，通过 add 方法逐个添加元素
  - \* 3. 堆维护策略：始终保持堆中元素个数不超过 k
  - \*
- \* 时间复杂度分析：
  - \* - 构造函数： $O(n \log(k))$ ，其中 n 是初始数组长度
  - \* - add 操作： $O(\log(k))$ ，堆操作的时间复杂度
- \* 空间复杂度分析：
  - \* -  $O(k)$ ，堆的大小始终不超过 k
- \* 是否最优解：是，这是解决动态第 K 大元素问题的最优方法
- \*
- \* 工程化考量：
  - \* 1. 参数验证：检查 k 的有效性和数组的合法性
  - \* 2. 内存管理：预设堆容量避免动态扩容
  - \* 3. 代码复用：复用 add 方法简化初始化逻辑
  - \*
- \* 实际应用场景：
  - \* 1. 实时排行榜：维护用户积分排名
  - \* 2. 数据分析：实时监控关键指标
  - \* 3. 游戏开发：实时更新玩家排名
  - \* 4. 金融系统：监控交易量排名
- \*/

```
public KthLargest(int k, int[] nums) {  
    this.k = k;  
    this.minHeap = new PriorityQueue<>(k); // 容量为 k 的最小堆  
  
    // 将数组中的元素添加到堆中  
    for (int num : nums) {  
        add(num); // 复用 add 方法  
    }  
}
```

```
/**  
 * 添加元素到数据流中，并返回当前第 K 大的元素  
 *  
 * @param val 要添加的元素  
 * @return 当前数据流中第 K 大的元素  
 *  
 * 算法详解与策略分析：

- 1. 插入策略：
  - 若堆未满：直接插入新元素

```

```

*      - 若堆满且新元素大于堆顶: 替换堆顶元素
*      - 否则: 忽略新元素
* 2. 堆维护: 通过堆的自动调整维护 K 个最大元素
* 3. 结果返回: 堆顶元素即为第 K 大元素
*
* 时间复杂度: O(log(k)), 堆操作的时间复杂度
* 空间复杂度: O(1), 不考虑堆本身的空间
*
* 优化策略:
* 1. 提前过滤: 对于小于堆顶的元素直接忽略
* 2. 边界处理: 妥善处理堆为空和堆满的情况
*/
public int add(int val) {
    // 如果堆的大小小于 k, 直接添加
    if (minHeap.size() < k) {
        minHeap.offer(val);
    }
    // 如果当前元素比堆顶元素大, 替换堆顶元素
    else if (val > minHeap.peek()) {
        minHeap.poll(); // 移除堆顶元素 (最小的元素)
        minHeap.offer(val); // 添加新元素
    }
    // 堆顶元素即为第 K 大元素
    return minHeap.peek();
}
}

```

```

// 补充题目 2: LeetCode 1046. 最后一块石头的重量 (最大堆应用)
// 题目描述: 有一堆石头, 每块石头的重量都是正整数。
// 每一回合, 从中选出两块 最重的 石头, 然后将它们一起粉碎。假设石头的重量分别为 x 和 y, 且 x
// <= y。那么粉碎的可能结果如下:
// 如果 x == y, 那么两块石头都会被完全粉碎;
// 如果 x != y, 那么重量为 x 的石头将会完全粉碎, 而重量为 y 的石头新重量为 y-x。
// 最后, 最多只会剩下一块石头。返回此石头的重量。如果没有石头剩下, 就返回 0。
// 链接: https://leetcode.cn/problems/last-stone-weight/
public static int lastStoneWeight(int[] stones) {
    /**
     * 最后一块石头的重量
     *
     * @param stones 石头重量数组
     * @return 最后剩下石头的重量, 没有则返回 0
     *
     * 算法思路详解:
    
```

```

* 1. 数据结构选择：使用最大堆维护所有石头的重量,
*   便于高效获取重量最大的两块石头
* 2. 粉碎过程：每次取出重量最大的两块石头进行粉碎
* 3. 剩余处理：若粉碎后有剩余，将剩余重量重新放入堆中
* 4. 终止条件：重复粉碎过程直到堆中最多剩下一块石头
*
* 时间复杂度分析：
* -  $O(n \log(n))$ ，其中 n 是石头数量
* - 初始建堆： $O(n \log(n))$ 
* - 最多  $n-1$  次操作：每次操作包含 2 次 poll 和可能的 1 次 offer，均为  $O(\log(n))$ 
* 空间复杂度分析：
* -  $O(n)$ ，最大堆最多存储 n 个元素
* 是否最优解：是，这是解决此类石头粉碎问题的最优方法
*
* 工程化最佳实践：
* 1. 输入验证：严格检查输入参数的有效性
* 2. 边界处理：妥善处理空数组和单元素数组
* 3. 资源管理：及时释放不再使用的对象
*
* 实际应用场景：
* 1. 资源合并：多个资源合并为更大的资源
* 2. 任务调度：多个任务合并执行
* 3. 数据处理：多个数据块合并处理
*/
// 异常处理：检查输入是否为空
if (stones == null || stones.length == 0) {
    return 0;
}

// 使用最大堆维护所有石头的重量
PriorityQueue<Integer> maxHeap = new PriorityQueue<>((a, b) -> b - a);

// 将所有石头加入最大堆
for (int stone : stones) {
    maxHeap.offer(stone);
}

// 重复粉碎过程
while (maxHeap.size() > 1) {
    // 取出两个最大的石头
    int y = maxHeap.poll(); // 第一大的石头
    int x = maxHeap.poll(); // 第二大的石头
}

```

```

// 如果两块石头重量不同，将剩余重量放回堆中
if (x != y) {
    maxHeap.offer(y - x);
}

}

// 返回最后剩下的石头重量，如果没有则返回 0
return maxHeap.isEmpty() ? 0 : maxHeap.poll();
}

// 补充题目 3: LeetCode 215. 数组中的第 K 个最大元素（最小堆实现）
// 题目描述：给定整数数组 nums 和整数 k，请返回数组中第 k 个最大的元素。
// 请注意，你需要找的是数组排序后的第 k 个最大的元素，而不是第 k 个不同的元素。
// 链接: https://leetcode.cn/problems/kth-largest-element-in-an-array/
public static int findKthLargest(int[] nums, int k) {

    /**
     * 数组中的第 K 个最大元素
     *
     * @param nums 整数数组
     * @param k 第 K 大的元素
     * @return 第 K 大的元素
     *
     * 算法思路详解：
     * 1. 数据结构选择：使用容量为 k 的最小堆维护 K 个最大元素，
     *    这样堆顶元素即为第 K 大元素
     * 2. 初始化策略：前 k 个元素直接加入堆中
     * 3. 维护机制：对于剩余元素，若大于堆顶则替换堆顶
     * 4. 结果返回：堆顶元素即为所求
     *
     * 时间复杂度分析：
     * - O(n * log(k))，其中 n 是数组长度
     * - 前 k 个元素插入：O(k * log(k))
     * - 剩余 n-k 个元素处理：每次操作 O(log(k))
     *
     * 空间复杂度分析：
     * - O(k)，堆的大小始终为 k
     *
     * 是否最优解：是，这是解决静态数组第 K 大元素问题的高效方法之一
     *
     * 工程化最佳实践：
     * 1. 参数验证：严格检查输入参数的有效性
     * 2. 边界处理：妥善处理 k 的边界情况
     * 3. 性能优化：通过堆维护避免全排序
     *
     * 算法对比与拓展：
    
```

```

* 1. 与快排分区法对比：此方法时间复杂度稳定但空间复杂度较高
* 2. 与全排序法对比：此方法在 k 较小时效率更高
*/
// 异常处理：检查输入是否有效
if (nums == null || nums.length == 0 || k <= 0 || k > nums.length) {
    throw new IllegalArgumentException("Invalid input");
}

// 使用最小堆维护 K 个最大元素
PriorityQueue<Integer> minHeap = new PriorityQueue<>(k);

// 前 k 个元素直接加入堆中
for (int i = 0; i < k; i++) {
    minHeap.offer(nums[i]);
}

// 对于剩下的元素，如果比堆顶元素大，则替换堆顶元素
for (int i = k; i < nums.length; i++) {
    if (nums[i] > minHeap.peek()) {
        minHeap.poll();
        minHeap.offer(nums[i]);
    }
}

// 堆顶元素即为第 K 大元素
return minHeap.peek();
}

// 补充题目 4：LeetCode 347. 前 K 个高频元素（最大堆应用）
// 题目描述：给你一个整数数组 nums 和一个整数 k，请你返回其中出现频率前 k 高的元素。你可以按任意顺序 返回答案。
// 链接：https://leetcode.cn/problems/top-k-frequent-elements/
public static int[] topKFrequent(int[] nums, int k) {
    /**
     * 前 K 个高频元素
     *
     * @param nums 整数数组
     * @param k 前 K 个高频元素
     * @return 出现频率前 K 高的元素数组
     */
    * 算法思路详解：
    * 1. 频率统计：使用哈希表统计每个元素出现的频率，
    * 2. 时间复杂度 O(n)
}

```

```

* 2. 数据结构选择：使用最大堆根据频率排序,
*   便于高效获取高频元素
* 3. 结果提取：取出前 K 个元素构成结果数组
*
* 时间复杂度分析：
* -  $O(n \log(n))$ , 其中 n 是数组长度
* - 频率统计:  $O(n)$ 
* - 建堆:  $O(m \log(m))$ , 其中 m 是不同元素个数
* - 取前 K 个:  $O(k \log(m))$ 
*
* 空间复杂度分析：
* -  $O(n)$ , 哈希表和堆的空间
* 是否最优解：是，这是解决频率统计问题的高效方法之一
*
* 工程化最佳实践：
* 1. 输入验证：严格检查输入参数的有效性
* 2. 边界处理：妥善处理 k 的边界情况
* 3. 内存优化：合理使用数据结构避免内存浪费
*
* 算法优化与变体：
* 1. 使用最小堆维护 K 个高频元素可优化至  $O(n \log(k))$ 
* 2. 使用桶排序可优化至  $O(n)$  时间复杂度
*/
// 异常处理：检查输入是否有效
if (nums == null || nums.length == 0 || k <= 0 || k > nums.length) {
    return new int[0];
}

// 使用哈希表统计每个元素出现的频率
java.util.HashMap<Integer, Integer> frequencyMap = new java.util.HashMap<>();
for (int num : nums) {
    frequencyMap.put(num, frequencyMap.getOrDefault(num, 0) + 1);
}

// 使用最大堆根据频率排序
// 堆中存储的是 Map.Entry<Integer, Integer>, 比较的是频率值
PriorityQueue<java.util.Map.Entry<Integer, Integer>> maxHeap =
    new PriorityQueue<>((a, b) -> b.getValue() - a.getValue());

// 将所有元素加入堆中
maxHeap.addAll(frequencyMap.entrySet());

// 取出前 K 个高频元素
int[] result = new int[k];

```

```

for (int i = 0; i < k; i++) {
    result[i] = maxHeap.poll().getKey();
}

return result;
}

// 补充题目 5: LeetCode 973. 最接近原点的 K 个点（最小堆应用）
// 题目描述: 给定一个数组 points , 其中 points[i] = [xi, yi] 表示 X-Y 平面上的一个点,
// 并且是一个整数 k , 返回离原点 (0, 0) 最近的 k 个点。
// 这里, 平面上两点之间的距离是 欧几里德距离 ( $\sqrt{(x_1^2 + y_1^2)}$ )。
// 你可以按 任何顺序 返回答案。除了点坐标的顺序之外, 答案确保是 唯一 的。
// 链接: https://leetcode.cn/problems/k-closest-points-to-origin/
public static int[][] kClosest(int[][] points, int k) {
    /**
     * 最接近原点的 K 个点
     *
     * @param points 点坐标数组
     * @param k 需要返回的点数
     * @return 离原点最近的 K 个点
     *
     * 算法思路详解:
     * 1. 数据结构选择: 使用最大堆维护 K 个最近的点,
     *    堆顶为距离最近的点
     * 2. 距离计算优化: 使用距离平方避免浮点数运算和开方操作
     * 3. 堆维护策略: 当堆大小超过 K 时, 移除距离最远的点
     * 4. 结果收集: 从堆中取出 K 个点构成结果
     *
     * 时间复杂度分析:
     * -  $O(n \log(k))$ , 其中 n 是点的数量
     * - 每个点的处理:  $O(\log(k))$ 
     * 空间复杂度分析:
     * -  $O(k)$ , 堆的大小始终为 k
     * 是否最优解: 是, 这是解决 K 近邻问题的高效方法之一
     *
     * 工程化最佳实践:
     * 1. 参数验证: 严格检查输入参数的有效性
     * 2. 边界处理: 妥善处理 k 的边界情况
     * 3. 计算优化: 使用距离平方避免浮点运算
     *
     * 实际应用场景:
     * 1. 机器学习: K 近邻算法中的近邻点查找
     * 2. 计算几何: 最近点对问题
    */
}

```

```

* 3. 地理信息系统：最近设施查找
* 4. 游戏开发：碰撞检测中的近邻物体查找
*/
// 异常处理：检查输入是否有效
if (points == null || points.length == 0 || k <= 0 || k > points.length) {
    return new int[0][0];
}

// 使用最大堆维护 K 个最近的点
// 堆中存储的是点的索引，比较的是点到原点的距离平方
PriorityQueue<Integer> maxHeap = new PriorityQueue<>((a, b) -> {
    int distA = points[a][0] * points[a][0] + points[a][1] * points[a][1];
    int distB = points[b][0] * points[b][0] + points[b][1] * points[b][1];
    return distB - distA; // 最大堆，距离大的优先级高
});

// 将点加入堆中
for (int i = 0; i < points.length; i++) {
    maxHeap.offer(i);
    // 如果堆的大小超过 K，移除距离最远的点
    if (maxHeap.size() > k) {
        maxHeap.poll();
    }
}

// 收集结果
int[][] result = new int[k][2];
for (int i = 0; i < k; i++) {
    int index = maxHeap.poll();
    result[i][0] = points[index][0];
    result[i][1] = points[index][1];
}

return result;
}

// 辅助方法：找到数组中最小的 K 个元素
public static int[] findKthSmallestElements(int[] nums, int k) {
    if (nums == null || nums.length == 0 || k <= 0 || k > nums.length) {
        return new int[0];
    }

    // 使用最小堆

```

```

PriorityQueue<Integer> minHeap = new PriorityQueue<>() ;
for (int num : nums) {
    minHeap.offer(num);
}

int[] result = new int[k];
for (int i = 0; i < k; i++) {
    result[i] = minHeap.poll();
}

return result;
}

// 辅助方法：找到数组中最大的 K 个元素
public static int[] findKthLargestElements(int[] nums, int k) {
    if (nums == null || nums.length == 0 || k <= 0 || k > nums.length) {
        return new int[0];
    }

    // 使用最大堆
    PriorityQueue<Integer> maxHeap = new PriorityQueue<>((a, b) -> b - a);
    for (int num : nums) {
        maxHeap.offer(num);
    }

    int[] result = new int[k];
    for (int i = 0; i < k; i++) {
        result[i] = maxHeap.poll();
    }

    return result;
}

// 辅助方法：获取数组中最小的 K 个数（使用最大堆优化）
public static int[] getLeastNumbers(int[] nums, int k) {
    if (nums == null || nums.length == 0 || k <= 0) {
        return new int[0];
    }

    if (k >= nums.length) {
        return nums;
    }
}

```

```

// 使用最大堆维护 K 个最小元素
PriorityQueue<Integer> maxHeap = new PriorityQueue<>((a, b) -> b - a);

// 前 k 个元素直接加入堆中
for (int i = 0; i < k; i++) {
    maxHeap.offer(nums[i]);
}

// 对于剩下的元素，如果比堆顶元素小，则替换堆顶元素
for (int i = k; i < nums.length; i++) {
    if (nums[i] < maxHeap.peek()) {
        maxHeap.poll();
        maxHeap.offer(nums[i]);
    }
}

// 收集结果
int[] result = new int[k];
for (int i = 0; i < k; i++) {
    result[i] = maxHeap.poll();
}

return result;
}
}

```

文件: Code11\_MergeFruits.py

```

# 合并果子 (Merge Fruits)
# 在一个果园里，多多已经将所有的果子打了下来，而且按果子的不同种类分成了不同的堆。
# 多多决定把所有的果子合成一堆。
# 每一次合并，多多可以把两堆果子合并到一起，消耗的体力等于两堆果子的重量之和。
# 可以看出，所有的果子经过 n-1 次合并之后，就只剩下一堆了。
# 多多在合并果子时总共消耗的体力等于每次合并所耗体力之和。
# 因为还要花大力气把这些果子搬回家，所以多多在合并果子时要尽可能地节省体力。
# 假定每个果子重量都为 1，并且已知果子的种类数和每种果子的数目，
# 你的任务是设计出合并的次序方案，使多多耗费的体力最少，并输出这个最小的体力耗费值。
#
# 算法标签：贪心算法(Greedy Algorithm)、堆(Heap)、哈夫曼编码(Huffman Coding)
# 时间复杂度：O(n * logn)，其中 n 是果子种类数
# 空间复杂度：O(n)，最小堆的空间

```

```
# 测试链接 : https://www.luogu.com.cn/problem/P1090
# 相关题目: LeetCode 1046. 最后一块石头的重量、LeetCode 703. 数据流中的第 K 大元素
# 贪心算法专题 - 堆与哈夫曼编码问题集合
```

"""

算法思路详解:

1. 贪心策略: 哈夫曼编码思想, 每次选择重量最小的两堆果子合并
  - 这个策略的核心思想是优先合并较小的果子堆
  - 通过这种方式可以最小化总体力消耗
  - 类似于哈夫曼编码中优先合并频率较低的字符
2. 使用最小堆维护所有果子堆
  - 最小堆能高效地获取当前最小的元素
  - 插入和删除操作的时间复杂度都是  $O(\log n)$
3. 每次取出重量最小的两堆果子合并, 合并后的重量重新放入堆中
  - 通过堆操作实现高效的元素管理
  - 保证每次都能取到当前最小的两个元素
4. 重复直到只剩下一堆果子, 累加合并过程中的体力消耗
  - 通过  $n-1$  次合并操作完成所有果子的合并
  - 累加每次合并的消耗得到总消耗

时间复杂度分析:

- 堆初始化时间复杂度:  $O(n)$
- 每次合并操作时间复杂度:  $O(\log n)$
- 总共需要  $n-1$  次合并操作
- 总体时间复杂度:  $O(n * \log n)$

空间复杂度分析:

- 最小堆存储空间:  $O(n)$
- 其他变量存储空间:  $O(1)$
- 总体空间复杂度:  $O(n)$

是否最优解:

- 是, 这是处理此类问题的最优解法
- 贪心策略保证了局部最优解能导致全局最优解

工程化最佳实践:

1. 异常处理: 检查输入是否为空或格式不正确
2. 边界条件: 处理空数组、单个元素等特殊情况
3. 性能优化: 使用优先队列维护最小值, 避免重复排序
4. 可读性: 清晰的变量命名和详细注释, 便于维护

极端场景与边界情况处理:

1. 空输入: fruits 为空数组
2. 极端值: 只有一种果子、所有果子重量相同
3. 重复数据: 多个果子堆重量相同
4. 有序/逆序数据: 果子堆重量按顺序排列

跨语言实现差异与优化:

1. Java: 使用 PriorityQueue 实现最小堆, 性能稳定
2. C++: 使用 priority\_queue 实现最小堆, 底层实现可能更优化
3. Python: 使用 heapq 实现最小堆, 基于二叉堆算法

调试与测试策略:

1. 打印中间过程: 在循环中打印当前合并的两堆果子和合并后的重量
2. 用断言验证中间结果: 确保每次合并后堆中元素数量正确
3. 性能退化排查: 检查堆操作的时间复杂度
4. 边界测试: 测试空数组、单元素等边界情况

实际应用场景与拓展:

1. 数据压缩: 哈夫曼编码是经典的贪心算法应用
2. 决策树构建: 用于特征选择的贪心策略
3. 聚类算法: 用于层次聚类的合并策略

算法深入解析:

贪心算法在合并果子问题中的应用体现了其核心思想:

1. 局部最优选择: 每次选择重量最小的两堆果子合并
2. 无后效性: 当前的选择不会影响之前的状态
3. 最优子结构: 问题的最优解包含子问题的最优解

这个问题的关键洞察是, 优先合并较小的果子堆能最小化总体力消耗, 这与哈夫曼编码的思想一致。

```
def mergeFruits(fruits):  
    """  
    合并果子主函数 - 使用贪心算法和最小堆计算最小体力耗费值  
    """
```

算法思路:

1. 贪心策略: 哈夫曼编码思想, 每次选择重量最小的两堆果子合并
2. 使用最小堆维护所有果子堆
3. 每次取出重量最小的两堆果子合并, 合并后的重量重新放入堆中

Args:

fruits (List[int]): 果子堆重量列表

`fruits[i]`表示第  $i$  堆果子的重量

Returns:

int: 最小体力耗费值

时间复杂度:  $O(n * \log n)$ , 其中  $n$  是果子种类数

空间复杂度:  $O(n)$ , 最小堆的空间

Examples:

```
>>> mergeFruits([1, 2, 9])
```

```
15
```

```
>>> mergeFruits([3, 5, 7, 9])
```

```
45
```

```
"""
```

# 异常处理: 检查输入是否为空

```
if not fruits:
```

```
    return 0
```

# 边界条件: 只有一堆果子, 不需要合并

```
if len(fruits) == 1:
```

```
    return 0
```

# 使用最小堆维护所有果子堆

# 时间复杂度:  $O(n)$

```
min_heap = fruits[:]
```

```
heapq.heapify(min_heap)
```

```
total_cost = 0 # 总体力消耗
```

# 重複合并直到只剩下一堆果子

# 需要进行  $n-1$  次合并操作

# 时间复杂度:  $O(n * \log n)$

```
while len(min_heap) > 1:
```

# 取出重量最小的两堆果子

# 时间复杂度:  $O(\log n)$

```
first = heapq.heappop(min_heap)
```

```
second = heapq.heappop(min_heap)
```

# 合并两堆果子

```
cost = first + second
```

```
total_cost += cost
```

# 将合并后的果子堆重新放入堆中

```

# 时间复杂度: O(logn)
heapq.heappush(min_heap, cost)

return total_cost

# 测试函数
if __name__ == "__main__":
    # 测试用例 1: 一般情况
    fruits1 = [1, 2, 9]
    print("测试用例 1 结果:", mergeFruits(fruits1)) # 期望输出: 15

    # 测试用例 2: 多个果子堆
    fruits2 = [3, 5, 7, 9]
    print("测试用例 2 结果:", mergeFruits(fruits2)) # 期望输出: 45

    # 测试用例 3: 相同重量的果子堆
    fruits3 = [1, 1, 1, 1, 1]
    print("测试用例 3 结果:", mergeFruits(fruits3)) # 期望输出: 12

    # 测试用例 4: 边界情况 - 只有一堆果子
    fruits4 = [5]
    print("测试用例 4 结果:", mergeFruits(fruits4)) # 期望输出: 0

    # 测试用例 5: 极端情况 - 重量差异很大
    fruits5 = [1, 100]
    print("测试用例 5 结果:", mergeFruits(fruits5)) # 期望输出: 101

# =====
# 贪心算法专题 - 堆与哈夫曼编码问题集合
# =====

# LeetCode 703. 数据流中的第 K 大元素
# https://leetcode.cn/problems/kth-largest-element-in-a-stream/
# 题目描述: 设计一个找到数据流中第 K 大元素的类。注意是排序后的第 K 大元素，不是第 K 个不同的元素。
# 算法思路: 使用最小堆维护前 K 大的元素，堆顶即为第 K 大元素

class KthLargest:
    """
    找到数据流中第 K 大元素的类 - 使用最小堆实现

```

算法思路：

1. 使用最小堆维护前 K 大的元素

2. 堆顶即为第 K 大元素

"""

```
def __init__(self, k, nums):
```

"""

初始化类

Args:

k (int): 第 K 大元素

nums (List[int]): 初始数组

"""

```
import heapq
```

```
self.k = k
```

```
self.min_heap = []
```

# 将初始数组元素添加到最小堆中

```
for num in nums:
```

```
    self.add(num)
```

```
def add(self, val):
```

"""

添加新元素到数据流中，并返回第 K 大元素

Args:

val (int): 新添加的元素

Returns:

int: 数据流中的第 K 大元素

"""

```
import heapq
```

# 如果堆的大小小于 k，直接添加

```
if len(self.min_heap) < self.k:
```

```
    heapq.heappush(self.min_heap, val)
```

# 如果新元素大于堆顶元素，替换堆顶

```
elif val > self.min_heap[0]:
```

```
    heapq.heappushpop(self.min_heap, val)
```

# 返回堆顶元素（第 K 大元素）

```
return self.min_heap[0]
```

# LeetCode 1046. 最后一块石头的重量

# <https://leetcode.cn/problems/last-stone-weight/>

```
# 题目描述：有一堆石头，每块石头的重量都是正整数。每次从中选出两块最重的石头，然后将它们一起粉碎。  
# 假设石头的重量分别为 x 和 y，且 x <= y。那么粉碎的可能结果如下：  
# 如果 x == y，那么两块石头都会被完全粉碎；  
# 如果 x != y，那么重量为 x 的石头会被完全粉碎，而重量为 y 的石头新重量为 y-x。  
# 最后，最多只会剩下一块石头。返回此石头的重量。如果没有石头剩下，就返回 0。  
# 算法思路：使用最大堆（Python 中通过负数实现）维护石头重量，每次取出最大的两个石头进行操作
```

```
def lastStoneWeight(stones):  
    """  
    计算最后一块石头的重量 - 使用最大堆实现
```

算法思路：

1. 使用最大堆维护石头重量
2. 每次取出最大的两个石头进行操作

Args:

stones (List[int]): 石头重量列表

Returns:

int: 最后剩下石头的重量，没有则返回 0

时间复杂度:  $O(n * \log n)$ ，其中  $n$  是石头数量

空间复杂度:  $O(n)$ ，堆的空间

"""

```
# 转换为负数实现最大堆  
# 时间复杂度:  $O(n)$   
max_heap = [-stone for stone in stones]  
heapq.heapify(max_heap)
```

# 循环直到堆中只剩 0 或 1 个元素

# 时间复杂度:  $O(n * \log n)$

```
while len(max_heap) > 1:  
    # 取出两个最大的石头  
    # 时间复杂度:  $O(\log n)$   
    stone1 = -heapq.heappop(max_heap)  
    stone2 = -heapq.heappop(max_heap)
```

# 如果重量不同，将差值放回堆中

# 时间复杂度:  $O(\log n)$

```
if stone1 != stone2:  
    heapq.heappush(max_heap, -(abs(stone1 - stone2)))
```

```
# 返回最后剩下的石头重量或 0
return -max_heap[0] if max_heap else 0

# LeetCode 215. 数组中的第 K 个最大元素
# https://leetcode.cn/problems/kth-largest-element-in-an-array/
# 题目描述：在未排序的数组中找到第 k 个最大的元素。请注意，你需要找的是数组排序后的第 k 个最大的元素，而不是第 k 个不同的元素。
# 算法思路：使用最小堆维护前 K 大的元素，堆顶即为第 K 大元素
```

```
def findKthLargest(nums, k):
    """
    找到数组中的第 K 个最大元素 - 使用最小堆实现
    
```

算法思路：

1. 使用最小堆维护前 K 大的元素
2. 堆顶即为第 K 大元素

Args:

```
nums (List[int]): 未排序数组
k (int): 第 K 大
```

Returns:

```
int: 第 K 大元素
```

时间复杂度:  $O(n * \log k)$ ，其中 n 是数组长度

空间复杂度:  $O(k)$ ，堆的空间

```
"""

```

```
min_heap = []
```

# 遍历数组

# 时间复杂度:  $O(n * \log k)$

```
for num in nums:
```

# 如果堆大小小于 k，直接添加

```
if len(min_heap) < k:
```

```
    heapq.heappush(min_heap, num)
```

# 如果当前元素大于堆顶，替换堆顶

```
elif num > min_heap[0]:
```

```
    heapq.heappushpop(min_heap, num)
```

# 堆顶即为第 K 大元素

```
return min_heap[0]
```

```
# LeetCode 347. 前 K 个高频元素
# https://leetcode.cn/problems/top-k-frequent-elements/
# 题目描述：给你一个整数数组 nums 和一个整数 k，请你返回其中出现频率前 k 高的元素。你可以按任意顺序
# 返回答案。
# 算法思路：使用最小堆维护前 K 高频率的元素，堆顶即为第 K 高频率的元素
```

```
def topKFrequent(nums, k):
    """
    找到前 K 个高频元素 - 使用最小堆实现
```

算法思路：

1. 统计每个元素出现的频率
2. 使用最小堆维护前 K 高频率的元素
3. 堆顶即为第 K 高频率的元素

Args:

nums (List[int]): 整数数组  
k (int): 要返回的元素个数

Returns:

List[int]: 前 K 个高频元素列表

时间复杂度:  $O(n * \log k)$ , 其中 n 是数组长度

空间复杂度:  $O(n + k)$ , 哈希表和堆的空间

"""

```
# 统计每个元素出现的频率
# 时间复杂度: O(n)
freq_map = {}
for num in nums:
    freq_map[num] = freq_map.get(num, 0) + 1
```

# 使用最小堆存储频率前 K 高的元素

```
min_heap = []
```

# 遍历频率映射

```
# 时间复杂度: O(m * log k), 其中 m 是不同元素的个数
for num, freq in freq_map.items():
    # 如果堆大小小于 k, 直接添加(频率, 元素)对
    if len(min_heap) < k:
        heapq.heappush(min_heap, (freq, num))
    # 如果当前元素频率大于堆顶元素的频率, 替换堆顶
    elif freq > min_heap[0][0]:
```

```
heapq.heappushpop(min_heap, (freq, num))
```

```
# 从堆中提取元素
```

```
# 时间复杂度: O(k)
```

```
return [item[1] for item in min_heap]
```

```
# LeetCode 973. 最接近原点的 K 个点
```

```
# https://leetcode.cn/problems/k-closest-points-to-origin/
```

```
# 题目描述: 我们有一个由平面上的点组成的列表 points，需要从中找出 K 个距离原点(0, 0)最近的点。
```

```
# 距离可以通过欧几里德距离计算，但为了简化计算，可以使用距离的平方（避免开根号）。
```

```
# 算法思路: 使用最大堆维护 K 个最近的点，堆顶即为第 K 近的点
```

```
def kClosest(points, k):
```

```
    """
```

```
找到最接近原点的 K 个点 - 使用最大堆实现
```

算法思路:

1. 使用最大堆维护 K 个最近的点

2. 堆顶即为第 K 近的点

Args:

points (List[List[int]]): 点坐标列表 [[x1, y1], [x2, y2], ...]

k (int): 要返回的点个数

Returns:

List[List[int]]: 最接近原点的 K 个点列表

时间复杂度:  $O(n * \log k)$ ，其中 n 是点的数量

空间复杂度:  $O(k)$ ，堆的空间

```
"""
```

```
# 使用最大堆（存储负的距离平方，实现最大堆）
```

```
max_heap = []
```

```
# 遍历所有点
```

```
# 时间复杂度:  $O(n * \log k)$ 
```

```
for point in points:
```

```
    x, y = point
```

```
# 计算距离的平方（避免开根号）
```

```
    distance_squared = x * x + y * y
```

```
# 如果堆大小小于 k，直接添加(负距离平方, 点)对
```

```
    if len(max_heap) < k:
```

```

    heapq.heappush(max_heap, (-distance_squared, point))

# 如果当前点距离小于堆顶元素的距离，替换堆顶
elif distance_squared < -max_heap[0][0]:
    heapq.heappushpop(max_heap, (-distance_squared, point))

# 从堆中提取点
# 时间复杂度: O(k)
return [item[1] for item in max_heap]

# 测试补充题目
if __name__ == "__main__":
    print("\n==== 测试补充题目 ====")

# 测试 LeetCode 703
print("\n测试 LeetCode 703 - 数据流中的第 K 大元素:")
kth_largest = KthLargest(3, [4, 5, 8, 2])
print(kth_largest.add(3)) # 输出: 4
print(kth_largest.add(5)) # 输出: 5
print(kth_largest.add(10)) # 输出: 5
print(kth_largest.add(9)) # 输出: 8
print(kth_largest.add(4)) # 输出: 8

# 测试 LeetCode 1046
print("\n测试 LeetCode 1046 - 最后一块石头的重量:")
print(lastStoneWeight([2, 7, 4, 1, 8, 1])) # 输出: 1
print(lastStoneWeight([1])) # 输出: 1
print(lastStoneWeight([])) # 输出: 0

# 测试 LeetCode 215
print("\n测试 LeetCode 215 - 数组中的第 K 个最大元素:")
print(findKthLargest([3, 2, 1, 5, 6, 4], 2)) # 输出: 5
print(findKthLargest([3, 2, 3, 1, 2, 4, 5, 5, 6], 4)) # 输出: 4

# 测试 LeetCode 347
print("\n测试 LeetCode 347 - 前 K 个高频元素:")
print(topKFrequent([1, 1, 1, 2, 2, 3], 2)) # 输出: [1, 2]或[2, 1]
print(topKFrequent([1], 1)) # 输出: [1]

# 测试 LeetCode 973
print("\n测试 LeetCode 973 - 最接近原点的 K 个点:")
print(kClosest([[1, 3], [-2, 2]], 1)) # 输出: [[-2, 2]]
print(kClosest([[3, 3], [5, -1], [-2, 4]], 2)) # 输出: [[3, 3], [-2, 4]]或[[-2, 4], [3, 3]]

```

=====

文件: Code12\_QueueWater.cpp

=====

```
// 排队接水  
// 有 n 个人在一个水龙头前排队接水，假如每个人接水的时间为 Ti,  
// 请编程找出这 n 个人排队的一种顺序，使得 n 个人的平均等待时间最小。  
// 一个人的等待时间不包括他的接水时间。  
// 如果两个人接水的时间相同，编号更小的人应当排在前面。  
// 测试链接 : https://www.luogu.com.cn/problem/P1223
```

```
/*  
 * 算法思路:  
 * 1. 贪心策略: 按接水时间升序排列  
 * 2. 接水时间短的人排在前面, 可以减少后面人的等待时间  
 * 3. 计算排列后的平均等待时间  
 *  
 * 时间复杂度: O(n * logn) - 主要是排序的时间复杂度  
 * 空间复杂度: O(n) - 存储排序后的索引  
 * 是否最优解: 是, 这是处理此类问题的最优解法  
 *  
 * 工程化考量:  
 * 1. 异常处理: 检查输入是否为空  
 * 2. 边界条件: 处理空数组、单个元素等特殊情况  
 * 3. 性能优化: 使用贪心策略避免穷举  
 * 4. 可读性: 清晰的变量命名和注释  
 *  
 * 极端场景与边界场景:  
 * 1. 空输入: times 为空数组  
 * 2. 极端值: 只有一人、所有人的接水时间相同  
 * 3. 重复数据: 多人接水时间相同  
 * 4. 有序/逆序数据: 接水时间按顺序排列  
 *  
 * 跨语言场景与语言特性差异:  
 * 1. Java: 使用 Arrays.sort 进行排序  
 * 2. C++: 使用 std::sort 进行排序  
 * 3. Python: 使用 sorted 函数或 list.sort() 方法  
 *  
 * 调试能力构建:  
 * 1. 打印中间过程: 在循环中打印当前排列顺序和等待时间  
 * 2. 用断言验证中间结果: 确保排列后等待时间最小  
 * 3. 性能退化排查: 检查排序和遍历的时间复杂度
```

```

/*
* 与机器学习、图像处理、自然语言处理的联系与应用:
* 1. 在任务调度问题中, 贪心算法可用于优化平均等待时间
* 2. 在操作系统中, 可用于进程调度算法设计
* 3. 在网络通信中, 可用于数据包调度优化
*/

