

=====

文件夹: class120_GreedyAlgorithms

=====

[Markdown 文件]

=====

文件: ExtendedProblems.md

=====

贪心算法扩展题目详解

🎯 概述

本文件收集了各大算法平台上的贪心算法经典题目，包括 LeetCode、LintCode、HackerRank、牛客网、洛谷、Codeforces、AtCoder 等，帮助全面掌握贪心算法的应用场景和解题技巧。

📄 题目分类与详解

1. 基础贪心题目

1.1 分发饼干 (LeetCode 455)

- **题目链接**: <https://leetcode.cn/problems/assign-cookies/>
- **难度**: 简单
- **题目描述**: 每个孩子有一个胃口值 $g[i]$ ，每块饼干有一个尺寸 $s[j]$ 。如果 $s[j] \geq g[i]$ ，可以将饼干 j 分配给孩子 i 。目标是尽可能满足越多数量的孩子。
- **贪心策略**: 将孩子和饼干都按升序排列，优先满足胃口小的孩子。
- **时间复杂度**: $O(m \log m + n \log n)$
- **空间复杂度**: $O(1)$
- **是否最优解**: 是

1.2 柠檬水找零 (LeetCode 860)

- **题目链接**: <https://leetcode.cn/problems/lemonade-change/>
- **难度**: 简单
- **题目描述**: 每杯柠檬水售价 5 美元，顾客支付 5、10 或 20 美元。初始时没有零钱，需要判断是否能给每个顾客正确找零。
- **贪心策略**: 收到 10 美元时优先用 5 美元找零，收到 20 美元时优先用 10+5 美元找零。
- **时间复杂度**: $O(n)$
- **空间复杂度**: $O(1)$
- **是否最优解**: 是

1.3 K 次取反后最大化的数组和 (LeetCode 1005)

- **题目链接**: <https://leetcode.cn/problems/maximize-sum-of-array-after-k-negations/>
- **难度**: 简单
- **题目描述**: 给定一个整数数组 $nums$ 和一个整数 k ，可以选择数组中的一个下标并将该下标对应元素取

反，重复 k 次，返回数组可能的最大和。

- **贪心策略**: 每次操作都选择绝对值最小的负数进行取反，如果没有负数则反复取反绝对值最小的数。
- **时间复杂度**: $O(n \log n)$
- **空间复杂度**: $O(1)$
- **是否最优解**: 是

2. 序列问题

2.1 摆动序列 (LeetCode 376)

- **题目链接**: <https://leetcode.cn/problems/wiggle-subsequence/>
- **难度**: 中等
- **题目描述**: 如果连续数字之间的差严格地在正数和负数之间交替，则数字序列为摆动序列。返回数组中最长摆动子序列的长度。
- **贪心策略**: 遇到上升就记录上升，遇到下降就记录下降，忽略相等的情况。
- **时间复杂度**: $O(n)$
- **空间复杂度**: $O(1)$
- **是否最优解**: 是

2.2 单调递增的数字 (LeetCode 738)

- **题目链接**: <https://leetcode.cn/problems/monotone-increasing-digits/>
- **难度**: 中等
- **题目描述**: 当且仅当每个相邻位数上的数字 x 和 y 满足 $x \leq y$ 时，我们称这个整数是单调递增的。给定一个整数 n ，返回小于或等于 n 的最大单调递增数字。
- **贪心策略**: 从右往左遍历，如果发现左边数字大于右边数字，则左边数字减 1，右边所有数字变为 9。
- **时间复杂度**: $O(\log n)$
- **空间复杂度**: $O(\log n)$
- **是否最优解**: 是

3. 股票问题

3.1 买卖股票的最佳时机 II (LeetCode 122)

- **题目链接**: <https://leetcode.cn/problems/best-time-to-buy-and-sell-stock-ii/>
- **难度**: 中等
- **题目描述**: 给你一个整数数组 prices ，其中 $\text{prices}[i]$ 表示某支股票第 i 天的价格。在每一天，你可以决定是否购买和/或出售股票。你在任何时候 最多 只能持有 一股 股票。你也可以先购买，然后在 同一天 出售。返回 你能获得的最大利润。
- **贪心策略**: 将问题转化为每天的收益，只要收益为正就累加。只要明天价格比今天高，就在今天买入明天卖出，累加所有正的收益差值。
- **时间复杂度**: $O(n)$
- **空间复杂度**: $O(1)$
- **是否最优解**: 是

3.2 买卖股票的最佳时机含手续费 (LeetCode 714)

- **题目链接**: <https://leetcode.cn/problems/best-time-to-buy-and-sell-stock-with-transaction-fee/>
- **难度**: 中等
- **题目描述**: 可以完成多次交易，但每笔交易都需要手续费，求最大利润。
- **贪心策略**: 在价格低点买入，在价格高点卖出，考虑手续费成本。
- **时间复杂度**: $O(n)$
- **空间复杂度**: $O(1)$
- **是否最优解**: 是

4. 两个维度权衡问题

4.1 分发糖果 (LeetCode 135)

- **题目链接**: <https://leetcode.cn/problems/candy/>
- **难度**: 困难
- **题目描述**: 每个孩子至少分配到 1 个糖果，相邻评分高的孩子会获得更多糖果，求最少糖果总数。
- **贪心策略**: 两次遍历，从左到右满足右邻条件，从右到左满足左邻条件。
- **时间复杂度**: $O(n)$
- **空间复杂度**: $O(n)$
- **是否最优解**: 是

4.2 根据身高重建队列 (LeetCode 406)

- **题目链接**: <https://leetcode.cn/problems/queue-reconstruction-by-height/>
- **难度**: 中等
- **题目描述**: 给定一个数组 people，其中 $people[i] = [hi, ki]$ 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，重新构造队列。
- **贪心策略**: 按身高降序、 k 值升序排列，然后按 k 值插入到结果队列中。
- **时间复杂度**: $O(n^2)$
- **空间复杂度**: $O(n)$
- **是否最优解**: 是

5. 区间问题

5.1 跳跃游戏 (LeetCode 55)

- **题目链接**: <https://leetcode.cn/problems/jump-game/>
- **难度**: 中等
- **题目描述**: 数组每个元素代表在该位置可以跳跃的最大长度，判断是否能到达最后一个下标。
- **贪心策略**: 维护能到达的最远位置，如果当前位置可达且能跳得更远就更新最远位置。
- **时间复杂度**: $O(n)$
- **空间复杂度**: $O(1)$
- **是否最优解**: 是

5.2 跳跃游戏 II (LeetCode 45)

- **题目链接**: <https://leetcode.cn/problems/jump-game-ii/>

- **难度**: 中等
- **题目描述**: 数组每个元素代表在该位置可以跳跃的最大长度，使用最少的跳跃次数到达最后一个下标。
- **贪心策略**: 每次在当前能跳到的范围内选择下一步能跳最远的位置。
- **时间复杂度**: $O(n)$
- **空间复杂度**: $O(1)$
- **是否最优解**: 是

5.3 用最少量的箭引爆气球 (LeetCode 452)

- **题目链接**: <https://leetcode.cn/problems/minimum-number-of-arrows-to-burst-balloons/>
- **难度**: 中等
- **题目描述**: 一些球形的气球贴在一堵用 XY 平面表示的墙面上，一支弓箭可以沿着 x 轴从不同点完全垂直地射出，求引爆所有气球所需的最小弓箭数。
- **贪心策略**: 按右端点排序，尽可能多地引爆重叠的气球。
- **时间复杂度**: $O(n \log n)$
- **空间复杂度**: $O(1)$
- **是否最优解**: 是

5.4 无重叠区间 (LeetCode 435)

- **题目链接**: <https://leetcode.cn/problems/non-overlapping-intervals/>
- **难度**: 中等
- **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。
- **贪心策略**: 按右端点排序，优先保留结束早的区间。
- **时间复杂度**: $O(n \log n)$
- **空间复杂度**: $O(1)$
- **是否最优解**: 是

5.5 划分字母区间 (LeetCode 763)

- **题目链接**: <https://leetcode.cn/problems/partition-labels/>
- **难度**: 中等
- **题目描述**: 字符串 S 由小写字母组成，需要将这个字符串划分为尽可能多的片段，同一个字母只会出现在一个片段中。
- **贪心策略**: 记录每个字符最后出现的位置，然后从头开始遍历，维护当前片段的结束位置。
- **时间复杂度**: $O(n)$
- **空间复杂度**: $O(1)$
- **是否最优解**: 是

5.6 合并区间 (LeetCode 56)

- **题目链接**: <https://leetcode.cn/problems/merge-intervals/>
- **难度**: 中等
- **题目描述**: 以数组 intervals 表示若干个区间的集合，合并所有重叠的区间，返回一个不重叠的区间数组。
- **贪心策略**: 按起始位置排序，然后依次合并重叠区间。
- **时间复杂度**: $O(n \log n)$

- **空间复杂度**: $O(n)$
- **是否最优解**: 是

6. 其他问题

6.1 最大子数组和 (LeetCode 53)

- **题目链接**: <https://leetcode.cn/problems/maximum-subarray/>
- **难度**: 简单
- **题目描述**: 找到一个具有最大和的连续子数组（子数组最少包含一个元素）。
- **贪心策略**: 如果当前累加和为负数，则重新开始计算。
- **时间复杂度**: $O(n)$
- **空间复杂度**: $O(1)$
- **是否最优解**: 是

6.2 加油站 (LeetCode 134)

- **题目链接**: <https://leetcode.cn/problems/gas-station/>
- **难度**: 中等
- **题目描述**: 在一条环路上有 n 个加油站，第 i 个加油站有汽油 $gas[i]$ 升，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，确定能绕环路行驶一周的出发加油站。
- **贪心策略**: 如果总油量小于总消耗量则无解；否则从某个点开始一定能走完一圈。
- **时间复杂度**: $O(n)$
- **空间复杂度**: $O(1)$
- **是否最优解**: 是

6.3 监控二叉树 (LeetCode 968)

- **题目链接**: <https://leetcode.cn/problems/binary-tree-cameras/>
- **难度**: 困难
- **题目描述**: 给定一个二叉树，在树的节点上安装摄像头，节点上的摄像头可以监控其父对象、自身及其直接子对象，计算监控树的所有节点所需的小摄像头数量。
- **贪心策略**: 后序遍历，从下往上尽可能在叶子节点的父节点安装摄像头。
- **时间复杂度**: $O(n)$
- **空间复杂度**: $O(n)$
- **是否最优解**: 是

🌐 各大平台贪心题目汇总

LeetCode 贪心题目

1. 11. 盛最多水的容器
2. 44. 通配符匹配
3. 45. 跳跃游戏 II
4. 53. 最大子数组和
5. 55. 跳跃游戏
6. 122. 买卖股票的最佳时机 II

7. 134. 加油站
8. 135. 分发糖果
9. 316. 去除重复字母
10. 376. 摆动序列
11. 402. 移掉 K 位数字
12. 406. 根据身高重建队列
13. 435. 无重叠区间
14. 452. 用最少数量的箭引爆气球
15. 455. 分发饼干
16. 502. IPO
17. 561. 数组拆分 I
18. 621. 任务调度器
19. 649. Dota2 参议院
20. 714. 买卖股票的最佳时机含手续费
21. 738. 单调递增的数字
22. 763. 划分子母区间
23. 860. 柠檬水找零
24. 861. 翻转矩阵后的得分
25. 870. 优势洗牌
26. 948. 令牌放置
27. 968. 监控二叉树
28. 1005. K 次取反后最大化的数组和
29. 1029. 两地调度
30. 1053. 交换一次的先前排列
31. 1094. 拼车
32. 1221. 分割平衡字符串
33. 1247. 交换字符使得字符串相同
34. 1328. 破坏回文串
35. 1405. 最长快乐字符串
36. 1414. 和为 K 的最少斐波那契数字数目
37. 1518. 换酒问题
38. 1529. 灯泡开关 IV
39. 1561. 你可以获得的最大硬币数目
40. 1578. 避免重复字母的最小删除成本
41. 1605. 给定行和列的和求可行矩阵
42. 1647. 字符频次唯一的最小删除次数
43. 1663. 具有给定数值的最小字符串
44. 1702. 修改后的最大二进制字符串
45. 1710. 卡车上的最大单元数
46. 1727. 重新排列后的最大子矩阵
47. 1798. 你能构造出连续值的最大数目
48. 1833. 雪糕的最大数量
49. 1846. 减小和重新排列数组后的最大元素

50. 1877. 数组中最大数对和的最小值

洛谷贪心题目

1. P1223 排队接水
2. P1803 凌乱的yyy / 线段覆盖
3. P1090 合并果子
4. P1478 陶陶摘苹果（升级版）
5. P3817 小A的糖果
6. P1106 删数问题
7. P5019 铺设道路
8. P1208 混合牛奶
9. P1094 纪念品分组
10. P4995 跳跳！

Codeforces 贪心题目

1. 479C - Exams
2. 489B - BerSU Ball
3. 478B - Random Teams
4. 441C - Valera and Tubes
5. 454B - Little Pony and Sort by Shift
6. 476B - Dreamoon and WiFi
7. 437C - The Child and Toy
8. 459B - Pashmak and Flowers
9. 463B - Caisa and Pylons
10. 448B - Suffix Structures

AtCoder 贪心题目

1. ABC143D - Triangles
2. ABC153D - Caracal vs Monster
3. ABC173D - Chat in a Circle
4. ABC164D - Multiple of 2019
5. ABC121C - Energy Drink Collector

💡 贪心算法总结

适用场景

1. **具有贪心选择性质**: 局部最优解能推导出全局最优解
2. **具有最优子结构**: 问题的最优解包含子问题的最优解
3. **无后效性**: 某个状态以前的过程不会影响以后的状态

常见题型

1. **区间调度问题**: 活动安排、区间合并等
2. **哈夫曼编码**: 合并果子等

3. **分数背包**: 性价比最高的选择
4. **股票买卖**: 收集所有上升波段
5. **字典序问题**: 移掉 K 位数字等
6. **序列重构**: 根据身高重建队列等

解题技巧

1. **排序**: 根据题目要求对数据进行排序
2. **双指针**: 在有序数组中进行操作
3. **优先队列**: 维护当前最优选择
4. **数学推导**: 找出规律直接计算

注意事项

1. **贪心不一定能得到最优解**: 需要严格证明
2. **反例验证**: 通过构造反例验证贪心策略
3. **边界条件**: 注意处理特殊情况
4. **时间复杂度**: 多数贪心算法时间复杂度较低

🏆 总结

贪心算法是算法设计中的一种重要思想，它通过每一步的局部最优选择来达到全局最优解。掌握贪心算法需要：

1. 熟悉各种经典题型和解法
2. 理解贪心策略的正确性证明
3. 多做练习，积累经验
4. 注意边界条件和特殊情况的处理

通过系统学习和大量练习 class089 中的题目以及扩展题目，可以全面掌握贪心算法的应用技巧，为解决实际问题和面试打下坚实基础。

📊 贪心算法复杂度分析表

问题类型	时间复杂度	空间复杂度	最优解	关键数据结构
分发饼干	$O(m \log m + n \log n)$	$O(1)$	是	排序、双指针
柠檬水找零	$O(n)$	$O(1)$	是	计数变量
买卖股票 II	$O(n)$	$O(1)$	是	累加变量
跳跃游戏	$O(n)$	$O(1)$	是	最远距离
跳跃游戏 II	$O(n)$	$O(1)$	是	双指针
最大子数组和	$O(n)$	$O(1)$	是	累加变量
加油站	$O(n)$	$O(1)$	是	油量统计
摆动序列	$O(n)$	$O(1)$	是	状态变量
无重叠区间	$O(n \log n)$	$O(1)$	是	排序
用箭引爆气球	$O(n \log n)$	$O(1)$	是	排序

分发糖果	$O(n)$	$O(n)$	是	数组
根据身高重建队列	$O(n^2)$	$O(n)$	是	排序、插入
监控二叉树	$O(n)$	$O(n)$	是	递归遍历

🔎 贪心算法证明技巧

1. 交换论证法

通过交换两个元素的位置来证明贪心策略的正确性。

2. 数学归纳法

证明贪心选择性质，即每一步的局部最优选择能导致全局最优解。

3. 反证法

假设存在更优解，然后推导出矛盾。

4. 剪枝法

证明某些选择不可能导致最优解，从而排除这些选择。

##💡 贪心算法实战技巧

1. 识别贪心特征

- 问题可以分解为多个子问题
- 每个子问题的最优解能组合成全局最优解
- 选择具有无后效性

2. 设计贪心策略

- 确定选择标准（如最小、最大、最早等）
- 验证策略的正确性
- 考虑边界情况

3. 实现与优化

- 选择合适的排序方法
- 使用高效的数据结构
- 优化空间复杂度

4. 测试与验证

- 测试边界情况
- 验证时间复杂度
- 检查是否是最优解

通过系统学习和大量练习，可以全面掌握贪心算法的应用技巧，为解决实际问题和面试打下坚实基础。

=====

文件: README.md

Class089 贪心算法专题

🎯 概述

Class089 专注于贪心算法的学习和应用。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。本章节通过多个经典题目，帮助深入理解贪心算法的设计思想和应用场景。

📚 题目列表

1. 最大数 (Largest Number)

- **文件**: [Code01_LargestNumber.java] (Code01_LargestNumber.java)
 - [Code01_LargestNumber.cpp] (Code01_LargestNumber.cpp)
 - [Code01_LargestNumber.py] (Code01_LargestNumber.py)
- **题目链接**: <https://leetcode.cn/problems/largest-number/>
- **难度**: 中等
- **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
- **解法**: 贪心算法 + 自定义排序
- **时间复杂度**: $O(n \log n m)$
- **空间复杂度**: $O(n m)$
- **是否最优解**: 是

2. 两地调度 (Two City Scheduling)

- **文件**: [Code02_TwoCityScheduling.java] (Code02_TwoCityScheduling.java)
- **题目链接**: <https://leetcode.cn/problems/two-city-scheduling/>
- **难度**: 中等
- **描述**: 公司计划面试 $2n$ 个人，给定一个数组 costs ，其中 $\text{costs}[i] = [\text{aCost}_i, \text{bCost}_i]$ 表示第 i 人飞往 a 市的费用为 aCost_i ，飞往 b 市的费用为 bCost_i 。返回将每个人都飞到 a、b 中某座城市的最低费用，要求每个城市都有 n 人抵达
- **解法**: 贪心算法
- **时间复杂度**: $O(n \log n)$
- **空间复杂度**: $O(n)$
- **是否最优解**: 是

3. 吃掉 N 个橘子的最少天数 (Minimum Number of Days to Eat N Oranges)

- **文件**: [Code03_MinimumNumberEatOranges.java] (Code03_MinimumNumberEatOranges.java)
- **题目链接**: <https://leetcode.cn/problems/minimum-number-of-days-to-eat-n-oranges/>
- **难度**: 困难
- **描述**: 厨房里总共有 n 个橘子，你决定每一天选择如下方式之一吃这些橘子：1) 吃掉一个橘子；2) 如

果剩余橘子数 n 能被 2 整除，那么你可以吃掉 $n/2$ 个橘子；3) 如果剩余橘子数 n 能被 3 整除，那么你可以吃掉 $2*(n/3)$ 个橘子。每天你只能从以上 3 种方案中选择一种方案。请你返回吃掉所有 n 个橘子的最少天数

- **解法**: 贪心算法 + 记忆化搜索
- **时间复杂度**: $O(\log n)$
- **空间复杂度**: $O(\log n)$
- **是否最优解**: 是

4. 会议室 II (Meeting Rooms II)

- **文件**: [Code04_MeetingRoomsII. java] (Code04_MeetingRoomsII. java)
- **题目链接**: <https://leetcode.cn/problems/meeting-rooms-ii/>
- **难度**: 中等
- **描述**: 给你一个会议时间安排的数组 intervals ，每个会议时间都会包括开始和结束的时间 $\text{intervals}[i]=[start_i, end_i]$ ，返回所需会议室的最小数量
- **解法**: 贪心算法 + 最小堆
- **时间复杂度**: $O(n * \log n)$
- **空间复杂度**: $O(n)$
- **是否最优解**: 是

5. 课程表 III (Course Schedule III)

- **文件**: [Code05_CourseScheduleIII. java] (Code05_CourseScheduleIII. java)
- **题目链接**: <https://leetcode.cn/problems/course-schedule-iii/>
- **难度**: 困难
- **描述**: 这里有 n 门不同的在线课程，按从 1 到 n 编号。给你一个数组 courses ，其中 $\text{courses}[i]=[duration_i, lastDay_i]$ 表示第 i 门课将会持续上 $duration_i$ 天课，并且必须在不晚于 $lastDay_i$ 的时候完成。你的学期从第 1 天开始，且不能同时修读两门及两门以上的课程。返回你最多可以修读的课程数目
- **解法**: 贪心算法 + 最大堆
- **时间复杂度**: $O(n * \log n)$
- **空间复杂度**: $O(n)$
- **是否最优解**: 是

6. 连接棒材的最低费用 (Minimum Cost to Connect Sticks)

- **文件**:
 - [Code06_MinimumCostToConnectSticks1. java] (Code06_MinimumCostToConnectSticks1. java) (LeetCode 版本)
 - [Code06_MinimumCostToConnectSticks2. java] (Code06_MinimumCostToConnectSticks2. java) (洛谷版本)
- **题目链接**:
 - LeetCode: <https://leetcode.cn/problems/minimum-cost-to-connect-sticks/>
 - 洛谷: <https://www.luogu.com.cn/problem/P1090>
- **难度**: 中等
- **描述**: 你有一些长度为正整数的棍子，这些长度以数组 sticks 的形式给出。你可以通过支付 $x+y$ 的成本将任意两个长度为 x 和 y 的棍子连接成一个棍子。你必须连接所有的棍子，直到剩下一个棍子。返回以这种方式将所有给定的棍子连接成一个棍子的最小成本

- **解法**: 贪心算法 + 最小堆

- **时间复杂度**: $O(n \log n)$

- **空间复杂度**: $O(n)$

- **是否最优解**: 是

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

- **文件**:

- [Code07_BestTimeBuySellStockII.java] (Code07_BestTimeBuySellStockII.java)

- [Code07_BestTimeBuySellStockII.cpp] (Code07_BestTimeBuySellStockII.cpp)

- [Code07_BestTimeBuySellStockII.py] (Code07_BestTimeBuySellStockII.py)

- **题目链接**: <https://leetcode.cn/problems/best-time-to-buy-and-sell-stock-ii/>

- **难度**: 中等

- **描述**: 给你一个整数数组 `prices`，其中 `prices[i]` 表示某支股票第 `i` 天的价格。在每一天，你可以决定是否购买和/或出售股票。你在任何时候 最多 只能持有 一股 股票。你也可以先购买，然后在 同一天 出售。返回 你能获得的最大利润

- **解法**: 贪心算法 + 累加正收益

- **时间复杂度**: $O(n)$

- **空间复杂度**: $O(1)$

- **是否最优解**: 是

8. 分发饼干 (Assign Cookies)

- **文件**:

- [Code08_AssignCookies.java] (Code08_AssignCookies.java)

- [Code08_AssignCookies.cpp] (Code08_AssignCookies.cpp)

- [Code08_AssignCookies.py] (Code08_AssignCookies.py)

- **题目链接**: <https://leetcode.cn/problems/assign-cookies/>

- **难度**: 简单

- **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`。目标是尽可能满足越多数量的孩子，并输出这个最大数值

- **解法**: 贪心算法 + 双指针

- **时间复杂度**: $O(m \log m + n \log n)$

- **空间复杂度**: $O(1)$

- **是否最优解**: 是

9. 柠檬水找零 (Lemonade Change)

- **文件**:

- [Code09_LemonadeChange.java] (Code09_LemonadeChange.java)

- [Code09_LemonadeChange.cpp] (Code09_LemonadeChange.cpp)

- [Code09_LemonadeChange.py] (Code09_LemonadeChange.py)

- **题目链接**: <https://leetcode.cn/problems/lemonade-change/>

- **难度**: 简单

- **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

- **解法**: 贪心算法 + 计数策略

- **时间复杂度**: $O(n)$

- **空间复杂度**: $O(1)$

- **是否最优解**: 是

💡 贪心算法核心思想

1. 基本概念

贪心算法 (greedy algorithm, 又称贪婪算法) 是指在对问题求解时，总是做出在当前看来是最好的选择。也就是说，不从整体最优上加以考虑，算法得到的是在某种意义上的局部最优解。

2. 适用场景

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**: 问题的最优解包含子问题的最优解

3. 解题步骤

1. 将问题分解为若干个子问题
2. 找出适合的贪心策略
3. 求解每一个子问题的最优解
4. 将局部最优解堆叠成全局最优解

4. 常见题型

1. **区间调度问题**: 活动安排、区间合并等
2. **哈夫曼编码**: 合并果子等
3. **分数背包**: 性价比最高的选择
4. **股票买卖**: 收集所有上升波段
5. **字典序问题**: 移掉 K 位数字等
6. **序列重构**: 根据身高重建队列等

🔧 工程化考量

1. 异常处理

- 空输入处理：检查输入是否为空或 null
- 边界条件：处理数组为空、单个元素等特殊情况
- 输入验证：检查输入参数的有效性

2. 性能优化

- 排序优化：合理选择排序算法和比较器
- 避免重复计算：缓存计算结果
- 数据结构选择：根据场景选择合适的数据结构

3. 可配置性

- 比较器定制：通过自定义比较器支持不同的排序需求
- 参数化设计：通过参数控制行为

4. 线程安全

- 在多线程环境中使用时，需要考虑同步机制

5. 内存管理

- 及时清理不需要的对象，避免内存泄漏

6. 代码可读性

- 清晰的变量命名和注释
- 模块化设计，将复杂逻辑分解为独立方法

7. 单元测试

- 覆盖各种边界情况和异常输入
- 验证时间和空间复杂度是否符合预期

8. 跨语言特性

- Java: PriorityQueue, Arrays.sort
- Python: heapq, sorted
- C++: priority_queue, sort

9. 调试技巧

- 打印中间状态用于调试
- 使用断言验证中间结果

10. 与标准库对比

- 理解标准库实现的优势和局限性
- 在性能要求极高时考虑自实现数据结构

复杂度分析总结

问题类型	时间复杂度	空间复杂度	最优解
最大数	$O(n \log n * m)$	$O(n * m)$	是
两地调度	$O(n \log n)$	$O(n)$	是
吃橘子	$O(1 \log n)$	$O(1 \log n)$	是
会议室 II	$O(n \log n)$	$O(n)$	是

	课程表 III		0(n*logn)		0(n)		是	
	连接棒材		0(n*logn)		0(n)		是	

🌐 扩展题目平台

LeetCode 贪心题目

1. 11. 盛最多水的容器
2. 44. 通配符匹配
3. 45. 跳跃游戏 II
4. 53. 最大子数组和
5. 55. 跳跃游戏
6. 122. 买卖股票的最佳时机 II
7. 134. 加油站
8. 135. 分发糖果
9. 316. 去除重复字母
10. 376. 摆动序列
11. 402. 移掉 K 位数字
12. 406. 根据身高重建队列
13. 435. 无重叠区间
14. 452. 用最少量的箭引爆气球
15. 455. 分发饼干
16. 502. IPO
17. 561. 数组拆分 I
18. 621. 任务调度器
19. 649. Dota2 参议院
20. 714. 买卖股票的最佳时机含手续费
21. 738. 单调递增的数字
22. 763. 划分字母区间
23. 860. 柠檬水找零
24. 861. 翻转矩阵后的得分
25. 870. 优势洗牌
26. 948. 令牌放置
27. 968. 监控二叉树
28. 1005. K 次取反后最大化的数组和
29. 1029. 两地调度
30. 1053. 交换一次的先前排列
31. 1094. 拼车
32. 1221. 分割平衡字符串
33. 1247. 交换字符使得字符串相同
34. 1328. 破坏回文串
35. 1405. 最长快乐字符串
36. 1414. 和为 K 的最少斐波那契数字数目
37. 1518. 换酒问题

38. 1529. 灯泡开关 IV
39. 1561. 你可以获得的最大硬币数目
40. 1578. 避免重复字母的最小删除成本
41. 1605. 给定行和列的和求可行矩阵
42. 1647. 字符频次唯一的最小删除次数
43. 1663. 具有给定数值的最小字符串
44. 1702. 修改后的最大二进制字符串
45. 1710. 卡车上的最大单元数
46. 1727. 重新排列后的最大子矩阵
47. 1798. 你能构造出连续值的最大数目
48. 1833. 雪糕的最大数量
49. 1846. 减小和重新排列数组后的最大元素
50. 1877. 数组中最大数对和的最小值

洛谷贪心题目

1. P1223 排队接水
2. P1803 凌乱的yyy / 线段覆盖
3. P1090 合并果子
4. P1478 陶陶摘苹果（升级版）
5. P3817 小A的糖果
6. P1106 删数问题
7. P5019 铺设道路
8. P1208 混合牛奶
9. P1094 纪念品分组
10. P4995 跳跳！

Codeforces 贪心题目

1. 479C - Exams
2. 489B - BerSU Ball
3. 478B - Random Teams
4. 441C - Valera and Tubes
5. 454B - Little Pony and Sort by Shift
6. 476B - Dreamoon and WiFi
7. 437C - The Child and Toy
8. 459B - Pashmak and Flowers
9. 463B - Caisa and Pylons
10. 448B - Suffix Structures

AtCoder 贪心题目

1. ABC143D - Triangles
2. ABC153D - Caracal vs Monster
3. ABC173D - Chat in a Circle
4. ABC164D - Multiple of 2019

5. ABC121C – Energy Drink Collector

🏆 总结

贪心算法是算法设计中的一种重要思想，它通过每一步的局部最优选择来达到全局最优解。掌握贪心算法需要：

1. 熟悉各种经典题型和解法
2. 理解贪心策略的正确性证明
3. 多做练习，积累经验
4. 注意边界条件和特殊情况的处理

通过系统学习和大量练习 class089 中的题目以及扩展题目，可以全面掌握贪心算法的应用技巧，为解决实际问题和面试打下坚实基础。

更详细的扩展题目分析请参考 [ExtendedProblems.md] (ExtendedProblems.md) 文件。

[代码文件]

文件: Code01_LargestNumber.cpp

```
/**  
 * 最大数 - 贪心算法解决方案 (C++实现)  
 *  
 * 题目描述:  
 * 给定一组非负整数 nums，重新排列每个数的顺序（每个数不可拆分）使之组成一个最大的整数  
 *  
 * 测试链接: https://leetcode.cn/problems/largest-number/  
 *  
 * 算法思想:  
 * 使用贪心算法 + 自定义排序，关键是比较两个字符串 a 和 b 时，比较 a+b 和 b+a 的大小  
 * 如果 a+b > b+a，则 a 应该排在 b 前面，这样拼接后的结果最大  
 *  
 * 时间复杂度分析:  
 * O(n*logn*m) - 其中 n 是数组长度，m 是数字的平均位数  
 * - 排序需要 O(n*logn) 次比较  
 * - 每次比较需要 O(m) 时间 (字符串拼接和比较)  
 *  
 * 空间复杂度分析:  
 * O(n*m) - 需要将整数转换为字符串存储  
 *  
 * 是否为最优解:
```

- * 是，这是解决该问题的最优解
- *
- * 工程化考量：
 - * 1. 边界条件处理：处理全为 0 的特殊情况
 - * 2. 内存管理：及时释放动态分配的内存
 - * 3. 异常处理：对非法输入进行检查
 - * 4. 可读性：使用清晰的变量命名和详细的注释
- *
- * 贪心策略证明：
 - * 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
 - * 这种排序方式满足传递性，因此可以得到全局最优解

*/

```
#include <iostream>
#include <vector>
#include <string>
#include <algorithm>
#include <sstream>

using namespace std;
```

```
/**  
 * 自定义比较函数，用于排序  
 * 比较规则：比较  $a+b$  和  $b+a$  的大小  
 * 如果  $a+b > b+a$ ，则 a 应该排在 b 前面（降序排列）  
 *  
 * @param a 第一个字符串  
 * @param b 第二个字符串  
 * @return 如果 a 应该排在 b 前面返回 true，否则返回 false  
 */
```

```
bool compare(const string& a, const string& b) {
    return (a + b) > (b + a);
}
```

```
/**  
 * 解决最大数问题的核心方法  
 *  
 * @param nums 非负整数数组  
 * @return 重新排列后组成最大整数（字符串形式）  
 *  
 * 算法步骤：

- 1. 将整数数组转换为字符串数组
- 2. 使用自定义比较器对字符串数组进行排序

