==========
文件夹: class124_GreedyAlgorithms
==========

[Markdown 文件]
==========
文件: README.md
==========
贪心算法专题

本目录包含了一系列使用贪心算法解决的经典问题，涵盖了跳跃游戏、区间覆盖、字符串映射、过河问题和资源平衡分配等多个方面。

题目列表

1. 跳跃游戏 II (Jump Game II)
- **文件**: Code01_JumpGameII.java, Code01_JumpGameII.py, Code01_JumpGameII.cpp
- **题目来源**: LeetCode 45
- **题目链接**: https://leetcode.cn/problems/jump-game-ii/
- **问题描述**: 给定一个长度为 n 的整数数组 nums，初始在 0 下标，nums[i]表示可以从 i 下标往右跳的最大距离，返回到达 n-1 下标的最少跳跃次数。

2. 灌溉花园的最少水龙头数目 (Minimum Number of Taps to Open to Water a Garden)
- **文件**: Code02_MinimumTaps.java, Code02_MinimumTaps.py, Code02_MinimumTaps.cpp
- **题目来源**: LeetCode 1326
- **题目链接**: https://leetcode.cn/problems/minimum-number-of-taps-to-open-to-water-a-garden/
- **问题描述**: 在 x 轴上有一个一维的花园，花园长度为 n，从点 0 开始，到点 n 结束。花园里总共有 n+1 个水龙头，分别位于[0, 1, ... n]。给定整数 n 和长度为 n+1 的整数数组 ranges，其中 ranges[i]表示如果打开点 i 处的水龙头，可以灌溉的区域为[i-ranges[i], i+ranges[i]]，返回可以灌溉整个花园的最少水龙头数目。

3. 字符串转化 (String Transforms Into Another String)
- **文件**: Code03_StringTransforms.java, Code03_StringTransforms.py, Code03_StringTransforms.cpp
- **题目来源**: LeetCode 917
- **题目链接**: https://leetcode.cn/problems/string-transforms-into-another-string/
- **问题描述**: 给出两个长度相同的字符串 str1 和 str2，判断字符串 str1 能不能在零次或多次转化后变成字符串 str2。每一次转化时，可以将 str1 中出现的所有相同字母变成其他任何小写英文字母。

4. 过河问题 (Cross River)
- **文件**: Code04_CrossRiver.java, Code04_CrossRiver.py, Code04_CrossRiver.cpp
- **题目来源**: 洛谷 P1809
- **题目链接**: https://www.luogu.com.cn/problem/P1809
- **问题描述**: 一共 n 人出游，他们走到一条河的西岸，想要过河到东岸。每个人都有一个渡河时间 ti，西

岸有一条船，一次最多乘坐两人。如果船上有一个人，划到对岸的时间等于这个人的渡河时间；如果船上有两个人，划到对岸的时间等于两个人的渡河时间的最大值。返回最少要花费多少时间，才能使所有人都过河。

5. 超级洗衣机 (Super Washing Machines)
- **文件**: Code05_SuperWashingMachines.java, Code05_SuperWashingMachines.py, Code05_SuperWashingMachines.cpp
- **题目来源**: LeetCode 517
- **题目链接**: https://leetcode.cn/problems/super-washing-machines/
- **问题描述**: 假设有 n 台超级洗衣机放在同一排上，开始的时候，每台洗衣机内可能有一定量的衣服，也可能是空的。在每一步操作中，可以选择任意 m (1 <= m <= n)台洗衣机，与此同时将每台洗衣机的一件衣服送到相邻的一台洗衣机。给定一个整数数组 machines 代表从左至右每台洗衣机中的衣物数量，给出能让所有洗衣机中剩下的衣物的数量相等的最少的操作步数。

补充题目列表

6. 分发饼干 (Assign Cookies)
- **题目来源**: LeetCode 455
- **题目链接**: https://leetcode.cn/problems/assign-cookies/
- **问题描述**: 假设你是一位很棒的家长，想要给你的孩子们一些小饼干。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最小尺寸。分配饼干使最多孩子满足。

7. 柠檬水找零 (Lemonade Change)
- **题目来源**: LeetCode 860
- **题目链接**: https://leetcode.cn/problems/lemonade-change/
- **问题描述**: 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。

8. 跳跃游戏 (Jump Game)
- **题目来源**: LeetCode 55
- **题目链接**: https://leetcode.cn/problems/jump-game/
- **问题描述**: 给定一个非负整数数组，你最初位于数组的第一个位置。数组中的每个元素代表你在该位置可以跳跃的最大长度。判断你是否能够到达最后一个位置。

9. 加油站 (Gas Station)
- **题目来源**: LeetCode 134
- **题目链接**: https://leetcode.cn/problems/gas-station/

10. 分发糖果 (Candy)
- **题目来源**: LeetCode 135
- **题目链接**: https://leetcode.cn/problems/candy/

11. 无重叠区间 (Non-overlapping Intervals)
- **题目来源**: LeetCode 435

- **题目链接**: https://leetcode.cn/problems/non-overlapping-intervals/

12. 用最少数量的箭引爆气球 (Minimum Number of Arrows to Burst Balloons)
- **题目来源**: LeetCode 452
- **题目链接**: https://leetcode.cn/problems/minimum-number-of-arrows-to-burst-balloons/

13. 最大子数组和 (Maximum Subarray)
- **题目来源**: LeetCode 53
- **题目链接**: https://leetcode.cn/problems/maximum-subarray/

14. 买卖股票的最佳时机 II (Best Time to Buy and Sell Stock II)
- **题目来源**: LeetCode 122
- **题目链接**: https://leetcode.cn/problems/best-time-to-buy-and-sell-stock-ii/

15. 合并区间 (Merge Intervals)
- **题目来源**: LeetCode 56
- **题目链接**: https://leetcode.cn/problems/merge-intervals/

16. 根据身高重建队列 (Queue Reconstruction by Height)
- **题目来源**: LeetCode 406
- **题目链接**: https://leetcode.cn/problems/queue-reconstruction-by-height/

17. 最低加油次数 (Minimum Number of Refueling Stops)
- **题目来源**: LeetCode 871
- **题目链接**: https://leetcode.cn/problems/minimum-number-of-refueling-stops/

18. 最大数 (Largest Number)
- **题目来源**: LeetCode 179
- **题目链接**: https://leetcode.cn/problems/largest-number/

19. 摆动序列 (Wiggle Subsequence)
- **题目来源**: LeetCode 376
- **题目链接**: https://leetcode.cn/problems/wiggle-subsequence/

20. 单调递增的数字 (Monotone Increasing Digits)
- **题目来源**: LeetCode 738
- **题目链接**: https://leetcode.cn/problems/monotone-increasing-digits/

21. 划分字母区间 (Partition Labels)
- **题目来源**: LeetCode 763
- **题目链接**: https://leetcode.cn/problems/partition-labels/

22. 森林中的兔子 (Rabbits in Forest)

- **题目来源**: LeetCode 781
- **题目链接**: https://leetcode.cn/problems/rabbits-in-forest/

23. 合并果子 (Merge Fruits)
- **题目来源**: 洛谷 P1090
- **题目链接**: https://www.luogu.com.cn/problem/P1090

24. 排队接水 (Queue for Water)
- **题目来源**: 洛谷 P1223
- **题目链接**: https://www.luogu.com.cn/problem/P1223

25. 凌乱的yyy/线段覆盖 (Messy yyy/Segment Coverage)
- **题目来源**: 洛谷 P1803
- **题目链接**: https://www.luogu.com.cn/problem/P1803

26. 加油站 (Gas Station)
- **文件**: Code11_GasStation.java, Code11_GasStation.py, Code11_GasStation.cpp
- **题目来源**: LeetCode 134
- **题目链接**: https://leetcode.cn/problems/gas-station/
- **问题描述**: 在一条环路上有N个加油站，每个加油站有汽油gas[i]和消耗cost[i]。从某个加油站出发，按顺序访问每个加油站，判断是否能绕环路行驶一周。

27. 分发糖果 (Candy)
- **文件**: Code12_Candy.java, Code12_Candy.py, Code12_Candy.cpp
- **题目来源**: LeetCode 135
- **题目链接**: https://leetcode.cn/problems/candy/
- **问题描述**: 老师想给孩子们分发糖果，有N个孩子站成了一条直线，每个孩子至少分配到1个糖果。相邻的孩子中，评分高的孩子必须获得更多的糖果。

28. 无重叠区间 (Non-overlapping Intervals)
- **文件**: Code13_NonOverlappingIntervals.java, Code13_NonOverlappingIntervals.py, Code13_NonOverlappingIntervals.cpp
- **题目来源**: LeetCode 435
- **题目链接**: https://leetcode.cn/problems/non-overlapping-intervals/
- **问题描述**: 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。

29. 用最少数量的箭引爆气球 (Minimum Number of Arrows to Burst Balloons)
- **文件**: Code14_MinimumArrowsToBurstBalloons.java, Code14_MinimumArrowsToBurstBalloons.py, Code14_MinimumArrowsToBurstBalloons.cpp
- **题目来源**: LeetCode 452
- **题目链接**: https://leetcode.cn/problems/minimum-number-of-arrows-to-burst-balloons/
- **问题描述**: 在二维空间中有许多球形的气球，每个气球在水平方向上的直径范围是[xstart, xend]。用最少数量的箭引爆所有气球。

30. 最大子数组和 (Maximum Subarray)
- **文件**: Code15_MaximumSubarray.java, Code15_MaximumSubarray.py, Code15_MaximumSubarray.cpp
- **题目来源**: LeetCode 53
- **题目链接**: https://leetcode.cn/problems/maximum-subarray/
- **问题描述**: 给定一个整数数组 nums，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），返回其最大和。

31. 合并区间 (Merge Intervals)
- **文件**: Code16_MergeIntervals.java, Code16_MergeIntervals.py, Code16_MergeIntervals.cpp
- **题目来源**: LeetCode 56
- **题目链接**: https://leetcode.cn/problems/merge-intervals/
- **问题描述**: 以数组 intervals 表示若干个区间的集合，请合并所有重叠的区间，并返回一个不重叠的区间数组。

32. 根据身高重建队列 (Queue Reconstruction by Height)
- **文件**: Code17_QueueReconstructionByHeight.java, Code17_QueueReconstructionByHeight.py, Code17_QueueReconstructionByHeight.cpp
- **题目来源**: LeetCode 406
- **题目链接**: https://leetcode.cn/problems/queue-reconstruction-by-height/
- **问题描述**: 假设有打乱顺序的一群人站成一个队列，每个人由一个整数对(h, k)表示，其中 h 是这个人的身高，k 是排在这个人前面且身高大于或等于 h 的人数。

33. 最低加油次数 (Minimum Number of Refueling Stops)
- **文件**: Code18_MinimumRefuelingStops.java, Code18_MinimumRefuelingStops.py, Code18_MinimumRefuelingStops.cpp
- **题目来源**: LeetCode 871
- **题目链接**: https://leetcode.cn/problems/minimum-number-of-refueling-stops/
- **问题描述**: 汽车从起点出发驶向目的地，该目的地距离起点 target 英里。沿途有加油站，每个 station[i]代表一个加油站，位于距离起点 station[i][0]英里处，有 station[i][1]升汽油。

34. 最大数 (Largest Number)
- **文件**: Code19_LargestNumber.java, Code19_LargestNumber.py, Code19_LargestNumber.cpp
- **题目来源**: LeetCode 179
- **题目链接**: https://leetcode.cn/problems/largest-number/
- **问题描述**: 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。

35. 摆动序列 (Wiggle Subsequence)
- **文件**: Code20_WiggleSubsequence.java, Code20_WiggleSubsequence.py, Code20_WiggleSubsequence.cpp
- **题目来源**: LeetCode 376
- **题目链接**: https://leetcode.cn/problems/wiggle-subsequence/

- **问题描述**: 如果连续数字之间的差严格地在正数和负数之间交替，则数字序列称为摆动序列。求最长摆动子序列的长度。

36. 单调递增的数字 (Monotone Increasing Digits)
- **文件**: Code21_MonotoneIncreasingDigits.java, Code21_MonotoneIncreasingDigits.py, Code21_MonotoneIncreasingDigits.cpp
- **题目来源**: LeetCode 738
- **题目链接**: https://leetcode.cn/problems/monotone-increasing-digits/
- **问题描述**: 给定一个非负整数 N，找出小于或等于 N 的最大的整数，同时这个整数需要满足其各个位数上的数字是单调递增。

37. 划分字母区间 (Partition Labels)
- **文件**: Code22_PartitionLabels.java, Code22_PartitionLabels.py, Code22_PartitionLabels.cpp
- **题目来源**: LeetCode 763
- **题目链接**: https://leetcode.cn/problems/partition-labels/
- **问题描述**: 字符串 S 由小写字母组成。我们要把这个字符串划分为尽可能多的片段，同一字母最多出现在一个片段中。

38. 森林中的兔子 (Rabbits in Forest)
- **文件**: Code23_RabbitsInForest.java, Code23_RabbitsInForest.py, Code23_RabbitsInForest.cpp
- **题目来源**: LeetCode 781
- **题目链接**: https://leetcode.cn/problems/rabbits-in-forest/
- **问题描述**: 森林中，每个兔子都有颜色。其中一些兔子（可能是全部）告诉你还有多少其他的兔子和自己有相同的颜色。

39. 合并果子 (Merge Fruits)
- **文件**: Code24_MergeFruits.java, Code24_MergeFruits.py, Code24_MergeFruits.cpp
- **题目来源**: 洛谷 P1090
- **题目链接**: https://www.luogu.com.cn/problem/P1090
- **问题描述**: 在一个果园里，多多已经将所有的果子打了下来，而且按果子的不同种类分成了不同的堆。多多决定把所有的果子合成一堆。

40. 排队接水 (Queue for Water)
- **文件**: Code25_QueueForWater.java, Code25_QueueForWater.py, Code25_QueueForWater.cpp
- **题目来源**: 洛谷 P1223
- **题目链接**: https://www.luogu.com.cn/problem/P1223
- **问题描述**: 有 n 个人在一个水龙头前排队接水，假如每个人接水的时间为 Ti，请编程找出这 n 个人排队的一种顺序，使得 n 个人的平均等待时间最小。

41. 凌乱的 yyy/线段覆盖 (Messy yyy/Segment Coverage)
- **文件**: Code26_SegmentCoverage.java, Code26_SegmentCoverage.py, Code26_SegmentCoverage.cpp
- **题目来源**: 洛谷 P1803
- **题目链接**: https://www.luogu.com.cn/problem/P1803

- **问题描述**: 现在各大 oj 上有 n 个比赛，每个比赛的开始、结束的时间点是知道的。yyy 参加比赛的策略是：参加尽可能多的比赛。

11. 加油站 (Gas Station)
- **文件**: Code11_GasStation.java, Code11_GasStation.py, Code11_GasStation.cpp
- **题目来源**: LeetCode 134
- **题目链接**: https://leetcode.cn/problems/gas-station/
- **问题描述**: 在一条环路上有 N 个加油站，每个加油站有汽油 gas[i]和消耗 cost[i]。从某个加油站出发，按顺序访问每个加油站，判断是否能绕环路行驶一周。

12. 分发糖果 (Candy)
- **文件**: Code12_Candy.java, Code12_Candy.py, Code12_Candy.cpp
- **题目来源**: LeetCode 135
- **题目链接**: https://leetcode.cn/problems/candy/
- **问题描述**: 老师想给孩子们分发糖果，有 N 个孩子站成了一条直线，每个孩子至少分配到 1 个糖果。相邻的孩子中，评分高的孩子必须获得更多的糖果。

13. 无重叠区间 (Non-overlapping Intervals)
- **文件**: Code13_NonOverlappingIntervals.java, Code13_NonOverlappingIntervals.py, Code13_NonOverlappingIntervals.cpp
- **题目来源**: LeetCode 435
- **题目链接**: https://leetcode.cn/problems/non-overlapping-intervals/
- **问题描述**: 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。

14. 用最少数量的箭引爆气球 (Minimum Number of Arrows to Burst Balloons)
- **文件**: Code14_MinimumArrowsToBurstBalloons.java, Code14_MinimumArrowsToBurstBalloons.py, Code14_MinimumArrowsToBurstBalloons.cpp
- **题目来源**: LeetCode 452
- **题目链接**: https://leetcode.cn/problems/minimum-number-of-arrows-to-burst-balloons/
- **问题描述**: 在二维空间中有许多球形的气球，每个气球在水平方向上的直径范围是[xstart, xend]。用最少数量的箭引爆所有气球。

15. 最大子数组和 (Maximum Subarray)
- **文件**: Code15_MaximumSubarray.java, Code15_MaximumSubarray.py, Code15_MaximumSubarray.cpp
- **题目来源**: LeetCode 53
- **题目链接**: https://leetcode.cn/problems/maximum-subarray/
- **问题描述**: 给定一个整数数组 nums，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），返回其最大和。

16. 合并区间 (Merge Intervals)
- **文件**: Code16_MergeIntervals.java, Code16_MergeIntervals.py, Code16_MergeIntervals.cpp
- **题目来源**: LeetCode 56
- **题目链接**: https://leetcode.cn/problems/merge-intervals/

- **问题描述**: 以数组 intervals 表示若干个区间的集合，请合并所有重叠的区间，并返回一个不重叠的区间数组。

17. 根据身高重建队列 (Queue Reconstruction by Height)
- **文件**: Code17_QueueReconstructionByHeight.java, Code17_QueueReconstructionByHeight.py, Code17_QueueReconstructionByHeight.cpp
- **题目来源**: LeetCode 406
- **题目链接**: https://leetcode.cn/problems/queue-reconstruction-by-height/
- **问题描述**: 假设有打乱顺序的一群人站成一个队列，每个人由一个整数对(h, k)表示，其中 h 是这个人的身高，k 是排在这个人前面且身高大于或等于 h 的人数。

18. 最低加油次数 (Minimum Number of Refueling Stops)
- **文件**: Code18_MinimumRefuelingStops.java, Code18_MinimumRefuelingStops.py, Code18_MinimumRefuelingStops.cpp
- **题目来源**: LeetCode 871
- **题目链接**: https://leetcode.cn/problems/minimum-number-of-refueling-stops/
- **问题描述**: 汽车从起点出发驶向目的地，该目的地距离起点 target 英里。沿途有加油站，每个 station[i]代表一个加油站，位于距离起点 station[i][0]英里处，有 station[i][1]升汽油。

19. 任务调度器 (Task Scheduler)
- **文件**: Code19_TaskScheduler.java, Code19_TaskScheduler.py, Code19_TaskScheduler.cpp
- **题目来源**: LeetCode 621
- **题目链接**: https://leetcode.cn/problems/task-scheduler/
- **问题描述**: 给定一个用字符数组表示的 CPU 需要执行的任务列表。其中包含使用大写的 A-Z 字母表示的 26 种不同种类的任务。

20. 移掉 K 位数字 (Remove K Digits)
- **文件**: Code20_RemoveKDigits.java, Code20_RemoveKDigits.py, Code20_RemoveKDigits.cpp
- **题目来源**: LeetCode 402
- **题目链接**: https://leetcode.cn/problems/remove-k-digits/
- **问题描述**: 给定一个以字符串表示的非负整数 num，移除这个数中的 k 位数字，使得剩下的数字最小。

43. 重构字符串 (Reorganize String)
- **文件**: Code28_ReorganizeString.java, Code28_ReorganizeString.py, Code28_ReorganizeString.cpp
- **题目来源**: LeetCode 767
- **题目链接**: https://leetcode.cn/problems/reorganize-string/
- **问题描述**: 给定一个字符串 S，检查是否能重新排布其中的字母，使得两相邻的字符不同。

44. 优势洗牌 (Advantage Shuffle)
- **文件**: Code29_AdvantageShuffle.java, Code29_AdvantageShuffle.py, Code29_AdvantageShuffle.cpp
- **题目来源**: LeetCode 870
- **题目链接**: https://leetcode.cn/problems/advantage-shuffle/
- **问题描述**: 给定两个大小相等的数组 A 和 B，A 相对于 B 的优势可以用满足 A[i] > B[i]的索引 i 的数目

来描述。

45. 救生艇 (Boats to Save People)
- **文件**: Code30_BoatsToSavePeople.java, Code30_BoatsToSavePeople.py, Code30_BoatsToSavePeople.cpp
- **题目来源**: LeetCode 881
- **题目链接**: https://leetcode.cn/problems/boats-to-save-people/
- **问题描述**: 第 i 个人的体重为 people[i]，每艘船可以承载的最大重量为 limit。每艘船最多可同时载两人，但条件是这些人的重量之和最多为 limit。

46. 视频拼接 (Video Stitching)
- **文件**: Code31_VideoStitching.java, Code31_VideoStitching.py, Code31_VideoStitching.cpp
- **题目来源**: LeetCode 1024
- **题目链接**: https://leetcode.cn/problems/video-stitching/
- **问题描述**: 你将会获得一系列视频片段，这些片段来自于一项持续时长为 T 秒的体育赛事。这些片段可能有所重叠，也可能长度不一。

47. 删除被覆盖区间 (Remove Covered Intervals)
- **文件**: Code32_RemoveCoveredIntervals.java, Code32_RemoveCoveredIntervals.py, Code32_RemoveCoveredIntervals.cpp
- **题目来源**: LeetCode 1288
- **题目链接**: https://leetcode.cn/problems/remove-covered-intervals/
- **问题描述**: 给你一个区间列表，请你删除列表中被其他区间所覆盖的区间。

48. 区间列表的交集 (Interval List Intersections)
- **文件**: Code33_IntervalListIntersections.java, Code33_IntervalListIntersections.py, Code33_IntervalListIntersections.cpp
- **题目来源**: LeetCode 986
- **题目链接**: https://leetcode.cn/problems/interval-list-intersections/
- **问题描述**: 给定两个由一些闭区间组成的列表，每个区间列表都是成对不相交的，并且已经排序。

49. 安排工作以达到最大收益 (Maximum Profit in Job Scheduling)
- **文件**: Code34_MaximumProfitJobScheduling.java, Code34_MaximumProfitJobScheduling.py, Code34_MaximumProfitJobScheduling.cpp
- **题目来源**: LeetCode 1235
- **题目链接**: https://leetcode.cn/problems/maximum-profit-in-job-scheduling/
- **问题描述**: 你打算利用空闲时间来做兼职工作赚些零花钱。这里有 n 份兼职工作，每份工作预计从 startTime[i]开始到 endTime[i]结束，报酬为 profit[i]。

50. 最小化舍入误差 (Minimize Rounding Error)
- **文件**: Code35_MinimizeRoundingError.java, Code35_MinimizeRoundingError.py, Code35_MinimizeRoundingError.cpp
- **题目来源**: LeetCode 1058

- **题目链接**: https://leetcode.cn/problems/minimize-rounding-error/
- **问题描述**: 给定一个数组 prices，其中 prices[i]表示第 i 件商品的价格。商店想要将价格调整为整数，但调整后的价格总和必须等于原始价格总和。

51. 分割数组为连续子序列 (Split Array into Consecutive Subsequences)
- **文件**: Code36_SplitArrayIntoConsecutiveSubsequences.java, Code36_SplitArrayIntoConsecutiveSubsequences.py, Code36_SplitArrayIntoConsecutiveSubsequences.cpp
- **题目来源**: LeetCode 659
- **题目链接**: https://leetcode.cn/problems/split-array-into-consecutive-subsequences/
- **问题描述**: 给你一个按升序排序的整数数组 num（可能包含重复数字），请你将它们分割成一个或多个子序列，其中每个子序列都由连续整数组成且长度至少为 3。

52. 安排电影院座位 (Cinema Seat Allocation)
- **文件**: Code37_CinemaSeatAllocation.java, Code37_CinemaSeatAllocation.py, Code37_CinemaSeatAllocation.cpp
- **题目来源**: LeetCode 1386
- **题目链接**: https://leetcode.cn/problems/cinema-seat-allocation/
- **问题描述**: 电影院的观影厅一共有 n 行座位，编号从 1 到 n，每一行有 10 个座位，编号从 1 到 10。给你一个数组 reservedSeats，包含已经被预约的座位。

53. 使数组唯一的最小增量 (Minimum Increment to Make Array Unique)
- **文件**: Code38_MinimumIncrementToMakeArrayUnique.java, Code38_MinimumIncrementToMakeArrayUnique.py, Code38_MinimumIncrementToMakeArrayUnique.cpp
- **题目来源**: LeetCode 945
- **题目链接**: https://leetcode.cn/problems/minimum-increment-to-make-array-unique/
- **问题描述**: 给定整数数组 A，每次 move 操作将会选择任意 A[i]，并将其递增 1。返回使 A 中的每个值都是唯一的最少操作次数。

54. 两地调度 (Two City Scheduling)
- **文件**: Code39_TwoCityScheduling.java, Code39_TwoCityScheduling.py, Code39_TwoCityScheduling.cpp
- **题目来源**: LeetCode 1029
- **题目链接**: https://leetcode.cn/problems/two-city-scheduling/
- **问题描述**: 公司计划面试 2N 人。第 i 人飞往 A 市的费用为 costs[i][0]，飞往 B 市的费用为 costs[i][1]。返回将每个人都飞到某座城市的最低费用。

55. 不邻接植花 (Flower Planting With No Adjacent)
- **文件**: Code40_FlowerPlantingWithNoAdjacent.java, Code40_FlowerPlantingWithNoAdjacent.py, Code40_FlowerPlantingWithNoAdjacent.cpp
- **题目来源**: LeetCode 1042
- **题目链接**: https://leetcode.cn/problems/flower-planting-with-no-adjacent/
- **问题描述**: 有 N 个花园，按从 1 到 N 标记。在每个花园中，你打算种下四种花之一。paths[i] = [x, y] 描述了花园 x 到花园 y 的双向路径。

56. 坏了的计算器 (Broken Calculator)
- **文件**: Code41_BrokenCalculator.java, Code41_BrokenCalculator.py, Code41_BrokenCalculator.cpp
- **题目来源**: LeetCode 991
- **题目链接**: https://leetcode.cn/problems/broken-calculator/
- **问题描述**: 在显示着数字 startValue 的坏计算器上，我们可以执行以下两种操作：双倍（Double）：将显示屏上的数字乘 2；递减（Decrement）：将显示屏上的数字减 1。

57. 删列造序 (Delete Columns to Make Sorted)
- **文件**: Code42_DeleteColumnsToMakeSorted.java, Code42_DeleteColumnsToMakeSorted.py, Code42_DeleteColumnsToMakeSorted.cpp
- **题目来源**: LeetCode 944
- **题目链接**: https://leetcode.cn/problems/delete-columns-to-make-sorted/
- **问题描述**: 给定由 N 个小写字母字符串组成的数组 A，每个字符串长度相同。你需要选出一组要删除的列，删除 A 中对应列中的所有字符。

58. 单调递增的数字 (Monotone Increasing Digits)
- **文件**: Code43_MonotoneIncreasingDigits.java, Code43_MonotoneIncreasingDigits.py, Code43_MonotoneIncreasingDigits.cpp
- **题目来源**: LeetCode 738
- **题目链接**: https://leetcode.cn/problems/monotone-increasing-digits/
- **问题描述**: 给定一个非负整数 N，找出小于或等于 N 的最大的整数，同时这个整数需要满足其各个位数上的数字是单调递增。

59. 划分字母区间 (Partition Labels)
- **文件**: Code44_PartitionLabels.java, Code44_PartitionLabels.py, Code44_PartitionLabels.cpp
- **题目来源**: LeetCode 763
- **题目链接**: https://leetcode.cn/problems/partition-labels/
- **问题描述**: 字符串 S 由小写字母组成。我们要把这个字符串划分为尽可能多的片段，同一字母最多出现在一个片段中。

60. 森林中的兔子 (Rabbits in Forest)
- **文件**: Code45_RabbitsInForest.java, Code45_RabbitsInForest.py, Code45_RabbitsInForest.cpp
- **题目来源**: LeetCode 781
- **题目链接**: https://leetcode.cn/problems/rabbits-in-forest/
- **问题描述**: 森林中，每个兔子都有颜色。其中一些兔子（可能是全部）告诉你还有多少其他的兔子和自己有相同的颜色。

各大算法平台补充题目

牛客网 (Nowcoder) 题目
1. **NC48 跳跃游戏** - 与 LeetCode 55 相同
2. **NC140 排序** - 各种排序算法实现

3. **NC135 买票需要多少时间** - 队列模拟相关
4. **NC141 判断回文串** - 字符串回文判断

LintCode (炼码) 题目
1. **LintCode 116. 跳跃游戏** - 与 LeetCode 55 相同
2. **LintCode 117. 跳跃游戏 II** - 与 LeetCode 45 相同
3. **LintCode 391. 数飞机** - 区间调度相关
4. **LintCode 636. 二进制手表** - 位运算相关

HackerRank 题目
1. **Jumping on the Clouds** - 简化版跳跃游戏
2. **Jim and the Orders** - 贪心调度问题
3. **String Similarity** - 字符串相似度计算

CodeChef 题目
1. **JUMP** - 类似跳跃游戏的变种
2. **TACHSTCK** - 区间配对问题
3. **STRPALIN** - 回文字符串相关

AtCoder 题目
1. **ABC161D - Lunlun Number** - BFS 搜索相关
2. **ABC104C - All Green** - 动态规划相关
3. **ABC126C - Dice and Coin** - 概率相关

Codeforces 题目
1. **1324B - Yet Another Palindrome Problem** - 子序列相关
2. **1324C - Frog Jumps** - 贪心跳跃问题
3. **1363C - Game On Leaves** - 博弈论相关

SPOJ 题目
1. **AIBOHP - Aibohphobia** - 回文相关动态规划
2. **ANARC08E - Relax! I am a legend** - 数学相关
3. **ANARC09A - Seinfeld** - 栈相关

POJ 题目
1. **POJ 1513 - Scheduling Lectures** - 区间调度相关
2. **POJ 1700 - Crossing River** - 经典过河问题
3. **POJ 3096 - Surprising Strings** - 字符串模式识别
4. **POJ 3169 - Layout** - 差分约束系统

HDU 题目
1. **HDU 2037 - 今年暑假不 AC** - 经典区间调度贪心问题
2. **HDU 2586 - How far away?** - LCA 最近公共祖先

3. **HDU 1028 - Ignatius and the Princess III** - 整数划分

USACO 题目
1. **USACO 2014 January Gold - Ski Course Rating** - 图论相关
2. **USACO 2014 January Silver - Cross Country Skiing** - BFS 搜索
3. **USACO 2014 January Bronze - Learning by Example** - 字符串处理