```

```

// 简单的排序函数实现(冒泡排序)
void bubbleSort(int times[], int ids[], int n) {
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            // 按接水时间升序排序, 时间相同时按编号升序排序
            if (times[j] > times[j + 1] || (times[j] == times[j + 1] && ids[j] > ids[j + 1])) {
                // 交换时间
                int tempTime = times[j];
                times[j] = times[j + 1];
                times[j + 1] = tempTime;

                // 交换编号
                int tempId = ids[j];
                ids[j] = ids[j + 1];
                ids[j + 1] = tempId;
            }
        }
    }
}

// 排队接水主函数
double queueWater(int times[], int n, int result[]) {
    // 异常处理: 检查输入是否为空
    if (times == 0 || n == 0) {
        result[0] = 0;
        return 0.0;
    }

    // 边界条件: 只有一个人
    if (n == 1) {
        result[0] = 1;
        result[1] = 0;
        return 0.0;
    }

    // 创建编号数组

```

```

int ids[1000]; // 假设最多 1000 人
for (int i = 0; i < n; i++) {
    ids[i] = i + 1; // 编号从 1 开始
}

// 按接水时间升序排序，时间相同时按编号升序排序
bubbleSort(times, ids, n);

// 计算平均等待时间
long long totalWaitTime = 0;
long long waitTime = 0;

// 计算每个人的等待时间
for (int i = 0; i < n - 1; i++) { // 最后一个人没有等待时间
    waitTime += times[i];
    totalWaitTime += waitTime;
}

// 返回排列顺序
for (int i = 0; i < n; i++) {
    result[i] = ids[i];
}

// 返回平均等待时间
return (double) totalWaitTime / n;
}

```

=====

文件: Code12\_QueueWater.java

=====

```

package class094;

import java.util.Arrays;

// 排队接水 (Queue Water)
// 有 n 个人在一个水龙头前排队接水，假如每个人接水的时间为 Ti,
// 请编程找出这 n 个人排队的一种顺序，使得 n 个人的平均等待时间最小。
// 一个人的等待时间不包括他的接水时间。
// 如果两个人接水的时间相同，编号更小的人应当排在前面。
//
// 算法标签：贪心算法(Greedy Algorithm)、排序(Sorting)、平均等待时间优化(Average Waiting Time Optimization)

```

```
// 时间复杂度: O(n*log(n)), 其中 n 是人数
// 空间复杂度: O(n), 存储人员信息数组
// 测试链接 : https://www.luogu.com.cn/problem/P1223
// 相关题目: 任务调度、进程调度优化问题
// 贪心算法专题 - 排队与调度问题集合
public class Code12_QueueWater {

    /*
     * 算法思路详解:
     * 1. 贪心策略核心: 按接水时间升序排列人员,
     * 接水时间短的人排在前面可以最小化后续人员的累计等待时间
     * 2. 排序优化: 通过排序预处理, 将问题转化为确定性最优解
     * 3. 等待时间计算: 第 i 个人的等待时间等于前 i-1 个人接水时间之和
     * 4. 平均时间统计: 总等待时间除以人数得到平均等待时间
     *
     * 时间复杂度分析:
     * - O(n*log(n)), 其中 n 是人数
     * - 排序阶段: O(n*log(n))
     * - 等待时间计算: O(n)
     * 空间复杂度分析:
     * - O(n), 需要额外数组存储人员信息和结果
     * 是否最优解: 是, 这是处理此类排队优化问题的最优解法
     *
     * 工程化最佳实践:
     * 1. 输入验证: 严格检查输入参数的有效性, 防止空指针异常
     * 2. 边界处理: 妥善处理各种边界情况, 如空数组、单人等
     * 3. 性能优化: 采用贪心策略避免穷举所有排列组合
     * 4. 代码可读性: 使用语义明确的变量名和详尽的注释
     * 5. 结果精度: 使用 long 类型避免整数溢出
     *
     * 极端场景与边界情况处理:
     * 1. 空输入场景: times 为空数组或 null 时返回默认结果
     * 2. 单人场景: 只有一个人时平均等待时间为 0
     * 3. 相同时间场景: 多人接水时间相同时按编号排序
     * 4. 有序序列场景: 接水时间已排序的情况
     * 5. 极值场景: 接水时间差异极大的情况
     *
     * 跨语言实现差异与优化:
     * 1. Java 实现: 使用 Arrays.sort 和自定义比较器
     * 2. C++ 实现: 使用 std::sort 和 lambda 表达式
     * 3. Python 实现: 使用 sorted 函数和 key 参数
     * 4. 内存管理: 不同语言的垃圾回收机制对性能的影响
     *
```

\* 调试与测试策略：

- \* 1. 过程可视化：在关键节点打印当前排列顺序和等待时间
- \* 2. 断言验证：在排序后添加断言确保时间升序
- \* 3. 性能监控：跟踪排序和计算的实际执行时间
- \* 4. 边界测试：设计覆盖所有边界条件的测试用例
- \* 5. 压力测试：使用大规模数据验证算法稳定性

\*

\* 实际应用场景与拓展：

- \* 1. 操作系统：进程调度算法优化
- \* 2. 网络通信：数据包传输顺序优化
- \* 3. 生产制造：任务加工顺序优化
- \* 4. 服务行业：客户服务顺序优化
- \* 5. 交通运输：车辆通行顺序优化

\*

\* 算法深入解析：

- \* 1. 贪心策略原理：短作业优先(SJF)调度算法思想
- \* 2. 最优性证明：通过交换论证法可以证明贪心策略的正确性
- \* 3. 数学推导：平均等待时间 =  $\Sigma$  (前  $i-1$  个人接水时间) /  $n$
- \* 4. 策略变体：可扩展为多服务台排队问题

\*/

```
public static class Person {  
    int id;      // 人员编号  
    int time;    // 接水时间  
  
    public Person(int id, int time) {  
        this.id = id;  
        this.time = time;  
    }  
}  
  
public static double[] queueWater(int[] times) {  
    // 异常处理：检查输入是否为空  
    if (times == null || times.length == 0) {  
        return new double[] {0.0};  
    }  
  
    // 边界条件：只有一个人  
    if (times.length == 1) {  
        return new double[] {1, 0.0};  
    }  
  
    int n = times.length;
```

```
// 创建人员数组，保存编号和接水时间
Person[] people = new Person[n];
for (int i = 0; i < n; i++) {
    people[i] = new Person(i + 1, times[i]); // 编号从1开始
}

// 按接水时间升序排序，时间相同时按编号升序排序
Arrays.sort(people, (a, b) -> {
    if (a.time != b.time) {
        return a.time - b.time;
    }
    return a.id - b.id;
});

// 计算平均等待时间
long totalWaitTime = 0;
long waitTime = 0;

// 计算每个人的等待时间
for (int i = 0; i < n - 1; i++) { // 最后一个人没有等待时间
    waitTime += people[i].time;
    totalWaitTime += waitTime;
}

// 返回排列顺序和平均等待时间
double[] result = new double[n + 1];
for (int i = 0; i < n; i++) {
    result[i] = people[i].id;
}
result[n] = (double) totalWaitTime / n;

return result;
}

// 测试函数
public static void main(String[] args) {
    // 测试用例1
    int[] times1 = {1, 2, 3};
    double[] result1 = queueWater(times1);
    System.out.print("测试用例1 排列顺序: ");
    for (int i = 0; i < result1.length - 1; i++) {
        System.out.print((int) result1[i] + " ");
    }
}
```

```
System.out.println();
System.out.println("测试用例 1 平均等待时间: " + String.format("%.2f",
result1[result1.length - 1])); // 期望输出: 1 2 3, 平均等待时间: 1.33

// 测试用例 2
int[] times2 = {5, 1, 3, 2};
double[] result2 = queueWater(times2);
System.out.print("测试用例 2 排列顺序: ");
for (int i = 0; i < result2.length - 1; i++) {
    System.out.print((int) result2[i] + " ");
}
System.out.println();
System.out.println("测试用例 2 平均等待时间: " + String.format("%.2f",
result2[result2.length - 1])); // 期望输出: 2 4 3 1, 平均等待时间: 2.75

// 测试用例 3: 边界情况
int[] times3 = {10};
double[] result3 = queueWater(times3);
System.out.print("测试用例 3 排列顺序: ");
for (int i = 0; i < result3.length - 1; i++) {
    System.out.print((int) result3[i] + " ");
}
System.out.println();
System.out.println("测试用例 3 平均等待时间: " + String.format("%.2f",
result3[result3.length - 1])); // 期望输出: 1, 平均等待时间: 0.00

// 测试用例 4: 相同时间
int[] times4 = {3, 3, 3};
double[] result4 = queueWater(times4);
System.out.print("测试用例 4 排列顺序: ");
for (int i = 0; i < result4.length - 1; i++) {
    System.out.print((int) result4[i] + " ");
}
System.out.println();
System.out.println("测试用例 4 平均等待时间: " + String.format("%.2f",
result4[result4.length - 1])); // 期望输出: 1 2 3, 平均等待时间: 4.00
}
```

=====

文件: Code12\_QueueWater.py

=====

```
# 排队接水 (Queue Water)
# 有 n 个人在一个水龙头前排队接水，假如每个人接水的时间为 Ti,
# 请编程找出这 n 个人排队的一种顺序，使得 n 个人的平均等待时间最小。
# 一个人的等待时间不包括他的接水时间。
# 如果两个人接水的时间相同，编号更小的人应当排在前面。
#
# 算法标签：贪心算法(Greedy Algorithm)、排序(Sorting)
# 时间复杂度: O(n * logn)，其中 n 是人数
# 空间复杂度: O(n)，存储排序后的索引
# 测试链接 : https://www.luogu.com.cn/problem/P1223
# 相关题目：任务调度、进程调度
# 贪心算法专题 - 排序与调度问题集合
```

"""

算法思路详解:

1. 贪心策略: 按接水时间升序排列
  - 这个策略的核心思想是让接水时间短的人先接水
  - 这样可以减少后面人的等待时间，从而最小化平均等待时间
  - 符合直觉: 短任务优先原则
2. 接水时间短的人排在前面，可以减少后面人的等待时间
  - 这是问题的关键洞察
  - 通过数学证明可以验证这个贪心策略的正确性
  - 如果有两个人 i 和 j,  $t_i < t_j$ , 那么 i 排在 j 前面能得到更小的平均等待时间
3. 计算排列后的平均等待时间
  - 等待时间是前面所有人接水时间的总和
  - 平均等待时间是所有人等待时间的平均值

时间复杂度分析:

- 排序时间复杂度:  $O(n * \log n)$ ，其中 n 是人数
- 计算等待时间时间复杂度:  $O(n)$
- 总体时间复杂度:  $O(n * \log n)$

空间复杂度分析:

- 存储人员信息空间:  $O(n)$
- 其他变量存储空间:  $O(1)$
- 总体空间复杂度:  $O(n)$

是否最优解:

- 是，这是处理此类问题的最优解法
- 贪心策略保证了局部最优解能导致全局最优解
- 可以通过交换论证法证明其正确性

## 工程化最佳实践:

1. 异常处理: 检查输入是否为空或格式不正确
2. 边界条件: 处理空数组、单个元素等特殊情况
3. 性能优化: 使用贪心策略避免穷举所有排列
4. 可读性: 清晰的变量命名和详细注释, 便于维护

## 极端场景与边界情况处理:

1. 空输入: times 为空数组
2. 极端值: 只有一人、所有人的接水时间相同
3. 重复数据: 多人接水时间相同
4. 有序/逆序数据: 接水时间按顺序排列

## 跨语言实现差异与优化:

1. Java: 使用 Arrays.sort 进行排序, 性能稳定
2. C++: 使用 std::sort 进行排序, 底层实现可能更优化
3. Python: 使用 sorted 函数或 list.sort() 方法, 基于 Timsort 算法

## 调试与测试策略:

1. 打印中间过程: 在循环中打印当前排列顺序和等待时间
2. 用断言验证中间结果: 确保排列后等待时间最小
3. 性能退化排查: 检查排序和遍历的时间复杂度
4. 边界测试: 测试空数组、单元素等边界情况

## 实际应用场景与拓展:

1. 任务调度问题: 在操作系统中优化平均等待时间
2. 进程调度: 在操作系统中用于进程调度算法设计
3. 网络通信: 在网络通信中用于数据包调度优化

## 算法深入解析:

贪心算法在排队接水问题中的应用体现了其核心思想:

1. 局部最优选择: 每次选择接水时间最短的人
2. 无后效性: 当前的选择不会影响之前的状态
3. 最优子结构: 问题的最优解包含子问题的最优解

这个问题的关键洞察是, 短任务优先策略能最小化平均等待时间, 这可以通过数学证明验证。

"""

```
def queueWater(times):
    """
    排队接水主函数 - 使用贪心算法计算最优排队顺序和平均等待时间
    """
```

## 算法思路:

1. 贪心策略：按接水时间升序排列
2. 接水时间短的人排在前面，可以减少后面人的等待时间
3. 计算排列后的平均等待时间

Args:

times (List[int]): 接水时间列表  
times[i]表示第 i+1 个人的接水时间（编号从 1 开始）

Returns:

tuple: (排列顺序列表, 平均等待时间)  
- 排列顺序列表: 最优排队顺序中每个人的编号  
- 平均等待时间: 所有人等待时间的平均值, 保留两位小数

时间复杂度:  $O(n * \log n)$ , 其中 n 是人数

空间复杂度:  $O(n)$ , 存储排序后的索引

Examples:

```
>>> queueWater([1, 2, 3])
([1, 2, 3], 1.33)
>>> queueWater([5, 1, 3, 2])
([2, 4, 3, 1], 2.75)
```

"""

```
# 异常处理: 检查输入是否为空
if not times:
    return ([], 0.0)

# 边界条件: 只有一个人
if len(times) == 1:
    return ([1], 0.0)

n = len(times)

# 创建人员列表, 保存编号和接水时间
# 时间复杂度:  $O(n)$ 
people = [(i + 1, times[i]) for i in range(n)] # 编号从 1 开始

# 按接水时间升序排序, 时间相同时按编号升序排序
# 关键点: 贪心策略的实现, 短任务优先
# 时间复杂度:  $O(n * \log n)$ 
people.sort(key=lambda x: (x[1], x[0]))

# 计算平均等待时间
total_wait_time = 0
```

```
wait_time = 0

# 计算每个人的等待时间
# 等待时间是前面所有人接水时间的总和
# 时间复杂度: O(n)
for i in range(n - 1): # 最后一个人没有等待时间
    wait_time += people[i][1]
    total_wait_time += wait_time

# 返回排列顺序和平均等待时间
order = [person[0] for person in people]
avg_wait_time = total_wait_time / n

return (order, round(avg_wait_time, 2))

# 测试函数
if __name__ == "__main__":
    # 测试用例 1: 一般情况
    times1 = [1, 2, 3]
    order1, avg_time1 = queueWater(times1)
    print("测试用例 1 排列顺序:", order1)
    print("测试用例 1 平均等待时间:", avg_time1) # 期望输出: [1, 2, 3], 平均等待时间: 1.33

    # 测试用例 2: 无序时间
    times2 = [5, 1, 3, 2]
    order2, avg_time2 = queueWater(times2)
    print("测试用例 2 排列顺序:", order2)
    print("测试用例 2 平均等待时间:", avg_time2) # 期望输出: [2, 4, 3, 1], 平均等待时间: 2.75

    # 测试用例 3: 边界情况 - 只有一人
    times3 = [10]
    order3, avg_time3 = queueWater(times3)
    print("测试用例 3 排列顺序:", order3)
    print("测试用例 3 平均等待时间:", avg_time3) # 期望输出: [1], 平均等待时间: 0.0

    # 测试用例 4: 相同时间 - 验证编号排序
    times4 = [3, 3, 3]
    order4, avg_time4 = queueWater(times4)
    print("测试用例 4 排列顺序:", order4)
    print("测试用例 4 平均等待时间:", avg_time4) # 期望输出: [1, 2, 3], 平均等待时间: 4.0
```

=====

文件: Code13\_SouvenirGrouping.cpp

```
=====  
// 纪念品分组  
// 元旦快到了，校学生会让乐乐负责新年晚会的纪念品发放工作。  
// 为使得参加晚会的同学所获得的纪念品价值相对均衡，他要把购来的纪念品根据价格进行分组，  
// 但每组最多只能包括两件纪念品，并且每组纪念品的价格之和不能超过一个给定的整数。  
// 为了保证在尽量短的时间内发完所有纪念品，乐乐希望分组的数目最少。  
// 你的任务是写一个程序，找出所有分组方案中分组数最少的一种，输出最少的分组数目。  
// 测试链接 : https://www.luogu.com.cn/problem/P1094
```

/\*

\* 算法思路:

- \* 1. 贪心策略：排序后使用双指针，最小价格和最大价格配对
- \* 2. 将纪念品价格数组排序
- \* 3. 使用双指针，左指针指向最小价格，右指针指向最大价格
- \* 4. 如果两件纪念品价格之和不超过上限，则分为一组，两个指针都移动
- \* 5. 如果超过上限，则最大价格的纪念品单独分为一组，只移动右指针

\*

\* 时间复杂度:  $O(n * \log n)$  - 主要是排序的时间复杂度

\* 空间复杂度:  $O(1)$  - 只使用了常数额外空间

\* 是否最优解: 是，这是处理此类问题的最优解法

\*

\* 工程化考量:

- \* 1. 异常处理: 检查输入是否为空
- \* 2. 边界条件: 处理空数组、单个元素等特殊情况
- \* 3. 性能优化: 使用双指针避免重复遍历
- \* 4. 可读性: 清晰的变量命名和注释

\*

\* 极端场景与边界场景:

- \* 1. 空输入: prices 为空数组
- \* 2. 极端值: 只有一件纪念品、所有纪念品价格相同
- \* 3. 重复数据: 多件纪念品价格相同
- \* 4. 有序/逆序数据: 纪念品价格按顺序排列

\*

\* 跨语言场景与语言特性差异:

- \* 1. Java: 使用 Arrays.sort 进行排序
- \* 2. C++: 使用 std::sort 进行排序
- \* 3. Python: 使用 sorted 函数或 list.sort() 方法

\*

\* 调试能力构建:

- \* 1. 打印中间过程: 在循环中打印当前配对的纪念品和分组情况
- \* 2. 用断言验证中间结果: 确保每组价格之和不超过上限

```
* 3. 性能退化排查：检查排序和遍历的时间复杂度  
*  
* 与机器学习、图像处理、自然语言处理的联系与应用：  
* 1. 在资源分配问题中，贪心算法可用于优化分组策略  
* 2. 在推荐系统中，可用于物品配对推荐  
* 3. 在数据挖掘中，可用于相似物品聚类  
*/
```

```
// 简单的排序函数实现（冒泡排序）  
void bubbleSort(int arr[], int n) {  
    for (int i = 0; i < n - 1; i++) {  
        for (int j = 0; j < n - i - 1; j++) {  
            if (arr[j] > arr[j + 1]) {  
                // 交换元素  
                int temp = arr[j];  
                arr[j] = arr[j + 1];  
                arr[j + 1] = temp;  
            }  
        }  
    }  
}
```

```
// 纪念品分组主函数  
int souvenirGrouping(int w, int prices[], int pricesSize) {  
    // 异常处理：检查输入是否为空  
    if (prices == 0 || pricesSize == 0) {  
        return 0;  
    }
```

```
// 边界条件：只有一件纪念品  
if (pricesSize == 1) {  
    return 1;  
}
```

```
// 使用冒泡排序对纪念品价格数组排序  
bubbleSort(prices, pricesSize);  
  
int left = 0;           // 左指针，指向最小价格  
int right = pricesSize - 1; // 右指针，指向最大价格  
int groups = 0;          // 分组数
```

```
// 双指针遍历  
while (left <= right) {
```

```

// 如果两件纪念品价格之和不超过上限
if (prices[left] + prices[right] <= w) {
    left++; // 最小价格纪念品被分组
}
// 最大价格纪念品被分组（无论是否与最小价格纪念品配对）
right--;
groups++; // 分组数加 1
}

return groups;
}
=====

文件: Code13_SouvenirGrouping.java
=====

package class094;

import java.util.Arrays;

// 纪念品分组 (Souvenir Grouping)
// 元旦快到了，校学生会让乐乐负责新年晚会的纪念品发放工作。
// 为使得参加晚会的同学所获得的纪念品价值相对均衡，他要把购来的纪念品根据价格进行分组，
// 但每组最多只能包括两件纪念品，并且每组纪念品的价格之和不能超过一个给定的整数。
// 为了保证在尽量短的时间内发完所有纪念品，乐乐希望分组的数目最少。
// 你的任务是写一个程序，找出所有分组方案中分组数最少的一种，输出最少的分组数目。
//
// 算法标签：贪心算法(Greedy Algorithm)、双指针(Double Pointers)、排序(Sorting)
// 时间复杂度：O(n*log(n))，其中 n 是纪念品数量
// 空间复杂度：O(1)，仅使用常数额外空间
// 测试链接：https://www.luogu.com.cn/problem/P1094
// 相关题目：LeetCode 11. 盛最多水的容器、LeetCode 167. 两数之和 II
// 贪心算法专题 - 配对与分组问题集合
public class Code13_SouvenirGrouping {

/*
 * 算法思路详解：
 * 1. 贪心策略核心：排序后使用双指针，将最小价格和最大价格的纪念品配对，这样可以最大化每组的利用率，从而最小化分组数
 * 2. 排序优化：通过升序排序，为双指针策略奠定基础
 * 3. 配对机制：使用双指针分别指向当前最小和最大价格的纪念品
 * 4. 分组决策：
 *      - 若两件纪念品价格之和不超过上限则配对
 */
}

```

- \* - 否则最大价格纪念品单独分组
- \*
- \* 时间复杂度分析:
  - \* -  $O(n \log(n))$ , 其中 n 是纪念品数量
  - \* - 排序阶段:  $O(n \log(n))$
  - \* - 双指针遍历:  $O(n)$
- \* 空间复杂度分析:
  - \* -  $O(1)$ , 仅使用了常数级别的额外空间存储指针和计数器
- \* 是否最优解: 是, 这是处理此类配对分组问题的最优解法
- \*
- \* 工程化最佳实践:
  1. 输入验证: 严格检查输入参数的有效性, 防止空指针异常
  2. 边界处理: 妥善处理各种边界情况, 如空数组、单元素等
  3. 性能优化: 采用双指针策略避免重复遍历
  4. 代码可读性: 使用语义明确的变量名和详尽的注释
  5. 条件判断优化: 合并边界条件判断
- \*
- \* 极端场景与边界情况处理:
  1. 空输入场景: prices 为空数组或 null 时直接返回 0
  2. 单物品场景: 只有一件纪念品时返回 1
  3. 相同价格场景: 多件纪念品价格相同时的处理
  4. 有序序列场景: 纪念品价格已排序的情况
  5. 极值场景: 价格差异极大的情况
- \*
- \* 跨语言实现差异与优化:
  1. Java 实现: 使用 Arrays.sort 和双指针遍历
  2. C++ 实现: 使用 std::sort 和数组索引操作
  3. Python 实现: 使用 sorted 函数和列表索引
  4. 内存管理: 不同语言的垃圾回收机制对性能的影响
- \*
- \* 调试与测试策略:
  1. 过程可视化: 在关键节点打印当前配对的纪念品和分组情况
  2. 断言验证: 在每次配对后添加断言确保价格和不超过上限
  3. 性能监控: 跟踪排序和遍历的实际执行时间
  4. 边界测试: 设计覆盖所有边界条件的测试用例
  5. 压力测试: 使用大规模数据验证算法稳定性
- \*
- \* 实际应用场景与拓展:
  1. 资源分配: 服务器资源配对分配
  2. 任务调度: 任务配对执行优化
  3. 物流配送: 货物配载优化
  4. 电商推荐: 商品组合推荐
  5. 金融投资: 资产配对投资

```
*  
* 算法深入解析:  
* 1. 贪心策略原理: 通过最小值与最大值配对实现整体最优  
* 2. 最优性证明: 使用交换论证法可以证明贪心策略的正确性  
* 3. 策略变体: 可扩展为多物品分组等变体问题  
* 4. 问题转换: 最少分组数 =  $(n+1)/2$  到 n 之间的最优值  
*/
```

```
public static int souvenirGrouping(int w, int[] prices) {  
    // 异常处理: 检查输入是否为空  
    if (prices == null || prices.length == 0) {  
        return 0;  
    }  
  
    // 边界条件: 只有一件纪念品  
    if (prices.length == 1) {  
        return 1;  
    }  
  
    // 排序纪念品价格数组  
    Arrays.sort(prices);  
  
    int left = 0;           // 左指针, 指向最小价格  
    int right = prices.length - 1; // 右指针, 指向最大价格  
    int groups = 0;         // 分组数  
  
    // 双指针遍历  
    while (left <= right) {  
        // 如果两件纪念品价格之和不超过上限  
        if (prices[left] + prices[right] <= w) {  
            left++; // 最小价格纪念品被分组  
        }  
        // 最大价格纪念品被分组 (无论是否与最小价格纪念品配对)  
        right--;  
        groups++; // 分组数加 1  
    }  
  
    return groups;  
}
```

```
// 测试函数  
public static void main(String[] args) {  
    // 测试用例 1  
    int w1 = 10;
```

```

int[] prices1 = {1, 2, 3, 4, 5, 6, 7, 8, 9};
System.out.println("测试用例 1 结果: " + souvenirGrouping(w1, prices1)); // 期望输出: 5

// 测试用例 2
int w2 = 5;
int[] prices2 = {1, 2, 3, 4, 5};
System.out.println("测试用例 2 结果: " + souvenirGrouping(w2, prices2)); // 期望输出: 3

// 测试用例 3
int w3 = 100;
int[] prices3 = {50, 50, 50, 50};
System.out.println("测试用例 3 结果: " + souvenirGrouping(w3, prices3)); // 期望输出: 2

// 测试用例 4: 边界情况
int w4 = 10;
int[] prices4 = {5};
System.out.println("测试用例 4 结果: " + souvenirGrouping(w4, prices4)); // 期望输出: 1

// 测试用例 5: 极端情况
int w5 = 15;
int[] prices5 = {1, 1, 1, 1, 1, 10, 10, 10, 10};
System.out.println("测试用例 5 结果: " + souvenirGrouping(w5, prices5)); // 期望输出: 6
}

}
=====
```

文件: Code13\_SouvenirGrouping.py

```

=====
# 纪念品分组 (Souvenir Grouping)
# 元旦快到了, 校学生会让乐乐负责新年晚会的纪念品发放工作。
# 为使得参加晚会的同学所获得的纪念品价值相对均衡, 他要把购来的纪念品根据价格进行分组,
# 但每组最多只能包括两件纪念品, 并且每组纪念品的价格之和不能超过一个给定的整数。
# 为了保证在尽量短的时间内发完所有纪念品, 乐乐希望分组的数目最少。
# 你的任务是写一个程序, 找出所有分组方案中分组数最少的一种, 输出最少的分组数目。
#
# 算法标签: 贪心算法(Greedy Algorithm)、双指针(Two Pointers)、排序(Sorting)
# 时间复杂度: O(n * logn), 其中 n 是纪念品数量
# 空间复杂度: O(1), 仅使用常数额外空间
# 测试链接 : https://www.luogu.com.cn/problem/P1094
# 相关题目: LeetCode 11. 盛最多水的容器、LeetCode 167. 两数之和 II
# 贪心算法专题 - 双指针与配对问题集合
```

"""

## 算法思路详解:

1. 贪心策略: 排序后使用双指针, 最小价格和最大价格配对
  - 这个策略的核心思想是让价格差异最大的纪念品配对
  - 如果最便宜和最贵的纪念品能配对, 那么其他配对方案都不会更优
  - 这样可以最大化每组的利用效率, 从而最小化分组数
2. 将纪念品价格数组排序
  - 排序是应用双指针技术的前提
  - 排序后可以使用左指针指向最小价格, 右指针指向最大价格
3. 使用双指针, 左指针指向最小价格, 右指针指向最大价格
  - 左指针从数组开始向右移动
  - 右指针从数组末尾向左移动
  - 双指针技术能高效地处理配对问题
4. 如果两件纪念品价格之和不超过上限, 则分为一组, 两个指针都移动
  - 这体现了贪心策略: 尽可能让两个纪念品配对
  - 配对成功后, 两个指针都向中间移动
5. 如果超过上限, 则最大价格的纪念品单独分为一组, 只移动右指针
  - 最大价格的纪念品无法与任何其他纪念品配对
  - 只能单独分为一组, 右指针向左移动

## 时间复杂度分析:

- 排序时间复杂度:  $O(n * \log n)$ , 其中  $n$  是纪念品数量
- 双指针遍历时间复杂度:  $O(n)$
- 总体时间复杂度:  $O(n * \log n)$

## 空间复杂度分析:

- 排序空间复杂度:  $O(\log n)$  (取决于排序算法实现)
- 其他变量存储空间:  $O(1)$
- 总体空间复杂度:  $O(1)$

## 是否最优解:

- 是, 这是处理此类问题的最优解法
- 贪心策略保证了局部最优解能导致全局最优解
- 可以通过交换论证法证明其正确性

## 工程化最佳实践:

1. 异常处理: 检查输入是否为空或格式不正确
2. 边界条件: 处理空数组、单个元素等特殊情况
3. 性能优化: 使用双指针避免重复遍历

4. 可读性：清晰的变量命名和详细注释，便于维护

极端场景与边界情况处理：

1. 空输入：prices 为空数组
2. 极端值：只有一件纪念品、所有纪念品价格相同
3. 重复数据：多件纪念品价格相同
4. 有序/逆序数据：纪念品价格按顺序排列

跨语言实现差异与优化：

1. Java：使用 Arrays.sort 进行排序，性能稳定
2. C++：使用 std::sort 进行排序，底层实现可能更优化
3. Python：使用 sorted 函数或 list.sort() 方法，基于 Timsort 算法

调试与测试策略：

1. 打印中间过程：在循环中打印当前配对的纪念品和分组情况
2. 用断言验证中间结果：确保每组价格之和不超过上限
3. 性能退化排查：检查排序和遍历的时间复杂度
4. 边界测试：测试空数组、单元素等边界情况

实际应用场景与拓展：

1. 资源分配问题：在资源分配中优化分组策略
2. 推荐系统：用于物品配对推荐
3. 数据挖掘：用于相似物品聚类

算法深入解析：

贪心算法在纪念品分组问题中的应用体现了其核心思想：

1. 局部最优选择：每次选择价格差异最大的纪念品配对
2. 无后效性：当前的选择不会影响之前的状态
3. 最优子结构：问题的最优解包含子问题的最优解

这个问题的关键洞察是，最小价格和最大价格的配对策略能最小化分组数，这可以通过数学证明验证。

"""

```
def souvenirGrouping(w, prices):
```

"""

纪念品分组主函数 - 使用贪心算法和双指针技术计算最少分组数目

算法思路：

1. 贪心策略：排序后使用双指针，最小价格和最大价格配对
2. 将纪念品价格数组排序
3. 使用双指针配对纪念品

Args:

w (int): 每组纪念品价格之和的上限  
prices (List[int]): 纪念品价格列表  
prices[i] 表示第 i 件纪念品的价格

Returns:

int: 最少的分组数目

时间复杂度:  $O(n * \log n)$ , 其中 n 是纪念品数量

空间复杂度:  $O(1)$ , 仅使用常数额外空间

Examples:

```
>>> souvenirGrouping(10, [1, 2, 3, 4, 5, 6, 7, 8, 9])  
5
```

```
>>> souvenirGrouping(5, [1, 2, 3, 4, 5])  
3
```

"""

# 异常处理: 检查输入是否为空

```
if not prices:  
    return 0
```

# 边界条件: 只有一件纪念品

```
if len(prices) == 1:  
    return 1
```

# 排序纪念品价格数组

# 时间复杂度:  $O(n * \log n)$   
prices.sort()

```
left = 0          # 左指针, 指向最小价格
```

```
right = len(prices) - 1 # 右指针, 指向最大价格
```

```
groups = 0        # 分组数
```

# 双指针遍历

# 时间复杂度:  $O(n)$

```
while left <= right:
```

# 如果两件纪念品价格之和不超过上限

```
if prices[left] + prices[right] <= w:
```

```
    left += 1      # 最小价格纪念品被分组
```

# 最大价格纪念品被分组 (无论是否与最小价格纪念品配对)

```
right -= 1
```

```
groups += 1        # 分组数加 1
```

```

return groups

# 测试函数
if __name__ == "__main__":
    # 测试用例 1: 一般情况
    w1 = 10
    prices1 = [1, 2, 3, 4, 5, 6, 7, 8, 9]
    print("测试用例 1 结果:", souvenirGrouping(w1, prices1)) # 期望输出: 5

    # 测试用例 2: 较小的上限
    w2 = 5
    prices2 = [1, 2, 3, 4, 5]
    print("测试用例 2 结果:", souvenirGrouping(w2, prices2)) # 期望输出: 3

    # 测试用例 3: 相同价格的纪念品
    w3 = 100
    prices3 = [50, 50, 50, 50]
    print("测试用例 3 结果:", souvenirGrouping(w3, prices3)) # 期望输出: 2

    # 测试用例 4: 边界情况 - 只有一件纪念品
    w4 = 10
    prices4 = [5]
    print("测试用例 4 结果:", souvenirGrouping(w4, prices4)) # 期望输出: 1

    # 测试用例 5: 极端情况 - 价格差异很大
    w5 = 15
    prices5 = [1, 1, 1, 1, 1, 10, 10, 10, 10]
    print("测试用例 5 结果:", souvenirGrouping(w5, prices5)) # 期望输出: 6

```

---

文件: Code14\_MinimumAbsoluteDifference.cpp

---

```

// 最小绝对差
// 给你一个整数数组，其中数组中任意两个元素之间的绝对差的最小值。
// 测试链接 : https://www.hackerrank.com/challenges/minimum-absolute-difference-in-an-
array/problem

```

```

/*
 * 算法思路:
 * 1. 贪心策略: 排序后相邻元素的差值最小
 * 2. 将数组排序

```

- \* 3. 遍历相邻元素，计算差值，找出最小值
- \*
- \* 时间复杂度:  $O(n * \log n)$  - 主要是排序的时间复杂度
- \* 空间复杂度:  $O(1)$  - 只使用了常数额外空间
- \* 是否最优解: 是，这是处理此类问题的最优解法
- \*
- \* 工程化考量:
  - \* 1. 异常处理: 检查输入是否为空
  - \* 2. 边界条件: 处理空数组、单个元素等特殊情况
  - \* 3. 性能优化: 一次遍历完成计算
  - \* 4. 可读性: 清晰的变量命名和注释
- \*
- \* 极端场景与边界场景:
  - \* 1. 空输入: arr 为空数组
  - \* 2. 极端值: 只有一个元素、所有元素相同
  - \* 3. 重复数据: 多个元素相同
  - \* 4. 有序/逆序数据: 元素按顺序排列
- \*
- \* 跨语言场景与语言特性差异:
  - \* 1. Java: 使用 Arrays.sort 进行排序
  - \* 2. C++: 使用 std::sort 进行排序
  - \* 3. Python: 使用 sorted 函数或 list.sort() 方法
- \*
- \* 调试能力构建:
  - \* 1. 打印中间过程: 在循环中打印相邻元素和差值
  - \* 2. 用断言验证中间结果: 确保差值不为负
  - \* 3. 性能退化排查: 检查排序和遍历的时间复杂度
- \*
- \* 与机器学习、图像处理、自然语言处理的联系与应用:
  - \* 1. 在聚类算法中，可用于计算数据点之间的最小距离
  - \* 2. 在异常检测中，可用于识别异常值
  - \* 3. 在推荐系统中，可用于计算用户或物品之间的相似度

```
// 简单的排序函数实现（冒泡排序）
void bubbleSort(int arr[], int n) {
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                // 交换元素
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
}
```

```

        }
    }
}

// 最小绝对差主函数
int minimumAbsoluteDifference(int arr[], int arrSize) {
    // 异常处理：检查输入是否为空
    if (arr == 0 || arrSize == 0) {
        return 0;
    }

    // 边界条件：只有一个元素
    if (arrSize == 1) {
        return 0;
    }

    // 使用冒泡排序对数组排序
    bubbleSort(arr, arrSize);

    // 初始化最小绝对差为最大值
    int minDiff = 2147483647; // INT_MAX

    // 遍历相邻元素，计算差值，找出最小值
    for (int i = 1; i < arrSize; i++) {
        int diff = arr[i] - arr[i - 1];
        if (diff < minDiff) {
            minDiff = diff;
        }
    }

    return minDiff;
}