```

```
* 3. 处理全为 0 的特殊情况
* 4. 拼接排序后的字符串
*
* 特殊处理:
* 如果排序后的第一个字符串是"0", 说明所有数字都是 0, 直接返回"0"
* 避免出现"000"这样的结果, 应该返回"0"
*/
string largestNumber(vector<int>& nums) {
    // 输入验证
    if (nums.empty()) {
        return "0";
    }

    // 将整数转换为字符串
    vector<string> strs;
    for (int num : nums) {
        strs.push_back(to_string(num));
    }

    // 自定义排序: 比较 a+b 和 b+a 的大小
    sort(strs.begin(), strs.end(), compare);

    // 处理全为 0 的特殊情况
    if (strs[0] == "0") {
        return "0";
    }

    // 拼接所有字符串
    string result;
    for (const string& s : strs) {
        result += s;
    }

    return result;
}

/**
 * 测试函数: 验证最大数算法的正确性
 */
void testLargestNumber() {
    cout << "最大数算法测试开始" << endl;
    cout << "======" << endl;
```

```
// 测试用例 1: [10, 2]
vector<int> nums1 = {10, 2};
string result1 = largestNumber(nums1);
cout << "输入: [10, 2]" << endl;
cout << "输出: " << result1 << endl;
cout << "预期: \"210\"" << endl;
cout << (result1 == "210" ? "✓ 通过" : "✗ 失败") << endl << endl;

// 测试用例 2: [3, 30, 34, 5, 9]
vector<int> nums2 = {3, 30, 34, 5, 9};
string result2 = largestNumber(nums2);
cout << "输入: [3, 30, 34, 5, 9]" << endl;
cout << "输出: " << result2 << endl;
cout << "预期: \"9534330\"" << endl;
cout << (result2 == "9534330" ? "✓ 通过" : "✗ 失败") << endl << endl;

// 测试用例 3: [0, 0, 0] - 全为 0 的特殊情况
vector<int> nums3 = {0, 0, 0};
string result3 = largestNumber(nums3);
cout << "输入: [0,0,0]" << endl;
cout << "输出: " << result3 << endl;
cout << "预期: \"0\"" << endl;
cout << (result3 == "0" ? "✓ 通过" : "✗ 失败") << endl << endl;

// 测试用例 4: [1] - 单个元素
vector<int> nums4 = {1};
string result4 = largestNumber(nums4);
cout << "输入: [1]" << endl;
cout << "输出: " << result4 << endl;
cout << "预期: \"1\"" << endl;
cout << (result4 == "1" ? "✓ 通过" : "✗ 失败") << endl << endl;

// 测试用例 5: [432, 43243] - 复杂比较
vector<int> nums5 = {432, 43243};
string result5 = largestNumber(nums5);
cout << "输入: [432,43243]" << endl;
cout << "输出: " << result5 << endl;
cout << "预期: \"43243432\"" << endl;
cout << (result5 == "43243432" ? "✓ 通过" : "✗ 失败") << endl << endl;

cout << "测试结束" << endl;
}
```

```

/**
 * 调试版本：打印排序过程中的中间结果
 *
 * @param nums 整数数组
 * @return 最大数结果
 */
string debugLargestNumber(vector<int>& nums) {
    if (nums.empty()) {
        return "0";
    }

    vector<string> strs;
    for (int num : nums) {
        strs.push_back(to_string(num));
    }

    cout << "原始字符串数组: ";
    for (const string& s : strs) {
        cout << s << " ";
    }
    cout << endl;

    // 打印比较过程
    cout << "比较过程:" << endl;
    for (int i = 0; i < strs.size(); i++) {
        for (int j = i + 1; j < strs.size(); j++) {
            string ab = strs[i] + strs[j];
            string ba = strs[j] + strs[i];
            cout << "比较 " << strs[i] << " 和 " << strs[j] << ": ";
            cout << ab << " vs " << ba << " -> ";
            if (ab > ba) {
                cout << strs[i] << " 应该在 " << strs[j] << " 前面" << endl;
            } else {
                cout << strs[j] << " 应该在 " << strs[i] << " 前面" << endl;
            }
        }
    }

    sort(strs.begin(), strs.end(), compare);

    cout << "排序后字符串数组: ";
    for (const string& s : strs) {
        cout << s << " ";
    }
}

```

```
}

cout << endl;

if (strs[0] == "0") {
    return "0";
}

string result;
for (const string& s : strs) {
    result += s;
}

return result;
}

/***
 * 主函数: 运行测试
 */
int main() {
    cout << "最大数 - 贪心算法解决方案 (C++实现)" << endl;
    cout << "======" << endl;

    // 运行基础测试
    testLargestNumber();

    cout << endl << "调试模式示例:" << endl;
    vector<int> debugNums = {3, 30, 34, 5, 9};
    cout << "对数组 [3,30,34,5,9] 进行调试跟踪:" << endl;
    string debugResult = debugLargestNumber(debugNums);
    cout << "最终结果: " << debugResult << endl;

    cout << endl << "算法分析:" << endl;
    cout << "- 时间复杂度: O(n*logn*m) - 其中 n 是数组长度, m 是数字的平均位数" << endl;
    cout << "- 空间复杂度: O(n*m) - 需要将整数转换为字符串存储" << endl;
    cout << "- 贪心策略: 比较 a+b 和 b+a 的大小, a+b 大的排在前面" << endl;
    cout << "- 最优性: 这种贪心策略能够得到全局最优解" << endl;

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

```
=====
package class089;

import java.util.ArrayList;
import java.util.Arrays;

/***
 * 最大数 - 贪心算法解决方案
 *
 * 题目描述:
 * 给定一组非负整数 nums，重新排列每个数的顺序（每个数不可拆分）使之组成一个最大的整数
 *
 * 测试链接: https://leetcode.cn/problems/largest-number/
 *
 * 算法思想:
 * 使用贪心算法 + 自定义排序，关键是比较两个字符串 a 和 b 时，比较 a+b 和 b+a 的大小
 * 如果 a+b > b+a，则 a 应该排在 b 前面，这样拼接后的结果最大
 *
 * 时间复杂度分析:
 * O(n*logn*m) - 其中 n 是数组长度，m 是数字的平均位数
 * - 排序需要 O(n*logn) 次比较
 * - 每次比较需要 O(m) 时间（字符串拼接和比较）
 *
 * 空间复杂度分析:
 * O(n*m) - 需要将整数转换为字符串存储
 *
 * 是否为最优解:
 * 是，这是解决该问题的最优解
 *
 * 工程化考量:
 * 1. 边界条件处理：处理全为 0 的特殊情况
 * 2. 输入验证：检查输入参数的有效性
 * 3. 异常处理：对非法输入进行检查
 * 4. 可读性：使用清晰的变量命名和详细的注释
 *
 * 贪心策略证明:
 * 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
 * 这种排序方式满足传递性，因此可以得到全局最优解
 */
public class Code01_LargestNumber {

    /**
     * 暴力方法：生成所有可能的排列，选择字典序最小的结果
}
```

```

* 用于验证贪心算法的正确性
*
* @param strs 字符串数组
* @return 字典序最小的拼接结果
*
* 时间复杂度: O(n! * n) - 生成所有排列并比较
* 空间复杂度: O(n!) - 存储所有排列结果
*/
public static String way1(String[] strs) {
    ArrayList<String> ans = new ArrayList<>();
    f(strs, 0, ans);
    ans.sort((a, b) -> a.compareTo(b));
    return ans.get(0);
}

/***
* 递归生成所有排列
*
* @param strs 字符串数组
* @param i 当前处理的位置
* @param ans 存储所有排列结果的列表
*/
public static void f(String[] strs, int i, ArrayList<String> ans) {
    if (i == strs.length) {
        StringBuilder path = new StringBuilder();
        for (String s : strs) {
            path.append(s);
        }
        ans.add(path.toString());
    } else {
        for (int j = i; j < strs.length; j++) {
            swap(strs, i, j);
            f(strs, i + 1, ans);
            swap(strs, i, j);
        }
    }
}

/***
* 交换数组中两个元素的位置
*
* @param strs 字符串数组
* @param i 第一个索引

```

```

* @param j 第二个索引
*/
public static void swap(String[] strs, int i, int j) {
    String tmp = strs[i];
    strs[i] = strs[j];
    strs[j] = tmp;
}

/***
 * 贪心算法：使用自定义排序规则
 *
 * @param strs 字符串数组
 * @return 字典序最小的拼接结果
 *
 * 时间复杂度：O(n*logn)
 * 空间复杂度：O(n)
 */
public static String way2(String[] strs) {
    Arrays.sort(strs, (a, b) -> (a + b).compareTo(b + a));
    StringBuilder path = new StringBuilder();
    for (int i = 0; i < strs.length; i++) {
        path.append(strs[i]);
    }
    return path.toString();
}

/***
 * 生成长度 1~n 的随机字符串数组
 *
 * @param n 最大数组长度
 * @param m 最大字符串长度
 * @param v 字符种类数
 * @return 随机字符串数组
 */
public static String[] randomStringArray(int n, int m, int v) {
    String[] ans = new String[(int) (Math.random() * n) + 1];
    for (int i = 0; i < ans.length; i++) {
        ans[i] = randomString(m, v);
    }
    return ans;
}

/***

```

```

* 生成长度 1~m，字符种类有 v 种的随机字符串
*
* @param m 最大字符串长度
* @param v 字符种类数
* @return 随机字符串
*/
public static String randomString(int m, int v) {
    int len = (int) (Math.random() * m) + 1;
    char[] ans = new char[len];
    for (int i = 0; i < len; i++) {
        ans[i] = (char) ('a' + (int) (Math.random() * v));
    }
    return String.valueOf(ans);
}

/**
* 对数器：验证贪心算法的正确性
*
* 测试策略：
* 1. 生成大量随机测试用例
* 2. 比较暴力解法和贪心解法的结果
* 3. 如果结果不一致，输出错误信息
*/
public static void main(String[] args) {
    int n = 8; // 数组中最多几个字符串
    int m = 5; // 字符串长度最大多长
    int v = 4; // 字符的种类有几种
    int testTimes = 2000;
    System.out.println("测试开始");
    for (int i = 1; i <= testTimes; i++) {
        String[] strs = randomStringArray(n, m, v);
        String ans1 = way1(strs);
        String ans2 = way2(strs);
        if (!ans1.equals(ans2)) {
            // 如果出错了，可以增加打印行为找到一组出错的例子，然后去 debug
            System.out.println("出错了！");
        }
        if (i % 100 == 0) {
            System.out.println("测试到第" + i + "组");
        }
    }
    System.out.println("测试结束");
}

```

```
/**  
 * 解决最大数问题的核心方法  
 *  
 * @param nums 非负整数数组  
 * @return 重新排列后组成的大整数（字符串形式）  
 *  
 * 算法步骤：  
 * 1. 将整数数组转换为字符串数组  
 * 2. 使用自定义比较器对字符串数组进行排序  
 * 3. 处理全为 0 的特殊情况  
 * 4. 拼接排序后的字符串  
 *  
 * 特殊处理：  
 * 如果排序后的第一个字符串是“0”，说明所有数字都是 0，直接返回“0”  
 * 避免出现“000”这样的结果，应该返回“0”  
 */  
  
public static String largestNumber(int[] nums) {  
    // 输入验证  
    if (nums == null || nums.length == 0) {  
        return "0";  
    }  
  
    int n = nums.length;  
    String[] strs = new String[n];  
    for (int i = 0; i < n; i++) {  
        strs[i] = String.valueOf(nums[i]);  
    }  
  
    // 自定义排序：比较 a+b 和 b+a 的大小  
    // 如果 b+a > a+b，则 a 应该排在 b 前面（降序排列）  
    Arrays.sort(strs, (a, b) -> (b + a).compareTo(a + b));  
  
    // 处理全为 0 的特殊情况  
    if (strs[0].equals("0")) {  
        return "0";  
    }  
  
    // 拼接所有字符串  
    StringBuilder path = new StringBuilder();  
    for (String s : strs) {  
        path.append(s);  
    }
```

```
    return path.toString();
}

/**
 * 测试函数：验证最大数算法的正确性
 */
public static void testLargestNumber() {
    // 测试用例 1: [10, 2]
    int[] nums1 = {10, 2};
    String result1 = largestNumber(nums1);
    System.out.println("输入: [10, 2]");
    System.out.println("输出: " + result1);
    System.out.println("预期: \"210\"");
    System.out.println();

    // 测试用例 2: [3, 30, 34, 5, 9]
    int[] nums2 = {3, 30, 34, 5, 9};
    String result2 = largestNumber(nums2);
    System.out.println("输入: [3, 30, 34, 5, 9]");
    System.out.println("输出: " + result2);
    System.out.println("预期: \"9534330\"");
    System.out.println();

    // 测试用例 3: [0, 0, 0] - 全为 0 的特殊情况
    int[] nums3 = {0, 0, 0};
    String result3 = largestNumber(nums3);
    System.out.println("输入: [0, 0, 0]");
    System.out.println("输出: " + result3);
    System.out.println("预期: \"0\"");
    System.out.println();

    // 测试用例 4: [1] - 单个元素
    int[] nums4 = {1};
    String result4 = largestNumber(nums4);
    System.out.println("输入: [1]");
    System.out.println("输出: " + result4);
    System.out.println("预期: \"1\"");
    System.out.println();

    // 测试用例 5: [432, 43243] - 复杂比较
    int[] nums5 = {432, 43243};
    String result5 = largestNumber(nums5);
    System.out.println("输入: [432, 43243]");
```

```
        System.out.println("输出: " + result5);
        System.out.println("预期: \"43243432\"");
        System.out.println();
    }
}
```

文件: Code01_LargestNumber.py

"""

最大数 - 贪心算法解决方案 (Python 实现)

题目描述:

给定一组非负整数 nums , 重新排列每个数的顺序 (每个数不可拆分) 使之组成一个最大的整数

测试链接: <https://leetcode.cn/problems/largest-number/>

算法思想:

使用贪心算法 + 自定义排序, 关键是比较两个字符串 a 和 b 时, 比较 $a+b$ 和 $b+a$ 的大小

如果 $a+b > b+a$, 则 a 应该排在 b 前面, 这样拼接后的结果最大

时间复杂度分析:

$O(n * \log n * m)$ - 其中 n 是数组长度, m 是数字的平均位数

- 排序需要 $O(n * \log n)$ 次比较

- 每次比较需要 $O(m)$ 时间 (字符串拼接和比较)

空间复杂度分析:

$O(n * m)$ - 需要将整数转换为字符串存储

是否为最优解:

是, 这是解决该问题的最优解

工程化考量:

1. 边界条件处理: 处理全为 0 的特殊情况
2. 输入验证: 检查输入参数的有效性
3. 异常处理: 对非法输入进行检查
4. 可读性: 使用清晰的变量命名和详细的注释

贪心策略证明:

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

这种排序方式满足传递性, 因此可以得到全局最优解

"""

```
from functools import cmp_to_key
from typing import List

class Code01_LargestNumber:

    @staticmethod
    def largestNumber(nums: List[int]) -> str:
        """

```

解决最大数问题的核心方法

Args:

 nums: 非负整数数组

Returns:

 重新排列后组成的大整数（字符串形式）

算法步骤:

1. 将整数数组转换为字符串数组
2. 使用自定义比较器对字符串数组进行排序
3. 处理全为 0 的特殊情况
4. 拼接排序后的字符串

特殊处理:

如果排序后的第一个字符串是"0", 说明所有数字都是 0, 直接返回"0"

避免出现"000"这样的结果, 应该返回"0"

"""

输入验证

```
if not nums:
```

```
    return "0"
```

将整数转换为字符串

```
strs = [str(num) for num in nums]
```

自定义比较函数

```
def compare(a: str, b: str) -> int:
```

"""

自定义比较函数

比较规则: 比较 $a+b$ 和 $b+a$ 的大小

如果 $a+b > b+a$, 则 a 应该排在 b 前面 (返回-1 表示 a 排在 b 前面)

Args:

 a: 第一个字符串

b: 第二个字符串

Returns:

比较结果: -1 表示 a 排在 b 前面, 1 表示 b 排在 a 前面

"""

```
if a + b > b + a:  
    return -1 # a 应该排在 b 前面  
else:  
    return 1 # b 应该排在 a 前面
```

使用自定义比较器进行排序

```
strs.sort(key=cmp_to_key(compare))
```

处理全为 0 的特殊情况

```
if strs[0] == "0":  
    return "0"
```

拼接所有字符串

```
return ''.join(strs)
```

@staticmethod

```
def debug_largest_number(nums: List[int]) -> str:
```

"""

调试版本: 打印排序过程中的中间结果

Args:

nums: 整数数组

Returns:

最大数结果

"""

```
if not nums:  
    return "0"
```

```
strs = [str(num) for num in nums]
```

```
print("原始字符串数组:", strs)
```

```
print("比较过程:")
```

打印比较过程

```
for i in range(len(strs)):  
    for j in range(i + 1, len(strs)):  
        ab = strs[i] + strs[j]
```

```

        ba = strs[j] + strs[i]
        print(f"比较 {strs[i]} 和 {strs[j]}: {ab} vs {ba} -> ", end="")
        if ab > ba:
            print(f"{strs[i]} 应该在 {strs[j]} 前面")
        else:
            print(f"{strs[j]} 应该在 {strs[i]} 前面")

# 自定义比较函数
def compare(a: str, b: str) -> int:
    if a + b > b + a:
        return -1
    else:
        return 1

strs.sort(key=cmp_to_key(compare))
print("排序后字符串数组:", strs)

if strs[0] == "0":
    return "0"

return ''.join(strs)

@staticmethod
def test_largest_number():
    """
    测试函数: 验证最大数算法的正确性
    """
    print("最大数算法测试开始")
    print("=====")

    # 测试用例 1: [10, 2]
    nums1 = [10, 2]
    result1 = Code01_LargestNumber.largestNumber(nums1)
    print("输入: [10, 2]")
    print("输出:", result1)
    print("预期: \"210\"")
    print("✓ 通过" if result1 == "210" else "✗ 失败")
    print()

    # 测试用例 2: [3, 30, 34, 5, 9]
    nums2 = [3, 30, 34, 5, 9]
    result2 = Code01_LargestNumber.largestNumber(nums2)
    print("输入: [3, 30, 34, 5, 9]")

```

```

print("输出:", result2)
print("预期: \"9534330\"")
print("✓ 通过" if result2 == "9534330" else "✗ 失败")
print()

# 测试用例 3: [0, 0, 0] - 全为 0 的特殊情况
nums3 = [0, 0, 0]
result3 = Code01_LargestNumber.largestNumber(nums3)
print("输入: [0, 0, 0]")
print("输出:", result3)
print("预期: \"0\"")
print("✓ 通过" if result3 == "0" else "✗ 失败")
print()

# 测试用例 4: [1] - 单个元素
nums4 = [1]
result4 = Code01_LargestNumber.largestNumber(nums4)
print("输入: [1]")
print("输出:", result4)
print("预期: \"1\"")
print("✓ 通过" if result4 == "1" else "✗ 失败")
print()

# 测试用例 5: [432, 43243] - 复杂比较
nums5 = [432, 43243]
result5 = Code01_LargestNumber.largestNumber(nums5)
print("输入: [432, 43243]")
print("输出:", result5)
print("预期: \"43243432\"")
print("✓ 通过" if result5 == "43243432" else "✗ 失败")
print()

print("测试结束")

```

```

@staticmethod
def performance_test():
    """
    性能测试: 测试算法在大规模数据下的表现
    """

    import time
    import random

    print("性能测试开始")

```

```
print("=====")

# 生成大规模测试数据
n = 10000
nums = [random.randint(0, 1000000) for _ in range(n)]


start_time = time.time()
result = Code01_LargestNumber.largestNumber(nums)
end_time = time.time()

print(f"数据规模: {n}")
print(f"执行时间: {end_time - start_time:.4f} 秒")
print(f"结果长度: {len(result)}")
print("性能测试结束")


def main():
    """
    主函数: 运行测试
    """
    print("最大数 - 贪心算法解决方案 (Python 实现)")
    print("=====")

    # 运行基础测试
    Code01_LargestNumber.test_largest_number()

    print("\n 调试模式示例:")
    debug_nums = [3, 30, 34, 5, 9]
    print("对数组 [3,30,34,5,9] 进行调试跟踪:")
    debug_result = Code01_LargestNumber.debug_largest_number(debug_nums)
    print("最终结果:", debug_result)

    print("\n 算法分析:")
    print("- 时间复杂度: O(n * log n * m) - 其中 n 是数组长度, m 是数字的平均位数")
    print("- 空间复杂度: O(n * m) - 需要将整数转换为字符串存储")
    print("- 贪心策略: 比较 a+b 和 b+a 的大小, a+b 大的排在前面")
    print("- 最优性: 这种贪心策略能够得到全局最优解")
    print("- Python 特性: 使用 functools.cmp_to_key 实现自定义比较器")

    # 可选: 运行性能测试 (取消注释以运行)
    # print("\n 性能测试:")
    # Code01_LargestNumber.performance_test()
```

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

文件: Code02_TwoCityScheduling.cpp

```
/***
 * 两地调度 - 贪心算法解决方案 (C++实现)
 *
 * 题目描述:
 * 公司计划面试  $2n$  个人, 给定一个数组 costs, 其中  $\text{costs}[i] = [\text{aCost}_i, \text{bCost}_i]$ 
 * 表示第  $i$  人飞往 a 市的费用为  $\text{aCost}_i$ , 飞往 b 市的费用为  $\text{bCost}_i$ 
 * 返回将每个人都飞到 a、b 中某座城市的最低费用, 要求每个城市都有  $n$  人抵达
 *
 * 测试链接: https://leetcode.cn/problems/two-city-scheduling/
 *
 * 算法思想:
 * 贪心算法 + 差值排序
 * 1. 假设所有人都先去 A 市, 总费用为所有  $\text{aCost}_i$  之和
 * 2. 计算每个人从 A 市改去 B 市的费用变化:  $\text{bCost}_i - \text{aCost}_i$ 
 * 3. 选择费用变化最小的  $n$  个人改去 B 市
 * 4. 总费用 = 所有  $\text{aCost}_i$  之和 + 最小的  $n$  个费用变化值之和
 *
 * 时间复杂度分析:
 *  $O(n \log n)$  - 主要时间消耗在排序上
 *
 * 空间复杂度分析:
 *  $O(n)$  - 需要额外的数组存储费用变化值
 *
 * 是否为最优解:
 * 是, 这是解决该问题的最优解
 *
 * 工程化考量:
 * 1. 边界条件处理: 处理空数组、奇数人数等特殊情况
 * 2. 输入验证: 检查输入参数的有效性
 * 3. 异常处理: 对非法输入进行检查
 * 4. 可读性: 使用清晰的变量命名和详细的注释
 *
 * 贪心策略证明:
 * 对于每个人, 选择去 A 市还是 B 市取决于  $\text{bCost}_i - \text{aCost}_i$  的值
 * 如果这个值很小 (负数), 说明去 B 市比去 A 市便宜很多, 应该优先选择去 B 市
```

```

* 通过排序选择最小的 n 个差值，可以保证总费用最小
*/

```

```

#include <iostream>
#include <vector>
#include <algorithm>
#include <stdexcept>

using namespace std;

/**
 * 计算两地调度的最低费用
 *
 * @param costs 费用数组，每个元素是[aCosti, bCosti]
 * @return 最低总费用
 *
 * 算法步骤：
 * 1. 假设所有人都先去 A 市，计算总费用
 * 2. 计算每个人从 A 市改去 B 市的费用变化
 * 3. 对费用变化进行排序
 * 4. 选择最小的 n 个费用变化值加到总费用中
 *
 * 数学原理：
 * 总费用 =  $\sum aCost_i + \sum (bCost_i - aCost_i) = \sum aCost_i + \sum bCost_i - \sum aCost_i = \sum bCost_i$ 
 * 但这里只选择最小的 n 个差值，所以实际上是部分人去 B 市
*/

```

```

int twoCitySchedCost(vector<vector<int>>& costs) {
    // 输入验证
    if (costs.empty()) {
        return 0;
    }

    int n = costs.size();
    // 检查人数是否为偶数
    if (n % 2 != 0) {
        throw invalid_argument("人数必须为偶数");
    }

    vector<int> diff(n); // 存储每个人从 A 市改去 B 市的费用变化
    int sumA = 0; // 所有人都去 A 市的总费用

    // 计算费用变化和 A 市总费用
    for (int i = 0; i < n; i++) {

```

```

    sumA += costs[i][0];           // 累加 A 市费用
    diff[i] = costs[i][1] - costs[i][0]; // 计算费用变化
}

// 对费用变化进行排序（升序）
sort(diff.begin(), diff.end());

int m = n / 2; // 每个城市需要的人数
// 选择最小的 m 个费用变化值
for (int i = 0; i < m; i++) {
    sumA += diff[i];
}

return sumA;
}

```

```

/**
 * 调试版本：打印计算过程中的中间结果
 *
 * @param costs 费用数组
 * @return 最低总费用
 */
int debugTwoCitySchedCost(vector<vector<int>>& costs) {
    if (costs.empty()) {
        return 0;
    }

    int n = costs.size();
    if (n % 2 != 0) {
        throw invalid_argument("人数必须为偶数");
    }

    vector<int> diff(n);
    int sumA = 0;

    cout << "原始费用数据:" << endl;
    cout << "序号\tA 市费用\tB 市费用\t费用变化(B-A)" << endl;
    for (int i = 0; i < n; i++) {
        int aCost = costs[i][0];
        int bCost = costs[i][1];
        int change = bCost - aCost;
        diff[i] = change;
        sumA += aCost;
    }
}
```

```

    cout << i << "\t" << aCost << "\t" << bCost << "\t" << change << endl;
}

cout << endl << "所有人都去 A 市的总费用: " << sumA << endl;

// 打印排序前的费用变化
cout << "排序前的费用变化数组: [";
for (int i = 0; i < n; i++) {
    cout << diff[i];
    if (i < n - 1) cout << ", ";
}
cout << "]" << endl;

sort(diff.begin(), diff.end());
cout << "排序后的费用变化数组: [";
for (int i = 0; i < n; i++) {
    cout << diff[i];
    if (i < n - 1) cout << ", ";
}
cout << "]" << endl;

int m = n / 2;
cout << "每个城市需要的人数: " << m << endl;
cout << "选择最小的" << m << "个费用变化值:" << endl;

int changeSum = 0;
for (int i = 0; i < m; i++) {
    changeSum += diff[i];
    cout << "选择第" << (i+1) << "个变化值: " << diff[i] << ", 累计变化: " << changeSum <<
endl;
}

int totalCost = sumA + changeSum;
cout << "最终总费用: " << totalCost << endl;

return totalCost;
}

/***
 * 打印二维数组的辅助函数
 *
 * @param arr 二维数组
 */

```

```

void printCosts(const vector<vector<int>>& costs) {
    cout << "[";
    for (int i = 0; i < costs.size(); i++) {
        cout << "[" << costs[i][0] << "," << costs[i][1] << "]";
        if (i < costs.size() - 1) cout << ",";
    }
    cout << "]";
}

/***
 * 测试函数：验证两地调度算法的正确性
 */
void testTwoCitySchedCost () {
    cout << "两地调度算法测试开始" << endl;
    cout << "======" << endl;

    // 测试用例 1: [[10, 20], [30, 200], [400, 50], [30, 20]]
    vector<vector<int>> costs1 = {{10, 20}, {30, 200}, {400, 50}, {30, 20}};
    int result1 = twoCitySchedCost(costs1);
    cout << "输入: ";
    printCosts(costs1);
    cout << endl;
    cout << "输出: " << result1 << endl;
    cout << "预期: 110" << endl;
    cout << (result1 == 110 ? "✓ 通过" : "✗ 失败") << endl << endl;

    // 测试用例 2: [[259, 770], [448, 54], [926, 667], [184, 139], [840, 118], [577, 469]]
    vector<vector<int>> costs2 = {{259, 770}, {448, 54}, {926, 667}, {184, 139}, {840, 118},
{577, 469}};
    int result2 = twoCitySchedCost(costs2);
    cout << "输入: 6 个人的测试用例" << endl;
    cout << "输出: " << result2 << endl;
    cout << "预期: 1859" << endl;
    cout << (result2 == 1859 ? "✓ 通过" : "✗ 失败") << endl << endl;

    // 测试用例 3: [[515, 563], [451, 713], [537, 709], [343, 819], [855, 779], [457, 60], [650, 359], [631, 42]]
    vector<vector<int>> costs3 = {{515, 563}, {451, 713}, {537, 709}, {343, 819}, {855, 779},
{457, 60}, {650, 359}, {631, 42}};
    int result3 = twoCitySchedCost(costs3);
    cout << "输入: 8 个人的复杂测试用例" << endl;
    cout << "输出: " << result3 << endl;
    cout << "预期: 3086" << endl;
    cout << (result3 == 3086 ? "✓ 通过" : "✗ 失败") << endl << endl;
}

```

```
    cout << "测试结束" << endl;
}

/***
 * 性能测试：测试算法在大规模数据下的表现
 */
void performanceTest() {
    cout << "性能测试开始" << endl;
    cout << "======" << endl;

    // 生成大规模测试数据
    int n = 10000; // 5000 对人
    vector<vector<int>> costs;
    for (int i = 0; i < n; i++) {
        int aCost = rand() % 1000 + 1;
        int bCost = rand() % 1000 + 1;
        costs.push_back({aCost, bCost});
    }

    clock_t start = clock();
    int result = twoCitySchedCost(costs);
    clock_t end = clock();

    double duration = double(end - start) / CLOCKS_PER_SEC;

    cout << "数据规模：" << n << " 对人" << endl;
    cout << "执行时间：" << duration << " 秒" << endl;
    cout << "计算结果：" << result << endl;
    cout << "性能测试结束" << endl;
}

/***
 * 主函数：运行测试
 */
int main() {
    cout << "两地调度 - 贪心算法解决方案 (C++实现)" << endl;
    cout << "======" << endl;

    // 运行基础测试
    testTwoCitySchedCost();

    cout << endl << "调试模式示例：" << endl;
```

```

vector<vector<int>> debugCosts = {{10, 20}, {30, 200}, {400, 50}, {30, 20}};
cout << "对测试用例 [[10, 20], [30, 200], [400, 50], [30, 20]] 进行调试跟踪:" << endl;
int debugResult = debugTwoCitySchedCost(debugCosts);
cout << "最终结果: " << debugResult << endl;

cout << endl << "算法分析:" << endl;
cout << "- 时间复杂度: O(n*logn) - 主要时间消耗在排序上" << endl;
cout << "- 空间复杂度: O(n) - 需要额外的数组存储费用变化值" << endl;
cout << "- 贪心策略: 假设所有人都去 A 市, 然后选择费用变化最小的 n 个人改去 B 市" << endl;
cout << "- 最优性: 这种贪心策略能够得到全局最优解" << endl;
cout << "- 数学原理: 总费用 =  $\sum aCost_i + \min_n(bCost_i - aCost_i)$ " << endl;

// 可选: 运行性能测试
// cout << endl << "性能测试:" << endl;
// performanceTest();

return 0;
}

```

=====

文件: Code02_TwoCityScheduling.java

```

package class089;

import java.util.Arrays;

/**
 * 两地调度 - 贪心算法解决方案
 *
 * 题目描述:
 * 公司计划面试 2n 个人, 给定一个数组 costs, 其中 costs[i]=[aCosti, bCosti]
 * 表示第 i 人飞往 a 市的费用为 aCosti, 飞往 b 市的费用为 bCosti
 * 返回将每个人都飞到 a、b 中某座城市的最低费用, 要求每个城市都有 n 人抵达
 *
 * 测试链接: https://leetcode.cn/problems/two-city-scheduling/
 *
 * 算法思想:
 * 贪心算法 + 差值排序
 * 1. 假设所有人都先去 A 市, 总费用为所有 aCosti 之和
 * 2. 计算每个人从 A 市改去 B 市的费用变化: bCosti - aCosti
 * 3. 选择费用变化最小的 n 个人改去 B 市
 * 4. 总费用 = 所有 aCosti 之和 + 最小的 n 个费用变化值之和

```

```

*
* 时间复杂度分析:
* O(n*logn) - 主要时间消耗在排序上
*
* 空间复杂度分析:
* O(n) - 需要额外的数组存储费用变化值
*
* 是否为最优解:
* 是, 这是解决该问题的最优解
*
* 工程化考量:
* 1. 边界条件处理: 处理空数组、奇数人数等特殊情况
* 2. 输入验证: 检查输入参数的有效性
* 3. 异常处理: 对非法输入进行检查
* 4. 可读性: 使用清晰的变量命名和详细的注释
*
* 贪心策略证明:
* 对于每个人, 选择去 A 市还是 B 市取决于 bCosti - aCosti 的值
* 如果这个值很小 (负数), 说明去 B 市比去 A 市便宜很多, 应该优先选择去 B 市
* 通过排序选择最小的 n 个差值, 可以保证总费用最小
*/
public class Code02_TwoCityScheduling {

    /**
     * 计算两地调度的最低费用
     *
     * @param costs 费用数组, 每个元素是[aCosti, bCosti]
     * @return 最低总费用
     *
     * 算法步骤:
     * 1. 假设所有人都先去 A 市, 计算总费用
     * 2. 计算每个人从 A 市改去 B 市的费用变化
     * 3. 对费用变化进行排序
     * 4. 选择最小的 n 个费用变化值加到总费用中
     *
     * 数学原理:
     * 总费用 =  $\sum aCosti + \sum (bCosti - aCosti) = \sum aCosti + \sum bCosti - \sum aCosti = \sum bCosti$ 
     * 但这里只选择最小的 n 个差值, 所以实际上是部分人去 B 市
    */
    public static int twoCitySchedCost(int[][] costs) {
        // 输入验证
        if (costs == null || costs.length == 0) {
            return 0;
        }
        ...
    }
}

```