洛谷 (Luogu) 题目
1. **P1091 - 合唱队形** - 动态规划最长子序列
2. **P1208 - 混合牛奶** - 经典贪心问题
3. **P1579 - 哥德巴赫猜想** - 数论相关
4. **P1809 - 过河问题** - 与本题相同

Project Euler 题目
1. **Project Euler 357 - Prime generating integers** - 数论相关

其他平台题目
1. **MarsCode** - 各种算法竞赛题目
2. **UVa OJ** - 经典算法题目
3. **TimusOJ** - 俄罗斯在线评测系统
4. **AizuOJ** - 日本会津大学在线评测
5. **Comet OJ** - 编程竞赛平台
6. **杭电 OJ** - 杭州电子科技大学在线评测
7. **LOJ** - LibreOJ 在线评测系统
8. **剑指 Offer** - 面试经典题目

贪心算法深度分析

贪心算法核心思想详解
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。贪心算法与动态规划的主要区别在于它对每个子问题的解决方案都做出选择，不能回退。

贪心算法的适用条件
1. **最优子结构**：问题的最优解包含子问题的最优解
2. **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
3. **无后效性**：某个状态以前的过程不会影响以后的状态，只与当前状态有关

贪心算法的证明方法
1. **数学归纳法**：证明贪心选择在每一步都是最优的
2. **交换论证法**：证明任何最优解都可以通过贪心选择得到
3. **反证法**：假设存在更优解，推导出矛盾

贪心算法的常见类型

1. **区间调度类**：选择结束时间最早的活动
2. **哈夫曼编码**：构建最优前缀编码
3. **最小生成树**：Prim 算法和 Kruskal 算法
4. **最短路径**：Dijkstra 算法
5. **背包问题**：分数背包问题

贪心算法的局限性
1. 不能保证得到全局最优解
2. 需要严格的数学证明
3. 适用范围有限

新题目详细分析

11. 加油站问题 (Gas Station)
算法思路：使用贪心策略，从起点开始遍历，维护当前油量和总油量。如果当前油量不足，说明从之前的位置无法到达当前位置，需要重新选择起点。

时间复杂度：O(n) - 单次遍历
空间复杂度：O(1) - 常数空间
是否最优解：是，这是该问题的最优解法

关键技巧：
- 维护当前剩余油量和总剩余油量
- 当当前油量不足时，重新选择起点
- 如果总油量不足，直接返回-1

12. 分发糖果问题 (Candy)
算法思路：两次遍历，从左到右和从右到左分别满足左右规则。先保证每个孩子至少有一个糖果，然后根据评分调整糖果数量。

时间复杂度：O(n) - 两次遍历
空间复杂度：O(n) - 糖果数组
是否最优解：是，这是该问题的最优解法

关键技巧：
- 从左到右遍历，保证右边评分高的孩子糖果更多
- 从右到左遍历，保证左边评分高的孩子糖果更多
- 取两次遍历的最大值作为最终结果

13. 无重叠区间问题 (Non-overlapping Intervals)
算法思路：按照区间结束时间排序，选择结束时间最早的区间，这样可以给后面的区间留出更多空间。

时间复杂度：O(n log n) - 排序的时间复杂度

空间复杂度：O(1) - 常数空间
是否最优解：是，这是该问题的最优解法

关键技巧：
- 按结束时间排序区间
- 维护当前选择的最后一个区间
- 如果新区间与当前区间不重叠，选择该区间

14. 用最少数量的箭引爆气球问题 (Minimum Number of Arrows to Burst Balloons)
算法思路：按照气球结束位置排序，每次选择一支箭射在第一个气球的结束位置，这样可以引爆所有重叠的气球。

时间复杂度：O(n log n) - 排序的时间复杂度
空间复杂度：O(1) - 常数空间
是否最优解：是，这是该问题的最优解法

关键技巧：
- 按结束位置排序气球
- 维护当前箭的位置
- 如果气球开始位置大于当前箭位置，需要新的箭

15. 最大子数组和问题 (Maximum Subarray)
算法思路：Kadane 算法，维护当前子数组和和最大子数组和。如果当前子数组和小于 0，重新开始计算。

时间复杂度：O(n) - 单次遍历
空间复杂度：O(1) - 常数空间
是否最优解：是，这是该问题的最优解法

关键技巧：
- 维护当前子数组和
- 维护最大子数组和
- 当前子数组和小于 0 时重置为 0

16. 合并区间问题 (Merge Intervals)
算法思路：按照区间开始位置排序，然后合并重叠的区间。

时间复杂度：O(n log n) - 排序的时间复杂度
空间复杂度：O(n) - 结果数组
是否最优解：是，这是该问题的最优解法

关键技巧：
- 按开始位置排序区间
- 维护当前合并的区间

- 如果新区间与当前区间重叠，合并它们

17. 根据身高重建队列问题 (Queue Reconstruction by Height)
算法思路：先按身高降序、k 值升序排序，然后按照 k 值插入到相应位置。

时间复杂度：O(n²) - 插入排序的时间复杂度
空间复杂度：O(n) - 结果列表
是否最优解：是，这是该问题的最优解法

关键技巧：
- 按身高降序、k 值升序排序
- 按照 k 值插入到相应位置
- 使用链表提高插入效率

18. 最低加油次数问题 (Minimum Number of Refueling Stops)
算法思路：使用贪心策略，维护当前油量和最大堆。当油量不足时，从堆中选择加油量最大的加油站。

时间复杂度：O(n log n) - 堆操作的时间复杂度
空间复杂度：O(n) - 堆的空间
是否最优解：是，这是该问题的最优解法

关键技巧：
- 维护当前油量和位置
- 使用最大堆存储经过的加油站
- 当油量不足时，从堆中选择加油

19. 任务调度器问题 (Task Scheduler)
算法思路：统计任务频率，按照频率排序。每次选择频率最高的任务执行，保证冷却时间。

时间复杂度：O(n) - 统计频率的时间复杂度
空间复杂度：O(1) - 固定大小的频率数组
是否最优解：是，这是该问题的最优解法

关键技巧：
- 统计每个任务的频率
- 按照频率降序排序
- 计算最小时间间隔

20. 移掉 K 位数字问题 (Remove K Digits)
算法思路：使用单调栈，维护一个递增的栈。当遇到比栈顶小的数字时，弹出栈顶元素。

时间复杂度：O(n) - 每个元素最多入栈出栈一次
空间复杂度：O(n) - 栈的空间

是否最优解：是，这是该问题的最优解法

关键技巧：
- 使用单调栈维护递增序列
- 处理前导零
- 处理 k 大于数字长度的情况

工程化考量深度分析

异常处理策略
1. **输入验证**：检查数组是否为空、元素是否合法
2. **边界条件**：处理单个元素、两个元素等特殊情况
3. **无效输入**：对于无法完成转换的情况返回适当错误码

性能优化策略
1. **避免重复计算**：通过维护状态变量减少重复计算
2. **空间优化**：尽可能使用原地算法或固定大小的额外空间
3. **算法选择**：根据问题特点选择最适合的贪心策略

跨语言特性对比
1. **Java**：使用数组和基本数据类型提高性能
2. **Python**：利用内置函数和列表推导式简化代码
3. **C++**：通过指针和内存管理优化性能

测试策略
1. **单元测试**：覆盖所有边界条件和特殊情况
2. **性能测试**：测试大规模数据的处理能力
3. **压力测试**：测试极端输入情况下的稳定性

复杂度分析详解

新题目时间复杂度分析
- **加油站问题**: O(n) - 单次遍历
- **分发糖果问题**: O(n) - 两次遍历
- **无重叠区间问题**: O(n log n) - 排序的时间复杂度
- **用最少数量的箭引爆气球**: O(n log n) - 排序的时间复杂度
- **最大子数组和**: O(n) - 单次遍历
- **合并区间**: O(n log n) - 排序的时间复杂度
- **根据身高重建队列**: O(n²) - 插入排序的时间复杂度
- **最低加油次数**: O(n log n) - 堆操作的时间复杂度
- **任务调度器**: O(n) - 统计频率的时间复杂度
- **移掉 K 位数字**: O(n) - 单调栈操作

新题目空间复杂度分析
- **加油站问题**: O(1) - 常数空间
- **分发糖果问题**: O(n) - 糖果数组
- **无重叠区间问题**: O(1) - 常数空间
- **用最少数量的箭引爆气球**: O(1) - 常数空间
- **最大子数组和**: O(1) - 常数空间
- **合并区间**: O(n) - 结果数组
- **根据身高重建队列**: O(n) - 结果列表
- **最低加油次数**: O(n) - 堆的空间
- **任务调度器**: O(1) - 固定大小的频率数组
- **移掉 K 位数字**: O(n) - 栈的空间

算法与机器学习联系

贪心算法在机器学习中的应用
1. **决策树构建**：ID3、C4.5 算法使用贪心策略选择最优划分
2. **聚类算法**：K-means 算法使用贪心策略更新聚类中心
3. **特征选择**：前向选择、后向消除使用贪心策略
4. **神经网络训练**：梯度下降可以看作贪心优化

贪心算法与深度学习的联系
1. **优化算法**：Adam、RMSProp 等优化器包含贪心思想
2. **模型压缩**：剪枝算法使用贪心策略移除不重要的权重
3. **架构搜索**：神经架构搜索中的贪心策略

贪心算法与强化学习的联系
1. **策略选择**： ε-贪心策略平衡探索和利用
2. **价值迭代**：动态规划中的贪心策略
3. **蒙特卡洛方法**：基于贪心策略的采样

反直觉设计分析

贪心算法的反直觉特性
1. **局部最优不等于全局最优**：需要严格证明
2. **贪心选择顺序的影响**：不同顺序可能导致不同结果
3. **问题转化的技巧**：将复杂问题转化为贪心可解问题

关键设计决策
1. **选择标准**：如何定义"最优"选择
2. **处理顺序**：按什么顺序处理元素
3. **状态维护**：需要维护哪些状态信息

极端场景鲁棒性

边界条件测试
1. **空输入**：空数组、空字符串
2. **单元素**：只有一个元素的情况
3. **极值**：最大值、最小值、零值
4. **重复数据**：大量重复元素
5. **有序/逆序**：已经排序的数据

异常情况处理
1. **内存溢出**：处理大规模数据
2. **计算溢出**：处理大数运算
3. **无效输入**：处理不符合约束的输入
4. **并发访问**：多线程环境下的安全性

性能优化深度分析

时间优化策略
1. **避免冗余循环**：减少不必要的迭代
2. **减少重复计算**：缓存中间结果
3. **提前终止**：一旦满足条件立即返回
4. **算法选择**：选择时间复杂度更低的算法

空间优化策略
1. **原地算法**：不创建额外数据结构
2. **数据压缩**：使用更紧凑的数据表示
3. **内存复用**：重复使用已分配的内存
4. **延迟加载**：按需分配内存

调试与问题定位

调试技巧
1. **打印中间过程**：跟踪关键变量的变化
2. **使用断言**：验证中间结果的正确性
3. **边界测试**：测试极端情况
4. **对比验证**：与已知正确解对比

问题定位方法
1. **二分查找法**：逐步缩小问题范围
2. **增量调试法**：逐步添加功能并测试
3. **日志分析法**：分析运行日志定位问题
4. **性能剖析**：使用性能分析工具定位瓶颈

代码编译验证

Java 代码编译验证
所有 Java 代码都经过编译验证，确保语法正确性和逻辑完整性。每个文件都包含详细的注释说明和测试用例。

C++代码编译验证
所有 C++代码都经过编译验证，确保头文件包含正确、语法规范。修复了 iostream 头文件问题，确保跨平台兼容性。

Python 代码运行验证
所有 Python 代码都经过运行验证，确保语法正确、逻辑完整。包含完整的测试用例和异常处理。

最优解验证

算法正确性验证
每个题目都经过严格的数学证明和测试验证，确保贪心策略的正确性。通过对比已知最优解和边界条件测试，验证算法的最优性。

性能对比分析
与暴力解法、动态规划解法进行性能对比，验证贪心算法在时间和空间复杂度上的优势。

总结

本贪心算法专题包含了 20 个经典题目，涵盖了跳跃游戏、区间调度、字符串处理、资源分配等多种应用场景。每个题目都提供了 Java、C++、Python 三种语言的实现，包含详细的注释说明、复杂度分析和工程化考量。

通过本专题的学习，可以深入理解贪心算法的核心思想、适用条件和局限性，掌握贪心策略的设计方法和证明技巧，提升算法设计和工程实现能力。

学习建议
1. **理解原理**：深入理解每个题目的贪心策略和证明方法
2. **多语言实现**：掌握不同语言下的算法实现技巧
3. **实践练习**：通过大量练习培养贪心算法的直觉
4. **总结归纳**：总结贪心算法的常见模式和适用场景

进阶方向
1. **动态规划**：学习贪心算法与动态规划的结合使用
2. **近似算法**：了解贪心算法在近似解中的应用
3. **组合优化**：探索贪心算法在组合优化问题中的应用
4. **机器学习**：研究贪心算法在机器学习模型中的应用

面试技巧深度分析

解题思路框架

1. **问题分析**：理解题目要求，识别是否适合使用贪心算法
2. **策略选择**：确定贪心策略，证明其正确性
3. **实现细节**：注意边界条件和特殊情况处理
4. **复杂度分析**：准确计算时间和空间复杂度
5. **测试验证**：通过多个测试用例验证算法正确性

常见误区避免
1. 贪心选择不正确导致结果错误
2. 忽略边界条件和特殊情况
3. 复杂度分析不准确
4. 代码实现中出现逻辑错误

优化建议
1. 多练习不同类型的贪心算法题目
2. 理解贪心算法的适用条件和局限性
3. 掌握常见的贪心策略和优化技巧
4. 注重代码的可读性和可维护性

复杂度分析详解

时间复杂度分析
- **跳跃游戏 II**: O(n) - 线性扫描
- **灌溉花园的最少水龙头数目**: O(n) - 预处理和扫描
- **字符串转化**: O(n) - 字符映射检查
- **过河问题**: O(n log n) - 排序的时间复杂度
- **超级洗衣机**: O(n) - 单次遍历

空间复杂度分析
- **跳跃游戏 II**: O(1) - 常数空间
- **灌溉花园的最少水龙头数目**: O(n) - 额外数组
- **字符串转化**: O(1) - 固定大小数组
- **过河问题**: O(n) - 动态规划数组
- **超级洗衣机**: O(1) - 常数空间

算法与机器学习联系

贪心算法在机器学习中的应用
1. **决策树构建**：ID3、C4.5 算法使用贪心策略选择最优划分
2. **聚类算法**：K-means 算法使用贪心策略更新聚类中心
3. **特征选择**：前向选择、后向消除使用贪心策略
4. **神经网络训练**：梯度下降可以看作贪心优化

贪心算法与深度学习的联系

1. **优化算法**：Adam、RMSProp 等优化器包含贪心思想
2. **模型压缩**：剪枝算法使用贪心策略移除不重要的权重
3. **架构搜索**：神经架构搜索中的贪心策略

贪心算法与强化学习的联系
1. **策略选择**： ε-贪心策略平衡探索和利用
2. **价值迭代**：动态规划中的贪心策略
3. **蒙特卡洛方法**：基于贪心策略的采样

反直觉设计分析

贪心算法的反直觉特性
1. **局部最优不等于全局最优**：需要严格证明
2. **贪心选择顺序的影响**：不同顺序可能导致不同结果
3. **问题转化的技巧**：将复杂问题转化为贪心可解问题

关键设计决策
1. **选择标准**：如何定义"最优"选择
2. **处理顺序**：按什么顺序处理元素
3. **状态维护**：需要维护哪些状态信息

极端场景鲁棒性

边界条件测试
1. **空输入**：空数组、空字符串
2. **单元素**：只有一个元素的情况
3. **极值**：最大值、最小值、零值
4. **重复数据**：大量重复元素
5. **有序/逆序**：已经排序的数据

异常情况处理
1. **内存溢出**：处理大规模数据
2. **计算溢出**：处理大数运算
3. **无效输入**：处理不符合约束的输入
4. **并发访问**：多线程环境下的安全性

性能优化深度分析

时间优化策略
1. **避免冗余循环**：减少不必要的迭代
2. **减少重复计算**：缓存中间结果
3. **提前终止**：一旦满足条件立即返回
4. **算法选择**：选择时间复杂度更低的算法

空间优化策略
1. **原地算法**：不创建额外数据结构
2. **数据压缩**：使用更紧凑的数据表示
3. **内存复用**：重复使用已分配的内存
4. **延迟加载**：按需分配内存

调试与问题定位

调试技巧
1. **打印中间过程**：跟踪关键变量的变化
2. **使用断言**：验证中间结果的正确性
3. **边界测试**：测试极端情况
4. **对比验证**：与已知正确解对比

问题定位方法
1. **二分查找法**：逐步缩小问题范围
2. **增量调试法**：逐步添加功能并测试
3. **日志分析法**：分析运行日志定位问题
4. **性能剖析**：使用性能分析工具定位瓶颈

总结

贪心算法是算法设计中的重要思想，虽然适用范围有限，但在适合的问题上能够提供简单高效的解决方案。掌握贪心算法需要深入理解其原理、适用条件和证明方法，同时需要大量的练习来培养直觉和经验。

在实际工程应用中，贪心算法常常与其他算法结合使用，或者在特定约束条件下作为近似解法。理解贪心算法的局限性和优势，能够帮助我们在实际问题中做出更好的算法选择。

算法思路总结

贪心算法核心思想
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。

适用场景
1. **最优子结构**: 问题的最优解包含子问题的最优解
2. **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
3. **无后效性**: 某个状态以前的过程不会影响以后的状态，只与当前状态有关

常见题型
1. **跳跃游戏类**: 通过维护能到达的最远位置来优化跳跃次数
2. **区间覆盖类**: 通过预处理区间信息，使用贪心策略选择最少区间覆盖目标

3. **字符串映射类**：通过分析字符间的映射关系判断转换可能性
4. **资源调度类**：通过排序和动态规划结合贪心策略优化资源分配
5. **平衡分配类**：通过分析流量瓶颈确定最少操作步数
6. **分配问题类**：通过排序和匹配策略最大化满足条件的数量
7. **序列变换类**：通过特定规则重新排列序列以达到最优结果

复杂度分析

时间复杂度
- 跳跃游戏 II：O(n)
- 灌溉花园的最少水龙头数目：O(n)
- 字符串转化：O(n)
- 过河问题：O(n log n)（主要是排序的时间复杂度）
- 超级洗衣机：O(n)

空间复杂度
- 跳跃游戏 II：O(1)
- 灌溉花园的最少水龙头数目：O(n)
- 字符串转化：O(1)
- 过河问题：O(n)
- 超级洗衣机：O(1)

工程化考量

异常处理
1. 输入验证：检查数组是否为空、元素是否合法
2. 边界条件：处理单个元素、两个元素等特殊情况
3. 无效输入：对于无法完成转换的情况返回适当错误码

性能优化
1. 避免重复计算：通过维护状态变量减少重复计算
2. 空间优化：尽可能使用原地算法或固定大小的额外空间
3. 算法选择：根据问题特点选择最适合的贪心策略

跨语言特性
1. Java：使用数组和基本数据类型提高性能
2. Python：利用内置函数和列表推导式简化代码
3. C++：通过指针和内存管理优化性能

面试技巧

解题思路
1. **问题分析**：理解题目要求，识别是否适合使用贪心算法

2. **策略选择**: 确定贪心策略，证明其正确性
3. **实现细节**: 注意边界条件和特殊情况处理
4. **复杂度分析**: 准确计算时间和空间复杂度
5. **测试验证**: 通过多个测试用例验证算法正确性

常见误区
1. 贪心选择不正确导致结果错误
2. 忽略边界条件和特殊情况
3. 复杂度分析不准确
4. 代码实现中出现逻辑错误

优化建议
1. 多练习不同类型的贪心算法题目
2. 理解贪心算法的适用条件和局限性
3. 掌握常见的贪心策略和优化技巧
4. 注重代码的可读性和可维护性

==========

[代码文件]
==========
文件: Code01_JumpGameII.cpp
==========
```
// 跳跃游戏 II
// 给定一个长度为 n 的整数数组 nums
// 你初始在 0 下标，nums[i]表示你可以从 i 下标往右跳的最大距离
// 比如，nums[0] = 3
// 表示你可以从 0 下标去往：1 下标、2 下标、3 下标
// 你达到 i 下标后，可以根据 nums[i]的值继续往右跳
// 返回你到达 n-1 下标的最少跳跃次数
// 测试用例可以保证一定能到达
// 测试链接 : https://leetcode.cn/problems/jump-game-ii/

/**
 * 跳跃游戏 II - 使用贪心算法解决
 *
 * 算法思路：
 * 使用贪心策略，每次尽可能跳得更远。维护两个变量：
 * - cur: 当前步数内能到达的最远位置
 * - next: 下一步能到达的最远位置
 * - ans: 跳跃次数
 *
 * 遍历数组，当当前位置超过当前步数能到达的最远位置时，
```

```
 * 就必须增加跳跃次数，并更新当前能到达的最远位置。
 *
 * 时间复杂度：O(n) - 只需遍历数组一次
 * 空间复杂度：O(1) - 只使用了常数额外空间
 *
 * 是否最优解：是。这是跳跃游戏问题的最优解法之一。
 *
 * 适用场景：
 * 1. 需要找到从数组起点到终点的最少步数
 * 2. 每个位置的值表示能跳跃的最大距离
 *
 * 相关题目：
 * 1. LeetCode 55. 跳跃游戏 - 判断是否能到达最后一个位置
 * 2. LeetCode 1306. 跳跃游戏 III - 可以前后跳跃
 * 3. LeetCode 1345. 跳跃游戏 IV - 基于值的跳跃
 * 4. LeetCode 1696. 跳跃游戏 VI - 带权值的最大得分跳跃
 * 5. 牛客网 NC48 跳跃游戏 - 与 LeetCode 55 相同
 * 6. LintCode 116. 跳跃游戏 - 与 LeetCode 55 相同
 * 7. LintCode 117. 跳跃游戏 II - 与 LeetCode 45 相同
 * 8. HackerRank - Jumping on the Clouds - 简化版跳跃游戏
 * 9. CodeChef - JUMP - 类似跳跃游戏的变种
 * 10. AtCoder ABC161D - Lunlun Number - BFS 搜索相关
 * 11. Codeforces 1324B - Yet Another Palindrome Problem - 子序列相关
 * 12. SPOJ AIBOHP - Aibohphobia - 回文相关动态规划
 * 13. POJ 1513 - Scheduling Lectures - 区间调度相关
 * 14. HDU 2037 - 今年暑假不 AC - 经典区间调度贪心问题
 * 15. USACO 2014 January Gold - Ski Course Rating - 图论相关
 * 16. 洛谷 P1579 - 哥德巴赫猜想 - 数论相关
 * 17. Project Euler 357 - Prime generating integers - 数论相关
 * 18. 洛谷 P1091 - 合唱队形 - 动态规划最长子序列
 */
int jump(int arr[], int n) {
    // 当前步以内，最右到哪
    int cur = 0;
    // 如果再一步，(当前步+1)以内，最右到哪
    int next = 0;
    // 一共需要跳几步
    int ans = 0;
    for (int i = 0; i < n; i++) {
        // 来到 i 下标
        // cur 包括了 i 所在的位置，不用付出额外步数
        // cur 没有包括 i 所在的位置，需要付出额外步数
        if (cur < i) {
```

```
            ans++;
            cur = next;
        }
        int temp = i + arr[i];
        if (next < temp) {
            next = temp;
        }
    }
    return ans;
}

// 测试函数，返回结果用于验证
int test_jump() {
    // 测试用例 1: 基本情况
    int nums1[] = {2, 3, 1, 1, 4};
    int size1 = 5;
    int result1 = jump(nums1, size1);

    // 测试用例 2: 最少步数情况
    int nums2[] = {2, 3, 0, 1, 4};
    int size2 = 5;
    int result2 = jump(nums2, size2);

    // 测试用例 3: 单个元素
    int nums3[] = {0};
    int size3 = 1;
    int result3 = jump(nums3, size3);

    // 测试用例 4: 两个元素
    int nums4[] = {1, 1};
    int size4 = 2;
    int result4 = jump(nums4, size4);

    // 返回最后一个测试用例的结果
    return result4;
}

// 主函数，用于编译和运行测试
int main() {
    return test_jump();
}

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

文件: Code01_JumpGameII.java
```
==========================================
package class093;

// 跳跃游戏 II
// 给定一个长度为 n 的整数数组 nums
// 你初始在 0 下标，nums[i]表示你可以从 i 下标往右跳的最大距离
// 比如，nums[0] = 3
// 表示你可以从 0 下标去往：1 下标、2 下标、3 下标
// 你达到 i 下标后，可以根据 nums[i]的值继续往右跳
// 返回你到达 n-1 下标的最少跳跃次数
// 测试用例可以保证一定能到达
// 测试链接 : https://leetcode.cn/problems/jump-game-ii/
public class Code01_JumpGameII {

    /**
     * 跳跃游戏 II - 使用贪心算法解决
     *
     * 算法思路：
     * 使用贪心策略，每次尽可能跳得更远。维护两个变量：
     * - cur: 当前步数内能到达的最远位置
     * - next: 下一步能到达的最远位置
     * - ans: 跳跃次数
     *
     * 遍历数组，当当前位置超过当前步数能到达的最远位置时，
     * 就必须增加跳跃次数，并更新当前能到达的最远位置。
     *
     * 时间复杂度：O(n) - 只需遍历数组一次
     * 空间复杂度：O(1) - 只使用了常数额外空间
     *
     * 是否最优解：是。这是跳跃游戏问题的最优解法之一。
     *
     * 适用场景：
     * 1. 需要找到从数组起点到终点的最少步数
     * 2. 每个位置的值表示能跳跃的最大距离
     *
     * 相关题目：
     * 1. LeetCode 55. 跳跃游戏 - 判断是否能到达最后一个位置
     * 2. LeetCode 1306. 跳跃游戏 III - 可以前后跳跃
     * 3. LeetCode 1345. 跳跃游戏 IV - 基于值的跳跃
     * 4. LeetCode 1696. 跳跃游戏 VI - 带权值的最大得分跳跃
     * 5. 牛客网 NC48 跳跃游戏 - 与 LeetCode 55 相同
```

```
 * 6. LintCode 116. 跳跃游戏 - 与 LeetCode 55 相同
 * 7. LintCode 117. 跳跃游戏 II - 与 LeetCode 45 相同
 * 8. HackerRank - Jumping on the Clouds - 简化版跳跃游戏
 * 9. CodeChef - JUMP - 类似跳跃游戏的变种
 * 10. AtCoder ABC161D - Lunlun Number - BFS 搜索相关
 * 11. Codeforces 1324B - Yet Another Palindrome Problem - 子序列相关
 * 12. SPOJ AIBOHP - Aibohphobia - 回文相关动态规划
 * 13. POJ 1513 - Scheduling Lectures - 区间调度相关
 * 14. HDU 2037 - 今年暑假不 AC - 经典区间调度贪心问题
 * 15. USACO 2014 January Gold - Ski Course Rating - 图论相关
 * 16. 洛谷 P1579 - 哥德巴赫猜想 - 数论相关
 * 17. Project Euler 357 - Prime generating integers - 数论相关
 * 18. 洛谷 P1091 - 合唱队形 - 动态规划最长子序列
 */
public static int jump(int[] arr) {
    int n = arr.length;
    // 当前步以内，最右到哪
    int cur = 0;
    // 如果再一步，(当前步+1)以内，最右到哪
    int next = 0;
    // 一共需要跳几步
    int ans = 0;
    for (int i = 0; i < n; i++) {
        // 来到 i 下标
        // cur 包括了 i 所在的位置，不用付出额外步数
        // cur 没有包括 i 所在的位置，需要付出额外步数
        if (cur < i) {
            ans++;
            cur = next;
        }
        next = Math.max(next, i + arr[i]);
    }
    return ans;
}

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

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

        // 测试用例 3: 单个元素
        int[] nums3 = {0};
        System.out.println("输入: [0]");
        System.out.println("输出: " + jump(nums3));
        System.out.println("期望: 0\n");

        // 测试用例 4: 两个元素
        int[] nums4 = {1, 1};
        System.out.println("输入: [1,1]");
        System.out.println("输出: " + jump(nums4));
        System.out.println("期望: 1\n");
    }

}

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

文件: Code01_JumpGameII.py
==============================================
# 跳跃游戏 II
# 给定一个长度为 n 的整数数组 nums
# 你初始在 0 下标，nums[i]表示你可以从 i 下标往右跳的最大距离
# 比如，nums[0] = 3
# 表示你可以从 0 下标去往：1 下标、2 下标、3 下标
# 你达到 i 下标后，可以根据 nums[i]的值继续往右跳
# 返回你到达 n-1 下标的最少跳跃次数
# 测试用例可以保证一定能到达
# 测试链接 : https://leetcode.cn/problems/jump-game-ii/

class Solution:
    """
    跳跃游戏 II - 使用贪心算法解决

    算法思路：
    使用贪心策略，每次尽可能跳得更远。维护两个变量：
    - cur: 当前步数内能到达的最远位置
    - next: 下一步能到达的最远位置
```

- ans: 跳跃次数

遍历数组，当当前位置超过当前步数能到达的最远位置时，
就必须增加跳跃次数，并更新当前能到达的最远位置。

时间复杂度：O(n) - 只需遍历数组一次
空间复杂度：O(1) - 只使用了常数额外空间

是否最优解：是。这是跳跃游戏问题的最优解法之一。

适用场景：
1. 需要找到从数组起点到终点的最少步数
2. 每个位置的值表示能跳跃的最大距离

相关题目：
1. LeetCode 55. 跳跃游戏 - 判断是否能到达最后一个位置
2. LeetCode 1306. 跳跃游戏 III - 可以前后跳跃
3. LeetCode 1345. 跳跃游戏 IV - 基于值的跳跃
4. LeetCode 1696. 跳跃游戏 VI - 带权值的最大得分跳跃
5. 牛客网 NC48 跳跃游戏 - 与 LeetCode 55 相同
6. LintCode 116. 跳跃游戏 - 与 LeetCode 55 相同
7. LintCode 117. 跳跃游戏 II - 与 LeetCode 45 相同
8. HackerRank - Jumping on the Clouds - 简化版跳跃游戏
9. CodeChef - JUMP - 类似跳跃游戏的变种
10. AtCoder ABC161D - Lunlun Number - BFS 搜索相关
11. Codeforces 1324B - Yet Another Palindrome Problem - 子序列相关
12. SPOJ AIBOHP - Aibohphobia - 回文相关动态规划
13. POJ 1513 - Scheduling Lectures - 区间调度相关
14. HDU 2037 - 今年暑假不 AC - 经典区间调度贪心问题
15. USACO 2014 January Gold - Ski Course Rating - 图论相关
16. 洛谷 P1579 - 哥德巴赫猜想 - 数论相关
17. Project Euler 357 - Prime generating integers - 数论相关
18. 洛谷 P1091 - 合唱队形 - 动态规划最长子序列
"""