```

=====

文件: Code14\_MinimumAbsoluteDifference.java

=====

```

package class094;

import java.util.Arrays;

// 最小绝对差 (Minimum Absolute Difference)

```

```
// 给你一个整数数组，其中数组中任意两个元素之间的绝对差的最小值。
//
// 算法标签：贪心算法(Greedy Algorithm)、排序(Sorting)、相邻元素比较(Adjacent Element Comparison)
// 时间复杂度：O(n*log(n))，其中 n 是数组长度
// 空间复杂度：O(1)，仅使用常数额外空间
// 测试链接：https://www.hackerrank.com/challenges/minimum-absolute-difference-in-an-array/problem
// 相关题目：LeetCode 532. 数组中的 K-diff 数对、LeetCode 1200. 最小绝对差
// 贪心算法专题 - 差值优化问题集合
public class Code14_MinimumAbsoluteDifference {

    /*
     * 算法思路详解：
     * 1. 贪心策略核心：排序后相邻元素的差值最小，
     *   这是因为对于任意三个按序排列的元素 a≤b≤c，有|b-a|≤|c-a|
     * 2. 排序优化：通过排序预处理，将问题转化为相邻元素比较
     * 3. 差值计算：遍历相邻元素计算差值，找出最小值
     *
     * 时间复杂度分析：
     * - O(n*log(n))，其中 n 是数组长度
     * - 排序阶段：O(n*log(n))
     * - 遍历阶段：O(n)
     * 空间复杂度分析：
     * - O(1)，仅使用了常数级别的额外空间存储最小差值
     * 是否最优解：是，这是处理此类最小绝对差问题的最优解法
     *
     * 工程化最佳实践：
     * 1. 输入验证：严格检查输入参数的有效性，防止空指针异常
     * 2. 边界处理：妥善处理各种边界情况，如空数组、单元素等
     * 3. 性能优化：采用单次遍历策略，避免重复计算
     * 4. 代码可读性：使用语义明确的变量名和详尽的注释
     * 5. 数值优化：使用整数运算避免浮点数误差
     *
     * 极端场景与边界情况处理：
     * 1. 空输入场景：arr 为空数组或 null 时直接返回 0
     * 2. 单元素场景：只有一个元素时返回 0
     * 3. 相同元素场景：多个相同元素时差值为 0
     * 4. 有序序列场景：数组已排序的情况
     * 5. 极值场景：元素值差异极大的情况
     *
     * 跨语言实现差异与优化：
     * 1. Java 实现：使用 Arrays.sort 进行排序
     * 2. C++ 实现：使用 std::sort 进行排序
    
```

- \* 3. Python 实现：使用 sorted 函数进行排序
- \* 4. 内存管理：不同语言的垃圾回收机制对性能的影响
- \*
- \* 调试与测试策略：
  - \* 1. 过程可视化：在关键节点打印相邻元素和差值
  - \* 2. 断言验证：在每次计算后添加断言确保差值非负
  - \* 3. 性能监控：跟踪排序和遍历的实际执行时间
  - \* 4. 边界测试：设计覆盖所有边界条件的测试用例
  - \* 5. 压力测试：使用大规模数据验证算法稳定性
- \*
- \* 实际应用场景与拓展：
  - \* 1. 数据分析：识别数据中的最小变化
  - \* 2. 信号处理：检测信号的最小波动
  - \* 3. 金融分析：识别价格的最小变化
  - \* 4. 质量控制：检测产品参数的最小偏差
  - \* 5. 科学计算：计算实验数据的最小误差
- \*
- \* 算法深入解析：
  - \* 1. 贪心策略原理：利用排序后相邻元素差值最小的性质
  - \* 2. 最优性证明：通过反证法可以证明贪心策略的正确性
  - \* 3. 数学基础： $|a-b| \geq 0$ , 当且仅当  $a=b$  时等号成立
  - \* 4. 策略变体：可扩展为找前 K 小的绝对差等问题
- \*/

```
public static int minimumAbsoluteDifference(int[] arr) {  
    // 异常处理：检查输入是否为空  
    if (arr == null || arr.length == 0) {  
        return 0;  
    }  
  
    // 边界条件：只有一个元素  
    if (arr.length == 1) {  
        return 0;  
    }  
  
    // 排序数组  
    Arrays.sort(arr);  
  
    // 初始化最小绝对差为最大值  
    int minDiff = Integer.MAX_VALUE;  
  
    // 遍历相邻元素，计算差值，找出最小值  
    for (int i = 1; i < arr.length; i++) {  
        int diff = arr[i] - arr[i - 1];  
        if (diff < minDiff) {  
            minDiff = diff;  
        }  
    }  
    return minDiff;  
}
```

```

        if (diff < minDiff) {
            minDiff = diff;
        }

    }

    return minDiff;
}

// 测试函数
public static void main(String[] args) {
    // 测试用例 1
    int[] arr1 = {3, -7, 0};
    System.out.println("测试用例 1 结果: " + minimumAbsoluteDifference(arr1)); // 期望输出: 3

    // 测试用例 2
    int[] arr2 = {-59, -36, -13, 1, -53, -92, -2, -96, -54, 75};
    System.out.println("测试用例 2 结果: " + minimumAbsoluteDifference(arr2)); // 期望输出: 1

    // 测试用例 3
    int[] arr3 = {1, -3, 71, 68, 17};
    System.out.println("测试用例 3 结果: " + minimumAbsoluteDifference(arr3)); // 期望输出: 3

    // 测试用例 4: 边界情况
    int[] arr4 = {5};
    System.out.println("测试用例 4 结果: " + minimumAbsoluteDifference(arr4)); // 期望输出: 0

    // 测试用例 5: 相同元素
    int[] arr5 = {1, 1, 1, 1};
    System.out.println("测试用例 5 结果: " + minimumAbsoluteDifference(arr5)); // 期望输出: 0
}
}
=====
```

文件: Code14\_MinimumAbsoluteDifference.py

```

=====
# 最小绝对差 (Minimum Absolute Difference)
# 给你一个整数数组，其中数组中任意两个元素之间的绝对差的最小值。
#
# 算法标签: 贪心算法(Greedy Algorithm)、排序(Sorting)
# 时间复杂度: O(n * logn)，其中 n 是数组长度
# 空间复杂度: O(1)，仅使用常数额外空间
# 测试链接 : https://www.hackerrank.com/challenges/minimum-absolute-difference-in-an-
```

```
array/problem  
# 相关题目: LeetCode 532. 数组中的 K-diff 数对、LeetCode 1200. 最小绝对差  
# 贪心算法专题 - 排序与差值问题集合
```

"""

算法思路详解:

1. 贪心策略: 排序后相邻元素的差值最小
  - 这个策略的核心思想是经过排序后, 最小的绝对差一定出现在相邻元素之间
  - 这是因为对于任意三个元素  $a \leq b \leq c$ , 有  $|a - c| \geq |a - b|$  且  $|a - c| \geq |b - c|$
  - 因此只需要检查相邻元素的差值即可
2. 将数组排序
  - 排序是应用这个贪心策略的前提
  - 排序后可以保证元素的有序性
3. 遍历相邻元素, 计算差值, 找出最小值
  - 通过一次遍历完成所有计算
  - 只需要比较相邻元素的差值

时间复杂度分析:

- 排序时间复杂度:  $O(n * \log n)$ , 其中  $n$  是数组长度
- 遍历时间复杂度:  $O(n)$
- 总体时间复杂度:  $O(n * \log n)$

空间复杂度分析:

- 排序空间复杂度:  $O(\log n)$  (取决于排序算法实现)
- 其他变量存储空间:  $O(1)$
- 总体空间复杂度:  $O(1)$

是否最优解:

- 是, 这是处理此类问题的最优解法
- 贪心策略保证了局部最优解能导致全局最优解
- 可以通过数学证明验证其正确性

工程化最佳实践:

1. 异常处理: 检查输入是否为空或格式不正确
2. 边界条件: 处理空数组、单个元素等特殊情况
3. 性能优化: 一次遍历完成计算, 避免重复操作
4. 可读性: 清晰的变量命名和详细注释, 便于维护

极端场景与边界情况处理:

1. 空输入: arr 为空数组
2. 极端值: 只有一个元素、所有元素相同

3. 重复数据: 多个元素相同
4. 有序/逆序数据: 元素按顺序排列

跨语言实现差异与优化:

1. Java: 使用 `Arrays.sort` 进行排序, 性能稳定
2. C++: 使用 `std::sort` 进行排序, 底层实现可能更优化
3. Python: 使用 `sorted` 函数或 `list.sort()` 方法, 基于 Timsort 算法

调试与测试策略:

1. 打印中间过程: 在循环中打印相邻元素和差值
2. 用断言验证中间结果: 确保差值不为负
3. 性能退化排查: 检查排序和遍历的时间复杂度
4. 边界测试: 测试空数组、单元素等边界情况

实际应用场景与拓展:

1. 聚类算法: 用于计算数据点之间的最小距离
2. 异常检测: 用于识别异常值
3. 推荐系统: 用于计算用户或物品之间的相似度

算法深入解析:

贪心算法在最小绝对差问题中的应用体现了其核心思想:

1. 局部最优选择: 只检查相邻元素的差值
2. 无后效性: 当前的选择不会影响之前的状态
3. 最优子结构: 问题的最优解包含子问题的最优解

这个问题的关键洞察是, 排序后最小绝对差一定出现在相邻元素之间, 这可以通过三角不等式证明。

```
def minimumAbsoluteDifference(arr):
```

```
    """
```

最小绝对差主函数 - 使用贪心算法计算数组中任意两个元素之间的绝对差的最小值

算法思路:

1. 贪心策略: 排序后相邻元素的差值最小
2. 将数组排序
3. 遍历相邻元素, 计算差值, 找出最小值

Args:

arr (`List[int]`): 整数数组

arr[i] 表示数组中的第 i 个元素

Returns:

int: 数组中任意两个元素之间的绝对差的最小值

时间复杂度:  $O(n * \log n)$ , 其中  $n$  是数组长度

空间复杂度:  $O(1)$ , 仅使用常数额外空间

Examples:

```
>>> minimumAbsoluteDifference([3, -7, 0])
```

```
3
```

```
>>> minimumAbsoluteDifference([-59, -36, -13, 1, -53, -92, -2, -96, -54, 75])
```

```
1
```

```
"""
```

# 异常处理: 检查输入是否为空

```
if not arr:
```

```
    return 0
```

# 边界条件: 只有一个元素

```
if len(arr) == 1:
```

```
    return 0
```

# 排序数组

# 时间复杂度:  $O(n * \log n)$

```
arr.sort()
```

# 初始化最小绝对差为最大值

```
min_diff = float('inf')
```

# 遍历相邻元素, 计算差值, 找出最小值

# 时间复杂度:  $O(n)$

```
for i in range(1, len(arr)):
```

```
    diff = arr[i] - arr[i - 1]
```

```
    if diff < min_diff:
```

```
        min_diff = diff
```

```
return min_diff
```

# 测试函数

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

# 测试用例 1: 一般情况

```
arr1 = [3, -7, 0]
```

```
print("测试用例 1 结果:", minimumAbsoluteDifference(arr1)) # 期望输出: 3
```

# 测试用例 2: 负数数组

```
arr2 = [-59, -36, -13, 1, -53, -92, -2, -96, -54, 75]
```

```

print("测试用例 2 结果:", minimumAbsoluteDifference(arr2)) # 期望输出: 1

# 测试用例 3: 正数数组
arr3 = [1, -3, 71, 68, 17]
print("测试用例 3 结果:", minimumAbsoluteDifference(arr3)) # 期望输出: 3

# 测试用例 4: 边界情况 - 只有一个元素
arr4 = [5]
print("测试用例 4 结果:", minimumAbsoluteDifference(arr4)) # 期望输出: 0

# 测试用例 5: 相同元素 - 最小差值为 0
arr5 = [1, 1, 1, 1]
print("测试用例 5 结果:", minimumAbsoluteDifference(arr5)) # 期望输出: 0

```

---

文件: Code15\_SouvenirGroupingNC.cpp

---

```

// 纪念品分组 (牛客网版本)
// 与洛谷版本类似, 但可能输入输出格式略有不同。
// 测试链接 : https://ac.nowcoder.com/acm/problem/16722

```

```

/*
 * 算法思路:
 * 1. 贪心策略: 排序后使用双指针, 最小价格和最大价格配对
 * 2. 将纪念品价格数组排序
 * 3. 使用双指针, 左指针指向最小价格, 右指针指向最大价格
 * 4. 如果两件纪念品价格之和不超过上限, 则分为一组, 两个指针都移动
 * 5. 如果超过上限, 则最大价格的纪念品单独分为一组, 只移动右指针
 *
 * 时间复杂度: O(n * logn) - 主要是排序的时间复杂度
 * 空间复杂度: O(1) - 只使用了常数额外空间
 * 是否最优解: 是, 这是处理此类问题的最优解法
 *
 * 工程化考量:
 * 1. 异常处理: 检查输入是否为空
 * 2. 边界条件: 处理空数组、单个元素等特殊情况
 * 3. 性能优化: 使用双指针避免重复遍历
 * 4. 可读性: 清晰的变量命名和注释
 *
 * 极端场景与边界场景:
 * 1. 空输入: prices 为空数组
 * 2. 极端值: 只有一件纪念品、所有纪念品价格相同

```

- \* 3. 重复数据：多件纪念品价格相同
- \* 4. 有序/逆序数据：纪念品价格按顺序排列
- \*
- \* 跨语言场景与语言特性差异：
- \* 1. Java：使用 Arrays.sort 进行排序
- \* 2. C++：使用 std::sort 进行排序
- \* 3. Python：使用 sorted 函数或 list.sort() 方法
- \*
- \* 调试能力构建：
- \* 1. 打印中间过程：在循环中打印当前配对的纪念品和分组情况
- \* 2. 用断言验证中间结果：确保每组价格之和不超过上限
- \* 3. 性能退化排查：检查排序和遍历的时间复杂度
- \*
- \* 与机器学习、图像处理、自然语言处理的联系与应用：
- \* 1. 在资源分配问题中，贪心算法可用于优化分组策略
- \* 2. 在推荐系统中，可用于物品配对推荐
- \* 3. 在数据挖掘中，可用于相似物品聚类

\*/

```
// 简单的排序函数实现（冒泡排序）
void bubbleSort(int arr[], int n) {
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                // 交换元素
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
}
```

```
// 纪念品分组主函数
int souvenirGrouping(int w, int prices[], int pricesSize) {
    // 异常处理：检查输入是否为空
    if (prices == 0 || pricesSize == 0) {
        return 0;
    }
}
```

```
// 边界条件：只有一件纪念品
if (pricesSize == 1) {
    return 1;
```

```
}
```

```
// 使用冒泡排序对纪念品价格数组排序
bubbleSort(prices, pricesSize);

int left = 0;           // 左指针，指向最小价格
int right = pricesSize - 1; // 右指针，指向最大价格
int groups = 0;         // 分组数

// 双指针遍历
while (left <= right) {
    // 如果两件纪念品价格之和不超过上限
    if (prices[left] + prices[right] <= w) {
        left++; // 最小价格纪念品被分组
    }
    // 最大价格纪念品被分组（无论是否与最小价格纪念品配对）
    right--;
    groups++; // 分组数加 1
}

return groups;
}
```

---

文件: Code15\_SouvenirGroupingNC.java

---

```
package class094;

import java.util.Arrays;

// 纪念品分组（牛客网版本）(Souvenir Grouping – NowCoder Version)
// 与洛谷版本类似，但可能输入输出格式略有不同。
//
// 算法标签：贪心算法(Greedy Algorithm)、双指针(Double Pointers)、排序(Sorting)
// 时间复杂度: O(n*log(n))，其中 n 是纪念品数量
// 空间复杂度: O(1)，仅使用常数额外空间
// 测试链接 : https://ac.nowcoder.com/acm/problem/16722
// 相关题目: 洛谷 P1094 纪念品分组、LeetCode 11. 盛最多水的容器
// 贪心算法专题 - 配对与分组问题集合
public class Code15_SouvenirGroupingNC {

    /*
```

- \* 算法思路详解:

- \* 1. 贪心策略核心: 排序后使用双指针, 将最小价格和最大价格的纪念品配对,

- \* 这样可以最大化每组的利用率, 从而最小化分组数

- \* 2. 排序优化: 通过升序排序, 为双指针策略奠定基础

- \* 3. 配对机制: 使用双指针分别指向当前最小和最大价格的纪念品

- \* 4. 分组决策:

- 若两件纪念品价格之和不超过上限则配对

- 否则最大价格纪念品单独分组

- \*

- \* 时间复杂度分析:

- \* -  $O(n \log n)$ , 其中 n 是纪念品数量

- \* - 排序阶段:  $O(n \log n)$

- \* - 双指针遍历:  $O(n)$

- \* 空间复杂度分析:

- \* -  $O(1)$ , 仅使用了常数级别的额外空间存储指针和计数器

- \* 是否最优解: 是, 这是处理此类配对分组问题的最优解法

- \*

- \* 工程化最佳实践:

- \* 1. 输入验证: 严格检查输入参数的有效性, 防止空指针异常

- \* 2. 边界处理: 妥善处理各种边界情况, 如空数组、单元素等

- \* 3. 性能优化: 采用双指针策略避免重复遍历

- \* 4. 代码可读性: 使用语义明确的变量名和详尽的注释

- \* 5. 条件判断优化: 合并边界条件判断

- \*

- \* 极端场景与边界情况处理:

- \* 1. 空输入场景: prices 为空数组或 null 时直接返回 0

- \* 2. 单物品场景: 只有一件纪念品时返回 1

- \* 3. 相同价格场景: 多件纪念品价格相同时的处理

- \* 4. 有序序列场景: 纪念品价格已排序的情况

- \* 5. 极值场景: 价格差异极大的情况

- \*

- \* 跨语言实现差异与优化:

- \* 1. Java 实现: 使用 Arrays.sort 和双指针遍历

- \* 2. C++ 实现: 使用 std::sort 和数组索引操作

- \* 3. Python 实现: 使用 sorted 函数和列表索引

- \* 4. 内存管理: 不同语言的垃圾回收机制对性能的影响

- \*

- \* 调试与测试策略:

- \* 1. 过程可视化: 在关键节点打印当前配对的纪念品和分组情况

- \* 2. 断言验证: 在每次配对后添加断言确保价格和不超过上限

- \* 3. 性能监控: 跟踪排序和遍历的实际执行时间

- \* 4. 边界测试: 设计覆盖所有边界条件的测试用例

- \* 5. 压力测试: 使用大规模数据验证算法稳定性

```

*
* 实际应用场景与拓展:
* 1. 资源分配: 服务器资源配对分配
* 2. 任务调度: 任务配对执行优化
* 3. 物流配送: 货物配载优化
* 4. 电商推荐: 商品组合推荐
* 5. 金融投资: 资产配对投资
*
* 算法深入解析:
* 1. 贪心策略原理: 通过最小值与最大值配对实现整体最优
* 2. 最优性证明: 使用交换论证法可以证明贪心策略的正确性
* 3. 策略变体: 可扩展为多物品分组等变体问题
* 4. 问题转换: 最少分组数 =  $(n+1)/2$  到 n 之间的最优值
*/
public static int souvenirGrouping(int w, int[] prices) {
    // 异常处理: 检查输入是否为空
    if (prices == null || prices.length == 0) {
        return 0;
    }

    // 边界条件: 只有一件纪念品
    if (prices.length == 1) {
        return 1;
    }

    // 排序纪念品价格数组
    Arrays.sort(prices);

    int left = 0;           // 左指针, 指向最小价格
    int right = prices.length - 1; // 右指针, 指向最大价格
    int groups = 0;         // 分组数

    // 双指针遍历
    while (left <= right) {
        // 如果两件纪念品价格之和不超过上限
        if (prices[left] + prices[right] <= w) {
            left++; // 最小价格纪念品被分组
        }
        // 最大价格纪念品被分组 (无论是否与最小价格纪念品配对)
        right--;
        groups++; // 分组数加 1
    }
}

```

```

    return groups;
}

// 测试函数
public static void main(String[] args) {
    // 测试用例 1
    int w1 = 10;
    int[] prices1 = {1, 2, 3, 4, 5, 6, 7, 8, 9};
    System.out.println("测试用例 1 结果: " + souvenirGrouping(w1, prices1)); // 期望输出: 5

    // 测试用例 2
    int w2 = 5;
    int[] prices2 = {1, 2, 3, 4, 5};
    System.out.println("测试用例 2 结果: " + souvenirGrouping(w2, prices2)); // 期望输出: 3

    // 测试用例 3
    int w3 = 100;
    int[] prices3 = {50, 50, 50, 50};
    System.out.println("测试用例 3 结果: " + souvenirGrouping(w3, prices3)); // 期望输出: 2

    // 测试用例 4: 边界情况
    int w4 = 10;
    int[] prices4 = {5};
    System.out.println("测试用例 4 结果: " + souvenirGrouping(w4, prices4)); // 期望输出: 1

    // 测试用例 5: 极端情况
    int w5 = 15;
    int[] prices5 = {1, 1, 1, 1, 1, 10, 10, 10, 10};
    System.out.println("测试用例 5 结果: " + souvenirGrouping(w5, prices5)); // 期望输出: 6
}
}
=====
```

文件: Code15\_SouvenirGroupingNC.py

```
=====
# 纪念品分组 (牛客网版本) (Souvenir Grouping - NowCoder Version)
# 与洛谷版本类似, 但可能输入输出格式略有不同。
#
# 算法标签: 贪心算法(Greedy Algorithm)、双指针(Two Pointers)、排序(Sorting)
# 时间复杂度: O(n * logn), 其中 n 是纪念品数量
# 空间复杂度: O(1), 仅使用常数额外空间
# 测试链接 : https://ac.nowcoder.com/acm/problem/16722
```

```
# 相关题目: LeetCode 11. 盛最多水的容器、LeetCode 167. 两数之和 II  
# 贪心算法专题 - 双指针与配对问题集合
```

"""

算法思路详解:

1. 贪心策略: 排序后使用双指针, 最小价格和最大价格配对
  - 这个策略的核心思想是让价格差异最大的纪念品配对
  - 如果最便宜和最贵的纪念品能配对, 那么其他配对方案都不会更优
  - 这样可以最大化每组的利用效率, 从而最小化分组数
2. 将纪念品价格数组排序
  - 排序是应用双指针技术的前提
  - 排序后可以使用左指针指向最小价格, 右指针指向最大价格
3. 使用双指针, 左指针指向最小价格, 右指针指向最大价格
  - 左指针从数组开始向右移动
  - 右指针从数组末尾向左移动
  - 双指针技术能高效地处理配对问题
4. 如果两件纪念品价格之和不超过上限, 则分为一组, 两个指针都移动
  - 这体现了贪心策略: 尽可能让两个纪念品配对
  - 配对成功后, 两个指针都向中间移动
5. 如果超过上限, 则最大价格的纪念品单独分为一组, 只移动右指针
  - 最大价格的纪念品无法与任何其他纪念品配对
  - 只能单独分为一组, 右指针向左移动

时间复杂度分析:

- 排序时间复杂度:  $O(n * \log n)$ , 其中  $n$  是纪念品数量
- 双指针遍历时间复杂度:  $O(n)$
- 总体时间复杂度:  $O(n * \log n)$

空间复杂度分析:

- 排序空间复杂度:  $O(\log n)$  (取决于排序算法实现)
- 其他变量存储空间:  $O(1)$
- 总体空间复杂度:  $O(1)$

是否最优解:

- 是, 这是处理此类问题的最优解法
- 贪心策略保证了局部最优解能导致全局最优解
- 可以通过交换论证法证明其正确性

工程化最佳实践:

1. 异常处理：检查输入是否为空或格式不正确
2. 边界条件：处理空数组、单个元素等特殊情况
3. 性能优化：使用双指针避免重复遍历
4. 可读性：清晰的变量命名和详细注释，便于维护

极端场景与边界情况处理：

1. 空输入：prices 为空数组
2. 极端值：只有一件纪念品、所有纪念品价格相同
3. 重复数据：多件纪念品价格相同
4. 有序/逆序数据：纪念品价格按顺序排列

跨语言实现差异与优化：

1. Java：使用 Arrays.sort 进行排序，性能稳定
2. C++：使用 std::sort 进行排序，底层实现可能更优化
3. Python：使用 sorted 函数或 list.sort() 方法，基于 Timsort 算法

调试与测试策略：

1. 打印中间过程：在循环中打印当前配对的纪念品和分组情况
2. 用断言验证中间结果：确保每组价格之和不超过上限
3. 性能退化排查：检查排序和遍历的时间复杂度
4. 边界测试：测试空数组、单元素等边界情况

实际应用场景与拓展：

1. 资源分配问题：在资源分配中优化分组策略
2. 推荐系统：用于物品配对推荐
3. 数据挖掘：用于相似物品聚类

算法深入解析：

贪心算法在纪念品分组问题中的应用体现了其核心思想：

1. 局部最优选择：每次选择价格差异最大的纪念品配对
2. 无后效性：当前的选择不会影响之前的状态
3. 最优子结构：问题的最优解包含子问题的最优解

这个问题的关键洞察是，最小价格和最大价格的配对策略能最小化分组数，这可以通过数学证明验证。

"""

```
def souvenirGrouping(w, prices):
```

```
    """
```

```
    纪念品分组主函数 - 使用贪心算法和双指针技术计算最少分组数目
```

算法思路：

1. 贪心策略：排序后使用双指针，最小价格和最大价格配对
2. 将纪念品价格数组排序

### 3. 使用双指针配对纪念品

Args:

w (int): 每组纪念品价格之和的上限  
prices (List[int]): 纪念品价格列表  
prices[i] 表示第 i 件纪念品的价格

Returns:

int: 最少的分组数目

时间复杂度:  $O(n * \log n)$ , 其中 n 是纪念品数量

空间复杂度:  $O(1)$ , 仅使用常数额外空间

Examples:

```
>>> souvenirGrouping(10, [1, 2, 3, 4, 5, 6, 7, 8, 9])  
5
```

```
>>> souvenirGrouping(5, [1, 2, 3, 4, 5])  
3
```

"""

# 异常处理: 检查输入是否为空

```
if not prices:  
    return 0
```

# 边界条件: 只有一件纪念品

```
if len(prices) == 1:  
    return 1
```

# 排序纪念品价格数组

# 时间复杂度:  $O(n * \log n)$   
prices.sort()

```
left = 0          # 左指针, 指向最小价格  
right = len(prices) - 1 # 右指针, 指向最大价格  
groups = 0        # 分组数
```

# 双指针遍历

# 时间复杂度:  $O(n)$

```
while left <= right:  
    # 如果两件纪念品价格之和不超过上限  
    if prices[left] + prices[right] <= w:  
        left += 1      # 最小价格纪念品被分组
```

# 最大价格纪念品被分组 (无论是否与最小价格纪念品配对)

```

right -= 1
groups += 1      # 分组数加 1

return groups

# 测试函数
if __name__ == "__main__":
    # 测试用例 1: 一般情况
    w1 = 10
    prices1 = [1, 2, 3, 4, 5, 6, 7, 8, 9]
    print("测试用例 1 结果:", souvenirGrouping(w1, prices1)) # 期望输出: 5

    # 测试用例 2: 较小的上限
    w2 = 5
    prices2 = [1, 2, 3, 4, 5]
    print("测试用例 2 结果:", souvenirGrouping(w2, prices2)) # 期望输出: 3

    # 测试用例 3: 相同价格的纪念品
    w3 = 100
    prices3 = [50, 50, 50, 50]
    print("测试用例 3 结果:", souvenirGrouping(w3, prices3)) # 期望输出: 2

    # 测试用例 4: 边界情况 - 只有一件纪念品
    w4 = 10
    prices4 = [5]
    print("测试用例 4 结果:", souvenirGrouping(w4, prices4)) # 期望输出: 1

    # 测试用例 5: 极端情况 - 价格差异很大
    w5 = 15
    prices5 = [1, 1, 1, 1, 1, 10, 10, 10, 10]
    print("测试用例 5 结果:", souvenirGrouping(w5, prices5)) # 期望输出: 6

```

=====

文件: Code16\_GreedyAdditionalProblems.cpp

=====

```

#include <iostream>
#include <vector>
#include <algorithm>
#include <string>
#include <deque>
#include <functional>

```

```

#include <cmath>

using namespace std;

/***
 * 贪心算法补充题目集合 - C++版本
 * 收集来自各大算法平台的经典贪心算法题目
 * 每个题目都提供详细的注释和复杂度分析
 */

/***
 * 题目 1: LeetCode 134. 加油站
 * 算法思路: 贪心策略, 维护当前剩余油量和总剩余油量
 * 时间复杂度: O(n), 空间复杂度: O(1)
 */
int canCompleteCircuit(vector<int>& gas, vector<int>& cost) {
    if (gas.size() != cost.size()) return -1;

    int totalGas = 0;
    int currentGas = 0;
    int startStation = 0;

    for (int i = 0; i < gas.size(); i++) {
        totalGas += gas[i] - cost[i];
        currentGas += gas[i] - cost[i];

        if (currentGas < 0) {
            startStation = i + 1;
            currentGas = 0;
        }
    }

    return totalGas >= 0 ? startStation : -1;
}

/***
 * 题目 2: LeetCode 402. 移掉 K 位数字
 * 算法思路: 使用单调栈保持数字序列递增
 * 时间复杂度: O(n), 空间复杂度: O(n)
 */
string removeKdigits(string num, int k) {
    if (num.length() <= k) return "0";

```

```

deque<char> stack;

for (char digit : num) {
    while (!stack.empty() && k > 0 && digit < stack.back()) {
        stack.pop_back();
        k--;
    }
    stack.push_back(digit);
}

// 处理剩余的删除次数
while (k > 0) {
    stack.pop_back();
    k--;
}

// 构建结果字符串
string result;
while (!stack.empty()) {
    result.push_back(stack.front());
    stack.pop_front();
}

// 去除前导零
int start = 0;
while (start < result.length() && result[start] == '0') {
    start++;
}

return start == result.length() ? "0" : result.substr(start);
}

/***
 * 题目 3: LeetCode 561. 数组拆分 I
 * 算法思路: 排序后取每对的第一个元素
 * 时间复杂度: O(n log n), 空间复杂度: O(1)
 */
int arrayPairSum(vector<int>& nums) {
    if (nums.size() % 2 != 0) return 0;

    sort(nums.begin(), nums.end());
    int sum = 0;

```

```
for (int i = 0; i < nums.size(); i += 2) {  
    sum += nums[i];  
}  
  
return sum;  
}
```

```
/**  
 * 题目 4: LeetCode 665. 非递减数列  
 * 算法思路: 遇到逆序对时优先修改前面的数字  
 * 时间复杂度: O(n), 空间复杂度: O(1)  
 */
```

```
bool checkPossibility(vector<int>& nums) {  
    if (nums.size() <= 1) return true;  
  
    int count = 0;  
    for (int i = 0; i < nums.size() - 1; i++) {  
        if (nums[i] > nums[i + 1]) {  
            count++;  
            if (count > 1) return false;  
  
            if (i > 0 && nums[i - 1] > nums[i + 1]) {  
                nums[i + 1] = nums[i];  
            } else {  
                nums[i] = nums[i + 1];  
            }  
        }  
    }  
}  
  
return true;  
}
```

```
/**  
 * 题目 5: HackerRank - Mark and Toys  
 * 算法思路: 排序价格后贪心购买  
 * 时间复杂度: O(n log n), 空间复杂度: O(1)  
 */
```

```
int maximumToys(vector<int>& prices, int budget) {  
    if (prices.empty() || budget <= 0) return 0;  
  
    sort(prices.begin(), prices.end());  
    int count = 0;  
    int currentCost = 0;
```

```

for (int price : prices) {
    if (currentCost + price <= budget) {
        currentCost += price;
        count++;
    } else {
        break;
    }
}

return count;
}

```

```

/**
 * 题目 6: HackerRank - Luck Balance
 * 算法思路: 输掉不重要比赛, 输掉 k 场重要比赛
 * 时间复杂度: O(n log n), 空间复杂度: O(n)
 */

```

```

int luckBalance(int k, vector<vector<int>>& contests) {
    if (contests.empty()) return 0;

    vector<int> important;
    int totalLuck = 0;

    for (auto& contest : contests) {
        int luck = contest[0];
        int importance = contest[1];

        if (importance == 1) {
            important.push_back(luck);
        } else {
            totalLuck += luck;
        }
    }
}

```

```

// 重要比赛按运气值降序排序
sort(important.begin(), important.end(), greater<int>());

```

```

// 输掉前 k 场重要比赛
for (int i = 0; i < important.size(); i++) {
    if (i < k) {
        totalLuck += important[i];
    } else {

```

```

        totalLuck -= important[i];
    }
}

return totalLuck;
}

/***
 * 题目 7: 牛客网 - 疯狂的采药 (分数背包贪心解法)
 * 算法思路: 按性价比排序后贪心选择
 * 时间复杂度: O(n log n), 空间复杂度: O(n)
 */
int crazyHerbs(int time, vector<vector<int>>& herbs) {
    if (herbs.empty() || time <= 0) return 0;

    // 计算性价比并排序
    vector<tuple<double, int, int>> herbList; // ratio, value, cost
    for (auto& herb : herbs) {
        int value = herb[0];
        int cost = herb[1];
        double ratio = static_cast<double>(value) / cost;
        herbList.push_back({ratio, value, cost});
    }

    sort(herbList.begin(), herbList.end(), [] (const auto& a, const auto& b) {
        return get<0>(a) > get<0>(b);
    });

    int totalValue = 0;
    int remainingTime = time;

    for (auto& herb : herbList) {
        int value = get<1>(herb);
        int cost = get<2>(herb);

        if (remainingTime >= cost) {
            totalValue += value;
            remainingTime -= cost;
        } else {
            totalValue += static_cast<int>(value * (static_cast<double>(remainingTime) / cost));
            break;
        }
    }
}

```

```

    return totalValue;
}

/***
 * 题目 8: Codeforces 1360B - Honest Coach
 * 算法思路: 排序后找最小相邻差值
 * 时间复杂度: O(n log n), 空间复杂度: O(1)
 */
int honestCoach(vector<int>& skills) {
    if (skills.size() <= 1) return 0;

    sort(skills.begin(), skills.end());
    int minDiff = INT_MAX;

    for (int i = 1; i < skills.size(); i++) {
        minDiff = min(minDiff, skills[i] - skills[i - 1]);
    }

    return minDiff;
}

/***
 * 题目 9: 洛谷 P1803 - 线段覆盖
 * 算法思路: 经典活动选择问题
 * 时间复杂度: O(n log n), 空间复杂度: O(1)
 */
int maxNonOverlappingIntervals(vector<vector<int>>& intervals) {
    if (intervals.empty()) return 0;

    // 按结束时间排序
    sort(intervals.begin(), intervals.end(), [] (const vector<int>& a, const vector<int>& b) {
        return a[1] < b[1];
    });

    int count = 1;
    int end = intervals[0][1];

    for (int i = 1; i < intervals.size(); i++) {
        if (intervals[i][0] >= end) {
            count++;
            end = intervals[i][1];
        }
    }
}
```

```
}

return count;
}

/***
 * 题目 10: LeetCode 1005. K 次取反后最大化的数组和
 * 算法思路: 优先处理负数, 然后处理最小正数
 * 时间复杂度: O(n log n), 空间复杂度: O(1)
 */

int largestSumAfterKNegations(vector<int>& nums, int k) {
    if (nums.empty()) return 0;

    // 排序处理负数
    sort(nums.begin(), nums.end());

    for (int i = 0; i < nums.size() && k > 0; i++) {
        if (nums[i] < 0) {
            nums[i] = -nums[i];
            k--;
        } else {
            break;
        }
    }

    // 处理剩余的 k
    if (k > 0 && k % 2 == 1) {
        sort(nums.begin(), nums.end());
        nums[0] = -nums[0];
    }

    int sum = 0;
    for (int num : nums) {
        sum += num;
    }

    return sum;
}

// 测试函数
int main() {
    // 测试加油站问题
    vector<int> gas = {1, 2, 3, 4, 5};
```

```

vector<int> cost = {3, 4, 5, 1, 2};
cout << "加油站测试: " << canCompleteCircuit(gas, cost) << endl; // 期望: 3

// 测试移掉 K 位数字
cout << "移掉 K 位数字测试: " << removeKdigits("1432219", 3) << endl; // 期望: "1219"

// 测试数组拆分
vector<int> nums = {1, 4, 3, 2};
cout << "数组拆分测试: " << arrayPairSum(nums) << endl; // 期望: 4

return 0;
}