```
}

int n = costs.length;
// 检查人数是否为偶数
if (n % 2 != 0) {
    throw new IllegalArgumentException("人数必须为偶数");
}

int[] diff = new int[n]; // 存储每个人从 A 市改去 B 市的费用变化
int sumA = 0;           // 所有人都去 A 市的总费用

// 计算费用变化和 A 市总费用
for (int i = 0; i < n; i++) {
    sumA += costs[i][0];           // 累加 A 市费用
    diff[i] = costs[i][1] - costs[i][0]; // 计算费用变化
}

// 对费用变化进行排序 (升序)
Arrays.sort(diff);

int m = n / 2; // 每个城市需要的人数
// 选择最小的 m 个费用变化值
for (int i = 0; i < m; i++) {
    sumA += diff[i];
}

return sumA;
}

/***
 * 调试版本：打印计算过程中的中间结果
 *
 * @param costs 费用数组
 * @return 最低总费用
 */
public static int debugTwoCitySchedCost(int[][] costs) {
    if (costs == null || costs.length == 0) {
        return 0;
    }

    int n = costs.length;
    if (n % 2 != 0) {
        throw new IllegalArgumentException("人数必须为偶数");
    }
}
```

```

}

int[] diff = new int[n];
int sumA = 0;

System.out.println("原始费用数据:");
System.out.println("序号\tA市费用\tB市费用\t费用变化(B-A)");
for (int i = 0; i < n; i++) {
    int aCost = costs[i][0];
    int bCost = costs[i][1];
    int change = bCost - aCost;
    diff[i] = change;
    sumA += aCost;
    System.out.printf("%d\t%d\t%d\t%d\n", i, aCost, bCost, change);
}

System.out.println("所有人都去A市的总费用: " + sumA);

// 打印排序前的费用变化
System.out.println("排序前的费用变化数组: " + Arrays.toString(diff));

Arrays.sort(diff);
System.out.println("排序后的费用变化数组: " + Arrays.toString(diff));

int m = n / 2;
System.out.println("每个城市需要的人数: " + m);
System.out.println("选择最小的" + m + "个费用变化值:");

int changeSum = 0;
for (int i = 0; i < m; i++) {
    changeSum += diff[i];
    System.out.println("选择第" + (i+1) + "个变化值: " + diff[i] + ", 累计变化: " +
changeSum);
}

int totalCost = sumA + changeSum;
System.out.println("最终总费用: " + totalCost);

return totalCost;
}

/**
 * 测试函数: 验证两地调度算法的正确性

```

```

*/
public static void testTwoCitySchedCost() {
    System.out.println("两地调度算法测试开始");
    System.out.println("====");

    // 测试用例 1: [[10, 20], [30, 200], [400, 50], [30, 20]]
    int[][] costs1 = {{10, 20}, {30, 200}, {400, 50}, {30, 20}};
    int result1 = twoCitySchedCost(costs1);
    System.out.println("输入: [[10, 20], [30, 200], [400, 50], [30, 20]]");
    System.out.println("输出: " + result1);
    System.out.println("预期: 110");
    System.out.println((result1 == 110 ? "✓ 通过" : "✗ 失败"));
    System.out.println();

    // 测试用例 2: [[259, 770], [448, 54], [926, 667], [184, 139], [840, 118], [577, 469]]
    int[][] costs2 = {{259, 770}, {448, 54}, {926, 667}, {184, 139}, {840, 118}, {577, 469}};
    int result2 = twoCitySchedCost(costs2);
    System.out.println("输入: [[259, 770], [448, 54], [926, 667], [184, 139], [840, 118], [577, 469]]");
    System.out.println("输出: " + result2);
    System.out.println("预期: 1859");
    System.out.println((result2 == 1859 ? "✓ 通过" : "✗ 失败"));
    System.out.println();

    // 测试用例 3:
    [[515, 563], [451, 713], [537, 709], [343, 819], [855, 779], [457, 60], [650, 359], [631, 42]]
    int[][] costs3 = {{515, 563}, {451, 713}, {537, 709}, {343, 819}, {855, 779}, {457, 60},
    {650, 359}, {631, 42}};
    int result3 = twoCitySchedCost(costs3);
    System.out.println("输入: 8 个人的复杂测试用例");
    System.out.println("输出: " + result3);
    System.out.println("预期: 3086");
    System.out.println((result3 == 3086 ? "✓ 通过" : "✗ 失败"));
    System.out.println();

    System.out.println("测试结束");
}

/**
 * 主函数: 运行测试
 */
public static void main(String[] args) {
    System.out.println("两地调度 - 贪心算法解决方案");
    System.out.println("====");
}

```

```

// 运行基础测试
testTwoCitySchedCost();

System.out.println("调试模式示例:");
int[][] debugCosts = {{10, 20}, {30, 200}, {400, 50}, {30, 20}};
System.out.println("对测试用例 [[10, 20], [30, 200], [400, 50], [30, 20]] 进行调试跟踪:");
int debugResult = debugTwoCitySchedCost(debugCosts);
System.out.println("最终结果: " + debugResult);

System.out.println("算法分析:");
System.out.println("- 时间复杂度: O(n*logn) - 主要时间消耗在排序上");
System.out.println("- 空间复杂度: O(n) - 需要额外的数组存储费用变化值");
System.out.println("- 贪心策略: 假设所有人都去 A 市, 然后选择费用变化最小的 n 个人改去 B 市");
System.out.println("- 最优性: 这种贪心策略能够得到全局最优解");
System.out.println("- 数学原理: 总费用 = Σ aCosti + min_n(bCosti - aCosti)");
}

}
=====

文件: Code02_TwoCityScheduling.py
=====
"""
两地调度 - 贪心算法解决方案 (Python 实现)

```

题目描述:

公司计划面试 $2n$ 个人, 给定一个数组 costs , 其中 $\text{costs}[i] = [\text{aCost}_i, \text{bCost}_i]$

表示第 i 人飞往 A 市的费用为 aCost_i , 飞往 B 市的费用为 bCost_i

返回将每个人都飞到 A 、 B 中某座城市的最低费用, 要求每个城市都有 n 人抵达

测试链接: <https://leetcode.cn/problems/two-city-scheduling/>

算法思想:

贪心算法 + 差值排序

1. 假设所有人都先去 A 市, 总费用为所有 aCost_i 之和
2. 计算每个人从 A 市改去 B 市的费用变化: $\text{bCost}_i - \text{aCost}_i$
3. 选择费用变化最小的 n 个人改去 B 市
4. 总费用 = 所有 aCost_i 之和 + 最小的 n 个费用变化值之和

时间复杂度分析:

$O(n * \log n)$ - 主要时间消耗在排序上

空间复杂度分析:

$O(n)$ - 需要额外的数组存储费用变化值

是否为最优解:

是, 这是解决该问题的最优解

工程化考量:

1. 边界条件处理: 处理空数组、奇数人数等特殊情况
2. 输入验证: 检查输入参数的有效性
3. 异常处理: 对非法输入进行检查
4. 可读性: 使用清晰的变量命名和详细的注释

贪心策略证明:

对于每个人, 选择去 A 市还是 B 市取决于 $bCosti - aCosti$ 的值

如果这个值很小 (负数), 说明去 B 市比去 A 市便宜很多, 应该优先选择去 B 市

通过排序选择最小的 n 个差值, 可以保证总费用最小

"""

```
from typing import List
```

```
class Code02_TwoCityScheduling:
```

```
    @staticmethod
```

```
    def two_city_sched_cost(costs: List[List[int]]) -> int:
```

```
        """
```

计算两地调度的最低费用

Args:

costs: 费用数组, 每个元素是 $[aCosti, bCosti]$

Returns:

最低总费用

算法步骤:

1. 假设所有人都先去 A 市, 计算总费用
2. 计算每个人从 A 市改去 B 市的费用变化
3. 对费用变化进行排序
4. 选择最小的 n 个费用变化值加到总费用中

数学原理:

$\text{总费用} = \sum aCosti + \sum (bCosti - aCosti) = \sum aCosti + \sum bCosti - \sum aCosti = \sum bCosti$
但这里只选择最小的 n 个差值, 所以实际上是部分人去 B 市

```
"""
# 输入验证
if not costs:
    return 0

n = len(costs)
# 检查人数是否为偶数
if n % 2 != 0:
    raise ValueError("人数必须为偶数")

diff = [0] * n # 存储每个人从 A 市改去 B 市的费用变化
sum_a = 0       # 所有人都去 A 市的总费用

# 计算费用变化和 A 市总费用
for i in range(n):
    sum_a += costs[i][0]           # 累加 A 市费用
    diff[i] = costs[i][1] - costs[i][0] # 计算费用变化

# 对费用变化进行排序 (升序)
diff.sort()

m = n // 2 # 每个城市需要的人数
# 选择最小的 m 个费用变化值
for i in range(m):
    sum_a += diff[i]

return sum_a
```

```
@staticmethod
def debug_two_city_sched_cost(costs: List[List[int]]) -> int:
    """
```

调试版本：打印计算过程中的中间结果

Args:

costs: 费用数组

Returns:

最低总费用

```
if not costs:
    return 0
```

n = len(costs)

```
if n % 2 != 0:  
    raise ValueError("人数必须为偶数")  
  
diff = [0] * n  
sum_a = 0  
  
print("原始费用数据:")  
print("序号\tA 市费用\tB 市费用\t费用变化(B-A)")  
for i in range(n):  
    a_cost = costs[i][0]  
    b_cost = costs[i][1]  
    change = b_cost - a_cost  
    diff[i] = change  
    sum_a += a_cost  
    print(f"{i}\t{a_cost}\t{b_cost}\t{change}")  
  
print(f"\n所有人都去 A 市的总费用: {sum_a}")  
  
# 打印排序前的费用变化  
print(f"排序前的费用变化数组: {diff}")  
  
diff.sort()  
print(f"排序后的费用变化数组: {diff}")  
  
m = n // 2  
print(f"每个城市需要的人数: {m}")  
print(f"选择最小的 {m} 个费用变化值:")  
  
change_sum = 0  
for i in range(m):  
    change_sum += diff[i]  
    print(f"选择第 {i+1} 个变化值: {diff[i]}, 累计变化: {change_sum}")  
  
total_cost = sum_a + change_sum  
print(f"最终总费用: {total_cost}")  
  
return total_cost  
  
@staticmethod  
def test_two_city_sched_cost():  
    """  
    测试函数: 验证两地调度算法的正确性  
    """
```

```

print("两地调度算法测试开始")
print("====")

# 测试用例 1: [[10, 20], [30, 200], [400, 50], [30, 20]]
costs1 = [[10, 20], [30, 200], [400, 50], [30, 20]]
result1 = Code02_TwoCityScheduling. two_city_sched_cost(costs1)
print("输入: [[10, 20], [30, 200], [400, 50], [30, 20]]")
print("输出:", result1)
print("预期: 110")
print("✓ 通过" if result1 == 110 else "✗ 失败")
print()

# 测试用例 2: [[259, 770], [448, 54], [926, 667], [184, 139], [840, 118], [577, 469]]
costs2 = [[259, 770], [448, 54], [926, 667], [184, 139], [840, 118], [577, 469]]
result2 = Code02_TwoCityScheduling. two_city_sched_cost(costs2)
print("输入: 6 个人的测试用例")
print("输出:", result2)
print("预期: 1859")
print("✓ 通过" if result2 == 1859 else "✗ 失败")
print()

# 测试用例 3:
[[515, 563], [451, 713], [537, 709], [343, 819], [855, 779], [457, 60], [650, 359], [631, 42]]
costs3 = [[515, 563], [451, 713], [537, 709], [343, 819], [855, 779], [457, 60], [650, 359],
[631, 42]]
result3 = Code02_TwoCityScheduling. two_city_sched_cost(costs3)
print("输入: 8 个人的复杂测试用例")
print("输出:", result3)
print("预期: 3086")
print("✓ 通过" if result3 == 3086 else "✗ 失败")
print()

print("测试结束")

@staticmethod
def performance_test():
    """
    性能测试: 测试算法在大规模数据下的表现
    """
    import time
    import random

    print("性能测试开始")

```

```

print("=====")

# 生成大规模测试数据
n = 10000 # 5000 对人
costs = []
for i in range(n):
    a_cost = random.randint(1, 1000)
    b_cost = random.randint(1, 1000)
    costs.append([a_cost, b_cost])

start_time = time.time()
result = Code02_TwoCityScheduling.two_city_sched_cost(costs)
end_time = time.time()

print(f"数据规模: {n} 对人")
print(f"执行时间: {end_time - start_time:.4f} 秒")
print(f"计算结果: {result}")
print("性能测试结束")

def main():
    """
    主函数: 运行测试
    """
    print("两地调度 - 贪心算法解决方案 (Python 实现)")
    print("=====")

    # 运行基础测试
    Code02_TwoCityScheduling.test_two_city_sched_cost()

    print("\n 调试模式示例:")
    debug_costs = [[10, 20], [30, 200], [400, 50], [30, 20]]
    print("对测试用例 [[10, 20], [30, 200], [400, 50], [30, 20]] 进行调试跟踪:")
    debug_result = Code02_TwoCityScheduling.debug_two_city_sched_cost(debug_costs)
    print("最终结果:", debug_result)

    print("\n 算法分析:")
    print("- 时间复杂度: O(n*logn) - 主要时间消耗在排序上")
    print("- 空间复杂度: O(n) - 需要额外的数组存储费用变化值")
    print("- 贪心策略: 假设所有人都去 A 市, 然后选择费用变化最小的 n 个人改去 B 市")
    print("- 最优性: 这种贪心策略能够得到全局最优解")
    print("- 数学原理: 总费用 = Σ aCosti + min_n(bCosti - aCosti)")



```

```
# 可选：运行性能测试
# print("\n 性能测试:")
# Code02_TwoCityScheduling.performance_test()

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

=====

文件: Code03_MinimumNumberEatOranges.cpp

=====

```
/***
 * 吃掉 N 个橘子的最少天数 - 贪心算法 + 记忆化搜索解决方案 (C++实现)
 *
 * 题目描述:
 * 厨房里总共有 n 个橘子，你决定每一天选择如下方式之一吃这些橘子
 * 1) 吃掉一个橘子
 * 2) 如果剩余橘子数 n 能被 2 整除，那么你可以吃掉 n/2 个橘子
 * 3) 如果剩余橘子数 n 能被 3 整除，那么你可以吃掉 2*(n/3) 个橘子
 * 每天你只能从以上 3 种方案中选择一种方案
 * 请你返回吃掉所有 n 个橘子的最少天数
 *
 * 测试链接: https://leetcode.cn/problems/minimum-number-of-days-to-eat-n-oranges/
 *
 * 算法思想:
 * 贪心算法 + 记忆化搜索 (动态规划)
 * 1. 优先使用按比例吃橘子的方法 (方法 2 和 3)，因为这样能更快减少橘子数量
 * 2. 对于每个 n，考虑两种可能性:
 *     - 先吃到 2 的倍数，然后吃掉一半
 *     - 先吃到 3 的倍数，然后吃掉三分之二
 * 3. 选择天数最少的方法
 *
 * 时间复杂度分析:
 * O(logn) - 由于每次递归都会将问题规模减半或减为三分之一
 *
 * 空间复杂度分析:
 * O(logn) - 递归深度为 logn，记忆化存储也需要 O(logn) 空间
 *
 * 是否为最优解:
 * 是，这是解决该问题的最优解
 *
 * 工程化考量:
```

```
* 1. 边界条件处理：处理 n=0 和 n=1 的特殊情况
* 2. 记忆化优化：避免重复计算
* 3. 递归深度：使用记忆化搜索控制递归深度
* 4. 可读性：使用清晰的变量命名和详细的注释
*
* 贪心策略证明：
* 由于按比例吃橘子（方法 2 和 3）能更快减少橘子数量，所以应该优先考虑这两种方法
* 对于每个 n，选择能最快减少橘子数量的方法
*/
```

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

using namespace std;

// 全局记忆化存储表
unordered_map<int, int> dp;

/***
 * 计算吃掉 n 个橘子的最少天数
 *
 * @param n 橘子数量
 * @return 最少天数
 *
 * 算法步骤：
 * 1. 基础情况：n=0 或 1 时，直接返回 n
 * 2. 如果已经计算过 n，直接返回结果
 * 3. 考虑两种贪心策略：
 *    - 先吃到 2 的倍数，然后吃掉一半
 *    - 先吃到 3 的倍数，然后吃掉三分之二
 * 4. 选择天数最少的方法
 *
 * 贪心策略解释：
 * 方法 2 和 3 能更快减少橘子数量，所以应该优先考虑
 * 但需要先通过方法 1 吃到合适的倍数
*/
```

```
int minDays(int n) {
    // 基础情况处理
    if (n <= 1) {
        return n;
    }
```

```

// 记忆化检查：如果已经计算过，直接返回结果
if (dp.find(n) != dp.end()) {
    return dp[n];
}

// 贪心策略 1：先吃到 2 的倍数，然后吃掉一半
// 需要先吃掉 n % 2 个橘子（方法 1），然后使用方法 2 吃掉一半
int option1 = n % 2 + 1 + minDays(n / 2);

// 贪心策略 2：先吃到 3 的倍数，然后吃掉三分之二
// 需要先吃掉 n % 3 个橘子（方法 1），然后使用方法 3 吃掉三分之二
int option2 = n % 3 + 1 + minDays(n / 3);

// 选择天数最少的方法
int ans = min(option1, option2);

// 存储结果到记忆化表
dp[n] = ans;
return ans;
}

/***
 * 调试版本：打印计算过程中的中间结果
 *
 * @param n 橘子数量
 * @return 最少天数
 */
int debugMinDays(int n) {
    cout << "计算吃掉 " << n << " 个橘子的最少天数：" << endl;

    if (n <= 1) {
        cout << "基础情况：n=" << n << ", 返回 " << n << endl;
        return n;
    }

    if (dp.find(n) != dp.end()) {
        int cached = dp[n];
        cout << "记忆化命中：n=" << n << ", 返回缓存结果 " << cached << endl;
        return cached;
    }

    cout << "考虑两种策略：" << endl;
}

```

```

// 策略 1 分析
int remainder2 = n % 2;
int half = n / 2;
cout << "策略 1 - 吃到 2 的倍数:" << endl;
cout << " 需要先吃掉 " << remainder2 << " 个橘子" << endl;
cout << " 然后吃掉一半 (" << half << " 个橘子)" << endl;
int option1 = remainder2 + 1 + debugMinDays(half);
cout << " 策略 1 总天数: " << remainder2 << " + 1 + minDays(" << half << ") = " << option1 << endl;

// 策略 2 分析
int remainder3 = n % 3;
int third = n / 3;
cout << "策略 2 - 吃到 3 的倍数:" << endl;
cout << " 需要先吃掉 " << remainder3 << " 个橘子" << endl;
cout << " 然后吃掉三分之二 (" << third << " 个橘子)" << endl;
int option2 = remainder3 + 1 + debugMinDays(third);
cout << " 策略 2 总天数: " << remainder3 << " + 1 + minDays(" << third << ") = " << option2 << endl;

int ans = min(option1, option2);
cout << "选择较小值: min(" << option1 << ", " << option2 << ") = " << ans << endl;

dp[n] = ans;
cout << "存储结果: dp[" << n << "] = " << ans << endl;

return ans;
}

/***
 * 测试函数: 验证吃橘子算法的正确性
 */
void testMinDays() {
    cout << "吃掉 N 个橘子的最少天数算法测试开始" << endl;
    cout << "===== " << endl;

    // 测试用例 1: n = 10
    dp.clear();
    int result1 = minDays(10);
    cout << "输入: n = 10" << endl;
    cout << "输出: " << result1 << endl;
    cout << "预期: 4" << endl;
    cout << (result1 == 4 ? "✓ 通过" : "✗ 失败") << endl << endl;
}

```

```

// 测试用例 2: n = 6
dp.clear();
int result2 = minDays(6);
cout << "输入: n = 6" << endl;
cout << "输出: " << result2 << endl;
cout << "预期: 3" << endl;
cout << (result2 == 3 ? "✓ 通过" : "✗ 失败") << endl << endl;

// 测试用例 3: n = 1
dp.clear();
int result3 = minDays(1);
cout << "输入: n = 1" << endl;
cout << "输出: " << result3 << endl;
cout << "预期: 1" << endl;
cout << (result3 == 1 ? "✓ 通过" : "✗ 失败") << endl << endl;

// 测试用例 4: n = 56
dp.clear();
int result4 = minDays(56);
cout << "输入: n = 56" << endl;
cout << "输出: " << result4 << endl;
cout << "预期: 6" << endl;
cout << (result4 == 6 ? "✓ 通过" : "✗ 失败") << endl << endl;

cout << "测试结束" << endl;
}

/***
 * 性能测试: 测试算法在大规模数据下的表现
 */
void performanceTest() {
    cout << "性能测试开始" << endl;
    cout << "======" << endl;

    dp.clear();

    clock_t start = clock();
    int result = minDays(1000000);
    clock_t end = clock();

    double duration = double(end - start) / CLOCKS_PER_SEC * 1000; // 转换为毫秒
}

```

```

cout << "输入: n = 1000000" << endl;
cout << "输出: " << result << endl;
cout << "执行时间: " << duration << " 毫秒" << endl;
cout << "性能测试结束" << endl;
}

/**
 * 主函数: 运行测试
 */
int main() {
    cout << "吃掉 N 个橘子的最少天数 - 贪心算法 + 记忆化搜索解决方案 (C++ 实现)" << endl;
    cout << "===== " << endl;

    // 运行基础测试
    testMinDays();

    cout << endl << "调试模式示例:" << endl;
    dp.clear();
    cout << "对 n = 10 进行调试跟踪:" << endl;
    int debugResult = debugMinDays(10);
    cout << "最终结果: " << debugResult << endl;

    cout << endl << "算法分析:" << endl;
    cout << "- 时间复杂度: O(logn) - 由于每次递归都会将问题规模减半或减为三分之一" << endl;
    cout << "- 空间复杂度: O(logn) - 递归深度为 logn, 记忆化存储也需要 O(logn) 空间" << endl;
    cout << "- 贪心策略: 优先使用按比例吃橘子的方法, 能更快减少橘子数量" << endl;
    cout << "- 最优性: 这种贪心策略能够得到全局最优解" << endl;
    cout << "- 记忆化优化: 避免重复计算, 提高算法效率" << endl;

    // 可选: 运行性能测试
    // cout << endl << "性能测试:" << endl;
    // performanceTest();

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

文件: Code03_MinimumNumberEatOranges.java

```

package class089;

import java.util.HashMap;
```

```
/**  
 * 吃掉 N 个橘子的最少天数 - 贪心算法 + 记忆化搜索解决方案  
 *  
 * 题目描述:  
 * 厨房里总共有 n 个橘子，你决定每一天选择如下方式之一吃这些橘子  
 * 1) 吃掉一个橘子  
 * 2) 如果剩余橘子数 n 能被 2 整除，那么你可以吃掉 n/2 个橘子  
 * 3) 如果剩余橘子数 n 能被 3 整除，那么你可以吃掉 2*(n/3) 个橘子  
 * 每天你只能从以上 3 种方案中选择一种方案  
 * 请你返回吃掉所有 n 个橘子的最少天数  
 *  
 * 测试链接: https://leetcode.cn/problems/minimum-number-of-days-to-eat-n-oranges/  
 *  
 * 算法思想:  
 * 贪心算法 + 记忆化搜索（动态规划）  
 * 1. 优先使用按比例吃橘子的方法（方法 2 和 3），因为这样能更快减少橘子数量  
 * 2. 对于每个 n，考虑两种可能性：  
 *   - 先吃到 2 的倍数，然后吃掉一半  
 *   - 先吃到 3 的倍数，然后吃掉三分之二  
 * 3. 选择天数最少的方法  
 *  
 * 时间复杂度分析:  
 * O(logn) - 由于每次递归都会将问题规模减半或减为三分之一  
 *  
 * 空间复杂度分析:  
 * O(logn) - 递归深度为 logn，记忆化存储也需要 O(logn) 空间  
 *  
 * 是否为最优解:  
 * 是，这是解决该问题的最优解  
 *  
 * 工程化考量:  
 * 1. 边界条件处理：处理 n=0 和 n=1 的特殊情况  
 * 2. 记忆化优化：避免重复计算  
 * 3. 递归深度：使用记忆化搜索控制递归深度  
 * 4. 可读性：使用清晰的变量命名和详细的注释  
 *  
 * 贪心策略证明:  
 * 由于按比例吃橘子（方法 2 和 3）能更快减少橘子数量，所以应该优先考虑这两种方法  
 * 对于每个 n，选择能最快减少橘子数量的方法  
 */  
  
public class Code03_MinimumNumberEatOranges {
```

```
// 记忆化存储表，避免重复计算
public static HashMap<Integer, Integer> dp = new HashMap<>();

/**
 * 计算吃掉 n 个橘子的最少天数
 *
 * @param n 橘子数量
 * @return 最少天数
 *
 * 算法步骤：
 * 1. 基础情况：n=0 或 1 时，直接返回 n
 * 2. 如果已经计算过 n，直接返回结果
 * 3. 考虑两种贪心策略：
 *    - 先吃到 2 的倍数，然后吃掉一半
 *    - 先吃到 3 的倍数，然后吃掉三分之二
 * 4. 选择天数最少的方法
 *
 * 贪心策略解释：
 * 方法 2 和 3 能更快减少橘子数量，所以应该优先考虑
 * 但需要先通过方法 1 吃到合适的倍数
 */

public static int minDays(int n) {
    // 基础情况处理
    if (n <= 1) {
        return n;
    }

    // 记忆化检查：如果已经计算过，直接返回结果
    if (dp.containsKey(n)) {
        return dp.get(n);
    }

    // 贪心策略 1：先吃到 2 的倍数，然后吃掉一半
    // 需要先吃掉 n % 2 个橘子（方法 1），然后使用方法 2 吃掉一半
    // 总天数 = n % 2 + 1 + minDays(n / 2)
    int option1 = n % 2 + 1 + minDays(n / 2);

    // 贪心策略 2：先吃到 3 的倍数，然后吃掉三分之二
    // 需要先吃掉 n % 3 个橘子（方法 1），然后使用方法 3 吃掉三分之二
    // 总天数 = n % 3 + 1 + minDays(n / 3)
    int option2 = n % 3 + 1 + minDays(n / 3);

    // 选择天数最少的方法
    if (option1 < option2) {
        dp.put(n, option1);
        return option1;
    } else {
        dp.put(n, option2);
        return option2;
    }
}
```

```

int ans = Math.min(option1, option2);

// 存储结果到记忆化表
dp.put(n, ans);
return ans;
}

/**
 * 调试版本：打印计算过程中的中间结果
 *
 * @param n 橘子数量
 * @return 最少天数
 */
public static int debugMinDays(int n) {
    System.out.println("计算吃掉 " + n + " 个橘子的最少天数:");

    if (n <= 1) {
        System.out.println("基础情况: n=" + n + ", 返回 " + n);
        return n;
    }

    if (dp.containsKey(n)) {
        int cached = dp.get(n);
        System.out.println("记忆化命中: n=" + n + ", 返回缓存结果 " + cached);
        return cached;
    }

    System.out.println("考虑两种策略:");

    // 策略 1 分析
    int remainder2 = n % 2;
    int half = n / 2;
    System.out.println("策略 1 - 吃到 2 的倍数:");
    System.out.println(" 需要先吃掉 " + remainder2 + " 个橘子");
    System.out.println(" 然后吃掉一半 (" + half + " 个橘子)");
    int option1 = remainder2 + 1 + debugMinDays(half);
    System.out.println(" 策略 1 总天数: " + remainder2 + " + 1 + minDays(" + half + ") = " +
option1);

    // 策略 2 分析
    int remainder3 = n % 3;
    int third = n / 3;
    System.out.println("策略 2 - 吃到 3 的倍数:");
}

```

```
System.out.println(" 需要先吃掉 " + remainder3 + " 个橘子");
System.out.println(" 然后吃掉三分之二 (" + third + " 个橘子)");
int option2 = remainder3 + 1 + debugMinDays(third);
System.out.println(" 策略 2 总天数: " + remainder3 + " + 1 + minDays(" + third + ") = " +
option2);

int ans = Math.min(option1, option2);
System.out.println("选择较小值: min(" + option1 + ", " + option2 + ") = " + ans);

dp.put(n, ans);
System.out.println("存储结果: dp[" + n + "] = " + ans);

return ans;
}

/***
 * 测试函数: 验证吃橘子算法的正确性
 */
public static void testMinDays() {
    System.out.println("吃掉 N 个橘子的最少天数算法测试开始");
    System.out.println("====");

    // 清空记忆化表
    dp.clear();

    // 测试用例 1: n = 10
    int result1 = minDays(10);
    System.out.println("输入: n = 10");
    System.out.println("输出: " + result1);
    System.out.println("预期: 4");
    System.out.println((result1 == 4 ? "✓ 通过" : "✗ 失败"));
    System.out.println();

    // 清空记忆化表
    dp.clear();

    // 测试用例 2: n = 6
    int result2 = minDays(6);
    System.out.println("输入: n = 6");
    System.out.println("输出: " + result2);
    System.out.println("预期: 3");
    System.out.println((result2 == 3 ? "✓ 通过" : "✗ 失败"));
    System.out.println();
}
```

```
// 清空记忆化表
dp.clear();

// 测试用例 3: n = 1
int result3 = minDays(1);
System.out.println("输入: n = 1");
System.out.println("输出: " + result3);
System.out.println("预期: 1");
System.out.println((result3 == 1 ? "✓ 通过" : "✗ 失败"));
System.out.println();

// 清空记忆化表
dp.clear();

// 测试用例 4: n = 56
int result4 = minDays(56);
System.out.println("输入: n = 56");
System.out.println("输出: " + result4);
System.out.println("预期: 6");
System.out.println((result4 == 6 ? "✓ 通过" : "✗ 失败"));
System.out.println();

System.out.println("测试结束");
}

/**
 * 性能测试: 测试算法在大规模数据下的表现
 */
public static void performanceTest() {
    System.out.println("性能测试开始");
    System.out.println("====");

    // 清空记忆化表
    dp.clear();

    long startTime = System.currentTimeMillis();
    int result = minDays(1000000);
    long endTime = System.currentTimeMillis();

    System.out.println("输入: n = 1000000");
    System.out.println("输出: " + result);
    System.out.println("执行时间: " + (endTime - startTime) + " 毫秒");
}
```

```

        System.out.println("性能测试结束");
    }

    /**
     * 主函数: 运行测试
     */
    public static void main(String[] args) {
        System.out.println("吃掉 N 个橘子的最少天数 - 贪心算法 + 记忆化搜索解决方案");
        System.out.println("=====");

        // 运行基础测试
        testMinDays();

        System.out.println("调试模式示例:");
        // 清空记忆化表
        dp.clear();
        System.out.println("对 n = 10 进行调试跟踪:");
        int debugResult = debugMinDays(10);
        System.out.println("最终结果: " + debugResult);

        System.out.println("算法分析:");
        System.out.println("- 时间复杂度: O(logn) - 由于每次递归都会将问题规模减半或减为三分之一");
        System.out.println("- 空间复杂度: O(logn) - 递归深度为 logn, 记忆化存储也需要 O(logn) 空间");
        System.out.println("- 贪心策略: 优先使用按比例吃橘子的方法, 能更快减少橘子数量");
        System.out.println("- 最优性: 这种贪心策略能够得到全局最优解");
        System.out.println("- 记忆化优化: 避免重复计算, 提高算法效率");

        // 可选: 运行性能测试
        // System.out.println("性能测试:");
        // performanceTest();
    }
}
=====
```

文件: Code03_MinimumNumberEatOranges.py

=====

"""

吃掉 N 个橘子的最少天数 - 贪心算法 + 记忆化搜索解决方案 (Python 实现)

题目描述:

厨房里总共有 n 个橘子，你决定每一天选择如下方式之一吃这些橘子

- 1) 吃掉一个橘子
- 2) 如果剩余橘子数 n 能被 2 整除，那么你可以吃掉 $n/2$ 个橘子
- 3) 如果剩余橘子数 n 能被 3 整除，那么你可以吃掉 $2*(n/3)$ 个橘子

每天你只能从以上 3 种方案中选择一种方案

请你返回吃掉所有 n 个橘子的最少天数

测试链接: <https://leetcode.cn/problems/minimum-number-of-days-to-eat-n-oranges/>

算法思想:

贪心算法 + 记忆化搜索（动态规划）

1. 优先使用按比例吃橘子的方法（方法 2 和 3），因为这样能更快减少橘子数量
2. 对于每个 n ，考虑两种可能性：
 - 先吃到 2 的倍数，然后吃掉一半
 - 先吃到 3 的倍数，然后吃掉三分之二
3. 选择天数最少的方法