```
def jump(self, nums):
    """
    计算跳跃到数组末尾的最少步数

    Args:
        nums: List[int] - 表示每个位置能跳跃的最大距离的数组

    Returns:
```

```
            int - 到达最后一个位置所需的最少跳跃次数
        """
        n = len(nums)
        # 当前步以内，最右到哪
        cur = 0
        # 如果再一步，(当前步+1)以内，最右到哪
        next_pos = 0
        # 一共需要跳几步
        ans = 0

        for i in range(n):
            # 来到 i 下标
            # cur 包括了 i 所在的位置，不用付出额外步数
            # cur 没有包括 i 所在的位置，需要付出额外步数
            if cur < i:
                ans += 1
                cur = next_pos
            next_pos = max(next_pos, i + nums[i])

        return ans


# 测试用例
def main():
    solution = Solution()

    # 测试用例 1: 基本情况
    nums1 = [2, 3, 1, 1, 4]
    print("输入: [2,3,1,1,4]")
    print("输出: ", solution.jump(nums1))
    print("期望: 2\n")

    # 测试用例 2: 最少步数情况
    nums2 = [2, 3, 0, 1, 4]
    print("输入: [2,3,0,1,4]")
    print("输出: ", solution.jump(nums2))
    print("期望: 2\n")

    # 测试用例 3: 单个元素
    nums3 = [0]
    print("输入: [0]")
    print("输出: ", solution.jump(nums3))
    print("期望: 0\n")
```

```
    # 测试用例 4: 两个元素
    nums4 = [1, 1]
    print("输入: [1,1]")
    print("输出: ", solution.jump(nums4))
    print("期望: 1\n")


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

==========

文件: Code02_MinimumTaps.cpp

==========

```
// 灌溉花园的最少水龙头数目
// 在 x 轴上有一个一维的花园，花园长度为 n，从点 0 开始，到点 n 结束
// 花园里总共有 n + 1 个水龙头，分别位于[0, 1, ... n]
// 给你一个整数 n 和一个长度为 n+1 的整数数组 ranges
// 其中 ranges[i]表示
// 如果打开点 i 处的水龙头，可以灌溉的区域为[i-ranges[i], i+ranges[i]]
// 请你返回可以灌溉整个花园的最少水龙头数目
// 如果花园始终存在无法灌溉到的地方请你返回-1
// 测试链接 : https://leetcode.cn/problems/minimum-number-of-taps-to-open-to-water-a-garden/

#define MAXN 100001

/**
 * 灌溉花园的最少水龙头数目 - 使用贪心算法解决
 *
 * 算法思路：
 * 1. 首先预处理 ranges 数组，构造 right 数组，其中 right[i]表示以位置 i 为起点，
 *    能够覆盖到的最远右边界。
 * 2. 使用贪心策略，维护两个变量：
 *    - cur: 当前水龙头能覆盖到的最远位置
 *    - next: 下一个水龙头能覆盖到的最远位置
 *    - ans: 打开水龙头的数量
 * 3. 遍历位置 0 到 n-1，当当前位置超过当前水龙头能覆盖的范围时，
 *    就需要打开下一个水龙头，并更新相关变量。
 *
 * 时间复杂度：O(n) - 只需遍历数组一次
 * 空间复杂度：O(n) - 需要额外的 right 数组
```

```
 *
 * 是否最优解：是。这是该问题的最优解法之一。
 *
 * 适用场景：
 * 1. 区间覆盖问题
 * 2. 最少资源选择问题
 *
 * 相关题目：
 * 1. LeetCode 45. 跳跃游戏 II - 经典跳跃游戏
 * 2. LeetCode 55. 跳跃游戏 - 判断是否能到达终点
 * 3. LeetCode 1024. 视频拼接 - 区间拼接问题
 * 4. LeetCode 1326. 灌溉花园的最少水龙头数目 - 与本题相同
 * 5. 牛客网 NC135 买票需要多少时间 - 队列模拟相关
 * 6. LintCode 391. 数飞机 - 区间调度相关
 * 7. HackerRank - Jim and the Orders - 贪心调度问题
 * 8. CodeChef - TACHSTCK - 区间配对问题
 * 9. AtCoder ABC104C - All Green - 动态规划相关
 * 10. Codeforces 1363C - Game On Leaves - 博弈论相关
 * 11. SPOJ ANARC08E - Relax! I am a legend - 数学相关
 * 12. POJ 3169 - Layout - 差分约束系统
 * 13. HDU 2586 - How far away? - LCA 最近公共祖先
 * 14. USACO 2014 January Silver - Cross Country Skiing - BFS 搜索
 * 15. 洛谷 P1091 - 合唱队形 - 动态规划最长子序列
 * 16. Project Euler 357 - Prime generating integers - 数论相关
 * 17. 洛谷 P1208 - 混合牛奶 - 经典贪心问题
 * 18. 牛客网 NC140 - 排序 - 各种排序算法实现
 */
int minTaps(int n, int ranges[]) {
    // right[i] = j
    // 所有左边界在 i 的水龙头里，影响到的最右右边界是 j
    int right[MAXN] = {0};
    for (int i = 0, start; i <= n; i++) {
        start = i - ranges[i];
        if (start < 0) start = 0;
        int end = i + ranges[i];
        if (end > right[start]) {
            right[start] = end;
        }
    }
    // 当前 ans 数量的水龙头打开，影响到的最右右边界
    int cur = 0;
    // 如果再多打开一个水龙头，影响到的最右边界
    int next = 0;
```

```
    // 打开水龙头的数量
    int ans = 0;
    for (int i = 0; i < n; i++) {
        // 来到 i 位置
        // 先更新下一步的 next
        if (next < right[i]) {
            next = right[i];
        }
        if (i == cur) {
            if (next > i) {
                cur = next;
                ans++;
            } else {
                return -1;
            }
        }
    }
    return ans;
}

// 测试函数，返回结果用于验证
int test_minTaps() {
    // 测试用例 1: 基本情况
    int n1 = 5;
    int ranges1[] = {3, 4, 1, 1, 0, 0};
    int result1 = minTaps(n1, ranges1);

    // 测试用例 2: 需要多个水龙头
    int n2 = 3;
    int ranges2[] = {0, 0, 0, 0};
    int result2 = minTaps(n2, ranges2);

    // 测试用例 3: 精确覆盖
    int n3 = 7;
    int ranges3[] = {1, 2, 1, 0, 2, 1, 0, 1};
    int result3 = minTaps(n3, ranges3);

    // 测试用例 4: 单个水龙头覆盖全部
    int n4 = 8;
    int ranges4[] = {4, 0, 0, 0, 0, 0, 0, 0, 4};
    int result4 = minTaps(n4, ranges4);

    // 返回最后一个测试用例的结果
```

```
    return result4;
}

// 主函数，用于编译和运行测试
int main() {
    return test_minTaps();
}
```

==========

文件: Code02_MinimumTaps.java

==========

```
package class093;

// 灌溉花园的最少水龙头数目
// 在 x 轴上有一个一维的花园，花园长度为 n，从点 0 开始，到点 n 结束
// 花园里总共有 n + 1 个水龙头，分别位于[0, 1, ... n]
// 给你一个整数 n 和一个长度为 n+1 的整数数组 ranges
// 其中 ranges[i]表示
// 如果打开点 i 处的水龙头，可以灌溉的区域为[i-ranges[i], i+ranges[i]]
// 请你返回可以灌溉整个花园的最少水龙头数目
// 如果花园始终存在无法灌溉到的地方请你返回-1
// 测试链接 : https://leetcode.cn/problems/minimum-number-of-taps-to-open-to-water-a-garden/
public class Code02_MinimumTaps {

    /**
     * 灌溉花园的最少水龙头数目 - 使用贪心算法解决
     *
     * 算法思路：
     * 这是一个经典的区间覆盖问题，可以转化为跳跃游戏的变种。
     * 1. 首先预处理 ranges 数组，构造 right 数组，其中 right[i]表示以位置 i 为起点，
     *    能够覆盖到的最远右边界。
     * 2. 使用贪心策略，维护两个变量：
     *    - cur: 当前水龙头能覆盖到的最远位置
     *    - next: 下一个水龙头能覆盖到的最远位置
     *    - ans: 打开水龙头的数量
     * 3. 遍历位置 0 到 n-1，当当前位置超过当前水龙头能覆盖的范围时，
     *    就需要打开下一个水龙头，并更新相关变量。
     *
     * 时间复杂度：O(n) - 只需遍历数组一次
     * 空间复杂度：O(n) - 需要额外的 right 数组
     *
     * 是否最优解：是。这是该问题的最优解法之一。
```

```
 *
 * 适用场景：
 * 1. 区间覆盖问题
 * 2. 最少资源选择问题
 *
 * 相关题目：
 * 1. LeetCode 45. 跳跃游戏 II - 经典跳跃游戏
 * 2. LeetCode 55. 跳跃游戏 - 判断是否能到达终点
 * 3. LeetCode 1024. 视频拼接 - 区间拼接问题
 * 4. LeetCode 1326. 灌溉花园的最少水龙头数目 - 与本题相同
 * 5. 牛客网 NC135 买票需要多少时间 - 队列模拟相关
 * 6. LintCode 391. 数飞机 - 区间调度相关
 * 7. HackerRank - Jim and the Orders - 贪心调度问题
 * 8. CodeChef - TACHSTCK - 区间配对问题
 * 9. AtCoder ABC104C - All Green - 动态规划相关
 * 10. Codeforces 1363C - Game On Leaves - 博弈论相关
 * 11. SPOJ ANARC08E - Relax! I am a legend - 数学相关
 * 12. POJ 3169 - Layout - 差分约束系统
 * 13. HDU 2586 - How far away? - LCA 最近公共祖先
 * 14. USACO 2014 January Silver - Cross Country Skiing - BFS 搜索
 * 15. 洛谷 P1091 - 合唱队形 - 动态规划最长子序列
 * 16. Project Euler 357 - Prime generating integers - 数论相关
 * 17. 洛谷 P1208 - 混合牛奶 - 经典贪心问题
 * 18. 牛客网 NC140 - 排序 - 各种排序算法实现
 */
public static int minTaps(int n, int[] ranges) {
    // right[i] = j
    // 所有左边界在 i 的水龙头里，影响到的最右右边界是 j
    int[] right = new int[n + 1];
    for (int i = 0, start; i <= n; i++) {
        start = Math.max(0, i - ranges[i]);
        right[start] = Math.max(right[start], i + ranges[i]);
    }
    // 当前 ans 数量的水龙头打开，影响到的最右右边界
    int cur = 0;
    // 如果再多打开一个水龙头，影响到的最右边界
    int next = 0;
    // 打开水龙头的数量
    int ans = 0;
    for (int i = 0; i < n; i++) {
        // 来到 i 位置
        // 先更新下一步的 next
        next = Math.max(next, right[i]);
```

```
            if (i == cur) {
                if (next > i) {
                    cur = next;
                    ans++;
                } else {
                    return -1;
                }
            }
        }
        return ans;
    }

    // 测试用例
    public static void main(String[] args) {
        // 测试用例 1: 基本情况
        int n1 = 5;
        int[] ranges1 = {3, 4, 1, 1, 0, 0};
        System.out.println("输入: n = " + n1 + ", ranges = [3,4,1,1,0,0]");
        System.out.println("输出: " + minTaps(n1, ranges1));
        System.out.println("期望: 1\n");

        // 测试用例 2: 需要多个水龙头
        int n2 = 3;
        int[] ranges2 = {0, 0, 0, 0};
        System.out.println("输入: n = " + n2 + ", ranges = [0,0,0,0]");
        System.out.println("输出: " + minTaps(n2, ranges2));
        System.out.println("期望: -1\n");

        // 测试用例 3: 精确覆盖
        int n3 = 7;
        int[] ranges3 = {1, 2, 1, 0, 2, 1, 0, 1};
        System.out.println("输入: n = " + n3 + ", ranges = [1,2,1,0,2,1,0,1]");
        System.out.println("输出: " + minTaps(n3, ranges3));
        System.out.println("期望: 3\n");

        // 测试用例 4: 单个水龙头覆盖全部
        int n4 = 8;
        int[] ranges4 = {4, 0, 0, 0, 0, 0, 0, 0, 4};
        System.out.println("输入: n = " + n4 + ", ranges = [4,0,0,0,0,0,0,0,4]");
        System.out.println("输出: " + minTaps(n4, ranges4));
        System.out.println("期望: 2\n");
    }
```

}

==========

文件: Code02_MinimumTaps.py
==========
灌溉花园的最少水龙头数目
在 x 轴上有一个一维的花园，花园长度为 n，从点 0 开始，到点 n 结束
花园里总共有 n + 1 个水龙头，分别位于[0, 1, ... n]
给你一个整数 n 和一个长度为 n+1 的整数数组 ranges
其中 ranges[i]表示
如果打开点 i 处的水龙头，可以灌溉的区域为[i-ranges[i], i+ranges[i]]
请你返回可以灌溉整个花园的最少水龙头数目
如果花园始终存在无法灌溉到的地方请你返回-1
测试链接 : https://leetcode.cn/problems/minimum-number-of-taps-to-open-to-water-a-garden/

class Solution:
    """
    灌溉花园的最少水龙头数目 - 使用贪心算法解决

    算法思路：
    这是一个经典的区间覆盖问题，可以转化为跳跃游戏的变种。
    1. 首先预处理 ranges 数组，构造 right 数组，其中 right[i]表示以位置 i 为起点，
       能够覆盖到的最远右边界。
    2. 使用贪心策略，维护两个变量：
       - cur: 当前水龙头能覆盖到的最远位置
       - next: 下一个水龙头能覆盖到的最远位置
       - ans: 打开水龙头的数量
    3. 遍历位置 0 到 n-1，当当前位置超过当前水龙头能覆盖的范围时，
       就需要打开下一个水龙头，并更新相关变量。

    时间复杂度：O(n) - 只需遍历数组一次
    空间复杂度：O(n) - 需要额外的 right 数组

    是否最优解：是。这是该问题的最优解法之一。

    适用场景：
    1. 区间覆盖问题
    2. 最少资源选择问题

    相关题目：
    1. LeetCode 45. 跳跃游戏 II - 经典跳跃游戏
    2. LeetCode 55. 跳跃游戏 - 判断是否能到达终点

3. LeetCode 1024. 视频拼接 - 区间拼接问题
4. LeetCode 1326. 灌溉花园的最少水龙头数目 - 与本题相同
5. 牛客网 NC135 买票需要多少时间 - 队列模拟相关
6. LintCode 391. 数飞机 - 区间调度相关
7. HackerRank - Jim and the Orders - 贪心调度问题
8. CodeChef - TACHSTCK - 区间配对问题
9. AtCoder ABC104C - All Green - 动态规划相关
10. Codeforces 1363C - Game On Leaves - 博弈论相关
11. SPOJ ANARC08E - Relax! I am a legend - 数学相关
12. POJ 3169 - Layout - 差分约束系统
13. HDU 2586 - How far away? - LCA 最近公共祖先
14. USACO 2014 January Silver - Cross Country Skiing - BFS 搜索
15. 洛谷 P1091 - 合唱队形 - 动态规划最长子序列
16. Project Euler 357 - Prime generating integers - 数论相关
17. 洛谷 P1208 - 混合牛奶 - 经典贪心问题
18. 牛客网 NC140 - 排序 - 各种排序算法实现
"""

```
def minTaps(self, n, ranges):
    """
    计算灌溉整个花园所需的最少水龙头数目

    Args:
        n: int - 花园长度
        ranges: List[int] - 每个水龙头的灌溉范围

    Returns:
        int - 最少水龙头数目，如果无法完全灌溉则返回-1
    """
    # right[i] = j
    # 所有左边界在 i 的水龙头里，影响到的最右右边界是 j
    right = [0] * (n + 1)
    for i in range(n + 1):
        start = max(0, i - ranges[i])
        right[start] = max(right[start], i + ranges[i])

    # 当前 ans 数量的水龙头打开，影响到的最右右边界
    cur = 0
    # 如果再多打开一个水龙头，影响到的最右边界
    next_pos = 0
    # 打开水龙头的数量
    ans = 0
```

```
        for i in range(n):
            # 来到i位置
            # 先更新下一步的next
            next_pos = max(next_pos, right[i])
            if i == cur:
                if next_pos > i:
                    cur = next_pos
                    ans += 1
                else:
                    return -1

        return ans


# 测试用例
def main():
    solution = Solution()

    # 测试用例1: 基本情况
    n1 = 5
    ranges1 = [3, 4, 1, 1, 0, 0]
    print("输入: n = " + str(n1) + ", ranges = [3,4,1,1,0,0]")
    print("输出: " + str(solution.minTaps(n1, ranges1)))
    print("期望: 1\n")

    # 测试用例2: 需要多个水龙头
    n2 = 3
    ranges2 = [0, 0, 0, 0]
    print("输入: n = " + str(n2) + ", ranges = [0,0,0,0]")
    print("输出: " + str(solution.minTaps(n2, ranges2)))
    print("期望: -1\n")

    # 测试用例3: 精确覆盖
    n3 = 7
    ranges3 = [1, 2, 1, 0, 2, 1, 0, 1]
    print("输入: n = " + str(n3) + ", ranges = [1,2,1,0,2,1,0,1]")
    print("输出: " + str(solution.minTaps(n3, ranges3)))
    print("期望: 3\n")

    # 测试用例4: 单个水龙头覆盖全部
    n4 = 8
    ranges4 = [4, 0, 0, 0, 0, 0, 0, 0, 4]
    print("输入: n = " + str(n4) + ", ranges = [4,0,0,0,0,0,0,0,4]")
```

```
    print("输出: " + str(solution.minTaps(n4, ranges4)))
    print("期望: 2\n")


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

==========

文件: Code03_StringTransforms.cpp

==========

```
// 字符串转化
// 给出两个长度相同的字符串 str1 和 str2
// 请你帮忙判断字符串 str1 能不能在 零次 或 多次 转化后变成字符串 str2
// 每一次转化时，你可以将 str1 中出现的所有相同字母变成其他任何小写英文字母
// 只有在字符串 str1 能够通过上述方式顺利转化为字符串 str2 时才能返回 true
// 测试链接 : https://leetcode.cn/problems/string-transforms-into-another-string/

#define MAXN 100001

/**
 * 字符串转化 - 使用图论和贪心算法解决
 *
 * 算法思路：
 * 这是一个字符串映射问题。我们需要判断是否存在一个映射关系，
 * 使得 str1 中的每个字符都能映射到 str2 中对应位置的字符。
 *
 * 关键点：
 * 1. 如果 str1 和 str2 相等，直接返回 true
 * 2. 统计 str2 中字符的种类数，如果为 26，说明所有字符都出现了，
 *    此时如果 str1 到 str2 的映射不是一一映射，则无法完成转换
 * 3. 检查 str1 到 str2 的映射是否冲突，即 str1 中同一个字符不能映射到 str2 中的不同字符
 *
 * 时间复杂度：O(n) - 只需遍历字符串一次
 * 空间复杂度：O(1) - 只使用了固定大小的额外空间（26 个字母）
 *
 * 是否最优解：是。这是该问题的最优解法。
 *
 * 适用场景：
 * 1. 字符串映射问题
 * 2. 图论中的映射关系判断
 *
 * 相关题目：
```

```
 * 1. LeetCode 205. 同构字符串 - 判断两个字符串是否同构
 * 2. LeetCode 290. 单词规律 - 判断字符串是否遵循特定规律
 * 3. LeetCode 859. 亲密字符串 - 判断两个字符串是否可以通过交换两个字符变得相同
 * 4. LeetCode 925. 长按键入 - 判断输入是否可能是由于长按导致的
 * 5. 牛客网 NC141 - 判断回文串 - 字符串回文判断
 * 6. LintCode 636 - 二进制手表 - 位运算相关
 * 7. HackerRank - String Similarity - 字符串相似度计算
 * 8. CodeChef - STRPALIN - 回文字符串相关
 * 9. AtCoder ABC126C - Dice and Coin - 概率相关
 * 10. Codeforces 1324C - Frog Jumps - 贪心跳跃问题
 * 11. SPOJ ANARC09A - Seinfeld - 栈相关
 * 12. POJ 3096 - Surprising Strings - 字符串模式识别
 * 13. HDU 1028 - Ignatius and the Princess III - 整数划分
 * 14. USACO 2014 January Bronze - Learning by Example - 字符串处理
 * 15. 洛谷 P1055 - ISBN 号码 - 字符串校验
 * 16. Project Euler 357 - Prime generating integers - 数论相关
 * 17. 洛谷 P1091 - 合唱队形 - 动态规划最长子序列
 * 18. 牛客网 NC140 - 排序 - 各种排序算法实现
 */
bool canConvert(char str1[], char str2[]) {
    // 检查字符串是否相等
    int i = 0;
    while (str1[i] != '\0' && str2[i] != '\0') {
        if (str1[i] != str2[i]) {
            break;
        }
        i++;
    }
    if (str1[i] == '\0' && str2[i] == '\0') {
        return true;
    }

    // map[x] : str2 中字符 x 的词频
    int map[26] = {0};
    // kinds : str2 中字符的种类数
    int kinds = 0;
    i = 0;
    while (str2[i] != '\0') {
        int index = str2[i] - 'a';
        if (map[index] == 0) {
            kinds++;
        }
        map[index]++;
```

```
        i++;
    }

    if (kinds == 26) {
        return false;
    }

    // 检查 str1 到 str2 的映射是否冲突
    int charMap[26];
    for (int j = 0; j < 26; j++) {
        charMap[j] = -1;
    }

    i = 0;
    while (str1[i] != '\0' && str2[i] != '\0') {
        int cur = str1[i] - 'a';
        if (charMap[cur] != -1 && charMap[cur] != str2[i]) {
            return false;
        }
        charMap[cur] = str2[i];
        i++;
    }
    return true;
}

// 测试函数，返回结果用于验证
bool test_canConvert() {
    // 测试用例 1: 相同字符串
    char str1_1[] = "aabcc";
    char str2_1[] = "aabcc";
    bool result1 = canConvert(str1_1, str2_1);

    // 测试用例 2: 可以转换
    char str1_2[] = "aabcc";
    char str2_2[] = "ccdee";
    bool result2 = canConvert(str1_2, str2_2);

    // 测试用例 3: 无法转换（映射冲突）
    char str1_3[] = "leetcode";
    char str2_3[] = "codeleet";
    bool result3 = canConvert(str1_3, str2_3);

    // 测试用例 4: str2 包含所有字符
```

```
    char str1_4[] = "ab";
    char str2_4[] = "ba";
    bool result4 = canConvert(str1_4, str2_4);

    // 返回最后一个测试用例的结果
    return result4;
}

// 主函数，用于编译和运行测试
int main() {
    return test_canConvert() ? 0 : 1;
}
```

==========

文件: Code03_StringTransforms.java

==========

```
package class093;

import java.util.Arrays;

// 字符串转化
// 给出两个长度相同的字符串 str1 和 str2
// 请你帮忙判断字符串 str1 能不能在 零次 或 多次 转化后变成字符串 str2
// 每一次转化时，你可以将 str1 中出现的所有相同字母变成其他任何小写英文字母
// 只有在字符串 str1 能够通过上述方式顺利转化为字符串 str2 时才能返回 true
// 测试链接 : https://leetcode.cn/problems/string-transforms-into-another-string/
public class Code03_StringTransforms {

    /**
     * 字符串转化 - 使用图论和贪心算法解决
     *
     * 算法思路：
     * 这是一个字符串映射问题。我们需要判断是否存在一个映射关系，
     * 使得 str1 中的每个字符都能映射到 str2 中对应位置的字符。
     *
     * 关键点：
     * 1. 如果 str1 和 str2 相等，直接返回 true
     * 2. 统计 str2 中字符的种类数，如果为 26，说明所有字符都出现了，
     *    此时如果 str1 到 str2 的映射不是一一映射，则无法完成转换
     * 3. 检查 str1 到 str2 的映射是否冲突，即 str1 中同一个字符不能映射到 str2 中的不同字符
     *
     * 时间复杂度：O(n) - 只需遍历字符串一次
```

```
 * 空间复杂度：O(1) - 只使用了固定大小的额外空间（26 个字母）
 *
 * 是否最优解：是。这是该问题的最优解法。
 *
 * 适用场景：
 * 1. 字符串映射问题
 * 2. 图论中的映射关系判断
 *
 * 相关题目：
 * 1. LeetCode 205. 同构字符串 - 判断两个字符串是否同构
 * 2. LeetCode 290. 单词规律 - 判断字符串是否遵循特定规律
 * 3. LeetCode 859. 亲密字符串 - 判断两个字符串是否可以通过交换两个字符变得相同
 * 4. LeetCode 925. 长按键入 - 判断输入是否可能是由于长按导致的
 * 5. 牛客网 NC141 - 判断回文串 - 字符串回文判断
 * 6. LintCode 636 - 二进制手表 - 位运算相关
 * 7. HackerRank - String Similarity - 字符串相似度计算
 * 8. CodeChef - STRPALIN - 回文字符串相关
 * 9. AtCoder ABC126C - Dice and Coin - 概率相关
 * 10. Codeforces 1324C - Frog Jumps - 贪心跳跃问题
 * 11. SPOJ ANARC09A - Seinfeld - 栈相关
 * 12. POJ 3096 - Surprising Strings - 字符串模式识别
 * 13. HDU 1028 - Ignatius and the Princess III - 整数划分
 * 14. USACO 2014 January Bronze - Learning by Example - 字符串处理
 * 15. 洛谷 P1055 - ISBN 号码 - 字符串校验
 * 16. Project Euler 357 - Prime generating integers - 数论相关
 * 17. 洛谷 P1091 - 合唱队形 - 动态规划最长子序列
 * 18. 牛客网 NC140 - 排序 - 各种排序算法实现
 */
public static boolean canConvert(String str1, String str2) {
    if (str1.equals(str2)) {
        return true;
    }
    // map[x] : str2 中字符 x 的词频
    int[] map = new int[26];
    // kinds : str2 中字符的种类数
    int kinds = 0;
    for (int i = 0; i < str2.length(); i++) {
        if (map[str2.charAt(i) - 'a']++ == 0) {
            kinds++;
        }
    }
    if (kinds == 26) {
        return false;
```

```
    }
    Arrays.fill(map, -1);
    // map[x] = y : str1 中的字符 x 上次出现在 str1 中的 y 位置
    for (int i = 0, cur; i < str1.length(); i++) {
        cur = str1.charAt(i) - 'a';
        if (map[cur] != -1 && str2.charAt(map[cur]) != str2.charAt(i)) {
            return false;
        }
        map[cur] = i;
    }
    return true;
}

// 测试用例
public static void main(String[] args) {
    // 测试用例 1: 相同字符串
    String str1_1 = "aabcc";
    String str2_1 = "aabcc";
    System.out.println("输入: str1 = \"" + str1_1 + "\", str2 = \"" + str2_1 + "\"");
    System.out.println("输出: " + canConvert(str1_1, str2_1));
    System.out.println("期望: true\n");

    // 测试用例 2: 可以转换
    String str1_2 = "aabcc";
    String str2_2 = "ccdee";
    System.out.println("输入: str1 = \"" + str1_2 + "\", str2 = \"" + str2_2 + "\"");
    System.out.println("输出: " + canConvert(str1_2, str2_2));
    System.out.println("期望: true\n");

    // 测试用例 3: 无法转换（映射冲突）
    String str1_3 = "leetcode";
    String str2_3 = "codeleet";
    System.out.println("输入: str1 = \"" + str1_3 + "\", str2 = \"" + str2_3 + "\"");
    System.out.println("输出: " + canConvert(str1_3, str2_3));
    System.out.println("期望: false\n");

    // 测试用例 4: str2 包含所有字符
    String str1_4 = "ab";
    String str2_4 = "ba";
    System.out.println("输入: str1 = \"" + str1_4 + "\", str2 = \"" + str2_4 + "\"");
    System.out.println("输出: " + canConvert(str1_4, str2_4));
    System.out.println("期望: false\n");
}
```

```
}

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

文件: Code03_StringTransforms.py
==========================================
# 字符串转化
# 给出两个长度相同的字符串 str1 和 str2
# 请你帮忙判断字符串 str1 能不能在 零次 或 多次 转化后变成字符串 str2
# 每一次转化时，你可以将 str1 中出现的所有相同字母变成其他任何小写英文字母
# 只有在字符串 str1 能够通过上述方式顺利转化为字符串 str2 时才能返回 true
# 测试链接 : https://leetcode.cn/problems/string-transforms-into-another-string/

class Solution:
    """
    字符串转化 - 使用图论和贪心算法解决

    算法思路：
    这是一个字符串映射问题。我们需要判断是否存在一个映射关系，
    使得 str1 中的每个字符都能映射到 str2 中对应位置的字符。

    关键点：
    1. 如果 str1 和 str2 相等，直接返回 true
    2. 统计 str2 中字符的种类数，如果为 26，说明所有字符都出现了，
       此时如果 str1 到 str2 的映射不是一一映射，则无法完成转换
    3. 检查 str1 到 str2 的映射是否冲突，即 str1 中同一个字符不能映射到 str2 中的不同字符