```

=====

文件: Code16\_GreedyAdditionalProblems.java

=====

```

package class094;

import java.util.*;

/**
 * 贪心算法补充题目集合 (Greedy Algorithm Additional Problems Collection)
 * 收集来自 LeetCode、HackerRank、牛客网、洛谷、Codeforces 等平台的经典贪心算法题目
 * 每个题目都提供详细的注释、复杂度分析和工程化考量
 *
 * 算法标签: 贪心算法(Greedy Algorithm)、多平台题目集合(Multi-platform Problem Collection)
 * 时间复杂度: 各题目不同, 详见具体函数注释
 * 空间复杂度: 各题目不同, 详见具体函数注释
 * 相关题目: 涵盖加油站、数字处理、数组分组、序列调整、背包问题等多个领域
 * 贪心算法专题 - 综合题目集合
 */
public class Code16_GreedyAdditionalProblems {

    /**
     * 题目 1: LeetCode 134. 加油站 (Gas Station)
     * 题目描述: 在一条环路上有 n 个加油站, 其中第 i 个加油站有汽油 gas[i] 升。
     * 你有一辆油箱容量无限的汽车, 从第 i 个加油站开往第 i+1 个加油站需要消耗汽油 cost[i] 升。
     * 你从其中的一个加油站出发, 开始时油箱为空。
     * 如果你可以绕环路行驶一周, 则返回出发时加油站的编号, 否则返回 -1。
     * 链接: https://leetcode.cn/problems/gas-station/
     *
     * 算法标签: 贪心算法(Greedy Algorithm)、环路遍历(Circular Traversal)、状态维护(State

```

Maintenance)

\* 时间复杂度:  $O(n)$  – 一次遍历

\* 空间复杂度:  $O(1)$  – 常数空间

\* 是否最优解: 是

\*

\* 算法思路详解:

\* 1. 贪心策略核心: 如果从 A 到 B 的路上没油了, 那么 A 到 B 之间的任何一个站点都不能作为起点,

\* 这是因为从 A 到 B 之间任何一点出发都会经过 A 到 B 这段油量不足的路径

\* 2. 状态维护: 同时维护当前剩余油量和总剩余油量两个状态

\* 3. 可行性判断: 如果总剩余油量为负, 说明无论从哪一点出发都无法绕行一周

\*

\* 工程化最佳实践:

\* 1. 输入验证: 严格检查输入参数的有效性

\* 2. 边界处理: 妥善处理空数组和长度不匹配的情况

\* 3. 状态一致性: 确保当前油量和总油量状态的一致性

\*

\* 实际应用场景:

\* 1. 物流配送: 配送车辆路线规划

\* 2. 无人机巡检: 无人机续航路径规划

\* 3. 游戏开发: 角色移动路径规划

\*/

```
public static int canCompleteCircuit(int[] gas, int[] cost) {  
    if (gas == null || cost == null || gas.length != cost.length) {  
        return -1;  
    }  
  
    int totalGas = 0;      // 总剩余油量  
    int currentGas = 0;    // 当前剩余油量  
    int startStation = 0;  // 起始加油站  
  
    for (int i = 0; i < gas.length; i++) {  
        totalGas += gas[i] - cost[i];  
        currentGas += gas[i] - cost[i];  
  
        // 如果当前剩余油量为负, 说明从 startStation 到 i 的路径不可行  
        if (currentGas < 0) {  
            startStation = i + 1; // 从下一个站点重新开始  
            currentGas = 0;       // 重置当前剩余油量  
        }  
    }  
  
    // 如果总剩余油量为负, 无法绕行一周  
    return totalGas >= 0 ? startStation : -1;  
}
```

```
}
```

```
/**
```

```
* 题目 2: LeetCode 402. 移掉 K 位数字 (Remove K Digits)
```

```
* 题目描述: 给你一个以字符串表示的非负整数 num 和一个整数 k ,
```

```
* 移除这个数中的 k 位数字, 使得剩下的数字最小。
```

```
* 请你以字符串形式返回这个最小的数字。
```

```
* 链接: https://leetcode.cn/problems/remove-k-digits/
```

```
*
```

```
* 算法标签: 贪心算法(Greedy Algorithm)、单调栈(Monotonic Stack)、字符串处理(String Processing)
```

```
* 时间复杂度: O(n) - n 是字符串长度
```

```
* 空间复杂度: O(n) - 栈的空间
```

```
* 是否最优解: 是
```

```
*
```

```
* 算法思路详解:
```

```
* 1. 贪心策略核心: 使用单调栈维护递增序列,
```

```
* 当遇到更小数字时删除栈顶较大数字以获得更小结果
```

```
* 2. 单调性维护: 保持栈内元素单调递增
```

```
* 3. 删除策略: 从左到右遍历, 优先删除高位的大数字
```

```
*
```

```
* 工程化最佳实践:
```

```
* 1. 字符串处理: 注意前导零的处理
```

```
* 2. 边界情况: 处理删除所有数字和空结果的情况
```

```
* 3. 内存优化: 及时释放不再使用的对象
```

```
*
```

```
* 实际应用场景:
```

```
* 1. 数字优化: 生成最小数字组合
```

```
* 2. 密码学: 数字序列优化
```

```
* 3. 数据处理: 数值压缩优化
```

```
*/
```

```
public static String removeKdigits(String num, int k) {
```

```
    if (num == null || num.length() <= k) {
```

```
        return "0";
```

```
}
```

```
Deque<Character> stack = new ArrayDeque<>();
```

```
for (int i = 0; i < num.length(); i++) {
```

```
    char digit = num.charAt(i);
```

```
    // 当栈不为空, 当前数字小于栈顶数字, 且还有删除次数时, 弹出栈顶
```

```
    while (!stack.isEmpty() && k > 0 && digit < stack.peek()) {
```

```
        stack.pop();
```

```

        k--;
    }
    stack.push(digit);
}

// 处理剩余的删除次数
while (k > 0) {
    stack.pop();
    k--;
}

// 构建结果字符串
StringBuilder result = new StringBuilder();
while (!stack.isEmpty()) {
    result.append(stack.pop());
}
result.reverse();

// 去除前导零
int start = 0;
while (start < result.length() && result.charAt(start) == '0') {
    start++;
}

return start == result.length() ? "0" : result.substring(start);
}

/**
 * 题目 3: LeetCode 561. 数组拆分 I (Array Partition I)
 * 题目描述: 给定长度为  $2n$  的整数数组  $\text{nums}$ ，你的任务是将这些数分成  $n$  对，
 * 例如  $(a_1, b_1), (a_2, b_2), \dots, (a_n, b_n)$ ，使得从 1 到  $n$  的  $\min(a_i, b_i)$  总和最大。
 * 返回该 最大总和 。
 * 链接: https://leetcode.cn/problems/array-partition-i/
 *
 * 算法标签: 贪心算法(Greedy Algorithm)、排序(Sorting)、配对优化(Pairing Optimization)
 * 时间复杂度:  $O(n \log n)$  – 排序的时间复杂度
 * 空间复杂度:  $O(1)$  – 常数空间
 * 是否最优解: 是
 *
 * 算法思路详解:
 * 1. 贪心策略核心: 排序后相邻两个数分为一组, 取较小的那个,
 *    这样能保证每组的最小值尽可能大, 从而最大化总和
 * 2. 排序优化: 通过排序预处理, 将问题转化为确定性最优解

```

```

* 3. 配对机制：相邻元素配对确保最小值最大化
*
* 工程化最佳实践：
* 1. 输入验证：检查数组长度是否为偶数
* 2. 边界处理：妥善处理空数组情况
* 3. 索引优化：使用步长为 2 的循环提高效率
*
* 实际应用场景：
* 1. 资源分配：最优配对资源分配
* 2. 团队组建：能力值最优配对
* 3. 任务分组：任务难度最优匹配
*/
public static int arrayPairSum(int[] nums) {
    if (nums == null || nums.length % 2 != 0) {
        return 0;
    }

    Arrays.sort(nums);
    int sum = 0;

    // 每隔一个元素取一个（即每对中的第一个元素）
    for (int i = 0; i < nums.length; i += 2) {
        sum += nums[i];
    }

    return sum;
}

/***
* 题目 4: LeetCode 665. 非递减数列 (Non-decreasing Array)
* 题目描述: 给你一个长度为 n 的整数数组 nums , 请你判断在 最多 改变 1 个元素的情况下,
* 该数组能否变成一个非递减数列。
* 链接: https://leetcode.cn/problems/non-decreasing-array/
*
* 算法标签: 贪心算法(Greedy Algorithm)、序列调整(Sequence Adjustment)、逆序处理(Reverse Order Processing)
* 时间复杂度: O(n) - 一次遍历
* 空间复杂度: O(1) - 常数空间
* 是否最优解: 是
*
* 算法思路详解:
* 1. 贪心策略核心: 遇到逆序对时, 优先修改前面的数字,
* 但如果修改前面数字会影响更多后续判断, 则修改后面的数字

```

```
* 2. 修改决策：通过比较 nums[i-1] 和 nums[i+1] 决定修改哪个元素
* 3. 次数控制：最多只能修改 1 个元素
*
* 工程化最佳实践：
* 1. 边界处理：妥善处理数组边界情况
* 2. 条件判断：优化修改决策条件判断
* 3. 计数管理：准确统计修改次数
*
* 实际应用场景：
* 1. 数据清洗：修正数据序列中的异常值
* 2. 信号处理：平滑信号序列中的噪声
* 3. 质量控制：调整产品参数序列
*/
```

```
public static boolean checkPossibility(int[] nums) {
    if (nums == null || nums.length <= 1) {
        return true;
    }

    int count = 0;
    for (int i = 0; i < nums.length - 1; i++) {
        if (nums[i] > nums[i + 1]) {
            count++;
            if (count > 1) {
                return false;
            }
        }
    }

    // 贪心选择：优先修改 nums[i] 而不是 nums[i+1]
    if (i > 0 && nums[i - 1] > nums[i + 1]) {
        nums[i + 1] = nums[i]; // 必须修改 nums[i+1]
    } else {
        nums[i] = nums[i + 1]; // 修改 nums[i]
    }
}

return true;
}
```

```
/**
* 题目 5: HackerRank - Mark and Toys
* 题目描述：给定一个价格数组和预算，计算最多能买多少件玩具
* 链接：https://www.hackerrank.com/challenges/mark-and-toys
*
```

\* 算法标签: 贪心算法(Greedy Algorithm)、预算优化(Budget Optimization)、排序(Sorting)

\* 时间复杂度:  $O(n \log n)$  – 排序的时间复杂度

\* 空间复杂度:  $O(1)$  – 常数空间

\* 是否最优解: 是

\*

\* 算法思路详解:

\* 1. 贪心策略核心: 优先购买价格最低的玩具,

\* 这样可以在预算内购买最多数量的玩具

\* 2. 排序优化: 通过排序预处理, 将问题转化为确定性最优解

\* 3. 购买策略: 依次购买直到预算不足

\*

\* 工程化最佳实践:

\* 1. 输入验证: 检查价格数组和预算的有效性

\* 2. 边界处理: 妥善处理空数组和零预算情况

\* 3. 成本控制: 准确计算累计成本

\*

\* 实际应用场景:

\* 1. 电商购物: 预算内商品选购优化

\* 2. 投资组合: 资金分配优化

\* 3. 资源采购: 有限预算下的采购策略

\*/

```
public static int maximumToys(int[] prices, int budget) {  
    if (prices == null || prices.length == 0 || budget <= 0) {  
        return 0;  
    }  
}
```

```
    Arrays.sort(prices);
```

```
    int count = 0;
```

```
    int currentCost = 0;
```

```
    for (int price : prices) {
```

```
        if (currentCost + price <= budget) {
```

```
            currentCost += price;
```

```
            count++;
```

```
        } else {
```

```
            break;
```

```
        }
```

```
}
```

```
    return count;
```

```
}
```

```
/**
```

\* 题目 6: HackerRank – Luck Balance

\* 题目描述: 参加比赛, 重要比赛输了会减少运气, 不重要比赛输了会增加运气

\* 最多可以输掉 k 场重要比赛, 求最大运气值

\* 链接: <https://www.hackerrank.com/challenges/luck-balance>

\*

\* 算法标签: 贪心算法(Greedy Algorithm)、策略选择(Strategy Selection)、排序(Sorting)

\* 时间复杂度:  $O(n \log n)$  – 排序的时间复杂度

\* 空间复杂度:  $O(n)$  – 存储重要比赛

\* 是否最优解: 是

\*

\* 算法思路详解:

\* 1. 贪心策略核心: 输掉所有不重要比赛 (增加运气),

\*     输掉 k 场运气值最大的重要比赛 (最大化收益)

\* 2. 分类处理: 将比赛分为重要和不重要两类分别处理

\* 3. 优先级排序: 重要比赛按运气值降序排序

\*

\* 工程化最佳实践:

\* 1. 数据分类: 有效分离重要和不重要比赛

\* 2. 排序优化: 使用高效排序算法

\* 3. 策略执行: 准确执行输赢策略

\*

\* 实际应用场景:

\* 1. 项目管理: 重要任务优先级分配

\* 2. 投资决策: 风险与收益平衡

\* 3. 竞争策略: 资源投入优化

\*/

```
public static int luckBalance(int k, int[][] contests) {  
    if (contests == null || contests.length == 0) {  
        return 0;  
    }  
}
```

```
List<Integer> important = new ArrayList<>();
```

```
int totalLuck = 0;
```

```
for (int[] contest : contests) {
```

```
    int luck = contest[0];
```

```
    int importance = contest[1];
```

```
    if (importance == 1) {
```

```
        important.add(luck);
```

```
    } else {
```

```
        totalLuck += luck; // 不重要比赛直接输掉
```

```
}
```

```

    }

    // 重要比赛按运气值降序排序
    important.sort(Collections.reverseOrder());

    // 输掉前 k 场重要比赛（运气值最大的 k 场）
    for (int i = 0; i < important.size(); i++) {
        if (i < k) {
            totalLuck += important.get(i); // 输掉比赛，增加运气
        } else {
            totalLuck -= important.get(i); // 赢得比赛，减少运气
        }
    }

    return totalLuck;
}

/**
 * 题目 7：牛客网 - 疯狂的采药 (Crazy Herbs Collection)
 * 题目描述：完全背包问题的贪心解法变种
 * 链接：https://ac.nowcoder.com/acm/problem/16557
 *
 * 算法标签：贪心算法(Greedy Algorithm)、背包问题变种(Knapsack Variant)、性价比优化(Cost-Performance Optimization)
 * 时间复杂度：O(n log n) – 排序的时间复杂度
 * 空间复杂度：O(1) – 常数空间
 * 是否最优解：对于分数背包是最优解，对于 0-1 背包是近似解
 *
 * 算法思路详解：
 * 1. 贪心策略核心：优先选择性价比（价值/时间）最高的草药，
 *    这样可以在有限时间内获得最大价值
 * 2. 排序优化：按性价比降序排序
 * 3. 采摘策略：依次采摘直到时间不足
 *
 * 工程化最佳实践：
 * 1. 性价比计算：准确计算价值与时间比
 * 2. 边界处理：处理时间不足时的部分采摘
 * 3. 精度控制：注意浮点数运算精度
 *
 * 实际应用场景：
 * 1. 资源分配：有限资源下的收益最大化
 * 2. 投资组合：投资标的性价比优化
 * 3. 任务调度：时间约束下的任务选择

```

```

*/
public static int crazyHerbs(int time, int[][] herbs) {
    if (herbs == null || herbs.length == 0 || time <= 0) {
        return 0;
    }

    // 计算每种草药的性价比
    List<double[]> herbList = new ArrayList<>();
    for (int[] herb : herbs) {
        int value = herb[0];
        int cost = herb[1];
        double ratio = (double) value / cost;
        herbList.add(new double[]{value, cost, ratio});
    }
}

// 按性价比降序排序
herbList.sort((a, b) -> Double.compare(b[2], a[2]));

int totalValue = 0;
int remainingTime = time;

for (double[] herb : herbList) {
    int value = (int) herb[0];
    int cost = (int) herb[1];

    if (remainingTime >= cost) {
        // 可以采摘完整的草药
        totalValue += value;
        remainingTime -= cost;
    } else {
        // 采摘部分草药（分数背包）
        totalValue += (int) (value * ((double) remainingTime / cost));
        break;
    }
}

return totalValue;
}

/**
* 题目 8: Codeforces 1360B - Honest Coach
* 题目描述: 将运动员分成两组, 使得两组运动员实力值的最小差值最大
* 链接: https://codeforces.com/problemset/problem/1360/B

```

```

*
* 算法标签: 贪心算法(Greedy Algorithm)、分组优化(Grouping Optimization)、排序(Sorting)
* 时间复杂度: O(n log n) - 排序的时间复杂度
* 空间复杂度: O(1) - 常数空间
* 是否最优解: 是
*
* 算法思路详解:
* 1. 贪心策略核心: 排序后相邻两个运动员的实力差最小,
*   因此最小的相邻差值就是两组实力最接近的分组
* 2. 排序优化: 通过排序预处理, 将问题转化为相邻元素比较
* 3. 差值计算: 找到最小的相邻差值
*
* 工程化最佳实践:
* 1. 输入验证: 检查运动员实力数组的有效性
* 2. 边界处理: 妥善处理少于 2 个运动员的情况
* 3. 差值优化: 使用一次遍历找到最小差值
*
* 实际应用场景:
* 1. 体育竞赛: 实力均衡的队伍分组
* 2. 教育分班: 学生能力均衡分班
* 3. 团队建设: 技能均衡的团队分配
*/
public static int honestCoach(int[] skills) {
    if (skills == null || skills.length <= 1) {
        return 0;
    }

    Arrays.sort(skills);
    int minDiff = Integer.MAX_VALUE;

    // 找到最小的相邻差值
    for (int i = 1; i < skills.length; i++) {
        minDiff = Math.min(minDiff, skills[i] - skills[i - 1]);
    }

    return minDiff;
}

/**
* 题目 9: 洛谷 P1803 - 凌乱的yyy / 线段覆盖 (Segment Coverage)
* 题目描述: 选择最多的不重叠区间 (活动选择问题)
* 链接: https://www.luogu.com.cn/problem/P1803
*

```

\* 算法标签: 贪心算法(Greedy Algorithm)、区间调度(Interval Scheduling)、活动选择(Activity Selection)

\* 时间复杂度:  $O(n \log n)$  – 排序的时间复杂度

\* 空间复杂度:  $O(1)$  – 常数空间

\* 是否最优解: 是

\*

\* 算法思路详解:

\* 1. 贪心策略核心: 按结束时间排序, 优先选择结束早的活动,

\* 这样可以为后续活动留出更多时间

\* 2. 经典应用: 活动选择问题的标准解法

\* 3. 选择机制: 遍历排序后的活动, 选择不重叠的活动

\*

\* 工程化最佳实践:

\* 1. 排序优化: 使用高效的比较器

\* 2. 边界处理: 妥善处理空数组情况

\* 3. 重叠判断: 准确判断区间重叠条件

\*

\* 实际应用场景:

\* 1. 会议调度: 会议室资源最优分配

\* 2. 任务安排: CPU 任务调度优化

\* 3. 广告投放: 广告位时间分配

\*/

```
public static int maxNonOverlappingIntervals(int[][] intervals) {
    if (intervals == null || intervals.length == 0) {
        return 0;
    }

    // 按结束时间排序
    Arrays.sort(intervals, (a, b) -> a[1] - b[1]);

    int count = 1;
    int end = intervals[0][1];

    for (int i = 1; i < intervals.length; i++) {
        if (intervals[i][0] >= end) {
            count++;
            end = intervals[i][1];
        }
    }

    return count;
}
```

```

/**
 * 题目 10: LeetCode 1005. K 次取反后最大化的数组和 (Maximize Sum After K Negations)
 * 题目描述: 给你一个整数数组 nums 和一个整数 k , 可以进行 k 次取反操作, 求最大数组和
 * 链接: https://leetcode.cn/problems/maximize-sum-of-array-after-k-negations/
 *
 * 算法标签: 贪心算法(Greedy Algorithm)、数组操作(Array Operations)、符号优化(Sign Optimization)
 *
 * 时间复杂度: O(n log n) - 排序的时间复杂度
 * 空间复杂度: O(1) - 常数空间
 * 是否最优解: 是
 *
 * 算法思路详解:
 * 1. 贪心策略核心: 优先将负数变为正数以最大化数组和,
 *    如果还有剩余次数且为奇数, 对最小的正数进行取反
 * 2. 分阶段处理: 先处理负数, 再处理剩余次数
 * 3. 符号优化: 通过取反操作最大化数组元素值
 *
 * 工程化最佳实践:
 * 1. 阶段处理: 分阶段执行不同的取反策略
 * 2. 边界优化: 处理剩余次数为奇数的情况
 * 3. 总和计算: 准确计算数组元素总和
 *
 * 实际应用场景:
 * 1. 数值优化: 数组元素符号调整
 * 2. 金融计算: 收益最大化调整
 * 3. 数据处理: 数值序列优化
 */

public static int largestSumAfterKNegations(int[] nums, int k) {
    if (nums == null || nums.length == 0) {
        return 0;
    }

    // 排序, 让负数在前面
    Arrays.sort(nums);

    // 优先将负数变为正数
    for (int i = 0; i < nums.length && k > 0; i++) {
        if (nums[i] < 0) {
            nums[i] = -nums[i];
            k--;
        } else {
            break;
        }
    }
}

```

```
}

// 如果 k 还有剩余且为奇数，对最小的数取反
if (k > 0 && k % 2 == 1) {
    Arrays.sort(nums); // 重新排序找到最小的数
    nums[0] = -nums[0];
}

// 计算总和
int sum = 0;
for (int num : nums) {
    sum += num;
}

return sum;
}

// 测试函数
public static void main(String[] args) {
    // 测试加油站问题
    int[] gas = {1, 2, 3, 4, 5};
    int[] cost = {3, 4, 5, 1, 2};
    System.out.println("加油站测试: " + canCompleteCircuit(gas, cost)); // 期望: 3

    // 测试移掉 K 位数字
    System.out.println("移掉 K 位数字测试: " + removeKdigits("1432219", 3)); // 期望: "1219"

    // 测试数组拆分
    int[] nums = {1, 4, 3, 2};
    System.out.println("数组拆分测试: " + arrayPairSum(nums)); // 期望: 4

    // 测试非递减数列
    int[] nums2 = {4, 2, 3};
    System.out.println("非递减数列测试: " + checkPossibility(nums2)); // 期望: true

    // 测试疯狂的采药 (近似解测试)
    int[][] herbs = {{60, 10}, {100, 20}, {120, 30}};
    System.out.println("疯狂的采药测试: " + crazyHerbs(50, herbs)); // 期望: 240
}
```

=====

"""

贪心算法补充题目集合 - Python 版本

收集来自各大算法平台的经典贪心算法题目

每个题目都提供详细的注释、复杂度分析和工程化考量

算法专题概览:

本文件包含 10 个经典的贪心算法问题，涵盖了不同领域的应用场景:

1. 资源分配问题（加油站、采药）
2. 字符串处理问题（移除数字）
3. 数组优化问题（数组拆分、取反操作）
4. 序列调整问题（非递减数列）
5. 预算优化问题（购买玩具）
6. 策略选择问题（运气平衡）
7. 区间调度问题（线段覆盖）
8. 差值优化问题（教练分配）

贪心算法核心思想:

贪心算法在每个决策点都做出局部最优选择，希望通过一系列局部最优选择达到全局最优解。

适用条件:

1. 贪心选择性质：局部最优选择能导致全局最优解
2. 最优子结构：问题的最优解包含子问题的最优解

工程化最佳实践:

1. 异常处理：检查输入参数的有效性
2. 边界条件：处理空输入、单元素等特殊情况
3. 性能优化：选择合适的数据结构和算法策略
4. 可读性：清晰的变量命名和详细注释

"""

```
def can_complete_circuit(gas, cost):
```

"""

题目 1: LeetCode 134. 加油站

问题描述: 在一条环路上有  $n$  个加油站，第  $i$  个加油站有汽油  $gas[i]$  升。你有一辆油箱容量无限的汽车，从第  $i$  个加油站开往第  $i+1$  个加油站需要消耗汽油  $cost[i]$  升。你从其中一个加油站出发，开始时油箱为空。

如果你可以绕环路行驶一周，返回出发时加油站的编号，否则返回-1。

算法思路: 贪心策略，维护当前剩余油量和总剩余油量

1. 如果总油量小于总消耗，肯定无法绕行一周
2. 贪心策略：如果从 A 到 B 的路上没油了，那么 A 到 B 之间的任何一个站点都不能作为起点

Args:

gas (List[int]): 每个加油站的汽油量  
cost (List[int]): 从当前加油站到下一站的消耗

Returns:

int: 能够完成环路的起始加油站编号, 如果无法完成返回-1

时间复杂度:  $O(n)$ , 其中  $n$  是加油站数量

空间复杂度:  $O(1)$ , 只使用常数额外空间

"""

# 异常处理: 检查输入长度是否匹配

```
if len(gas) != len(cost):  
    return -1
```

```
total_gas = 0      # 总油量
```

```
current_gas = 0    # 当前剩余油量
```

```
start_station = 0  # 起始加油站
```

# 遍历所有加油站

# 时间复杂度:  $O(n)$

```
for i in range(len(gas)):
```

```
    total_gas += gas[i] - cost[i]
```

```
    current_gas += gas[i] - cost[i]
```

# 如果当前剩余油量为负, 说明从 start\_station 到 i 的路径不可行

```
if current_gas < 0:
```

```
    start_station = i + 1  # 从下一个站点重新开始计算
```

```
    current_gas = 0         # 重置当前剩余油量
```

# 如果总油量小于总消耗, 那么无论如何都不可能绕行一周

```
return start_station if total_gas >= 0 else -1
```

```
def remove_k_digits(num, k):
```

"""

题目 2: LeetCode 402. 移掉 K 位数字

问题描述: 给定一个以字符串表示的非负整数 num, 移除这个数中的 k 位数字, 使得剩下的数字最小。

算法思路: 使用单调栈保持数字序列递增

1. 贪心策略: 从左到右遍历, 如果当前数字小于栈顶数字且还有删除次数, 则弹出栈顶数字
2. 使用栈来存储需要保留的数字

Args:

num (str): 表示非负整数的字符串

k (int): 需要移除的数位数

Returns:

str: 移除 k 位数字后的最小数字字符串

时间复杂度: O(n), 其中 n 是字符串长度

空间复杂度: O(n), 栈的空间

"""

# 异常处理: 如果移除位数大于等于字符串长度, 返回"0"

```
if len(num) <= k:  
    return "0"
```

```
stack = [] # 使用栈存储保留的数字
```

# 贪心策略: 从左到右遍历, 如果当前数字小于栈顶数字, 且还有删除次数, 则弹出栈顶数字

# 时间复杂度: O(n)

```
for digit in num:
```

# 当栈不为空, 当前数字小于栈顶数字, 且还有删除次数时, 弹出栈顶

```
while stack and k > 0 and digit < stack[-1]:
```

```
    stack.pop()
```

```
    k -= 1
```

```
stack.append(digit)
```

# 处理剩余的删除次数

```
while k > 0:
```

```
    stack.pop()
```

```
    k -= 1
```

# 构建结果字符串, 去除前导零

```
result = ''.join(stack).lstrip('0')
```

```
return result if result else "0"
```

```
def array_pair_sum(nums):
```

"""

题目 3: LeetCode 561. 数组拆分 I

问题描述: 给定长度为  $2n$  的整数数组  $\text{nums}$ , 将这些数分成  $n$  对, 使得从 1 到  $n$  的  $\min(a_i, b_i)$  总和最大。

算法思路: 排序后取每对的第一个元素

1. 贪心策略: 将数组排序后, 每两个相邻的数分为一组, 取较小的那个 (即每对中的第一个数)

2. 这样可以最大化较小值的总和

Args:

nums (List[int]): 长度为  $2n$  的整数数组

Returns:

int: 最大  $\min(a_i, b_i)$  总和

时间复杂度:  $O(n \log n)$ , 主要是排序的时间复杂度

空间复杂度:  $O(1)$ , 只使用常数额外空间

"""

# 异常处理: 检查数组长度是否为偶数

```
if len(nums) % 2 != 0:  
    return 0
```

# 贪心策略: 将数组排序后, 每两个相邻的数分为一组, 取较小的那个

# 时间复杂度:  $O(n \log n)$

```
nums.sort()
```

```
total_sum = 0
```

# 每隔一个元素取一个 (即每对中的第一个元素)

# 时间复杂度:  $O(n)$

```
for i in range(0, len(nums), 2):  
    total_sum += nums[i]
```

```
return total_sum
```

def check\_possibility(nums):

"""

题目 4: LeetCode 665. 非递减数列

问题描述: 给定一个长度为  $n$  的整数数组  $nums$ , 判断在最多改变 1 个元素的情况下, 该数组能否变成非递减数列。

算法思路: 遇到逆序对时优先修改前面的数字

1. 遍历数组, 统计需要修改的次数

2. 贪心策略: 尽可能修改  $nums[i]$  而不是  $nums[i+1]$ , 这样对后续影响更小

Args:

$nums$  (List[int]): 整数数组

Returns:

bool: 是否可以通过最多修改一个元素使数组变为非递减数列

时间复杂度:  $O(n)$ , 其中  $n$  是数组长度

空间复杂度:  $O(1)$ , 只使用常数额外空间

"""

# 异常处理: 空数组或单元素数组已经是非递减数列

```

if len(nums) <= 1:
    return True

count = 0 # 记录需要修改的次数

# 时间复杂度: O(n)
for i in range(len(nums) - 1):
    if nums[i] > nums[i + 1]:
        count += 1
    if count > 1:
        return False # 需要修改超过 1 次

# 贪心选择: 优先修改 nums[i] 而不是 nums[i+1]
if i > 0 and nums[i - 1] > nums[i + 1]:
    nums[i + 1] = nums[i] # 必须修改 nums[i+1]
else:
    nums[i] = nums[i + 1] # 修改 nums[i]

return True

```

def maximum\_toys(prices, budget):

"""

题目 5: HackerRank – Mark and Toys

问题描述: 马克想用给定的预算购买最多的玩具, 每个玩具都有价格, 求能购买的最大玩具数量。

算法思路: 排序价格后贪心购买

1. 贪心策略: 优先购买价格最低的玩具
2. 将玩具价格排序后依次购买, 直到预算不足

Args:

prices (List[int]): 玩具价格列表  
 budget (int): 预算金额

Returns:

int: 能购买的最大玩具数量

时间复杂度:  $O(n \log n)$ , 主要是排序的时间复杂度

空间复杂度:  $O(1)$ , 只使用常数额外空间

"""

# 异常处理: 检查输入是否有效

if not prices or budget <= 0:
 return 0

```

# 贪心策略：优先购买价格最低的玩具
# 时间复杂度：O(n log n)
prices.sort()
count = 0          # 购买的玩具数量
current_cost = 0   # 当前花费

# 时间复杂度：O(n)
for price in prices:
    if current_cost + price <= budget:
        current_cost += price
        count += 1
    else:
        break # 预算不足，无法购买更多玩具

return count

```

```

def luck_balance(k, contests):
"""

```

题目 6: HackerRank – Luck Balance

问题描述：参赛者可以输掉一些比赛来获得运气值，但重要比赛最多只能输掉 k 场，求能获得的最大运气值。

算法思路：输掉不重要比赛，输掉 k 场重要比赛

1. 贪心策略：输掉所有不重要比赛，输掉 k 场运气值最高的重要比赛
2. 将重要比赛按运气值降序排序，前 k 场输掉，其余赢取

Args:

```

k (int): 最多能输掉的重要比赛场数
contests (List[List[int]]): 比赛信息列表，每个元素为[luck, importance]

```

Returns:

int: 能获得的最大运气值

时间复杂度：O(n log n)，主要是排序的时间复杂度

空间复杂度：O(n)，存储重要比赛运气值的空间

"""

# 异常处理：检查输入是否有效

if not contests:

return 0

important = [] # 重要比赛的运气值

total\_luck = 0 # 总运气值

```

# 分离重要比赛和不重要比赛
# 时间复杂度: O(n)
for contest in contests:
    luck, importance = contest
    if importance == 1:
        important.append(luck)
    else:
        total_luck += luck # 不重要比赛直接输掉

# 重要比赛按运气值降序排序
# 时间复杂度: O(m log m), 其中 m 是重要比赛数量
important.sort(reverse=True)

# 输掉前 k 场重要比赛
# 时间复杂度: O(m)
for i in range(len(important)):
    if i < k:
        total_luck += important[i] # 输掉比赛, 增加运气
    else:
        total_luck -= important[i] # 赢得比赛, 减少运气

return total_luck

```

def crazy\_herbs(time, herbs):

"""

题目 7: 牛客网 - 疯狂的采药 (分数背包贪心解法)

问题描述: 在给定时间内采药, 每种草药有价值和采摘时间, 求能获得的最大价值 (可部分采摘)。

算法思路: 按性价比排序后贪心选择

1. 贪心策略: 按性价比 (价值/时间) 降序排序
2. 优先采摘性价比高的草药, 可部分采摘

Args:

time (int): 总采摘时间

herbs (List[List[int]]): 草药信息列表, 每个元素为[value, cost]

Returns:

int: 能获得的最大价值

时间复杂度:  $O(n \log n)$ , 主要是排序的时间复杂度

空间复杂度:  $O(n)$ , 存储草药信息的空间

"""

# 异常处理: 检查输入是否有效

```

if not herbs or time <= 0:
    return 0

# 计算每种草药的性价比
# 时间复杂度: O(n)
herb_list = []
for herb in herbs:
    value, cost = herb
    ratio = value / cost
    herb_list.append((ratio, value, cost))

```

```

# 按性价比降序排序
# 时间复杂度: O(n log n)
herb_list.sort(key=lambda x: x[0], reverse=True)

```

```

total_value = 0          # 总价值
remaining_time = time   # 剩余时间

```

```

# 贪心选择草药
# 时间复杂度: O(n)
for herb in herb_list:
    ratio, value, cost = herb

    if remaining_time >= cost:
        # 可以采摘完整的草药
        total_value += value
        remaining_time -= cost
    else:
        # 采摘部分草药 (分数背包)
        total_value += int(value * (remaining_time / cost))
        break

```

```

return total_value

```

```

def honest_coach.skills):
    """

```

题目 8: Codeforces 1360B – Honest Coach

问题描述: 教练要将学生分成两组, 使得两组学生能力值的最大差值最小。

算法思路: 排序后找最小相邻差值

1. 贪心策略: 排序后相邻元素的差值最小
2. 最小相邻差值即为两组学生能力值的最小差值

Args:

skills (List[int]): 学生能力值列表

Returns:

int: 两组学生能力值的最小差值

时间复杂度:  $O(n \log n)$ , 主要是排序的时间复杂度

空间复杂度:  $O(1)$ , 只使用常数额外空间

"""

# 异常处理: 空数组或单元素数组

if len.skills) <= 1:

    return 0

# 贪心策略: 排序后相邻元素的差值最小

# 时间复杂度:  $O(n \log n)$

skills.sort()

min\_diff = float('inf')

# 找最小相邻差值

# 时间复杂度:  $O(n)$

for i in range(1, len.skills)):

    min\_diff = min(min\_diff, skills[i] - skills[i - 1])

return min\_diff

def max\_non\_overlapping\_intervals(intervals):

"""

题目 9: 洛谷 P1803 – 线段覆盖

问题描述: 给定一些区间, 选择最多的不重叠区间。

算法思路: 经典活动选择问题

1. 贪心策略: 按结束时间排序, 优先选择结束时间早的区间

2. 这样能为后续区间留出更多空间

Args:

intervals (List[List[int]]): 区间列表, 每个元素为 [start, end]