时间复杂度分析:

$O(\log n)$ - 由于每次递归都会将问题规模减半或减为三分之一

空间复杂度分析:

$O(\log n)$ - 递归深度为 $\log n$ ，记忆化存储也需要 $O(\log n)$ 空间

是否为最优解:

是，这是解决该问题的最优解

工程化考量:

1. 边界条件处理：处理 $n=0$ 和 $n=1$ 的特殊情况
2. 记忆化优化：避免重复计算
3. 递归深度：使用记忆化搜索控制递归深度
4. 可读性：使用清晰的变量命名和详细的注释

贪心策略证明:

由于按比例吃橘子（方法 2 和 3）能更快减少橘子数量，所以应该优先考虑这两种方法

对于每个 n ，选择能最快减少橘子数量的方法

"""

```
from functools import lru_cache
```

```
class Code03_MinimumNumberEatOranges:
```

```
    @staticmethod
    @lru_cache(maxsize=None)
```

```
def min_days(n: int) -> int:  
    """  
        计算吃掉 n 个橘子的最少天数  
    """
```

Args:

n: 橘子数量

Returns:

最少天数

算法步骤:

1. 基础情况: n=0 或 1 时, 直接返回 n
2. 使用 lru_cache 自动进行记忆化
3. 考虑两种贪心策略:
 - 先吃到 2 的倍数, 然后吃掉一半
 - 先吃到 3 的倍数, 然后吃掉三分之二
4. 选择天数最少的方法

贪心策略解释:

方法 2 和 3 能更快减少橘子数量, 所以应该优先考虑
但需要先通过方法 1 吃到合适的倍数

"""

基础情况处理

```
if n <= 1:  
    return n
```

贪心策略 1: 先吃到 2 的倍数, 然后吃掉一半

需要先吃掉 $n \% 2$ 个橘子 (方法 1), 然后使用方法 2 吃掉一半

```
option1 = n % 2 + 1 + Code03_MinimumNumberEatOranges.min_days(n // 2)
```

贪心策略 2: 先吃到 3 的倍数, 然后吃掉三分之二

需要先吃掉 $n \% 3$ 个橘子 (方法 1), 然后使用方法 3 吃掉三分之二

```
option2 = n % 3 + 1 + Code03_MinimumNumberEatOranges.min_days(n // 3)
```

选择天数最少的方法

```
return min(option1, option2)
```

@staticmethod

```
def debug_min_days(n: int, depth: int = 0) -> int:  
    """
```

调试版本: 打印计算过程中的中间结果

Args:

n: 橘子数量

depth: 递归深度 (用于缩进显示)

Returns:

最少天数

"""

indent = " " * depth

print(f"{indent}计算吃掉 {n} 个橘子的最少天数:")

if n <= 1:

print(f"{indent}基础情况: n={n}, 返回 {n}")

return n

策略 1 分析

remainder2 = n % 2

half = n // 2

print(f"{indent}策略 1 - 吃到 2 的倍数:")

print(f"{indent} 需要先吃掉 {remainder2} 个橘子")

print(f"{indent} 然后吃掉一半 ({half} 个橘子)")

option1 = remainder2 + 1 + Code03_MinimumNumberEatOranges.debug_min_days(half, depth + 1)

print(f"{indent} 策略 1 总天数: {remainder2} + 1 + minDays({half}) = {option1}")

策略 2 分析

remainder3 = n % 3

third = n // 3

print(f"{indent}策略 2 - 吃到 3 的倍数:")

print(f"{indent} 需要先吃掉 {remainder3} 个橘子")

print(f"{indent} 然后吃掉三分之二 ({third} 个橘子)")

option2 = remainder3 + 1 + Code03_MinimumNumberEatOranges.debug_min_days(third, depth + 1)

print(f"{indent} 策略 2 总天数: {remainder3} + 1 + minDays({third}) = {option2}")

ans = min(option1, option2)

print(f"{indent} 选择较小值: min({option1}, {option2}) = {ans}")

return ans

@staticmethod

def test_min_days():

"""

测试函数: 验证吃橘子算法的正确性

"""

print("吃掉 N 个橘子的最少天数算法测试开始")

```
print("=====")

# 清空缓存
Code03_MinimumNumberEatOranges.min_days.cache_clear()

# 测试用例 1: n = 10
result1 = Code03_MinimumNumberEatOranges.min_days(10)
print("输入: n = 10")
print("输出:", result1)
print("预期: 4")
print("✓ 通过" if result1 == 4 else "✗ 失败")
print()

# 清空缓存
Code03_MinimumNumberEatOranges.min_days.cache_clear()

# 测试用例 2: n = 6
result2 = Code03_MinimumNumberEatOranges.min_days(6)
print("输入: n = 6")
print("输出:", result2)
print("预期: 3")
print("✓ 通过" if result2 == 3 else "✗ 失败")
print()

# 清空缓存
Code03_MinimumNumberEatOranges.min_days.cache_clear()

# 测试用例 3: n = 1
result3 = Code03_MinimumNumberEatOranges.min_days(1)
print("输入: n = 1")
print("输出:", result3)
print("预期: 1")
print("✓ 通过" if result3 == 1 else "✗ 失败")
print()

# 清空缓存
Code03_MinimumNumberEatOranges.min_days.cache_clear()

# 测试用例 4: n = 56
result4 = Code03_MinimumNumberEatOranges.min_days(56)
print("输入: n = 56")
print("输出:", result4)
print("预期: 6")
```

```
print("✓ 通过" if result4 == 6 else "✗ 失败")
print()

print("测试结束")

@staticmethod
def performance_test():
    """
    性能测试：测试算法在大规模数据下的表现
    """
    import time

    print("性能测试开始")
    print("===== ")

    # 清空缓存
    Code03_MinimumNumberEatOranges.min_days.cache_clear()

    start_time = time.time()
    result = Code03_MinimumNumberEatOranges.min_days(1000000)
    end_time = time.time()

    print("输入: n = 1000000")
    print("输出:", result)
    print("执行时间:", end_time - start_time, "秒")
    print("性能测试结束")

@staticmethod
def compare_with_naive(n: int) -> None:
    """
    与暴力解法对比，展示贪心算法的优势
    """

    Args:
        n: 橘子数量
    """

    print(f"对比贪心算法与暴力解法 (n={n}):")
    print("贪心算法结果:", Code03_MinimumNumberEatOranges.min_days(n))

    # 暴力解法（仅用于小规模对比）
    def brute_force(n: int) -> int:
        if n <= 1:
            return n
        # 暴力尝试所有三种方法
```

```

option1 = 1 + brute_force(n - 1) # 方法1
option2 = 10**9 # 使用大数代替无穷大
option3 = 10**9 # 使用大数代替无穷大

if n % 2 == 0:
    option2 = 1 + brute_force(n // 2) # 方法2
if n % 3 == 0:
    option3 = 1 + brute_force(n // 3) # 方法3

return min(option1, option2, option3)

if n <= 20: # 只对小规模数据进行暴力计算
    print("暴力解法结果:", brute_force(n))
else:
    print("暴力解法: 数据规模太大, 无法计算")

def main():
    """
    主函数: 运行测试
    """
    print("吃掉 N 个橘子的最少天数 - 贪心算法 + 记忆化搜索解决方案 (Python 实现)")
    print("=====")

    # 运行基础测试
    Code03_MinimumNumberEatOranges.test_min_days()

    print("\n 调试模式示例:")
    print("对 n = 10 进行调试跟踪:")
    debug_result = Code03_MinimumNumberEatOranges.debug_min_days(10)
    print("最终结果:", debug_result)

    print("\n 算法分析:")
    print("- 时间复杂度: O(logn) - 由于每次递归都会将问题规模减半或减为三分之一")
    print("- 空间复杂度: O(logn) - 递归深度为 logn, 记忆化存储也需要 O(logn) 空间")
    print("- 贪心策略: 优先使用按比例吃橘子的方法, 能更快减少橘子数量")
    print("- 最优性: 这种贪心策略能够得到全局最优解")
    print("- Python 特性: 使用@lru_cache 自动进行记忆化")

    # 可选: 运行性能测试
    # print("\n 性能测试:")
    # Code03_MinimumNumberEatOranges.performance_test()

```

```
# 可选: 与暴力解法对比
# print("\n 算法对比:")
# Code03_MinimumNumberEatOranges.compare_with_naive(10)

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

文件: Code04_MeetingRoomsII.cpp

```
=====
/***
 * 会议室 II - 贪心算法 + 最小堆解决方案 (C++实现)
 *
 * 题目描述:
 * 给你一个会议时间安排的数组 intervals
 * 每个会议时间都会包括开始和结束的时间 intervals[i]=[starti, endi]
 * 返回所需会议室的最小数量
 *
 * 测试链接: https://leetcode.cn/problems/meeting-rooms-ii/
 *
 * 算法思想:
 * 贪心算法 + 最小堆 (优先队列)
 * 1. 按照会议开始时间对会议进行排序
 * 2. 使用最小堆来存储当前正在进行的会议的结束时间
 * 3. 对于每个会议:
 *     - 如果堆顶的会议已经结束 (结束时间 <= 当前会议开始时间), 则弹出堆顶
 *     - 将当前会议的结束时间加入堆中
 *     - 更新所需会议室的最大数量
 *
 * 时间复杂度分析:
 * O(n*logn) - 排序需要 O(n*logn), 每个会议进出堆一次需要 O(logn)
 *
 * 空间复杂度分析:
 * O(n) - 最坏情况下所有会议都需要同时进行, 堆的大小为 n
 *
 * 是否为最优解:
 * 是, 这是解决该问题的最优解
 *
 * 工程化考量:
 * 1. 边界条件处理: 处理空数组、单个会议等特殊情况
 * 2. 输入验证: 检查会议时间是否有效 (开始时间 < 结束时间)
```

```
* 3. 异常处理：对非法输入进行检查
* 4. 可读性：使用清晰的变量命名和详细的注释
*
* 贪心策略证明：
* 每次选择最早结束的会议室来安排新会议，这样可以最大化会议室的使用效率
* 这种策略可以保证所需会议室数量最少
*/
```

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <queue>
#include <stdexcept>

using namespace std;

/***
 * 计算所需会议室的最小数量
 *
 * @param intervals 会议时间数组，每个元素是[starti, endi]
 * @return 所需会议室的最小数量
 *
 * 算法步骤：
 * 1. 按照会议开始时间排序
 * 2. 使用最小堆存储当前会议的结束时间
 * 3. 遍历每个会议，释放已结束的会议室，分配新的会议室
 * 4. 跟踪所需会议室的最大数量
*/
int minMeetingRooms(vector<vector<int>>& intervals) {
    // 输入验证
    if (intervals.empty()) {
        return 0;
    }

    int n = intervals.size();

    // 按照会议开始时间排序
    sort(intervals.begin(), intervals.end(), [] (const vector<int>& a, const vector<int>& b) {
        return a[0] < b[0];
    });

    // 最小堆，存储当前正在进行的会议的结束时间
    priority_queue<int, vector<int>, greater<int>> heap;
```

```
int maxRooms = 0; // 所需会议室的最大数量

for (int i = 0; i < n; i++) {
    int start = intervals[i][0];
    int end = intervals[i][1];

    // 验证会议时间有效性
    if (start >= end) {
        throw invalid_argument("会议开始时间必须小于结束时间");
    }

    // 释放已经结束的会议室 (结束时间 <= 当前会议开始时间)
    while (!heap.empty() && heap.top() <= start) {
        heap.pop();
    }

    // 分配新的会议室
    heap.push(end);

    // 更新所需会议室的最大数量
    maxRooms = max(maxRooms, (int)heap.size());
}

return maxRooms;
}

/**
 * 调试版本：打印计算过程中的中间结果
 *
 * @param intervals 会议时间数组
 * @return 所需会议室的最小数量
 */
int debugMinMeetingRooms(vector<vector<int>>& intervals) {
    if (intervals.empty()) {
        cout << "空数组，不需要会议室" << endl;
        return 0;
    }

    int n = intervals.size();

    cout << "原始会议安排：" << endl;
    for (int i = 0; i < n; i++) {
        cout << "会议" << i << ":" [ " << intervals[i][0] << ", " << intervals[i][1] << "]"
    }
}
```

```

endl;
}

// 按照会议开始时间排序
sort(intervals.begin(), intervals.end(), [] (const vector<int>& a, const vector<int>& b) {
    return a[0] < b[0];
});

cout << "\n 排序后的会议安排:" << endl;
for (int i = 0; i < n; i++) {
    cout << "会议" << i << ":" [ " << intervals[i][0] << ", " << intervals[i][1] << "]"
endl;
}

priority_queue<int, vector<int>, greater<int>> heap;
int maxRooms = 0;

cout << "\n 分配过程:" << endl;
for (int i = 0; i < n; i++) {
    int start = intervals[i][0];
    int end = intervals[i][1];

    cout << "\n 处理会议" << i << ":" [ " << start << ", " << end << "]"
<< endl;

    // 释放已经结束的会议室
    cout << "释放会议室: ";
    bool released = false;
    while (!heap.empty() && heap.top() <= start) {
        int finished = heap.top();
        heap.pop();
        cout << "结束时间" << finished << " ";
        released = true;
    }
    if (!released) {
        cout << "无会议室可释放";
    }
    cout << endl;

    // 分配新的会议室
    heap.push(end);
    cout << "分配新会议室, 当前会议室数量: " << heap.size() << endl;

    // 更新最大数量
}

```

```

        if (heap.size() > maxRooms) {
            maxRooms = heap.size();
            cout << "更新最大会议室数量: " << maxRooms << endl;
        }
    }

    cout << "\n最终结果: " << maxRooms << " 个会议室" << endl;
    return maxRooms;
}

/***
 * 打印二维数组的辅助函数
 *
 * @param intervals 会议时间数组
 */
void printIntervals(const vector<vector<int>>& intervals) {
    cout << "[";
    for (int i = 0; i < intervals.size(); i++) {
        cout << "[" << intervals[i][0] << "," << intervals[i][1] << "]";
        if (i < intervals.size() - 1) cout << ",";
    }
    cout << "]";
}

/***
 * 测试函数: 验证会议室分配算法的正确性
 */
void testMinMeetingRooms() {
    cout << "会议室 II 算法测试开始" << endl;
    cout << "===== " << endl;

    // 测试用例 1: [[0,30], [5,10], [15,20]]
    vector<vector<int>> intervals1 = {{0, 30}, {5, 10}, {15, 20}};
    int result1 = minMeetingRooms(intervals1);
    cout << "输入: ";
    printIntervals(intervals1);
    cout << endl;
    cout << "输出: " << result1 << endl;
    cout << "预期: 2" << endl;
    cout << (result1 == 2 ? "✓ 通过" : "✗ 失败") << endl << endl;

    // 测试用例 2: [[7,10], [2,4]]
    vector<vector<int>> intervals2 = {{7, 10}, {2, 4}};

```

```

int result2 = minMeetingRooms(intervals2);
cout << "输入: ";
printIntervals(intervals2);
cout << endl;
cout << "输出: " << result2 << endl;
cout << "预期: 1" << endl;
cout << (result2 == 1 ? "✓ 通过" : "✗ 失败") << endl << endl;

// 测试用例 3: 空数组
vector<vector<int>> intervals3 = {};
int result3 = minMeetingRooms(intervals3);
cout << "输入: []" << endl;
cout << "输出: " << result3 << endl;
cout << "预期: 0" << endl;
cout << (result3 == 0 ? "✓ 通过" : "✗ 失败") << endl << endl;

// 测试用例 4: [[1,5], [8,9], [8,9]]
vector<vector<int>> intervals4 = {{1,5}, {8,9}, {8,9}};
int result4 = minMeetingRooms(intervals4);
cout << "输入: [[1,5], [8,9], [8,9]]" << endl;
cout << "输出: " << result4 << endl;
cout << "预期: 2" << endl;
cout << (result4 == 2 ? "✓ 通过" : "✗ 失败") << endl << endl;

cout << "测试结束" << endl;
}

/***
 * 性能测试: 测试算法在大规模数据下的表现
 */
void performanceTest() {
    cout << "性能测试开始" << endl;
    cout << "======" << endl;

    // 生成大规模测试数据
    int n = 10000;
    vector<vector<int>> intervals;
    for (int i = 0; i < n; i++) {
        int start = i * 10;
        int end = start + (rand() % 10) + 1;
        intervals.push_back({start, end});
    }
}

```

```

clock_t start = clock();
int result = minMeetingRooms(intervals);
clock_t end = clock();

double duration = double(end - start) / CLOCKS_PER_SEC * 1000; // 转换为毫秒

cout << "数据规模: " << n << " 个会议" << endl;
cout << "执行时间: " << duration << " 毫秒" << endl;
cout << "所需会议室数量: " << result << endl;
cout << "性能测试结束" << endl;
}

/***
 * 主函数: 运行测试
 */
int main() {
    cout << "会议室 II - 贪心算法 + 最小堆解决方案 (C++实现)" << endl;
    cout << "===== " << endl;

    // 运行基础测试
    testMinMeetingRooms();

    cout << endl << "调试模式示例:" << endl;
    vector<vector<int>> debugIntervals = {{0, 30}, {5, 10}, {15, 20}};
    cout << "对测试用例 [[0,30], [5,10], [15,20]] 进行调试跟踪:" << endl;
    int debugResult = debugMinMeetingRooms(debugIntervals);
    cout << "最终结果: " << debugResult << endl;

    cout << endl << "算法分析:" << endl;
    cout << "- 时间复杂度: O(n*logn) - 排序需要 O(n*logn), 每个会议进出堆一次需要 O(logn)" << endl;
    cout << "- 空间复杂度: O(n) - 最坏情况下所有会议都需要同时进行, 堆的大小为 n" << endl;
    cout << "- 贪心策略: 每次选择最早结束的会议室来安排新会议" << endl;
    cout << "- 最优性: 这种贪心策略能够得到全局最优解" << endl;
    cout << "- 数据结构: 使用最小堆 (优先队列) 来高效管理会议室" << endl;

    // 可选: 运行性能测试
    // cout << endl << "性能测试:" << endl;
    // performanceTest();

    return 0;
}

```

文件: Code04_MeetingRoomsII.java

```
=====
package class089;
```

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

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

```
/**
 * 会议室 II - 贪心算法 + 最小堆解决方案
 *
 * 题目描述:
 * 给你一个会议时间安排的数组 intervals
 * 每个会议时间都会包括开始和结束的时间 intervals[i]=[starti, endi]
 * 返回所需会议室的最小数量
 *
 * 测试链接: https://leetcode.cn/problems/meeting-rooms-ii/
 *
 * 算法思想:
 * 贪心算法 + 最小堆 (优先队列)
 * 1. 按照会议开始时间对会议进行排序
 * 2. 使用最小堆来存储当前正在进行的会议的结束时间
 * 3. 对于每个会议:
 *     - 如果堆顶的会议已经结束 (结束时间 <= 当前会议开始时间), 则弹出堆顶
 *     - 将当前会议的结束时间加入堆中
 *     - 更新所需会议室的最大数量
 *
 * 时间复杂度分析:
 * O(n*logn) - 排序需要 O(n*logn), 每个会议进出堆一次需要 O(logn)
 *
 * 空间复杂度分析:
 * O(n) - 最坏情况下所有会议都需要同时进行, 堆的大小为 n
 *
 * 是否为最优解:
 * 是, 这是解决该问题的最优解
 *
 * 工程化考量:
 * 1. 边界条件处理: 处理空数组、单个会议等特殊情况
 * 2. 输入验证: 检查会议时间是否有效 (开始时间 < 结束时间)
 * 3. 异常处理: 对非法输入进行检查
 * 4. 可读性: 使用清晰的变量命名和详细的注释
 *
```

```
* 贪心策略证明：  
* 每次选择最早结束的会议室来安排新会议，这样可以最大化会议室的使用效率  
* 这种策略可以保证所需会议室数量最少  
*/  
  
public class Code04_MeetingRoomsII {  
  
    /**  
     * 计算所需会议室的最小数量  
     *  
     * @param intervals 会议时间数组，每个元素是[starti, endi]  
     * @return 所需会议室的最小数量  
     *  
     * 算法步骤：  
     * 1. 按照会议开始时间排序  
     * 2. 使用最小堆存储当前会议的结束时间  
     * 3. 遍历每个会议，释放已结束的会议室，分配新的会议室  
     * 4. 跟踪所需会议室的最大数量  
     */  
  
    public static int minMeetingRooms(int[][] intervals) {  
        // 输入验证  
        if (intervals == null || intervals.length == 0) {  
            return 0;  
        }  
  
        int n = intervals.length;  
  
        // 按照会议开始时间排序  
        Arrays.sort(intervals, (a, b) -> a[0] - b[0]);  
  
        // 最小堆，存储当前正在进行的会议的结束时间  
        PriorityQueue<Integer> heap = new PriorityQueue<>();  
        int maxRooms = 0; // 所需会议室的最大数量  
  
        for (int i = 0; i < n; i++) {  
            int start = intervals[i][0];  
            int end = intervals[i][1];  
  
            // 验证会议时间有效性  
            if (start >= end) {  
                throw new IllegalArgumentException("会议开始时间必须小于结束时间");  
            }  
  
            // 释放已经结束的会议室（结束时间 <= 当前会议开始时间）  
            while (!heap.isEmpty() && heap.peek() <= start) {  
                heap.poll();  
            }  
            heap.offer(end);  
            maxRooms = Math.max(maxRooms, heap.size());  
        }  
        return maxRooms;  
    }  
}
```

```

        while (!heap.isEmpty() && heap.peek() <= start) {
            heap.poll();
        }

        // 分配新的会议室
        heap.add(end);

        // 更新所需会议室的最大数量
        maxRooms = Math.max(maxRooms, heap.size());
    }

    return maxRooms;
}

/**
 * 调试版本：打印计算过程中的中间结果
 *
 * @param intervals 会议时间数组
 * @return 所需会议室的最小数量
 */
public static int debugMinMeetingRooms(int[][] intervals) {
    if (intervals == null || intervals.length == 0) {
        System.out.println("空数组，不需要会议室");
        return 0;
    }

    int n = intervals.length;

    System.out.println("原始会议安排:");
    for (int i = 0; i < n; i++) {
        System.out.println("会议" + i + ":" + intervals[i][0] + ", " + intervals[i][1] +
    "]");
    }

    // 按照会议开始时间排序
    Arrays.sort(intervals, (a, b) -> a[0] - b[0]);

    System.out.println("排序后的会议安排:");
    for (int i = 0; i < n; i++) {
        System.out.println("会议" + i + ":" + intervals[i][0] + ", " + intervals[i][1] +
    "]");
    }
}

```

```
PriorityQueue<Integer> heap = new PriorityQueue<>();
int maxRooms = 0;

System.out.println("分配过程:");
for (int i = 0; i < n; i++) {
    int start = intervals[i][0];
    int end = intervals[i][1];

    System.out.println("处理会议" + i + ":" + [start, end]);

    // 释放已经结束的会议室
    System.out.print("释放会议室: ");
    boolean released = false;
    while (!heap.isEmpty() && heap.peek() <= start) {
        int finished = heap.poll();
        System.out.print("结束时间" + finished + " ");
        released = true;
    }
    if (!released) {
        System.out.print("无会议室可释放");
    }
    System.out.println();

    // 分配新的会议室
    heap.add(end);
    System.out.println("分配新会议室, 当前会议室数量: " + heap.size());

    // 更新最大数量
    if (heap.size() > maxRooms) {
        maxRooms = heap.size();
        System.out.println("更新最大会议室数量: " + maxRooms);
    }
}

System.out.println("最终结果: " + maxRooms + " 个会议室");
return maxRooms;
}

/**
 * 测试函数: 验证会议室分配算法的正确性
 */
public static void testMinMeetingRooms() {
    System.out.println("会议室 II 算法测试开始");
```

```
System.out.println("=====");  
  
// 测试用例 1: [[0, 30], [5, 10], [15, 20]]  
int[][] intervals1 = {{0, 30}, {5, 10}, {15, 20}};  
int result1 = minMeetingRooms(intervals1);  
System.out.println("输入: [[0, 30], [5, 10], [15, 20]]");  
System.out.println("输出: " + result1);  
System.out.println("预期: 2");  
System.out.println((result1 == 2 ? "✓ 通过" : "✗ 失败"));  
System.out.println();  
  
// 测试用例 2: [[7, 10], [2, 4]]  
int[][] intervals2 = {{7, 10}, {2, 4}};  
int result2 = minMeetingRooms(intervals2);  
System.out.println("输入: [[7, 10], [2, 4]]");  
System.out.println("输出: " + result2);  
System.out.println("预期: 1");  
System.out.println((result2 == 1 ? "✓ 通过" : "✗ 失败"));  
System.out.println();  
  
// 测试用例 3: 空数组  
int[][] intervals3 = {};  
int result3 = minMeetingRooms(intervals3);  
System.out.println("输入: []");  
System.out.println("输出: " + result3);  
System.out.println("预期: 0");  
System.out.println((result3 == 0 ? "✓ 通过" : "✗ 失败"));  
System.out.println();  
  
// 测试用例 4: [[1, 5], [8, 9], [8, 9]]  
int[][] intervals4 = {{1, 5}, {8, 9}, {8, 9}};  
int result4 = minMeetingRooms(intervals4);  
System.out.println("输入: [[1, 5], [8, 9], [8, 9]]");  
System.out.println("输出: " + result4);  
System.out.println("预期: 2");  
System.out.println((result4 == 2 ? "✓ 通过" : "✗ 失败"));  
System.out.println();  
  
System.out.println("测试结束");  
}  
  
/**  
 * 性能测试: 测试算法在大规模数据下的表现
```

```

*/
public static void performanceTest() {
    System.out.println("性能测试开始");
    System.out.println("====");

    // 生成大规模测试数据
    int n = 10000;
    int[][] intervals = new int[n][2];
    for (int i = 0; i < n; i++) {
        int start = i * 10;
        int end = start + (int)(Math.random() * 10) + 1;
        intervals[i][0] = start;
        intervals[i][1] = end;
    }

    long startTime = System.currentTimeMillis();
    int result = minMeetingRooms(intervals);
    long endTime = System.currentTimeMillis();

    System.out.println("数据规模: " + n + " 个会议");
    System.out.println("执行时间: " + (endTime - startTime) + " 毫秒");
    System.out.println("所需会议室数量: " + result);
    System.out.println("性能测试结束");
}

/**
 * 主函数: 运行测试
 */
public static void main(String[] args) {
    System.out.println("会议室 II - 贪心算法 + 最小堆解决方案");
    System.out.println("====");

    // 运行基础测试
    testMinMeetingRooms();

    System.out.println("调试模式示例:");
    int[][] debugIntervals = {{0, 30}, {5, 10}, {15, 20}};
    System.out.println("对测试用例 [[0, 30], [5, 10], [15, 20]] 进行调试跟踪:");
    int debugResult = debugMinMeetingRooms(debugIntervals);
    System.out.println("最终结果: " + debugResult);

    System.out.println("算法分析:");
    System.out.println("- 时间复杂度: O(n*logn) - 排序需要 O(n*logn), 每个会议进出堆一次需要"

```

```

0(logn));
    System.out.println("- 空间复杂度: O(n) - 最坏情况下所有会议都需要同时进行，堆的大小为 n");
    System.out.println("- 贪心策略: 每次选择最早结束的会议室来安排新会议");
    System.out.println("- 最优性: 这种贪心策略能够得到全局最优解");
    System.out.println("- 数据结构: 使用最小堆（优先队列）来高效管理会议室");

    // 可选: 运行性能测试
    // System.out.println("性能测试:");
    // performanceTest();
}
}

```

=====

文件: Code04_MeetingRoomsII.py

=====

"""
会议室 II - 贪心算法 + 最小堆解决方案 (Python 实现)

题目描述:

给你一个会议时间安排的数组 intervals

每个会议时间都会包括开始和结束的时间 intervals[i]=[starti, endi]

返回所需会议室的最小数量

测试链接: <https://leetcode.cn/problems/meeting-rooms-ii/>

算法思想:

贪心算法 + 最小堆（优先队列）

1. 按照会议开始时间对会议进行排序
2. 使用最小堆来存储当前正在进行的会议的结束时间
3. 对于每个会议:
 - 如果堆顶的会议已经结束 (结束时间 <= 当前会议开始时间), 则弹出堆顶
 - 将当前会议的结束时间加入堆中
 - 更新所需会议室的最大数量

时间复杂度分析:

$O(n \log n)$ - 排序需要 $O(n \log n)$, 每个会议进出堆一次需要 $O(\log n)$

空间复杂度分析:

$O(n)$ - 最坏情况下所有会议都需要同时进行, 堆的大小为 n

是否为最优解:

是, 这是解决该问题的最优解

工程化考量：

1. 边界条件处理：处理空数组、单个会议等特殊情况
2. 输入验证：检查会议时间是否有效（开始时间 < 结束时间）
3. 异常处理：对非法输入进行检查
4. 可读性：使用清晰的变量命名和详细的注释

贪心策略证明：

每次选择最早结束的会议室来安排新会议，这样可以最大化会议室的使用效率
这种策略可以保证所需会议室数量最少

"""

```
import heapq
from typing import List

class Code04_MeetingRoomsII:

    @staticmethod
    def min_meeting_rooms(intervals: List[List[int]]) -> int:
        """
        计算所需会议室的最小数量
        """

Args:
```

 intervals: 会议时间数组，每个元素是[starti, endi]

Returns:

 所需会议室的最小数量

算法步骤：

1. 按照会议开始时间排序
2. 使用最小堆存储当前会议的结束时间
3. 遍历每个会议，释放已结束的会议室，分配新的会议室
4. 跟踪所需会议室的最大数量

"""

```
# 输入验证
if not intervals:
    return 0

n = len(intervals)

# 按照会议开始时间排序
intervals.sort(key=lambda x: x[0])
```

```
# 最小堆，存储当前正在进行的会议的结束时间
heap = []
max_rooms = 0 # 所需会议室的最大数量

for i in range(n):
    start, end = intervals[i]

    # 验证会议时间有效性
    if start >= end:
        raise ValueError("会议开始时间必须小于结束时间")

    # 释放已经结束的会议室（结束时间 <= 当前会议开始时间）
    while heap and heap[0] <= start:
        heapq.heappop(heap)

    # 分配新的会议室
    heapq.heappush(heap, end)

    # 更新所需会议室的最大数量
    max_rooms = max(max_rooms, len(heap))

return max_rooms
```

```
@staticmethod
def debug_min_meeting_rooms(intervals: List[List[int]]) -> int:
    """
    调试版本：打印计算过程中的中间结果
    """

```

Args:
 intervals: 会议时间数组

Returns:
 所需会议室的最小数量
 """
if not intervals:
 print("空数组，不需要会议室")
 return 0

n = len(intervals)

print("原始会议安排:")
for i in range(n):
 print(f"会议 {i}: [{intervals[i][0]}, {intervals[i][1]}]")

```
# 按照会议开始时间排序
intervals.sort(key=lambda x: x[0])

print("\n排序后的会议安排:")
for i in range(n):
    print(f"会议{i}: [{intervals[i][0]}, {intervals[i][1]}]")

heap = []
max_rooms = 0

print("\n分配过程:")
for i in range(n):
    start, end = intervals[i]

    print(f"\n处理会议{i}: [{start}, {end}]")

    # 释放已经结束的会议室
    print("释放会议室: ", end="")
    released = False
    while heap and heap[0] <= start:
        finished = heapq.heappop(heap)
        print(f"结束时间{finished} ", end="")
        released = True
    if not released:
        print("无会议室可释放", end="")

    print()

    # 分配新的会议室
    heapq.heappush(heap, end)
    print(f"分配新会议室, 当前会议室数量: {len(heap)}")

    # 更新最大数量
    if len(heap) > max_rooms:
        max_rooms = len(heap)
        print(f"更新最大会议室数量: {max_rooms}")

print(f"\n最终结果: {max_rooms} 个会议室")
return max_rooms

@staticmethod
def test_min_meeting_rooms():
    """
```