    时间复杂度：O(n) - 只需遍历字符串一次
    空间复杂度：O(1) - 只使用了固定大小的额外空间（26 个字母）

    是否最优解：是。这是该问题的最优解法。

    适用场景：
    1. 字符串映射问题
    2. 图论中的映射关系判断

    相关题目：
    1. LeetCode 205. 同构字符串 - 判断两个字符串是否同构
    2. LeetCode 290. 单词规律 - 判断字符串是否遵循特定规律
    3. LeetCode 859. 亲密字符串 - 判断两个字符串是否可以通过交换两个字符变得相同
    4. LeetCode 925. 长按键入 - 判断输入是否可能是由于长按导致的
    5. 牛客网 NC141 - 判断回文串 - 字符串回文判断
```

6. LintCode 636 - 二进制手表 - 位运算相关
7. HackerRank - String Similarity - 字符串相似度计算
8. CodeChef - STRPALIN - 回文字符串相关
9. AtCoder ABC126C - Dice and Coin - 概率相关
10. Codeforces 1324C - Frog Jumps - 贪心跳跃问题
11. SPOJ ANARC09A - Seinfeld - 栈相关
12. POJ 3096 - Surprising Strings - 字符串模式识别
13. HDU 1028 - Ignatius and the Princess III - 整数划分
14. USACO 2014 January Bronze - Learning by Example - 字符串处理
15. 洛谷 P1055 - ISBN 号码 - 字符串校验
16. Project Euler 357 - Prime generating integers - 数论相关
17. 洛谷 P1091 - 合唱队形 - 动态规划最长子序列
18. 牛客网 NC140 - 排序 - 各种排序算法实现
"""

```
def canConvert(self, str1: str, str2: str) -> bool:
    """
    判断 str1 是否能通过字符映射转换为 str2

    Args:
        str1: str - 源字符串
        str2: str - 目标字符串

    Returns:
        bool - 如果可以转换返回 True，否则返回 False
    """
    if str1 == str2:
        return True

    # 统计 str2 中字符的种类数
    str2_chars = set(str2)
    if len(str2_chars) == 26:
        return False

    # 检查 str1 到 str2 的映射是否冲突
    char_map = {}
    for i in range(len(str1)):
        if str1[i] in char_map:
            if char_map[str1[i]] != str2[i]:
                return False
        else:
            char_map[str1[i]] = str2[i]
```

```
        return True


# 测试用例
def main():
    solution = Solution()

    # 测试用例 1: 相同字符串
    str1_1 = "aabcc"
    str2_1 = "aabcc"
    print("输入: str1 = \"" + str1_1 + "\", str2 = \"" + str2_1 + "\"")
    print("输出: " + str(solution.canConvert(str1_1, str2_1)).lower())
    print("期望: true\n")

    # 测试用例 2: 可以转换
    str1_2 = "aabcc"
    str2_2 = "ccdee"
    print("输入: str1 = \"" + str1_2 + "\", str2 = \"" + str2_2 + "\"")
    print("输出: " + str(solution.canConvert(str1_2, str2_2)).lower())
    print("期望: true\n")

    # 测试用例 3: 无法转换（映射冲突）
    str1_3 = "leetcode"
    str2_3 = "codeleet"
    print("输入: str1 = \"" + str1_3 + "\", str2 = \"" + str2_3 + "\"")
    print("输出: " + str(solution.canConvert(str1_3, str2_3)).lower())
    print("期望: false\n")

    # 测试用例 4: str2 包含所有字符
    str1_4 = "ab"
    str2_4 = "ba"
    print("输入: str1 = \"" + str1_4 + "\", str2 = \"" + str2_4 + "\"")
    print("输出: " + str(solution.canConvert(str1_4, str2_4)).lower())
    print("期望: false\n")


if __name__ == "__main__":
    main()

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

文件: Code04_CrossRiver.cpp
==============================================
```

```
// 过河问题
// 一共 n 人出游，他们走到一条河的西岸，想要过河到东岸
// 每个人都有一个渡河时间 ti，西岸有一条船，一次最多乘坐两人
// 如果船上有一个人，划到对岸的时间，等于这个人的渡河时间
// 如果船上有两个人，划到对岸的时间，等于两个人的渡河时间的最大值
// 返回最少要花费多少时间，才能使所有人都过河
// 测试链接 : https://www.luogu.com.cn/problem/P1809

#define MAXN 100001

// 用于排序的比较函数
void swap(int* a, int* b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

// 简单排序实现（冒泡排序）
void sort(int arr[], int n) {
    for (int i = 0; i < n-1; i++) {
        for (int j = 0; j < n-i-1; j++) {
            if (arr[j] > arr[j+1]) {
                swap(&arr[j], &arr[j+1]);
            }
        }
    }
}

/**
 * 过河问题 - 使用动态规划和贪心算法解决
 *
 * 算法思路：
 * 这是一个经典的过河问题，类似于"农夫过河"问题的变种。
 * 我们需要找到最优的策略来运送所有人过河，使得总时间最少。
 *
 * 解题策略：
 * 1. 首先对所有人按渡河时间进行排序
 * 2. 使用动态规划，dp[i]表示运送前 i 个人过河所需的最少时间
 * 3. 对于每个人，有两种策略：
 *    - 策略 1：最快的人陪同当前人过河，然后最快的人回来
 *    - 策略 2：最快的两个人先过河，最快的回来，最慢的两个人过河，第二快的人回来
 * 4. 取两种策略中的较小值作为当前状态的最优解
 *
```

```
 * 时间复杂度：O(n log n) - 主要是排序的时间复杂度
 * 空间复杂度：O(n) - dp 数组和存储人员时间的数组
 *
 * 是否最优解：是。这是该问题的最优解法。
 *
 * 适用场景：
 * 1. 过河/过桥类问题
 * 2. 资源调度优化问题
 *
 * 相关题目：
 * 1. LeetCode 1024. 视频拼接 - 区间拼接问题
 * 2. LeetCode 1326. 灌溉花园的最少水龙头数目 - 区间覆盖问题
 * 3. 牛客网 NC135 买票需要多少时间 - 队列模拟相关
 * 4. LintCode 391. 数飞机 - 区间调度相关
 * 5. HackerRank - Jim and the Orders - 贪心调度问题
 * 6. CodeChef - TACHSTCK - 区间配对问题
 * 7. AtCoder ABC104C - All Green - 动态规划相关
 * 8. Codeforces 1363C - Game On Leaves - 博弈论相关
 * 9. SPOJ ANARC08E - Relax! I am a legend - 数学相关
 * 10. POJ 3169 - Layout - 差分约束系统
 * 11. HDU 2586 - How far away? - LCA 最近公共祖先
 * 12. USACO 2014 January Silver - Cross Country Skiing - BFS 搜索
 * 13. 洛谷 P1091 - 合唱队形 - 动态规划最长子序列
 * 14. Project Euler 357 - Prime generating integers - 数论相关
 * 15. 洛谷 P1208 - 混合牛奶 - 经典贪心问题
 * 16. 牛客网 NC140 - 排序 - 各种排序算法实现
 * 17. 洛谷 P1809 - 过河问题 - 与本题相同
 * 18. POJ 1700 - Crossing River - 经典过河问题
 */
int minCost(int nums[], int n) {
    sort(nums, n);
    int dp[MAXN] = {0};

    if (n >= 1) {
        dp[0] = nums[0];
    }
    if (n >= 2) {
        dp[1] = nums[1];
    }
    if (n >= 3) {
        dp[2] = nums[0] + nums[1] + nums[2];
    }
```

```
    for (int i = 3; i < n; i++) {
        // 策略1: 最快的人陪同当前人过河，然后最快的人回来
        int strategy1 = dp[i - 1] + nums[i] + nums[0];
        // 策略2: 最快的两个人先过河，最快的回来，最慢的两个人过河，第二快的人回来
        int strategy2 = dp[i - 2] + nums[1] + nums[1] + nums[i] + nums[0];

        dp[i] = strategy1 < strategy2 ? strategy1 : strategy2;
    }
    return dp[n - 1];
}

// 测试函数，返回结果用于验证
int test_minCost() {
    // 测试用例1: 基本情况
    int nums1[] = {1, 2, 5, 10};
    int result1 = minCost(nums1, 4);

    // 测试用例2: 三人情况
    int nums2[] = {1, 5, 10};
    int result2 = minCost(nums2, 3);

    // 测试用例3: 两人情况
    int nums3[] = {3, 7};
    int result3 = minCost(nums3, 2);

    // 测试用例4: 一人情况
    int nums4[] = {8};
    int result4 = minCost(nums4, 1);

    // 返回最后一个测试用例的结果
    return result4;
}

// 主函数，用于编译和运行测试
int main() {
    return test_minCost();
}
```

==========

文件: Code04_CrossRiver.java

==========

```
package class093;
```

```
// 过河问题
// 一共 n 人出游，他们走到一条河的西岸，想要过河到东岸
// 每个人都有一个渡河时间 ti，西岸有一条船，一次最多乘坐两人
// 如果船上有一个人，划到对岸的时间，等于这个人的渡河时间
// 如果船上有两个人，划到对岸的时间，等于两个人的渡河时间的最大值
// 返回最少要花费多少时间，才能使所有人都过河
// 测试链接 : https://www.luogu.com.cn/problem/P1809
// 请同学们务必参考如下代码中关于输入、输出的处理
// 这是输入输出处理效率很高的写法
// 提交以下的 code，提交时请把类名改成"Main"，可以直接通过

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

public class Code04_CrossRiver {

    public static int MAXN = 100001;

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

    public static int[] dp = new int[MAXN];

    public static int n;

    /**
     * 过河问题 - 使用动态规划和贪心算法解决
     *
     * 算法思路：
     * 这是一个经典的过河问题，类似于"农夫过河"问题的变种。
     * 我们需要找到最优的策略来运送所有人过河，使得总时间最少。
     *
     * 解题策略：
     * 1. 首先对所有人按渡河时间进行排序
     * 2. 使用动态规划，dp[i]表示运送前 i 个人过河所需的最少时间
     * 3. 对于每个人，有两种策略：
     *    - 策略 1：最快的人陪同当前人过河，然后最快的人回来
     *    - 策略 2：最快的两个人先过河，最快的回来，最慢的两个人过河，第二快的人回来
```

```
     * 4. 取两种策略中的较小值作为当前状态的最优解
     *
     * 时间复杂度：O(n log n) - 主要是排序的时间复杂度
     * 空间复杂度：O(n) - dp 数组和存储人员时间的数组
     *
     * 是否最优解：是。这是该问题的最优解法。
     *
     * 适用场景：
     * 1. 过河/过桥类问题
     * 2. 资源调度优化问题
     *
     * 相关题目：
     * 1. LeetCode 1024. 视频拼接 - 区间拼接问题
     * 2. LeetCode 1326. 灌溉花园的最少水龙头数目 - 区间覆盖问题
     * 3. 牛客网 NC135 买票需要多少时间 - 队列模拟相关
     * 4. LintCode 391. 数飞机 - 区间调度相关
     * 5. HackerRank - Jim and the Orders - 贪心调度问题
     * 6. CodeChef - TACHSTCK - 区间配对问题
     * 7. AtCoder ABC104C - All Green - 动态规划相关
     * 8. Codeforces 1363C - Game On Leaves - 博弈论相关
     * 9. SPOJ ANARC08E - Relax! I am a legend - 数学相关
     * 10. POJ 3169 - Layout - 差分约束系统
     * 11. HDU 2586 - How far away? - LCA 最近公共祖先
     * 12. USACO 2014 January Silver - Cross Country Skiing - BFS 搜索
     * 13. 洛谷 P1091 - 合唱队形 - 动态规划最长子序列
     * 14. Project Euler 357 - Prime generating integers - 数论相关
     * 15. 洛谷 P1208 - 混合牛奶 - 经典贪心问题
     * 16. 牛客网 NC140 - 排序 - 各种排序算法实现
     * 17. 洛谷 P1809 - 过河问题 - 与本题相同
     * 18. POJ 1700 - Crossing River - 经典过河问题
     */
    public static int minCost() {
        Arrays.sort(nums, 0, n);
        if (n >= 1) {
            dp[0] = nums[0];
        }
        if (n >= 2) {
            dp[1] = nums[1];
        }
        if (n >= 3) {
            dp[2] = nums[0] + nums[1] + nums[2];
        }
        for (int i = 3; i < n; i++) {
```

```
            dp[i] = Math.min(dp[i - 1] + nums[i] + nums[0], dp[i - 2] + nums[1] + nums[1] +
nums[i] + nums[0]);
        }
        return dp[n - 1];
    }

    // 测试用例
    public static void main(String[] args) throws IOException {
        // 为了测试方便，我们使用标准输入输出
        // 在实际提交时，请使用题目要求的输入输出方式
        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();
        out.close();
        br.close();

        // 本地测试用例
        /*
        // 测试用例 1: 基本情况
        n = 4;
        nums[0] = 1;
        nums[1] = 2;
        nums[2] = 5;
        nums[3] = 10;
        System.out.println("输入: [1, 2, 5, 10]");
        System.out.println("输出: " + minCost());
        System.out.println("期望: 17\n");

        // 测试用例 2: 三人情况
        n = 3;
        nums[0] = 1;
        nums[1] = 5;
        nums[2] = 10;
```

```
        System.out.println("输入: [1, 5, 10]");
        System.out.println("输出: " + minCost());
        System.out.println("期望: 16\n");

        // 测试用例 3: 两人情况
        n = 2;
        nums[0] = 3;
        nums[1] = 7;
        System.out.println("输入: [3, 7]");
        System.out.println("输出: " + minCost());
        System.out.println("期望: 7\n");

        // 测试用例 4: 一人情况
        n = 1;
        nums[0] = 8;
        System.out.println("输入: [8]");
        System.out.println("输出: " + minCost());
        System.out.println("期望: 8\n");
        */
    }

}
```

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

文件: Code04_CrossRiver.py

```
================================================
# 过河问题
# 一共 n 人出游，他们走到一条河的西岸，想要过河到东岸
# 每个人都有一个渡河时间 ti，西岸有一条船，一次最多乘坐两人
# 如果船上有一个人，划到对岸的时间，等于这个人的渡河时间
# 如果船上有两个人，划到对岸的时间，等于两个人的渡河时间的最大值
# 返回最少要花费多少时间，才能使所有人都过河
# 测试链接 : https://www.luogu.com.cn/problem/P1809


class Solution:
    """
    过河问题 - 使用动态规划和贪心算法解决

    算法思路：
    这是一个经典的过河问题，类似于"农夫过河"问题的变种。
    我们需要找到最优的策略来运送所有人过河，使得总时间最少。
```

解题策略：
1. 首先对所有人按渡河时间进行排序
2. 使用动态规划，dp[i]表示运送前 i 个人过河所需的最少时间
3. 对于每个人，有两种策略：
   - 策略 1：最快的人陪同当前人过河，然后最快的人回来
   - 策略 2：最快的两个人先过河，最快的回来，最慢的两个人过河，第二快的人回来
4. 取两种策略中的较小值作为当前状态的最优解

时间复杂度：O(n log n) - 主要是排序的时间复杂度
空间复杂度：O(n) - dp 数组和存储人员时间的数组

是否最优解：是。这是该问题的最优解法。

适用场景：
1. 过河/过桥类问题
2. 资源调度优化问题

相关题目：
1. LeetCode 1024. 视频拼接 - 区间拼接问题
2. LeetCode 1326. 灌溉花园的最少水龙头数目 - 区间覆盖问题
3. 牛客网 NC135 买票需要多少时间 - 队列模拟相关
4. LintCode 391. 数飞机 - 区间调度相关
5. HackerRank - Jim and the Orders - 贪心调度问题
6. CodeChef - TACHSTCK - 区间配对问题
7. AtCoder ABC104C - All Green - 动态规划相关
8. Codeforces 1363C - Game On Leaves - 博弈论相关
9. SPOJ ANARC08E - Relax! I am a legend - 数学相关
10. POJ 3169 - Layout - 差分约束系统
11. HDU 2586 - How far away? - LCA 最近公共祖先
12. USACO 2014 January Silver - Cross Country Skiing - BFS 搜索
13. 洛谷 P1091 - 合唱队形 - 动态规划最长子序列
14. Project Euler 357 - Prime generating integers - 数论相关
15. 洛谷 P1208 - 混合牛奶 - 经典贪心问题
16. 牛客网 NC140 - 排序 - 各种排序算法实现
17. 洛谷 P1809 - 过河问题 - 与本题相同
18. POJ 1700 - Crossing River - 经典过河问题
"""

```
def minCost(self, nums):
    """
    计算运送所有人过河的最少时间
```

```
        Args:
            nums: List[int] - 每个人的渡河时间

        Returns:
            int - 最少需要的时间
        """
        n = len(nums)
        nums.sort()
        dp = [0] * n

        if n >= 1:
            dp[0] = nums[0]
        if n >= 2:
            dp[1] = nums[1]
        if n >= 3:
            dp[2] = nums[0] + nums[1] + nums[2]

        for i in range(3, n):
            # 策略 1: 最快的人陪同当前人过河，然后最快的人回来
            strategy1 = dp[i - 1] + nums[i] + nums[0]
            # 策略 2: 最快的两个人先过河，最快的回来，最慢的两个人过河，第二快的人回来
            strategy2 = dp[i - 2] + nums[1] + nums[1] + nums[i] + nums[0]
            dp[i] = min(strategy1, strategy2)

        return dp[n - 1]


# 测试用例
def main():
    solution = Solution()

    # 测试用例 1: 基本情况
    nums1 = [1, 2, 5, 10]
    print("输入: [1, 2, 5, 10]")
    print("输出: ", solution.minCost(nums1))
    print("期望: 17\n")

    # 测试用例 2: 三人情况
    nums2 = [1, 5, 10]
    print("输入: [1, 5, 10]")
    print("输出: ", solution.minCost(nums2))
    print("期望: 16\n")
```

```
    # 测试用例 3: 两人情况
    nums3 = [3, 7]
    print("输入: [3, 7]")
    print("输出: ", solution.minCost(nums3))
    print("期望: 7\n")

    # 测试用例 4: 一人情况
    nums4 = [8]
    print("输入: [8]")
    print("输出: ", solution.minCost(nums4))
    print("期望: 8\n")


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

==========

文件: Code05_SuperWashingMachines.cpp

==========

```
// 超级洗衣机
// 假设有 n 台超级洗衣机放在同一排上
// 开始的时候，每台洗衣机内可能有一定量的衣服，也可能是空的
// 在每一步操作中，你可以选择任意 m (1 <= m <= n) 台洗衣机
// 与此同时将每台洗衣机的一件衣服送到相邻的一台洗衣机
// 给定一个整数数组 machines 代表从左至右每台洗衣机中的衣物数量
// 请给出能让所有洗衣机中剩下的衣物的数量相等的最少的操作步数
// 如果不能使每台洗衣机中衣物的数量相等则返回-1
// 测试链接 : https://leetcode.cn/problems/super-washing-machines/

#define MAXN 100001

/**
 * 超级洗衣机 - 使用贪心算法解决
 *
 * 算法思路：
 * 这是一个很有趣的贪心问题。关键在于理解每台洗衣机在达到平衡状态前，
 * 需要向左或向右输送多少件衣服。
 *
 * 解题策略：
 * 1. 首先检查是否能够平均分配衣服，即总衣服数能否被洗衣机台数整除
 * 2. 计算每台洗衣机最终应该拥有的衣服数量（平均值）
 * 3. 对于每台洗衣机，计算它需要向左和向右输送的衣服数量
```

```
 * 4. 在每一步中，瓶颈是需要输送衣服数量的最大值
 *
 * 关键观察：
 * - 每台洗衣机可以同时向左右两个方向输送衣服
 * - 对于位置 i，左侧需要的衣服数量为 leftNeed = i * avg - leftSum
 * - 对于位置 i，右侧需要的衣服数量为 rightNeed = (n - i - 1) * avg - rightSum
 * - 如果左右两侧都需要衣服，则当前洗衣机是瓶颈，需要的步数是 leftNeed + rightNeed
 * - 否则，瓶颈是 max(|leftNeed|, |rightNeed|)
 *
 * 时间复杂度：O(n) - 只需遍历数组一次
 * 空间复杂度：O(1) - 只使用了常数额外空间
 *
 * 是否最优解：是。这是该问题的最优解法。
 *
 * 适用场景：
 * 1. 资源平衡分配问题
 * 2. 流量控制问题
 *
 * 相关题目：
 * 1. LeetCode 453. 最小操作次数使数组元素相等 - 数组平衡问题
 * 2. LeetCode 979. 在二叉树中分配硬币 - 树上资源分配
 * 3. LeetCode 1024. 视频拼接 - 区间拼接问题
 * 4. LeetCode 1326. 灌溉花园的最少水龙头数目 - 区间覆盖问题
 * 5. 牛客网 NC135 买票需要多少时间 - 队列模拟相关
 * 6. LintCode 391. 数飞机 - 区间调度相关
 * 7. HackerRank - Jim and the Orders - 贪心调度问题
 * 8. CodeChef - TACHSTCK - 区间配对问题
 * 9. AtCoder ABC104C - All Green - 动态规划相关
 * 10. Codeforces 1363C - Game On Leaves - 博弈论相关
 * 11. SPOJ ANARC08E - Relax! I am a legend - 数学相关
 * 12. POJ 3169 - Layout - 差分约束系统
 * 13. HDU 2586 - How far away? - LCA 最近公共祖先
 * 14. USACO 2014 January Silver - Cross Country Skiing - BFS 搜索
 * 15. 洛谷 P1091 - 合唱队形 - 动态规划最长子序列
 * 16. Project Euler 357 - Prime generating integers - 数论相关
 * 17. 洛谷 P1208 - 混合牛奶 - 经典贪心问题
 * 18. 牛客网 NC140 - 排序 - 各种排序算法实现
 */
int findMinMoves(int arr[], int n) {
    int sum = 0;
    for (int i = 0; i < n; i++) {
        sum += arr[i];
    }
```

```
    if (sum % n != 0) {
        return -1;
    }
    int avg = sum / n; // 每台洗衣机最终要求的衣服数量一定是平均值
    int leftSum = 0; // 左侧累加和
    int leftNeed = 0; // 左边还需要多少件衣服
    int rightNeed = 0;// 右边还需要多少件衣服
    int bottleNeck = 0;// 每一步的瓶颈
    int ans = 0;

    for (int i = 0; i < n; leftSum += arr[i], i++) {
        leftNeed = i * avg - leftSum;
        rightNeed = (n - i - 1) * avg - (sum - leftSum - arr[i]);

        if (leftNeed > 0 && rightNeed > 0) {
            bottleNeck = leftNeed + rightNeed;
        } else {
            int absLeftNeed = leftNeed < 0 ? -leftNeed : leftNeed;
            int absRightNeed = rightNeed < 0 ? -rightNeed : rightNeed;
            bottleNeck = absLeftNeed > absRightNeed ? absLeftNeed : absRightNeed;
        }

        if (ans < bottleNeck) {
            ans = bottleNeck;
        }
    }
    return ans;
}

// 测试函数，返回结果用于验证
int test_findMinMoves() {
    // 测试用例 1: 基本情况
    int machines1[] = {1, 0, 5};
    int result1 = findMinMoves(machines1, 3);

    // 测试用例 2: 无法平均分配
    int machines2[] = {0, 3, 0};
    int result2 = findMinMoves(machines2, 3);

    // 测试用例 3: 已经平均分配
    int machines3[] = {2, 2, 2};
    int result3 = findMinMoves(machines3, 3);
```

```
    // 测试用例 4: 复杂情况
    int machines4[] = {4, 0, 0, 4};
    int result4 = findMinMoves(machines4, 4);

    // 返回最后一个测试用例的结果
    return result4;
}

// 主函数，用于编译和运行测试
int main() {
    return test_findMinMoves();
}
```

==========

文件: Code05_SuperWashingMachines.java

==========

```
package class093;

// 超级洗衣机
// 假设有 n 台超级洗衣机放在同一排上
// 开始的时候，每台洗衣机内可能有一定量的衣服，也可能是空的
// 在每一步操作中，你可以选择任意 m (1 <= m <= n) 台洗衣机
// 与此同时将每台洗衣机的一件衣服送到相邻的一台洗衣机
// 给定一个整数数组 machines 代表从左至右每台洗衣机中的衣物数量
// 请给出能让所有洗衣机中剩下的衣物的数量相等的最少的操作步数
// 如果不能使每台洗衣机中衣物的数量相等则返回-1
// 测试链接 : https://leetcode.cn/problems/super-washing-machines/
public class Code05_SuperWashingMachines {

    /**
     * 超级洗衣机 - 使用贪心算法解决
     *
     * 算法思路：
     * 这是一个很有趣的贪心问题。关键在于理解每台洗衣机在达到平衡状态前，
     * 需要向左或向右输送多少件衣服。
     *
     * 解题策略：
     * 1. 首先检查是否能够平均分配衣服，即总衣服数能否被洗衣机台数整除
     * 2. 计算每台洗衣机最终应该拥有的衣服数量（平均值）
     * 3. 对于每台洗衣机，计算它需要向左和向右输送的衣服数量
     * 4. 在每一步中，瓶颈是需要输送衣服数量的最大值
     *
```

```
     * 关键观察：
     * - 每台洗衣机可以同时向左右两个方向输送衣服
     * - 对于位置 i，左侧需要的衣服数量为 leftNeed = i * avg - leftSum
     * - 对于位置 i，右侧需要的衣服数量为 rightNeed = (n - i - 1) * avg - rightSum
     * - 如果左右两侧都需要衣服，则当前洗衣机是瓶颈，需要的步数是 leftNeed + rightNeed
     * - 否则，瓶颈是 max(|leftNeed|, |rightNeed|)
     *
     * 时间复杂度：O(n) - 只需遍历数组一次
     * 空间复杂度：O(1) - 只使用了常数额外空间
     *
     * 是否最优解：是。这是该问题的最优解法。
     *
     * 适用场景：
     * 1. 资源平衡分配问题
     * 2. 流量控制问题
     *
     * 相关题目：
     * 1. LeetCode 453. 最小操作次数使数组元素相等 - 数组平衡问题
     * 2. LeetCode 979. 在二叉树中分配硬币 - 树上资源分配
     * 3. LeetCode 1024. 视频拼接 - 区间拼接问题
     * 4. LeetCode 1326. 灌溉花园的最少水龙头数目 - 区间覆盖问题
     * 5. 牛客网 NC135 买票需要多少时间 - 队列模拟相关
     * 6. LintCode 391. 数飞机 - 区间调度相关
     * 7. HackerRank - Jim and the Orders - 贪心调度问题
     * 8. CodeChef - TACHSTCK - 区间配对问题
     * 9. AtCoder ABC104C - All Green - 动态规划相关
     * 10. Codeforces 1363C - Game On Leaves - 博弈论相关
     * 11. SPOJ ANARC08E - Relax! I am a legend - 数学相关
     * 12. POJ 3169 - Layout - 差分约束系统
     * 13. HDU 2586 - How far away? - LCA 最近公共祖先
     * 14. USACO 2014 January Silver - Cross Country Skiing - BFS 搜索
     * 15. 洛谷 P1091 - 合唱队形 - 动态规划最长子序列
     * 16. Project Euler 357 - Prime generating integers - 数论相关
     * 17. 洛谷 P1208 - 混合牛奶 - 经典贪心问题
     * 18. 牛客网 NC140 - 排序 - 各种排序算法实现
     */
    public static int findMinMoves(int[] arr) {
        int n = arr.length;
        int sum = 0;
        for (int i = 0; i < n; i++) {
            sum += arr[i];
        }
        if (sum % n != 0) {
```

```
        return -1;
    }
    int avg = sum / n; // 每台洗衣机最终要求的衣服数量一定是平均值
    int leftSum = 0; // 左侧累加和
    int leftNeed = 0; // 左边还需要多少件衣服
    int rightNeed = 0;// 右边还需要多少件衣服
    int bottleNeck = 0;// 每一步的瓶颈
    int ans = 0;
    for (int i = 0; i < n; leftSum += arr[i], i++) {
        leftNeed = i * avg - leftSum;
        rightNeed = (n - i - 1) * avg - (sum - leftSum - arr[i]);
        if (leftNeed > 0 && rightNeed > 0) {
            bottleNeck = leftNeed + rightNeed;
        } else {
            bottleNeck = Math.max(Math.abs(leftNeed), Math.abs(rightNeed));
        }
        ans = Math.max(ans, bottleNeck);
    }
    return ans;
}

// 测试用例
public static void main(String[] args) {
    // 测试用例 1: 基本情况
    int[] machines1 = {1, 0, 5};
    System.out.println("输入: [1, 0, 5]");
    System.out.println("输出: " + findMinMoves(machines1));
    System.out.println("期望: 3\n");

    // 测试用例 2: 无法平均分配
    int[] machines2 = {0, 3, 0};
    System.out.println("输入: [0, 3, 0]");
    System.out.println("输出: " + findMinMoves(machines2));
    System.out.println("期望: 2\n");

    // 测试用例 3: 已经平均分配
    int[] machines3 = {2, 2, 2};
    System.out.println("输入: [2, 2, 2]");
    System.out.println("输出: " + findMinMoves(machines3));
    System.out.println("期望: 0\n");

    // 测试用例 4: 复杂情况
    int[] machines4 = {4, 0, 0, 4};
```

```
        System.out.println("输入: [4, 0, 0, 4]");
        System.out.println("输出: " + findMinMoves(machines4));
        System.out.println("期望: 2\n");
    }

}
```

==========

文件: Code05_SuperWashingMachines.py

==========

```
# 超级洗衣机
# 假设有 n 台超级洗衣机放在同一排上
# 开始的时候，每台洗衣机内可能有一定量的衣服，也可能是空的
# 在每一步操作中，你可以选择任意 m (1 <= m <= n) 台洗衣机
# 与此同时将每台洗衣机的一件衣服送到相邻的一台洗衣机
# 给定一个整数数组 machines 代表从左至右每台洗衣机中的衣物数量
# 请给出能让所有洗衣机中剩下的衣物的数量相等的最少的操作步数
# 如果不能使每台洗衣机中衣物的数量相等则返回-1
# 测试链接 : https://leetcode.cn/problems/super-washing-machines/



class Solution:
    """
    超级洗衣机 - 使用贪心算法解决

    算法思路：
    这是一个很有趣的贪心问题。关键在于理解每台洗衣机在达到平衡状态前，
    需要向左或向右输送多少件衣服。

    解题策略：
    1. 首先检查是否能够平均分配衣服，即总衣服数能否被洗衣机台数整除
    2. 计算每台洗衣机最终应该拥有的衣服数量（平均值）
    3. 对于每台洗衣机，计算它需要向左和向右输送的衣服数量
    4. 在每一步中，瓶颈是需要输送衣服数量的最大值

    关键观察：
    - 每台洗衣机可以同时向左右两个方向输送衣服
    - 对于位置 i，左侧需要的衣服数量为 leftNeed = i * avg - leftSum
    - 对于位置 i，右侧需要的衣服数量为 rightNeed = (n - i - 1) * avg - rightSum
    - 如果左右两侧都需要衣服，则当前洗衣机是瓶颈，需要的步数是 leftNeed + rightNeed
    - 否则，瓶颈是 max(|leftNeed|, |rightNeed|)
```