Returns:

int: 最多不重叠区间数量

时间复杂度:  $O(n \log n)$ , 主要是排序的时间复杂度

空间复杂度:  $O(1)$ , 只使用常数额外空间

"""

```

# 异常处理: 空区间列表
if not intervals:
    return 0

# 贪心策略: 按结束时间排序, 优先选择结束时间早的区间
# 时间复杂度: O(n log n)
intervals.sort(key=lambda x: x[1])

count = 1           # 不重叠区间数量
end = intervals[0][1] # 上一个选择区间的结束时间

# 时间复杂度: O(n)
for i in range(1, len(intervals)):
    if intervals[i][0] >= end:
        count += 1
        end = intervals[i][1]

return count

```

```
def largest_sum_after_k_negations(nums, k):
```

```
"""

```

题目 10: LeetCode 1005. K 次取反后最大化的数组和

问题描述: 给定一个整数数组 `nums` 和整数 `k`, 按以下方法修改数组: 选择某个下标 `i` 并将 `nums[i]` 替换为 `-nums[i]`,

重复这个过程恰好 `k` 次, 返回数组可能的最大和。

算法思路: 优先处理负数, 然后处理最小正数

1. 贪心策略: 优先将负数变为正数
2. 如果 `k` 还有剩余且为奇数, 对最小的数取反

Args:

`nums` (`List[int]`): 整数数组

`k` (`int`): 取反操作次数

Returns:

`int`: 取反后可能的最大数组和

时间复杂度:  $O(n \log n)$ , 主要是排序的时间复杂度

空间复杂度:  $O(1)$ , 只使用常数额外空间

```
"""

```

# 异常处理: 空数组

```
if not nums:
    return 0
```

```
# 贪心策略：优先处理负数
# 时间复杂度：O(n log n)
nums.sort()

# 优先将负数变为正数
# 时间复杂度：O(n)
for i in range(len(nums)):
    if k > 0 and nums[i] < 0:
        nums[i] = -nums[i]
        k -= 1
    else:
        break

# 如果 k 还有剩余且为奇数，对最小的数取反
if k > 0 and k % 2 == 1:
    nums.sort() # 重新排序找到最小的数
    nums[0] = -nums[0]

# 计算数组和
# 时间复杂度：O(n)
return sum(nums)

# 测试函数
if __name__ == "__main__":
    # 测试加油站问题
    gas = [1, 2, 3, 4, 5]
    cost = [3, 4, 5, 1, 2]
    print("加油站测试:", can_complete_circuit(gas, cost)) # 期望: 3

    # 测试移掉 K 位数字
    print("移掉 K 位数字测试:", remove_k_digits("1432219", 3)) # 期望: "1219"

    # 测试数组拆分
    nums = [1, 4, 3, 2]
    print("数组拆分测试:", array_pair_sum(nums)) # 期望: 4

    # 测试非递减数列
    nums2 = [4, 2, 3]
    print("非递减数列测试:", check_possibility(nums2)) # 期望: True

    # 测试疯狂的采药
    herbs = [[60, 10], [100, 20], [120, 30]]
```

```
print("疯狂的采药测试:", crazy_herbs(50, herbs)) # 期望: 240
```

```
=====
```

文件: Code17\_GreedyAdvancedProblems.cpp

```
=====
```

```
#include <vector>
#include <algorithm>
#include <queue>
#include <string>
#include <deque>
#include <functional>
#include <iostream>
```

```
using namespace std;
```

```
/***
 * 贪心算法高级题目集合 - C++版本
 * 包含更复杂的贪心算法问题和优化技巧
 */
```

```
/**
 * 题目 1: LeetCode 630. 课程表 III
 * 算法思路: 按截止时间排序, 使用最大堆维护已选课程的持续时间
 * 时间复杂度: O(n log n), 空间复杂度: O(n)
 */
```

```
int scheduleCourse(vector<vector<int>>& courses) {
    if (courses.empty()) return 0;
```

```
// 按截止时间排序
```

```
sort(courses.begin(), courses.end(), [] (const vector<int>& a, const vector<int>& b) {
    return a[1] < b[1];
});
```

```
// 最大堆, 存储已选课程的持续时间
```

```
priority_queue<int> maxHeap;
int currentTime = 0;
```

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

```
    if (currentTime + duration <= lastDay) {
```

```

    // 可以完成当前课程
    currentTime += duration;
    maxHeap.push(duration);
} else if (!maxHeap.empty() && maxHeap.top() > duration) {
    // 替换掉持续时间最长的课程
    currentTime = currentTime - maxHeap.top() + duration;
    maxHeap.pop();
    maxHeap.push(duration);
}
}

return maxHeap.size();
}

```

```

/***
 * 题目 2: LeetCode 757. 设置交集大小至少为 2
 * 算法思路: 按结束位置排序, 维护两个最大的点
 * 时间复杂度: O(n log n), 空间复杂度: O(1)
 */
int intersectionSizeTwo(vector<vector<int>>& intervals) {
    if (intervals.empty()) return 0;

    // 按结束位置升序排序, 结束位置相同时按开始位置降序排序
    sort(intervals.begin(), intervals.end(), [] (const vector<int>& a, const vector<int>& b) {
        if (a[1] != b[1]) {
            return a[1] < b[1];
        } else {
            return a[0] > b[0];
        }
    });
}

int result = 0;
int first = -1, second = -1;

for (auto& interval : intervals) {
    int start = interval[0];
    int end = interval[1];

    if (start > second) {
        // 需要添加两个新点
        result += 2;
        first = end - 1;
        second = end;
    }
}

```

```

    } else if (start > first) {
        // 需要添加一个新点
        result += 1;
        first = second;
        second = end;
    }
}

return result;
}

/***
 * 题目 3: LeetCode 1353. 最多可以参加的会议数目
 * 算法思路: 按开始时间排序, 使用最小堆维护当前可参加的会议
 * 时间复杂度: O(n log n), 空间复杂度: O(n)
 */
int maxEvents(vector<vector<int>>& events) {
    if (events.empty()) return 0;

    // 按开始时间排序
    sort(events.begin(), events.end(), [] (const vector<int>& a, const vector<int>& b) {
        return a[0] < b[0];
    });

    // 最小堆, 存储当前可参加的会议的结束时间
    priority_queue<int, vector<int>, greater<int>> minHeap;
    int result = 0;
    int day = 1;
    int index = 0;
    int n = events.size();

    while (index < n || !minHeap.empty()) {
        // 将今天开始的会议加入堆中
        while (index < n && events[index][0] == day) {
            minHeap.push(events[index][1]);
            index++;
        }

        // 移除已经过期的会议
        while (!minHeap.empty() && minHeap.top() < day) {
            minHeap.pop();
        }
    }
}

```

```

// 参加结束时间最早的会议
if (!minHeap.empty()) {
    minHeap.pop();
    result++;
}

day++;
}

return result;
}

/***
 * 题目 4: LeetCode 1642. 可以到达的最远建筑
 * 算法思路: 使用最小堆维护已使用的梯子
 * 时间复杂度: O(n log k), 空间复杂度: O(k)
 */
int furthestBuilding(vector<int>& heights, int bricks, int ladders) {
    if (heights.size() <= 1) return 0;

    // 最小堆, 存储已使用的梯子对应的高度差
    priority_queue<int, vector<int>, greater<int>> minHeap;

    for (int i = 1; i < heights.size(); i++) {
        int diff = heights[i] - heights[i - 1];

        if (diff > 0) {
            // 需要爬升
            minHeap.push(diff);

            // 如果梯子不够用, 用砖块替换最小的梯子使用
            if (minHeap.size() > ladders) {
                bricks -= minHeap.top();
                minHeap.pop();

                if (bricks < 0) {
                    return i - 1; // 无法到达当前建筑
                }
            }
        }
    }

    return heights.size() - 1;
}

```

```

/***
 * 题目 5: LeetCode 316. 去除重复字母
 * 算法思路: 使用单调栈维护结果字符串
 * 时间复杂度: O(n), 空间复杂度: O(n)
 */
string removeDuplicateLetters(string s) {
    if (s.empty()) return "";

    // 记录每个字符的最后出现位置
    vector<int> lastPos(26, 0);
    for (int i = 0; i < s.length(); i++) {
        lastPos[s[i] - 'a'] = i;
    }

    vector<bool> visited(26, false);
    deque<char> stack;

    for (int i = 0; i < s.length(); i++) {
        char c = s[i];

        // 如果字符已经在栈中, 跳过
        if (visited[c - 'a']) {
            continue;
        }

        // 维护单调栈: 当栈顶字符大于当前字符且后面还会出现时, 弹出栈顶
        while (!stack.empty() && stack.back() > c && lastPos[stack.back() - 'a'] > i) {
            visited[stack.back() - 'a'] = false;
            stack.pop_back();
        }

        stack.push_back(c);
        visited[c - 'a'] = true;
    }

    return string(stack.begin(), stack.end());
}

/***
 * 题目 6: LeetCode 768. 最多能完成排序的块 II
 * 算法思路: 维护当前块的最大值和前缀最大值
 * 时间复杂度: O(n), 空间复杂度: O(n)
*/

```

```

*/
int maxChunksToSorted(vector<int>& arr) {
    if (arr.empty()) return 0;

    int n = arr.size();
    vector<int> maxLeft(n);
    vector<int> minRight(n);

    // 计算从左到右的最大值
    maxLeft[0] = arr[0];
    for (int i = 1; i < n; i++) {
        maxLeft[i] = max(maxLeft[i - 1], arr[i]);
    }

    // 计算从右到左的最小值
    minRight[n - 1] = arr[n - 1];
    for (int i = n - 2; i >= 0; i--) {
        minRight[i] = min(minRight[i + 1], arr[i]);
    }

    int chunks = 0;
    for (int i = 0; i < n - 1; i++) {
        if (maxLeft[i] <= minRight[i + 1]) {
            chunks++;
        }
    }

    return chunks + 1; // 最后一块
}

/**
 * 题目 7: LeetCode 1326. 灌溉花园的最少水龙头数目
 * 算法思路: 区间覆盖问题, 贪心选择
 * 时间复杂度: O(n), 空间复杂度: O(n)
 */
int minTaps(int n, vector<int>& ranges) {
    if (ranges.size() != n + 1) return -1;

    // 创建区间数组
    vector<vector<int>> intervals(n + 1, vector<int>(2));
    for (int i = 0; i <= n; i++) {
        int left = max(0, i - ranges[i]);
        int right = min(n, i + ranges[i]);
        intervals[i][0] = left;
        intervals[i][1] = right;
    }

    int current_tap = 0, last_tap = 0, max_reach = 0;
    int taps = 0;
    while (last_tap < n) {
        int new_max_reach = max_reach;
        for (int i = last_tap + 1; i <= max_reach; i++) {
            if (intervals[i][0] <= new_max_reach) {
                new_max_reach = max(new_max_reach, intervals[i][1]);
            }
        }
        if (new_max_reach == last_tap) {
            return -1; // 无法灌溉整个花园
        }
        last_tap = new_max_reach;
        taps++;
    }

    return taps;
}

```

```

intervals[i] = {left, right};

}

// 按左端点排序
sort(intervals.begin(), intervals.end(), [] (const vector<int>& a, const vector<int>& b) {
    return a[0] < b[0];
});

int taps = 0;
int currentEnd = 0;
int farthest = 0;
int i = 0;

while (currentEnd < n) {
    // 找到能覆盖当前结束位置的最远水龙头
    while (i <= n && intervals[i][0] <= currentEnd) {
        farthest = max(farthest, intervals[i][1]);
        i++;
    }

    if (farthest <= currentEnd) {
        return -1; // 无法覆盖
    }

    taps++;
    currentEnd = farthest;

    if (currentEnd >= n) {
        break;
    }
}

return taps;
}

// 测试函数
int main() {
    // 测试课程表 III
    vector<vector<int>> courses = {{100, 200}, {200, 1300}, {1000, 1250}, {2000, 3200}};
    cout << "课程表 III 测试: " << scheduleCourse(courses) << endl; // 期望: 3

    // 测试去除重复字母
    cout << "去除重复字母测试: " << removeDuplicateLetters("bcabc") << endl; // 期望: "abc"
}

```

```
// 测试最多可以参加的会议数目
vector<vector<int>> events = {{1, 2}, {2, 3}, {3, 4}, {1, 2}};
cout << "最多会议测试: " << maxEvents(events) << endl; // 期望: 4

return 0;
}
```

---

文件: Code17\_GreedyAdvancedProblems.java

---

```
=====
package class094;

import java.util.*;

/**
 * 贪心算法高级题目集合 (Greedy Algorithm Advanced Problems Collection)
 * 包含更复杂的贪心算法问题和优化技巧
 * 涵盖区间调度、资源分配、路径优化等高级应用
 *
 * 算法标签: 贪心算法(Greedy Algorithm)、高级应用(Advanced Applications)、多领域问题(Multi-domain Problems)
 * 时间复杂度: 各题目不同, 详见具体函数注释
 * 空间复杂度: 各题目不同, 详见具体函数注释
 * 相关题目: 课程调度、区间交集、会议安排、建筑攀爬、字符串处理等
 * 贪心算法专题 - 高级题目集合
 */
public class Code17_GreedyAdvancedProblems {

    /**
     * 题目 1: LeetCode 630. 课程表 III (Course Schedule III)
     * 题目描述: 有 n 门课程, 每门课程有持续时间 duration 和截止时间 lastDay
     * 选择课程使得在截止时间前完成, 求最多能选多少门课程
     * 链接: https://leetcode.cn/problems/course-schedule-iii/
     *
     * 算法标签: 贪心算法(Greedy Algorithm)、优先队列(Priority Queue)、课程调度(Course Scheduling)
     * 时间复杂度: O(n log n) - 排序和堆操作
     * 空间复杂度: O(n) - 最大堆存储已选课程
     * 是否最优解: 是
     *
     * 算法思路详解:
```

- \* 1. 贪心策略核心：按截止时间排序，使用最大堆维护已选课程的持续时间，这样可以确保在时间不够时替换掉持续时间最长的课程
- \* 2. 选择机制：如果当前课程可以完成则直接加入，否则替换最长课程
- \* 3. 堆维护：使用最大堆快速获取已选课程中最长的持续时间
- \*
- \* 工程化最佳实践：
- \* 1. 排序优化：使用高效的比较器进行截止时间排序
- \* 2. 堆操作：合理维护堆中元素的一致性
- \* 3. 边界处理：妥善处理空数组和无课程情况
- \*
- \* 实际应用场景：
- \* 1. 教育管理：课程安排优化
- \* 2. 项目管理：任务调度优化
- \* 3. 资源分配：时间资源最优利用

\*/

```

public static int scheduleCourse(int[][] courses) {
    if (courses == null || courses.length == 0) {
        return 0;
    }

    // 按截止时间排序
    Arrays.sort(courses, (a, b) -> a[1] - b[1]);

    // 最大堆，存储已选课程的持续时间
    PriorityQueue<Integer> maxHeap = new PriorityQueue<>((a, b) -> b - a);
    int currentTime = 0;

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

        if (currentTime + duration <= lastDay) {
            // 可以完成当前课程
            currentTime += duration;
            maxHeap.offer(duration);
        } else if (!maxHeap.isEmpty() && maxHeap.peek() > duration) {
            // 替换掉持续时间最长的课程
            currentTime = currentTime - maxHeap.poll() + duration;
            maxHeap.offer(duration);
        }
    }

    return maxHeap.size();
}

```

```
}

/**
 * 题目 2: LeetCode 757. 设置交集大小至少为 2 (Set Intersection Size At Least Two)
 * 题目描述: 给定一组区间, 选择最少的点使得每个区间至少包含 2 个点
 * 链接: https://leetcode.cn/problems/set-intersection-size-at-least-two/
 *
 * 算法标签: 贪心算法(Greedy Algorithm)、区间覆盖(Interval Covering)、点选择(Point Selection)
 * 时间复杂度: O(n log n) - 排序的时间复杂度
 * 空间复杂度: O(1) - 常数空间维护两个点
 * 是否最优解: 是
 *
 * 算法思路详解:
 * 1. 贪心策略核心: 按结束位置排序, 从后往前选择点,
 *    这样可以最大化点的复用率, 最小化点的数量
 * 2. 点维护: 维护两个最大的点, 确保每个区间包含至少两个点
 * 3. 选择策略: 根据当前区间与已选点的关系决定是否添加新点
 *
 * 工程化最佳实践:
 * 1. 排序策略: 结束位置升序, 相同时开始位置降序
 * 2. 状态维护: 准确维护两个最大点的位置
 * 3. 条件判断: 优化点添加条件判断
 *
 * 实际应用场景:
 * 1. 传感器部署: 最少传感器覆盖所有区域
 * 2. 监控系统: 最少摄像头覆盖所有通道
 * 3. 网络部署: 最少基站覆盖所有区域
 */

public static int intersectionSizeTwo(int[][] intervals) {
    if (intervals == null || intervals.length == 0) {
        return 0;
    }

    // 按结束位置升序排序, 结束位置相同时按开始位置降序排序
    Arrays.sort(intervals, (a, b) -> {
        if (a[1] != b[1]) {
            return a[1] - b[1];
        } else {
            return b[0] - a[0];
        }
    });

    int result = 0;
```

```

int first = -1, second = -1; // 维护两个最大的点

for (int[] interval : intervals) {
    int start = interval[0];
    int end = interval[1];

    if (start > second) {
        // 需要添加两个新点
        result += 2;
        first = end - 1;
        second = end;
    } else if (start > first) {
        // 需要添加一个新点
        result += 1;
        first = second;
        second = end;
    }
    // 否则，当前区间已经包含至少两个点
}

return result;
}

```

/\*\*

\* 题目 3: LeetCode 1353. 最多可以参加的会议数目 (Maximum Number of Events That Can Be Attended)

\* 题目描述: 给定会议的开始和结束时间, 每天只能参加一个会议, 求最多能参加多少会议

\* 链接: <https://leetcode.cn/problems/maximum-number-of-events-that-can-be-attended/>

\*

\* 算法标签: 贪心算法(Greedy Algorithm)、时间调度(Time Scheduling)、优先队列(Priority Queue)

\* 时间复杂度:  $O(n \log n)$  – 排序和堆操作

\* 空间复杂度:  $O(n)$  – 最小堆存储可参加会议

\* 是否最优解: 是

\*

\* 算法思路详解:

\* 1. 贪心策略核心: 按开始时间排序, 使用最小堆维护当前可参加会议,

\* 每天选择结束时间最早的会议参加, 这样可以为后续会议留出更多时间

\* 2. 时间模拟: 逐天模拟会议参加过程

\* 3. 堆维护: 动态维护当前可参加会议的结束时间

\*

\* 工程化最佳实践:

\* 1. 时间管理: 准确模拟每天的会议安排

\* 2. 堆操作: 及时添加和移除过期会议

```

* 3. 边界处理：妥善处理无会议和会议冲突情况
*
* 实际应用场景：
* 1. 会议安排：最优会议参与策略
* 2. 任务调度：时间资源最优分配
* 3. 活动规划：活动参与优化
*/
public static int maxEvents(int[][] events) {
    if (events == null || events.length == 0) {
        return 0;
    }

    // 按开始时间排序
    Arrays.sort(events, (a, b) -> a[0] - b[0]);

    // 最小堆，存储当前可参加的会议的结束时间
    PriorityQueue<Integer> minHeap = new PriorityQueue<>();
    int result = 0;
    int day = 1;
    int index = 0;
    int n = events.length;

    while (index < n || !minHeap.isEmpty()) {
        // 将今天开始的会议加入堆中
        while (index < n && events[index][0] == day) {
            minHeap.offer(events[index][1]);
            index++;
        }

        // 移除已经过期的会议
        while (!minHeap.isEmpty() && minHeap.peek() < day) {
            minHeap.poll();
        }

        // 参加结束时间最早的会议
        if (!minHeap.isEmpty()) {
            minHeap.poll();
            result++;
        }
    }

    day++;
}

```

```

    return result;
}

/***
 * 题目 4: LeetCode 1642. 可以到达的最远建筑 (Furthest Building You Can Reach)
 * 题目描述: 使用梯子和砖块爬建筑, 求能到达的最远建筑
 * 链接: https://leetcode.cn/problems/furthest-building-you-can-reach/
 *
 * 算法标签: 贪心算法(Greedy Algorithm)、资源分配(Resource Allocation)、优先队列(Priority Queue)
 *
 * 时间复杂度: O(n log k) - k 是梯子数量
 * 空间复杂度: O(k) - 最小堆存储梯子使用
 * 是否最优解: 是
 *
 * 算法思路详解:
 * 1. 贪心策略核心: 使用最小堆维护已使用的梯子,
 * 优先使用梯子处理大的高度差, 当梯子不够时用砖块替换最小的梯子使用
 * 2. 资源管理: 合理分配梯子和砖块两种资源
 * 3. 替换策略: 用砖块替换最小梯子使用以节省梯子
 *
 * 工程化最佳实践:
 * 1. 资源优化: 最大化梯子的使用价值
 * 2. 堆维护: 准确维护梯子使用情况
 * 3. 边界处理: 妥善处理无高度差和资源不足情况
 *
 * 实际应用场景:
 * 1. 资源规划: 有限资源下的最优分配
 * 2. 项目管理: 工具和人力的最优使用
 * 3. 物流运输: 运输工具的最优调度
 */

public static int furthestBuilding(int[] heights, int bricks, int ladders) {
    if (heights == null || heights.length <= 1) {
        return 0;
    }

    // 最小堆, 存储已使用的梯子对应的高度差
    PriorityQueue<Integer> minHeap = new PriorityQueue<>();

    for (int i = 1; i < heights.length; i++) {
        int diff = heights[i] - heights[i - 1];

        if (diff > 0) {
            // 需要爬升
            if (minHeap.size() < ladders) {
                minHeap.add(diff);
            } else if (minHeap.peek() < diff) {
                minHeap.poll();
                minHeap.add(diff);
            } else {
                bricks -= diff - minHeap.peek();
            }
        }
    }

    return minHeap.size() < ladders ? heights.length - 1 : heights.length;
}

```

```

        minHeap.offer(diff);

        // 如果梯子不够用，用砖块替换最小的梯子使用
        if (minHeap.size() > ladders) {
            bricks -= minHeap.poll();
            if (bricks < 0) {
                return i - 1; // 无法到达当前建筑
            }
        }
    }

}

return heights.length - 1;
}

/***
 * 题目 5: LeetCode 321. 拼接最大数 (Create Maximum Number)
 * 题目描述: 从两个数组中保持相对顺序地取数字，拼接成最大的 k 位数
 * 链接: https://leetcode.cn/problems/create-maximum-number/
 *
 * 算法标签: 贪心算法(Greedy Algorithm)、序列合并(Sequence Merging)、单调栈(Monotonic Stack)
 * 时间复杂度: O(k * (m + n)) - m 和 n 分别是两个数组长度
 * 空间复杂度: O(k) - 存储结果数组
 * 是否最优解: 是
 *
 * 算法思路详解:
 * 1. 贪心策略核心: 分别从两个数组中取指定长度的最大子序列,
 *   然后合并两个子序列得到最大结果
 * 2. 子序列获取: 使用单调栈获取指定长度的最大子序列
 * 3. 序列合并: 合并两个子序列时保持相对顺序
 *
 * 工程化最佳实践:
 * 1. 枚举优化: 枚举合理的子序列长度组合
 * 2. 序列比较: 准确比较两个序列的字典序
 * 3. 内存管理: 及时释放临时数组
 *
 * 实际应用场景:
 * 1. 数据处理: 最优数据组合生成
 * 2. 密码学: 最大数字序列生成
 * 3. 金融计算: 最优数字组合
 */
public static int[] maxNumber(int[] nums1, int[] nums2, int k) {
    int m = nums1.length, n = nums2.length;

```

```

int[] result = new int[k];

// 枚举从 nums1 中取 i 个数字，从 nums2 中取 k-i 个数字
for (int i = Math.max(0, k - n); i <= Math.min(k, m); i++) {
    int[] seq1 = getMaxSubsequence(nums1, i);
    int[] seq2 = getMaxSubsequence(nums2, k - i);
    int[] merged = merge(seq1, seq2);

    if (compare(merged, 0, result, 0) > 0) {
        result = merged;
    }
}

```

return result;

}

// 获取长度为 k 的最大子序列（单调栈）

```

private static int[] getMaxSubsequence(int[] nums, int k) {
    if (k == 0) return new int[0];
    if (k == nums.length) return nums.clone();

    int[] stack = new int[k];
    int top = -1;
    int remain = nums.length - k; // 可以删除的数字数量

    for (int num : nums) {
        while (top >= 0 && stack[top] < num && remain > 0) {
            top--;
            remain--;
        }
        if (top < k - 1) {
            stack[++top] = num;
        } else {
            remain--;
        }
    }
}

return stack;
}

```

// 合并两个子序列

```

private static int[] merge(int[] nums1, int[] nums2) {
    int m = nums1.length, n = nums2.length;

```

```

    if (m == 0) return nums2;
    if (n == 0) return nums1;

    int[] result = new int[m + n];
    int i = 0, j = 0, idx = 0;

    while (i < m && j < n) {
        if (compare(nums1, i, nums2, j) > 0) {
            result[idx++] = nums1[i++];
        } else {
            result[idx++] = nums2[j++];
        }
    }

    while (i < m) result[idx++] = nums1[i++];
    while (j < n) result[idx++] = nums2[j++];

    return result;
}

```

// 比较两个序列的大小

```

private static int compare(int[] nums1, int i, int[] nums2, int j) {
    while (i < nums1.length && j < nums2.length) {
        if (nums1[i] != nums2[j]) {
            return nums1[i] - nums2[j];
        }
        i++;
        j++;
    }
    return (nums1.length - i) - (nums2.length - j);
}

```

/\*\*

- \* 题目 6: LeetCode 316. 去除重复字母 (Remove Duplicate Letters)
- \* 题目描述: 去除字符串中的重复字母, 使得每个字母只出现一次, 并且结果字典序最小
- \* 链接: <https://leetcode.cn/problems/remove-duplicate-letters/>
- \*
- \* 算法标签: 贪心算法(Greedy Algorithm)、单调栈(Monotonic Stack)、字符串处理(String Processing)

- \* 时间复杂度: O(n) - 一次遍历

- \* 空间复杂度: O(n) - 栈和辅助数组空间

- \* 是否最优解: 是

- \*

```
* 算法思路详解:  
* 1. 贪心策略核心: 使用单调栈维护结果字符串,  
*   记录每个字符的最后出现位置, 保持栈内字符单调递增  
* 2. 栈维护: 维护单调递增的字符序列  
* 3. 字符选择: 当栈顶字符大于当前字符且后面还会出现时, 弹出栈顶  
*  
* 工程化最佳实践:  
* 1. 位置记录: 准确记录每个字符的最后出现位置  
* 2. 栈操作: 合理维护栈中字符的访问状态  
* 3. 结果构建: 正确构建最终结果字符串  
*  
* 实际应用场景:  
* 1. 字符串优化: 生成字典序最小的字符串  
* 2. 密码处理: 字符去重和优化  
* 3. 数据清洗: 重复数据去除  
*/  
  
public static String removeDuplicateLetters(String s) {  
    if (s == null || s.isEmpty()) {  
        return "";  
    }  
  
    // 记录每个字符的最后出现位置  
    int[] lastPos = new int[26];  
    for (int i = 0; i < s.length(); i++) {  
        lastPos[s.charAt(i) - 'a'] = i;  
    }  
  
    boolean[] visited = new boolean[26];  
    Deque<Character> stack = new ArrayDeque<>();  
  
    for (int i = 0; i < s.length(); i++) {  
        char c = s.charAt(i);  
  
        // 如果字符已经在栈中, 跳过  
        if (visited[c - 'a']) {  
            continue;  
        }  
  
        // 维护单调栈: 当栈顶字符大于当前字符且后面还会出现时, 弹出栈顶  
        while (!stack.isEmpty() && stack.peek() > c && lastPos[stack.peek() - 'a'] > i) {  
            visited[stack.pop() - 'a'] = false;  
        }  
        stack.push(c);  
        visited[c - 'a'] = true;  
    }  
    return stack.toString();  
}
```

```

        stack.push(c);
        visited[c - 'a'] = true;
    }

    // 构建结果字符串
    StringBuilder result = new StringBuilder();
    while (!stack.isEmpty()) {
        result.append(stack.pop());
    }

    return result.reverse().toString();
}

/**
 * 题目 7: LeetCode 768. 最多能完成排序的块 II (Max Chunks To Make Sorted II)
 * 题目描述: 将数组分成最多的块, 使得每个块排序后连接起来的结果与整个数组排序后的结果相同
 * 链接: https://leetcode.cn/problems/max-chunks-to-make-sorted-ii/
 *
 * 算法标签: 贪心算法(Greedy Algorithm)、数组分块(Array Chunking)、前缀处理(Prefix Processing)
 * 时间复杂度: O(n) - 一次遍历
 * 空间复杂度: O(n) - 前缀最大值和后缀最小值数组
 * 是否最优解: 是
 *
 * 算法思路详解:
 * 1. 贪心策略核心: 维护当前块的最大值和前缀最大值,
 *    当当前块的最大值小于等于后面所有数的最小值时, 可以分块
 * 2. 分块条件: 前缀最大值≤后缀最小值
 * 3. 块数统计: 准确统计可分块数量
 *
 * 工程化最佳实践:
 * 1. 前缀处理: 准确计算前缀最大值和后缀最小值
 * 2. 分块判断: 优化分块条件判断
 * 3. 边界处理: 妥善处理数组边界情况
 *
 * 实际应用场景:
 * 1. 数据分片: 数组最优分片策略
 * 2. 并行处理: 数据块并行处理优化
 * 3. 内存管理: 大数组分块处理
 */
public static int maxChunksToSorted(int[] arr) {
    if (arr == null || arr.length == 0) {
        return 0;
    }
}

```

```

}

int n = arr.length;
int[] maxLeft = new int[n];
int[] minRight = new int[n];

// 计算从左到右的最大值
maxLeft[0] = arr[0];
for (int i = 1; i < n; i++) {
    maxLeft[i] = Math.max(maxLeft[i - 1], arr[i]);
}

// 计算从右到左的最小值
minRight[n - 1] = arr[n - 1];
for (int i = n - 2; i >= 0; i--) {
    minRight[i] = Math.min(minRight[i + 1], arr[i]);
}

int chunks = 0;
for (int i = 0; i < n - 1; i++) {
    if (maxLeft[i] <= minRight[i + 1]) {
        chunks++;
    }
}

return chunks + 1; // 最后一块
}

/***
 * 题目 8: LeetCode 1326. 灌溉花园的最少水龙头数目 (Minimum Number of Taps to Open to Water a Garden)
 * 题目描述: 水龙头可以灌溉一定范围的花园, 求最少需要多少个水龙头
 * 链接: https://leetcode.cn/problems/minimum-number-of-taps-to-open-to-water-a-garden/
 *
 * 算法标签: 贪心算法(Greedy Algorithm)、区间覆盖(Interval Covering)、资源优化(Resource Optimization)
 * 时间复杂度: O(n) - 一次遍历
 * 空间复杂度: O(n) - 区间数组存储
 * 是否最优解: 是
 *
 * 算法思路详解:
 * 1. 贪心策略核心: 区间覆盖问题, 将水龙头转换为区间,
 * 使用贪心选择覆盖整个花园的最少水龙头
 */

```

```

* 2. 覆盖策略：找到能覆盖当前结束位置的最远水龙头
* 3. 数量统计：准确统计所需水龙头数量
*
* 工程化最佳实践：
* 1. 区间转换：准确将水龙头转换为灌溉区间
* 2. 覆盖判断：优化区间覆盖条件判断
* 3. 异常处理：妥善处理无法覆盖情况
*
* 实际应用场景：
* 1. 设施部署：最少设施覆盖所有区域
* 2. 网络建设：最少基站覆盖所有用户
* 3. 监控部署：最少摄像头覆盖所有区域
*/

```

public static int minTaps(int n, int[] ranges) {

```

    if (ranges == null || ranges.length != n + 1) {
        return -1;
    }

    // 创建区间数组
    int[][] intervals = new int[n + 1][2];
    for (int i = 0; i <= n; i++) {
        int left = Math.max(0, i - ranges[i]);
        int right = Math.min(n, i + ranges[i]);
        intervals[i] = new int[]{left, right};
    }

    // 按左端点排序
    Arrays.sort(intervals, (a, b) -> a[0] - b[0]);

    int taps = 0;
    int currentEnd = 0;
    int farthest = 0;
    int i = 0;

    while (currentEnd < n) {
        // 找到能覆盖当前结束位置的最远水龙头
        while (i <= n && intervals[i][0] <= currentEnd) {
            farthest = Math.max(farthest, intervals[i][1]);
            i++;
        }

        if (farthest <= currentEnd) {
            return -1; // 无法覆盖
        }
        currentEnd = farthest;
        taps++;
    }
}
```

```

    }

    taps++;
    currentEnd = farthest;

    if (currentEnd >= n) {
        break;
    }
}

return taps;
}

// 测试函数
public static void main(String[] args) {
    // 测试课程表 III
    int[][] courses = {{100, 200}, {200, 1300}, {1000, 1250}, {2000, 3200}};
    System.out.println("课程表 III 测试: " + scheduleCourse(courses)); // 期望: 3

    // 测试去除重复字母
    System.out.println("去除重复字母测试: " + removeDuplicateLetters("bcabc")); // 期望:
"abc"

    // 测试最多可以参加的会议数目
    int[][] events = {{1, 2}, {2, 3}, {3, 4}, {1, 2}};
    System.out.println("最多会议测试: " + maxEvents(events)); // 期望: 4
}
}
=====
```

文件: Code17\_GreedyAdvancedProblems.py  
=====

"""
贪心算法高级题目集合 – Python 版本  
包含更复杂的贪心算法问题和优化技巧  
涵盖区间调度、资源分配、路径优化等高级应用

算法专题概览:

本文件包含 7 个高级贪心算法问题，展示了贪心算法在复杂场景中的应用：

1. 时间调度优化（课程表、会议安排）
2. 区间覆盖问题（交集大小、水龙头覆盖）
3. 资源分配优化（建筑爬升、块排序）

## 4. 字符串处理（去重优化）

贪心算法高级应用核心思想：

在复杂问题中，贪心算法通常需要结合其他数据结构（如堆、栈）来实现，并且需要更精细的策略设计来保证局部最优选择能导致全局最优解。

工程化最佳实践：

1. 异常处理：检查输入参数的有效性
2. 边界条件：处理空输入、单元素等特殊情况
3. 性能优化：选择合适的数据结构和算法策略
4. 可读性：清晰的变量命名和详细注释

"""

```
import heapq
from collections import deque
```

```
def schedule_course(courses):
```

"""

题目 1: LeetCode 630. 课程表 III

问题描述：有 n 门课程，每门课程有持续时间和截止时间，求最多能完成多少门课程。

算法思路：按截止时间排序，使用最大堆维护已选课程的持续时间

1. 贪心策略：按截止时间排序，优先考虑截止时间早的课程
2. 使用最大堆维护已选课程的持续时间，当无法加入新课程时，替换掉持续时间最长的课程

Args:

courses (List[List[int]]): 课程信息列表，每个元素为 [duration, last\_day]

Returns:

int: 最多能完成的课程数量

时间复杂度:  $O(n \log n)$ ，其中 n 是课程数量，主要是排序和堆操作的时间复杂度

空间复杂度:  $O(n)$ ，最大堆的空间

"""

```
# 异常处理：空课程列表
```

```
if not courses:
    return 0
```

```
# 按截止时间排序
```

```
# 时间复杂度:  $O(n \log n)$ 
```

```
courses.sort(key=lambda x: x[1])
```

```
# 最大堆（使用最小堆存储负值来模拟最大堆）
```

```

max_heap = []
current_time = 0 # 当前累计时间

# 遍历所有课程
# 时间复杂度: O(n log n)
for course in courses:
    duration, last_day = course

    if current_time + duration <= last_day:
        # 可以完成当前课程
        current_time += duration
        heapq.heappush(max_heap, -duration)
    elif max_heap and -max_heap[0] > duration:
        # 替换掉持续时间最长的课程
        current_time = current_time - (-max_heap[0]) + duration
        heapq.heappop(max_heap)
        heapq.heappush(max_heap, -duration)

return len(max_heap)