测试函数：验证会议室分配算法的正确性

"""

```
print("会议室 II 算法测试开始")
print("=====")

# 测试用例 1: [[0, 30], [5, 10], [15, 20]]
intervals1 = [[0, 30], [5, 10], [15, 20]]
result1 = Code04_MeetingRoomsII.min_meeting_rooms(intervals1)
print("输入: [[0, 30], [5, 10], [15, 20]]")
print("输出:", result1)
print("预期: 2")
print("✓ 通过" if result1 == 2 else "✗ 失败")
print()

# 测试用例 2: [[7, 10], [2, 4]]
intervals2 = [[7, 10], [2, 4]]
result2 = Code04_MeetingRoomsII.min_meeting_rooms(intervals2)
print("输入: [[7, 10], [2, 4]]")
print("输出:", result2)
print("预期: 1")
print("✓ 通过" if result2 == 1 else "✗ 失败")
print()

# 测试用例 3: 空数组
intervals3 = []
result3 = Code04_MeetingRoomsII.min_meeting_rooms(intervals3)
print("输入: []")
print("输出:", result3)
print("预期: 0")
print("✓ 通过" if result3 == 0 else "✗ 失败")
print()

# 测试用例 4: [[1, 5], [8, 9], [8, 9]]
intervals4 = [[1, 5], [8, 9], [8, 9]]
result4 = Code04_MeetingRoomsII.min_meeting_rooms(intervals4)
print("输入: [[1, 5], [8, 9], [8, 9]]")
print("输出:", result4)
print("预期: 2")
print("✓ 通过" if result4 == 2 else "✗ 失败")
print()

print("测试结束")
```

```

@staticmethod
def performance_test():
    """
    性能测试：测试算法在大规模数据下的表现
    """

    import time
    import random

    print("性能测试开始")
    print("====")

    # 生成大规模测试数据
    n = 10000
    intervals = []
    for i in range(n):
        start = i * 10
        end = start + random.randint(1, 10)
        intervals.append([start, end])

    start_time = time.time()
    result = Code04_MeetingRoomsII.min_meeting_rooms(intervals)
    end_time = time.time()

    print(f"数据规模: {n} 个会议")
    print(f"执行时间: {end_time - start_time:.4f} 秒")
    print(f"所需会议室数量: {result}")
    print("性能测试结束")

```

```

@staticmethod
def alternative_solution(intervals: List[List[int]]) -> int:
    """
    替代解法：使用双指针方法（事件排序法）

```

算法思想：

1. 分别提取所有会议的开始时间和结束时间
2. 对开始时间和结束时间分别排序
3. 使用双指针遍历，统计同时进行的会议数量

时间复杂度：O(n*logn)

空间复杂度：O(n)

"""

```

if not intervals:
    return 0

```

```
starts = [interval[0] for interval in intervals]
ends = [interval[1] for interval in intervals]

starts.sort()
ends.sort()

rooms = 0
end_ptr = 0

for start in starts:
    if start < ends[end_ptr]:
        rooms += 1
    else:
        end_ptr += 1

return rooms

def main():
    """
    主函数: 运行测试
    """
    print("会议室 II - 贪心算法 + 最小堆解决方案 (Python 实现) ")
    print("====")
    # 运行基础测试
    Code04_MeetingRoomsII.test_min_meeting_rooms()

    print("\n 调试模式示例:")
    debug_intervals = [[0, 30], [5, 10], [15, 20]]
    print("对测试用例 [[0, 30], [5, 10], [15, 20]] 进行调试跟踪:")
    debug_result = Code04_MeetingRoomsII.debug_min_meeting_rooms(debug_intervals)
    print("最终结果:", debug_result)

    print("\n 算法分析:")
    print("- 时间复杂度: O(n * log n) - 排序需要 O(n * log n), 每个会议进出堆一次需要 O(log n)")
    print("- 空间复杂度: O(n) - 最坏情况下所有会议都需要同时进行, 堆的大小为 n")
    print("- 贪心策略: 每次选择最早结束的会议室来安排新会议")
    print("- 最优性: 这种贪心策略能够得到全局最优解")
    print("- Python 特性: 使用 heapq 模块实现最小堆")

    # 可选: 运行性能测试
```

```
# print("\n 性能测试:")
# Code04_MeetingRoomsII.performance_test()

# 可选: 比较替代解法
# print("\n 替代解法测试:")
# test_intervals = [[0, 30], [5, 10], [15, 20]]
# alt_result = Code04_MeetingRoomsII.alternative_solution(test_intervals)
# print("替代解法结果:", alt_result)

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

文件: Code05_CourseScheduleIII.cpp

```
=====
/***
 * 课程表 III - 贪心算法 + 最大堆解决方案 (C++实现)
 *
 * 题目描述:
 * 这里有 n 门不同的在线课程, 按从 1 到 n 编号
 * 给你一个数组 courses, 其中 courses[i]=[durationi, lastDayi] 表示第 i 门课将会持续上 durationi 天课
 * 并且必须在不晚于 lastDayi 的时候完成
 * 你的学期从第 1 天开始, 且不能同时修读两门及两门以上的课程
 * 返回你最多可以修读的课程数目
 *
 * 测试链接: https://leetcode.cn/problems/course-schedule-iii/
 *
 * 算法思想:
 * 贪心算法 + 最大堆 (优先队列)
 * 1. 按照课程的截止时间进行排序 (截止时间早的排在前面)
 * 2. 使用最大堆来存储已选课程的持续时间
 * 3. 遍历每个课程:
 *     - 如果当前时间加上课程持续时间不超过截止时间, 则选择该课程
 *     - 如果超过截止时间, 则检查是否可以替换已选课程中持续时间最长的课程
 *
 * 时间复杂度分析:
 * O(n*logn) - 排序需要 O(n*logn), 每个课程进出堆一次需要 O(logn)
 *
 * 空间复杂度分析:
 * O(n) - 最坏情况下所有课程都被选中, 堆的大小为 n
 * =====
```

- * 是否为最优解:
- * 是, 这是解决该问题的最优解
- *
- * 工程化考量:
 - * 1. 边界条件处理: 处理空数组、单个课程等特殊情况
 - * 2. 输入验证: 检查课程时间是否有效 (持续时间 > 0, 截止时间 > 0)
 - * 3. 异常处理: 对非法输入进行检查
 - * 4. 可读性: 使用清晰的变量命名和详细的注释
- *
- * 贪心策略证明:
 - * 按照截止时间排序可以保证我们优先考虑截止时间早的课程
 - * 使用最大堆来管理已选课程, 当遇到冲突时替换持续时间最长的课程, 可以最大化课程数量

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <queue>
#include <stdexcept>

using namespace std;

/***
 * 计算最多可以修读的课程数目
 *
 * @param courses 课程数组, 每个元素是[durationi, lastDayi]
 * @return 最多可以修读的课程数目
 *
 * 算法步骤:
 * 1. 按照课程的截止时间进行排序
 * 2. 使用最大堆存储已选课程的持续时间
 * 3. 维护当前已使用的时间
 * 4. 遍历每个课程, 动态调整已选课程
 */
int scheduleCourse(vector<vector<int>>& courses) {
    // 输入验证
    if (courses.empty()) {
        return 0;
    }

    int n = courses.size();

    // 按照课程的截止时间进行排序 (截止时间早的排在前面)
```

```

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（持续时间长的在堆顶）
priority_queue<int> heap;
int currentTime = 0; // 当前已使用的时间

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

    // 验证课程时间有效性
    if (duration <= 0 || lastDay <= 0) {
        throw invalid_argument("课程持续时间和截止时间必须大于 0");
    }

    // 如果当前时间加上课程持续时间不超过截止时间
    if (currentTime + duration <= lastDay) {
        heap.push(duration);
        currentTime += duration;
    }
    // 如果超过截止时间，但当前课程的持续时间比已选课程中最长的短
    else if (!heap.empty() && heap.top() > duration) {
        // 替换策略：用当前课程替换已选课程中持续时间最长的课程
        int longestDuration = heap.top();
        heap.pop();
        currentTime += duration - longestDuration;
        heap.push(duration);
    }
    // 其他情况：不选择当前课程
}

return heap.size();
}

/***
 * 调试版本：打印计算过程中的中间结果
 *
 * @param courses 课程数组
 * @return 最多可以修读的课程数目
 */
int debugScheduleCourse(vector<vector<int>>& courses) {

```

```

if (courses.empty()) {
    cout << "空数组，无法修读任何课程" << endl;
    return 0;
}

int n = courses.size();

cout << "原始课程安排:" << endl;
for (int i = 0; i < n; i++) {
    cout << "课程" << i << ":" [持续时间=" << courses[i][0] << ", 截止时间=" << courses[i][1]
<< "]" << endl;
}

// 按照课程的截止时间进行排序
sort(courses.begin(), courses.end(), [] (const vector<int>& a, const vector<int>& b) {
    return a[1] < b[1];
});

cout << "\n按截止时间排序后的课程安排:" << endl;
for (int i = 0; i < n; i++) {
    cout << "课程" << i << ":" [持续时间=" << courses[i][0] << ", 截止时间=" << courses[i][1]
<< "]" << endl;
}

priority_queue<int> heap;
int currentTime = 0;
int selectedCount = 0;

cout << "\n选课过程:" << endl;
for (int i = 0; i < n; i++) {
    int duration = courses[i][0];
    int lastDay = courses[i][1];

    cout << "\n考虑课程" << i << ":" [持续时间=" << duration << ", 截止时间=" << lastDay <<
"]" << endl;
    cout << "当前时间: " << currentTime << endl;

    // 如果当前时间加上课程持续时间不超过截止时间
    if (currentTime + duration <= lastDay) {
        heap.push(duration);
        currentTime += duration;
        selectedCount++;
        cout << "选择该课程，当前已选课程数: " << selectedCount << ", 当前时间更新为: " <<
    }
}

```

```

currentTime << endl;
}

// 如果超过截止时间，但当前课程的持续时间比已选课程中最长的短
else if (!heap.empty() && heap.top() > duration) {
    int longestDuration = heap.top();
    heap.pop();

    cout << "替换策略：用当前课程(持续时间=" << duration << ")替换已选课程中持续时间最长的课程(持续时间=" << longestDuration << ")" << endl;

    currentTime += duration - longestDuration;
    heap.push(duration);

    cout << "替换完成，当前时间更新为：" << currentTime << ", 已选课程数保持不变：" <<
selectedCount << endl;
}

else {
    cout << "无法选择该课程，跳过" << endl;
}
}

// 打印当前堆的内容
cout << "当前已选课程的持续时间：[";
priority_queue<int> temp = heap;
while (!temp.empty()) {
    cout << temp.top();
    temp.pop();
    if (!temp.empty()) cout << ", ";
}
cout << "]" << endl;
}

cout << "\n最终结果：最多可以修读 " << heap.size() << " 门课程" << endl;
return heap.size();
}

```

```

/**
 * 打印二维数组的辅助函数
 *
 * @param courses 课程数组
 */
void printCourses(const vector<vector<int>>& courses) {
    cout << "[";
    for (int i = 0; i < courses.size(); i++) {
        cout << "[" << courses[i][0] << "," << courses[i][1] << "]";
        if (i < courses.size() - 1) cout << ",";
    }
}
```

```

    }

    cout << "][";

}

/***
 * 测试函数：验证课程表 III 算法的正确性
 */
void testScheduleCourse() {
    cout << "课程表 III 算法测试开始" << endl;
    cout << "======" << endl;

    // 测试用例 1: [[100, 200], [200, 1300], [1000, 1250], [2000, 3200]]
    vector<vector<int>> courses1 = {{100, 200}, {200, 1300}, {1000, 1250}, {2000, 3200}};
    int result1 = scheduleCourse(courses1);
    cout << "输入: ";
    printCourses(courses1);
    cout << endl;
    cout << "输出: " << result1 << endl;
    cout << "预期: 3" << endl;
    cout << (result1 == 3 ? "✓ 通过" : "✗ 失败") << endl << endl;

    // 测试用例 2: [[1, 2]]
    vector<vector<int>> courses2 = {{1, 2}};
    int result2 = scheduleCourse(courses2);
    cout << "输入: [[1, 2]]" << endl;
    cout << "输出: " << result2 << endl;
    cout << "预期: 1" << endl;
    cout << (result2 == 1 ? "✓ 通过" : "✗ 失败") << endl << endl;

    // 测试用例 3: [[3, 2], [4, 3]]
    vector<vector<int>> courses3 = {{3, 2}, {4, 3}};
    int result3 = scheduleCourse(courses3);
    cout << "输入: [[3, 2], [4, 3]]" << endl;
    cout << "输出: " << result3 << endl;
    cout << "预期: 0" << endl;
    cout << (result3 == 0 ? "✓ 通过" : "✗ 失败") << endl << endl;

    // 测试用例 4: [[5, 5], [4, 6], [2, 6]]
    vector<vector<int>> courses4 = {{5, 5}, {4, 6}, {2, 6}};
    int result4 = scheduleCourse(courses4);
    cout << "输入: [[5, 5], [4, 6], [2, 6]]" << endl;
    cout << "输出: " << result4 << endl;
    cout << "预期: 2" << endl;
}

```

```

cout << (result4 == 2 ? "✓ 通过" : "✗ 失败") << endl << endl;

cout << "测试结束" << endl;
}

/***
 * 性能测试：测试算法在大规模数据下的表现
 */
void performanceTest() {
    cout << "性能测试开始" << endl;
    cout << "======" << endl;

    // 生成大规模测试数据
    int n = 10000;
    vector<vector<int>> courses;
    for (int i = 0; i < n; i++) {
        int duration = rand() % 100 + 1;
        int lastDay = rand() % 1000 + duration;
        courses.push_back({duration, lastDay});
    }

    clock_t start = clock();
    int result = scheduleCourse(courses);
    clock_t end = clock();

    double duration = double(end - start) / CLOCKS_PER_SEC * 1000; // 转换为毫秒

    cout << "数据规模：" << n << " 门课程" << endl;
    cout << "执行时间：" << duration << " 毫秒" << endl;
    cout << "最多可以修读的课程数：" << result << endl;
    cout << "性能测试结束" << endl;
}

/***
 * 主函数：运行测试
 */
int main() {
    cout << "课程表 III - 贪心算法 + 最大堆解决方案 (C++实现)" << endl;
    cout << "======" << endl;

    // 运行基础测试
    testScheduleCourse();
}

```

```

cout << endl << "调试模式示例:" << endl;
vector<vector<int>> debugCourses = {{100, 200}, {200, 1300}, {1000, 1250}, {2000, 3200}};
cout << "对测试用例 [[100, 200], [200, 1300], [1000, 1250], [2000, 3200]] 进行调试跟踪:" << endl;
int debugResult = debugScheduleCourse(debugCourses);
cout << "最终结果: " << debugResult << endl;

cout << endl << "算法分析:" << endl;
cout << "- 时间复杂度: O(n*logn) - 排序需要 O(n*logn)，每个课程进出堆一次需要 O(logn)" << endl;
cout << "- 空间复杂度: O(n) - 最坏情况下所有课程都被选中，堆的大小为 n" << endl;
cout << "- 贪心策略: 按照截止时间排序，使用最大堆管理已选课程" << endl;
cout << "- 最优性: 这种贪心策略能够得到全局最优解" << endl;
cout << "- 替换策略: 当遇到冲突时，用短课程替换长课程可以最大化课程数量" << endl;

// 可选: 运行性能测试
// cout << endl << "性能测试:" << endl;
// performanceTest();

return 0;
}
=====

文件: Code05_CourseScheduleIII.java
=====

package class089;

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

/**
 * 课程表 III - 贪心算法 + 最大堆解决方案
 *
 * 题目描述:
 * 这里有 n 门不同的在线课程，按从 1 到 n 编号
 * 给你一个数组 courses，其中 courses[i]=[durationi, lastDayi] 表示第 i 门课将会持续上 durationi 天课
 * 并且必须在不晚于 lastDayi 的时候完成
 * 你的学期从第 1 天开始，且不能同时修读两门及两门以上的课程
 * 返回你最多可以修读的课程数目
 *
 * 测试链接: https://leetcode.cn/problems/course-schedule-iii/
 *
 * 算法思想:
 */

```

- * 贪心算法 + 最大堆（优先队列）
- * 1. 按照课程的截止时间进行排序（截止时间早的排在前面）
- * 2. 使用最大堆来存储已选课程的持续时间
- * 3. 遍历每个课程：
 - 如果当前时间加上课程持续时间不超过截止时间，则选择该课程
 - 如果超过截止时间，则检查是否可以替换已选课程中持续时间最长的课程
- *
- * 时间复杂度分析：
- * $O(n * \log n)$ - 排序需要 $O(n * \log n)$ ，每个课程进出堆一次需要 $O(\log n)$
- *
- * 空间复杂度分析：
- * $O(n)$ - 最坏情况下所有课程都被选中，堆的大小为 n
- *
- * 是否为最优解：
- * 是，这是解决该问题的最优解
- *
- * 工程化考量：
- * 1. 边界条件处理：处理空数组、单个课程等特殊情况
- * 2. 输入验证：检查课程时间是否有效（持续时间 > 0 , 截止时间 > 0 ）
- * 3. 异常处理：对非法输入进行检查
- * 4. 可读性：使用清晰的变量命名和详细的注释
- *
- * 贪心策略证明：
- * 按照截止时间排序可以保证我们优先考虑截止时间早的课程
- * 使用最大堆来管理已选课程，当遇到冲突时替换持续时间最长的课程，可以最大化课程数量

```
public class Code05_CourseScheduleIII {
```

```
    /**
     * 计算最多可以修读的课程数目
     *
     * @param courses 课程数组，每个元素是 [durationi, lastDayi]
     * @return 最多可以修读的课程数目
     *
     * 算法步骤：
     * 1. 按照课程的截止时间进行排序
     * 2. 使用最大堆存储已选课程的持续时间
     * 3. 维护当前已使用的时间
     * 4. 遍历每个课程，动态调整已选课程
     */
    public static int scheduleCourse(int[][] courses) {
        // 输入验证
        if (courses == null || courses.length == 0) {
```

```

    return 0;
}

int n = courses.length;

// 按照课程的截止时间进行排序（截止时间早的排在前面）
Arrays.sort(courses, (a, b) -> a[1] - b[1]);

// 最大堆，存储已选课程的持续时间（持续时间长的在堆顶）
PriorityQueue<Integer> heap = new PriorityQueue<>((a, b) -> b - a);
int currentTime = 0; // 当前已使用的时间

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

    // 验证课程时间有效性
    if (duration <= 0 || lastDay <= 0) {
        throw new IllegalArgumentException("课程持续时间和截止时间必须大于 0");
    }

    // 如果当前时间加上课程持续时间不超过截止时间
    if (currentTime + duration <= lastDay) {
        heap.add(duration);
        currentTime += duration;
    }
    // 如果超过截止时间，但当前课程的持续时间比已选课程中最长的短
    else if (!heap.isEmpty() && heap.peek() > duration) {
        // 替换策略：用当前课程替换已选课程中持续时间最长的课程
        int longestDuration = heap.poll();
        currentTime += duration - longestDuration;
        heap.add(duration);
    }
    // 其他情况：不选择当前课程
}

return heap.size();
}

/**
 * 调试版本：打印计算过程中的中间结果
 *
 * @param courses 课程数组

```

```
* @return 最多可以修读的课程数目
*/
public static int debugScheduleCourse(int[][] courses) {
    if (courses == null || courses.length == 0) {
        System.out.println("空数组，无法修读任何课程");
        return 0;
    }

    int n = courses.length;

    System.out.println("原始课程安排:");
    for (int i = 0; i < n; i++) {
        System.out.println("课程" + i + ": [持续时间=" + courses[i][0] + ", 截止时间=" +
courses[i][1] + "]");
    }

    // 按照课程的截止时间进行排序
    Arrays.sort(courses, (a, b) -> a[1] - b[1]);

    System.out.println("按截止时间排序后的课程安排:");
    for (int i = 0; i < n; i++) {
        System.out.println("课程" + i + ": [持续时间=" + courses[i][0] + ", 截止时间=" +
courses[i][1] + "]");
    }

    PriorityQueue<Integer> heap = new PriorityQueue<>((a, b) -> b - a);
    int currentTime = 0;
    int selectedCount = 0;

    System.out.println("选课过程:");
    for (int i = 0; i < n; i++) {
        int duration = courses[i][0];
        int lastDay = courses[i][1];

        System.out.println("考虑课程" + i + ": [持续时间=" + duration + ", 截止时间=" +
lastDay + "]");
        System.out.println("当前时间: " + currentTime);

        // 如果当前时间加上课程持续时间不超过截止时间
        if (currentTime + duration <= lastDay) {
            heap.add(duration);
            currentTime += duration;
            selectedCount++;
        }
    }
}
```

```

        System.out.println("选择该课程, 当前已选课程数: " + selectedCount + ", 当前时间更新为: " + currentTime);
    }

    // 如果超过截止时间, 但当前课程的持续时间比已选课程中最长的短
    else if (!heap.isEmpty() && heap.peek() > duration) {
        int longestDuration = heap.poll();

        System.out.println("替换策略: 用当前课程(持续时间=" + duration + ") 替换已选课程中持续时间最长的课程(持续时间=" + longestDuration + ")");

        currentTime += duration - longestDuration;
        heap.add(duration);

        System.out.println("替换完成, 当前时间更新为: " + currentTime + ", 已选课程数保持不变: " + selectedCount);
    }
    else {
        System.out.println("无法选择该课程, 跳过");
    }

    System.out.println("当前已选课程的持续时间: " + heap);
}

System.out.println("最终结果: 最多可以修读 " + heap.size() + " 门课程");
return heap.size();
}

/***
 * 测试函数: 验证课程表 III 算法的正确性
 */
public static void testScheduleCourse() {
    System.out.println("课程表 III 算法测试开始");
    System.out.println("=====");

    // 测试用例 1: [[100, 200], [200, 1300], [1000, 1250], [2000, 3200]]
    int[][] courses1 = {{100, 200}, {200, 1300}, {1000, 1250}, {2000, 3200}};
    int result1 = scheduleCourse(courses1);
    System.out.println("输入: [[100, 200], [200, 1300], [1000, 1250], [2000, 3200]]");
    System.out.println("输出: " + result1);
    System.out.println("预期: 3");
    System.out.println((result1 == 3 ? "✓ 通过" : "✗ 失败"));
    System.out.println();

    // 测试用例 2: [[1, 2]]
    int[][] courses2 = {{1, 2}};

```

```

int result2 = scheduleCourse(courses2);
System.out.println("输入: [[1, 2]]");
System.out.println("输出: " + result2);
System.out.println("预期: 1");
System.out.println((result2 == 1 ? "✓ 通过" : "✗ 失败"));
System.out.println();

// 测试用例 3: [[3, 2], [4, 3]]
int[][] courses3 = {{3, 2}, {4, 3}};
int result3 = scheduleCourse(courses3);
System.out.println("输入: [[3, 2], [4, 3]]");
System.out.println("输出: " + result3);
System.out.println("预期: 0");
System.out.println((result3 == 0 ? "✓ 通过" : "✗ 失败"));
System.out.println();

// 测试用例 4: [[5, 5], [4, 6], [2, 6]]
int[][] courses4 = {{5, 5}, {4, 6}, {2, 6}};
int result4 = scheduleCourse(courses4);
System.out.println("输入: [[5, 5], [4, 6], [2, 6]]");
System.out.println("输出: " + result4);
System.out.println("预期: 2");
System.out.println((result4 == 2 ? "✓ 通过" : "✗ 失败"));
System.out.println();

System.out.println("测试结束");
}

*/

```

```

/***
 * 性能测试：测试算法在大规模数据下的表现
 */
public static void performanceTest() {
    System.out.println("性能测试开始");
    System.out.println("====");

    // 生成大规模测试数据
    int n = 10000;
    int[][] courses = new int[n][2];
    for (int i = 0; i < n; i++) {
        int duration = (int)(Math.random() * 100) + 1;
        int lastDay = (int)(Math.random() * 1000) + duration;
        courses[i][0] = duration;
        courses[i][1] = lastDay;
    }
}
```

```

    }

    long startTime = System.currentTimeMillis();
    int result = scheduleCourse(courses);
    long endTime = System.currentTimeMillis();

    System.out.println("数据规模: " + n + " 门课程");
    System.out.println("执行时间: " + (endTime - startTime) + " 毫秒");
    System.out.println("最多可以修读的课程数: " + result);
    System.out.println("性能测试结束");

}

/***
 * 主函数: 运行测试
 */
public static void main(String[] args) {
    System.out.println("课程表 III - 贪心算法 + 最大堆解决方案");
    System.out.println("=====");

    // 运行基础测试
    testScheduleCourse();

    System.out.println("调试模式示例:");
    int[][] debugCourses = {{100, 200}, {200, 1300}, {1000, 1250}, {2000, 3200}};
    System.out.println("对测试用例 [[100, 200], [200, 1300], [1000, 1250], [2000, 3200]] 进行调试跟踪:");
    int debugResult = debugScheduleCourse(debugCourses);
    System.out.println("最终结果: " + debugResult);

    System.out.println("算法分析:");
    System.out.println("- 时间复杂度: O(n*logn) - 排序需要 O(n*logn), 每个课程进出堆一次需要 O(logn)");
    System.out.println("- 空间复杂度: O(n) - 最坏情况下所有课程都被选中, 堆的大小为 n");
    System.out.println("- 贪心策略: 按照截止时间排序, 使用最大堆管理已选课程");
    System.out.println("- 最优性: 这种贪心策略能够得到全局最优解");
    System.out.println("- 替换策略: 当遇到冲突时, 用短课程替换长课程可以最大化课程数量");

    // 可选: 运行性能测试
    // System.out.println("性能测试:");
    // performanceTest();
}
}

```

文件: Code05_CourseScheduleIII.py

"""

课程表 III - 贪心算法 + 最大堆解决方案 (Python 实现)

题目描述:

这里有 n 门不同的在线课程，按从 1 到 n 编号

给你一个数组 courses，其中 courses[i]=[durationi, lastDayi] 表示第 i 门课将会持续上 durationi 天课并且必须在不晚于 lastDayi 的时候完成

你的学期从第 1 天开始，且不能同时修读两门及两门以上的课程

返回你最多可以修读的课程数目

测试链接: <https://leetcode.cn/problems/course-schedule-iii/>

算法思想:

贪心算法 + 最大堆 (优先队列)

1. 按照课程的截止时间进行排序 (截止时间早的排在前面)
2. 使用最大堆来存储已选课程的持续时间
3. 遍历每个课程:
 - 如果当前时间加上课程持续时间不超过截止时间，则选择该课程
 - 如果超过截止时间，则检查是否可以替换已选课程中持续时间最长的课程

时间复杂度分析:

$O(n \log n)$ - 排序需要 $O(n \log n)$ ，每个课程进出堆一次需要 $O(\log n)$

空间复杂度分析:

$O(n)$ - 最坏情况下所有课程都被选中，堆的大小为 n

是否为最优解:

是，这是解决该问题的最优解

工程化考量:

1. 边界条件处理: 处理空数组、单个课程等特殊情况
2. 输入验证: 检查课程时间是否有效 (持续时间 > 0, 截止时间 > 0)
3. 异常处理: 对非法输入进行检查
4. 可读性: 使用清晰的变量命名和详细的注释

贪心策略证明:

按照截止时间排序可以保证我们优先考虑截止时间早的课程

使用最大堆来管理已选课程，当遇到冲突时替换持续时间最长的课程，可以最大化课程数量

"""

```
import heapq
from typing import List

class Code05_CourseScheduleIII:

    @staticmethod
    def schedule_course(courses: List[List[int]]) -> int:
        """
        计算最多可以修读的课程数目

        Args:
            courses: 课程数组，每个元素是[durationi, lastDayi]

        Returns:
            最多可以修读的课程数目

        算法步骤:
        1. 按照课程的截止时间进行排序
        2. 使用最大堆存储已选课程的持续时间
        3. 维护当前已使用的时间
        4. 遍历每个课程，动态调整已选课程
        """

        # 输入验证
        if not courses:
            return 0

        # 按照课程的截止时间进行排序（截止时间早的排在前面）
        courses.sort(key=lambda x: x[1])

        # 最大堆，存储已选课程的持续时间（使用负值实现最大堆）
        heap = []
        current_time = 0  # 当前已使用的时间

        for duration, last_day in courses:
            # 验证课程时间有效性
            if duration <= 0 or last_day <= 0:
                raise ValueError("课程持续时间和截止时间必须大于 0")

            # 如果当前时间加上课程持续时间不超过截止时间
            if current_time + duration <= last_day:
                # 使用负值实现最大堆
                heapq.heappush(heap, -duration)
```

```
    current_time += duration
    # 如果超过截止时间，但当前课程的持续时间比已选课程中最长的短
    elif heap and -heap[0] > duration:
        # 替换策略：用当前课程替换已选课程中持续时间最长的课程
        longest_duration = -heapq.heappop(heap)
        current_time += duration - longest_duration
        heapq.heappush(heap, -duration)
    # 其他情况：不选择当前课程

    return len(heap)
```

```
@staticmethod
def debug_schedule_course(courses: List[List[int]]) -> int:
    """
```

调试版本：打印计算过程中的中间结果

Args:

courses: 课程数组

Returns:

最多可以修读的课程数目

```
"""
```

```
if not courses:
```

```
    print("空数组，无法修读任何课程")
```

```
    return 0
```

```
print("原始课程安排:")
```

```
for i, (duration, last_day) in enumerate(courses):
```

```
    print(f"课程{i}: [持续时间={duration}, 截止时间={last_day}]")
```

```
# 按照课程的截止时间进行排序
```

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

```
print("\n按截止时间排序后的课程安排:")
```

```
for i, (duration, last_day) in enumerate(courses):
```

```
    print(f"课程{i}: [持续时间={duration}, 截止时间={last_day}]")
```

```
heap = []
```

```
current_time = 0
```

```
selected_count = 0
```

```
print("\n选课过程:")
```

```
for i, (duration, last_day) in enumerate(courses):
```

```

print(f"\n 考虑课程 {i}: [持续时间={duration}, 截止时间={last_day}]")
print(f"当前时间: {current_time}")

# 如果当前时间加上课程持续时间不超过截止时间
if current_time + duration <= last_day:
    heapq.heappush(heap, -duration)
    current_time += duration
    selected_count += 1
    print(f"选择该课程, 当前已选课程数: {selected_count}, 当前时间更新为:
{current_time}")

# 如果超过截止时间, 但当前课程的持续时间比已选课程中最长的短
elif heap and -heap[0] > duration:
    longest_duration = -heapq.heappop(heap)
    print(f"替换策略: 用当前课程(持续时间={duration})替换已选课程中持续时间最长的课程
(持续时间={longest_duration})")

    current_time += duration - longest_duration
    heapq.heappush(heap, -duration)
    print(f"替换完成, 当前时间更新为: {current_time}, 已选课程数保持不变:
{selected_count}")

else:
    print("无法选择该课程, 跳过")

# 打印当前堆的内容
heap_contents = [-x for x in heap]
heap_contents.sort(reverse=True)
print(f"当前已选课程的持续时间: {heap_contents}")

print(f"\n 最终结果: 最多可以修读 {len(heap)} 门课程")
return len(heap)

@staticmethod
def test_schedule_course():
    """
    测试函数: 验证课程表 III 算法的正确性
    """
    print("课程表 III 算法测试开始")
    print("=====")

    # 测试用例 1: [[100, 200], [200, 1300], [1000, 1250], [2000, 3200]]
    courses1 = [[100, 200], [200, 1300], [1000, 1250], [2000, 3200]]
    result1 = Code05_CourseScheduleIII.schedule_course(courses1)
    print("输入: [[100, 200], [200, 1300], [1000, 1250], [2000, 3200]]")

```

```

print("输出:", result1)
print("预期: 3")
print("✓ 通过" if result1 == 3 else "✗ 失败")
print()

# 测试用例 2: [[1,2]]
courses2 = [[1,2]]
result2 = Code05_CourseScheduleIII.schedule_course(courses2)
print("输入: [[1,2]]")
print("输出:", result2)
print("预期: 1")
print("✓ 通过" if result2 == 1 else "✗ 失败")
print()

# 测试用例 3: [[3,2], [4,3]]
courses3 = [[3,2], [4,3]]
result3 = Code05_CourseScheduleIII.schedule_course(courses3)
print("输入: [[3,2], [4,3]]")
print("输出:", result3)
print("预期: 0")
print("✓ 通过" if result3 == 0 else "✗ 失败")
print()

# 测试用例 4: [[5,5], [4,6], [2,6]]
courses4 = [[5,5], [4,6], [2,6]]
result4 = Code05_CourseScheduleIII.schedule_course(courses4)
print("输入: [[5,5], [4,6], [2,6]]")
print("输出:", result4)
print("预期: 2")
print("✓ 通过" if result4 == 2 else "✗ 失败")
print()

print("测试结束")

```

```

@staticmethod
def performance_test():
    """
    性能测试: 测试算法在大规模数据下的表现
    """

    import time
    import random

    print("性能测试开始")

```

```
print("=====")

# 生成大规模测试数据
n = 10000
courses = []
for i in range(n):
    duration = random.randint(1, 100)
    last_day = random.randint(duration, 1000)
    courses.append([duration, last_day])

start_time = time.time()
result = Code05_CourseScheduleIII.schedule_course(courses)
end_time = time.time()

print(f"数据规模: {n} 门课程")
print(f"执行时间: {end_time - start_time:.4f} 秒")
print(f"最多可以修读的课程数: {result}")
print("性能测试结束")
```

@staticmethod

```
def compare_with_alternative(courses: List[List[int]]) -> None:
```

"""