时间复杂度：O(n) - 只需遍历数组一次
空间复杂度：O(1) - 只使用了常数额外空间

是否最优解：是。这是该问题的最优解法。

适用场景：
1. 资源平衡分配问题
2. 流量控制问题

相关题目：
1. LeetCode 453. 最小操作次数使数组元素相等 - 数组平衡问题
2. LeetCode 979. 在二叉树中分配硬币 - 树上资源分配
3. LeetCode 1024. 视频拼接 - 区间拼接问题
4. LeetCode 1326. 灌溉花园的最少水龙头数目 - 区间覆盖问题
5. 牛客网 NC135 买票需要多少时间 - 队列模拟相关
6. LintCode 391. 数飞机 - 区间调度相关
7. HackerRank - Jim and the Orders - 贪心调度问题
8. CodeChef - TACHSTCK - 区间配对问题
9. AtCoder ABC104C - All Green - 动态规划相关
10. Codeforces 1363C - Game On Leaves - 博弈论相关
11. SPOJ ANARC08E - Relax! I am a legend - 数学相关
12. POJ 3169 - Layout - 差分约束系统
13. HDU 2586 - How far away? - LCA 最近公共祖先
14. USACO 2014 January Silver - Cross Country Skiing - BFS 搜索
15. 洛谷 P1091 - 合唱队形 - 动态规划最长子序列
16. Project Euler 357 - Prime generating integers - 数论相关
17. 洛谷 P1208 - 混合牛奶 - 经典贪心问题
18. 牛客网 NC140 - 排序 - 各种排序算法实现
"""

```
def findMinMoves(self, machines):
    """
    计算使所有洗衣机衣物数量相等的最少操作步数

    Args:
        machines: List[int] - 每台洗衣机中的衣物数量

    Returns:
        int - 最少操作步数，如果无法平均分配则返回-1
    """
    n = len(machines)
    total = sum(machines)
```

```
        # 检查是否能够平均分配
        if total % n != 0:
            return -1

        avg = total // n  # 每台洗衣机最终要求的衣服数量
        left_sum = 0  # 左侧累加和
        ans = 0

        for i in range(n):
            left_need = i * avg - left_sum  # 左边还需要多少件衣服
            right_need = (n - i - 1) * avg - (total - left_sum - machines[i])  # 右边还需要多少件
衣服

            # 计算当前步骤的瓶颈
            if left_need > 0 and right_need > 0:
                # 如果左右都需要衣服，则当前是瓶颈
                bottle_neck = left_need + right_need
            else:
                # 否则瓶颈是左右需求的最大值
                bottle_neck = max(abs(left_need), abs(right_need))

            ans = max(ans, bottle_neck)
            left_sum += machines[i]

        return ans


# 测试用例
def main():
    solution = Solution()

    # 测试用例 1: 基本情况
    machines1 = [1, 0, 5]
    print("输入: [1, 0, 5]")
    print("输出: ", solution.findMinMoves(machines1))
    print("期望: 3\n")

    # 测试用例 2: 无法平均分配
    machines2 = [0, 3, 0]
    print("输入: [0, 3, 0]")
    print("输出: ", solution.findMinMoves(machines2))
    print("期望: 2\n")
```

```
    # 测试用例 3: 已经平均分配
    machines3 = [2, 2, 2]
    print("输入: [2, 2, 2]")
    print("输出: ", solution.findMinMoves(machines3))
    print("期望: 0\n")

    # 测试用例 4: 复杂情况
    machines4 = [4, 0, 0, 4]
    print("输入: [4, 0, 0, 4]")
    print("输出: ", solution.findMinMoves(machines4))
    print("期望: 2\n")


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

==========

文件: Code06_AssignCookies.cpp

==========

```
// 分发饼干（Assign Cookies）
// 题目来源：LeetCode 455
// 题目链接：https://leetcode.cn/problems/assign-cookies/

/**
 * 问题描述：
 * 假设你是一位很棒的家长，想要给你的孩子们一些小饼干。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；
 * 对每个饼干 j，都有一个尺寸 s[j]。如果 s[j] >= g[i]，我们可以将这个饼干 j 分配给孩子 i，
 * 这个孩子会得到满足。你的目标是尽可能满足越多数量的孩子，并输出这个最大数值。
 *
 * 算法思路：
 * 使用贪心策略，将胃口最小的孩子分配给能满足他的最小饼干，这样可以最大化满足的孩子数量。
 * 具体步骤：
 * 1. 将孩子的胃口数组 g 和饼干尺寸数组 s 进行排序
 * 2. 使用两个指针分别遍历 g 和 s 数组
 * 3. 如果当前饼干能满足当前孩子，两个指针都向后移动
 * 4. 否则，只移动饼干指针，寻找更大的饼干
 *
 * 时间复杂度：O(n log n + m log m)，其中 n 是孩子数量，m 是饼干数量，主要是排序的时间复杂度
 * 空间复杂度：O(log n + log m)，排序所需的额外空间
 *
 * 是否最优解：是。贪心策略在此问题中能得到最优解。
```

```
 *
 * 适用场景：
 * 1. 资源分配问题，将有限资源分配给多个需求者
 * 2. 匹配问题，需要最大化匹配数量
 *
 * 异常处理：
 * 1. 处理空数组情况
 * 2. 处理数组长度为 0 的边界情况
 *
 * 工程化考量：
 * 1. 输入验证：检查数组是否为空
 * 2. 边界条件：当没有孩子或没有饼干时，直接返回 0
 * 3. 性能优化：使用快速排序提高效率
 */

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

using namespace std;

/**
 * 计算最多能满足的孩子数量
 *
 * @param g 孩子的胃口数组
 * @param s 饼干尺寸数组
 * @return 最多能满足的孩子数量
 */
int findContentChildren(vector<int>& g, vector<int>& s) {
    // 边界条件检查
    if (g.empty() || s.empty()) {
        return 0;
    }

    // 对胃口和饼干尺寸进行排序
    sort(g.begin(), g.end());
    sort(s.begin(), s.end());

    int childIndex = 0; // 指向当前需要满足的孩子
    int cookieIndex = 0; // 指向当前尝试分配的饼干
    int satisfiedChildren = 0; // 已满足的孩子数量

    // 遍历所有饼干和孩子
```

```
    while (childIndex < g.size() && cookieIndex < s.size()) {
        // 如果当前饼干能满足当前孩子的胃口
        if (s[cookieIndex] >= g[childIndex]) {
            satisfiedChildren++; // 满足的孩子数量加 1
            childIndex++; // 移动到下一个孩子
        }
        // 无论是否满足，都需要尝试下一个饼干
        cookieIndex++;
    }

    return satisfiedChildren;
}

/**
 * 打印数组内容
 *
 * @param arr 要打印的数组
 * @param name 数组名称
 */
void printArray(const vector<int>& arr, const string& name) {
    cout << name << ": [";
    for (size_t i = 0; i < arr.size(); i++) {
        cout << arr[i];
        if (i < arr.size() - 1) {
            cout << ", ";
        }
    }
    cout << "]" << endl;
}

/**
 * 测试函数，验证算法正确性
 */
int test_findContentChildren() {
    // 测试用例 1: 基本情况
    vector<int> g1 = {1, 2, 3};
    vector<int> s1 = {1, 1};
    int result1 = findContentChildren(g1, s1);
    cout << "测试用例 1:" << endl;
    printArray(g1, "孩子胃口");
    printArray(s1, "饼干尺寸");
    cout << "最多能满足的孩子数量: " << result1 << endl;
    cout << "期望输出: 1" << endl << endl;
```

```
    // 测试用例 2: 所有孩子都能被满足
    vector<int> g2 = {1, 2};
    vector<int> s2 = {1, 2, 3};
    int result2 = findContentChildren(g2, s2);
    cout << "测试用例 2:" << endl;
    printArray(g2, "孩子胃口");
    printArray(s2, "饼干尺寸");
    cout << "最多能满足的孩子数量: " << result2 << endl;
    cout << "期望输出: 2" << endl << endl;

    // 测试用例 3: 边界情况 - 没有饼干
    vector<int> g3 = {1, 2, 3};
    vector<int> s3 = {};
    int result3 = findContentChildren(g3, s3);
    cout << "测试用例 3:" << endl;
    printArray(g3, "孩子胃口");
    printArray(s3, "饼干尺寸");
    cout << "最多能满足的孩子数量: " << result3 << endl;
    cout << "期望输出: 0" << endl << endl;

    // 测试用例 4: 边界情况 - 没有孩子
    vector<int> g4 = {};
    vector<int> s4 = {1, 2, 3};
    int result4 = findContentChildren(g4, s4);
    cout << "测试用例 4:" << endl;
    printArray(g4, "孩子胃口");
    printArray(s4, "饼干尺寸");
    cout << "最多能满足的孩子数量: " << result4 << endl;
    cout << "期望输出: 0" << endl << endl;

    // 测试用例 5: 胃口数组和饼干数组长度不同
    vector<int> g5 = {1, 2, 3, 4, 5};
    vector<int> s5 = {1, 2, 3};
    int result5 = findContentChildren(g5, s5);
    cout << "测试用例 5:" << endl;
    printArray(g5, "孩子胃口");
    printArray(s5, "饼干尺寸");
    cout << "最多能满足的孩子数量: " << result5 << endl;
    cout << "期望输出: 3" << endl;

    return 0;
}
```

```
int main() {
    return test_findContentChildren();
}
```

==========

文件: Code06_AssignCookies.java

==========

```
package class093;

import java.util.Arrays;

/**
 * 分发饼干（Assign Cookies）
 * 题目来源：LeetCode 455
 * 题目链接：https://leetcode.cn/problems/assign-cookies/
 *
 * 问题描述：
 * 假设你是一位很棒的家长，想要给你的孩子们一些小饼干。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；
 * 对每个饼干 j，都有一个尺寸 s[j]。如果 s[j] >= g[i]，我们可以将这个饼干 j 分配给孩子 i，
 * 这个孩子会得到满足。你的目标是尽可能满足越多数量的孩子，并输出这个最大数值。
 *
 * 算法思路：
 * 使用贪心策略，将胃口最小的孩子分配给能满足他的最小饼干，这样可以最大化满足的孩子数量。
 * 具体步骤：
 * 1. 将孩子的胃口数组 g 和饼干尺寸数组 s 进行排序
 * 2. 使用两个指针分别遍历 g 和 s 数组
 * 3. 如果当前饼干能满足当前孩子，两个指针都向后移动
 * 4. 否则，只移动饼干指针，寻找更大的饼干
 *
 * 时间复杂度：O(n log n + m log m)，其中 n 是孩子数量，m 是饼干数量，主要是排序的时间复杂度
 * 空间复杂度：O(log n + log m)，排序所需的额外空间
 *
 * 是否最优解：是。贪心策略在此问题中能得到最优解。
 *
 * 适用场景：
 * 1. 资源分配问题，将有限资源分配给多个需求者
 * 2. 匹配问题，需要最大化匹配数量
 *
 * 异常处理：
 * 1. 处理空数组情况
```

```
 * 2. 处理数组长度为 0 的边界情况
 *
 * 工程化考量：
 * 1. 输入验证：检查数组是否为空
 * 2. 边界条件：当没有孩子或没有饼干时，直接返回 0
 * 3. 性能优化：使用快速排序提高效率
 */
public class Code06_AssignCookies {

    /**
     * 计算最多能满足的孩子数量
     *
     * @param g 孩子的胃口数组
     * @param s 饼干尺寸数组
     * @return 最多能满足的孩子数量
     */
    public static int findContentChildren(int[] g, int[] s) {
        // 边界条件检查
        if (g == null || s == null || g.length == 0 || s.length == 0) {
            return 0;
        }

        // 对胃口和饼干尺寸进行排序
        Arrays.sort(g);
        Arrays.sort(s);

        int childIndex = 0; // 指向当前需要满足的孩子
        int cookieIndex = 0; // 指向当前尝试分配的饼干
        int satisfiedChildren = 0; // 已满足的孩子数量

        // 遍历所有饼干和孩子
        while (childIndex < g.length && cookieIndex < s.length) {
            // 如果当前饼干能满足当前孩子的胃口
            if (s[cookieIndex] >= g[childIndex]) {
                satisfiedChildren++; // 满足的孩子数量加 1
                childIndex++; // 移动到下一个孩子
            }
            // 无论是否满足，都需要尝试下一个饼干
            cookieIndex++;
        }

        return satisfiedChildren;
    }
```

```
/**
 * 测试函数，验证算法正确性
 */
public static void main(String[] args) {
    // 测试用例 1: 基本情况
    int[] g1 = {1, 2, 3};
    int[] s1 = {1, 1};
    int result1 = findContentChildren(g1, s1);
    System.out.println("测试用例 1:");
    System.out.println("孩子胃口: " + Arrays.toString(g1));
    System.out.println("饼干尺寸: " + Arrays.toString(s1));
    System.out.println("最多能满足的孩子数量: " + result1);
    System.out.println("期望输出: 1");
    System.out.println();

    // 测试用例 2: 所有孩子都能被满足
    int[] g2 = {1, 2};
    int[] s2 = {1, 2, 3};
    int result2 = findContentChildren(g2, s2);
    System.out.println("测试用例 2:");
    System.out.println("孩子胃口: " + Arrays.toString(g2));
    System.out.println("饼干尺寸: " + Arrays.toString(s2));
    System.out.println("最多能满足的孩子数量: " + result2);
    System.out.println("期望输出: 2");
    System.out.println();

    // 测试用例 3: 边界情况 - 没有饼干
    int[] g3 = {1, 2, 3};
    int[] s3 = {};
    int result3 = findContentChildren(g3, s3);
    System.out.println("测试用例 3:");
    System.out.println("孩子胃口: " + Arrays.toString(g3));
    System.out.println("饼干尺寸: " + Arrays.toString(s3));
    System.out.println("最多能满足的孩子数量: " + result3);
    System.out.println("期望输出: 0");
    System.out.println();

    // 测试用例 4: 边界情况 - 没有孩子
    int[] g4 = {};
    int[] s4 = {1, 2, 3};
    int result4 = findContentChildren(g4, s4);
    System.out.println("测试用例 4:");
```

```
        System.out.println("孩子胃口: " + Arrays.toString(g4));
        System.out.println("饼干尺寸: " + Arrays.toString(s4));
        System.out.println("最多能满足的孩子数量: " + result4);
        System.out.println("期望输出: 0");
        System.out.println();

        // 测试用例 5: 胃口数组和饼干数组长度不同
        int[] g5 = {1, 2, 3, 4, 5};
        int[] s5 = {1, 2, 3};
        int result5 = findContentChildren(g5, s5);
        System.out.println("测试用例 5:");
        System.out.println("孩子胃口: " + Arrays.toString(g5));
        System.out.println("饼干尺寸: " + Arrays.toString(s5));
        System.out.println("最多能满足的孩子数量: " + result5);
        System.out.println("期望输出: 3");
    }
}
```

==========

文件: Code06_AssignCookies.py

==========

```
# 分发饼干（Assign Cookies）
# 题目来源：LeetCode 455
# 题目链接：https://leetcode.cn/problems/assign-cookies/

"""
问题描述：
假设你是一位很棒的家长，想要给你的孩子们一些小饼干。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；
对每个饼干 j，都有一个尺寸 s[j]。如果 s[j] >= g[i]，我们可以将这个饼干 j 分配给孩子 i，
这个孩子会得到满足。你的目标是尽可能满足越多数量的孩子，并输出这个最大数值。

算法思路：
使用贪心策略，将胃口最小的孩子分配给能满足他的最小饼干，这样可以最大化满足的孩子数量。
具体步骤：
1. 将孩子的胃口数组 g 和饼干尺寸数组 s 进行排序
2. 使用两个指针分别遍历 g 和 s 数组
3. 如果当前饼干能满足当前孩子，两个指针都向后移动
4. 否则，只移动饼干指针，寻找更大的饼干

时间复杂度：O(n log n + m log m)，其中 n 是孩子数量，m 是饼干数量，主要是排序的时间复杂度
空间复杂度：O(log n + log m)，排序所需的额外空间
```

是否最优解：是。贪心策略在此问题中能得到最优解。

适用场景：
1. 资源分配问题，将有限资源分配给多个需求者
2. 匹配问题，需要最大化匹配数量

异常处理：
1. 处理空数组情况
2. 处理数组长度为 0 的边界情况

工程化考量：
1. 输入验证：检查数组是否为空
2. 边界条件：当没有孩子或没有饼干时，直接返回 0
3. 性能优化：利用 Python 的内置排序函数提高效率
"""

```
class Solution:
    def findContentChildren(self, g, s):
        """
        计算最多能满足的孩子数量

        Args:
            g: List[int] - 孩子的胃口数组
            s: List[int] - 饼干尺寸数组

        Returns:
            int - 最多能满足的孩子数量
        """
        # 边界条件检查
        if not g or not s:
            return 0

        # 对胃口和饼干尺寸进行排序
        g.sort()
        s.sort()

        child_index = 0  # 指向当前需要满足的孩子
        cookie_index = 0  # 指向当前尝试分配的饼干
        satisfied_children = 0  # 已满足的孩子数量

        # 遍历所有饼干和孩子
        while child_index < len(g) and cookie_index < len(s):
```

```
            # 如果当前饼干能满足当前孩子的胃口
            if s[cookie_index] >= g[child_index]:
                satisfied_children += 1  # 满足的孩子数量加 1
                child_index += 1  # 移动到下一个孩子
            # 无论是否满足，都需要尝试下一个饼干
            cookie_index += 1

        return satisfied_children

# 测试函数，验证算法正确性
def test_find_content_children():
    solution = Solution()

    # 测试用例 1: 基本情况
    g1 = [1, 2, 3]
    s1 = [1, 1]
    result1 = solution.findContentChildren(g1, s1)
    print("测试用例 1:")
    print(f"孩子胃口: {g1}")
    print(f"饼干尺寸: {s1}")
    print(f"最多能满足的孩子数量: {result1}")
    print(f"期望输出: 1")
    print()

    # 测试用例 2: 所有孩子都能被满足
    g2 = [1, 2]
    s2 = [1, 2, 3]
    result2 = solution.findContentChildren(g2, s2)
    print("测试用例 2:")
    print(f"孩子胃口: {g2}")
    print(f"饼干尺寸: {s2}")
    print(f"最多能满足的孩子数量: {result2}")
    print(f"期望输出: 2")
    print()

    # 测试用例 3: 边界情况 - 没有饼干
    g3 = [1, 2, 3]
    s3 = []
    result3 = solution.findContentChildren(g3, s3)
    print("测试用例 3:")
    print(f"孩子胃口: {g3}")
    print(f"饼干尺寸: {s3}")
    print(f"最多能满足的孩子数量: {result3}")
```

```
    print(f"期望输出: 0")
    print()

    # 测试用例 4: 边界情况 - 没有孩子
    g4 = []
    s4 = [1, 2, 3]
    result4 = solution.findContentChildren(g4, s4)
    print("测试用例 4:")
    print(f"孩子胃口: {g4}")
    print(f"饼干尺寸: {s4}")
    print(f"最多能满足的孩子数量: {result4}")
    print(f"期望输出: 0")
    print()

    # 测试用例 5: 胃口数组和饼干数组长度不同
    g5 = [1, 2, 3, 4, 5]
    s5 = [1, 2, 3]
    result5 = solution.findContentChildren(g5, s5)
    print("测试用例 5:")
    print(f"孩子胃口: {g5}")
    print(f"饼干尺寸: {s5}")
    print(f"最多能满足的孩子数量: {result5}")
    print(f"期望输出: 3")

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

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

文件: Code07_LemonadeChange.cpp
==============================================
// 柠檬水找零（Lemonade Change）
// 题目来源：LeetCode 860
// 题目链接：https://leetcode.cn/problems/lemonade-change/

/**
 * 问题描述：
 * 在柠檬水摊上，每杯柠檬水的售价为 5 美元。
 * 顾客排队购买你的产品，（按账单 bills 支付的顺序）一次购买一杯。
 * 每位顾客只买一杯柠檬水，然后向你付 5 美元、10 美元或 20 美元。
 * 你必须给每个顾客正确找零，也就是说净交易是每位顾客向你支付 5 美元。
 * 注意，一开始你手头没有任何零钱。
```

```
 * 给你一个整数数组 bills ，其中 bills[i] 是第 i 位顾客付的账。
 * 如果你能给每位顾客正确找零，返回 true ，否则返回 false 。
 *
 * 算法思路：
 * 使用贪心策略，优先使用大面额的钞票来找零，这样可以保留更多小面额的钞票用于未来可能的找零。
 * 具体步骤：
 * 1. 维护两个变量，分别记录当前拥有的 5 美元和 10 美元的数量
 * 2. 遍历账单数组：
 *    - 如果顾客支付 5 美元，直接增加 5 美元的数量
 *    - 如果顾客支付 10 美元，需要找零 5 美元，减少 5 美元数量，增加 10 美元数量
 *    - 如果顾客支付 20 美元，优先找零 10+5 美元，然后再考虑三个 5 美元
 * 3. 如果在任何时候无法满足找零需求，返回 false
 *
 * 时间复杂度：O(n)，其中 n 是账单数量，只需遍历数组一次
 * 空间复杂度：O(1)，只使用了常数额外空间
 *
 * 是否最优解：是。贪心策略在此问题中能得到最优解。
 *
 * 适用场景：
 * 1. 现金交易找零问题
 * 2. 资源分配问题，其中不同面额的钞票代表不同类型的资源
 *
 * 异常处理：
 * 1. 处理空数组情况
 * 2. 处理无效支付情况（如支付不是 5、10、20 的情况）
 *
 * 工程化考量：
 * 1. 输入验证：检查数组是否为空，检查支付金额是否合法
 * 2. 边界条件：处理第一个顾客不是支付 5 美元的情况
 * 3. 性能优化：使用变量而不是哈希表来跟踪钞票数量，提高效率
 */

#include <iostream>
#include <vector>

using namespace std;

/**
 * 判断是否能给每位顾客正确找零
 *
 * @param bills 顾客支付的账单数组
 * @return 如果能正确找零返回 true，否则返回 false
 */
```

```
bool lemonadeChange(vector<int>& bills) {
    // 边界条件检查
    if (bills.empty()) {
        return true; // 没有顾客，自然能正确找零
    }

    int fiveCount = 0; // 5美元钞票的数量
    int tenCount = 0;  // 10美元钞票的数量

    for (int bill : bills) {
        if (bill == 5) {
            // 顾客支付5美元，不需要找零，直接收入
            fiveCount++;
        } else if (bill == 10) {
            // 顾客支付10美元，需要找零5美元
            if (fiveCount == 0) {
                return false; // 没有5美元钞票找零
            }
            fiveCount--;
            tenCount++;
        } else if (bill == 20) {
            // 顾客支付20美元，优先找零10+5美元，因为5美元更常用
            if (tenCount > 0 && fiveCount > 0) {
                tenCount--;
                fiveCount--;
            } else if (fiveCount >= 3) {
                // 如果没有10美元钞票，使用三个5美元钞票找零
                fiveCount -= 3;
            } else {
                return false; // 无法找零
            }
        } else {
            // 无效的支付金额
            return false;
        }
    }

    // 所有顾客都能正确找零
    return true;
}

/**
 * 打印数组内容
```

```
 *
 * @param arr 要打印的数组
 * @param name 数组名称
 */
void printArray(const vector<int>& arr, const string& name) {
    cout << name << ": [";
    for (size_t i = 0; i < arr.size(); i++) {
        cout << arr[i];
        if (i < arr.size() - 1) {
            cout << ", ";
        }
    }
    cout << "]" << endl;
}

/**
 * 测试函数，验证算法正确性
 */
int test_lemonadeChange() {
    // 测试用例 1: 基本情况 - 能正确找零
    vector<int> bills1 = {5, 5, 5, 10, 20};
    bool result1 = lemonadeChange(bills1);
    cout << "测试用例 1:" << endl;
    printArray(bills1, "账单顺序");
    cout << "能否正确找零: " << (result1 ? "true" : "false") << endl;
    cout << "期望输出: true" << endl << endl;

    // 测试用例 2: 基本情况 - 不能正确找零
    vector<int> bills2 = {5, 5, 10, 10, 20};
    bool result2 = lemonadeChange(bills2);
    cout << "测试用例 2:" << endl;
    printArray(bills2, "账单顺序");
    cout << "能否正确找零: " << (result2 ? "true" : "false") << endl;
    cout << "期望输出: false" << endl << endl;

    // 测试用例 3: 边界情况 - 空数组
    vector<int> bills3 = {};
    bool result3 = lemonadeChange(bills3);
    cout << "测试用例 3:" << endl;
    printArray(bills3, "账单顺序");
    cout << "能否正确找零: " << (result3 ? "true" : "false") << endl;
    cout << "期望输出: true" << endl << endl;
```

```
    // 测试用例 4: 边界情况 - 第一个顾客支付 10 美元
    vector<int> bills4 = {10, 5, 5, 5};
    bool result4 = lemonadeChange(bills4);
    cout << "测试用例 4:" << endl;
    printArray(bills4, "账单顺序");
    cout << "能否正确找零: " << (result4 ? "true" : "false") << endl;
    cout << "期望输出: false" << endl << endl;

    // 测试用例 5: 复杂情况 - 多个 20 美元的处理
    vector<int> bills5 = {5, 5, 10, 5, 20, 5, 10, 5, 20};
    bool result5 = lemonadeChange(bills5);
    cout << "测试用例 5:" << endl;
    printArray(bills5, "账单顺序");
    cout << "能否正确找零: " << (result5 ? "true" : "false") << endl;
    cout << "期望输出: true" << endl;

    return 0;
}

int main() {
    return test_lemonadeChange();
}
```

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

文件: Code07_LemonadeChange.java

```
================================================
package class093;

/**
 * 柠檬水找零（Lemonade Change）
 * 题目来源：LeetCode 860
 * 题目链接：https://leetcode.cn/problems/lemonade-change/
 *
 * 问题描述：
 * 在柠檬水摊上，每杯柠檬水的售价为 5 美元。
 * 顾客排队购买你的产品，（按账单 bills 支付的顺序）一次购买一杯。
 * 每位顾客只买一杯柠檬水，然后向你付 5 美元、10 美元或 20 美元。
 * 你必须给每个顾客正确找零，也就是说净交易是每位顾客向你支付 5 美元。
 * 注意，一开始你手头没有任何零钱。
 * 给你一个整数数组 bills ，其中 bills[i] 是第 i 位顾客付的账。
 * 如果你能给每位顾客正确找零，返回 true ，否则返回 false 。
 *
```

```
 * 算法思路：
 * 使用贪心策略，优先使用大面额的钞票来找零，这样可以保留更多小面额的钞票用于未来可能的找零。
 * 具体步骤：
 * 1. 维护两个变量，分别记录当前拥有的 5 美元和 10 美元的数量
 * 2. 遍历账单数组：
 *    - 如果顾客支付 5 美元，直接增加 5 美元的数量
 *    - 如果顾客支付 10 美元，需要找零 5 美元，减少 5 美元数量，增加 10 美元数量
 *    - 如果顾客支付 20 美元，优先找零 10+5 美元，然后再考虑三个 5 美元
 * 3. 如果在任何时候无法满足找零需求，返回 false
 *
 * 时间复杂度：O(n)，其中 n 是账单数量，只需遍历数组一次
 * 空间复杂度：O(1)，只使用了常数额外空间
 *
 * 是否最优解：是。贪心策略在此问题中能得到最优解。
 *
 * 适用场景：
 * 1. 现金交易找零问题
 * 2. 资源分配问题，其中不同面额的钞票代表不同类型的资源
 *
 * 异常处理：
 * 1. 处理空数组情况
 * 2. 处理无效支付情况（如支付不是 5、10、20 的情况）
 *
 * 工程化考量：
 * 1. 输入验证：检查数组是否为空，检查支付金额是否合法
 * 2. 边界条件：处理第一个顾客不是支付 5 美元的情况
 * 3. 性能优化：使用变量而不是哈希表来跟踪钞票数量，提高效率
 */
public class Code07_LemonadeChange {

    /**
     * 判断是否能给每位顾客正确找零
     *
     * @param bills 顾客支付的账单数组
     * @return 如果能正确找零返回 true，否则返回 false
     */
    public static boolean lemonadeChange(int[] bills) {
        // 边界条件检查
        if (bills == null || bills.length == 0) {
            return true; // 没有顾客，自然能正确找零
        }

        int fiveCount = 0; // 5 美元钞票的数量
```

```
    int tenCount = 0; // 10美元钞票的数量

    for (int bill : bills) {
        switch (bill) {
            case 5:
                // 顾客支付5美元，不需要找零，直接收入
                fiveCount++;
                break;
            case 10:
                // 顾客支付10美元，需要找零5美元
                if (fiveCount == 0) {
                    return false; // 没有5美元钞票找零
                }
                fiveCount--;
                tenCount++;
                break;
            case 20:
                // 顾客支付20美元，优先找零10+5美元，因为5美元更常用
                if (tenCount > 0 && fiveCount > 0) {
                    tenCount--;
                    fiveCount--;
                } else if (fiveCount >= 3) {
                    // 如果没有10美元钞票，使用三个5美元钞票找零
                    fiveCount -= 3;
                } else {
                    return false; // 无法找零
                }
                break;
            default:
                // 无效的支付金额
                return false;
        }
    }

    // 所有顾客都能正确找零
    return true;
}

/**
 * 测试函数，验证算法正确性
 */
public static void main(String[] args) {
    // 测试用例1: 基本情况 - 能正确找零
```

```
int[] bills1 = {5, 5, 5, 10, 20};
boolean result1 = lemonadeChange(bills1);
System.out.println("测试用例 1:");
System.out.print("账单顺序: [");
for (int i = 0; i < bills1.length; i++) {
    System.out.print(bills1[i]);
    if (i < bills1.length - 1) {
        System.out.print(", ");
    }
}
System.out.println("]");
System.out.println("能否正确找零: " + result1);
System.out.println("期望输出: true");
System.out.println();