```

```
def intersection_size_two(intervals):
```

```
"""
```

题目 2: LeetCode 757. 设置交集大小至少为 2

问题描述: 给定一些区间, 找到最小的集合 S, 使得 S 与每个区间至少有 2 个整数相交。

算法思路: 按结束位置排序, 维护两个最大的点

1. 贪心策略: 按结束位置升序排序, 结束位置相同时按开始位置降序排序
2. 维护两个最大的点, 尽可能重用已选的点

Args:

intervals (List[List[int]]): 区间列表, 每个元素为 [start, end]

Returns:

int: 最小集合 S 的大小

时间复杂度: O(n log n), 主要是排序的时间复杂度

空间复杂度: O(1), 只使用常数额外空间

```
"""
```

# 异常处理: 空区间列表

```
if not intervals:
    return 0
```

# 按结束位置升序排序, 结束位置相同时按开始位置降序排序

```
# 时间复杂度: O(n log n)
intervals.sort(key=lambda x: (x[1], -x[0]))

result = 0      # 集合 S 的大小
first, second = -1, -1 # 维护的两个最大点

# 遍历所有区间
# 时间复杂度: O(n)
for interval in intervals:
    start, end = interval

    if start > second:
        # 需要添加两个新点
        result += 2
        first = end - 1
        second = end
    elif start > first:
        # 需要添加一个新点
        result += 1
        first = second
        second = end

return result
```

```
def max_events(events):
```

```
"""
```

题目 3: LeetCode 1353. 最多可以参加的会议数目

问题描述: 给定一些会议的开始和结束时间, 求最多能参加多少个会议。

算法思路: 按开始时间排序, 使用最小堆维护当前可参加的会议

1. 贪心策略: 按时间顺序处理, 每天参加结束时间最早的会议
2. 使用最小堆维护当前可参加的会议的结束时间

Args:

events (List[List[int]]): 会议信息列表, 每个元素为[start\_day, end\_day]

Returns:

int: 最多能参加的会议数目

时间复杂度:  $O(n \log n)$ , 其中  $n$  是会议数量, 主要是排序和堆操作的时间复杂度

空间复杂度:  $O(n)$ , 最小堆的空间

```
"""
```

# 异常处理: 空会议列表

```

if not events:
    return 0

# 按开始时间排序
# 时间复杂度: O(n log n)
events.sort(key=lambda x: x[0])

# 最小堆，存储当前可参加的会议的结束时间
min_heap = []
result = 0      # 参加的会议数
day = 1         # 当前天数
index = 0       # 会议索引
n = len(events)

# 按天处理
while index < n or min_heap:
    # 将今天开始的会议加入堆中
    while index < n and events[index][0] == day:
        heapq.heappush(min_heap, events[index][1])
        index += 1

    # 移除已经过期的会议
    while min_heap and min_heap[0] < day:
        heapq.heappop(min_heap)

    # 参加结束时间最早的会议
    if min_heap:
        heapq.heappop(min_heap)
        result += 1

    day += 1

return result

```

```
def furthest_building(heights, bricks, ladders):
```

```
    """
```

题目 4: LeetCode 1642. 可以到达的最远建筑

问题描述: 从建筑物 0 开始, 使用砖块和梯子爬升到尽可能远的建筑物。

算法思路: 使用最小堆维护已使用的梯子

1. 贪心策略: 优先使用梯子处理最大的高度差, 用砖块处理较小的高度差
2. 使用最小堆维护已使用的梯子对应的高度差

Args:

```
heights (List[int]): 建筑物高度列表  
bricks (int): 砖块数量  
ladders (int): 梯子数量
```

Returns:

```
int: 可以到达的最远建筑物索引
```

时间复杂度:  $O(n \log k)$ , 其中  $n$  是建筑物数量,  $k$  是梯子数量

空间复杂度:  $O(k)$ , 最小堆的空间

```
"""
```

```
# 异常处理: 建筑物数量不足
```

```
if len(heights) <= 1:  
    return 0
```

```
# 最小堆, 存储已使用的梯子对应的高度差
```

```
min_heap = []
```

```
# 遍历所有相邻建筑物
```

```
# 时间复杂度:  $O(n \log k)$ 
```

```
for i in range(1, len(heights)):  
    diff = heights[i] - heights[i - 1]  
  
    if diff > 0:  
        # 需要爬升  
        heapq.heappush(min_heap, diff)
```

```
# 如果梯子不够用, 用砖块替换最小的梯子使用
```

```
if len(min_heap) > ladders:  
    bricks -= heapq.heappop(min_heap)  
    if bricks < 0:  
        return i - 1 # 无法到达当前建筑
```

```
return len(heights) - 1
```

```
def remove_duplicate_letters(s):
```

```
"""
```

题目 5: LeetCode 316. 去除重复字母

问题描述: 去除字符串中的重复字母, 使得每个字母只出现一次, 且结果字典序最小。

算法思路: 使用单调栈维护结果字符串

1. 贪心策略: 维护单调递增的栈, 当遇到更小字符时, 如果后面还会出现, 则弹出栈顶
2. 使用栈来构建结果字符串

Args:

s (str): 输入字符串

Returns:

str: 去除重复字母后的字典序最小字符串

时间复杂度:  $O(n)$ , 其中  $n$  是字符串长度

空间复杂度:  $O(n)$ , 栈和集合的空间

"""

# 异常处理: 空字符串

if not s:

    return ""

# 记录每个字符的最后出现位置

# 时间复杂度:  $O(n)$

last\_pos = {}

for i, char in enumerate(s):

    last\_pos[char] = i

visited = set() # 记录已在栈中的字符

stack = [] # 单调栈

# 遍历字符串

# 时间复杂度:  $O(n)$

for i, char in enumerate(s):

    # 如果字符已经在栈中, 跳过

    if char in visited:

        continue

    # 维护单调栈: 当栈顶字符大于当前字符且后面还会出现时, 弹出栈顶

    while stack and stack[-1] > char and last\_pos[stack[-1]] > i:

        visited.remove(stack.pop())

    stack.append(char)

    visited.add(char)

return ''.join(stack)

def max\_chunks\_to\_sorted(arr):

"""

题目 6: LeetCode 768. 最多能完成排序的块 II

问题描述: 将数组分割成尽可能多的块, 使得每个块排序后连接起来与原数组排序后相同。

算法思路：维护当前块的最大值和前缀最大值

1. 贪心策略：当左半部分的最大值小于等于右半部分的最小值时，可以分割
2. 预处理计算前缀最大值和后缀最小值

Args:

arr (List[int]): 输入数组

Returns:

int: 最多能分割的块数

时间复杂度:  $O(n)$ , 其中  $n$  是数组长度

空间复杂度:  $O(n)$ , 存储前缀最大值和后缀最小值的空间

"""

# 异常处理: 空数组

if not arr:

return 0

n = len(arr)

max\_left = [0] \* n # max\_left[i] 表示 arr[0..i] 的最大值

min\_right = [0] \* n # min\_right[i] 表示 arr[i..n-1] 的最小值

# 计算从左到右的最大值

# 时间复杂度:  $O(n)$

max\_left[0] = arr[0]

for i in range(1, n):

max\_left[i] = max(max\_left[i - 1], arr[i])

# 计算从右到左的最小值

# 时间复杂度:  $O(n)$

min\_right[n - 1] = arr[n - 1]

for i in range(n - 2, -1, -1):

min\_right[i] = min(min\_right[i + 1], arr[i])

chunks = 0 # 块数

# 判断每个位置是否可以分割

# 时间复杂度:  $O(n)$

for i in range(n - 1):

if max\_left[i] <= min\_right[i + 1]:

chunks += 1

return chunks + 1 # 最后一块

```
def min_taps(n, ranges):
```

```
    """
```

题目 7: LeetCode 1326. 灌溉花园的最少水龙头数目

问题描述: 在长度为 n 的花园中, 每个位置有一个水龙头, 求灌溉整个花园的最少水龙头数目。

算法思路: 区间覆盖问题, 贪心选择

1. 贪心策略: 转换为区间覆盖问题, 每次选择能覆盖当前位置的最远水龙头
2. 按左端点排序后贪心选择

Args:

n (int): 花园长度

ranges (List[int]): 每个位置水龙头的灌溉范围

Returns:

int: 灌溉整个花园的最少水龙头数目, 无法灌溉返回-1

时间复杂度: O(n), 其中 n 是花园长度

空间复杂度: O(n), 存储区间信息的空间

```
"""
```

# 异常处理: 输入参数不匹配

```
if len(ranges) != n + 1:  
    return -1
```

# 创建区间数组

# 时间复杂度: O(n)

```
intervals = []
```

```
for i in range(n + 1):
```

```
    left = max(0, i - ranges[i])
```

```
    right = min(n, i + ranges[i])
```

```
    intervals.append((left, right))
```

# 按左端点排序

# 时间复杂度: O(n log n)

```
intervals.sort(key=lambda x: x[0])
```

taps = 0 # 水龙头数目

current\_end = 0 # 当前覆盖的结束位置

farthest = 0 # 能覆盖的最远位置

i = 0 # 区间索引

# 贪心选择水龙头

```
while current_end < n:
```

```

# 找到能覆盖当前结束位置的最远水龙头
while i <= n and intervals[i][0] <= current_end:
    farthest = max(farthest, intervals[i][1])
    i += 1

if farthest <= current_end:
    return -1 # 无法覆盖

taps += 1
current_end = farthest

if current_end >= n:
    break

return taps

# 测试函数
if __name__ == "__main__":
    # 测试课程表 III
    courses = [[100, 200], [200, 1300], [1000, 1250], [2000, 3200]]
    print("课程表 III 测试:", schedule_course(courses)) # 期望: 3

    # 测试去除重复字母
    print("去除重复字母测试:", remove_duplicate_letters("bcabc")) # 期望: "abc"

    # 测试最多可以参加的会议数目
    events = [[1, 2], [2, 3], [3, 4], [1, 2]]
    print("最多会议测试:", max_events(events)) # 期望: 4

    # 测试可以到达的最远建筑
    heights = [4, 2, 7, 6, 9, 14, 12]
    print("最远建筑测试:", furthest_building(heights, 5, 1)) # 期望: 4

```

=====

文件: Code18\_GreedyMathematicalProblems.cpp

=====

```

#include <vector>
#include <algorithm>
#include <queue>
#include <string>
#include <functional>
#include <iostream>

```

```
using namespace std;

/***
 * 贪心算法数学相关问题集合 - C++版本
 */

/***
 * 题目 1: LeetCode 179. 最大数
 * 算法思路: 自定义排序规则, 比较两个数字拼接后的结果
 * 时间复杂度: O(n log n), 空间复杂度: O(n)
 */
string largestNumber(vector<int>& nums) {
    if (nums.empty()) return "";

    // 将数字转换为字符串数组
    vector<string> strNums;
    for (int num : nums) {
        strNums.push_back(to_string(num));
    }

    // 自定义排序: 比较 a+b 和 b+a 的大小
    sort(strNums.begin(), strNums.end(), [](const string& a, const string& b) {
        return a + b > b + a; // 降序排列
    });

    // 处理前导零的情况
    if (strNums[0] == "0") {
        return "0";
    }

    // 拼接结果
    string result;
    for (const string& num : strNums) {
        result += num;
    }

    return result;
}

/***
 * 题目 2: LeetCode 991. 坏了的计算器
 * 算法思路: 反向思考, 从 Y 到 X, 只能进行除 2 和加 1 操作
 */
```

\* 时间复杂度:  $O(\log Y)$ , 空间复杂度:  $O(1)$

\*/

```
int brokenCalc(int X, int Y) {  
    if (X >= Y) {  
        return X - Y; // 只能减 1 操作  
    }  
  
    int operations = 0;  
    while (Y > X) {  
        operations++;  
        if (Y % 2 == 0) {  
            Y /= 2;  
        } else {  
            Y++;  
        }  
    }  
  
    return operations + (X - Y);  
}
```

/\*\*

\* 题目 3: LeetCode 910. 最小差值 II

\* 算法思路: 排序后寻找最佳分割点

\* 时间复杂度:  $O(n \log n)$ , 空间复杂度:  $O(1)$

\*/

```
int smallestRangeII(vector<int>& nums, int k) {  
    if (nums.size() <= 1) return 0;  
  
    sort(nums.begin(), nums.end());  
    int n = nums.size();  
    int result = nums[n - 1] - nums[0]; // 初始差值  
  
    // 尝试每个可能的分割点  
    for (int i = 0; i < n - 1; i++) {  
        int high = max(nums[n - 1] - k, nums[i] + k);  
        int low = min(nums[0] + k, nums[i + 1] - k);  
        result = min(result, high - low);  
    }  
  
    return result;  
}
```

/\*\*

\* 题目 4: LeetCode 1526. 形成目标数组的子数组最少增加次数

\* 算法思路: 相邻元素差值法

\* 时间复杂度: O(n), 空间复杂度: O(1)

\*/

```
int minNumberOperations(vector<int>& target) {  
    if (target.empty()) return 0;  
  
    int operations = target[0]; // 第一个元素需要的操作次数  
  
    for (int i = 1; i < target.size(); i++) {  
        if (target[i] > target[i - 1]) {  
            operations += target[i] - target[i - 1];  
        }  
    }  
  
    return operations;  
}
```

/\*\*

\* 题目 5: LeetCode 1247. 交换字符使得字符串相同

\* 算法思路: 统计不匹配的位置类型

\* 时间复杂度: O(n), 空间复杂度: O(1)

\*/

```
int minimumSwap(string s1, string s2) {  
    if (s1.length() != s2.length()) return -1;  
  
    int xy = 0, yx = 0; // 统计不匹配类型  
  
    for (int i = 0; i < s1.length(); i++) {  
        char c1 = s1[i];  
        char c2 = s2[i];  
  
        if (c1 == 'x' && c2 == 'y') {  
            xy++;  
        } else if (c1 == 'y' && c2 == 'x') {  
            yx++;  
        }  
    }  
}
```

// 如果总的不匹配数是奇数, 无法完成

```
if ((xy + yx) % 2 != 0) {  
    return -1;  
}
```

```

// 计算最少交换次数
return xy / 2 + yx / 2 + (xy % 2) * 2;
}

/***
* 题目 6: LeetCode 1561. 你可以获得的最大硬币数目
* 算法思路: 排序后每次取第二大的堆
* 时间复杂度: O(n log n), 空间复杂度: O(1)
*/
int maxCoins(vector<int>& piles) {
    if (piles.empty()) return 0;

    sort(piles.begin(), piles.end());
    int result = 0;
    int n = piles.size();

    // 每次取第二大的堆 (从倒数第二个开始, 每隔一个取一个)
    for (int i = n - 2; i >= n / 3; i -= 2) {
        result += piles[i];
    }

    return result;
}

/***
* 题目 7: LeetCode 1689. 十二进制数的最少数目
* 算法思路: 最少数量等于数字中最大的数字
* 时间复杂度: O(n), 空间复杂度: O(1)
*/
int minPartitions(string n) {
    if (n.empty()) return 0;

    int maxDigit = 0;
    for (char c : n) {
        maxDigit = max(maxDigit, c - '0');
    }

    return maxDigit;
}

/***
* 题目 8: LeetCode 1710. 卡车上的最大单元数
*/

```

```

* 算法思路：按单位容量价值降序排序
* 时间复杂度：O(n log n)，空间复杂度：O(1)
*/
int maximumUnits(vector<vector<int>>& boxTypes, int truckSize) {
    if (boxTypes.empty() || truckSize <= 0) return 0;

    // 按每个箱子的单元数降序排序
    sort(boxTypes.begin(), boxTypes.end(), [] (const vector<int>& a, const vector<int>& b) {
        return a[1] > b[1];
    });

    int totalUnits = 0;
    int remainingSize = truckSize;

    for (auto& box : boxTypes) {
        int number_of_Boxes = box[0];
        int units_per_Box = box[1];

        int boxes_to_Take = min(number_of_Boxes, remainingSize);
        totalUnits += boxes_to_Take * units_per_Box;
        remainingSize -= boxes_to_Take;

        if (remainingSize == 0) {
            break;
        }
    }

    return totalUnits;
}

/**
* 题目 9: LeetCode 1405. 最长快乐字符串
* 算法思路：优先使用剩余数量最多的字符
* 时间复杂度：O(n)，空间复杂度：O(1)
*/
string longestDiverseString(int a, int b, int c) {
    // 使用最大堆存储字符和剩余数量
    priority_queue<pair<int, char>> maxHeap;

    if (a > 0) maxHeap.push({a, 'a'});
    if (b > 0) maxHeap.push({b, 'b'});
    if (c > 0) maxHeap.push({c, 'c'});
}

```

```
string result;

while (!maxHeap.empty()) {
    auto first = maxHeap.top();
    maxHeap.pop();
    char char1 = first.second;
    int count1 = first.first;

    // 检查是否已经连续使用了两个相同字符
    int len = result.length();
    if (len >= 2 && result[len - 1] == char1 && result[len - 2] == char1) {
        // 需要换一个字符
        if (maxHeap.empty()) {
            break; // 没有其他字符可用
        }

        auto second = maxHeap.top();
        maxHeap.pop();
        char char2 = second.second;
        int count2 = second.first;

        result += char2;
        count2--;

        if (count2 > 0) {
            maxHeap.push({count2, char2});
        }
        maxHeap.push(first); // 把第一个字符放回去
    } else {
        // 可以使用当前字符
        result += char1;
        count1--;

        if (count1 > 0) {
            maxHeap.push({count1, char1});
        }
    }
}

return result;
}

/**
```

```

* 题目 10: LeetCode 1665. 完成所有任务的最少初始能量
* 算法思路: 按(最小要求 - 实际消耗)降序排序
* 时间复杂度: O(n log n), 空间复杂度: O(1)
*/
int minimumEffort(vector<vector<int>>& tasks) {
    if (tasks.empty()) return 0;

    // 按(最小要求 - 实际消耗)降序排序
    sort(tasks.begin(), tasks.end(), [] (const vector<int>& a, const vector<int>& b) {
        int diffA = a[1] - a[0];
        int diffB = b[1] - b[0];
        return diffB < diffA; // 降序排列
    });

    int result = 0;
    int currentEnergy = 0;

    for (auto& task : tasks) {
        int actual = task[0];      // 实际消耗
        int minimum = task[1];     // 最小要求

        if (currentEnergy < minimum) {
            // 需要补充能量
            int need = minimum - currentEnergy;
            result += need;
            currentEnergy += need;
        }

        // 完成任务, 消耗能量
        currentEnergy -= actual;
    }

    return result;
}

// 测试函数
int main() {
    // 测试最大数
    vector<int> nums1 = {3, 30, 34, 5, 9};
    cout << "最大数测试: " << largestNumber(nums1) << endl; // 期望: "9534330"

    // 测试坏了的计算器
    cout << "坏了的计算器测试: " << brokenCalc(2, 3) << endl; // 期望: 2
}

```

```

// 测试最小差值 II
vector<int> nums2 = {1, 3, 6};
cout << "最小差值 II 测试: " << smallestRangeII(nums2, 3) << endl; // 期望: 3

// 测试最大硬币数
vector<int> piles = {2, 4, 1, 2, 7, 8};
cout << "最大硬币数测试: " << maxCoins(piles) << endl; // 期望: 9

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

文件: Code18\_GreedyMathematicalProblems.java

```

package class094;

import java.util.*;

/**
 * 贪心算法数学相关问题集合 (Greedy Algorithm Mathematical Problems Collection)
 * 包含与数学相关的贪心算法问题，如分数背包、最优分配等
 * 涵盖数论、组合数学、概率统计等数学领域的应用
 *
 * 算法标签: 贪心算法(Greedy Algorithm)、数学应用(Mathematical Applications)、数值优化(Numerical Optimization)
 * 时间复杂度: 各题目不同, 详见具体函数注释
 * 空间复杂度: 各题目不同, 详见具体函数注释
 * 相关题目: 数字组合、能量管理、计算器操作、字符串构造等
 * 贪心算法专题 - 数学问题集合
 */
public class Code18_GreedyMathematicalProblems {

    /**
     * 题目 1: LeetCode 179. 最大数 (Largest Number)
     * 题目描述: 给定一组非负整数, 重新排列它们的顺序使之组成一个最大的整数
     * 链接: https://leetcode.cn/problems/largest-number/
     *
     * 算法标签: 贪心算法(Greedy Algorithm)、自定义排序(Custom Sorting)、字符串处理(String Processing)
     * 时间复杂度: O(n log n) - 排序的时间复杂度
     * 空间复杂度: O(n) - 字符串数组存储
}
```

```
* 是否最优解: 是
*
* 算法思路详解:
* 1. 贪心策略核心: 自定义排序规则, 比较两个数字拼接后的结果,
*   对于数字 a 和 b, 比较 a+b 和 b+a 的大小来决定排列顺序
* 2. 排序优化: 通过自定义比较器实现最优排列
* 3. 结果处理: 拼接排序后的数字并处理前导零
*
* 工程化最佳实践:
* 1. 字符串转换: 准确将数字转换为字符串
* 2. 排序规则: 使用正确的比较规则
* 3. 边界处理: 妥善处理全零数组情况
*
* 实际应用场景:
* 1. 数据展示: 生成最大数字组合
* 2. 密码学: 数字序列优化
* 3. 金融计算: 最大收益数字组合
*/
public static String largestNumber(int[] nums) {
    if (nums == null || nums.length == 0) {
        return "";
    }

    // 将数字转换为字符串数组
    String[] strNums = new String[nums.length];
    for (int i = 0; i < nums.length; i++) {
        strNums[i] = String.valueOf(nums[i]);
    }

    // 自定义排序: 比较 a+b 和 b+a 的大小
    Arrays.sort(strNums, (a, b) -> {
        String order1 = a + b;
        String order2 = b + a;
        return order2.compareTo(order1); // 降序排列
    });

    // 处理前导零的情况
    if (strNums[0].equals("0")) {
        return "0";
    }

    // 拼接结果
    StringBuilder result = new StringBuilder();
```

```

        for (String num : strNums) {
            result.append(num);
        }

        return result.toString();
    }

    /**
     * 题目 2: LeetCode 1665. 完成所有任务的最少初始能量 (Minimum Initial Energy to Finish Tasks)
     * 题目描述: 每个任务有实际消耗和最小初始能量要求, 求完成所有任务的最小初始能量
     * 链接: https://leetcode.cn/problems/minimum-initial-energy-to-finish-tasks/
     *
     * 算法标签: 贪心算法(Greedy Algorithm)、能量管理(Energy Management)、任务调度(Task Scheduling)
     *
     * 时间复杂度: O(n log n) - 排序的时间复杂度
     * 空间复杂度: O(1) - 常数空间
     * 是否最优解: 是
     *
     * 算法思路详解:
     * 1. 贪心策略核心: 按(最小要求 - 实际消耗)降序排序,
     * 优先处理差值大的任务, 这样可以最小化初始能量需求
     * 2. 能量维护: 动态维护当前能量状态
     * 3. 补充策略: 当能量不足时补充所需能量
     *
     * 工程化最佳实践:
     * 1. 排序优化: 使用高效的比较器
     * 2. 能量计算: 准确计算能量补充量
     * 3. 边界处理: 妥善处理空任务数组
     *
     * 实际应用场景:
     * 1. 资源管理: 有限资源下的任务执行
     * 2. 电池优化: 设备电池使用优化
     * 3. 项目管理: 任务资源分配优化
     */
}

public static int minimumEffort(int[][] tasks) {
    if (tasks == null || tasks.length == 0) {
        return 0;
    }

    // 按(最小要求 - 实际消耗)降序排序
    Arrays.sort(tasks, (a, b) -> {
        int diffA = a[1] - a[0]; // 最小要求 - 实际消耗
        int diffB = b[1] - b[0];
    });
}

```

```

        return diffB - diffA; // 降序排列
    });

    int result = 0;
    int currentEnergy = 0;

    for (int[] task : tasks) {
        int actual = task[0];      // 实际消耗
        int minimum = task[1];    // 最小要求

        if (currentEnergy < minimum) {
            // 需要补充能量
            int need = minimum - currentEnergy;
            result += need;
            currentEnergy += need;
        }

        // 完成任务，消耗能量
        currentEnergy -= actual;
    }

    return result;
}

/**
 * 题目 3: LeetCode 991. 坏了的计算器 (Broken Calculator)
 * 题目描述: 计算器只能进行乘 2 和减 1 操作, 求从 X 到 Y 的最小操作次数
 * 链接: https://leetcode.cn/problems/broken-calculator/
 *
 * 算法标签: 贪心算法(Greedy Algorithm)、反向思维(Reverse Thinking)、数学运算(Mathematical Operations)
 * 时间复杂度: O(log Y) – 反向操作次数
 * 空间复杂度: O(1) – 常数空间
 * 是否最优解: 是
 *
 * 算法思路详解:
 * 1. 贪心策略核心: 反向思考, 从 Y 到 X, 只能进行除 2 和加 1 操作,
 *   当 Y > X 时优先除 2, 当 Y < X 时只能加 1
 * 2. 操作优化: 偶数直接除 2, 奇数先加 1 变偶数
 * 3. 终止条件: 当 Y ≤ X 时结束反向操作
 *
 * 工程化最佳实践:
 * 1. 反向思维: 将正向复杂问题转化为反向简单问题

```

\* 2. 操作判断：准确判断除法和加法条件

\* 3. 边界处理：妥善处理  $X \geq Y$  的情况

\*

\* 实际应用场景：

\* 1. 算法优化：复杂问题的反向求解

\* 2. 数学计算：特殊运算规则下的计算优化

\* 3. 游戏开发：受限操作下的最优策略

\*/

```
public static int brokenCalc(int X, int Y) {
```

```
    if (X >= Y) {
```

```
        return X - Y; // 只能减 1 操作
```

```
}
```

```
    int operations = 0;
```

```
    while (Y > X) {
```

```
        operations++;
```

```
        if (Y % 2 == 0) {
```

```
            Y /= 2;
```

```
        } else {
```

```
            Y++;
        }
```

```
}
```

```
    return operations + (X - Y);
```

```
}
```

/\*\*

\* 题目 4: LeetCode 910. 最小差值 II (Smallest Range II)

\* 题目描述：对数组中的每个元素可以加上或减去 K，求调整后数组的最大最小差值

\* 链接：<https://leetcode.cn/problems/smallest-range-ii/>

\*

\* 算法标签：贪心算法(Greedy Algorithm)、数组调整(Array Adjustment)、分割优化(Partition Optimization)

\* 时间复杂度： $O(n \log n)$  – 排序的时间复杂度

\* 空间复杂度： $O(1)$  – 常数空间

\* 是否最优解：是

\*

\* 算法思路详解：

\* 1. 贪心策略核心：排序后寻找最佳分割点，

\* 将数组分为两部分：前一部分加 K，后一部分减 K

\* 2. 分割优化：尝试每个可能的分割点

\* 3. 差值计算：计算调整后的最大最小差值

\*

- \* 工程化最佳实践:
  - \* 1. 排序预处理: 通过排序简化问题
  - \* 2. 分割枚举: 准确枚举所有分割点
  - \* 3. 边界处理: 妥善处理单元素数组
  - \*
- \* 实际应用场景:
  - \* 1. 数据调整: 数组元素范围优化
  - \* 2. 信号处理: 信号幅度调整
  - \* 3. 金融分析: 价格波动范围控制
  - \*/

```

public static int smallestRangeII(int[] nums, int k) {
    if (nums == null || nums.length <= 1) {
        return 0;
    }

    Arrays.sort(nums);
    int n = nums.length;
    int result = nums[n - 1] - nums[0]; // 初始差值

    // 尝试每个可能的分割点
    for (int i = 0; i < n - 1; i++) {
        int high = Math.max(nums[n - 1] - k, nums[i] + k);
        int low = Math.min(nums[0] + k, nums[i + 1] - k);
        result = Math.min(result, high - low);
    }

    return result;
}

/***
 * 题目 5: LeetCode 1526. 形成目标数组的子数组最少增加次数 (Minimum Number of Increments on Subarrays to Form a Target Array)
 * 题目描述: 通过增加连续子数组的值来形成目标数组, 求最少操作次数
 * 链接: https://leetcode.cn/problems/minimum-number-of-increments-on-subarrays-to-form-a-target-array/
 *
 * 算法标签: 贪心算法(Greedy Algorithm)、差值分析(Difference Analysis)、子数组操作(Subarray Operations)
 * 时间复杂度: O(n) - 一次遍历
 * 空间复杂度: O(1) - 常数空间
 * 是否最优解: 是
 *
 * 算法思路详解:

```

```

* 1. 贪心策略核心：相邻元素差值法，  

*    操作次数等于第一个元素的值加上所有正差值之和  

* 2. 差值计算：准确计算相邻元素差值  

* 3. 操作统计：统计所需最少操作次数  

*  

* 工程化最佳实践：  

* 1. 差值处理：准确识别正差值  

* 2. 累计计算：正确累计操作次数  

* 3. 边界处理：妥善处理空数组  

*  

* 实际应用场景：  

* 1. 数组构造：最少操作构造目标数组  

* 2. 资源分配：资源增量分配优化  

* 3. 任务规划：任务增量执行优化  

*/  

public static int minNumberOperations(int[] target) {  

    if (target == null || target.length == 0) {  

        return 0;  

    }  

  

    int operations = target[0]; // 第一个元素需要的操作次数  

  

    for (int i = 1; i < target.length; i++) {  

        if (target[i] > target[i - 1]) {  

            operations += target[i] - target[i - 1];  

        }
    }  

  

    return operations;  

}  

  

/**  

 * 题目 6: LeetCode 1247. 交换字符使得字符串相同 (Minimum Swaps to Make Strings Equal)  

 * 题目描述：通过交换两个字符串对应位置的字符，使得两个字符串相同  

 * 链接: https://leetcode.cn/problems/minimum-swaps-to-make-strings-equal/  

 *  

 * 算法标签：贪心算法(Greedy Algorithm)、字符串匹配(String Matching)、交换优化(Swap Optimization)  

 * 时间复杂度: O(n) - 一次遍历  

 * 空间复杂度: O(1) - 常数空间  

 * 是否最优解：是  

 *  

 * 算法思路详解：

```

```

* 1. 贪心策略核心：统计不匹配的位置类型,
*     有两种不匹配类型: s1[i]='x', s2[i]='y' 和 s1[i]='y', s2[i]='x'
* 2. 交换策略：相同类型的不匹配可以一次交换解决两个，不同类型需要两次交换
* 3. 次数计算：准确计算最少交换次数
*
* 工程化最佳实践：
* 1. 类型统计：准确统计不匹配类型
* 2. 交换计算：正确计算交换次数
* 3. 可行性判断：判断是否可以完成交换
*
* 实际应用场景：
* 1. 字符串处理：字符串匹配优化
* 2. 数据同步：数据一致性维护
* 3. 网络通信：数据包顺序调整
*/
public static int minimumSwap(String s1, String s2) {
    if (s1 == null || s2 == null || s1.length() != s2.length()) {
        return -1;
    }

    int xy = 0, yx = 0; // 统计不匹配类型

    for (int i = 0; i < s1.length(); i++) {
        char c1 = s1.charAt(i);
        char c2 = s2.charAt(i);

        if (c1 == 'x' && c2 == 'y') {
            xy++;
        } else if (c1 == 'y' && c2 == 'x') {
            yx++;
        }
    }

    // 如果总的不匹配数是奇数，无法完成
    if ((xy + yx) % 2 != 0) {
        return -1;
    }

    // 计算最少交换次数
    return xy / 2 + yx / 2 + (xy % 2) * 2;
}

/**/

```

- \* 题目 7: LeetCode 1405. 最长快乐字符串 (Longest Happy String)
- \* 题目描述: 使用给定数量的 a、b、c 字符构造最长字符串, 不能有三个连续相同字符
- \* 链接: <https://leetcode.cn/problems/longest-happy-string/>
- \*
- \* 算法标签: 贪心算法(Greedy Algorithm)、字符串构造(String Construction)、约束满足(Constraint Satisfaction)
- \* 时间复杂度: O(n) – n 是构造字符串长度
- \* 空间复杂度: O(1) – 常数空间
- \* 是否最优解: 是
- \*
- \* 算法思路详解:
- \* 1. 贪心策略核心: 优先使用剩余数量最多的字符,
  - \* 如果连续使用了两个相同字符, 下次使用次多的字符
- \* 2. 字符选择: 使用最大堆维护字符优先级
- \* 3. 约束满足: 确保不出现三个连续相同字符
- \*
- \* 工程化最佳实践:
- \* 1. 优先级维护: 使用堆维护字符数量优先级
- \* 2. 约束检查: 准确检查连续字符约束
- \* 3. 边界处理: 妥善处理字符不足情况
- \*
- \* 实际应用场景:
- \* 1. 密码生成: 满足约束的密码构造
- \* 2. 字符串处理: 约束字符串生成
- \* 3. 游戏开发: 规则字符串构造

```
*/  
public static String longestDiverseString(int a, int b, int c) {  
    // 使用最大堆存储字符和剩余数量  
    PriorityQueue<int[]> maxHeap = new PriorityQueue<>((x, y) -> y[1] - x[1]);  
  
    if (a > 0) maxHeap.offer(new int[] {'a', a});  
    if (b > 0) maxHeap.offer(new int[] {'b', b});  
    if (c > 0) maxHeap.offer(new int[] {'c', c});  
  
    StringBuilder result = new StringBuilder();  
  
    while (!maxHeap.isEmpty()) {  
        int[] first = maxHeap.poll();  
        char char1 = (char) first[0];  
        int count1 = first[1];  
  
        // 检查是否已经连续使用了两个相同字符  
        int len = result.length();  
        if (len >= 2 && result.charAt(len - 1) == char1 && result.charAt(len - 2) == char1)  
            maxHeap.offer(first);  
        else {  
            result.append(char1);  
            if (count1 - 1 > 0)  
                maxHeap.offer(new int[] {char1, count1 - 1});  
        }  
    }  
    return result.toString();  
}
```

```

    if (len >= 2 && result.charAt(len - 1) == char1 && result.charAt(len - 2) == char1) {
        // 需要换一个字符
        if (maxHeap.isEmpty()) {
            break; // 没有其他字符可用
        }

        int[] second = maxHeap.poll();
        char char2 = (char) second[0];
        int count2 = second[1];

        result.append(char2);
        count2--;

        if (count2 > 0) {
            maxHeap.offer(new int[] {char2, count2});
        }
        maxHeap.offer(first); // 把第一个字符放回去
    } else {
        // 可以使用当前字符
        result.append(char1);
        count1--;
    }

    if (count1 > 0) {
        maxHeap.offer(new int[] {char1, count1});
    }
}
}

return result.toString();
}

```

/\*\*

\* 题目 8: LeetCode 1561. 你可以获得的最大硬币数目 (Maximum Number of Coins You Can Get)

\* 题目描述: 三人分硬币, 你总是拿第二多的那堆, 求最大硬币数

\* 链接: <https://leetcode.cn/problems/maximum-number-of-coins-you-can-get/>

\*

\* 算法标签: 贪心算法(Greedy Algorithm)、资源分配(Resource Allocation)、排序优化(Sorting Optimization)

\* 时间复杂度:  $O(n \log n)$  – 排序的时间复杂度

\* 空间复杂度:  $O(1)$  – 常数空间

\* 是否最优解: 是

\*

\* 算法思路详解:

- \* 1. 贪心策略核心：排序后每次取第二大的堆，
- \* 让另外两人拿最大和最小的堆，这样可以最大化自己的收益
- \* 2. 分配策略：按排序后的顺序进行最优分配
- \* 3. 收益计算：累计所获得的硬币数

\*

- \* 工程化最佳实践：

- \* 1. 排序优化：使用高效的排序算法
- \* 2. 分配计算：准确计算分配策略
- \* 3. 边界处理：妥善处理空数组

\*

- \* 实际应用场景：

- \* 1. 资源分配：三人资源最优分配
- \* 2. 游戏策略：公平分配下的最优策略
- \* 3. 经济学：资源分配博弈

\*/

```
public static int maxCoins(int[] piles) {
    if (piles == null || piles.length == 0) {
        return 0;
    }
```