与替代解法对比

Args:

courses: 课程数组

"""

```
print("贪心算法结果:", Code05_CourseScheduleIII.schedule_course(courses))
```

替代解法: 暴力回溯 (仅用于小规模对比)

```
def backtrack(courses, index, current_time, count):
```

if index == len(courses):

return count

不选择当前课程

```
max_count = backtrack(courses, index + 1, current_time, count)
```

尝试选择当前课程

```
duration, last_day = courses[index]
```

if current_time + duration <= last_day:

```
    max_count = max(max_count, backtrack(courses, index + 1, current_time + duration,
count + 1))
```

```

    return max_count

    if len(courses) <= 10: # 只对小规模数据进行暴力计算
        sorted_courses = sorted(courses, key=lambda x: x[1])
        brute_result = backtrack(sorted_courses, 0, 0, 0)
        print("暴力回溯结果:", brute_result)
    else:
        print("暴力回溯: 数据规模太大, 无法计算")

def main():
    """
    主函数: 运行测试
    """

    print("课程表 III - 贪心算法 + 最大堆解决方案 (Python 实现)")
    print("=====")

    # 运行基础测试
    Code05_CourseScheduleIII.test_schedule_course()

    print("\n调试模式示例:")
    debug_courses = [[100, 200], [200, 1300], [1000, 1250], [2000, 3200]]
    print("对测试用例 [[100, 200], [200, 1300], [1000, 1250], [2000, 3200]] 进行调试跟踪:")
    debug_result = Code05_CourseScheduleIII.debug_schedule_course(debug_courses)
    print("最终结果:", debug_result)

    print("\n算法分析:")
    print("- 时间复杂度: O(n*logn) - 排序需要 O(n*logn), 每个课程进出堆一次需要 O(logn)")
    print("- 空间复杂度: O(n) - 最坏情况下所有课程都被选中, 堆的大小为 n")
    print("- 贪心策略: 按照截止时间排序, 使用最大堆管理已选课程")
    print("- 最优性: 这种贪心策略能够得到全局最优解")
    print("- Python 特性: 使用负值实现最大堆")
    print("- 替换策略: 当遇到冲突时, 用短课程替换长课程可以最大化课程数量")

    # 可选: 运行性能测试
    # print("\n性能测试:")
    # Code05_CourseScheduleIII.performance_test()

    # 可选: 与替代解法对比
    # print("\n算法对比:")
    # test_courses = [[100, 200], [200, 1300], [1000, 1250], [2000, 3200]]
    # Code05_CourseScheduleIII.compare_with_alternative(test_courses)

```

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

文件: Code06_MinimumCostToConnectSticks1.cpp

```
/***
 * 连接棒材的最低费用 - 贪心算法 + 最小堆解决方案 (C++实现, LeetCode 版本)
 *
 * 题目描述:
 * 你有一些长度为正整数的棍子, 这些长度以数组 sticks 的形式给出
 * sticks[i]是第 i 个木棍的长度
 * 你可以通过支付  $x+y$  的成本将任意两个长度为  $x$  和  $y$  的棍子连接成一个棍子
 * 你必须连接所有的棍子, 直到剩下一个棍子
 * 返回以这种方式将所有给定的棍子连接成一个棍子的最小成本
 *
 * 测试链接: https://leetcode.cn/problems/minimum-cost-to-connect-sticks/
 *
 * 算法思想:
 * 贪心算法 + 最小堆 (优先队列)
 * 1. 使用最小堆存储所有棍子的长度
 * 2. 每次从堆中取出两根最短的棍子进行连接
 * 3. 将连接后的新棍子放回堆中
 * 4. 重复直到只剩下一根棍子
 *
 * 时间复杂度分析:
 *  $O(n \log n)$  - 每个棍子进出堆一次需要  $O(\log n)$ , 总共需要  $n-1$  次连接操作
 *
 * 空间复杂度分析:
 *  $O(n)$  - 堆的大小为  $n$ 
 *
 * 是否为最优解:
 * 是, 这是解决该问题的最优解 (哈夫曼编码思想)
 *
 * 工程化考量:
 * 1. 边界条件处理: 处理空数组、单个棍子等特殊情况
 * 2. 输入验证: 检查棍子长度是否为正整数
 * 3. 异常处理: 对非法输入进行检查
 * 4. 可读性: 使用清晰的变量命名和详细的注释
 *
 * 贪心策略证明:
```

```
* 这是经典的哈夫曼编码问题，每次选择最小的两个元素合并可以保证总成本最小
* 这种策略满足贪心选择性质和最优子结构性质
*/
```

```
#include <iostream>
#include <vector>
#include <queue>
#include <stdexcept>
#include <algorithm>

using namespace std;

/***
 * 计算连接所有棍子的最小成本
 *
 * @param sticks 棍子长度数组
 * @return 最小连接成本
 *
 * 算法步骤：
 * 1. 将棍子长度加入最小堆
 * 2. 当堆中棍子数量大于 1 时：
 *     - 取出两根最短的棍子
 *     - 计算连接成本并累加
 *     - 将连接后的新棍子放回堆中
 * 3. 返回总成本
*/
int connectSticks(vector<int>& sticks) {
    // 输入验证
    if (sticks.empty()) {
        return 0;
    }

    // 单个棍子不需要连接
    if (sticks.size() == 1) {
        return 0;
    }

    // 最小堆，存储棍子长度
    priority_queue<int, vector<int>, greater<int>> heap;
    for (int stick : sticks) {
        // 验证棍子长度有效性
        if (stick <= 0) {
            throw invalid_argument("棍子长度必须为正整数");
        }
    }
}
```

```
    }

    heap.push(stick);

}

int totalCost = 0; // 总连接成本

// 当堆中还有多于一根棍子时继续连接
while (heap.size() > 1) {
    // 取出两根最短的棍子
    int first = heap.top();
    heap.pop();
    int second = heap.top();
    heap.pop();

    // 计算连接成本
    int cost = first + second;
    totalCost += cost;

    // 将连接后的新棍子放回堆中
    heap.push(cost);
}

return totalCost;
}

/***
 * 调试版本：打印计算过程中的中间结果
 *
 * @param sticks 棍子长度数组
 * @return 最小连接成本
 */
int debugConnectSticks(vector<int>& sticks) {
    if (sticks.empty()) {
        cout << "空数组，不需要连接" << endl;
        return 0;
    }

    if (sticks.size() == 1) {
        cout << "单个棍子，不需要连接" << endl;
        return 0;
    }

    cout << "原始棍子长度：";
```

```

for (int stick : sticks) {
    cout << stick << " ";
}
cout << endl;

priority_queue<int, vector<int>, greater<int>> heap;
for (int stick : sticks) {
    heap.push(stick);
}

int totalCost = 0;
int step = 1;

cout << "\n连接过程:" << endl;
while (heap.size() > 1) {
    cout << "步骤 " << step << ":" << endl;

    // 取出两根最短的棍子
    int first = heap.top();
    heap.pop();
    int second = heap.top();
    heap.pop();
    cout << " 取出两根最短的棍子: " << first << " 和 " << second << endl;

    // 计算连接成本
    int cost = first + second;
    totalCost += cost;
    cout << " 连接成本: " << first << " + " << second << " = " << cost << endl;
    cout << " 累计总成本: " << totalCost << endl;

    // 将连接后的新棍子放回堆中
    heap.push(cost);
    cout << " 将新棍子(" << cost << ")放回堆中" << endl;

    // 打印当前堆的状态
    cout << " 当前堆中棍子: ";
    priority_queue<int, vector<int>, greater<int>> temp = heap;
    while (!temp.empty()) {
        cout << temp.top() << " ";
        temp.pop();
    }
    cout << endl;
}

```

```

        step++;
    }

    cout << "\n 最终结果: 最小连接成本 = " << totalCost << endl;
    return totalCost;
}

/***
 * 打印数组的辅助函数
 *
 * @param sticks 棍子长度数组
 */
void printSticks(const vector<int>& sticks) {
    cout << "[";
    for (int i = 0; i < sticks.size(); i++) {
        cout << sticks[i];
        if (i < sticks.size() - 1) cout << ",";
    }
    cout << "]";
}

/***
 * 测试函数: 验证连接棒材算法的正确性
 */
void testConnectSticks() {
    cout << "连接棒材的最低费用算法测试开始" << endl;
    cout << "===== " << endl;

    // 测试用例 1: [2, 4, 3]
    vector<int> sticks1 = {2, 4, 3};
    int result1 = connectSticks(sticks1);
    cout << "输入: ";
    printSticks(sticks1);
    cout << endl;
    cout << "输出: " << result1 << endl;
    cout << "预期: 14" << endl;
    cout << (result1 == 14 ? "✓ 通过" : "✗ 失败") << endl << endl;

    // 测试用例 2: [1, 8, 3, 5]
    vector<int> sticks2 = {1, 8, 3, 5};
    int result2 = connectSticks(sticks2);
    cout << "输入: ";
    printSticks(sticks2);
}

```

```

cout << endl;
cout << "输出: " << result2 << endl;
cout << "预期: 30" << endl;
cout << (result2 == 30 ? "✓ 通过" : "✗ 失败") << endl << endl;

// 测试用例 3: 空数组
vector<int> sticks3 = {};
int result3 = connectSticks(sticks3);
cout << "输入: []" << endl;
cout << "输出: " << result3 << endl;
cout << "预期: 0" << endl;
cout << (result3 == 0 ? "✓ 通过" : "✗ 失败") << endl << endl;

// 测试用例 4: 单个棍子
vector<int> sticks4 = {5};
int result4 = connectSticks(sticks4);
cout << "输入: [5]" << endl;
cout << "输出: " << result4 << endl;
cout << "预期: 0" << endl;
cout << (result4 == 0 ? "✓ 通过" : "✗ 失败") << endl << endl;

cout << "测试结束" << endl;
}

```

```

/**
 * 性能测试: 测试算法在大规模数据下的表现
 */
void performanceTest() {
    cout << "性能测试开始" << endl;
    cout << "======" << endl;

    // 生成大规模测试数据
    int n = 10000;
    vector<int> sticks;
    for (int i = 0; i < n; i++) {
        sticks.push_back(rand() % 1000 + 1); // 1-1000 的随机数
    }

    clock_t start = clock();
    int result = connectSticks(sticks);
    clock_t end = clock();

    double duration = double(end - start) / CLOCKS_PER_SEC * 1000; // 转换为毫秒
}

```

```

cout << "数据规模: " << n << " 根棍子" << endl;
cout << "执行时间: " << duration << " 毫秒" << endl;
cout << "最小连接成本: " << result << endl;
cout << "性能测试结束" << endl;
}

/***
 * 主函数: 运行测试
 */
int main() {
    cout << "连接棒材的最低费用 - 贪心算法 + 最小堆解决方案 (C++实现, LeetCode 版本)" << endl;
    cout << "===== " << endl;

    // 运行基础测试
    testConnectSticks();

    cout << endl << "调试模式示例:" << endl;
    vector<int> debugSticks = {2, 4, 3};
    cout << "对测试用例 [2, 4, 3] 进行调试跟踪:" << endl;
    int debugResult = debugConnectSticks(debugSticks);
    cout << "最终结果: " << debugResult << endl;

    cout << endl << "算法分析:" << endl;
    cout << "- 时间复杂度: O(n*logn) - 每个棍子进出堆一次需要 O(logn), 总共需要 n-1 次连接操作" << endl;
    cout << "- 空间复杂度: O(n) - 堆的大小为 n" << endl;
    cout << "- 贪心策略: 每次选择最短的两根棍子进行连接 (哈夫曼编码思想)" << endl;
    cout << "- 最优性: 这种贪心策略能够得到全局最优解" << endl;
    cout << "- 数学原理: 这是经典的哈夫曼编码问题" << endl;

    // 可选: 运行性能测试
    // cout << endl << "性能测试:" << endl;
    // performanceTest();

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

文件: Code06_MinimumCostToConnectSticks1.java

```
=====
package class089;
```

```
import java.util.PriorityQueue;

/**
 * 连接棒材的最低费用 - 贪心算法 + 最小堆解决方案 (LeetCode 版本)
 *
 * 题目描述:
 * 你有一些长度为正整数的棍子, 这些长度以数组 sticks 的形式给出
 * sticks[i]是第 i 个木棍的长度
 * 你可以通过支付  $x+y$  的成本将任意两个长度为  $x$  和  $y$  的棍子连接成一个棍子
 * 你必须连接所有的棍子, 直到剩下一个棍子
 * 返回以这种方式将所有给定的棍子连接成一个棍子的最小成本
 *
 * 测试链接: https://leetcode.cn/problems/minimum-cost-to-connect-sticks/
 *
 * 算法思想:
 * 贪心算法 + 最小堆 (优先队列)
 * 1. 使用最小堆存储所有棍子的长度
 * 2. 每次从堆中取出两根最短的棍子进行连接
 * 3. 将连接后的新棍子放回堆中
 * 4. 重复直到只剩下一根棍子
 *
 * 时间复杂度分析:
 *  $O(n \log n)$  - 每个棍子进出堆一次需要  $O(\log n)$ , 总共需要  $n-1$  次连接操作
 *
 * 空间复杂度分析:
 *  $O(n)$  - 堆的大小为  $n$ 
 *
 * 是否为最优解:
 * 是, 这是解决该问题的最优解 (哈夫曼编码思想)
 *
 * 工程化考量:
 * 1. 边界条件处理: 处理空数组、单个棍子等特殊情况
 * 2. 输入验证: 检查棍子长度是否为正整数
 * 3. 异常处理: 对非法输入进行检查
 * 4. 可读性: 使用清晰的变量命名和详细的注释
 *
 * 贪心策略证明:
 * 这是经典的哈夫曼编码问题, 每次选择最小的两个元素合并可以保证总成本最小
 * 这种策略满足贪心选择性质和最优子结构性质
 */

public class Code06_MinimumCostToConnectSticks1 {
```

```
/**  
 * 计算连接所有棍子的最小成本  
 *  
 * @param sticks 棍子长度数组  
 * @return 最小连接成本  
 *  
 * 算法步骤：  
 * 1. 将棍子长度加入最小堆  
 * 2. 当堆中棍子数量大于 1 时：  
 *     - 取出两根最短的棍子  
 *     - 计算连接成本并累加  
 *     - 将连接后的新棍子放回堆中  
 * 3. 返回总成本  
 */  
  
public static int connectSticks(int[] sticks) {  
    // 输入验证  
    if (sticks == null || sticks.length == 0) {  
        return 0;  
    }  
  
    // 单个棍子不需要连接  
    if (sticks.length == 1) {  
        return 0;  
    }  
  
    // 最小堆，存储棍子长度  
    PriorityQueue<Integer> heap = new PriorityQueue<>();  
    for (int stick : sticks) {  
        // 验证棍子长度有效性  
        if (stick <= 0) {  
            throw new IllegalArgumentException("棍子长度必须为正整数");  
        }  
        heap.add(stick);  
    }  
  
    int totalCost = 0; // 总连接成本  
  
    // 当堆中还有多于一根棍子时继续连接  
    while (heap.size() > 1) {  
        // 取出两根最短的棍子  
        int first = heap.poll();  
        int second = heap.poll();
```

```
// 计算连接成本
int cost = first + second;
totalCost += cost;

// 将连接后的新棍子放回堆中
heap.add(cost);

}

return totalCost;
}

/***
 * 调试版本：打印计算过程中的中间结果
 *
 * @param sticks 棍子长度数组
 * @return 最小连接成本
 */
public static int debugConnectSticks(int[] sticks) {
    if (sticks == null || sticks.length == 0) {
        System.out.println("空数组，不需要连接");
        return 0;
    }

    if (sticks.length == 1) {
        System.out.println("单个棍子，不需要连接");
        return 0;
    }

    System.out.println("原始棍子长度：");
    for (int i = 0; i < sticks.length; i++) {
        System.out.print(sticks[i] + " ");
    }
    System.out.println();

PriorityQueue<Integer> heap = new PriorityQueue<>();
for (int stick : sticks) {
    heap.add(stick);
}

int totalCost = 0;
int step = 1;

System.out.println("连接过程:");

```

```

while (heap.size() > 1) {
    System.out.println("步骤 " + step + ":");

    // 取出两根最短的棍子
    int first = heap.poll();
    int second = heap.poll();
    System.out.println(" 取出两根最短的棍子: " + first + " 和 " + second);

    // 计算连接成本
    int cost = first + second;
    totalCost += cost;
    System.out.println(" 连接成本: " + first + " + " + second + " = " + cost);
    System.out.println(" 累计总成本: " + totalCost);

    // 将连接后的新棍子放回堆中
    heap.add(cost);
    System.out.println(" 将新棍子(" + cost + ")放回堆中");

    // 打印当前堆的状态
    System.out.print(" 当前堆中棍子: ");
    PriorityQueue<Integer> temp = new PriorityQueue<>(heap);
    while (!temp.isEmpty()) {
        System.out.print(temp.poll() + " ");
    }
    System.out.println();

    step++;
}

System.out.println("最终结果: 最小连接成本 = " + totalCost);
return totalCost;
}

/**
 * 测试函数: 验证连接棒材算法的正确性
 */
public static void testConnectSticks() {
    System.out.println("连接棒材的最低费用算法测试开始");
    System.out.println("=====");

    // 测试用例 1: [2, 4, 3]
    int[] sticks1 = {2, 4, 3};
    int result1 = connectSticks(sticks1);
}

```

```
System.out.println("输入: [2, 4, 3]");
System.out.println("输出: " + result1);
System.out.println("预期: 14");
System.out.println((result1 == 14 ? "✓ 通过" : "✗ 失败"));
System.out.println();

// 测试用例 2: [1, 8, 3, 5]
int[] sticks2 = {1, 8, 3, 5};
int result2 = connectSticks(sticks2);
System.out.println("输入: [1, 8, 3, 5]");
System.out.println("输出: " + result2);
System.out.println("预期: 30");
System.out.println((result2 == 30 ? "✓ 通过" : "✗ 失败"));
System.out.println();

// 测试用例 3: 空数组
int[] sticks3 = {};
int result3 = connectSticks(sticks3);
System.out.println("输入: []");
System.out.println("输出: " + result3);
System.out.println("预期: 0");
System.out.println((result3 == 0 ? "✓ 通过" : "✗ 失败"));
System.out.println();

// 测试用例 4: 单个棍子
int[] sticks4 = {5};
int result4 = connectSticks(sticks4);
System.out.println("输入: [5]");
System.out.println("输出: " + result4);
System.out.println("预期: 0");
System.out.println((result4 == 0 ? "✓ 通过" : "✗ 失败"));
System.out.println();

System.out.println("测试结束");
}

/**
 * 性能测试: 测试算法在大规模数据下的表现
 */
public static void performanceTest() {
    System.out.println("性能测试开始");
    System.out.println("=====");
```

```

// 生成大规模测试数据
int n = 10000;
int[] sticks = new int[n];
for (int i = 0; i < n; i++) {
    sticks[i] = (int)(Math.random() * 1000) + 1; // 1-1000 的随机数
}

long startTime = System.currentTimeMillis();
int result = connectSticks(sticks);
long endTime = System.currentTimeMillis();

System.out.println("数据规模: " + n + " 根棍子");
System.out.println("执行时间: " + (endTime - startTime) + " 毫秒");
System.out.println("最小连接成本: " + result);
System.out.println("性能测试结束");
}

/**
 * 主函数: 运行测试
 */
public static void main(String[] args) {
    System.out.println("连接棒材的最低费用 - 贪心算法 + 最小堆解决方案 (LeetCode 版本)");
    System.out.println("=====");

    // 运行基础测试
    testConnectSticks();

    System.out.println("调试模式示例:");
    int[] debugSticks = {2, 4, 3};
    System.out.println("对测试用例 [2,4,3] 进行调试跟踪:");
    int debugResult = debugConnectSticks(debugSticks);
    System.out.println("最终结果: " + debugResult);

    System.out.println("算法分析:");
    System.out.println("- 时间复杂度: O(n*logn) - 每个棍子进出堆一次需要 O(logn), 总共需要 n-1 次连接操作");
    System.out.println("- 空间复杂度: O(n) - 堆的大小为 n");
    System.out.println("- 贪心策略: 每次选择最短的两根棍子进行连接 (哈夫曼编码思想)");
    System.out.println("- 最优性: 这种贪心策略能够得到全局最优解");
    System.out.println("- 数学原理: 这是经典的哈夫曼编码问题");

    // 可选: 运行性能测试
    // System.out.println("性能测试:");
}

```

```
// performanceTest();  
}  
}  
  
=====
```

文件: Code06_MinimumCostToConnectSticks1.py

```
=====
```

```
"""
```

连接棒材的最低费用 - 贪心算法 + 最小堆解决方案 (Python 实现, LeetCode 版本)

题目描述:

你有一些长度为正整数的棍子, 这些长度以数组 sticks 的形式给出

sticks[i] 是第 i 个木棍的长度

你可以通过支付 $x+y$ 的成本将任意两个长度为 x 和 y 的棍子连接成一个棍子

你必须连接所有的棍子, 直到剩下一个棍子

返回以这种方式将所有给定的棍子连接成一个棍子的最小成本

测试链接: <https://leetcode.cn/problems/minimum-cost-to-connect-sticks/>

算法思想:

贪心算法 + 最小堆 (优先队列)

1. 使用最小堆存储所有棍子的长度
2. 每次从堆中取出两根最短的棍子进行连接
3. 将连接后的棍子放回堆中
4. 重复直到只剩下一根棍子

时间复杂度分析:

$O(n * \log n)$ - 每个棍子进出堆一次需要 $O(\log n)$, 总共需要 $n-1$ 次连接操作

空间复杂度分析:

$O(n)$ - 堆的大小为 n

是否为最优解:

是, 这是解决该问题的最优解 (哈夫曼编码思想)

工程化考量:

1. 边界条件处理: 处理空数组、单个棍子等特殊情况
2. 输入验证: 检查棍子长度是否为正整数
3. 异常处理: 对非法输入进行检查
4. 可读性: 使用清晰的变量命名和详细的注释

贪心策略证明:

这是经典的哈夫曼编码问题，每次选择最小的两个元素合并可以保证总成本最小
这种策略满足贪心选择性质和最优子结构性质

"""

```
import heapq
from typing import List

class Code06_MinimumCostToConnectSticks1:

    @staticmethod
    def connect_sticks(sticks: List[int]) -> int:
        """
        计算连接所有棍子的最小成本

        Args:
            sticks: 棍子长度数组

        Returns:
            最小连接成本

        算法步骤:
        1. 将棍子长度加入最小堆
        2. 当堆中棍子数量大于 1 时:
            - 取出两根最短的棍子
            - 计算连接成本并累加
            - 将连接后的新棍子放回堆中
        3. 返回总成本
        """
        # 输入验证
        if not sticks:
            return 0

        # 单个棍子不需要连接
        if len(sticks) == 1:
            return 0

        # 最小堆，存储棍子长度
        heap = []
        for stick in sticks:
            # 验证棍子长度有效性
            if stick <= 0:
                raise ValueError("棍子长度必须为正整数")
            heapq.heappush(heap, stick)
```

```
total_cost = 0 # 总连接成本

# 当堆中还有多于一根棍子时继续连接
while len(heap) > 1:
    # 取出两根最短的棍子
    first = heapq.heappop(heap)
    second = heapq.heappop(heap)

    # 计算连接成本
    cost = first + second
    total_cost += cost

    # 将连接后的新棍子放回堆中
    heapq.heappush(heap, cost)

return total_cost
```

```
@staticmethod
def debug_connect_sticks(sticks: List[int]) -> int:
    """
```

调试版本：打印计算过程中的中间结果

Args:

sticks: 棍子长度数组

Returns:

最小连接成本

"""

if not sticks:

print("空数组，不需要连接")

return 0

if len(sticks) == 1:

print("单个棍子，不需要连接")

return 0

print("原始棍子长度:", sticks)

heap = []

for stick in sticks:

heapq.heappush(heap, stick)

```

total_cost = 0
step = 1

print("\n连接过程:")
while len(heap) > 1:
    print(f"步骤 {step}:")

    # 取出两根最短的棍子
    first = heapq.heappop(heap)
    second = heapq.heappop(heap)
    print(f" 取出两根最短的棍子: {first} 和 {second}")

    # 计算连接成本
    cost = first + second
    total_cost += cost
    print(f" 连接成本: {first} + {second} = {cost}")
    print(f" 累计总成本: {total_cost}")

    # 将连接后的新棍子放回堆中
    heapq.heappush(heap, cost)
    print(f" 将新棍子({cost})放回堆中")

    # 打印当前堆的状态
    print(f" 当前堆中棍子: {heap}")

    step += 1

print(f"\n最终结果: 最小连接成本 = {total_cost}")
return total_cost

@staticmethod
def test_connect_sticks():
    """
    测试函数: 验证连接棒材算法的正确性
    """
    print("连接棒材的最低费用算法测试开始")
    print("=====")

    # 测试用例 1: [2, 4, 3]
    sticks1 = [2, 4, 3]
    result1 = Code06_MinimumCostToConnectSticks1.connect_sticks(sticks1)
    print("输入: [2, 4, 3]")
    print("输出:", result1)

```

```
print("预期: 14")
print("✓ 通过" if result1 == 14 else "✗ 失败")
print()

# 测试用例 2: [1, 8, 3, 5]
sticks2 = [1, 8, 3, 5]
result2 = Code06_MinimumCostToConnectSticks1.connect_sticks(sticks2)
print("输入: [1, 8, 3, 5]")
print("输出:", result2)
print("预期: 30")
print("✓ 通过" if result2 == 30 else "✗ 失败")
print()

# 测试用例 3: 空数组
sticks3 = []
result3 = Code06_MinimumCostToConnectSticks1.connect_sticks(sticks3)
print("输入: []")
print("输出:", result3)
print("预期: 0")
print("✓ 通过" if result3 == 0 else "✗ 失败")
print()

# 测试用例 4: 单个棍子
sticks4 = [5]
result4 = Code06_MinimumCostToConnectSticks1.connect_sticks(sticks4)
print("输入: [5]")
print("输出:", result4)
print("预期: 0")
print("✓ 通过" if result4 == 0 else "✗ 失败")
print()

print("测试结束")

@staticmethod
def performance_test():
    """
    性能测试: 测试算法在大规模数据下的表现
    """
    import time
    import random

    print("性能测试开始")
    print("=====")
```

```

# 生成大规模测试数据
n = 10000
sticks = [random.randint(1, 1000) for _ in range(n)] # 1-1000 的随机数

start_time = time.time()
result = Code06_MinimumCostToConnectSticks1.connect_sticks(sticks)
end_time = time.time()

print(f"数据规模: {n} 根棍子")
print(f"执行时间: {end_time - start_time:.4f} 秒")
print(f"最小连接成本: {result}")
print("性能测试结束")

@staticmethod
def compare_with_naive(sticks: List[int]) -> None:
    """
    与暴力解法对比，展示贪心算法的优势
    """

    Args:
        sticks: 棍子长度数组
    """
    print("贪心算法结果:", Code06_MinimumCostToConnectSticks1.connect_sticks(sticks))

    # 暴力解法: 尝试所有可能的连接顺序 (仅用于小规模对比)
    def brute_force(sticks: List[int]) -> int:
        if len(sticks) <= 1:
            return 0

        min_cost = 10**9 # 使用大数代替无穷大
        # 尝试所有可能的连接顺序
        for i in range(len(sticks)):
            for j in range(i + 1, len(sticks)):
                # 连接第 i 和第 j 根棍子
                new_sticks = sticks.copy()
                cost = new_sticks[i] + new_sticks[j]
                new_sticks[i] = cost
                new_sticks.pop(j)

                # 递归计算剩余棍子的最小成本
                total_cost = cost + brute_force(new_sticks)
                min_cost = min(min_cost, total_cost)

    return min_cost

```

```
    return min_cost

    if len(sticks) <= 5: # 只对小规模数据进行暴力计算
        print("暴力解法结果:", brute_force(sticks))
    else:
        print("暴力解法: 数据规模太大, 无法计算")

def main():
    """
    主函数: 运行测试
    """

    print("连接棒材的最低费用 - 贪心算法 + 最小堆解决方案 (Python 实现, LeetCode 版本)")
    print("=====")
    # 运行基础测试
    Code06_MinimumCostToConnectSticks1.test_connect_sticks()

    print("\n调试模式示例:")
    debug_sticks = [2, 4, 3]
    print("对测试用例 [2, 4, 3] 进行调试跟踪:")
    debug_result = Code06_MinimumCostToConnectSticks1.debug_connect_sticks(debug_sticks)
    print("最终结果:", debug_result)

    print("\n算法分析:")
    print("- 时间复杂度: O(n*logn) - 每个棍子进出堆一次需要 O(logn), 总共需要 n-1 次连接操作")
    print("- 空间复杂度: O(n) - 堆的大小为 n")
    print("- 贪心策略: 每次选择最短的两根棍子进行连接 (哈夫曼编码思想)")
    print("- 最优性: 这种贪心策略能够得到全局最优解")
    print("- 数学原理: 这是经典的哈夫曼编码问题")
    print("- Python 特性: 使用 heapq 模块实现最小堆")

    # 可选: 运行性能测试
    # print("\n性能测试:")
    # Code06_MinimumCostToConnectSticks1.performance_test()

    # 可选: 与暴力解法对比
    # print("\n算法对比:")
    # test_sticks = [2, 4, 3]
    # Code06_MinimumCostToConnectSticks1.compare_with_naive(test_sticks)

if __name__ == "__main__":
```

```
main()
```

```
=====
```

文件: Code06_MinimumCostToConnectSticks2.java

```
=====
```

```
package class089;

// 连接棒材的最低费用(洛谷测试)
// 你有一些长度为正整数的棍子
// 这些长度以数组 sticks 的形式给出
// sticks[i]是第 i 个木棍的长度
// 你可以通过支付 x+y 的成本将任意两个长度为 x 和 y 的棍子连接成一个棍子
// 你必须连接所有的棍子, 直到剩下一个棍子
// 返回以这种方式将所有给定的棍子连接成一个棍子的最小成本
// 测试链接 : https://www.luogu.com.cn/problem/P1090
// 请同学们务必参考如下代码中关于输入、输出的处理
// 这是输入输出处理效率很高的写法
// 提交以下的 code, 提交时请把类名改成"Main", 可以直接通过
```

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

```
public class Code06_MinimumCostToConnectSticks2 {
```

```
    public static int MAXN = 10001;
```

```
    public static int[] nums = new int[MAXN];
```

```
    public static int n;
```

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

```
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
```

```
        StreamTokenizer in = new StreamTokenizer(br);
```

```
        PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));
```

```
        while (in.nextToken() != StreamTokenizer.TT_EOF) {
```

```
            n = (int) in.nval;
```

```
            for (int i = 0; i < n; i++) {
```

```
                in.nextToken();
```

```
    nums[i] = (int) in.nval;
}
out.println(minCost());
out.flush();
br.close();
}
```

```
public static int minCost() {
    size = 0;
    for (int i = 0; i < n; i++) {
        add(nums[i]);
    }
    int sum = 0;
    int cur = 0;
    while (size > 1) {
        cur = pop() + pop();
        sum += cur;
        add(cur);
    }
    return sum;
}
```

```
// 手写小根堆
public static int[] heap = new int[MAXN];
```

```
// 堆的大小
public static int size;
```

```
public static void add(int x) {
    heap[size] = x;
    int i = size++;
    while (heap[i] < heap[(i - 1) / 2]) {
        swap(i, (i - 1) / 2);
        i = (i - 1) / 2;
    }
}
```

```
public static int pop() {
    int ans = heap[0];
    swap(0, --size);
```

```

int i = 0, l = 1, best;
while (l < size) {
    best = l + 1 < size && heap[l + 1] < heap[l] ? l + 1 : l;
    best = heap[best] < heap[i] ? best : i;
    if (best == i) {
        break;
    }
    swap(i, best);
    i = best;
    l = i * 2 + 1;
}
return ans;
}

public static void swap(int i, int j) {
    int tmp = heap[i];
    heap[i] = heap[j];
    heap[j] = tmp;
}

}

```

文件: Code07_BestTimeBuySellStockII.cpp

```

// 买卖股票的最佳时机 II
// 给你一个整数数组 prices，其中 prices[i] 表示某支股票第 i 天的价格
// 在每一天，你可以决定是否购买和/或出售股票。你在任何时候 最多 只能持有 一股 股票
// 你也可以先购买，然后在 同一天 出售
// 返回 你能获得的最大利润
// 测试链接：https://leetcode.cn/problems/best-time-to-buy-and-sell-stock-ii/

```

```

#include <iostream>
#include <vector>
#include <string>
#include <sstream>

using namespace std;

/***
 * 使用贪心算法解决股票买卖问题
 *

```

* 解题思路：

* 贪心算法 + 累加正收益

* 1. 将问题转化为每天的收益，只要收益为正就累加

* 2. 只要明天价格比今天高，就在今天买入明天卖出

* 3. 累加所有正的收益差值

*

* 时间复杂度分析：

* $O(n)$ – 只需要遍历一次数组

*

* 空间复杂度分析：

* $O(1)$ – 只使用了常数级别的额外空间

*

* 是否为最优解：

* 是，这是解决该问题的最优解

*

* 工程化考量：

* 1. 边界条件处理：空数组、单个元素数组等特殊情况

* 2. 输入验证：检查输入是否为有效数组

* 3. 异常处理：对非法输入进行检查

* 4. 可读性：添加详细注释和变量命名

*

* 跨语言特性：

* C++版本使用了 vector 容器，与 Java 的 ArrayList 和 Python 的 list 类似

* 但 C++提供了更多的内存控制能力

*

* 调试技巧：

* 1. 对于错误情况，可以使用 cout 打印中间变量值

* 2. 关键循环中可以添加断点观察 maxProfit 的变化

*

* 与标准库对比：

* 标准库中的算法通常经过高度优化，但对于这类简单问题，自定义实现效率相当

*/