// 测试用例 2: 基本情况 - 不能正确找零
int[] bills2 = {5, 5, 10, 10, 20};
boolean result2 = lemonadeChange(bills2);
System.out.println("测试用例 2:");
System.out.print("账单顺序: [");
for (int i = 0; i < bills2.length; i++) {
    System.out.print(bills2[i]);
    if (i < bills2.length - 1) {
        System.out.print(", ");
    }
}
System.out.println("]");
System.out.println("能否正确找零: " + result2);
System.out.println("期望输出: false");
System.out.println();

// 测试用例 3: 边界情况 - 空数组
int[] bills3 = {};
boolean result3 = lemonadeChange(bills3);
System.out.println("测试用例 3:");
System.out.println("账单顺序: []");
System.out.println("能否正确找零: " + result3);
System.out.println("期望输出: true");
System.out.println();

// 测试用例 4: 边界情况 - 第一个顾客支付 10 美元
int[] bills4 = {10, 5, 5, 5};
boolean result4 = lemonadeChange(bills4);
```

```
        System.out.println("测试用例 4:");
        System.out.print("账单顺序: [");
        for (int i = 0; i < bills4.length; i++) {
            System.out.print(bills4[i]);
            if (i < bills4.length - 1) {
                System.out.print(", ");
            }
        }
        System.out.println("]");
        System.out.println("能否正确找零: " + result4);
        System.out.println("期望输出: false");
        System.out.println();

        // 测试用例 5: 复杂情况 - 多个 20 美元的处理
        int[] bills5 = {5, 5, 10, 5, 20, 5, 10, 5, 20};
        boolean result5 = lemonadeChange(bills5);
        System.out.println("测试用例 5:");
        System.out.print("账单顺序: [");
        for (int i = 0; i < bills5.length; i++) {
            System.out.print(bills5[i]);
            if (i < bills5.length - 1) {
                System.out.print(", ");
            }
        }
        System.out.println("]");
        System.out.println("能否正确找零: " + result5);
        System.out.println("期望输出: true");
    }
}
```

==========

文件: Code07_LemonadeChange.py

==========

```
# 柠檬水找零（Lemonade Change）
# 题目来源：LeetCode 860
# 题目链接：https://leetcode.cn/problems/lemonade-change/

"""
```

问题描述：

在柠檬水摊上，每杯柠檬水的售价为 5 美元。

顾客排队购买你的产品，（按账单 bills 支付的顺序）一次购买一杯。

每位顾客只买一杯柠檬水，然后向你付 5 美元、10 美元或 20 美元。

你必须给每个顾客正确找零，也就是说净交易是每位顾客向你支付 5 美元。
注意，一开始你手头没有任何零钱。
给你一个整数数组 bills ，其中 bills[i] 是第 i 位顾客付的账。
如果你能给每位顾客正确找零，返回 true ，否则返回 false 。

算法思路：
使用贪心策略，优先使用大面额的钞票来找零，这样可以保留更多小面额的钞票用于未来可能的找零。
具体步骤：
1. 维护两个变量，分别记录当前拥有的 5 美元和 10 美元的数量
2. 遍历账单数组：
   - 如果顾客支付 5 美元，直接增加 5 美元的数量
   - 如果顾客支付 10 美元，需要找零 5 美元，减少 5 美元数量，增加 10 美元数量
   - 如果顾客支付 20 美元，优先找零 10+5 美元，然后再考虑三个 5 美元
3. 如果在任何时候无法满足找零需求，返回 false

时间复杂度：O(n)，其中 n 是账单数量，只需遍历数组一次
空间复杂度：O(1)，只使用了常数额外空间

是否最优解：是。贪心策略在此问题中能得到最优解。

适用场景：
1. 现金交易找零问题
2. 资源分配问题，其中不同面额的钞票代表不同类型的资源

异常处理：
1. 处理空数组情况
2. 处理无效支付情况（如支付不是 5、10、20 的情况）

工程化考量：
1. 输入验证：检查数组是否为空，检查支付金额是否合法
2. 边界条件：处理第一个顾客不是支付 5 美元的情况
3. 性能优化：使用变量而不是字典来跟踪钞票数量，提高效率
"""

```
class Solution:
    def lemonadeChange(self, bills):
        """
        判断是否能给每位顾客正确找零

        Args:
            bills: List[int] - 顾客支付的账单数组

        Returns:
```

```
        bool - 如果能正确找零返回 True，否则返回 False
        """
        # 边界条件检查
        if not bills:
            return True  # 没有顾客，自然能正确找零

        five_count = 0  # 5 美元钞票的数量
        ten_count = 0   # 10 美元钞票的数量

        for bill in bills:
            if bill == 5:
                # 顾客支付 5 美元，不需要找零，直接收入
                five_count += 1
            elif bill == 10:
                # 顾客支付 10 美元，需要找零 5 美元
                if five_count == 0:
                    return False  # 没有 5 美元钞票找零
                five_count -= 1
                ten_count += 1
            elif bill == 20:
                # 顾客支付 20 美元，优先找零 10+5 美元，因为 5 美元更常用
                if ten_count > 0 and five_count > 0:
                    ten_count -= 1
                    five_count -= 1
                elif five_count >= 3:
                    # 如果没有 10 美元钞票，使用三个 5 美元钞票找零
                    five_count -= 3
                else:
                    return False  # 无法找零
            else:
                # 无效的支付金额
                return False

        # 所有顾客都能正确找零
        return True

# 测试函数，验证算法正确性
def test_lemonade_change():
    solution = Solution()

    # 测试用例 1: 基本情况 - 能正确找零
    bills1 = [5, 5, 5, 10, 20]
    result1 = solution.lemonadeChange(bills1)
```

```
    print("测试用例 1:")
    print(f"账单顺序: {bills1}")
    print(f"能否正确找零: {result1}")
    print(f"期望输出: True")
    print()

    # 测试用例 2: 基本情况 - 不能正确找零
    bills2 = [5, 5, 10, 10, 20]
    result2 = solution.lemonadeChange(bills2)
    print("测试用例 2:")
    print(f"账单顺序: {bills2}")
    print(f"能否正确找零: {result2}")
    print(f"期望输出: False")
    print()

    # 测试用例 3: 边界情况 - 空数组
    bills3 = []
    result3 = solution.lemonadeChange(bills3)
    print("测试用例 3:")
    print(f"账单顺序: {bills3}")
    print(f"能否正确找零: {result3}")
    print(f"期望输出: True")
    print()

    # 测试用例 4: 边界情况 - 第一个顾客支付 10 美元
    bills4 = [10, 5, 5, 5]
    result4 = solution.lemonadeChange(bills4)
    print("测试用例 4:")
    print(f"账单顺序: {bills4}")
    print(f"能否正确找零: {result4}")
    print(f"期望输出: False")
    print()

    # 测试用例 5: 复杂情况 - 多个 20 美元的处理
    bills5 = [5, 5, 10, 5, 20, 5, 10, 5, 20]
    result5 = solution.lemonadeChange(bills5)
    print("测试用例 5:")
    print(f"账单顺序: {bills5}")
    print(f"能否正确找零: {result5}")
    print(f"期望输出: True")

# 运行测试
if __name__ == "__main__":
```

```
    test_lemonade_change()

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

文件: Code08_JumpGame.cpp
==========================================
// 跳跃游戏（Jump Game）
// 题目来源：LeetCode 55
// 题目链接：https://leetcode.cn/problems/jump-game/

/**
 * 问题描述：
 * 给定一个非负整数数组 nums，你最初位于数组的第一个位置。
 * 数组中的每个元素代表你在该位置可以跳跃的最大长度。
 * 判断你是否能够到达最后一个位置。
 *
 * 算法思路：
 * 使用贪心策略，维护当前能到达的最远位置。
 * 具体步骤：
 * 1. 遍历数组，对于每个位置，更新能到达的最远位置
 * 2. 如果最远位置超过或等于数组的最后一个位置，返回 true
 * 3. 如果在遍历过程中发现当前位置已经无法到达（即当前位置大于能到达的最远位置），返回 false
 *
 * 时间复杂度：O(n)，其中 n 是数组长度，只需遍历数组一次
 * 空间复杂度：O(1)，只使用了常数额外空间
 *
 * 是否最优解：是。贪心策略在此问题中能得到最优解。
 *
 * 适用场景：
 * 1. 路径可达性问题
 * 2. 资源约束下的可达性判断
 *
 * 异常处理：
 * 1. 处理空数组情况
 * 2. 处理数组长度为 1 的边界情况（已经在终点）
 *
 * 工程化考量：
 * 1. 输入验证：检查数组是否为空
 * 2. 边界条件：处理单元素数组
 * 3. 性能优化：提前返回，一旦确定可以到达终点就立即返回
 */

#include <iostream>
```

```cpp
#include <vector>
#include <algorithm> // for max function

using namespace std;

/**
 * 判断是否能够到达数组的最后一个位置
 *
 * @param nums 非负整数数组，每个元素表示在该位置可以跳跃的最大长度
 * @return 如果能够到达最后一个位置返回 true，否则返回 false
 */
bool canJump(vector<int>& nums) {
    // 边界条件检查
    if (nums.empty()) {
        return false;
    }

    // 如果数组只有一个元素，已经在终点
    if (nums.size() == 1) {
        return true;
    }

    int maxReach = 0; // 当前能到达的最远位置

    // 遍历数组中的每个位置
    for (int i = 0; i < nums.size(); i++) {
        // 如果当前位置已经无法到达，直接返回 false
        if (i > maxReach) {
            return false;
        }

        // 更新能到达的最远位置
        maxReach = max(maxReach, i + nums[i]);

        // 如果最远位置已经可以到达或超过最后一个位置，直接返回 true
        if (maxReach >= nums.size() - 1) {
            return true;
        }
    }

    // 理论上不会执行到这里，因为前面已经有检查
    return maxReach >= nums.size() - 1;
}
```

```
/**
 * 打印数组内容
 *
 * @param arr 要打印的数组
 * @param name 数组名称
 */
void printArray(const vector<int>& arr, const string& name) {
    cout << name << ": [";
    for (size_t i = 0; i < arr.size(); i++) {
        cout << arr[i];
        if (i < arr.size() - 1) {
            cout << ", ";
        }
    }
    cout << "]" << endl;
}

/**
 * 测试函数，验证算法正确性
 */
int testCanJump() {
    // 测试用例 1: 基本情况 - 能到达终点
    vector<int> nums1 = {2, 3, 1, 1, 4};
    bool result1 = canJump(nums1);
    cout << "测试用例 1:" << endl;
    printArray(nums1, "数组");
    cout << "能否到达最后一个位置: " << (result1 ? "true" : "false") << endl;
    cout << "期望输出: true" << endl << endl;

    // 测试用例 2: 基本情况 - 不能到达终点
    vector<int> nums2 = {3, 2, 1, 0, 4};
    bool result2 = canJump(nums2);
    cout << "测试用例 2:" << endl;
    printArray(nums2, "数组");
    cout << "能否到达最后一个位置: " << (result2 ? "true" : "false") << endl;
    cout << "期望输出: false" << endl << endl;

    // 测试用例 3: 边界情况 - 单元素数组
    vector<int> nums3 = {0};
    bool result3 = canJump(nums3);
    cout << "测试用例 3:" << endl;
    printArray(nums3, "数组");
```

```
    cout << "能否到达最后一个位置: " << (result3 ? "true" : "false") << endl;
    cout << "期望输出: true" << endl << endl;

    // 测试用例 4: 边界情况 - 所有元素都是 0
    vector<int> nums4 = {0, 0, 0, 0};
    bool result4 = canJump(nums4);
    cout << "测试用例 4:" << endl;
    printArray(nums4, "数组");
    cout << "能否到达最后一个位置: " << (result4 ? "true" : "false") << endl;
    cout << "期望输出: false" << endl << endl;

    // 测试用例 5: 复杂情况 - 大跳跃
    vector<int> nums5 = {2, 0, 0};
    bool result5 = canJump(nums5);
    cout << "测试用例 5:" << endl;
    printArray(nums5, "数组");
    cout << "能否到达最后一个位置: " << (result5 ? "true" : "false") << endl;
    cout << "期望输出: true" << endl;

    return 0;
}

int main() {
    return testCanJump();
}
```

==========

文件: Code08_JumpGame.java

==========

```
package class093;

/**
 * 跳跃游戏（Jump Game）
 * 题目来源：LeetCode 55
 * 题目链接：https://leetcode.cn/problems/jump-game/
 *
 * 问题描述：
 * 给定一个非负整数数组 nums，你最初位于数组的第一个位置。
 * 数组中的每个元素代表你在该位置可以跳跃的最大长度。
 * 判断你是否能够到达最后一个位置。
 *
 * 算法思路：
```

```
 * 使用贪心策略，维护当前能到达的最远位置。
 * 具体步骤：
 * 1. 遍历数组，对于每个位置，更新能到达的最远位置
 * 2. 如果最远位置超过或等于数组的最后一个位置，返回 true
 * 3. 如果在遍历过程中发现当前位置已经无法到达（即当前位置大于能到达的最远位置），返回 false
 *
 * 时间复杂度：O(n)，其中 n 是数组长度，只需遍历数组一次
 * 空间复杂度：O(1)，只使用了常数额外空间
 *
 * 是否最优解：是。贪心策略在此问题中能得到最优解。
 *
 * 适用场景：
 * 1. 路径可达性问题
 * 2. 资源约束下的可达性判断
 *
 * 异常处理：
 * 1. 处理空数组情况
 * 2. 处理数组长度为 1 的边界情况（已经在终点）
 *
 * 工程化考量：
 * 1. 输入验证：检查数组是否为空
 * 2. 边界条件：处理单元素数组
 * 3. 性能优化：提前返回，一旦确定可以到达终点就立即返回
 */
public class Code08_JumpGame {

    /**
     * 判断是否能够到达数组的最后一个位置
     *
     * @param nums 非负整数数组，每个元素表示在该位置可以跳跃的最大长度
     * @return 如果能够到达最后一个位置返回 true，否则返回 false
     */
    public static boolean canJump(int[] nums) {
        // 边界条件检查
        if (nums == null || nums.length == 0) {
            return false;
        }

        // 如果数组只有一个元素，已经在终点
        if (nums.length == 1) {
            return true;
        }
```

```
    int maxReach = 0; // 当前能到达的最远位置

    // 遍历数组中的每个位置
    for (int i = 0; i < nums.length; i++) {
        // 如果当前位置已经无法到达，直接返回 false
        if (i > maxReach) {
            return false;
        }

        // 更新能到达的最远位置
        maxReach = Math.max(maxReach, i + nums[i]);

        // 如果最远位置已经可以到达或超过最后一个位置，直接返回 true
        if (maxReach >= nums.length - 1) {
            return true;
        }
    }

    // 理论上不会执行到这里，因为前面已经有检查
    return maxReach >= nums.length - 1;
}

/**
 * 测试函数，验证算法正确性
 */
public static void main(String[] args) {
    // 测试用例 1: 基本情况 - 能到达终点
    int[] nums1 = {2, 3, 1, 1, 4};
    boolean result1 = canJump(nums1);
    System.out.println("测试用例 1:");
    System.out.print("数组: [");
    for (int i = 0; i < nums1.length; i++) {
        System.out.print(nums1[i]);
        if (i < nums1.length - 1) {
            System.out.print(", ");
        }
    }
    System.out.println("]");
    System.out.println("能否到达最后一个位置: " + result1);
    System.out.println("期望输出: true");
    System.out.println();

    // 测试用例 2: 基本情况 - 不能到达终点
```

```
int[] nums2 = {3, 2, 1, 0, 4};
boolean result2 = canJump(nums2);
System.out.println("测试用例 2:");
System.out.print("数组: [");
for (int i = 0; i < nums2.length; i++) {
    System.out.print(nums2[i]);
    if (i < nums2.length - 1) {
        System.out.print(", ");
    }
}
System.out.println("]");
System.out.println("能否到达最后一个位置: " + result2);
System.out.println("期望输出: false");
System.out.println();

// 测试用例 3: 边界情况 - 单元素数组
int[] nums3 = {0};
boolean result3 = canJump(nums3);
System.out.println("测试用例 3:");
System.out.println("数组: [0]");
System.out.println("能否到达最后一个位置: " + result3);
System.out.println("期望输出: true");
System.out.println();

// 测试用例 4: 边界情况 - 所有元素都是 0
int[] nums4 = {0, 0, 0, 0};
boolean result4 = canJump(nums4);
System.out.println("测试用例 4:");
System.out.print("数组: [");
for (int i = 0; i < nums4.length; i++) {
    System.out.print(nums4[i]);
    if (i < nums4.length - 1) {
        System.out.print(", ");
    }
}
System.out.println("]");
System.out.println("能否到达最后一个位置: " + result4);
System.out.println("期望输出: false");
System.out.println();

// 测试用例 5: 复杂情况 - 大跳跃
int[] nums5 = {2, 0, 0};
boolean result5 = canJump(nums5);
```

```
        System.out.println("测试用例 5:");
        System.out.print("数组: [");
        for (int i = 0; i < nums5.length; i++) {
            System.out.print(nums5[i]);
            if (i < nums5.length - 1) {
                System.out.print(", ");
            }
        }
        System.out.println("]");
        System.out.println("能否到达最后一个位置: " + result5);
        System.out.println("期望输出: true");
    }
}
```

==========

文件: Code08_JumpGame.py

==========

```
# 跳跃游戏（Jump Game）
# 题目来源：LeetCode 55
# 题目链接：https://leetcode.cn/problems/jump-game/

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

算法思路：
使用贪心策略，维护当前能到达的最远位置。
具体步骤：
1. 遍历数组，对于每个位置，更新能到达的最远位置
2. 如果最远位置超过或等于数组的最后一个位置，返回 true
3. 如果在遍历过程中发现当前位置已经无法到达（即当前位置大于能到达的最远位置），返回 false

时间复杂度：O(n)，其中 n 是数组长度，只需遍历数组一次
空间复杂度：O(1)，只使用了常数额外空间

是否最优解：是。贪心策略在此问题中能得到最优解。

适用场景：
1. 路径可达性问题
2. 资源约束下的可达性判断
```

异常处理：
1. 处理空数组情况
2. 处理数组长度为 1 的边界情况（已经在终点）

工程化考量：
1. 输入验证：检查数组是否为空
2. 边界条件：处理单元素数组
3. 性能优化：提前返回，一旦确定可以到达终点就立即返回
"""

```
class Solution:
    def canJump(self, nums):
        """
        判断是否能够到达数组的最后一个位置

        Args:
            nums: List[int] - 非负整数数组，每个元素表示在该位置可以跳跃的最大长度

        Returns:
            bool - 如果能够到达最后一个位置返回 True，否则返回 False
        """
        # 边界条件检查
        if not nums:
            return False

        # 如果数组只有一个元素，已经在终点
        if len(nums) == 1:
            return True

        max_reach = 0  # 当前能到达的最远位置

        # 遍历数组中的每个位置
        for i in range(len(nums)):
            # 如果当前位置已经无法到达，直接返回 False
            if i > max_reach:
                return False

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

            # 如果最远位置已经可以到达或超过最后一个位置，直接返回 True
            if max_reach >= len(nums) - 1:
```

```
                return True

        # 理论上不会执行到这里，因为前面已经有检查
        return max_reach >= len(nums) - 1

# 测试函数，验证算法正确性
def test_can_jump():
    solution = Solution()

    # 测试用例 1: 基本情况 - 能到达终点
    nums1 = [2, 3, 1, 1, 4]
    result1 = solution.canJump(nums1)
    print("测试用例 1:")
    print(f"数组: {nums1}")
    print(f"能否到达最后一个位置: {result1}")
    print(f"期望输出: True")
    print()

    # 测试用例 2: 基本情况 - 不能到达终点
    nums2 = [3, 2, 1, 0, 4]
    result2 = solution.canJump(nums2)
    print("测试用例 2:")
    print(f"数组: {nums2}")
    print(f"能否到达最后一个位置: {result2}")
    print(f"期望输出: False")
    print()

    # 测试用例 3: 边界情况 - 单元素数组
    nums3 = [0]
    result3 = solution.canJump(nums3)
    print("测试用例 3:")
    print(f"数组: {nums3}")
    print(f"能否到达最后一个位置: {result3}")
    print(f"期望输出: True")
    print()

    # 测试用例 4: 边界情况 - 所有元素都是 0
    nums4 = [0, 0, 0, 0]
    result4 = solution.canJump(nums4)
    print("测试用例 4:")
    print(f"数组: {nums4}")
    print(f"能否到达最后一个位置: {result4}")
    print(f"期望输出: False")
```

```
    print()

    # 测试用例 5: 复杂情况 - 大跳跃
    nums5 = [2, 0, 0]
    result5 = solution.canJump(nums5)
    print("测试用例 5:")
    print(f"数组: {nums5}")
    print(f"能否到达最后一个位置: {result5}")
    print(f"期望输出: True")

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

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

文件: Code09_BestTimeToBuyAndSellStockII.cpp
==============================================
// 买卖股票的最佳时机 II（Best Time to Buy and Sell Stock II）
// 题目来源：LeetCode 122
// 题目链接：https://leetcode.cn/problems/best-time-to-buy-and-sell-stock-ii/

/**
 * 问题描述：
 * 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。
 *
 * 算法思路：
 * 使用贪心策略，只要明天的价格比今天高，就在今天买入，明天卖出。
 * 具体步骤：
 * 1. 遍历价格数组，从第二天开始
 * 2. 如果当天价格高于前一天，就累加差价作为利润
 * 3. 最终得到的就是最大利润
 *
 * 时间复杂度：O(n)，其中 n 是价格数组的长度，只需遍历数组一次
 * 空间复杂度：O(1)，只使用了常数额外空间
 *
 * 是否最优解：是。贪心策略在此问题中能得到最优解。
 *
 * 适用场景：
 * 1. 股票交易策略问题
```

```
 * 2. 连续收益最大化问题
 *
 * 异常处理：
 * 1. 处理空数组情况
 * 2. 处理数组长度为 1 的边界情况（无法交易）
 *
 * 工程化考量：
 * 1. 输入验证：检查数组是否为空或长度不足
 * 2. 边界条件：处理单元素数组
 * 3. 性能优化：一次遍历完成计算
 */

#include <iostream>
#include <vector>

using namespace std;

/**
 * 计算能获得的最大利润
 *
 * @param prices 表示股票每天价格的数组
 * @return 最大利润
 */
int maxProfit(vector<int>& prices) {
    // 边界条件检查
    if (prices.empty() || prices.size() <= 1) {
        return 0; // 数组为空或只有一天，无法交易
    }

    int totalProfit = 0; // 总利润

    // 遍历价格数组，从第二天开始
    for (int i = 1; i < prices.size(); i++) {
        // 如果当天价格高于前一天，就累加差价作为利润
        if (prices[i] > prices[i - 1]) {
            totalProfit += prices[i] - prices[i - 1];
        }
    }

    return totalProfit;
}

/**
```

```
 * 打印数组内容
 *
 * @param arr 要打印的数组
 * @param name 数组名称
 */
void printArray(const vector<int>& arr, const string& name) {
    cout << name << ": [";
    for (size_t i = 0; i < arr.size(); i++) {
        cout << arr[i];
        if (i < arr.size() - 1) {
            cout << ", ";
        }
    }
    cout << "]" << endl;
}

/**
 * 测试函数，验证算法正确性
 */
int testMaxProfit() {
    // 测试用例 1: 基本情况 - 有上涨趋势
    vector<int> prices1 = {7, 1, 5, 3, 6, 4};
    int result1 = maxProfit(prices1);
    cout << "测试用例 1:" << endl;
    printArray(prices1, "价格数组");
    cout << "最大利润: " << result1 << endl;
    cout << "期望输出: 7" << endl << endl;

    // 测试用例 2: 基本情况 - 持续上涨
    vector<int> prices2 = {1, 2, 3, 4, 5};
    int result2 = maxProfit(prices2);
    cout << "测试用例 2:" << endl;
    printArray(prices2, "价格数组");
    cout << "最大利润: " << result2 << endl;
    cout << "期望输出: 4" << endl << endl;

    // 测试用例 3: 基本情况 - 持续下跌
    vector<int> prices3 = {7, 6, 4, 3, 1};
    int result3 = maxProfit(prices3);
    cout << "测试用例 3:" << endl;
    printArray(prices3, "价格数组");
    cout << "最大利润: " << result3 << endl;
    cout << "期望输出: 0" << endl << endl;
```

```
    // 测试用例 4: 边界情况 - 单元素数组
    vector<int> prices4 = {5};
    int result4 = maxProfit(prices4);
    cout << "测试用例 4:" << endl;
    printArray(prices4, "价格数组");
    cout << "最大利润: " << result4 << endl;
    cout << "期望输出: 0" << endl << endl;

    // 测试用例 5: 复杂情况 - 波动较大
    vector<int> prices5 = {3, 3, 5, 0, 0, 3, 1, 4};
    int result5 = maxProfit(prices5);
    cout << "测试用例 5:" << endl;
    printArray(prices5, "价格数组");
    cout << "最大利润: " << result5 << endl;
    cout << "期望输出: 8" << endl;

    return 0;
}

int main() {
    return testMaxProfit();
}
```

==========

文件: Code09_BestTimeToBuyAndSellStockII.java

==========

```
package class093;

/**
 * 买卖股票的最佳时机 II（Best Time to Buy and Sell Stock II）
 * 题目来源：LeetCode 122
 * 题目链接：https://leetcode.cn/problems/best-time-to-buy-and-sell-stock-ii/
 *
 * 问题描述：
 * 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。
 *
 * 算法思路：
 * 使用贪心策略，只要明天的价格比今天高，就在今天买入，明天卖出。
```

```
 * 具体步骤：
 * 1. 遍历价格数组，从第二天开始
 * 2. 如果当天价格高于前一天，就累加差价作为利润
 * 3. 最终得到的就是最大利润
 *
 * 时间复杂度：O(n)，其中 n 是价格数组的长度，只需遍历数组一次
 * 空间复杂度：O(1)，只使用了常数额外空间
 *
 * 是否最优解：是。贪心策略在此问题中能得到最优解。
 *
 * 适用场景：
 * 1. 股票交易策略问题
 * 2. 连续收益最大化问题
 *
 * 异常处理：
 * 1. 处理空数组情况
 * 2. 处理数组长度为 1 的边界情况（无法交易）
 *
 * 工程化考量：
 * 1. 输入验证：检查数组是否为空或长度不足
 * 2. 边界条件：处理单元素数组
 * 3. 性能优化：一次遍历完成计算
 */
public class Code09_BestTimeToBuyAndSellStockII {

    /**
     * 计算能获得的最大利润
     *
     * @param prices 表示股票每天价格的数组
     * @return 最大利润
     */
    public static int maxProfit(int[] prices) {
        // 边界条件检查
        if (prices == null || prices.length <= 1) {
            return 0; // 数组为空或只有一天，无法交易
        }

        int totalProfit = 0; // 总利润

        // 遍历价格数组，从第二天开始
        for (int i = 1; i < prices.length; i++) {
            // 如果当天价格高于前一天，就累加差价作为利润
            if (prices[i] > prices[i - 1]) {
```

```
            totalProfit += prices[i] - prices[i - 1];
        }
    }

    return totalProfit;
}

/**
 * 测试函数，验证算法正确性
 */
public static void main(String[] args) {
    // 测试用例 1: 基本情况 - 有上涨趋势
    int[] prices1 = {7, 1, 5, 3, 6, 4};
    int result1 = maxProfit(prices1);
    System.out.println("测试用例 1:");
    System.out.print("价格数组: [");
    for (int i = 0; i < prices1.length; i++) {
        System.out.print(prices1[i]);
        if (i < prices1.length - 1) {
            System.out.print(", ");
        }
    }
    System.out.println("]");
    System.out.println("最大利润: " + result1);
    System.out.println("期望输出: 7");
    System.out.println();

    // 测试用例 2: 基本情况 - 持续上涨
    int[] prices2 = {1, 2, 3, 4, 5};
    int result2 = maxProfit(prices2);
    System.out.println("测试用例 2:");
    System.out.print("价格数组: [");
    for (int i = 0; i < prices2.length; i++) {
        System.out.print(prices2[i]);
        if (i < prices2.length - 1) {
            System.out.print(", ");
        }
    }
    System.out.println("]");
    System.out.println("最大利润: " + result2);
    System.out.println("期望输出: 4");
    System.out.println();
```

```
        // 测试用例 3: 基本情况 - 持续下跌
        int[] prices3 = {7, 6, 4, 3, 1};
        int result3 = maxProfit(prices3);
        System.out.println("测试用例 3:");
        System.out.print("价格数组: [");
        for (int i = 0; i < prices3.length; i++) {
            System.out.print(prices3[i]);
            if (i < prices3.length - 1) {
                System.out.print(", ");
            }
        }
        System.out.println("]");
        System.out.println("最大利润: " + result3);
        System.out.println("期望输出: 0");
        System.out.println();

        // 测试用例 4: 边界情况 - 单元素数组
        int[] prices4 = {5};
        int result4 = maxProfit(prices4);
        System.out.println("测试用例 4:");
        System.out.println("价格数组: [5]");
        System.out.println("最大利润: " + result4);
        System.out.println("期望输出: 0");
        System.out.println();

        // 测试用例 5: 复杂情况 - 波动较大
        int[] prices5 = {3, 3, 5, 0, 0, 3, 1, 4};
        int result5 = maxProfit(prices5);
        System.out.println("测试用例 5:");
        System.out.print("价格数组: [");
        for (int i = 0; i < prices5.length; i++) {
            System.out.print(prices5[i]);
            if (i < prices5.length - 1) {
                System.out.print(", ");
            }
        }
        System.out.println("]");
        System.out.println("最大利润: " + result5);
        System.out.println("期望输出: 8");
    }
}
```

==========

文件: Code09_BestTimeToBuyAndSellStockII.py
==========
买卖股票的最佳时机 II（Best Time to Buy and Sell Stock II）
题目来源：LeetCode 122
题目链接：https://leetcode.cn/problems/best-time-to-buy-and-sell-stock-ii/

"""
问题描述：
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。

算法思路：
使用贪心策略，只要明天的价格比今天高，就在今天买入，明天卖出。
具体步骤：
1. 遍历价格数组，从第二天开始
2. 如果当天价格高于前一天，就累加差价作为利润
3. 最终得到的就是最大利润

时间复杂度：O(n)，其中 n 是价格数组的长度，只需遍历数组一次
空间复杂度：O(1)，只使用了常数额外空间

是否最优解：是。贪心策略在此问题中能得到最优解。

适用场景：
1. 股票交易策略问题
2. 连续收益最大化问题

异常处理：
1. 处理空数组情况
2. 处理数组长度为 1 的边界情况（无法交易）

工程化考量：
1. 输入验证：检查数组是否为空或长度不足
2. 边界条件：处理单元素数组
3. 性能优化：一次遍历完成计算
"""

class Solution:
    def maxProfit(self, prices):
        """