```
    Arrays.sort(piles);
```

```
    int result = 0;
```

```
    int n = piles.length;
```

```
// 每次取第二大的堆（从倒数第二个开始，每隔一个取一个）
```

```
for (int i = n - 2; i >= n / 3; i -= 2) {
    result += piles[i];
}
```

```
return result;
```

```
}
```

/\*\*

\* 题目 9: LeetCode 1689. 十二进制数的最少数目 (Partitioning Into Minimum Number of Decimal Binary Numbers)

\* 题目描述：用最少的十二进制数（只包含 0 和 1）相加得到给定数字

\* 链接：<https://leetcode.cn/problems/partitioning-into-minimum-number-of-deci-binary-numbers/>

\*

\* 算法标签：贪心算法(Greedy Algorithm)、数字分解(Number Decomposition)、位运算(Bit Operations)

\* 时间复杂度：O(n) – 一次遍历

\* 空间复杂度：O(1) – 常数空间

\* 是否最优解: 是

\*

\* 算法思路详解:

\* 1. 贪心策略核心: 最少数量等于数字中最大的数字,

\* 因为每个位置至少需要对应数量的 1

\* 2. 分解策略: 按位分解数字

\* 3. 数量计算: 找到最大数位

\*

\* 工程化最佳实践:

\* 1. 数字处理: 准确处理字符串数字

\* 2. 最大值查找: 高效找到最大数位

\* 3. 边界处理: 妥善处理空字符串

\*

\* 实际应用场景:

\* 1. 数字处理: 数字分解优化

\* 2. 编码理论: 二进制编码优化

\* 3. 密码学: 数字分解应用

\*/

```
public static int minPartitions(String n) {
```

```
    if (n == null || n.isEmpty()) {
```

```
        return 0;
```

```
}
```

```
    int maxDigit = 0;
```

```
    for (char c : n.toCharArray()) {
```

```
        maxDigit = Math.max(maxDigit, c - '0');
```

```
}
```

```
    return maxDigit;
```

```
}
```

/\*\*

\* 题目 10: LeetCode 1710. 卡车上的最大单元数 (Maximum Units on a Truck)

\* 题目描述: 卡车有容量限制, 选择箱子使得总单元数最大

\* 链接: <https://leetcode.cn/problems/maximum-units-on-a-truck/>

\*

\* 算法标签: 贪心算法(Greedy Algorithm)、背包问题变种(Knapsack Variant)、价值优化(Value Optimization)

\* 时间复杂度:  $O(n \log n)$  – 排序的时间复杂度

\* 空间复杂度:  $O(1)$  – 常数空间

\* 是否最优解: 是

\*

\* 算法思路详解:

```
* 1. 贪心策略核心：按单位容量价值（每个箱子的单元数）降序排序，  
*    优先选择单位价值高的箱子以最大化总单元数  
* 2. 容量管理：动态维护剩余容量  
* 3. 选择策略：按排序后的顺序选择箱子  
*  
* 工程化最佳实践：  
* 1. 排序优化：使用高效的比较器  
* 2. 容量控制：准确管理剩余容量  
* 3. 边界处理：妥善处理零容量情况  
*  
* 实际应用场景：  
* 1. 物流运输：货物装载优化  
* 2. 资源分配：有限资源下的价值最大化  
* 3. 投资组合：投资标的优选  
*/
```

```
public static int maximumUnits(int[][] boxTypes, int truckSize) {  
    if (boxTypes == null || boxTypes.length == 0 || truckSize <= 0) {  
        return 0;  
    }  
  
    // 按每个箱子的单元数降序排序  
    Arrays.sort(boxTypes, (a, b) -> b[1] - a[1]);  
  
    int totalUnits = 0;  
    int remainingSize = truckSize;  
  
    for (int[] box : boxTypes) {  
        int number0fBoxes = box[0];  
        int unitsPerBox = box[1];  
  
        int boxesToTake = Math.min(number0fBoxes, remainingSize);  
        totalUnits += boxesToTake * unitsPerBox;  
        remainingSize -= boxesToTake;  
  
        if (remainingSize == 0) {  
            break;  
        }  
    }  
  
    return totalUnits;  
}  
  
// 测试函数
```

```
public static void main(String[] args) {  
    // 测试最大数  
    int[] nums1 = {3, 30, 34, 5, 9};  
    System.out.println("最大数测试: " + largestNumber(nums1)); // 期望: "9534330"  
  
    // 测试坏了的计算器  
    System.out.println("坏了的计算器测试: " + brokenCalc(2, 3)); // 期望: 2  
  
    // 测试最小差值 II  
    int[] nums2 = {1, 3, 6};  
    System.out.println("最小差值 II 测试: " + smallestRangeII(nums2, 3)); // 期望: 3  
  
    // 测试最长快乐字符串  
    System.out.println("最长快乐字符串测试: " + longestDiverseString(1, 1, 7)); // 期望类似:  
    "ccaccbcc"  
  
    // 测试最大硬币数  
    int[] piles = {2, 4, 1, 2, 7, 8};  
    System.out.println("最大硬币数测试: " + maxCoins(piles)); // 期望: 9  
}  
}
```

=====

文件: Code18\_GreedyMathematicalProblems.py

=====

"""  
贪心算法数学相关问题集合 – Python 版本  
包含与数学相关的贪心算法问题

算法专题概览:

本文件包含 10 个与数学相关的贪心算法问题，展示了贪心算法在数学优化中的应用：

1. 数字组合优化（最大数、十-二进制数）
2. 计算器操作优化（坏了的计算器）
3. 差值优化（最小差值、最大单元数）
4. 字符串构造（快乐字符串）
5. 能量管理（任务能量）
6. 硬币分配（最大硬币数）
7. 字符交换（字符串相同）

贪心算法数学应用核心思想:

数学相关的贪心问题通常需要深入理解问题的数学性质，  
通过数学分析找到最优的贪心策略。

工程化最佳实践：

1. 异常处理：检查输入参数的有效性
2. 边界条件：处理空输入、极值等特殊情况
3. 性能优化：选择合适的数学方法和数据结构
4. 可读性：清晰的变量命名和详细注释

"""

```
import heapq
from functools import cmp_to_key
```

```
def largest_number(nums):
```

题目 1: LeetCode 179. 最大数

问题描述：给定一组非负整数，重新排列每个数的顺序使之组成一个最大的整数。

算法思路：自定义排序规则，比较两个数字拼接后的结果

1. 贪心策略：对于任意两个数字 a 和 b，如果  $a+b > b+a$ ，则 a 应该排在 b 前面
2. 使用自定义比较函数进行排序

Args:

nums (List[int]): 非负整数列表

Returns:

str: 重新排列后组成的大整数字符串

时间复杂度:  $O(n \log n)$ ，其中 n 是数字个数，主要是排序的时间复杂度

空间复杂度:  $O(n)$ ，存储字符串的空间

"""

```
# 异常处理：空数字列表
```

```
if not nums:
```

```
    return ""
```

```
# 将数字转换为字符串
```

```
# 时间复杂度:  $O(n)$ 
```

```
str_nums = list(map(str, nums))
```

```
# 自定义排序函数
```

```
def compare(a, b):
```

```
    if a + b > b + a:
```

```
        return -1 # a 应该排在 b 前面
```

```
    elif a + b < b + a:
```

```
        return 1 # b 应该排在 a 前面
```

```

else:
    return 0

# 排序（降序排列）
# 时间复杂度: O(n log n)
str_nums.sort(key=cmp_to_key(compare))

# 处理前导零的情况
if str_nums[0] == "0":
    return "0"

# 拼接结果
# 时间复杂度: O(n)
return ''.join(str_nums)

```

def broken\_calc(start, target):

"""

题目 2: LeetCode 991. 坏了的计算器

问题描述: 有一个坏了的计算器, 只能进行两种操作: 乘 2 和减 1, 求从 start 到 target 的最少操作次数。

算法思路: 反向思考, 从 target 到 start, 只能进行除 2 和加 1 操作

1. 贪心策略: 反向操作, 当 target > start 时, 优先进行除 2 操作
2. 如果 target 是奇数, 先加 1 使其变为偶数

Args:

start (int): 起始数字

target (int): 目标数字

Returns:

int: 最少操作次数

时间复杂度:  $O(\log \text{target})$ , 主要是除 2 操作的次数

空间复杂度:  $O(1)$ , 只使用常数额外空间

"""

# 异常处理: start >= target 的情况

if start >= target:

return start - target # 只能减 1 操作

operations = 0 # 操作次数

current = target # 当前数字

# 反向操作

```

# 时间复杂度: O(log target)
while current > start:
    operations += 1
    if current % 2 == 0:
        current //= 2
    else:
        current += 1

return operations + (start - current)

```

```
def smallest_range_ii(nums, k):
```

```
"""
```

题目 3: LeetCode 910. 最小差值 II

问题描述: 对于数组中的每个元素, 可以选择加上 k 或减去 k, 求变换后数组的最大值与最小值的差的最小值。

算法思路: 排序后寻找最佳分割点

1. 贪心策略: 排序后, 对于前 i 个元素加 k, 后 n-i 个元素减 k
2. 遍历所有可能的分割点, 找到最小差值

Args:

nums (List[int]): 整数数组  
k (int): 可以加减的数值

Returns:

int: 变换后数组的最大值与最小值的差的最小值

时间复杂度:  $O(n \log n)$ , 主要是排序的时间复杂度

空间复杂度:  $O(1)$ , 只使用常数额外空间

```
"""
```

# 异常处理: 数组长度不足

```
if len(nums) <= 1:
    return 0
```

# 排序数组

# 时间复杂度:  $O(n \log n)$

```
nums.sort()
```

```
n = len(nums)
```

```
result = nums[-1] - nums[0] # 初始差值 (都不变换的情况)
```

# 尝试每个可能的分割点

# 时间复杂度:  $O(n)$

```
for i in range(n - 1):
```

```

# 前 i+1 个元素加 k, 后 n-i-1 个元素减 k
high = max(nums[-1] - k, nums[i] + k) # 最大值
low = min(nums[0] + k, nums[i + 1] - k) # 最小值
result = min(result, high - low)

return result

```

```
def min_number_operations(target):
```

```
"""

```

题目 4: LeetCode 1526. 形成目标数组的子数组最少增加次数

问题描述: 给定目标数组, 每次可以选择任意子数组并将其中每个元素增加 1, 求最少操作次数。

算法思路: 相邻元素差值法

1. 贪心策略: 第  $i$  个元素比第  $i-1$  个元素多出的部分需要额外的操作次数
2. 第一个元素需要的操作次数等于其值

Args:

target (List[int]): 目标数组

Returns:

int: 最少操作次数

时间复杂度:  $O(n)$ , 其中  $n$  是数组长度

空间复杂度:  $O(1)$ , 只使用常数额外空间

```
"""

```

# 异常处理: 空目标数组

```
if not target:
```

```
    return 0
```

# 第一个元素需要的操作次数

```
operations = target[0]
```

# 计算后续元素需要的额外操作次数

# 时间复杂度:  $O(n)$

```
for i in range(1, len(target)):
```

```
    if target[i] > target[i - 1]:
```

```
        operations += target[i] - target[i - 1]
```

```
return operations
```

```
def minimum_swap(s1, s2):
```

```
"""

```

题目 5: LeetCode 1247. 交换字符使得字符串相同

问题描述：给定两个长度相等的字符串，求使两个字符串相同的最少交换次数。

算法思路：统计不匹配的位置类型

1. 贪心策略：统计两种不匹配类型(xy 和 yx)的数量
2. 每两个相同类型的不匹配可以通过一次交换解决

Args:

s1 (str): 第一个字符串  
s2 (str): 第二个字符串

Returns:

int: 最少交换次数，无法完成返回-1

时间复杂度:  $O(n)$ ，其中 n 是字符串长度

空间复杂度:  $O(1)$ ，只使用常数额外空间

"""

# 异常处理：字符串长度不匹配

```
if len(s1) != len(s2):  
    return -1
```

```
xy = 0 # s1[i]='x', s2[i]='y' 的不匹配数量  
yx = 0 # s1[i]='y', s2[i]='x' 的不匹配数量
```

# 统计不匹配类型

```
# 时间复杂度:  $O(n)$   
for i in range(len(s1)):  
    c1, c2 = s1[i], s2[i]  
    if c1 == 'x' and c2 == 'y':  
        xy += 1  
    elif c1 == 'y' and c2 == 'x':  
        yx += 1
```

# 如果总的不匹配数是奇数，无法完成

```
if (xy + yx) % 2 != 0:  
    return -1
```

# 计算最少交换次数

```
# 每两个相同类型的不匹配可以通过一次交换解决  
return xy // 2 + yx // 2 + (xy % 2) * 2
```

```
def max_coins(piles):
```

"""

题目 6: LeetCode 1561. 你可以获得的最大硬币数目

问题描述：有  $3n$  堆硬币，每次选择 3 堆，你拿第二多的那堆，求能获得的最大硬币数。

算法思路：排序后每次取第二大的堆

1. 贪心策略：排序后，每次选择最大的 3 堆，你拿第二大的那堆
2. 从大到小每隔一个取一个

Args:

piles (List[int]): 硬币堆数量列表

Returns:

int: 能获得的最大硬币数

时间复杂度:  $O(n \log n)$ , 主要是排序的时间复杂度

空间复杂度:  $O(1)$ , 只使用常数额外空间

"""

# 异常处理: 空硬币堆列表

```
if not piles:  
    return 0
```

# 排序硬币堆

# 时间复杂度:  $O(n \log n)$

```
piles.sort()
```

```
result = 0 # 获得的硬币数
```

```
n = len(piles)
```

# 每次取第二大的堆（从倒数第二个开始，每隔一个取一个）

# 时间复杂度:  $O(n)$

```
for i in range(n - 2, n // 3 - 1, -2):
```

```
    result += piles[i]
```

```
return result
```

```
def min_partitions(n):
```

"""

题目 7: LeetCode 1689. 十二进制数的最少数目

问题描述：十二进制数是只包含 0 和 1 的十进制数，求表示数字  $n$  需要的最少十二进制数数目。

算法思路：最少数量等于数字中最大的数字

1. 贪心策略：每个位置需要的十二进制数数目等于该位置的数字
2. 总数等于所有位置的最大值

Args:

n (str): 表示正整数的字符串

Returns:

int: 最少十-二进制数数目

时间复杂度:  $O(n)$ , 其中  $n$  是字符串长度

空间复杂度:  $O(1)$ , 只使用常数额外空间

"""

# 异常处理: 空字符串

if not n:

    return 0

max\_digit = 0 # 最大数字

# 找到数字中最大的数字

# 时间复杂度:  $O(n)$

for char in n:

    max\_digit = max(max\_digit, int(char))

return max\_digit

def maximum\_units(box\_types, truck\_size):

"""

题目 8: LeetCode 1710. 卡车上的最大单元数

问题描述: 给定箱子类型和卡车容量, 求卡车能装载的最大单元数。

算法思路: 按单位容量价值降序排序

1. 贪心策略: 优先装单位容量价值高的箱子
2. 按每个箱子的单元数降序排序后依次装载

Args:

box\_types (List[List[int]]): 箱子类型列表, 每个元素为[箱子数量, 每箱单元数]

truck\_size (int): 卡车容量 (箱子数量)

Returns:

int: 卡车能装载的最大单元数

时间复杂度:  $O(n \log n)$ , 主要是排序的时间复杂度

空间复杂度:  $O(1)$ , 只使用常数额外空间

"""

# 异常处理: 空箱子类型列表或卡车容量为 0

if not box\_types or truck\_size <= 0:

    return 0

```

# 按每个箱子的单元数降序排序
# 时间复杂度: O(n log n)
box_types.sort(key=lambda x: x[1], reverse=True)

total_units = 0          # 总单元数
remaining_size = truck_size # 剩余容量

# 贪心装载箱子
# 时间复杂度: O(n)
for box in box_types:
    number_of_boxes, units_per_box = box
    boxes_to_take = min(number_of_boxes, remaining_size)
    total_units += boxes_to_take * units_per_box
    remaining_size -= boxes_to_take

    if remaining_size == 0:
        break

return total_units

```

```
def longest_diverse_string(a, b, c):
```

```
"""

```

题目 9: LeetCode 1405. 最长快乐字符串

问题描述: 构造最长的字符串, 使得不包含连续三个相同字符。

算法思路: 优先使用剩余数量最多的字符

1. 贪心策略: 优先使用剩余数量最多的字符
2. 使用最大堆维护字符及其剩余数量
3. 避免连续使用相同字符超过两次

Args:

- a (int): 字符'a'的数量
- b (int): 字符'b'的数量
- c (int): 字符'c'的数量

Returns:

str: 最长的快乐字符串

时间复杂度: O(n), 其中 n 是字符总数

空间复杂度: O(1), 只使用常数额外空间

```
"""

```

# 使用最大堆 (用最小堆存储负值来模拟最大堆)

```
heap = []
```

```

if a > 0:
    heapq.heappush(heap, (-a, 'a'))
if b > 0:
    heapq.heappush(heap, (-b, 'b'))
if c > 0:
    heapq.heappush(heap, (-c, 'c'))

result = [] # 结果字符串

# 构造快乐字符串
# 时间复杂度: O(n)
while heap:
    count1, char1 = heapq.heappop(heap)
    count1 = -count1 # 转换为正数

    # 检查是否已经连续使用了两个相同字符
    if len(result) >= 2 and result[-1] == char1 and result[-2] == char1:
        # 需要换一个字符
        if not heap:
            break # 没有其他字符可用

        count2, char2 = heapq.heappop(heap)
        count2 = -count2

        result.append(char2)
        count2 -= 1

        if count2 > 0:
            heapq.heappush(heap, (-count2, char2))
            heapq.heappush(heap, (-count1, char1)) # 把第一个字符放回去
    else:
        # 可以使用当前字符
        result.append(char1)
        count1 -= 1

        if count1 > 0:
            heapq.heappush(heap, (-count1, char1))

return ''.join(result)

```

def minimum\_effort(tasks):

"""

题目 10: LeetCode 1665. 完成所有任务的最少初始能量

问题描述：给定任务列表，每个任务有最小要求能量和实际消耗能量，求完成所有任务的最少初始能量。

算法思路：按(最小要求 - 实际消耗)降序排序

1. 贪心策略：按(最小要求 - 实际消耗)降序排序
2. 优先处理差值大的任务，这样能更好地利用剩余能量

Args:

tasks (List[List[int]]): 任务列表，每个元素为[实际消耗, 最小要求]

Returns:

int: 完成所有任务的最少初始能量

时间复杂度:  $O(n \log n)$ , 主要是排序的时间复杂度

空间复杂度:  $O(1)$ , 只使用常数额外空间

"""

# 异常处理: 空任务列表

```
if not tasks:  
    return 0
```

# 按(最小要求 - 实际消耗)降序排序

# 时间复杂度:  $O(n \log n)$

```
tasks.sort(key=lambda x: x[1] - x[0], reverse=True)
```

result = 0 # 需要的初始能量

current\_energy = 0 # 当前能量

# 处理每个任务

# 时间复杂度:  $O(n)$

```
for task in tasks:
```

actual, minimum = task

if current\_energy < minimum:

# 需要补充能量

need = minimum - current\_energy

result += need

current\_energy += need

# 完成任务，消耗能量

current\_energy -= actual

```
return result
```

# 测试函数

```

if __name__ == "__main__":
    # 测试最大数
    nums1 = [3, 30, 34, 5, 9]
    print("最大数测试:", largest_number(nums1)) # 期望: "9534330"

    # 测试坏了的计算器
    print("坏了的计算器测试:", broken_calc(2, 3)) # 期望: 2

    # 测试最小差值 II
    nums2 = [1, 3, 6]
    print("最小差值 II 测试:", smallest_range_ii(nums2, 3)) # 期望: 3

    # 测试最大硬币数
    piles = [2, 4, 1, 2, 7, 8]
    print("最大硬币数测试:", max_coins(piles)) # 期望: 9

    # 测试最长快乐字符串
    print("最长快乐字符串测试:", longest_diverse_string(1, 1, 7)) # 期望类似: "ccaccbcc"

```

=====

文件: Code19\_GreedySummaryAndPractice.cpp

=====

```

#include <vector>
#include <algorithm>
#include <queue>
#include <string>
#include <deque>
#include <functional>
#include <numeric>
#include <iostream>

using namespace std;

/***
 * 贪心算法总结与实战练习 - C++版本
 */

```

```

/**
 * 区间调度问题模板
 */
int intervalScheduling(vector<vector<int>>& intervals) {
    if (intervals.empty()) return 0;

```

```

sort(intervals.begin(), intervals.end(), [] (const vector<int>& a, const vector<int>& b) {
    return a[1] < b[1];
});

int count = 1;
int end = intervals[0][1];

for (int i = 1; i < intervals.size(); i++) {
    if (intervals[i][0] >= end) {
        count++;
        end = intervals[i][1];
    }
}

return count;
}

/***
 * 资源分配问题模板
 */
int resourceAllocation(vector<vector<int>>& tasks, int resource) {
    if (tasks.empty() || resource <= 0) return 0;

    sort(tasks.begin(), tasks.end(), [] (const vector<int>& a, const vector<int>& b) {
        return (double) b[1] / b[0] > (double) a[1] / a[0];
    });

    int profit = 0;
    int remaining = resource;

    for (auto& task : tasks) {
        int cost = task[0];
        int value = task[1];

        if (remaining >= cost) {
            profit += value;
            remaining -= cost;
        } else {
            profit += value * remaining / cost;
            break;
        }
    }
}

```

```

    return profit;
}

/***
 * 综合区间调度练习
 */
int comprehensiveIntervalProblems(vector<vector<int>>& intervals) {
    if (intervals.empty()) return 0;

    // 最多不重叠区间数
    sort(intervals.begin(), intervals.end(), [] (const vector<int>& a, const vector<int>& b) {
        return a[1] < b[1];
    });

    int nonOverlapCount = 1;
    int end = intervals[0][1];

    for (int i = 1; i < intervals.size(); i++) {
        if (intervals[i][0] >= end) {
            nonOverlapCount++;
            end = intervals[i][1];
        }
    }

    // 合并重叠区间
    vector<vector<int>> merged;
    merged.push_back(intervals[0]);

    for (int i = 1; i < intervals.size(); i++) {
        vector<int>& last = merged.back();
        if (intervals[i][0] <= last[1]) {
            last[1] = max(last[1], intervals[i][1]);
        } else {
            merged.push_back(intervals[i]);
        }
    }

    return merged.size();
}

/***
 * 综合资源分配练习
 */

```

```

*/
int comprehensiveResourceAllocation(vector<vector<int>>& items, int capacity) {
    if (items.empty() || capacity <= 0) return 0;

    // 策略 1: 按价值排序
    sort(items.begin(), items.end(), [] (const vector<int>& a, const vector<int>& b) {
        return a[1] > b[1];
    });

    int valueStrategy = 0;
    int remaining = capacity;

    for (auto& item : items) {
        if (item[0] <= remaining) {
            valueStrategy += item[1];
            remaining -= item[0];
        }
    }

    // 策略 2: 按价值密度排序
    sort(items.begin(), items.end(), [] (const vector<int>& a, const vector<int>& b) {
        return (double) a[1] / a[0] > (double) b[1] / b[0];
    });

    int densityStrategy = 0;
    remaining = capacity;

    for (auto& item : items) {
        if (item[0] <= remaining) {
            densityStrategy += item[1];
            remaining -= item[0];
        } else {
            densityStrategy += item[1] * remaining / item[0];
            break;
        }
    }

    return max(valueStrategy, densityStrategy);
}

/**
 * 调试技巧: 打印中间结果
*/

```

```

void printIntermediateResults(vector<int>& arr) {
    cout << "原始数组: ";
    for (int num : arr) cout << num << " ";
    cout << endl;

    vector<int> sorted = arr;
    sort(sorted.begin(), sorted.end());

    cout << "排序后数组: ";
    for (int num : sorted) cout << num << " ";
    cout << endl;

    int sum = 0;
    for (int i = 0; i < sorted.size(); i++) {
        sum += sorted[i];
        cout << "第" << (i + 1) << "步选择: " << sorted[i]
            << ", 当前总和: " << sum << endl;
    }
}

```

```

/**
 * 贪心算法核心思想总结
 */
class GreedyPrinciples {
public:
    /**
     * 贪心选择性质验证
     */
    static bool verifyGreedyProperty(vector<int>& arr) {
        // 验证排序后选择是否最优
        vector<int> sorted = arr;
        sort(sorted.begin(), sorted.end());

        int greedySum = 0;
        for (int num : sorted) {
            greedySum += num;
        }

        // 简单验证: 贪心解应该不小于任意其他选择
        return greedySum >= accumulate(arr.begin(), arr.end(), 0);
    }
}

/**

```

```

* 最优子结构验证
*/
static bool verifyOptimalSubstructure(vector<vector<int>>& intervals) {
    if (intervals.empty()) return true;

    // 验证移除一个区间后，剩余问题的最优解是否包含在原问题最优解中
    sort(intervals.begin(), intervals.end(), [] (const vector<int>& a, const vector<int>& b) {
        return a[1] < b[1];
    });

    int originalCount = 1;
    int end = intervals[0][1];

    for (int i = 1; i < intervals.size(); i++) {
        if (intervals[i][0] >= end) {
            originalCount++;
            end = intervals[i][1];
        }
    }

    // 测试移除第一个区间
    vector<vector<int>> subproblem(intervals.begin() + 1, intervals.end());
    int subCount = intervalScheduling(subproblem);

    return originalCount == subCount || originalCount == subCount + 1;
}

// 测试函数
int main() {
    // 测试区间调度
    vector<vector<int>> intervals = {{1, 3}, {2, 4}, {3, 5}, {4, 6}};
    cout << "区间调度结果: " << intervalScheduling(intervals) << endl;

    // 测试资源分配
    vector<vector<int>> items = {{2, 10}, {3, 15}, {5, 20}};
    cout << "资源分配结果: " << resourceAllocation(items, 7) << endl;

    // 测试调试技巧
    vector<int> testArray = {3, 1, 4, 1, 5};
    printIntermediateResults(testArray);

    return 0;
}

```

}

=====

文件: Code19\_GreedySummaryAndPractice.java

=====

```
package class094;
```

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

```
/**
```

```
* 贪心算法总结与实战练习 (Greedy Algorithm Summary and Practice)
```

```
* 包含贪心算法的核心思想、常见题型、解题模板和综合练习
```

```
*
```

```
* 算法标签: 贪心算法(Greedy Algorithm)、算法总结(Algorithm Summary)、实战练习(Practice Exercises)
```

```
* 时间复杂度: 各练习题不同, 详见具体函数注释
```

```
* 空间复杂度: 各练习题不同, 详见具体函数注释
```

```
* 相关内容: 核心思想总结、题型分类、解题模板、调试技巧
```

```
* 贪心算法专题 - 总结与练习
```

```
*/
```

```
public class Code19_GreedySummaryAndPractice {
```

```
/**
```

```
* 贪心算法核心思想总结 (Greedy Algorithm Core Concepts Summary)
```

```
*
```

```
* 1. 贪心选择性质: 每一步都选择当前最优解, 希望通过局部最优达到全局最优
```

```
* 2. 最优子结构: 问题的最优解包含其子问题的最优解
```

```
* 3. 无后效性: 当前选择不会影响后续选择的最优性
```

```
*
```

```
* 适用场景详解:
```

```
* - 活动选择问题 (区间调度): 选择最多的不重叠活动
```

```
* - 分数背包问题: 可分割物品的背包问题
```

```
* - 哈夫曼编码: 数据压缩的最优前缀码
```

```
* - 最短路径问题 (Dijkstra 算法): 单源最短路径
```

```
* - 最小生成树 (Prim、Kruskal 算法): 连通图的最小权重生成树
```

```
*
```

```
* 算法特点:
```

```
* - 高效性: 通常具有较低的时间复杂度
```

```
* - 简洁性: 算法逻辑相对简单
```

```
* - 局限性: 并非所有问题都具有贪心选择性质
```

```
*
```

```
* 证明方法:
```

\* - 交换论证法：通过交换证明贪心选择的最优性

\* - 数学归纳法：证明最优子结构

\*/

/\*\*

\* 常见贪心算法题型分类

\*/

```
public static class GreedyProblemTypes {
```

/\*\*

\* 类型 1：区间调度问题 (Interval Scheduling Problem)

\* 特征：需要选择不重叠的区间或活动

\* 解题模板：按结束时间排序，贪心选择结束最早的

\*

\* 问题特点：

\* - 区间表示：[start, end]形式

\* - 优化目标：选择最多的不重叠区间

\* - 约束条件：区间不能重叠

\*

\* 算法思路：

\* 1. 按结束时间升序排序

\* 2. 贪心选择结束时间最早的区间

\* 3. 排除与已选区间重叠的区间

\*

\* 时间复杂度： $O(n \log n)$  – 排序时间

\* 空间复杂度： $O(1)$  – 常数空间

\* 是否最优解：是

\*

\* 实际应用：

\* - 会议室调度

\* - 任务安排

\* - 广告投放

\*/

```
public static int intervalScheduling(int[][] intervals) {
```

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

```
    Arrays.sort(intervals, (a, b) -> a[1] - b[1]);
```

```
    int count = 1;
```

```
    int end = intervals[0][1];
```

```
    for (int i = 1; i < intervals.length; i++) {
```

```
        if (intervals[i][0] >= end) {
```

```

        count++;
        end = intervals[i][1];
    }
}

return count;
}

/***
 * 类型 2: 资源分配问题 (Resource Allocation Problem)
 * 特征: 有限资源分配给多个任务, 最大化收益
 * 解题模板: 按单位资源收益排序, 贪心分配
 *
 * 问题特点:
 * - 资源约束: 总量有限的资源
 * - 任务属性: 每个任务有成本和收益
 * - 优化目标: 最大化总收益
 *
 * 算法思路:
 * 1. 计算每个任务的价值密度 (收益/成本)
 * 2. 按价值密度降序排序
 * 3. 贪心选择价值密度最高的任务
 *
 * 时间复杂度: O(n log n) - 排序时间
 * 空间复杂度: O(1) - 常数空间
 * 是否最优解: 对于分数背包是最优解
 *
 * 实际应用:
 * - 投资组合
 * - 任务调度
 * - 预算分配
 */

public static int resourceAllocation(int[][] tasks, int resource) {
    if (tasks == null || tasks.length == 0 || resource <= 0) return 0;

    Arrays.sort(tasks, (a, b) -> Double.compare(
        (double) b[1] / b[0], (double) a[1] / a[0]
    ));

    int profit = 0;
    int remaining = resource;

    for (int[] task : tasks) {

```

```

        int cost = task[0];
        int value = task[1];

        if (remaining >= cost) {
            profit += value;
            remaining -= cost;
        } else {
            profit += value * remaining / cost;
            break;
        }
    }

    return profit;
}

/***
 * 类型 3: 路径优化问题 (Path Optimization Problem)
 * 特征: 在路径上选择最优停留点或加油点
 * 解题模板: 维护当前可达范围, 贪心选择最远可达点
 *
 * 问题特点:
 * - 路径表示: 一系列位置点
 * - 资源约束: 每次移动消耗资源
 * - 优化目标: 最小化停留次数
 *
 * 算法思路:
 * 1. 维护当前可达范围
 * 2. 贪心选择能到达的最远点
 * 3. 更新可达范围和停留次数
 *
 * 时间复杂度: O(n) - 一次遍历
 * 空间复杂度: O(1) - 常数空间
 * 是否最优解: 是
 *
 * 实际应用:
 * - 加油站选址
 * - 网络路由
 * - 旅行规划
 */
public static int pathOptimization(int[] positions, int range) {
    if (positions == null || positions.length == 0) return 0;

    int count = 0;

```

```

int currentPos = 0;
int farthest = 0;

for (int i = 0; i < positions.length; i++) {
    if (positions[i] > farthest) {
        if (currentPos >= positions.length - 1) break;
        count++;
        currentPos = farthest;
        if (currentPos >= positions.length - 1) break;
    }
    farthest = Math.max(farthest, positions[i] + range);
}

return count;
}

/***
 * 类型 4: 字符串重构问题 (String Reconstruction Problem)
 * 特征: 重新排列字符串满足特定条件
 * 解题模板: 使用单调栈或自定义排序
 *
 * 问题特点:
 * - 字符约束: 字符频率和排列限制
 * - 优化目标: 满足特定条件的字符串
 * - 约束条件: 如避免连续相同字符
 *
 * 算法思路:
 * 1. 统计字符频率
 * 2. 使用优先队列维护字符优先级
 * 3. 贪心选择满足约束的字符
 *
 * 时间复杂度: O(n log k) - k 是不同字符数
 * 空间复杂度: O(k) - 字符频率存储
 * 是否最优解: 是
 *
 * 实际应用:
 * - 密码生成
 * - 数据编码
 * - 字符串处理
 */
public static String stringReconstruction(String s, int k) {
    if (s == null || s.isEmpty()) return "";

```

```

// 统计字符频率
int[] freq = new int[26];
for (char c : s.toCharArray()) {
    freq[c - 'a']++;
}

// 构建结果字符串
StringBuilder result = new StringBuilder();
while (result.length() < s.length()) {
    // 选择当前可用的最大字符
    for (int i = 25; i >= 0; i--) {
        if (freq[i] > 0) {
            // 检查是否可以添加（避免连续 k 个相同字符）
            int len = result.length();
            if (len >= k - 1) {
                boolean valid = true;
                for (int j = 1; j < k; j++) {
                    if (result.charAt(len - j) != (char) ('a' + i)) {
                        valid = false;
                        break;
                    }
                }
                if (valid) continue;
            }
            result.append((char) ('a' + i));
            freq[i]--;
            break;
        }
    }
}

return result.toString();
}

}

/***
 * 贪心算法解题模板
 */
public static class GreedyTemplates {

    /**
     * 模板 1：排序 + 贪心选择 (Sort + Greedy Selection Template)

```

- \* 适用：需要按某种规则排序后选择的问题
- \*
- \* 模板特点：
- \* - 预处理：通过排序建立选择顺序
- \* - 选择策略：按排序后的顺序贪心选择
- \* - 适用问题：具有明确排序规则的问题
- \*
- \* 实现步骤：
- \* 1. 根据问题特点设计排序规则
- \* 2. 对数据进行排序
- \* 3. 按排序顺序进行贪心选择
- \*
- \* 时间复杂度： $O(n \log n)$  - 排序时间
- \* 空间复杂度： $O(1)$  - 常数空间
- \*
- \* 典型应用：
- \* - 区间调度
- \* - 资源分配
- \* - 任务选择
- \*/

```

public static void templateSortAndGreedy(int[] arr) {
    // 1. 排序数组
    Arrays.sort(arr);

    // 2. 贪心选择
    int result = 0;
    for (int i = 0; i < arr.length; i++) {
        // 根据问题需求进行选择
        if /* 满足选择条件 */ true) {
            result += arr[i];
        }
    }
}

/**
 * 模板 2：堆 + 贪心选择 (Heap + Greedy Selection Template)
 * 适用：需要动态维护最优选择的问题
 *
 * 模板特点：
 * - 数据结构：使用堆维护元素优先级
 * - 动态选择：实时获取最优元素
 * - 适用问题：需要动态调整选择策略
 *

```

```

* 实现步骤:
* 1. 根据问题特点构建最大堆或最小堆
* 2. 将元素加入堆中
* 3. 动态获取并处理最优元素
*
* 时间复杂度: O(n log n) - 堆操作
* 空间复杂度: O(n) - 堆存储
*
* 典型应用:
* - 任务调度
* - 资源管理
* - 优先队列
*/
public static void templateHeapAndGreedy(int[] arr) {
    // 1. 构建堆 (最大堆或最小堆)
    PriorityQueue<Integer> heap = new PriorityQueue<>();
    for (int num : arr) {
        heap.offer(num);
    }

    // 2. 贪心选择
    int result = 0;
    while (!heap.isEmpty()) {
        int current = heap.poll();
        // 根据问题需求进行选择
        result += current;
    }
}

/**
* 模板 3: 双指针 + 贪心选择 (Two Pointers + Greedy Selection Template)
* 适用: 需要同时处理两个序列的问题
*
* 模板特点:
* - 双序列处理: 同时遍历两个有序序列
* - 指针移动: 根据匹配条件移动指针
* - 适用问题: 序列匹配和优化问题
*
* 实现步骤:
* 1. 对两个序列进行排序
* 2. 初始化双指针
* 3. 根据匹配条件移动指针
*

```

```
* 时间复杂度: O(n log n) - 排序时间
* 空间复杂度: O(1) - 常数空间
*
* 典型应用:
* - 序列匹配
* - 任务分配
* - 优化选择
*/
public static void templateTwoPointers(int[] arr1, int[] arr2) {
    // 1. 排序两个数组
    Arrays.sort(arr1);
    Arrays.sort(arr2);