```
int maxProfit(vector<int>& prices) {
    // 输入验证：处理边界情况
    if (prices.empty() || prices.size() == 1) {
        return 0; // 没有交易日或只有一天，无法获得利润
    }
```

```
    int maxProfit = 0; // 累计最大利润
```

```
    // 遍历数组，累加所有正的收益差值
```

```
    for (int i = 1; i < prices.size(); i++) {
```

```
        // 贪心策略：只要今天价格比昨天高，就累加差值作为利润
```

```

        if (prices[i] > prices[i - 1]) {
            maxProfit += prices[i] - prices[i - 1];
        }
    }

    return maxProfit;
}

/***
 * 打印数组的辅助函数
 */
string printVector(const vector<int>& v) {
    stringstream ss;
    ss << "[";
    for (size_t i = 0; i < v.size(); i++) {
        ss << v[i];
        if (i < v.size() - 1) {
            ss << ", ";
        }
    }
    ss << "]";
    return ss.str();
}

/***
 * 测试函数：测试各种边界条件和典型用例
 */
void test_maxProfit() {
    // 测试用例 1: [7, 1, 5, 3, 6, 4]
    vector<int> prices1 = {7, 1, 5, 3, 6, 4};
    int result1 = maxProfit(prices1);
    cout << "输入: " << printVector(prices1) << endl;
    cout << "输出: " << result1 << endl;
    cout << "预期: 7" << endl;
    cout << endl;

    // 测试用例 2: [1, 2, 3, 4, 5] - 单调递增数组
    vector<int> prices2 = {1, 2, 3, 4, 5};
    int result2 = maxProfit(prices2);
    cout << "输入: " << printVector(prices2) << endl;
    cout << "输出: " << result2 << endl;
    cout << "预期: 4" << endl;
    cout << endl;
}

```

```
// 测试用例 3: [7, 6, 4, 3, 1] - 单调递减数组
vector<int> prices3 = {7, 6, 4, 3, 1};
int result3 = maxProfit(prices3);
cout << "输入: " << printVector(prices3) << endl;
cout << "输出: " << result3 << endl;
cout << "预期: 0" << endl;
cout << endl;

// 测试用例 4: 空数组
vector<int> prices4 = {};
int result4 = maxProfit(prices4);
cout << "输入: []" << endl;
cout << "输出: " << result4 << endl;
cout << "预期: 0" << endl;
cout << endl;

// 测试用例 5: 单个元素
vector<int> prices5 = {1};
int result5 = maxProfit(prices5);
cout << "输入: [1]" << endl;
cout << "输出: " << result5 << endl;
cout << "预期: 0" << endl;
cout << endl;

// 测试用例 6: 极端大数组 (模拟, 使用小规模数组表示)
vector<int> prices6 = {1, 3, 2, 8, 4, 9};
int result6 = maxProfit(prices6);
cout << "输入: " << printVector(prices6) << endl;
cout << "输出: " << result6 << endl;
cout << "预期: 13" << endl;
cout << endl;
}

/**
 * 主函数: 运行测试
 */
int main() {
    cout << "买卖股票的最佳时机 II - 贪心算法解决方案" << endl;
    cout << "===== " << endl;
    test_maxProfit();
    return 0;
}
```

=====

文件: Code07_BestTimeBuySellStockII.java

=====

```
package class089;

import java.util.Arrays;

/**
 * 买卖股票的最佳时机 II - 贪心算法解决方案
 * <p>
 * 题目描述:
 * 给你一个整数数组 prices，其中 prices[i] 表示某支股票第 i 天的价格
 * 在每一天，你可以决定是否购买和/或出售股票。你在任何时候 最多 只能持有 一股 股票
 * 你也可以先购买，然后在 同一天 出售
 * 返回 你能获得的最大利润
 * <p>
 * 测试链接: https://leetcode.cn/problems/best-time-to-buy-and-sell-stock-ii/
 * <p>
 * 算法思想:
 * 使用贪心算法，将问题转化为每天的收益，只要收益为正就累加
 * 只要明天价格比今天高，就在今天买入明天卖出
 * 累加所有正的收益差值
 */
public class Code07_BestTimeBuySellStockII {

    /**
     * 计算买卖股票能获得的最大利润
     *
     * @param prices 股票每天的价格数组
     * @return 能获得的最大利润
     *
     * 时间复杂度: O(n) - 只需要遍历一次数组
     * 空间复杂度: O(1) - 只使用了常数级别的额外空间
     *
     * 工程化考量:
     * 1. 边界条件处理: 处理 null 输入、空数组和单元素数组
     * 2. 输入验证: 检查输入参数的有效性
     * 3. 异常处理: 对非法输入进行检查
     * 4. 可读性: 使用清晰的变量命名和详细的注释
     *
     * 算法证明:
```

```

* 由于可以进行多次交易，且交易次数没有限制，我们可以将所有的上涨波段都捕获
* 任何一段上涨都可以分解为连续的每日上涨，因此贪心策略是正确的
*/
public static int maxProfit(int[] prices) {
    // 输入验证：处理边界情况
    if (prices == null || prices.length <= 1) {
        return 0; // 没有交易日或只有一天，无法获得利润
    }

    int maxProfit = 0; // 累计最大利润

    // 遍历数组，累加所有正的收益差值
    for (int i = 1; i < prices.length; i++) {
        // 贪心策略：只要今天价格比昨天高，就累加差值作为利润
        if (prices[i] > prices[i - 1]) {
            maxProfit += prices[i] - prices[i - 1];
        }
    }

    return maxProfit;
}

/***
 * 调试版本的 maxProfit 函数，打印中间过程
 *
 * @param prices 股票每天的价格数组
 * @return 能获得的最大利润
 *
 * 调试技巧：
 * 1. 打印中间变量值
 * 2. 显示每一步的决策过程
 * 3. 帮助理解算法执行流程
 */
public static int debugMaxProfit(int[] prices) {
    if (prices == null || prices.length <= 1) {
        return 0;
    }

    int maxProfit = 0;

    System.out.println("执行过程详情:");
    System.out.println("天\t价格\t操作\t收益变化\t累计利润");
    System.out.printf("%d\t%d\t%s\t%d\t%d\n", 0, prices[0], "买入", 0, 0);

```

```

for (int i = 1; i < prices.length; i++) {
    if (prices[i] > prices[i - 1]) {
        int profit = prices[i] - prices[i - 1];
        maxProfit += profit;
        System.out.printf("%d\t%d\t%s\t+%d\t\t%d\n",
                           i, prices[i], "卖出再买入", profit, maxProfit);
    } else {
        System.out.printf("%d\t%d\t%s\t%d\t\t%d\n",
                           i, prices[i], "持有", 0, maxProfit);
    }
}

return maxProfit;
}

/***
 * 打印数组的辅助方法
 *
 * @param array 要打印的数组
 * @return 格式化的数组字符串
 */
private static String printArray(int[] array) {
    if (array == null) {
        return "null";
    }
    return Arrays.toString(array);
}

/***
 * 测试函数：测试各种边界条件和典型用例
 */
public static void testMaxProfit() {
    // 测试用例 1: [7, 1, 5, 3, 6, 4]
    int[] prices1 = {7, 1, 5, 3, 6, 4};
    int result1 = maxProfit(prices1);
    System.out.println("输入: " + printArray(prices1));
    System.out.println("输出: " + result1);
    System.out.println("预期: 7");
    System.out.println();

    // 测试用例 2: [1, 2, 3, 4, 5] - 单调递增数组
    int[] prices2 = {1, 2, 3, 4, 5};
}

```

```
int result2 = maxProfit(prices2);
System.out.println("输入: " + printArray(prices2));
System.out.println("输出: " + result2);
System.out.println("预期: 4");
System.out.println();

// 测试用例 3: [7, 6, 4, 3, 1] - 单调递减数组
int[] prices3 = {7, 6, 4, 3, 1};
int result3 = maxProfit(prices3);
System.out.println("输入: " + printArray(prices3));
System.out.println("输出: " + result3);
System.out.println("预期: 0");
System.out.println();

// 测试用例 4: 空数组
int[] prices4 = {};
int result4 = maxProfit(prices4);
System.out.println("输入: " + printArray(prices4));
System.out.println("输出: " + result4);
System.out.println("预期: 0");
System.out.println();

// 测试用例 5: 单个元素
int[] prices5 = {1};
int result5 = maxProfit(prices5);
System.out.println("输入: " + printArray(prices5));
System.out.println("输出: " + result5);
System.out.println("预期: 0");
System.out.println();

// 测试用例 6: 复杂波动
int[] prices6 = {1, 3, 2, 8, 4, 9};
int result6 = maxProfit(prices6);
System.out.println("输入: " + printArray(prices6));
System.out.println("输出: " + result6);
System.out.println("预期: 13");
System.out.println();

// 测试用例 7: 多次波动
int[] prices7 = {3, 3, 5, 0, 0, 3, 1, 4};
int result7 = maxProfit(prices7);
System.out.println("输入: " + printArray(prices7));
System.out.println("输出: " + result7);
```

```

        System.out.println("预期: 8");
        System.out.println();
    }

    /**
     * 主函数: 运行测试
     */
    public static void main(String[] args) {
        System.out.println("买卖股票的最佳时机 II - 贪心算法解决方案");
        System.out.println("=====");
        System.out.println("基础测试:");
        testMaxProfit();

        System.out.println("\n 调试模式示例:");
        int[] debugPrices = {7, 1, 5, 3, 6, 4};
        System.out.println("\n 对数组 " + printArray(debugPrices) + " 进行调试跟踪:");
        int finalProfit = debugMaxProfit(debugPrices);
        System.out.println("\n 最终利润: " + finalProfit);

        System.out.println("\n 算法分析:");
        System.out.println("- 时间复杂度: O(n) - 只需要遍历一次数组");
        System.out.println("- 空间复杂度: O(1) - 只使用常数级别额外空间");
        System.out.println("- 贪心策略: 只要今天比昨天贵, 就累加差值作为利润");
        System.out.println("- 最优性: 这种贪心策略能够得到全局最优解, 因为所有可能的上涨波段都被捕获");
    }
}

```

=====

文件: Code07_BestTimeBuySellStockII.py

=====

```

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

# 买卖股票的最佳时机 II
# 给你一个整数数组 prices , 其中 prices[i] 表示某支股票第 i 天的价格
# 在每一天, 你可以决定是否购买和/或出售股票。你在任何时候 最多 只能持有 一股 股票
# 你也可以先购买, 然后在 同一天 出售
# 返回 你能获得的最大利润
# 测试链接 : https://leetcode.cn/problems/best-time-to-buy-and-sell-stock-ii/

from typing import List

```

```
def max_profit(prices: List[int]) -> int:  
    """
```

使用贪心算法解决股票买卖问题

解题思路：

贪心算法 + 累加正收益

1. 将问题转化为每天的收益，只要收益为正就累加
2. 只要明天价格比今天高，就在今天买入明天卖出
3. 累加所有正的收益差值

参数：

prices: List[int] - 股票每天的价格数组

返回：

int - 能获得的最大利润

时间复杂度分析：

$O(n)$ - 只需要遍历一次数组

空间复杂度分析：

$O(1)$ - 只使用了常数级别的额外空间

是否为最优解：

是，这是解决该问题的最优解

异常处理：

处理了空数组和单元素数组的情况

Python 语言特性：

使用了类型注解提高代码可读性

利用 Python 简洁的语法实现贪心逻辑

```
"""
```

```
# 输入验证：处理边界情况
```

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

```
    return 0 # 没有交易日或只有一天，无法获得利润
```

```
max_profit_value = 0 # 累计最大利润
```

```
# 遍历数组，累加所有正的收益差值
```

```
for i in range(1, len(prices)):
```

```
    # 贪心策略：只要今天价格比昨天高，就累加差值作为利润
```

```
    if prices[i] > prices[i - 1]:  
        max_profit_value += prices[i] - prices[i - 1]  
  
    return max_profit_value
```

```
def test_max_profit():
```

```
    """
```

```
    测试函数：测试各种边界条件和典型用例
```

```
    测试用例覆盖：
```

1. 标准情况：既有上涨也有下跌
2. 单调递增数组：每天都上涨
3. 单调递减数组：每天都下跌
4. 空数组：边界情况
5. 单元素数组：边界情况
6. 其他情况：复杂波动

```
    """
```

```
# 测试用例 1: [7, 1, 5, 3, 6, 4]
```

```
prices1 = [7, 1, 5, 3, 6, 4]  
result1 = max_profit(prices1)  
print(f"输入: {prices1}")  
print(f"输出: {result1}")  
print(f"预期: 7")  
print()
```

```
# 测试用例 2: [1, 2, 3, 4, 5] - 单调递增数组
```

```
prices2 = [1, 2, 3, 4, 5]  
result2 = max_profit(prices2)  
print(f"输入: {prices2}")  
print(f"输出: {result2}")  
print(f"预期: 4")  
print()
```

```
# 测试用例 3: [7, 6, 4, 3, 1] - 单调递减数组
```

```
prices3 = [7, 6, 4, 3, 1]  
result3 = max_profit(prices3)  
print(f"输入: {prices3}")  
print(f"输出: {result3}")  
print(f"预期: 0")  
print()
```

```
# 测试用例 4: 空数组
```

```
prices4 = []
result4 = max_profit(prices4)
print(f"输入: {prices4}")
print(f"输出: {result4}")
print(f"预期: 0")
print()
```

测试用例 5: 单个元素

```
prices5 = [1]
result5 = max_profit(prices5)
print(f"输入: {prices5}")
print(f"输出: {result5}")
print(f"预期: 0")
print()
```

测试用例 6: 复杂波动

```
prices6 = [1, 3, 2, 8, 4, 9]
result6 = max_profit(prices6)
print(f"输入: {prices6}")
print(f"输出: {result6}")
print(f"预期: 13")
print()
```

测试用例 7: 多次波动

```
prices7 = [3, 3, 5, 0, 0, 3, 1, 4]
result7 = max_profit(prices7)
print(f"输入: {prices7}")
print(f"输出: {result7}")
print(f"预期: 8")
print()
```

```
def debug_max_profit(prices: List[int]) -> int:
    """
```

调试版本的 max_profit 函数, 打印中间过程

调试技巧:

1. 打印中间变量值
2. 显示每一步的决策过程
3. 帮助理解算法执行流程

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

```

max_profit_value = 0

print("执行过程详情:")
print("天\t价格\t操作\t收益变化\t累计利润")
print(f"0\t{prices[0]}\t买入\t0\t0")

for i in range(1, len(prices)):
    if prices[i] > prices[i - 1]:
        profit = prices[i] - prices[i - 1]
        max_profit_value += profit
        print(f"{i}\t{prices[i]}\t卖出再买入\t+{profit}\t\t{max_profit_value}")
    else:
        print(f"{i}\t{prices[i]}\t持有\t\t{max_profit_value}")

return max_profit_value

if __name__ == "__main__":
    print("买卖股票的最佳时机 II - 贪心算法解决方案")
    print("====")
    print("基础测试:")
    test_max_profit()

    print("\n调试模式示例:")
    debug_prices = [7, 1, 5, 3, 6, 4]
    print(f"\n对数组 {debug_prices} 进行调试跟踪:")
    final_profit = debug_max_profit(debug_prices)
    print(f"\n最终利润: {final_profit}")

    print("\n算法分析:")
    print("- 时间复杂度: O(n) - 只需要遍历一次数组")
    print("- 空间复杂度: O(1) - 只使用常数级别额外空间")
    print("- 贪心策略: 只要今天比昨天贵, 就累加差值作为利润")
    print("- 最优性: 这种贪心策略能够得到全局最优解, 因为所有可能的上涨波段都被捕获")

```

文件: Code08_AssignCookies.cpp

```

/**
 * 分发饼干 - 贪心算法 + 双指针解决方案 (C++实现, LeetCode 版本)
 *
```

- * 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
 - * 目标是尽可能满足越多多数的孩子，并输出这个最大数值
 - *
- * 测试链接: <https://leetcode.cn/problems/assign-cookies/>
- *
- * 算法思想:
 - * 贪心算法 + 双指针
 - * 1. 将孩子胃口值数组 g 和饼干尺寸数组 s 都按升序排列
 - * 2. 使用双指针分别指向当前考虑的孩子和饼干
 - * 3. 如果当前饼干能满足当前孩子，则两个指针都前移，满足孩子数加 1
 - * 4. 如果当前饼干不能满足当前孩子，则饼干指针前移，寻找更大的饼干
 - * 5. 直到其中一个数组遍历完为止
 - *
- * 时间复杂度分析:
 - * $O(m \log m + n \log n)$ – 其中 m 是孩子数量， n 是饼干数量
 - * - 对孩子胃口值数组排序需要 $O(m \log m)$
 - * - 对饼干尺寸数组排序需要 $O(n \log n)$
 - * - 双指针遍历需要 $O(m+n)$
- *
- * 空间复杂度分析:
 - * $O(1)$ – 只使用了常数级别的额外空间（不考虑排序所需的额外空间）
- *
- * 是否为最优解:
 - * 是，这是解决该问题的最优解
- *
- * 工程化考量:
 - * 1. 边界条件处理：处理空数组、单个元素数组等特殊情况
 - * 2. 输入验证：检查输入是否为有效数组
 - * 3. 异常处理：对非法输入进行检查
 - * 4. 可读性：添加详细注释和变量命名
- *
- * 贪心策略证明:
 - * 使用最小的饼干满足最小的孩子，可以最大化满足的孩子数量
 - * 这种策略满足贪心选择性质和最优子结构性质
- */

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

```
using namespace std;

class Solution {
public:
    /**
     * 计算最多能满足的孩子数量
     *
     * @param g 孩子胃口值数组
     * @param s 饼干尺寸数组
     * @return 最多能满足的孩子数量
     *
     * 算法步骤:
     * 1. 对孩子胃口值和饼干尺寸进行排序
     * 2. 使用双指针遍历两个数组
     * 3. 如果当前饼干能满足当前孩子，则满足孩子数加 1
     * 4. 返回最终满足的孩子数量
     */
    int findContentChildren(vector<int>& g, vector<int>& s) {
        // 输入验证
        if (g.empty() || s.empty()) {
            return 0;
        }

        // 对孩子胃口值数组和饼干尺寸数组都按升序排列
        sort(g.begin(), g.end());
        sort(s.begin(), s.end());

        int childIndex = 0;    // 孩子指针
        int cookieIndex = 0;   // 饼干指针
        int satisfiedChildren = 0; // 满足的孩子数

        // 双指针遍历
        while (childIndex < g.size() && cookieIndex < s.size()) {
            // 如果当前饼干能满足当前孩子
            if (s[cookieIndex] >= g[childIndex]) {
                satisfiedChildren++; // 满足孩子数加 1
                childIndex++;        // 孩子指针前移
            }
            // 无论是否满足，饼干指针都要前移
            cookieIndex++;
        }
    }
}
```

```
    return satisfiedChildren;
}

/***
 * 调试版本：打印计算过程中的中间结果
 *
 * @param g 孩子胃口值数组
 * @param s 饼干尺寸数组
 * @return 最多能满足的孩子数量
 */
int debugFindContentChildren(vector<int>& g, vector<int>& s) {
    if (g.empty() || s.empty()) {
        cout << "孩子或饼干数组为空，无法分配" << endl;
        return 0;
    }

    cout << "孩子胃口值数组: ";
    for (int appetite : g) {
        cout << appetite << " ";
    }
    cout << endl;

    cout << "饼干尺寸数组: ";
    for (int size : s) {
        cout << size << " ";
    }
    cout << endl;

    // 排序
    sort(g.begin(), g.end());
    sort(s.begin(), s.end());

    cout << "排序后孩子胃口值: ";
    for (int appetite : g) {
        cout << appetite << " ";
    }
    cout << endl;

    cout << "排序后饼干尺寸: ";
    for (int size : s) {
        cout << size << " ";
    }
    cout << endl;
```

```

int childIndex = 0;
int cookieIndex = 0;
int satisfiedChildren = 0;

cout << "\n 分配过程:" << endl;
while (childIndex < g.size() && cookieIndex < s.size()) {
    cout << "考虑孩子" << childIndex << "(胃口=" << g[childIndex]
        << ") 和饼干" << cookieIndex << "(尺寸=" << s[cookieIndex] << ")";

    if (s[cookieIndex] >= g[childIndex]) {
        satisfiedChildren++;
        cout << " -> 分配成功, 满足孩子数: " << satisfiedChildren << endl;
        childIndex++;
    } else {
        cout << " -> 饼干太小, 跳过此饼干" << endl;
    }
    cookieIndex++;
}

cout << "\n 最终结果: 最多能满足 " << satisfiedChildren << " 个孩子" << endl;
return satisfiedChildren;
}

};

/***
 * 测试函数: 验证分发饼干算法的正确性
 */
void testFindContentChildren() {
    Solution solution;

    cout << "分发饼干算法测试开始" << endl;
    cout << "======" << endl;

    // 测试用例 1: g = [1, 2, 3], s = [1, 1]
    vector<int> g1 = {1, 2, 3};
    vector<int> s1 = {1, 1};
    int result1 = solution.findContentChildren(g1, s1);
    cout << "输入: g = [1, 2, 3], s = [1, 1]" << endl;
    cout << "输出: " << result1 << endl;
    cout << "预期: 1" << endl;
    cout << (result1 == 1 ? "✓ 通过" : "✗ 失败") << endl;
    cout << endl;
}

```

```

// 测试用例 2: g = [1, 2], s = [1, 2, 3]
vector<int> g2 = {1, 2};
vector<int> s2 = {1, 2, 3};
int result2 = solution.findContentChildren(g2, s2);
cout << "输入: g = [1, 2], s = [1, 2, 3]" << endl;
cout << "输出: " << result2 << endl;
cout << "预期: 2" << endl;
cout << (result2 == 2 ? "✓ 通过" : "✗ 失败") << endl;
cout << endl;

// 测试用例 3: g = [1, 2, 7, 8, 9], s = [1, 3, 5, 9, 10]
vector<int> g3 = {1, 2, 7, 8, 9};
vector<int> s3 = {1, 3, 5, 9, 10};
int result3 = solution.findContentChildren(g3, s3);
cout << "输入: g = [1, 2, 7, 8, 9], s = [1, 3, 5, 9, 10]" << endl;
cout << "输出: " << result3 << endl;
cout << "预期: 4" << endl;
cout << (result3 == 4 ? "✓ 通过" : "✗ 失败") << endl;
cout << endl;

// 测试用例 4: 空孩子数组
vector<int> g4 = {};
vector<int> s4 = {1, 2, 3};
int result4 = solution.findContentChildren(g4, s4);
cout << "输入: g = [], s = [1, 2, 3]" << endl;
cout << "输出: " << result4 << endl;
cout << "预期: 0" << endl;
cout << (result4 == 0 ? "✓ 通过" : "✗ 失败") << endl;
cout << endl;

// 测试用例 5: 空饼干数组
vector<int> g5 = {1, 2, 3};
vector<int> s5 = {};
int result5 = solution.findContentChildren(g5, s5);
cout << "输入: g = [1, 2, 3], s = []" << endl;
cout << "输出: " << result5 << endl;
cout << "预期: 0" << endl;
cout << (result5 == 0 ? "✓ 通过" : "✗ 失败") << endl;
cout << endl;
}

/**
```

```

* 性能测试：测试算法在大规模数据下的表现
*/
void performanceTest() {
    Solution solution;

    cout << "性能测试开始" << endl;
    cout << "======" << endl;

    // 生成大规模测试数据
    int n = 10000; // 孩子数量
    int m = 15000; // 饼干数量

    vector<int> g(n);
    vector<int> s(m);

    // 生成随机胃口值和饼干尺寸
    for (int i = 0; i < n; i++) {
        g[i] = rand() % 1000 + 1; // 胃口值在 1-1000 之间
    }
    for (int i = 0; i < m; i++) {
        s[i] = rand() % 1000 + 1; // 饼干尺寸在 1-1000 之间
    }

    long startTime = clock();
    int result = solution.findContentChildren(g, s);
    long endTime = clock();

    cout << "数据规模：" << n << "个孩子，" << m << "块饼干" << endl;
    cout << "执行时间：" << (endTime - startTime) * 1000.0 / CLOCKS_PER_SEC << "毫秒" << endl;
    cout << "满足孩子数：" << result << endl;
    cout << "性能测试结束" << endl;
}

/**
 * 主函数：运行测试
*/
int main() {
    cout << "分发饼干 - 贪心算法 + 双指针解决方案" << endl;
    cout << "======" << endl;

    // 运行基础测试
    testFindContentChildren();
}

```

```

cout << "\n 调试模式示例:" << endl;
Solution solution;
vector<int> debugG = {1, 2, 3};
vector<int> debugS = {1, 1};
cout << "对测试用例 g = [1,2,3], s = [1,1] 进行调试跟踪:" << endl;
int debugResult = solution.debugFindContentChildren(debugG, debugS);
cout << "最终结果: " << debugResult << endl;

cout << "\n 算法分析:" << endl;
cout << "- 时间复杂度: O(m*logm + n*logn) - 排序和双指针遍历" << endl;
cout << "- 空间复杂度: O(1) - 只使用常数级别额外空间" << endl;
cout << "- 贪心策略: 使用最小的饼干满足最小的孩子" << endl;
cout << "- 最优性: 这种贪心策略能够得到全局最优解" << endl;
cout << "- 证明: 反证法可以证明这是最优分配策略" << endl;

// 可选: 运行性能测试
// cout << "\n 性能测试:" << endl;
// performanceTest();

return 0;
}

```

=====

文件: Code08_AssignCookies.java

=====

```

package class089;

import java.util.Arrays;

// 分发饼干
// 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
// 目标是尽可能满足越多数量的孩子，并输出这个最大数值
// 测试链接 : https://leetcode.cn/problems/assign-cookies/
public class Code08_AssignCookies {

    /**
     * 使用贪心算法解决分发饼干问题
     *
     * 解题思路:
     * 贪心算法 + 双指针
     */
}
```

- * 1. 将孩子胃口值数组 g 和饼干尺寸数组 s 都按升序排列
- * 2. 使用双指针分别指向当前考虑的孩子和饼干
- * 3. 如果当前饼干能满足当前孩子，则两个指针都前移，满足孩子数加 1
- * 4. 如果当前饼干不能满足当前孩子，则饼干指针前移，寻找更大的饼干
- * 5. 直到其中一个数组遍历完为止

*

* 时间复杂度分析：

- * $O(m \log m + n \log n)$ - 其中 m 是孩子数量，n 是饼干数量
- * - 对孩子胃口值数组排序需要 $O(m \log m)$
- * - 对饼干尺寸数组排序需要 $O(n \log n)$
- * - 双指针遍历需要 $O(m+n)$

*

* 空间复杂度分析：

- * $O(1)$ - 只使用了常数级别的额外空间（不考虑排序所需的额外空间）

*

* 是否为最优解：

- * 是，这是解决该问题的最优解

*

* 工程化考量：

- * 1. 边界条件处理：空数组、单个元素数组等特殊情况
- * 2. 输入验证：检查输入是否为有效数组
- * 3. 异常处理：对非法输入进行检查
- * 4. 可读性：添加详细注释和变量命名

*/

```
public static int findContentChildren(int[] g, int[] s) {
```

// 输入验证

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

// 将孩子胃口值数组和饼干尺寸数组都按升序排列

```
Arrays.sort(g);
Arrays.sort(s);
```

```
int childIndex = 0; // 孩子指针
```

```
int cookieIndex = 0; // 饼干指针
```

```
int satisfiedChildren = 0; // 满足的孩子数
```

// 双指针遍历

```
while (childIndex < g.length && cookieIndex < s.length) {
    // 如果当前饼干能满足当前孩子
    if (s[cookieIndex] >= g[childIndex]) {
        satisfiedChildren++; // 满足孩子数加 1
    }
}
```

```
        childIndex++;           // 孩子指针前移
    }
    // 无论是否满足，饼干指针都要前移
    cookieIndex++;
}

return satisfiedChildren;
}

// 测试函数
public static void main(String[] args) {
    // 测试用例 1: g = [1, 2, 3], s = [1, 1]
    int[] g1 = {1, 2, 3};
    int[] s1 = {1, 1};
    int result1 = findContentChildren(g1, s1);
    System.out.println("输入: g = [1, 2, 3], s = [1, 1]");
    System.out.println("输出: " + result1);
    System.out.println("预期: 1");
    System.out.println();

    // 测试用例 2: g = [1, 2], s = [1, 2, 3]
    int[] g2 = {1, 2};
    int[] s2 = {1, 2, 3};
    int result2 = findContentChildren(g2, s2);
    System.out.println("输入: g = [1, 2], s = [1, 2, 3]");
    System.out.println("输出: " + result2);
    System.out.println("预期: 2");
    System.out.println();

    // 测试用例 3: g = [1, 2, 7, 8, 9], s = [1, 3, 5, 9, 10]
    int[] g3 = {1, 2, 7, 8, 9};
    int[] s3 = {1, 3, 5, 9, 10};
    int result3 = findContentChildren(g3, s3);
    System.out.println("输入: g = [1, 2, 7, 8, 9], s = [1, 3, 5, 9, 10]");
    System.out.println("输出: " + result3);
    System.out.println("预期: 4");
    System.out.println();

    // 测试用例 4: 空孩子数组
    int[] g4 = {};
    int[] s4 = {1, 2, 3};
    int result4 = findContentChildren(g4, s4);
    System.out.println("输入: g = [], s = [1, 2, 3]");
}
```

```

        System.out.println("输出: " + result4);
        System.out.println("预期: 0");
        System.out.println();

        // 测试用例 5: 空饼干数组
        int[] g5 = {1, 2, 3};
        int[] s5 = {};
        int result5 = findContentChildren(g5, s5);
        System.out.println("输入: g = [1,2,3], s = []");
        System.out.println("输出: " + result5);
        System.out.println("预期: 0");
        System.out.println();
    }
}

```

=====

文件: Code08_AssignCookies.py

=====

"""

分发饼干 - 贪心算法 + 双指针解决方案 (Python 实现, LeetCode 版本)

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
目标是尽可能满足越多数量的孩子，并输出这个最大数值

测试链接: <https://leetcode.cn/problems/assign-cookies/>

算法思想:

贪心算法 + 双指针

1. 将孩子胃口值数组 g 和饼干尺寸数组 s 都按升序排列
2. 使用双指针分别指向当前考虑的孩子和饼干
3. 如果当前饼干能满足当前孩子，则两个指针都前移，满足孩子数加 1
4. 如果当前饼干不能满足当前孩子，则饼干指针前移，寻找更大的饼干
5. 直到其中一个数组遍历完为止

时间复杂度分析:

$O(m \log m + n \log n)$ - 其中 m 是孩子数量， n 是饼干数量

- 对孩子胃口值数组排序需要 $O(m \log m)$
- 对饼干尺寸数组排序需要 $O(n \log n)$
- 双指针遍历需要 $O(m+n)$

空间复杂度分析:

$O(1)$ - 只使用了常数级别的额外空间（不考虑排序所需的额外空间）

是否为最优解:

是，这是解决该问题的最优解

工程化考量:

1. 边界条件处理：处理空数组、单个元素数组等特殊情况
2. 输入验证：检查输入是否为有效数组
3. 异常处理：对非法输入进行检查
4. 可读性：添加详细注释和变量命名

贪心策略证明:

使用最小的饼干满足最小的孩子，可以最大化满足的孩子数量

这种策略满足贪心选择性质和最优子结构性质

"""

```
from typing import List
import time
import random

class Solution:
    """
    分发饼干问题的解决方案类
    """

    def findContentChildren(self, g: List[int], s: List[int]) -> int:
        """
        计算最多能满足的孩子数量
        """

Args:
```

g: 孩子胃口值数组

s: 饼干尺寸数组

Returns:

int: 最多能满足的孩子数量

Raises:

TypeError: 如果输入不是列表类型

ValueError: 如果数组元素不是正整数

算法步骤:

```
1. 对孩子胃口值和饼干尺寸进行排序
2. 使用双指针遍历两个数组
3. 如果当前饼干能满足当前孩子，则满足孩子数加 1
4. 返回最终满足的孩子数量
"""

# 输入验证
if not isinstance(g, list) or not isinstance(s, list):
    raise TypeError("输入必须是列表类型")

if len(g) == 0 or len(s) == 0:
    return 0

# 验证数组元素是否为正整数
for appetite in g:
    if not isinstance(appetite, int) or appetite <= 0:
        raise ValueError("孩子胃口值必须是正整数")

for size in s:
    if not isinstance(size, int) or size <= 0:
        raise ValueError("饼干尺寸必须是正整数")