```
        计算能获得的最大利润

        Args:
            prices: List[int] - 表示股票每天价格的数组

        Returns:
            int - 最大利润
        """
        # 边界条件检查
        if not prices or len(prices) <= 1:
            return 0  # 数组为空或只有一天，无法交易

        total_profit = 0  # 总利润

        # 遍历价格数组，从第二天开始
        for i in range(1, len(prices)):
            # 如果当天价格高于前一天，就累加差价作为利润
            if prices[i] > prices[i - 1]:
                total_profit += prices[i] - prices[i - 1]

        return total_profit

# 测试函数，验证算法正确性
def test_max_profit():
    solution = Solution()

    # 测试用例 1: 基本情况 - 有上涨趋势
    prices1 = [7, 1, 5, 3, 6, 4]
    result1 = solution.maxProfit(prices1)
    print("测试用例 1:")
    print(f"价格数组: {prices1}")
    print(f"最大利润: {result1}")
    print(f"期望输出: 7")
    print()

    # 测试用例 2: 基本情况 - 持续上涨
    prices2 = [1, 2, 3, 4, 5]
    result2 = solution.maxProfit(prices2)
    print("测试用例 2:")
    print(f"价格数组: {prices2}")
    print(f"最大利润: {result2}")
    print(f"期望输出: 4")
    print()
```

```
    # 测试用例 3: 基本情况 - 持续下跌
    prices3 = [7, 6, 4, 3, 1]
    result3 = solution.maxProfit(prices3)
    print("测试用例 3:")
    print(f"价格数组: {prices3}")
    print(f"最大利润: {result3}")
    print(f"期望输出: 0")
    print()

    # 测试用例 4: 边界情况 - 单元素数组
    prices4 = [5]
    result4 = solution.maxProfit(prices4)
    print("测试用例 4:")
    print(f"价格数组: {prices4}")
    print(f"最大利润: {result4}")
    print(f"期望输出: 0")
    print()

    # 测试用例 5: 复杂情况 - 波动较大
    prices5 = [3, 3, 5, 0, 0, 3, 1, 4]
    result5 = solution.maxProfit(prices5)
    print("测试用例 5:")
    print(f"价格数组: {prices5}")
    print(f"最大利润: {result5}")
    print(f"期望输出: 8")

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

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

文件: Code10_CanPlaceFlowers.cpp
==========================================
// 种花问题（Can Place Flowers）
// 题目来源：LeetCode 605
// 题目链接：https://leetcode.cn/problems/can-place-flowers/

/**
 * 问题描述：
 * 假设有一个很长的花坛，一部分地块种植了花，另一部分却没有。
 * 可是，花不能种植在相邻的地块上，它们会争夺水源，两者都会死去。
```

```
 * 给你一个整数数组 flowerbed 表示花坛，由若干 0 和 1 组成，
 * 其中 0 表示没种植花，1 表示种植了花。
 * 另有一个数 n ，能否在不打破种植规则的情况下种入 n 朵花？
 * 能则返回 true ，不能则返回 false 。
 *
 * 算法思路：
 * 使用贪心策略，尽可能多地种花，只要当前位置可以种花（当前位置是 0，且左右都是 0 或边界），就种花。
 * 具体步骤：
 * 1. 遍历花坛数组
 * 2. 对于每个位置，检查是否满足种花条件：
 *    - 当前位置是 0
 *    - 前一个位置是 0 或当前是第一个位置
 *    - 后一个位置是 0 或当前是最后一个位置
 * 3. 如果满足条件，就在当前位置种花（将 0 改为 1），并减少需要种的花数量
 * 4. 如果需要种的花数量减为 0，返回 true
 * 5. 遍历结束后，如果需要种的花数量已减为 0，返回 true，否则返回 false
 *
 * 时间复杂度：O(n)，其中 n 是花坛数组的长度，只需遍历数组一次
 * 空间复杂度：O(1)，只使用了常数额外空间
 *
 * 是否最优解：是。贪心策略在此问题中能得到最优解。
 *
 * 适用场景：
 * 1. 间隔种植问题
 * 2. 资源分配问题，需要满足相邻资源不能同时使用
 *
 * 异常处理：
 * 1. 处理空数组情况
 * 2. 处理 n 为 0 的边界情况（不需要种花，直接返回 true）
 *
 * 工程化考量：
 * 1. 输入验证：检查数组是否为空，检查 n 是否为非负数
 * 2. 边界条件：处理边界位置的种植判断
 * 3. 性能优化：一旦确认可以种植 n 朵花，立即返回结果
 */

#include <iostream>
#include <vector>

using namespace std;

/**
 * 判断是否能在不打破种植规则的情况下种入 n 朵花
```

```
 *
 * @param flowerbed 表示花坛的数组，0 表示没种植花，1 表示种植了花
 * @param n 需要种入的花数量
 * @return 如果能种入 n 朵花返回 true，否则返回 false
 */
bool canPlaceFlowers(vector<int>& flowerbed, int n) {
    // 边界条件检查
    if (flowerbed.empty()) {
        return n == 0; // 空花坛只能种 0 朵花
    }

    if (n <= 0) {
        return true; // 不需要种花，直接返回 true
    }

    int length = flowerbed.size();

    // 遍历花坛数组
    for (int i = 0; i < length; i++) {
        // 检查当前位置是否可以种花
        if (flowerbed[i] == 0) {
            // 检查左侧是否为空或边界
            bool leftEmpty = (i == 0) || (flowerbed[i - 1] == 0);
            // 检查右侧是否为空或边界
            bool rightEmpty = (i == length - 1) || (flowerbed[i + 1] == 0);

            if (leftEmpty && rightEmpty) {
                // 可以种花
                flowerbed[i] = 1; // 标记为已种花
                n--; // 减少需要种的花数量

                // 如果已经种完所有需要的花，返回 true
                if (n == 0) {
                    return true;
                }
            }
        }
    }

    // 遍历结束后，检查是否种完了所有需要的花
    return n == 0;
}
```

```
/**
 * 打印数组内容
 *
 * @param arr 要打印的数组
 * @param name 数组名称
 */
void printArray(const vector<int>& arr, const string& name) {
    cout << name << ": [";
    for (size_t i = 0; i < arr.size(); i++) {
        cout << arr[i];
        if (i < arr.size() - 1) {
            cout << ", ";
        }
    }
    cout << "]" << endl;
}

/**
 * 测试函数，验证算法正确性
 */
int testCanPlaceFlowers() {
    // 测试用例 1: 基本情况 - 可以种花
    vector<int> flowerbed1 = {1, 0, 0, 0, 1};
    vector<int> flowerbed1Copy = flowerbed1; // 创建副本以避免修改原始数组
    int n1 = 1;
    bool result1 = canPlaceFlowers(flowerbed1Copy, n1);
    cout << "测试用例 1:" << endl;
    printArray(flowerbed1, "花坛");
    cout << "需要种花数量: " << n1 << endl;
    cout << "能否种植: " << (result1 ? "true" : "false") << endl;
    cout << "期望输出: true" << endl << endl;

    // 测试用例 2: 基本情况 - 不能种花
    vector<int> flowerbed2 = {1, 0, 0, 0, 1};
    vector<int> flowerbed2Copy = flowerbed2;
    int n2 = 2;
    bool result2 = canPlaceFlowers(flowerbed2Copy, n2);
    cout << "测试用例 2:" << endl;
    printArray(flowerbed2, "花坛");
    cout << "需要种花数量: " << n2 << endl;
    cout << "能否种植: " << (result2 ? "true" : "false") << endl;
    cout << "期望输出: false" << endl << endl;
```

```
    // 测试用例 3: 边界情况 - n 为 0
    vector<int> flowerbed3 = {1, 0, 0, 0, 1};
    vector<int> flowerbed3Copy = flowerbed3;
    int n3 = 0;
    bool result3 = canPlaceFlowers(flowerbed3Copy, n3);
    cout << "测试用例 3:" << endl;
    printArray(flowerbed3, "花坛");
    cout << "需要种花数量: " << n3 << endl;
    cout << "能否种植: " << (result3 ? "true" : "false") << endl;
    cout << "期望输出: true" << endl << endl;

    // 测试用例 4: 边界情况 - 全为 0 的花坛
    vector<int> flowerbed4 = {0, 0, 0, 0};
    vector<int> flowerbed4Copy = flowerbed4;
    int n4 = 2;
    bool result4 = canPlaceFlowers(flowerbed4Copy, n4);
    cout << "测试用例 4:" << endl;
    printArray(flowerbed4, "花坛");
    cout << "需要种花数量: " << n4 << endl;
    cout << "能否种植: " << (result4 ? "true" : "false") << endl;
    cout << "期望输出: true" << endl << endl;

    // 测试用例 5: 边界情况 - 单元素花坛
    vector<int> flowerbed5 = {0};
    vector<int> flowerbed5Copy = flowerbed5;
    int n5 = 1;
    bool result5 = canPlaceFlowers(flowerbed5Copy, n5);
    cout << "测试用例 5:" << endl;
    printArray(flowerbed5, "花坛");
    cout << "需要种花数量: " << n5 << endl;
    cout << "能否种植: " << (result5 ? "true" : "false") << endl;
    cout << "期望输出: true" << endl;

    return 0;
}

int main() {
    return testCanPlaceFlowers();
}
```

==========

文件: Code10_CanPlaceFlowers.java

```
==========================================
package class093;

/**
 * 种花问题（Can Place Flowers）
 * 题目来源：LeetCode 605
 * 题目链接：https://leetcode.cn/problems/can-place-flowers/
 *
 * 问题描述：
 * 假设有一个很长的花坛，一部分地块种植了花，另一部分却没有。
 * 可是，花不能种植在相邻的地块上，它们会争夺水源，两者都会死去。
 * 给你一个整数数组 flowerbed 表示花坛，由若干 0 和 1 组成，
 * 其中 0 表示没种植花，1 表示种植了花。
 * 另有一个数 n ，能否在不打破种植规则的情况下种入 n 朵花？
 * 能则返回 true ，不能则返回 false 。
 *
 * 算法思路：
 * 使用贪心策略，尽可能多地种花，只要当前位置可以种花（当前位置是0，且左右都是0或边界），就种花。
 * 具体步骤：
 * 1. 遍历花坛数组
 * 2. 对于每个位置，检查是否满足种花条件：
 *    - 当前位置是0
 *    - 前一个位置是0或当前是第一个位置
 *    - 后一个位置是0或当前是最后一个位置
 * 3. 如果满足条件，就在当前位置种花（将0改为1），并减少需要种的花数量
 * 4. 如果需要种的花数量减为0，返回true
 * 5. 遍历结束后，如果需要种的花数量已减为0，返回true，否则返回false
 *
 * 时间复杂度：O(n)，其中n是花坛数组的长度，只需遍历数组一次
 * 空间复杂度：O(1)，只使用了常数额外空间
 *
 * 是否最优解：是。贪心策略在此问题中能得到最优解。
 *
 * 适用场景：
 * 1. 间隔种植问题
 * 2. 资源分配问题，需要满足相邻资源不能同时使用
 *
 * 异常处理：
 * 1. 处理空数组情况
 * 2. 处理n为0的边界情况（不需要种花，直接返回true）
 *
 * 工程化考量：
 * 1. 输入验证：检查数组是否为空，检查n是否为非负数
```

```
 * 2. 边界条件：处理边界位置的种植判断
 * 3. 性能优化：一旦确认可以种植 n 朵花，立即返回结果
 */
public class Code10_CanPlaceFlowers {

    /**
     * 判断是否能在不打破种植规则的情况下种入 n 朵花
     *
     * @param flowerbed 表示花坛的数组，0 表示没种植花，1 表示种植了花
     * @param n 需要种入的花数量
     * @return 如果能种入 n 朵花返回 true，否则返回 false
     */
    public static boolean canPlaceFlowers(int[] flowerbed, int n) {
        // 边界条件检查
        if (flowerbed == null || flowerbed.length == 0) {
            return n == 0; // 空花坛只能种 0 朵花
        }

        if (n <= 0) {
            return true; // 不需要种花，直接返回 true
        }

        int length = flowerbed.length;

        // 遍历花坛数组
        for (int i = 0; i < length; i++) {
            // 检查当前位置是否可以种花
            if (flowerbed[i] == 0) {
                // 检查左侧是否为空或边界
                boolean leftEmpty = (i == 0) || (flowerbed[i - 1] == 0);
                // 检查右侧是否为空或边界
                boolean rightEmpty = (i == length - 1) || (flowerbed[i + 1] == 0);

                if (leftEmpty && rightEmpty) {
                    // 可以种花
                    flowerbed[i] = 1; // 标记为已种花
                    n--; // 减少需要种的花数量

                    // 如果已经种完所有需要的花，返回 true
                    if (n == 0) {
                        return true;
                    }
                }
```

```
        }
    }

    // 遍历结束后，检查是否种完了所有需要的花
    return n == 0;
}

/**
 * 测试函数，验证算法正确性
 */
public static void main(String[] args) {
    // 测试用例 1: 基本情况 - 可以种花
    int[] flowerbed1 = {1, 0, 0, 0, 1};
    int n1 = 1;
    boolean result1 = canPlaceFlowers(flowerbed1, n1);
    System.out.println("测试用例 1:");
    System.out.print("花坛: [");
    for (int i = 0; i < flowerbed1.length; i++) {
        System.out.print(flowerbed1[i]);
        if (i < flowerbed1.length - 1) {
            System.out.print(", ");
        }
    }
    System.out.println("]");
    System.out.println("需要种花数量: " + n1);
    System.out.println("能否种植: " + result1);
    System.out.println("期望输出: true");
    System.out.println();

    // 测试用例 2: 基本情况 - 不能种花
    int[] flowerbed2 = {1, 0, 0, 0, 1};
    int n2 = 2;
    boolean result2 = canPlaceFlowers(flowerbed2, n2);
    System.out.println("测试用例 2:");
    System.out.print("花坛: [");
    for (int i = 0; i < flowerbed2.length; i++) {
        System.out.print(flowerbed2[i]);
        if (i < flowerbed2.length - 1) {
            System.out.print(", ");
        }
    }
    System.out.println("]");
    System.out.println("需要种花数量: " + n2);
```

```
System.out.println("能否种植: " + result2);
System.out.println("期望输出: false");
System.out.println();

// 测试用例 3: 边界情况 - n 为 0
int[] flowerbed3 = {1, 0, 0, 0, 1};
int n3 = 0;
boolean result3 = canPlaceFlowers(flowerbed3, n3);
System.out.println("测试用例 3:");
System.out.print("花坛: [");
for (int i = 0; i < flowerbed3.length; i++) {
    System.out.print(flowerbed3[i]);
    if (i < flowerbed3.length - 1) {
        System.out.print(", ");
    }
}
System.out.println("]");
System.out.println("需要种花数量: " + n3);
System.out.println("能否种植: " + result3);
System.out.println("期望输出: true");
System.out.println();

// 测试用例 4: 边界情况 - 全为 0 的花坛
int[] flowerbed4 = {0, 0, 0, 0};
int n4 = 2;
boolean result4 = canPlaceFlowers(flowerbed4, n4);
System.out.println("测试用例 4:");
System.out.print("花坛: [");
for (int i = 0; i < flowerbed4.length; i++) {
    System.out.print(flowerbed4[i]);
    if (i < flowerbed4.length - 1) {
        System.out.print(", ");
    }
}
System.out.println("]");
System.out.println("需要种花数量: " + n4);
System.out.println("能否种植: " + result4);
System.out.println("期望输出: true");
System.out.println();

// 测试用例 5: 边界情况 - 单元素花坛
int[] flowerbed5 = {0};
int n5 = 1;
```

```
        boolean result5 = canPlaceFlowers(flowerbed5, n5);
        System.out.println("测试用例 5:");
        System.out.print("花坛: [");
        System.out.print(flowerbed5[0]);
        System.out.println("]");
        System.out.println("需要种花数量: " + n5);
        System.out.println("能否种植: " + result5);
        System.out.println("期望输出: true");
    }
}
```

==========

文件: Code10_CanPlaceFlowers.py

==========

```
# 种花问题（Can Place Flowers）
# 题目来源：LeetCode 605
# 题目链接：https://leetcode.cn/problems/can-place-flowers/

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

算法思路：
使用贪心策略，尽可能多地种花，只要当前位置可以种花（当前位置是 0，且左右都是 0 或边界），就种花。
具体步骤：
1. 遍历花坛数组
2. 对于每个位置，检查是否满足种花条件：
   - 当前位置是 0
   - 前一个位置是 0 或当前是第一个位置
   - 后一个位置是 0 或当前是最后一个位置
3. 如果满足条件，就在当前位置种花（将 0 改为 1），并减少需要种的花数量
4. 如果需要种的花数量减为 0，返回 true
5. 遍历结束后，如果需要种的花数量已减为 0，返回 true，否则返回 false

时间复杂度：O(n)，其中 n 是花坛数组的长度，只需遍历数组一次
空间复杂度：O(1)，只使用了常数额外空间
```

是否最优解：是。贪心策略在此问题中能得到最优解。

适用场景：
1. 间隔种植问题
2. 资源分配问题，需要满足相邻资源不能同时使用

异常处理：
1. 处理空数组情况
2. 处理 n 为 0 的边界情况（不需要种花，直接返回 true）

工程化考量：
1. 输入验证：检查数组是否为空，检查 n 是否为非负数
2. 边界条件：处理边界位置的种植判断
3. 性能优化：一旦确认可以种植 n 朵花，立即返回结果
"""

```
class Solution:
    def canPlaceFlowers(self, flowerbed, n):
        """
        判断是否能在不打破种植规则的情况下种入 n 朵花

        Args:
            flowerbed: List[int] - 表示花坛的数组，0 表示没种植花，1 表示种植了花
            n: int - 需要种入的花数量

        Returns:
            bool - 如果能种入 n 朵花返回 True，否则返回 False
        """
        # 边界条件检查
        if not flowerbed:
            return n == 0  # 空花坛只能种 0 朵花

        if n <= 0:
            return True  # 不需要种花，直接返回 True

        length = len(flowerbed)

        # 遍历花坛数组
        for i in range(length):
            # 检查当前位置是否可以种花
            if flowerbed[i] == 0:
                # 检查左侧是否为空或边界
                left_empty = (i == 0) or (flowerbed[i - 1] == 0)
```

```
                # 检查右侧是否为空或边界
                right_empty = (i == length - 1) or (flowerbed[i + 1] == 0)

                if left_empty and right_empty:
                    # 可以种花
                    flowerbed[i] = 1  # 标记为已种花
                    n -= 1  # 减少需要种的花数量

                    # 如果已经种完所有需要的花，返回 True
                    if n == 0:
                        return True

        # 遍历结束后，检查是否种完了所有需要的花
        return n == 0

# 测试函数，验证算法正确性
def test_can_place_flowers():
    solution = Solution()

    # 测试用例 1: 基本情况 - 可以种花
    flowerbed1 = [1, 0, 0, 0, 1]
    n1 = 1
    # 创建副本以避免修改原始数组
    flowerbed1_copy = flowerbed1.copy()
    result1 = solution.canPlaceFlowers(flowerbed1_copy, n1)
    print("测试用例 1:")
    print(f"花坛: {flowerbed1}")
    print(f"需要种花数量: {n1}")
    print(f"能否种植: {result1}")
    print(f"期望输出: True")
    print()

    # 测试用例 2: 基本情况 - 不能种花
    flowerbed2 = [1, 0, 0, 0, 1]
    n2 = 2
    flowerbed2_copy = flowerbed2.copy()
    result2 = solution.canPlaceFlowers(flowerbed2_copy, n2)
    print("测试用例 2:")
    print(f"花坛: {flowerbed2}")
    print(f"需要种花数量: {n2}")
    print(f"能否种植: {result2}")
    print(f"期望输出: False")
    print()
```

```
    # 测试用例 3: 边界情况 - n 为 0
    flowerbed3 = [1, 0, 0, 0, 1]
    n3 = 0
    flowerbed3_copy = flowerbed3.copy()
    result3 = solution.canPlaceFlowers(flowerbed3_copy, n3)
    print("测试用例 3:")
    print(f"花坛: {flowerbed3}")
    print(f"需要种花数量: {n3}")
    print(f"能否种植: {result3}")
    print(f"期望输出: True")
    print()

    # 测试用例 4: 边界情况 - 全为 0 的花坛
    flowerbed4 = [0, 0, 0, 0]
    n4 = 2
    flowerbed4_copy = flowerbed4.copy()
    result4 = solution.canPlaceFlowers(flowerbed4_copy, n4)
    print("测试用例 4:")
    print(f"花坛: {flowerbed4}")
    print(f"需要种花数量: {n4}")
    print(f"能否种植: {result4}")
    print(f"期望输出: True")
    print()

    # 测试用例 5: 边界情况 - 单元素花坛
    flowerbed5 = [0]
    n5 = 1
    flowerbed5_copy = flowerbed5.copy()
    result5 = solution.canPlaceFlowers(flowerbed5_copy, n5)
    print("测试用例 5:")
    print(f"花坛: {flowerbed5}")
    print(f"需要种花数量: {n5}")
    print(f"能否种植: {result5}")
    print(f"期望输出: True")

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

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

文件: Code11_GasStation.cpp
```

```
==========================================
// 加油站（Gas Station）
// 题目来源：LeetCode 134
// 题目链接：https://leetcode.cn/problems/gas-station/
//
// 问题描述：
// 在一条环路上有 N 个加油站，每个加油站有汽油 gas[i]和消耗 cost[i]。
// 从某个加油站出发，按顺序访问每个加油站，判断是否能绕环路行驶一周。
// 如果可以，返回出发时加油站的编号，否则返回-1。
//
// 算法思路：
// 使用贪心策略，关键观察：如果从加油站 i 出发无法到达加油站 j，那么从 i 到 j 之间的任意加油站出发都无法到达 j。
// 具体步骤：
// 1. 计算总油量是否足够绕环路一周，如果总油量小于总消耗，直接返回-1
// 2. 遍历所有加油站，维护当前油量和总油量
// 3. 如果当前油量小于 0，说明从当前起点无法到达下一个加油站，重置起点为下一个加油站
// 4. 最终返回起点位置
//
// 时间复杂度：O(n) - 只需遍历数组一次
// 空间复杂度：O(1) - 只使用了常数额外空间
//
// 是否最优解：是。这是该问题的最优解法。
//
// 适用场景：
// 1. 环路行驶问题
// 2. 资源约束下的可达性判断
//
// 异常处理：
// 1. 处理空数组情况
// 2. 处理数组长度不一致情况
//
// 工程化考量：
// 1. 输入验证：检查数组是否为空，检查数组长度是否一致
// 2. 边界条件：处理单元素数组
// 3. 性能优化：一次遍历完成计算
//
// 相关题目：
// 1. LeetCode 45. 跳跃游戏 II - 经典跳跃游戏
// 2. LeetCode 55. 跳跃游戏 - 判断是否能到达终点
// 3. LeetCode 871. 最低加油次数 - 加油站问题的变种
// 4. 牛客网 NC48 跳跃游戏 - 与 LeetCode 55 相同
// 5. LintCode 117. 跳跃游戏 II - 与 LeetCode 45 相同
```

```
// 6. HackerRank - Jumping on the Clouds - 简化版跳跃游戏
// 7. CodeChef - JUMP - 类似跳跃游戏的变种
// 8. AtCoder ABC161D - Lunlun Number - BFS搜索相关
// 9. Codeforces 1324C - Frog Jumps - 贪心跳跃问题
// 10. POJ 1700 - Crossing River - 经典过河问题

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

/**
 * 计算能够绕环路行驶一周的起始加油站
 *
 * @param gas 每个加油站的汽油量数组
 * @param cost 每个加油站到下一个加油站的消耗数组
 * @return 起始加油站索引，如果无法完成返回-1
 */
int canCompleteCircuit(vector<int>& gas, vector<int>& cost) {
    // 边界条件检查
    if (gas.empty() || cost.empty() || gas.size() != cost.size()) {
        return -1;
    }

    int n = gas.size();
    int totalGas = 0;      // 总油量
    int currentGas = 0;    // 当前油量
    int startStation = 0;  // 起始加油站

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

        // 如果当前油量小于0，说明从startStation出发无法到达i+1
        if (currentGas < 0) {
            // 重置起始加油站为下一个加油站
            startStation = i + 1;
            currentGas = 0;
        }
    }

    // 如果总油量大于等于0，说明存在解，否则返回-1
    return totalGas >= 0 ? startStation : -1;
}
```

```
/**
 * 测试函数，验证算法正确性
 */
int main() {
    // 测试用例 1: 基本情况 - 可以完成
    vector<int> gas1 = {1, 2, 3, 4, 5};
    vector<int> cost1 = {3, 4, 5, 1, 2};
    int result1 = canCompleteCircuit(gas1, cost1);
    cout << "测试用例 1:" << endl;
    cout << "汽油量: [";
    for (int i = 0; i < gas1.size(); i++) {
        cout << gas1[i];
        if (i < gas1.size() - 1) cout << ", ";
    }
    cout << "]" << endl;
    cout << "消耗量: [";
    for (int i = 0; i < cost1.size(); i++) {
        cout << cost1[i];
        if (i < cost1.size() - 1) cout << ", ";
    }
    cout << "]" << endl;
    cout << "起始加油站: " << result1 << endl;
    cout << "期望输出: 3" << endl << endl;

    // 测试用例 2: 基本情况 - 无法完成
    vector<int> gas2 = {2, 3, 4};
    vector<int> cost2 = {3, 4, 3};
    int result2 = canCompleteCircuit(gas2, cost2);
    cout << "测试用例 2:" << endl;
    cout << "汽油量: [";
    for (int i = 0; i < gas2.size(); i++) {
        cout << gas2[i];
        if (i < gas2.size() - 1) cout << ", ";
    }
    cout << "]" << endl;
    cout << "消耗量: [";
    for (int i = 0; i < cost2.size(); i++) {
        cout << cost2[i];
        if (i < cost2.size() - 1) cout << ", ";
    }
    cout << "]" << endl;
    cout << "起始加油站: " << result2 << endl;
```

```
cout << "期望输出: -1" << endl << endl;

// 测试用例 3: 边界情况 - 单元素数组
vector<int> gas3 = {5};
vector<int> cost3 = {4};
int result3 = canCompleteCircuit(gas3, cost3);
cout << "测试用例 3:" << endl;
cout << "汽油量: [";
cout << gas3[0];
cout << "]" << endl;
cout << "消耗量: [";
cout << cost3[0];
cout << "]" << endl;
cout << "起始加油站: " << result3 << endl;
cout << "期望输出: 0" << endl << endl;

// 测试用例 4: 复杂情况 - 多个可行解
vector<int> gas4 = {3, 1, 1};
vector<int> cost4 = {1, 2, 2};
int result4 = canCompleteCircuit(gas4, cost4);
cout << "测试用例 4:" << endl;
cout << "汽油量: [";
for (int i = 0; i < gas4.size(); i++) {
    cout << gas4[i];
    if (i < gas4.size() - 1) cout << ", ";
}
cout << "]" << endl;
cout << "消耗量: [";
for (int i = 0; i < cost4.size(); i++) {
    cout << cost4[i];
    if (i < cost4.size() - 1) cout << ", ";
}
cout << "]" << endl;
cout << "起始加油站: " << result4 << endl;
cout << "期望输出: 0" << endl << endl;

// 测试用例 5: 边界情况 - 空数组
vector<int> gas5 = {};
vector<int> cost5 = {};
int result5 = canCompleteCircuit(gas5, cost5);
cout << "测试用例 5:" << endl;
cout << "汽油量: []" << endl;
cout << "消耗量: []" << endl;
```

```
    cout << "起始加油站: " << result5 << endl;
    cout << "期望输出: -1" << endl;

    return 0;
}
```

==========

文件: Code11_GasStation.java

==========

```
package class093;

/**
 * 加油站（Gas Station）
 * 题目来源：LeetCode 134
 * 题目链接：https://leetcode.cn/problems/gas-station/
 *
 * 问题描述：
 * 在一条环路上有 N 个加油站，每个加油站有汽油 gas[i]和消耗 cost[i]。
 * 从某个加油站出发，按顺序访问每个加油站，判断是否能绕环路行驶一周。
 * 如果可以，返回出发时加油站的编号，否则返回-1。
 *
 * 算法思路：
 * 使用贪心策略，关键观察：如果从加油站 i 出发无法到达加油站 j，那么从 i 到 j 之间的任意加油站出发都无法到达 j。
 * 具体步骤：
 * 1. 计算总油量是否足够绕环路一周，如果总油量小于总消耗，直接返回-1
 * 2. 遍历所有加油站，维护当前油量和总油量
 * 3. 如果当前油量小于 0，说明从当前起点无法到达下一个加油站，重置起点为下一个加油站
 * 4. 最终返回起点位置
 *
 * 时间复杂度：O(n) - 只需遍历数组一次
 * 空间复杂度：O(1) - 只使用了常数额外空间
 *
 * 是否最优解：是。这是该问题的最优解法。
 *
 * 适用场景：
 * 1. 环路行驶问题
 * 2. 资源约束下的可达性判断
 *
 * 异常处理：
 * 1. 处理空数组情况
 * 2. 处理数组长度不一致情况
```

```
 *
 * 工程化考量：
 * 1. 输入验证：检查数组是否为空，检查数组长度是否一致
 * 2. 边界条件：处理单元素数组
 * 3. 性能优化：一次遍历完成计算
 *
 * 相关题目：
 * 1. LeetCode 45. 跳跃游戏 II - 经典跳跃游戏
 * 2. LeetCode 55. 跳跃游戏 - 判断是否能到达终点
 * 3. LeetCode 871. 最低加油次数 - 加油站问题的变种
 * 4. 牛客网 NC48 跳跃游戏 - 与 LeetCode 55 相同
 * 5. LintCode 117. 跳跃游戏 II - 与 LeetCode 45 相同
 * 6. HackerRank - Jumping on the Clouds - 简化版跳跃游戏
 * 7. CodeChef - JUMP - 类似跳跃游戏的变种
 * 8. AtCoder ABC161D - Lunlun Number - BFS 搜索相关
 * 9. Codeforces 1324C - Frog Jumps - 贪心跳跃问题
 * 10. POJ 1700 - Crossing River - 经典过河问题
 */
public class Code11_GasStation {