    // 2. 双指针遍历
    int i = 0, j = 0;
    int result = 0;

    while (i < arr1.length && j < arr2.length) {
        if /* 满足匹配条件 */ arr1[i] <= arr2[j]) {
            result++;
            i++;
            j++;
        } else {
            j++;
        }
    }
}

/**
 * 综合实战练习题目 (Comprehensive Practice Problems)
 * 通过综合题目加深对贪心算法的理解和应用
*
* 练习目标:
* - 掌握不同类型问题的解法
* - 理解算法间的联系和区别
* - 提高问题分析和解决能力
*
* 练习方法:
* - 分析问题特点
* - 选择合适模板
* - 实现算法细节
* - 验证结果正确性
```

```
*/  
  
/**  
 * 练习 1: 综合区间调度 (Comprehensive Interval Scheduling)  
 * 结合多种区间问题的综合解法  
 *  
 * 问题描述:  
 * - 计算最多不重叠区间数  
 * - 计算需要移除的最少区间数  
 * - 合并所有重叠区间  
 *  
 * 算法思路:  
 * 1. 按结束时间排序解决区间选择  
 * 2. 通过总数减去选择数得到移除数  
 * 3. 遍历合并重叠区间  
 *  
 * 时间复杂度: O(n log n) - 排序和遍历  
 * 空间复杂度: O(n) - 存储合并结果  
 * 是否最优解: 是  
 *  
 * 实际应用:  
 * - 会议安排优化  
 * - 任务调度管理  
 * - 资源分配规划  
 */
```

```
public static int comprehensiveIntervalProblems(int[][] intervals) {  
    if (intervals == null || intervals.length == 0) return 0;  
  
    // 问题 1: 最多不重叠区间数  
    Arrays.sort(intervals, (a, b) -> a[1] - b[1]);  
    int nonOverlapCount = 1;  
    int end = intervals[0][1];  
  
    for (int i = 1; i < intervals.length; i++) {  
        if (intervals[i][0] >= end) {  
            nonOverlapCount++;  
            end = intervals[i][1];  
        }  
    }  
  
    // 问题 2: 需要移除的最少区间数  
    int removeCount = intervals.length - nonOverlapCount;
```

```

// 问题 3: 合并所有重叠区间
List<int[]> merged = new ArrayList<>();
merged.add(intervals[0]);

for (int i = 1; i < intervals.length; i++) {
    int[] last = merged.get(merged.size() - 1);
    if (intervals[i][0] <= last[1]) {
        last[1] = Math.max(last[1], intervals[i][1]);
    } else {
        merged.add(intervals[i]);
    }
}

return merged.size(); // 返回合并后的区间数
}

/**
 * 练习 2: 资源分配综合问题 (Comprehensive Resource Allocation)
 * 多种资源分配策略的比较
 *
 * 问题描述:
 * - 按价值排序策略
 * - 按价值密度排序策略
 * - 按重量排序策略
 *
 * 算法思路:
 * 1. 实现三种不同排序策略
 * 2. 分别计算每种策略的结果
 * 3. 返回最优策略的结果
 *
 * 时间复杂度: O(n log n) - 排序时间
 * 空间复杂度: O(1) - 常数空间
 * 是否最优解: 是
 *
 * 实际应用:
 * - 投资组合优化
 * - 任务分配策略
 * - 预算管理方案
 */
public static int comprehensiveResourceAllocation(int[][] items, int capacity) {
    if (items == null || items.length == 0 || capacity <= 0) return 0;

    // 策略 1: 按价值排序 (0-1 背包贪心近似)

```

```

Arrays.sort(items, (a, b) -> b[1] - a[1]);
int valueStrategy = 0;
int remaining = capacity;

for (int[] item : items) {
    if (item[0] <= remaining) {
        valueStrategy += item[1];
        remaining -= item[0];
    }
}

// 策略 2: 按价值密度排序 (分数背包最优)
Arrays.sort(items, (a, b) ->
    Double.compare((double) b[1] / b[0], (double) a[1] / a[0])
);

int densityStrategy = 0;
remaining = capacity;

for (int[] item : items) {
    if (item[0] <= remaining) {
        densityStrategy += item[1];
        remaining -= item[0];
    } else {
        densityStrategy += item[1] * remaining / item[0];
        break;
    }
}

// 策略 3: 按重量排序 (尽量多装)
Arrays.sort(items, (a, b) -> a[0] - b[0]);
int weightStrategy = 0;
remaining = capacity;

for (int[] item : items) {
    if (item[0] <= remaining) {
        weightStrategy += item[1];
        remaining -= item[0];
    }
}

// 返回三种策略中的最大值
return Math.max(valueStrategy, Math.max(densityStrategy, weightStrategy));

```

```
}
```

```
/**
```

```
* 练习 3: 字符串处理综合问题 (Comprehensive String Processing)
```

```
* 多种字符串重构和优化问题
```

```
*
```

```
* 问题描述:
```

```
* - 去除重复字母并使字典序最小
```

```
* - 重构字符串避免连续 k 个相同字符
```

```
*
```

```
* 算法思路:
```

```
* 1. 使用单调栈去除重复字母
```

```
* 2. 使用优先队列重构字符串
```

```
* 3. 确保满足连续字符约束
```

```
*
```

```
* 时间复杂度: O(n log k) - k 是不同字符数
```

```
* 空间复杂度: O(k) - 字符频率存储
```

```
* 是否最优解: 是
```

```
*
```

```
* 实际应用:
```

```
* - 密码生成优化
```

```
* - 数据编码处理
```

```
* - 字符串规范化
```

```
*/
```

```
public static String comprehensiveStringProblems(String s, int k) {
```

```
    if (s == null || s.isEmpty()) return "";
```

```
// 问题 1: 去除重复字母并使字典序最小
```

```
int[] lastPos = new int[26];
```

```
for (int i = 0; i < s.length(); i++) {
```

```
    lastPos[s.charAt(i) - 'a'] = i;
```

```
}
```

```
boolean[] visited = new boolean[26];
```

```
Deque<Character> stack = new ArrayDeque<>();
```

```
for (int i = 0; i < s.length(); i++) {
```

```
    char c = s.charAt(i);
```

```
    if (visited[c - 'a']) continue;
```

```
    while (!stack.isEmpty() && stack.peek() > c &&
```

```
        lastPos[stack.peek() - 'a'] > i) {
```

```
        visited[stack.pop() - 'a'] = false;
```

```

    }

    stack.push(c);
    visited[c - 'a'] = true;
}

StringBuilder result = new StringBuilder();
while (!stack.isEmpty()) {
    result.append(stack.pop());
}
String removeDuplicate = result.reverse().toString();

// 问题2：重构字符串避免连续 k 个相同字符
int[] freq = new int[26];
for (char c : removeDuplicate.toCharArray()) {
    freq[c - 'a']++;
}

PriorityQueue<Character> maxHeap = new PriorityQueue<>(
    (a, b) -> freq[b - 'a'] - freq[a - 'a']
);

for (char c = 'a'; c <= 'z'; c++) {
    if (freq[c - 'a'] > 0) {
        maxHeap.offer(c);
    }
}

StringBuilder finalResult = new StringBuilder();
while (!maxHeap.isEmpty()) {
    List<Character> temp = new ArrayList<>();

    for (int i = 0; i < k && !maxHeap.isEmpty(); i++) {
        char current = maxHeap.poll();
        finalResult.append(current);
        freq[current - 'a']--;
    }

    if (freq[current - 'a'] > 0) {
        temp.add(current);
    }
}

for (char c : temp) {
}

```

```
        maxHeap.offer(c);
    }

}

return finalResult.toString();
}

/***
 * 贪心算法调试技巧 (Greedy Algorithm Debugging Techniques)
 * 提高算法实现正确性和效率的方法
 *
 * 调试目标:
 * - 验证算法正确性
 * - 分析性能瓶颈
 * - 优化实现细节
 *
 * 调试方法:
 * - 中间结果打印
 * - 测试用例验证
 * - 性能分析评估
 */
public static class DebuggingTechniques {

    /**
     * 技巧 1: 打印中间结果 (Print Intermediate Results)
     * 在关键步骤打印变量值, 帮助理解算法执行过程
     *
     * 调试价值:
     * - 追踪算法执行流程
     * - 验证中间状态正确性
     * - 发现逻辑错误位置
     *
     * 实施方法:
     * 1. 在关键步骤添加打印语句
     * 2. 输出关键变量的值
     * 3. 分析输出结果
     *
     * 注意事项:
     * - 避免打印过多信息
     * - 使用清晰的输出格式
     * - 及时移除调试代码
     */
    public static void printIntermediateResults(int[] arr) {
```

```

System.out.println("原始数组: " + Arrays.toString(arr));

// 排序后
int[] sorted = arr.clone();
Arrays.sort(sorted);
System.out.println("排序后数组: " + Arrays.toString(sorted));

// 贪心选择过程
int sum = 0;
for (int i = 0; i < sorted.length; i++) {
    sum += sorted[i];
    System.out.println("第" + (i + 1) + "步选择: " + sorted[i] +
        ", 当前总和: " + sum);
}
}

/**
 * 技巧 2: 验证贪心选择正确性 (Verify Greedy Choice Correctness)
 * 通过小规模测试用例验证算法正确性
 *
 * 验证方法:
 * - 设计边界测试用例
 * - 构造典型问题实例
 * - 比较期望与实际结果
 *
 * 测试策略:
 * 1. 边界条件测试
 * 2. 典型场景测试
 * 3. 异常情况测试
 *
 * 验证要点:
 * - 确保算法逻辑正确
 * - 验证结果符合预期
 * - 检查边界处理
 */
public static boolean verifyGreedyChoice(int[][] testCases) {
    for (int[] testCase : testCases) {
        int expected = testCase[0]; // 期望结果
        int[] input = Arrays.copyOfRange(testCase, 1, testCase.length);

        // 执行贪心算法
        int actual = greedyAlgorithm(input);
    }
}

```

```
        if (actual != expected) {
            System.out.println("测试失败: 输入=" + Arrays.toString(input) +
                               ", 期望=" + expected + ", 实际=" + actual);
            return false;
        }
    }
    return true;
}

private static int greedyAlgorithm(int[] arr) {
    // 示例贪心算法实现
    Arrays.sort(arr);
    return arr.length > 0 ? arr[arr.length - 1] : 0;
}

/**
 * 技巧 3: 性能分析 (Performance Analysis)
 * 分析算法的时间复杂度和空间复杂度
 *
 * 分析内容:
 * - 时间复杂度评估
 * - 空间复杂度评估
 * - 实际运行时间
 *
 * 分析方法:
 * 1. 理论复杂度计算
 * 2. 实际运行时间测量
 * 3. 瓶颈识别和优化
 *
 * 优化建议:
 * - 选择高效数据结构
 * - 减少不必要的计算
 * - 优化算法实现
 */
public static void analyzePerformance(int[] arr) {
    long startTime = System.nanoTime();

    // 执行算法
    Arrays.sort(arr);
    int result = 0;
    for (int num : arr) {
        result += num;
    }
}
```

```

        long endTime = System.nanoTime();
        long duration = endTime - startTime;

        System.out.println("算法执行时间: " + duration + " 纳秒");
        System.out.println("输入规模: " + arr.length);
        System.out.println("时间复杂度: O(n log n)");
        System.out.println("空间复杂度: O(1)");

    }

}

// 测试函数
public static void main(String[] args) {
    // 测试综合区间调度
    int[][] intervals = {{1, 3}, {2, 4}, {3, 5}, {4, 6}};
    System.out.println("综合区间调度结果: " + comprehensiveIntervalProblems(intervals));

    // 测试资源分配综合问题
    int[][] items = {{2, 10}, {3, 15}, {5, 20}};
    System.out.println("资源分配结果: " + comprehensiveResourceAllocation(items, 7));

    // 测试调试技巧
    int[] testArray = {3, 1, 4, 1, 5};
    DebuggingTechniques.printIntermediateResults(testArray);
}
}

```

=====

文件: Code19\_GreedySummaryAndPractice.py

=====

```

"""
贪心算法总结与实战练习 - Python 版本
包含贪心算法的核心思想、常见题型、解题模板和综合练习
"""

```

```

import heapq
from typing import List, Tuple
from functools import cmp_to_key

class GreedySummary:
    """
    贪心算法核心思想总结

```

```

"""
"""

@staticmethod
def greedy_principles():
    """
    贪心算法的三个核心性质：
    1. 贪心选择性质：每一步都选择当前最优解
    2. 最优子结构：问题的最优解包含其子问题的最优解
    3. 无后效性：当前选择不会影响后续选择的最优性
    """

    principles = {
        "贪心选择性质": "每一步都选择当前最优解，希望通过局部最优达到全局最优",
        "最优子结构": "问题的最优解包含其子问题的最优解",
        "无后效性": "当前选择不会影响后续选择的最优性"
    }
    return principles


@staticmethod
def applicable_scenarios():
    """
    贪心算法适用场景
    """

    scenarios = {
        "区间调度问题": "活动选择、会议安排等",
        "资源分配问题": "分数背包、任务调度等",
        "路径优化问题": "最短路径、加油站问题等",
        "字符串处理": "字典序最小、字符重组等",
        "数学优化问题": "最大数、最小差值等"
    }
    return scenarios


class GreedyTemplates:
    """
    贪心算法解题模板
    """

    @staticmethod
    def template_sort_and_greedy(arr: List[int]) -> int:
        """
        模板 1：排序 + 贪心选择
        适用：需要按某种规则排序后选择的问题
        """

        # 1. 排序数组

```

```
arr.sort()

# 2. 贪心选择
result = 0
for i in range(len(arr)):
    # 根据问题需求进行选择
    if True: # 满足选择条件
        result += arr[i]

return result

@staticmethod
def template_heap_and_greedy(arr: List[int]) -> int:
    """
    模板 2: 堆 + 贪心选择
    适用: 需要动态维护最优选择的问题
    """

    # 1. 构建堆
    heap = arr[:]
    heapq.heapify(heap)

    # 2. 贪心选择
    result = 0
    while heap:
        current = heapq.heappop(heap)
        result += current

    return result

@staticmethod
def template_two_pointers(arr1: List[int], arr2: List[int]) -> int:
    """
    模板 3: 双指针 + 贪心选择
    适用: 需要同时处理两个序列的问题
    """

    # 1. 排序两个数组
    arr1.sort()
    arr2.sort()

    # 2. 双指针遍历
    i, j = 0, 0
    result = 0
```

```
while i < len(arr1) and j < len(arr2):
    if arr1[i] <= arr2[j]: # 满足匹配条件
        result += 1
        i += 1
        j += 1
    else:
        j += 1

return result

class ComprehensivePractice:
    """
    综合实战练习
    """

    @staticmethod
    def comprehensive_interval_problems(intervals: List[List[int]]) -> int:
        """
        练习 1: 综合区间调度
        结合多种区间问题的综合解法
        """

        if not intervals:
            return 0

        # 问题 1: 最多不重叠区间数
        intervals.sort(key=lambda x: x[1])
        non_overlap_count = 1
        end = intervals[0][1]

        for i in range(1, len(intervals)):
            if intervals[i][0] >= end:
                non_overlap_count += 1
                end = intervals[i][1]

        # 问题 2: 合并所有重叠区间
        merged = [intervals[0]]
        for i in range(1, len(intervals)):
            last = merged[-1]
            if intervals[i][0] <= last[1]:
                last[1] = max(last[1], intervals[i][1])
            else:
                merged.append(intervals[i])

        return non_overlap_count
```

```
return len(merged) # 返回合并后的区间数

@staticmethod
def comprehensive_resource_allocation(items: List[List[int]], capacity: int) -> int:
    """
    练习 2: 资源分配综合问题
    多种资源分配策略的比较
    """
    if not items or capacity <= 0:
        return 0

    # 策略 1: 按价值排序 (0-1 背包贪心近似)
    items_by_value = sorted(items, key=lambda x: x[1], reverse=True)
    value_strategy = 0
    remaining = capacity

    for item in items_by_value:
        cost, value = item
        if cost <= remaining:
            value_strategy += value
            remaining -= cost

    # 策略 2: 按价值密度排序 (分数背包最优)
    items_by_density = sorted(items, key=lambda x: x[1] / x[0], reverse=True)
    density_strategy = 0
    remaining = capacity

    for item in items_by_density:
        cost, value = item
        if cost <= remaining:
            density_strategy += value
            remaining -= cost
        else:
            density_strategy += value * remaining / cost
            break

    # 策略 3: 按重量排序 (尽量多装)
    items_by_weight = sorted(items, key=lambda x: x[0])
    weight_strategy = 0
    remaining = capacity

    for item in items_by_weight:
        cost, value = item
        if cost <= remaining:
            weight_strategy += value
            remaining -= cost
        else:
            weight_strategy += value * remaining / cost
            break
```

```

        if cost <= remaining:
            weight_strategy += value
            remaining -= cost

    # 返回三种策略中的最大值
    return max(value_strategy, density_strategy, weight_strategy)

@staticmethod
def comprehensive_string_problems(s: str, k: int) -> str:
    """
    练习 3: 字符串处理综合问题
    多种字符串重构和优化问题
    """

    if not s:
        return ""

    # 问题 1: 去除重复字母并使字典序最小
    last_pos = {}
    for i, char in enumerate(s):
        last_pos[char] = i

    visited = set()
    stack = []

    for i, char in enumerate(s):
        if char in visited:
            continue

        while stack and stack[-1] > char and last_pos[stack[-1]] > i:
            visited.remove(stack.pop())

        stack.append(char)
        visited.add(char)

    remove_duplicate = ''.join(stack)

    # 问题 2: 重构字符串避免连续 k 个相同字符
    freq = {}
    for char in remove_duplicate:
        freq[char] = freq.get(char, 0) + 1

    # 使用最大堆（用最小堆存储负值模拟）
    heap = []

```

```
for char, count in freq.items():
    heapq.heappush(heap, (-count, char))

result = []
while heap:
    temp = []

    for _ in range(min(k, len(heap))):
        if not heap:
            break
        count, char = heapq.heappop(heap)
        result.append(char)
        count = -count - 1 # 转换为正数并减1

        if count > 0:
            temp.append((-count, char))

    for item in temp:
        heapq.heappush(heap, item)

return ''.join(result)
```

```
class DebuggingTechniques:

    """
    贪心算法调试技巧
    """

    @staticmethod
    def print_intermediate_results(arr: List[int]):
        """
        技巧 1: 打印中间结果
        在关键步骤打印变量值，帮助理解算法执行过程
        """

        print(f"原始数组: {arr}")

        # 排序后
        sorted_arr = sorted(arr)
        print(f"排序后数组: {sorted_arr}")

        # 贪心选择过程
        total = 0
        for i, num in enumerate(sorted_arr):
            total += num
```

```
print(f"第{i+1}步选择: {num}, 当前总和: {total}")

@staticmethod
def verify_greedy_choice(test_cases: List[Tuple[int, List[int]]]) -> bool:
    """
    技巧 2: 验证贪心选择正确性
    通过小规模测试用例验证算法正确性
    """
    for expected, input_arr in test_cases:
        # 执行贪心算法
        actual = DebuggingTechniques.greedy_algorithm(input_arr)

        if actual != expected:
            print(f"测试失败: 输入={input_arr}, 期望={expected}, 实际={actual}")
            return False

    print("所有测试用例通过!")
    return True

@staticmethod
def greedy_algorithm(arr: List[int]) -> int:
    """
    示例贪心算法实现
    """
    if not arr:
        return 0
    return max(arr)  # 简单示例: 选择最大值

@staticmethod
def analyze_performance(arr: List[int]):
    """
    技巧 3: 性能分析
    分析算法的时间复杂度和空间复杂度
    """
    import time

    start_time = time.time()

    # 执行算法
    sorted_arr = sorted(arr)
    result = sum(sorted_arr)

    end_time = time.time()
```

```
duration = end_time - start_time

print(f"算法执行时间: {duration:.6f} 秒")
print(f"输入规模: {len(arr)}")
print("时间复杂度: O(n log n)")
print("空间复杂度: O(n)")

def main():
    """
    主测试函数
    """

    # 测试贪心算法核心思想
    print("== 贪心算法核心思想 ==")
    principles = GreedySummary.greedy_principles()
    for key, value in principles.items():
        print(f"{key}: {value}")

    # 测试综合区间调度
    print("\n== 综合区间调度练习 ==")
    intervals = [[1, 3], [2, 4], [3, 5], [4, 6]]
    result = ComprehensivePractice.comprehensive_interval_problems(intervals)
    print(f"区间调度结果: {result}")

    # 测试资源分配综合问题
    print("\n== 资源分配综合练习 ==")
    items = [[2, 10], [3, 15], [5, 20]]
    result = ComprehensivePractice.comprehensive_resource_allocation(items, 7)
    print(f"资源分配结果: {result}")

    # 测试调试技巧
    print("\n== 调试技巧演示 ==")
    test_array = [3, 1, 4, 1, 5]
    DebuggingTechniques.print_intermediate_results(test_array)

    # 测试贪心选择验证
    print("\n== 贪心选择验证 ==")
    test_cases = [
        (5, [1, 2, 3, 4, 5]),  # 期望结果, 输入数组
        (3, [1, 2, 3]),
        (0, [])
    ]
    DebuggingTechniques.verify_greedy_choice(test_cases)
```

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

文件: TestAllGreedyAlgorithms.java

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

package class094;

import java.util.Arrays;

/**
 * 贪心算法综合测试类 (Greedy Algorithm Comprehensive Test Class)
 * 验证所有贪心算法实现的正确性
 *
 * 测试目标:
 * - 验证基础贪心算法实现
 * - 验证高级贪心算法实现
 * - 验证数学相关贪心算法实现
 *
 * 测试方法:
 * - 使用预定义测试用例
 * - 比较期望结果与实际结果
 * - 输出测试通过情况
 *
 * 算法标签: 贪心算法(Greedy Algorithm)、综合测试(Comprehensive Testing)、算法验证(Algorithm Verification)
 */
public class TestAllGreedyAlgorithms {

    public static void main(String[] args) {
        System.out.println("==> 贪心算法综合测试开始 ==>\n");

        // 测试基础贪心算法
        testBasicGreedyAlgorithms();

        // 测试高级贪心算法
        testAdvancedGreedyAlgorithms();

        // 测试数学相关贪心算法
        testMathematicalGreedyAlgorithms();

        System.out.println("\n==> 贪心算法综合测试完成 ==>");
    }
}
```

```
}
```

```
/**
```

```
* 测试基础贪心算法 (Test Basic Greedy Algorithms)  
* 验证分发饼干、买卖股票、跳跃游戏等基础贪心算法的正确性
```

```
*
```

```
* 测试内容:
```

```
* 1. 分发饼干: 验证孩子满足数计算  
* 2. 买卖股票: 验证最大利润计算  
* 3. 跳跃游戏: 验证可达性判断
```

```
*
```

```
* 算法特点:
```

```
* - 时间复杂度:  $O(n \log n)$  - 主要消耗在排序  
* - 空间复杂度:  $O(1)$  - 常数额外空间  
* - 测试方法: 使用预定义用例验证结果
```

```
*/
```

```
private static void testBasicGreedyAlgorithms() {
```

```
    System.out.println("--- 基础贪心算法测试 ---");
```

```
    // 测试分发饼干 (使用独立实现)
```

```
    // 算法: 贪心匹配最小胃口和最小饼干
```

```
    int[] g = {1, 2, 3}; // 孩子胃口
```

```
    int[] s = {1, 1}; // 饼干尺寸
```

```
    int result = findContentChildren(g, s);
```

```
    System.out.println("分发饼干测试: " + (result == 1 ? "✓ 通过" : "✗ 失败"));
```

```
    // 测试买卖股票 (使用独立实现)
```

```
    // 算法: 收集所有正收益交易
```

```
    int[] prices = {7, 1, 5, 3, 6, 4}; // 股票价格
```

```
    int profit = maxProfit(prices);
```

```
    System.out.println("买卖股票测试: " + (profit == 7 ? "✓ 通过" : "✗ 失败"));
```

```
    // 测试跳跃游戏 (使用独立实现)
```

```
    // 算法: 维护最远可达位置
```

```
    int[] nums = {2, 3, 1, 1, 4}; // 跳跃距离
```

```
    boolean canJump = canJump(nums);
```

```
    System.out.println("跳跃游戏测试: " + (canJump ? "✓ 通过" : "✗ 失败"));
```

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

```
}
```

```
/**
```

```
* 测试高级贪心算法 (Test Advanced Greedy Algorithms)
```

```

* 验证课程表、字符串处理等高级贪心算法的正确性
*
* 测试内容:
* 1. 课程表 III: 验证最多课程数计算
* 2. 去除重复字母: 验证字典序最小字符串
* 3. 最多会议: 验证最多可参加会议数
*
* 算法特点:
* - 时间复杂度:  $O(n \log n)$  - 排序和遍历
* - 空间复杂度:  $O(n)$  - 辅助数据结构
* - 测试方法: 使用典型用例验证结果
*/
private static void testAdvancedGreedyAlgorithms() {
    System.out.println("--- 高级贪心算法测试 ---");

    // 测试课程表 III (使用独立实现)
    // 算法: 按截止时间排序, 贪心选择
    int[][] courses = {{100, 200}, {200, 1300}, {1000, 1250}, {2000, 3200}};
    int courseCount = scheduleCourse(courses);
    System.out.println("课程表 III 测试: " + (courseCount == 3 ? "✓ 通过" : "✗ 失败"));

    // 测试去除重复字母 (使用独立实现)
    // 算法: 单调栈维护字典序最小
    String s = "bcabc";
    String result = removeDuplicateLetters(s);
    System.out.println("去除重复字母测试: " + ("abc".equals(result) ? "✓ 通过" : "✗ 失败"));

    // 测试最多会议 (使用独立实现)
    // 算法: 按结束时间排序, 贪心选择
    int[][] events = {{1, 2}, {2, 3}, {3, 4}, {1, 2}};
    int eventCount = maxEvents(events);
    System.out.println("最多会议测试: " + (eventCount == 4 ? "✓ 通过" : "✗ 失败"));

    System.out.println();
}

/***
 * 测试数学贪心算法 (Test Mathematical Greedy Algorithms)
 * 验证数字处理、计算器操作等数学相关贪心算法的正确性
*
* 测试内容:
* 1. 最大数: 验证数字组合最大值
* 2. 坏了的计算器: 验证最少操作次数

```

```

* 3. 最大硬币数: 验证最优分配策略
*
* 算法特点:
* - 时间复杂度:  $O(n \log n)$  - 排序和遍历
* - 空间复杂度:  $O(n)$  - 字符串和数组存储
* - 测试方法: 使用数学用例验证结果
*/
private static void testMathematicalGreedyAlgorithms() {
    System.out.println("--- 数学贪心算法测试 ---");

    // 测试最大数 (使用独立实现)
    // 算法: 自定义排序比较拼接结果
    int[] nums = {3, 30, 34, 5, 9};
    String largestNum = largestNumber(nums);
    System.out.println("最大数测试: " + ("9534330".equals(largestNum) ? "✓ 通过" : "✗ 失败"));
}

// 测试坏了的计算器 (使用独立实现)
// 算法: 反向思维, 贪心操作
int operations = brokenCalc(2, 3);
System.out.println("坏了的计算器测试: " + (operations == 2 ? "✓ 通过" : "✗ 失败"));

// 测试最大硬币数 (使用独立实现)
// 算法: 排序后贪心选择
int[] piles = {2, 4, 1, 2, 7, 8};
int coins = maxCoins(piles);
System.out.println("最大硬币数测试: " + (coins == 9 ? "✓ 通过" : "✗ 失败"));

System.out.println();
}

// 以下是独立的贪心算法实现, 用于测试

/**
 * 分发饼干 (贪心算法) (Assign Cookies - Greedy Algorithm)
 * 使用贪心策略为孩子分配饼干, 最大化满足的孩子数
*
* 算法思路:
* 1. 排序: 对孩子胃口和饼干尺寸升序排序
* 2. 匹配: 使用双指针贪心匹配最小胃口和最小饼干
* 3. 计数: 统计满足的孩子数
*
* 时间复杂度:  $O(m \log m + n \log n)$  - m 是孩子数, n 是饼干数

```

```

* 空间复杂度: O(1) - 常数额外空间
* 是否最优解: 是
*/
private static int findContentChildren(int[] g, int[] s) {
    Arrays.sort(g); // 按胃口升序排序
    Arrays.sort(s); // 按尺寸升序排序
    int i = 0, j = 0; // 双指针
    while (i < g.length && j < s.length) {
        if (s[j] >= g[i]) { // 当前饼干能满足当前孩子
            i++; // 满足孩子数加 1
        }
        j++; // 饼干指针前移
    }
    return i; // 返回满足的孩子数
}

/***
* 买卖股票的最佳时机 II (贪心算法) (Best Time to Buy and Sell Stock II - Greedy Algorithm)
* 使用贪心策略计算股票交易的最大利润
*
* 算法思路:
* 1. 收集: 收集所有正的价格差
* 2. 累加: 将所有正收益累加
* 3. 返回: 得到最大总利润
*
* 时间复杂度: O(n) - 一次遍历
* 空间复杂度: O(1) - 常数额外空间
* 是否最优解: 是
*/
private static int maxProfit(int[] prices) {
    int profit = 0;
    for (int i = 1; i < prices.length; i++) {
        if (prices[i] > prices[i - 1]) { // 价格上涨
            profit += prices[i] - prices[i - 1]; // 累加利润
        }
    }
    return profit;
}

/***
* 跳跃游戏 (贪心算法) (Jump Game - Greedy Algorithm)
* 使用贪心策略判断是否能到达数组末尾
*

```

```

* 算法思路:
* 1. 维护: 维护能到达的最远位置
* 2. 更新: 动态更新最远可达位置
* 3. 判断: 检查是否能到达终点
*
* 时间复杂度: O(n) - 一次遍历
* 空间复杂度: O(1) - 常数额外空间
* 是否最优解: 是
*/
private static boolean canJump(int[] nums) {
    int maxReach = 0; // 最远可达位置
    for (int i = 0; i < nums.length; i++) {
        if (i > maxReach) return false; // 无法到达位置 i
        maxReach = Math.max(maxReach, i + nums[i]); // 更新最远位置
    }
    return true;
}

/***
* 课程表 III (贪心算法) (Course Schedule III - Greedy Algorithm)
* 使用贪心策略选择最多的课程
*
* 算法思路:
* 1. 排序: 按截止时间升序排序
* 2. 选择: 贪心选择能在截止时间前完成的课程
* 3. 计数: 统计可选课程数
*
* 时间复杂度: O(n log n) - 排序时间
* 空间复杂度: O(1) - 常数额外空间
* 是否最优解: 是
*/
private static int scheduleCourse(int[][] courses) {
    Arrays.sort(courses, (a, b) -> a[1] - b[1]); // 按截止时间排序
    int time = 0; // 累计时间
    int count = 0; // 课程数
    for (int[] course : courses) {
        if (time + course[0] <= course[1]) { // 能在截止时间前完成
            time += course[0]; // 累加课程时间
            count++; // 课程数加 1
        }
    }
    return count;
}

```

```

/**
 * 去除重复字母（贪心算法）(Remove Duplicate Letters - Greedy Algorithm)
 * 使用贪心策略构造字典序最小的无重复字符串
 *
 * 算法思路：
 * 1. 预处理：记录每个字符最后出现位置
 * 2. 构造：使用单调栈构造结果
 * 3. 贪心：维护字典序最小的字符序列
 *
 * 时间复杂度：O(n) - 一次遍历
 * 空间复杂度：O(1) - 固定大小数组
 * 是否最优解：是
 */

private static String removeDuplicateLetters(String s) {
    boolean[] visited = new boolean[26]; // 字符访问状态
    int[] lastIndex = new int[26]; // 字符最后位置
    for (int i = 0; i < s.length(); i++) {
        lastIndex[s.charAt(i) - 'a'] = i; // 记录最后位置
    }

    StringBuilder result = new StringBuilder(); // 结果字符串
    for (int i = 0; i < s.length(); i++) {
        char c = s.charAt(i);
        if (visited[c - 'a']) continue; // 已访问过跳过

        // 维护单调栈：弹出大于当前字符且后面还会出现的字符
        while (result.length() > 0 && result.charAt(result.length() - 1) > c &&
               lastIndex[result.charAt(result.length() - 1) - 'a'] > i) {
            visited[result.charAt(result.length() - 1) - 'a'] = false;
            result.deleteCharAt(result.length() - 1);
        }

        result.append(c); // 添加当前字符
        visited[c - 'a'] = true; // 标记已访问
    }
    return result.toString();
}

/**
 * 最多可以参加的会议数目（贪心算法）(Maximum Events - Greedy Algorithm)
 * 使用贪心策略选择最多的会议参加
 *

```

```

* 算法思路:
* 1. 排序: 按结束时间升序排序
* 2. 选择: 贪心选择最早可参加的会议
* 3. 标记: 标记已参加的日期
*
* 时间复杂度: O(n log n + n*d) - n 是会议数, d 是日期范围
* 空间复杂度: O(d) - 日期标记数组
* 是否最优解: 是
*/
private static int maxEvents(int[][] events) {
    Arrays.sort(events, (a, b) -> a[1] - b[1]); // 按结束时间排序
    boolean[] attended = new boolean[100001]; // 日期参加状态
    int count = 0; // 参加会议数
    for (int[] event : events) {
        for (int day = event[0]; day <= event[1]; day++) {
            if (!attended[day]) { // 找到可参加日期
                attended[day] = true; // 标记已参加
                count++; // 会议数加 1
                break;
            }
        }
    }
    return count;
}

/***
 * 最大数 (贪心算法) (Largest Number - Greedy Algorithm)
 * 使用贪心策略构造最大的数字组合
*
* 算法思路:
* 1. 转换: 将数字转换为字符串
* 2. 排序: 按拼接结果降序排序
* 3. 构造: 拼接排序后的字符串
*
* 时间复杂度: O(n log n) - 排序时间
* 空间复杂度: O(n) - 字符串数组
* 是否最优解: 是
*/
private static String largestNumber(int[] nums) {
    String[] strNums = new String[nums.length];
    for (int i = 0; i < nums.length; i++) {
        strNums[i] = String.valueOf(nums[i]); // 转换为字符串
    }
}

```

```

// 自定义排序: 比较拼接结果
Arrays.sort(strNums, (a, b) -> (b + a).compareTo(a + b));

if (strNums[0].equals("0")) { // 处理全零情况
    return "0";
}

StringBuilder result = new StringBuilder();
for (String num : strNums) {
    result.append(num); // 拼接结果
}
return result.toString();
}

/***
 * 坏了的计算器 (贪心算法) (Broken Calculator - Greedy Algorithm)
 * 使用贪心策略计算最少操作次数
 *
 * 算法思路:
 * 1. 反向: 从目标值反向计算
 * 2. 贪心: 偶数除 2, 奇数加 1
 * 3. 补偿: 处理剩余差值
 *
 * 时间复杂度: O(log target) - 反向操作次数
 * 空间复杂度: O(1) - 常数额外空间
 * 是否最优解: 是
 */
private static int brokenCalc(int start, int target) {
    int operations = 0;
    while (target > start) { // 反向计算
        operations++; // 操作数加 1
        if (target % 2 == 0) { // 偶数
            target /= 2; // 除 2
        } else { // 奇数
            target++; // 加 1
        }
    }
    return operations + (start - target); // 补偿剩余差值
}

/***
 * 最大硬币数 (贪心算法) (Maximum Coins - Greedy Algorithm)

```

```
* 使用贪心策略获得最大硬币数
*
* 算法思路:
* 1. 排序: 按硬币数升序排序
* 2. 选择: 贪心选择第二大的堆
* 3. 累加: 累加所选堆的硬币数
*
* 时间复杂度: O(n log n) - 排序时间
* 空间复杂度: O(1) - 常数额外空间
* 是否最优解: 是
*/
private static int maxCoins(int[] piles) {
    Arrays.sort(piles);           // 升序排序
    int result = 0;               // 累计硬币数
    int n = piles.length;
    // 从倒数第二个开始, 每隔一个取一个
    for (int i = n - 2; i >= n / 3; i -= 2) {
        result += piles[i];       // 累加硬币数
    }
    return result;
}
```

---