# 对孩子胃口值数组和饼干尺寸数组都按升序排列
g.sort()
s.sort()

child_index = 0    # 孩子指针
cookie_index = 0  # 饼干指针
satisfied_children = 0 # 满足的孩子数

# 双指针遍历
while child_index < len(g) and cookie_index < len(s):
    # 如果当前饼干能满足当前孩子
    if s[cookie_index] >= g[child_index]:
        satisfied_children += 1 # 满足孩子数加 1
        child_index += 1         # 孩子指针前移

    # 无论是否满足，饼干指针都要前移
    cookie_index += 1

return satisfied_children

def debug_findContentChildren(self, g: List[int], s: List[int]) -> int:
    """
```

调试版本：打印计算过程中的中间结果

Args:

g: 孩子胃口值数组
s: 饼干尺寸数组

Returns:

int: 最多能满足的孩子数量

```
"""
if len(g) == 0 or len(s) == 0:
    print("孩子或饼干数组为空，无法分配")
    return 0

print("孩子胃口值数组:", g)
print("饼干尺寸数组:", s)

# 排序
g_sorted = sorted(g)
s_sorted = sorted(s)

print("排序后孩子胃口值:", g_sorted)
print("排序后饼干尺寸:", s_sorted)

child_index = 0
cookie_index = 0
satisfied_children = 0

print("\n分配过程:")
while child_index < len(g_sorted) and cookie_index < len(s_sorted):
    print(f"考虑孩子{child_index} (胃口={g_sorted[child_index]}) "
          f"和饼干{cookie_index} (尺寸={s_sorted[cookie_index]})", end="")
    if s_sorted[cookie_index] >= g_sorted[child_index]:
        satisfied_children += 1
        print(f" -> 分配成功，满足孩子数: {satisfied_children}")
        child_index += 1
    else:
        print(" -> 饼干太小，跳过此饼干")

    cookie_index += 1

print(f"\n最终结果: 最多能满足 {satisfied_children} 个孩子")
return satisfied_children
```

```
def test_findContentChildren():
    """
    测试函数：验证分发饼干算法的正确性
    """
    solution = Solution()

    print("分发饼干算法测试开始")
    print("=" * 50)

    # 测试用例 1: g = [1, 2, 3], s = [1, 1]
    g1 = [1, 2, 3]
    s1 = [1, 1]
    result1 = solution.findContentChildren(g1, s1)
    print("输入: g = [1, 2, 3], s = [1, 1]")
    print("输出:", result1)
    print("预期: 1")
    print("✓ 通过" if result1 == 1 else "✗ 失败")
    print()

    # 测试用例 2: g = [1, 2], s = [1, 2, 3]
    g2 = [1, 2]
    s2 = [1, 2, 3]
    result2 = solution.findContentChildren(g2, s2)
    print("输入: g = [1, 2], s = [1, 2, 3]")
    print("输出:", result2)
    print("预期: 2")
    print("✓ 通过" if result2 == 2 else "✗ 失败")
    print()

    # 测试用例 3: g = [1, 2, 7, 8, 9], s = [1, 3, 5, 9, 10]
    g3 = [1, 2, 7, 8, 9]
    s3 = [1, 3, 5, 9, 10]
    result3 = solution.findContentChildren(g3, s3)
    print("输入: g = [1, 2, 7, 8, 9], s = [1, 3, 5, 9, 10]")
    print("输出:", result3)
    print("预期: 4")
    print("✓ 通过" if result3 == 4 else "✗ 失败")
    print()

    # 测试用例 4: 空孩子数组
    g4 = []
    s4 = [1, 2, 3]
```

```
result4 = solution.findContentChildren(g4, s4)
print("输入: g = [], s = [1, 2, 3]")
print("输出:", result4)
print("预期: 0")
print("✓ 通过" if result4 == 0 else "✗ 失败")
print()

# 测试用例 5: 空饼干数组
g5 = [1, 2, 3]
s5 = []
result5 = solution.findContentChildren(g5, s5)
print("输入: g = [1, 2, 3], s = []")
print("输出:", result5)
print("预期: 0")
print("✓ 通过" if result5 == 0 else "✗ 失败")
print()

def performance_test():
    """
    性能测试: 测试算法在大规模数据下的表现
    """
    solution = Solution()

    print("性能测试开始")
    print("=" * 30)

    # 生成大规模测试数据
    n = 10000  # 孩子数量
    m = 15000  # 饼干数量

    g = [random.randint(1, 1000) for _ in range(n)]
    s = [random.randint(1, 1000) for _ in range(m)]

    start_time = time.time()
    result = solution.findContentChildren(g, s)
    end_time = time.time()

    print(f"数据规模: {n} 个孩子, {m} 块饼干")
    print(f"执行时间: {(end_time - start_time) * 1000:.2f} 毫秒")
    print(f"满足孩子数: {result}")
    print("性能测试结束")

if __name__ == "__main__":
```

```
print("分发饼干 - 贪心算法 + 双指针解决方案")
print("=" * 50)

# 运行基础测试
test_findContentChildren()

print("\n 调试模式示例:")
solution = Solution()
debug_g = [1, 2, 3]
debug_s = [1, 1]
print("对测试用例 g = [1,2,3], s = [1,1] 进行调试跟踪:")
debug_result = solution.debug_findContentChildren(debug_g, debug_s)
print("最终结果:", debug_result)

print("\n 算法分析:")
print("- 时间复杂度: O(m*logm + n*logn) - 排序和双指针遍历")
print("- 空间复杂度: O(1) - 只使用常数级别额外空间")
print("- 贪心策略: 使用最小的饼干满足最小的孩子")
print("- 最优性: 这种贪心策略能够得到全局最优解")
print("- 证明: 反证法可以证明这是最优分配策略")

# 可选: 运行性能测试
# print("\n 性能测试:")
# performance_test()

# 测试异常处理
print("\n 异常处理测试:")
try:
    # 创建一个无效的输入来测试类型检查
    invalid_g = "invalid" # 字符串而不是列表
    solution.findContentChildren(invalid_g, [1, 2])
except TypeError as e:
    print(f"类型错误测试通过: {e}")

try:
    # 测试包含 0 的无效胃口值
    solution.findContentChildren([1, 0], [1, 2])
except ValueError as e:
    print(f"数值错误测试通过: {e}")

=====
```

```
=====
/**  
 * 柠檬水找零 - 贪心算法 + 计数策略解决方案 (C++实现, LeetCode 版本)  
 *  
 * 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  
 *  
 * 测试链接: https://leetcode.cn/problems/lemonade-change/  
 *  
 * 算法思想:  
 * 贪心算法 + 计数策略  
 * 1. 维护 5 美元和 10 美元纸币的数量  
 * 2. 遇到 5 美元: 直接收下, 5 美元数量加 1  
 * 3. 遇到 10 美元: 需要找零 5 美元, 检查是否有 5 美元纸币, 如果有则 5 美元数量减 1, 10 美元数量加 1, 否则返回 false  
 * 4. 遇到 20 美元: 需要找零 15 美元, 优先使用一张 10 美元和一张 5 美元的组合, 如果没有 10 美元则使用三张 5 美元, 如果都不满足则返回 false  
 *  
 * 贪心策略解释:  
 * 在找零 15 美元时, 优先使用 10+5 的组合而不是 5+5+5, 因为 5 美元更万能, 可以用于 10 美元和 20 美元的找零,  
 * 而 10 美元只能用于 20 美元的找零, 所以要尽可能保留 5 美元纸币  
 *  
 * 时间复杂度分析:  
 * O(n) - 其中 n 是数组 bills 的长度, 只需要遍历一次数组  
 *  
 * 空间复杂度分析:  
 * O(1) - 只使用了常数级别的额外空间  
 *  
 * 是否为最优解:  
 * 是, 这是解决该问题的最优解  
 *  
 * 工程化考量:  
 * 1. 边界条件处理: 空数组等特殊情况  
 * 2. 输入验证: 检查输入是否为有效数组  
 * 3. 异常处理: 对非法输入进行检查  
 * 4. 可读性: 添加详细注释和变量命名
```

```
*/\n\n#include <iostream>\n#include <vector>\n#include <stdexcept>\n\nusing namespace std;\n\n\nclass Solution {\npublic:\n    /**\n     * 判断是否能正确找零\n     *\n     * @param bills 顾客付款数组\n     * @return 是否能正确找零\n     *\n     * 算法步骤:\n     * 1. 维护 5 美元和 10 美元纸币的数量\n     * 2. 遍历付款数组, 根据面额进行相应处理\n     * 3. 如果无法找零, 立即返回 false\n     * 4. 所有顾客都能正确找零则返回 true\n     */\n\n    bool lemonadeChange(vector<int>& bills) {\n        // 输入验证\n        if (bills.empty()) {\n            return true; // 空数组表示没有顾客, 可以正确找零\n        }\n\n        int fiveCount = 0; // 5 美元纸币数量\n        int tenCount = 0; // 10 美元纸币数量\n\n        // 遍历顾客付款数组\n        for (int bill : bills) {\n            // 验证付款面额\n            if (bill != 5 && bill != 10 && bill != 20) {\n                throw invalid_argument("付款面额必须是 5、10 或 20 美元");\n            }\n\n            if (bill == 5) {\n                // 顾客支付 5 美元, 无需找零, 直接收下\n                fiveCount++;\n            } else if (bill == 10) {\n                // 顾客支付 10 美元, 需要找零 5 美元\n                if (fiveCount > 0) {\n                    fiveCount--;\n                } else {\n                    return false;\n                }\n            }\n        }\n\n        return true;\n    }\n}
```

```
    if (fiveCount > 0) {
        fiveCount--; // 找零一张 5 美元
        tenCount++; // 收下一张 10 美元
    } else {
        // 没有 5 美元纸币找零，返回 false
        return false;
    }
} else if (bill == 20) {
    // 顾客支付 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 美元的组合
    if (tenCount > 0 && fiveCount > 0) {
        tenCount--; // 找零一张 10 美元
        fiveCount--; // 找零一张 5 美元
    }
    // 如果没有 10 美元，则尝试使用三张 5 美元
    else if (fiveCount >= 3) {
        fiveCount -= 3; // 找零三张 5 美元
    }
    // 如果两种方式都不满足，则无法正确找零
    else {
        return false;
    }
}
}

// 所有顾客都能正确找零
return true;
}

/***
 * 调试版本：打印计算过程中的中间结果
 *
 * @param bills 顾客付款数组
 * @return 是否能正确找零
 */
bool debugLemonadeChange(vector<int>& bills) {
    if (bills.empty()) {
        cout << "没有顾客，可以正确找零" << endl;
        return true;
    }

    cout << "顾客付款序列：" << endl;
    for (int i = 0; i < bills.size(); i++) {
```

```
cout << "第" << (i+1) << "位顾客支付: " << bills[i] << "美元" << endl;
}

int fiveCount = 0;
int tenCount = 0;

cout << "\n 找零过程:" << endl;
for (int i = 0; i < bills.size(); i++) {
    int bill = bills[i];
    cout << "第" << (i+1) << "位顾客支付" << bill << "美元: ";

    if (bill == 5) {
        fiveCount++;
        cout << "收下 5 美元, 无需找零。当前 5 美元数量: " << fiveCount
            << ", 10 美元数量: " << tenCount << endl;
    } else if (bill == 10) {
        if (fiveCount > 0) {
            fiveCount--;
            tenCount++;
            cout << "找零 5 美元, 收下 10 美元。当前 5 美元数量: " << fiveCount
                << ", 10 美元数量: " << tenCount << endl;
        } else {
            cout << "无法找零 5 美元, 交易失败" << endl;
            return false;
        }
    } else if (bill == 20) {
        if (tenCount > 0 && fiveCount > 0) {
            tenCount--;
            fiveCount--;
            cout << "找零 10+5 美元组合。当前 5 美元数量: " << fiveCount
                << ", 10 美元数量: " << tenCount << endl;
        } else if (fiveCount >= 3) {
            fiveCount -= 3;
            cout << "找零 5+5+5 美元组合。当前 5 美元数量: " << fiveCount
                << ", 10 美元数量: " << tenCount << endl;
        } else {
            cout << "无法找零 15 美元, 交易失败" << endl;
            return false;
        }
    }
}

cout << "\n 所有顾客都能正确找零" << endl;
```

```
    return true;
}
};

/***
 * 测试函数：验证柠檬水找零算法的正确性
 */
void testLemonadeChange() {
    Solution solution;

    cout << "柠檬水找零算法测试开始" << endl;
    cout << "======" << endl;

    // 测试用例 1: [5, 5, 5, 10, 20]
    vector<int> bills1 = {5, 5, 5, 10, 20};
    bool result1 = solution.lemonadeChange(bills1);
    cout << "输入: [5, 5, 5, 10, 20]" << endl;
    cout << "输出: " << (result1 ? "true" : "false") << endl;
    cout << "预期: true" << endl;
    cout << (result1 ? "✓ 通过" : "✗ 失败") << endl;
    cout << endl;

    // 测试用例 2: [5, 5, 10, 10, 20]
    vector<int> bills2 = {5, 5, 10, 10, 20};
    bool result2 = solution.lemonadeChange(bills2);
    cout << "输入: [5, 5, 10, 10, 20]" << endl;
    cout << "输出: " << (result2 ? "true" : "false") << endl;
    cout << "预期: false" << endl;
    cout << (!result2 ? "✓ 通过" : "✗ 失败") << endl;
    cout << endl;

    // 测试用例 3: [10, 10]
    vector<int> bills3 = {10, 10};
    bool result3 = solution.lemonadeChange(bills3);
    cout << "输入: [10, 10]" << endl;
    cout << "输出: " << (result3 ? "true" : "false") << endl;
    cout << "预期: false" << endl;
    cout << (!result3 ? "✓ 通过" : "✗ 失败") << endl;
    cout << endl;

    // 测试用例 4: [5, 5, 10]
    vector<int> bills4 = {5, 5, 10};
    bool result4 = solution.lemonadeChange(bills4);
```

```

cout << "输入: [5, 5, 10]" << endl;
cout << "输出: " << (result4 ? "true" : "false") << endl;
cout << "预期: true" << endl;
cout << (result4 ? "✓ 通过" : "X 失败") << endl;
cout << endl;

// 测试用例 5: 空数组
vector<int> bills5 = {};
bool result5 = solution.lemonadeChange(bills5);
cout << "输入: []" << endl;
cout << "输出: " << (result5 ? "true" : "false") << endl;
cout << "预期: true" << endl;
cout << (result5 ? "✓ 通过" : "X 失败") << endl;
cout << endl;

// 测试用例 6: 复杂情况
vector<int> bills6 = {5, 5, 5, 5, 5, 5, 5, 5, 5, 10, 20, 20, 20, 20, 20, 20};
bool result6 = solution.lemonadeChange(bills6);
cout << "输入: [5, 5, 5, 5, 5, 5, 5, 5, 5, 10, 20, 20, 20, 20, 20, 20]" << endl;
cout << "输出: " << (result6 ? "true" : "false") << endl;
cout << "预期: false" << endl;
cout << (!result6 ? "✓ 通过" : "X 失败") << endl;
cout << endl;
}

/**
 * 性能测试: 测试算法在大规模数据下的表现
 */
void performanceTest() {
    Solution solution;

    cout << "性能测试开始" << endl;
    cout << "======" << endl;

    // 生成大规模测试数据
    int n = 100000;
    vector<int> bills(n);

    // 生成随机付款序列 (5、10、20 美元)
    for (int i = 0; i < n; i++) {
        int randomValue = rand() % 3;
        if (randomValue == 0) bills[i] = 5;
        else if (randomValue == 1) bills[i] = 10;
        else bills[i] = 20;
    }
}

```

```

        else bills[i] = 20;
    }

long startTime = clock();
bool result = solution.lemonadeChange(bills);
long endTime = clock();

cout << "数据规模: " << n << " 位顾客" << endl;
cout << "执行时间: " << (endTime - startTime) * 1000.0 / CLOCKS_PER_SEC << " 毫秒" << endl;
cout << "找零结果: " << (result ? "成功" : "失败") << endl;
cout << "性能测试结束" << endl;
}

/***
 * 主函数: 运行测试
 */
int main() {
    cout << "柠檬水找零 - 贪心算法 + 计数策略解决方案" << endl;
    cout << "===== " << endl;

    // 运行基础测试
    testLemonadeChange();

    cout << "\n 调试模式示例:" << endl;
    Solution solution;
    vector<int> debugBills = {5, 5, 5, 10, 20};
    cout << "对测试用例 [5,5,5,10,20] 进行调试跟踪:" << endl;
    bool debugResult = solution.debugLemonadeChange(debugBills);
    cout << "最终结果: " << (debugResult ? "true" : "false") << endl;

    cout << "\n 算法分析:" << endl;
    cout << "- 时间复杂度: O(n) - 只需要遍历一次付款数组" << endl;
    cout << "- 空间复杂度: O(1) - 只使用常数级别额外空间" << endl;
    cout << "- 贪心策略: 优先使用 10+5 的组合找零, 保留更多 5 美元纸币" << endl;
    cout << "- 最优性: 这种贪心策略能够得到全局最优解" << endl;
    cout << "- 关键点: 5 美元纸币的通用性比 10 美元更强" << endl;

    // 可选: 运行性能测试
    // cout << "\n 性能测试:" << endl;
    // performanceTest();

    // 测试异常处理
    cout << "\n 异常处理测试:" << endl;
}

```

```

try {
    vector<int> invalidBills = {5, 15, 10}; // 包含非法面额 15
    solution.lemonadeChange(invalidBills);
} catch (const invalid_argument& e) {
    cout << "异常处理测试通过: " << e.what() << endl;
}

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

文件: Code09_LemonadeChange.java

```

package class089;

// 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/
public class Code09_LemonadeChange {

    /**
     * 使用贪心算法解决柠檬水找零问题
     *
     * 解题思路:
     * 贪心算法 + 计数策略
     * 1. 维护 5 美元和 10 美元纸币的数量
     * 2. 遇到 5 美元: 直接收下, 5 美元数量加 1
     * 3. 遇到 10 美元: 需要找零 5 美元, 检查是否有 5 美元纸币, 如果有则 5 美元数量减 1, 10 美元数量加 1, 否则返回 false
     * 4. 遇到 20 美元: 需要找零 15 美元, 优先使用一张 10 美元和一张 5 美元的组合, 如果没有 10 美元则使用三张 5 美元, 如果都不满足则返回 false
     *
     * 贪心策略解释:
     * 在找零 15 美元时, 优先使用 10+5 的组合而不是 5+5+5, 因为 5 美元更万能, 可以用于 10 美元和 20 美元的找零,
     * 而 10 美元只能用于 20 美元的找零, 所以要尽可能保留 5 美元纸币
}
```

```
*  
* 时间复杂度分析:  
* O(n) - 其中 n 是数组 bills 的长度, 只需要遍历一次数组  
*  
* 空间复杂度分析:  
* O(1) - 只使用了常数级别的额外空间  
*  
* 是否为最优解:  
* 是, 这是解决该问题的最优解  
*  
* 工程化考量:  
* 1. 边界条件处理: 空数组等特殊情况  
* 2. 输入验证: 检查输入是否为有效数组  
* 3. 异常处理: 对非法输入进行检查  
* 4. 可读性: 添加详细注释和变量命名  
*/  
  
public static boolean lemonadeChange(int[] bills) {  
    // 输入验证  
    if (bills == null) {  
        return true; // 空数组表示没有顾客, 可以正确找零  
    }  
  
    int fiveCount = 0; // 5 美元纸币数量  
    int tenCount = 0; // 10 美元纸币数量  
  
    // 遍历顾客付款数组  
    for (int bill : bills) {  
        if (bill == 5) {  
            // 顾客支付 5 美元, 无需找零, 直接收下  
            fiveCount++;  
        } else if (bill == 10) {  
            // 顾客支付 10 美元, 需要找零 5 美元  
            if (fiveCount > 0) {  
                fiveCount--; // 找零一张 5 美元  
                tenCount++; // 收下一张 10 美元  
            } else {  
                // 没有 5 美元纸币找零, 返回 false  
                return false;  
            }  
        } else if (bill == 20) {  
            // 顾客支付 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 美元的组合  
            if (tenCount > 0 && fiveCount > 0) {  
                tenCount--;  
                fiveCount--;  
            } else {  
                // 没有 10 和 5 美元的组合, 返回 false  
                return false;  
            }  
        }  
    }  
}
```

```
    tenCount--; // 找零一张 10 美元
    fiveCount--; // 找零一张 5 美元
}
// 如果没有 10 美元，则尝试使用三张 5 美元
else if (fiveCount >= 3) {
    fiveCount -= 3; // 找零三张 5 美元
}
// 如果两种方式都不满足，则无法正确找零
else {
    return false;
}
}

// 注意：题目保证输入只包含 5、10、20 三种面额，所以不需要处理其他情况
}

// 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: [5, 5, 5, 10, 20]
    int[] bills1 = {5, 5, 5, 10, 20};
    boolean result1 = lemonadeChange(bills1);
    System.out.println("输入: [5, 5, 5, 10, 20]");
    System.out.println("输出: " + result1);
    System.out.println("预期: true");
    System.out.println();

    // 测试用例 2: [5, 5, 10, 10, 20]
    int[] bills2 = {5, 5, 10, 10, 20};
    boolean result2 = lemonadeChange(bills2);
    System.out.println("输入: [5, 5, 10, 10, 20]");
    System.out.println("输出: " + result2);
    System.out.println("预期: false");
    System.out.println();

    // 测试用例 3: [10, 10]
    int[] bills3 = {10, 10};
    boolean result3 = lemonadeChange(bills3);
    System.out.println("输入: [10, 10]");
    System.out.println("输出: " + result3);
    System.out.println("预期: false");
}
```

```

System.out.println();

// 测试用例 4: [5, 5, 10]
int[] bills4 = {5, 5, 10};
boolean result4 = lemonadeChange(bills4);
System.out.println("输入: [5, 5, 10]");
System.out.println("输出: " + result4);
System.out.println("预期: true");
System.out.println();

// 测试用例 5: 空数组
int[] bills5 = {};
boolean result5 = lemonadeChange(bills5);
System.out.println("输入: []");
System.out.println("输出: " + result5);
System.out.println("预期: true");
System.out.println();

// 测试用例 6: [5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 10, 20, 20, 20, 20, 20, 20]
// 前 10 个 5 美元: five_count = 10
// 一个 10 美元: 找零 5 美元, five_count = 9, ten_count = 1
// 一个 20 美元: 优先用 10+5 找零, five_count = 8, ten_count = 0
// 剩下的 4 个 20 美元: 都需要找零 15 美元, 但只有 8 张 5 美元, 不够找零, 所以返回 false
int[] bills6 = {5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 10, 20, 20, 20, 20, 20, 20};
boolean result6 = lemonadeChange(bills6);
System.out.println("输入: [5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 10, 20, 20, 20, 20, 20, 20]");
System.out.println("输出: " + result6);
System.out.println("预期: false");
System.out.println();
}

}
=====

文件: Code09_LemonadeChange.py
=====
"""

柠檬水找零 - 贪心算法 + 计数策略解决方案 (Python 实现, LeetCode 版本)

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

```

文件: Code09_LemonadeChange.py

=====

柠檬水找零 - 贪心算法 + 计数策略解决方案 (Python 实现, LeetCode 版本)

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

测试链接: <https://leetcode.cn/problems/lemonade-change/>

算法思想:

贪心算法 + 计数策略

1. 维护 5 美元和 10 美元纸币的数量
2. 遇到 5 美元: 直接收下, 5 美元数量加 1
3. 遇到 10 美元: 需要找零 5 美元, 检查是否有 5 美元纸币, 如果有则 5 美元数量减 1, 10 美元数量加 1, 否则返回 false
4. 遇到 20 美元: 需要找零 15 美元, 优先使用一张 10 美元和一张 5 美元的组合, 如果没有 10 美元则使用三张 5 美元, 如果都不满足则返回 false

贪心策略解释:

在找零 15 美元时, 优先使用 10+5 的组合而不是 5+5+5, 因为 5 美元更万能, 可以用于 10 美元和 20 美元的找零,

而 10 美元只能用于 20 美元的找零, 所以要尽可能保留 5 美元纸币

时间复杂度分析:

$O(n)$ - 其中 n 是数组 bills 的长度, 只需要遍历一次数组

空间复杂度分析:

$O(1)$ - 只使用了常数级别的额外空间

是否为最优解:

是, 这是解决该问题的最优解

工程化考量:

1. 边界条件处理: 空数组等特殊情况
2. 输入验证: 检查输入是否为有效数组
3. 异常处理: 对非法输入进行检查
4. 可读性: 添加详细注释和变量命名

"""

```
from typing import List
```

```
import time
```

```
import random
```

```
class Solution:
```

```
    """
```

柠檬水找零问题的解决方案类

"""

```
def lemonadeChange(self, bills: List[int]) -> bool:
```

"""

判断是否能正确找零

Args:

bills: 顾客付款数组

Returns:

bool: 是否能正确找零

Raises:

TypeError: 如果输入不是列表类型

ValueError: 如果付款面额不是 5、10 或 20 美元

算法步骤:

1. 维护 5 美元和 10 美元纸币的数量
2. 遍历付款数组，根据面额进行相应处理
3. 如果无法找零，立即返回 false
4. 所有顾客都能正确找零则返回 true

"""

输入验证

```
if not isinstance(bills, list):
```

```
    raise TypeError("输入必须是列表类型")
```

```
if len(bills) == 0:
```

```
    return True # 空数组表示没有顾客，可以正确找零
```

```
five_count = 0 # 5 美元纸币数量
```

```
ten_count = 0 # 10 美元纸币数量
```

遍历顾客付款数组

```
for bill in bills:
```

验证付款面额

```
if bill not in [5, 10, 20]:
```

```
    raise ValueError("付款面额必须是 5、10 或 20 美元")
```

```
if bill == 5:
```

顾客支付 5 美元，无需找零，直接收下

```
    five_count += 1
```

```
elif bill == 10:
```

```
# 顾客支付 10 美元，需要找零 5 美元
if five_count > 0:
    five_count -= 1 # 找零一张 5 美元
    ten_count += 1 # 收下一张 10 美元
else:
    # 没有 5 美元纸币找零，返回 false
    return False

elif bill == 20:
    # 顾客支付 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 美元的组合
    if ten_count > 0 and five_count > 0:
        ten_count -= 1 # 找零一张 10 美元
        five_count -= 1 # 找零一张 5 美元
    # 如果没有 10 美元，则尝试使用三张 5 美元
    elif five_count >= 3:
        five_count -= 3 # 找零三张 5 美元
    # 如果两种方式都不满足，则无法正确找零
    else:
        return False

# 所有顾客都能正确找零
return True
```

```
def debug_lemonadeChange(self, bills: List[int]) -> bool:
    """
```

调试版本：打印计算过程中的中间结果

Args:

bills: 顾客付款数组

Returns:

bool: 是否能正确找零

"""

```
if len(bills) == 0:
    print("没有顾客，可以正确找零")
    return True
```

```
print("顾客付款序列:")
for i, bill in enumerate(bills):
    print(f"第{i+1}位顾客支付: {bill} 美元")
```

```
five_count = 0
ten_count = 0
```

```

print("\n 找零过程:")
for i, bill in enumerate(bills):
    print(f"第{i+1}位顾客支付{bill}美元: ", end="")

    if bill == 5:
        five_count += 1
        print(f"收下 5 美元, 无需找零。当前 5 美元数量: {five_count}, 10 美元数量:
{ten_count}")

    elif bill == 10:
        if five_count > 0:
            five_count -= 1
            ten_count += 1
            print(f"找零 5 美元, 收下 10 美元。当前 5 美元数量: {five_count}, 10 美元数量:
{ten_count}")

        else:
            print("无法找零 5 美元, 交易失败")
            return False

    elif bill == 20:
        if ten_count > 0 and five_count > 0:
            ten_count -= 1
            five_count -= 1
            print(f"找零 10+5 美元组合。当前 5 美元数量: {five_count}, 10 美元数量:
{ten_count}")

        elif five_count >= 3:
            five_count -= 3
            print(f"找零 5+5+5 美元组合。当前 5 美元数量: {five_count}, 10 美元数量:
{ten_count}")

        else:
            print("无法找零 15 美元, 交易失败")
            return False

print("\n 所有顾客都能正确找零")
return True

```

```

def test_lemonadeChange():
    """
    测试函数: 验证柠檬水找零算法的正确性
    """
    solution = Solution()

    print("柠檬水找零算法测试开始")
    print("=" * 50)

```

```
# 测试用例 1: [5, 5, 5, 10, 20]
bills1 = [5, 5, 5, 10, 20]
result1 = solution.lemonadeChange(bills1)
print("输入: [5, 5, 5, 10, 20]")
print("输出:", result1)
print("预期: True")
print("✓ 通过" if result1 else "✗ 失败")
print()

# 测试用例 2: [5, 5, 10, 10, 20]
bills2 = [5, 5, 10, 10, 20]
result2 = solution.lemonadeChange(bills2)
print("输入: [5, 5, 10, 10, 20]")
print("输出:", result2)
print("预期: False")
print("✓ 通过" if not result2 else "✗ 失败")
print()

# 测试用例 3: [10, 10]
bills3 = [10, 10]
result3 = solution.lemonadeChange(bills3)
print("输入: [10, 10]")
print("输出:", result3)
print("预期: False")
print("✓ 通过" if not result3 else "✗ 失败")
print()

# 测试用例 4: [5, 5, 10]
bills4 = [5, 5, 10]
result4 = solution.lemonadeChange(bills4)
print("输入: [5, 5, 10]")
print("输出:", result4)
print("预期: True")
print("✓ 通过" if result4 else "✗ 失败")
print()

# 测试用例 5: 空数组
bills5 = []
result5 = solution.lemonadeChange(bills5)
print("输入: []")
print("输出:", result5)
print("预期: True")
```

```
print("✓ 通过" if result5 else "✗ 失败")
print()

# 测试用例 6: 复杂情况
bills6 = [5, 5, 5, 5, 5, 5, 5, 5, 5, 10, 20, 20, 20, 20, 20]
result6 = solution.lemonadeChange(bills6)
print("输入: [5, 5, 5, 5, 5, 5, 5, 5, 5, 10, 20, 20, 20, 20, 20]")
print("输出:", result6)
print("预期: False")
print("✓ 通过" if not result6 else "✗ 失败")
print()

def performance_test():
    """
    性能测试: 测试算法在大规模数据下的表现
    """
    solution = Solution()

    print("性能测试开始")
    print("=" * 30)

    # 生成大规模测试数据
    n = 100000
    bills = []

    # 生成随机付款序列 (5、10、20 美元)
    for _ in range(n):
        random_value = random.randint(0, 2)
        if random_value == 0:
            bills.append(5)
        elif random_value == 1:
            bills.append(10)
        else:
            bills.append(20)

    start_time = time.time()
    result = solution.lemonadeChange(bills)
    end_time = time.time()

    print(f"数据规模: {n} 位顾客")
    print(f"执行时间: {(end_time - start_time) * 1000:.2f} 毫秒")
    print(f"找零结果: {'成功' if result else '失败'}")
    print("性能测试结束")
```

```
if __name__ == "__main__":
    print("柠檬水找零 - 贪心算法 + 计数策略解决方案")
    print("=" * 50)

# 运行基础测试
test_lemonadeChange()

print("\n 调试模式示例:")
solution = Solution()
debug_bills = [5, 5, 5, 10, 20]
print("对测试用例 [5, 5, 5, 10, 20] 进行调试跟踪:")
debug_result = solution.debug_lemonadeChange(debug_bills)
print("最终结果:", debug_result)

print("\n 算法分析:")
print("- 时间复杂度: O(n) - 只需要遍历一次付款数组")
print("- 空间复杂度: O(1) - 只使用常数级别额外空间")
print("- 贪心策略: 优先使用 10+5 的组合找零, 保留更多 5 美元纸币")
print("- 最优性: 这种贪心策略能够得到全局最优解")
print("- 关键点: 5 美元纸币的通用性比 10 美元更强")

# 可选: 运行性能测试
# print("\n 性能测试:")
# performance_test()

# 测试异常处理
print("\n 异常处理测试:")
try:
    # 创建一个无效的输入来测试类型检查
    invalid_bills = "invalid" # 字符串而不是列表
    solution.lemonadeChange(invalid_bills)
except TypeError as e:
    print(f"类型错误测试通过: {e}")

try:
    # 测试包含非法面额的付款序列
    solution.lemonadeChange([5, 15, 10]) # 包含非法面额 15
except ValueError as e:
    print(f"数值错误测试通过: {e}")

=====
```

文件: TestLargestNumber.java

```
=====
package class089;

public class TestLargestNumber {
    public static void main(String[] args) {
        // 测试 largestNumber 方法
        Code01_LargestNumber code = new Code01_LargestNumber();
        code.testLargestNumber();
    }
}
```