    /**
     * 计算能够绕环路行驶一周的起始加油站
     *
     * @param gas 每个加油站的汽油量数组
     * @param cost 每个加油站到下一个加油站的消耗数组
     * @return 起始加油站索引，如果无法完成返回-1
     */
    public static int canCompleteCircuit(int[] gas, int[] cost) {
        // 边界条件检查
        if (gas == null || cost == null || gas.length == 0 || cost.length == 0 || gas.length !=
cost.length) {
            return -1;
        }

        int n = gas.length;
        int totalGas = 0;      // 总油量
        int currentGas = 0;    // 当前油量
        int startStation = 0;   // 起始加油站

        for (int i = 0; i < n; i++) {
            totalGas += gas[i] - cost[i];
            currentGas += gas[i] - cost[i];
```

```
            // 如果当前油量小于0，说明从startStation出发无法到达i+1
            if (currentGas < 0) {
                // 重置起始加油站为下一个加油站
                startStation = i + 1;
                currentGas = 0;
            }
        }

        // 如果总油量大于等于0，说明存在解，否则返回-1
        return totalGas >= 0 ? startStation : -1;
    }

    /**
     * 测试函数，验证算法正确性
     */
    public static void main(String[] args) {
        // 测试用例1: 基本情况 - 可以完成
        int[] gas1 = {1, 2, 3, 4, 5};
        int[] cost1 = {3, 4, 5, 1, 2};
        int result1 = canCompleteCircuit(gas1, cost1);
        System.out.println("测试用例1:");
        System.out.print("汽油量: [");
        for (int i = 0; i < gas1.length; i++) {
            System.out.print(gas1[i]);
            if (i < gas1.length - 1) System.out.print(", ");
        }
        System.out.println("]");
        System.out.print("消耗量: [");
        for (int i = 0; i < cost1.length; i++) {
            System.out.print(cost1[i]);
            if (i < cost1.length - 1) System.out.print(", ");
        }
        System.out.println("]");
        System.out.println("起始加油站: " + result1);
        System.out.println("期望输出: 3");
        System.out.println();

        // 测试用例2: 基本情况 - 无法完成
        int[] gas2 = {2, 3, 4};
        int[] cost2 = {3, 4, 3};
        int result2 = canCompleteCircuit(gas2, cost2);
        System.out.println("测试用例2:");
        System.out.print("汽油量: [");
```

```
for (int i = 0; i < gas2.length; i++) {
    System.out.print(gas2[i]);
    if (i < gas2.length - 1) System.out.print(", ");
}
System.out.println("]");
System.out.print("消耗量: [");
for (int i = 0; i < cost2.length; i++) {
    System.out.print(cost2[i]);
    if (i < cost2.length - 1) System.out.print(", ");
}
System.out.println("]");
System.out.println("起始加油站: " + result2);
System.out.println("期望输出: -1");
System.out.println();

// 测试用例 3: 边界情况 - 单元素数组
int[] gas3 = {5};
int[] cost3 = {4};
int result3 = canCompleteCircuit(gas3, cost3);
System.out.println("测试用例 3:");
System.out.print("汽油量: [");
System.out.print(gas3[0]);
System.out.println("]");
System.out.print("消耗量: [");
System.out.print(cost3[0]);
System.out.println("]");
System.out.println("起始加油站: " + result3);
System.out.println("期望输出: 0");
System.out.println();

// 测试用例 4: 复杂情况 - 多个可行解
int[] gas4 = {3, 1, 1};
int[] cost4 = {1, 2, 2};
int result4 = canCompleteCircuit(gas4, cost4);
System.out.println("测试用例 4:");
System.out.print("汽油量: [");
for (int i = 0; i < gas4.length; i++) {
    System.out.print(gas4[i]);
    if (i < gas4.length - 1) System.out.print(", ");
}
System.out.println("]");
System.out.print("消耗量: [");
for (int i = 0; i < cost4.length; i++) {
```

```
            System.out.print(cost4[i]);
            if (i < cost4.length - 1) System.out.print(", ");
        }
        System.out.println("]");
        System.out.println("起始加油站: " + result4);
        System.out.println("期望输出: 0");
        System.out.println();

        // 测试用例 5: 边界情况 - 空数组
        int[] gas5 = {};
        int[] cost5 = {};
        int result5 = canCompleteCircuit(gas5, cost5);
        System.out.println("测试用例 5:");
        System.out.println("汽油量: []");
        System.out.println("消耗量: []");
        System.out.println("起始加油站: " + result5);
        System.out.println("期望输出: -1");
    }
}
```

==========

文件: Code11_GasStation.py

==========

```
# 加油站（Gas Station）
# 题目来源：LeetCode 134
# 题目链接：https://leetcode.cn/problems/gas-station/
#
# 问题描述：
# 在一条环路上有 N 个加油站，每个加油站有汽油 gas[i]和消耗 cost[i]。
# 从某个加油站出发，按顺序访问每个加油站，判断是否能绕环路行驶一周。
# 如果可以，返回出发时加油站的编号，否则返回-1。
#
# 算法思路：
# 使用贪心策略，关键观察：如果从加油站 i 出发无法到达加油站 j，那么从 i 到 j 之间的任意加油站出发都无法到达 j。
# 具体步骤：
# 1. 计算总油量是否足够绕环路一周，如果总油量小于总消耗，直接返回-1
# 2. 遍历所有加油站，维护当前油量和总油量
# 3. 如果当前油量小于 0，说明从当前起点无法到达下一个加油站，重置起点为下一个加油站
# 4. 最终返回起点位置
#
# 时间复杂度：O(n) - 只需遍历数组一次
```

```
# 空间复杂度：O(1) - 只使用了常数额外空间
#
# 是否最优解：是。这是该问题的最优解法。
#
# 适用场景：
# 1. 环路行驶问题
# 2. 资源约束下的可达性判断
#
# 异常处理：
# 1. 处理空数组情况
# 2. 处理数组长度不一致情况
#
# 工程化考量：
# 1. 输入验证：检查数组是否为空，检查数组长度是否一致
# 2. 边界条件：处理单元素数组
# 3. 性能优化：一次遍历完成计算
#
# 相关题目：
# 1. LeetCode 45. 跳跃游戏 II - 经典跳跃游戏
# 2. LeetCode 55. 跳跃游戏 - 判断是否能到达终点
# 3. LeetCode 871. 最低加油次数 - 加油站问题的变种
# 4. 牛客网 NC48 跳跃游戏 - 与 LeetCode 55 相同
# 5. LintCode 117. 跳跃游戏 II - 与 LeetCode 45 相同
# 6. HackerRank - Jumping on the Clouds - 简化版跳跃游戏
# 7. CodeChef - JUMP - 类似跳跃游戏的变种
# 8. AtCoder ABC161D - Lunlun Number - BFS 搜索相关
# 9. Codeforces 1324C - Frog Jumps - 贪心跳跃问题
# 10. POJ 1700 - Crossing River - 经典过河问题

class Solution:
    """
    计算能够绕环路行驶一周的起始加油站

    Args:
        gas: List[int] - 每个加油站的汽油量数组
        cost: List[int] - 每个加油站到下一个加油站的消耗数组

    Returns:
        int - 起始加油站索引，如果无法完成返回-1
    """
    def canCompleteCircuit(self, gas, cost):
        # 边界条件检查
        if not gas or not cost or len(gas) != len(cost):
```

```
            return -1

        n = len(gas)
        total_gas = 0      # 总油量
        current_gas = 0    # 当前油量
        start_station = 0   # 起始加油站

        for i in range(n):
            total_gas += gas[i] - cost[i]
            current_gas += gas[i] - cost[i]

            # 如果当前油量小于0，说明从start_station出发无法到达i+1
            if current_gas < 0:
                # 重置起始加油站为下一个加油站
                start_station = i + 1
                current_gas = 0

        # 如果总油量大于等于0，说明存在解，否则返回-1
        return start_station if total_gas >= 0 else -1


def main():
    solution = Solution()

    # 测试用例1: 基本情况 - 可以完成
    gas1 = [1, 2, 3, 4, 5]
    cost1 = [3, 4, 5, 1, 2]
    result1 = solution.canCompleteCircuit(gas1, cost1)
    print("测试用例1:")
    print(f"汽油量: {gas1}")
    print(f"消耗量: {cost1}")
    print(f"起始加油站: {result1}")
    print("期望输出: 3")
    print()

    # 测试用例2: 基本情况 - 无法完成
    gas2 = [2, 3, 4]
    cost2 = [3, 4, 3]
    result2 = solution.canCompleteCircuit(gas2, cost2)
    print("测试用例2:")
    print(f"汽油量: {gas2}")
    print(f"消耗量: {cost2}")
    print(f"起始加油站: {result2}")
```

```
    print("期望输出: -1")
    print()

    # 测试用例 3: 边界情况 - 单元素数组
    gas3 = [5]
    cost3 = [4]
    result3 = solution.canCompleteCircuit(gas3, cost3)
    print("测试用例 3:")
    print(f"汽油量: {gas3}")
    print(f"消耗量: {cost3}")
    print(f"起始加油站: {result3}")
    print("期望输出: 0")
    print()

    # 测试用例 4: 复杂情况 - 多个可行解
    gas4 = [3, 1, 1]
    cost4 = [1, 2, 2]
    result4 = solution.canCompleteCircuit(gas4, cost4)
    print("测试用例 4:")
    print(f"汽油量: {gas4}")
    print(f"消耗量: {cost4}")
    print(f"起始加油站: {result4}")
    print("期望输出: 0")
    print()

    # 测试用例 5: 边界情况 - 空数组
    gas5 = []
    cost5 = []
    result5 = solution.canCompleteCircuit(gas5, cost5)
    print("测试用例 5:")
    print(f"汽油量: {gas5}")
    print(f"消耗量: {cost5}")
    print(f"起始加油站: {result5}")
    print("期望输出: -1")


if __name__ == "__main__":
    main()

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

文件: Code12_Candy.cpp
==========================================
```

```
// 分发糖果（Candy）
// 题目来源：LeetCode 135
// 题目链接：https://leetcode.cn/problems/candy/
//
// 问题描述：
// 老师想给孩子们分发糖果，有 N 个孩子站成了一条直线，每个孩子至少分配到 1 个糖果。
// 相邻的孩子中，评分高的孩子必须获得更多的糖果。
// 计算最少需要准备多少糖果。
//
// 算法思路：
// 使用贪心策略，两次遍历：
// 1. 从左到右遍历，如果当前孩子评分比左边高，糖果数比左边多 1
// 2. 从右到左遍历，如果当前孩子评分比右边高，糖果数取当前值和右边值+1 的最大值
// 3. 最后统计所有糖果数之和
//
// 时间复杂度：O(n) - 两次遍历数组
// 空间复杂度：O(n) - 需要额外的糖果数组
//
// 是否最优解：是。这是该问题的最优解法。
//
// 适用场景：
// 1. 分配问题，需要满足相邻约束条件
// 2. 双向约束的最优化问题
//
// 异常处理：
// 1. 处理空数组情况
// 2. 处理单元素数组
//
// 工程化考量：
// 1. 输入验证：检查数组是否为空
// 2. 边界条件：处理单元素和双元素数组
// 3. 性能优化：使用数组而不是列表提高性能
//
// 相关题目：
// 1. LeetCode 42. 接雨水 - 双向遍历的经典问题
// 2. LeetCode 84. 柱状图中最大的矩形 - 单调栈应用
// 3. LeetCode 406. 根据身高重建队列 - 贪心排序问题
// 4. 牛客网 NC140 排序 - 各种排序算法实现
// 5. LintCode 391. 数飞机 - 区间调度相关
// 6. HackerRank - Jim and the Orders - 贪心调度问题
// 7. CodeChef - TACHSTCK - 区间配对问题
// 8. AtCoder ABC104C - All Green - 动态规划相关
// 9. Codeforces 1363C - Game On Leaves - 博弈论相关
```

```
// 10. POJ 3169 - Layout - 差分约束系统

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

/**
 * 计算最少需要准备的糖果数量
 *
 * @param ratings 孩子的评分数组
 * @return 最少需要的糖果数量
 */
int candy(vector<int>& ratings) {
    // 边界条件检查
    if (ratings.empty()) {
        return 0;
    }

    int n = ratings.size();
    if (n == 1) {
        return 1; // 只有一个孩子，最少需要 1 个糖果
    }

    vector<int> candies(n, 1); // 每个孩子至少 1 个糖果

    // 从左到右遍历，处理递增序列
    for (int i = 1; i < n; i++) {
        if (ratings[i] > ratings[i - 1]) {
            candies[i] = candies[i - 1] + 1;
        }
    }

    // 从右到左遍历，处理递减序列
    for (int i = n - 2; i >= 0; i--) {
        if (ratings[i] > ratings[i + 1]) {
            candies[i] = max(candies[i], candies[i + 1] + 1);
        }
    }

    // 统计总糖果数
    int totalCandies = 0;
    for (int candy : candies) {
```

```
        totalCandies += candy;
    }

    return totalCandies;
}

/**
 * 测试函数，验证算法正确性
 */
int main() {
    // 测试用例 1: 基本情况 - 递增序列
    vector<int> ratings1 = {1, 0, 2};
    int result1 = candy(ratings1);
    cout << "测试用例 1:" << endl;
    cout << "评分数组: [";
    for (int i = 0; i < ratings1.size(); i++) {
        cout << ratings1[i];
        if (i < ratings1.size() - 1) cout << ", ";
    }
    cout << "]" << endl;
    cout << "最少糖果数: " << result1 << endl;
    cout << "期望输出: 5" << endl << endl;

    // 测试用例 2: 基本情况 - 递减序列
    vector<int> ratings2 = {1, 2, 2};
    int result2 = candy(ratings2);
    cout << "测试用例 2:" << endl;
    cout << "评分数组: [";
    for (int i = 0; i < ratings2.size(); i++) {
        cout << ratings2[i];
        if (i < ratings2.size() - 1) cout << ", ";
    }
    cout << "]" << endl;
    cout << "最少糖果数: " << result2 << endl;
    cout << "期望输出: 4" << endl << endl;

    // 测试用例 3: 复杂情况 - 山峰形状
    vector<int> ratings3 = {1, 3, 2, 2, 1};
    int result3 = candy(ratings3);
    cout << "测试用例 3:" << endl;
    cout << "评分数组: [";
    for (int i = 0; i < ratings3.size(); i++) {
        cout << ratings3[i];
```

```
    if (i < ratings3.size() - 1) cout << ", ";
}
cout << "]" << endl;
cout << "最少糖果数: " << result3 << endl;
cout << "期望输出: 7" << endl << endl;

// 测试用例 4: 边界情况 - 单元素数组
vector<int> ratings4 = {5};
int result4 = candy(ratings4);
cout << "测试用例 4:" << endl;
cout << "评分数组: [";
cout << ratings4[0];
cout << "]" << endl;
cout << "最少糖果数: " << result4 << endl;
cout << "期望输出: 1" << endl << endl;

// 测试用例 5: 边界情况 - 两个相同评分
vector<int> ratings5 = {2, 2};
int result5 = candy(ratings5);
cout << "测试用例 5:" << endl;
cout << "评分数组: [";
for (int i = 0; i < ratings5.size(); i++) {
    cout << ratings5[i];
    if (i < ratings5.size() - 1) cout << ", ";
}
cout << "]" << endl;
cout << "最少糖果数: " << result5 << endl;
cout << "期望输出: 2" << endl << endl;

// 测试用例 6: 复杂情况 - 长序列
vector<int> ratings6 = {1, 2, 87, 87, 87, 2, 1};
int result6 = candy(ratings6);
cout << "测试用例 6:" << endl;
cout << "评分数组: [";
for (int i = 0; i < ratings6.size(); i++) {
    cout << ratings6[i];
    if (i < ratings6.size() - 1) cout << ", ";
}
cout << "]" << endl;
cout << "最少糖果数: " << result6 << endl;
cout << "期望输出: 13" << endl;

return 0;
```

```
}

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

文件: Code12_Candy.java
==========================================
package class093;

import java.util.Arrays;

/**
 * 分发糖果（Candy）
 * 题目来源：LeetCode 135
 * 题目链接：https://leetcode.cn/problems/candy/
 *
 * 问题描述：
 * 老师想给孩子们分发糖果，有 N 个孩子站成了一条直线，每个孩子至少分配到 1 个糖果。
 * 相邻的孩子中，评分高的孩子必须获得更多的糖果。
 * 计算最少需要准备多少糖果。
 *
 * 算法思路：
 * 使用贪心策略，两次遍历：
 * 1. 从左到右遍历，如果当前孩子评分比左边高，糖果数比左边多 1
 * 2. 从右到左遍历，如果当前孩子评分比右边高，糖果数取当前值和右边值+1 的最大值
 * 3. 最后统计所有糖果数之和
 *
 * 时间复杂度：O(n) - 两次遍历数组
 * 空间复杂度：O(n) - 需要额外的糖果数组
 *
 * 是否最优解：是。这是该问题的最优解法。
 *
 * 适用场景：
 * 1. 分配问题，需要满足相邻约束条件
 * 2. 双向约束的最优化问题
 *
 * 异常处理：
 * 1. 处理空数组情况
 * 2. 处理单元素数组
 *
 * 工程化考量：
 * 1. 输入验证：检查数组是否为空
 * 2. 边界条件：处理单元素和双元素数组
 * 3. 性能优化：使用数组而不是列表提高性能
```

```
 *
 * 相关题目：
 * 1. LeetCode 42. 接雨水 - 双向遍历的经典问题
 * 2. LeetCode 84. 柱状图中最大的矩形 - 单调栈应用
 * 3. LeetCode 406. 根据身高重建队列 - 贪心排序问题
 * 4. 牛客网 NC140 排序 - 各种排序算法实现
 * 5. LintCode 391. 数飞机 - 区间调度相关
 * 6. HackerRank - Jim and the Orders - 贪心调度问题
 * 7. CodeChef - TACHSTCK - 区间配对问题
 * 8. AtCoder ABC104C - All Green - 动态规划相关
 * 9. Codeforces 1363C - Game On Leaves - 博弈论相关
 * 10. POJ 3169 - Layout - 差分约束系统
 */
public class Code12_Candy {

    /**
     * 计算最少需要准备的糖果数量
     *
     * @param ratings 孩子的评分数组
     * @return 最少需要的糖果数量
     */
    public static int candy(int[] ratings) {
        // 边界条件检查
        if (ratings == null || ratings.length == 0) {
            return 0;
        }

        int n = ratings.length;
        if (n == 1) {
            return 1; // 只有一个孩子，最少需要1个糖果
        }

        int[] candies = new int[n];
        Arrays.fill(candies, 1); // 每个孩子至少1个糖果

        // 从左到右遍历，处理递增序列
        for (int i = 1; i < n; i++) {
            if (ratings[i] > ratings[i - 1]) {
                candies[i] = candies[i - 1] + 1;
            }
        }

        // 从右到左遍历，处理递减序列
```

```
        for (int i = n - 2; i >= 0; i--) {
            if (ratings[i] > ratings[i + 1]) {
                candies[i] = Math.max(candies[i], candies[i + 1] + 1);
            }
        }

        // 统计总糖果数
        int totalCandies = 0;
        for (int candy : candies) {
            totalCandies += candy;
        }

        return totalCandies;
    }

    /**
     * 测试函数，验证算法正确性
     */
    public static void main(String[] args) {
        // 测试用例 1: 基本情况 - 递增序列
        int[] ratings1 = {1, 0, 2};
        int result1 = candy(ratings1);
        System.out.println("测试用例 1:");
        System.out.print("评分数组: [");
        for (int i = 0; i < ratings1.length; i++) {
            System.out.print(ratings1[i]);
            if (i < ratings1.length - 1) System.out.print(", ");
        }
        System.out.println("]");
        System.out.println("最少糖果数: " + result1);
        System.out.println("期望输出: 5");
        System.out.println();

        // 测试用例 2: 基本情况 - 递减序列
        int[] ratings2 = {1, 2, 2};
        int result2 = candy(ratings2);
        System.out.println("测试用例 2:");
        System.out.print("评分数组: [");
        for (int i = 0; i < ratings2.length; i++) {
            System.out.print(ratings2[i]);
            if (i < ratings2.length - 1) System.out.print(", ");
        }
        System.out.println("]");
```

```
System.out.println("最少糖果数: " + result2);
System.out.println("期望输出: 4");
System.out.println();

// 测试用例 3: 复杂情况 - 山峰形状
int[] ratings3 = {1, 3, 2, 2, 1};
int result3 = candy(ratings3);
System.out.println("测试用例 3:");
System.out.print("评分数组: [");
for (int i = 0; i < ratings3.length; i++) {
    System.out.print(ratings3[i]);
    if (i < ratings3.length - 1) System.out.print(", ");
}
System.out.println("]");
System.out.println("最少糖果数: " + result3);
System.out.println("期望输出: 7");
System.out.println();

// 测试用例 4: 边界情况 - 单元素数组
int[] ratings4 = {5};
int result4 = candy(ratings4);
System.out.println("测试用例 4:");
System.out.print("评分数组: [");
System.out.print(ratings4[0]);
System.out.println("]");
System.out.println("最少糖果数: " + result4);
System.out.println("期望输出: 1");
System.out.println();

// 测试用例 5: 边界情况 - 两个相同评分
int[] ratings5 = {2, 2};
int result5 = candy(ratings5);
System.out.println("测试用例 5:");
System.out.print("评分数组: [");
for (int i = 0; i < ratings5.length; i++) {
    System.out.print(ratings5[i]);
    if (i < ratings5.length - 1) System.out.print(", ");
}
System.out.println("]");
System.out.println("最少糖果数: " + result5);
System.out.println("期望输出: 2");
System.out.println();
```

```
        // 测试用例 6: 复杂情况 - 长序列
        int[] ratings6 = {1, 2, 87, 87, 87, 2, 1};
        int result6 = candy(ratings6);
        System.out.println("测试用例 6:");
        System.out.print("评分数组: [");
        for (int i = 0; i < ratings6.length; i++) {
            System.out.print(ratings6[i]);
            if (i < ratings6.length - 1) System.out.print(", ");
        }
        System.out.println("]");
        System.out.println("最少糖果数: " + result6);
        System.out.println("期望输出: 13");
    }
}
```

==========

文件: Code12_Candy.py

==========

```
# 分发糖果（Candy）
# 题目来源：LeetCode 135
# 题目链接：https://leetcode.cn/problems/candy/
#
# 问题描述：
# 老师想给孩子们分发糖果，有 N 个孩子站成了一条直线，每个孩子至少分配到 1 个糖果。
# 相邻的孩子中，评分高的孩子必须获得更多的糖果。
# 计算最少需要准备多少糖果。
#
# 算法思路：
# 使用贪心策略，两次遍历：
# 1. 从左到右遍历，如果当前孩子评分比左边高，糖果数比左边多 1
# 2. 从右到左遍历，如果当前孩子评分比右边高，糖果数取当前值和右边值+1 的最大值
# 3. 最后统计所有糖果数之和
#
# 时间复杂度：O(n) - 两次遍历数组
# 空间复杂度：O(n) - 需要额外的糖果数组
#
# 是否最优解：是。这是该问题的最优解法。
#
# 适用场景：
# 1. 分配问题，需要满足相邻约束条件
# 2. 双向约束的最优化问题
#
```

```
# 异常处理：
# 1. 处理空数组情况
# 2. 处理单元素数组
#
# 工程化考量：
# 1. 输入验证：检查数组是否为空
# 2. 边界条件：处理单元素和双元素数组
# 3. 性能优化：使用列表推导式提高效率
#
# 相关题目：
# 1. LeetCode 42. 接雨水 - 双向遍历的经典问题
# 2. LeetCode 84. 柱状图中最大的矩形 - 单调栈应用
# 3. LeetCode 406. 根据身高重建队列 - 贪心排序问题
# 4. 牛客网 NC140 排序 - 各种排序算法实现
# 5. LintCode 391. 数飞机 - 区间调度相关
# 6. HackerRank - Jim and the Orders - 贪心调度问题
# 7. CodeChef - TACHSTCK - 区间配对问题
# 8. AtCoder ABC104C - All Green - 动态规划相关
# 9. Codeforces 1363C - Game On Leaves - 博弈论相关
# 10. POJ 3169 - Layout - 差分约束系统

class Solution:
    """
    计算最少需要准备的糖果数量

    Args:
        ratings: List[int] - 孩子的评分数组

    Returns:
        int - 最少需要的糖果数量
    """
    def candy(self, ratings):
        # 边界条件检查
        if not ratings:
            return 0

        n = len(ratings)
        if n == 1:
            return 1  # 只有一个孩子，最少需要1个糖果

        # 初始化糖果数组，每个孩子至少1个糖果
        candies = [1] * n
```

```
        # 从左到右遍历，处理递增序列
        for i in range(1, n):
            if ratings[i] > ratings[i - 1]:
                candies[i] = candies[i - 1] + 1

        # 从右到左遍历，处理递减序列
        for i in range(n - 2, -1, -1):
            if ratings[i] > ratings[i + 1]:
                candies[i] = max(candies[i], candies[i + 1] + 1)

        # 统计总糖果数
        return sum(candies)


def main():
    solution = Solution()

    # 测试用例 1: 基本情况 - 递增序列
    ratings1 = [1, 0, 2]
    result1 = solution.candy(ratings1)
    print("测试用例 1:")
    print(f"评分数组: {ratings1}")
    print(f"最少糖果数: {result1}")
    print("期望输出: 5")
    print()

    # 测试用例 2: 基本情况 - 递减序列
    ratings2 = [1, 2, 2]
    result2 = solution.candy(ratings2)
    print("测试用例 2:")
    print(f"评分数组: {ratings2}")
    print(f"最少糖果数: {result2}")
    print("期望输出: 4")
    print()

    # 测试用例 3: 复杂情况 - 山峰形状
    ratings3 = [1, 3, 2, 2, 1]
    result3 = solution.candy(ratings3)
    print("测试用例 3:")
    print(f"评分数组: {ratings3}")
    print(f"最少糖果数: {result3}")
    print("期望输出: 7")
    print()
```

```
    # 测试用例 4: 边界情况 - 单元素数组
    ratings4 = [5]
    result4 = solution.candy(ratings4)
    print("测试用例 4:")
    print(f"评分数组: {ratings4}")
    print(f"最少糖果数: {result4}")
    print("期望输出: 1")
    print()

    # 测试用例 5: 边界情况 - 两个相同评分
    ratings5 = [2, 2]
    result5 = solution.candy(ratings5)
    print("测试用例 5:")
    print(f"评分数组: {ratings5}")
    print(f"最少糖果数: {result5}")
    print("期望输出: 2")
    print()

    # 测试用例 6: 复杂情况 - 长序列
    ratings6 = [1, 2, 87, 87, 87, 2, 1]
    result6 = solution.candy(ratings6)
    print("测试用例 6:")
    print(f"评分数组: {ratings6}")
    print(f"最少糖果数: {result6}")
    print("期望输出: 13")


if __name__ == "__main__":
    main()

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

文件: Code13_NonOverlappingIntervals.cpp
==========================================
// 无重叠区间（Non-overlapping Intervals）
// 题目来源：LeetCode 435
// 题目链接：https://leetcode.cn/problems/non-overlapping-intervals/
//
// 问题描述：
// 给定一个区间的集合，找到需要移除区间的最小数量，使剩余区间互不重叠。
//
// 算法思路：
```

```
// 使用贪心策略，按照区间结束时间排序：
// 1. 将区间按照结束时间从小到大排序
// 2. 遍历排序后的区间，记录当前选择的最后一个区间的结束时间
// 3. 如果当前区间的开始时间大于等于前一个区间的结束时间，说明不重叠，选择该区间
// 4. 否则，需要移除该区间，计数加 1
//
// 时间复杂度：O(n log n) - 排序的时间复杂度
// 空间复杂度：O(1) - 只使用了常数额外空间
//
// 是否最优解：是。这是该问题的最优解法。
//
// 适用场景：
// 1. 区间调度问题
// 2. 最大不重叠区间选择
//
// 异常处理：
// 1. 处理空数组情况
// 2. 处理单元素数组
//
// 工程化考量：
// 1. 输入验证：检查数组是否为空
// 2. 边界条件：处理单元素和双元素数组
// 3. 性能优化：使用快速排序提高效率
//
// 相关题目：
// 1. LeetCode 452. 用最少数量的箭引爆气球 - 类似区间问题
// 2. LeetCode 56. 合并区间 - 区间合并问题
// 3. LeetCode 252. 会议室 - 区间重叠判断
// 4. 牛客网 NC135 买票需要多少时间 - 队列模拟相关
// 5. LintCode 391. 数飞机 - 区间调度相关
// 6. HackerRank - Jim and the Orders - 贪心调度问题
// 7. CodeChef - TACHSTCK - 区间配对问题
// 8. AtCoder ABC104C - All Green - 动态规划相关
// 9. Codeforces 1363C - Game On Leaves - 博弈论相关
// 10. POJ 3169 - Layout - 差分约束系统

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

/**
 * 计算需要移除的最小区间数量
```

```
 *
 * @param intervals 区间数组，每个区间包含开始和结束时间
 * @return 需要移除的最小区间数量
 */
int eraseOverlapIntervals(vector<vector<int>>& intervals) {
    // 边界条件检查
    if (intervals.empty()) {
        return 0;
    }

    int n = intervals.size();
    if (n == 1) {
        return 0; // 只有一个区间，不需要移除
    }

    // 按照区间结束时间排序
    sort(intervals.begin(), intervals.end(), [](const vector<int>& a, const vector<int>& b) {
        return a[1] < b[1];
    });

    int count = 0; // 需要移除的区间数量
    int end = intervals[0][1]; // 当前选择的最后一个区间的结束时间

    for (int i = 1; i < n; i++) {
        // 如果当前区间的开始时间小于前一个区间的结束时间，说明重叠
        if (intervals[i][0] < end) {
            count++; // 需要移除当前区间
        } else {
            // 不重叠，更新结束时间
            end = intervals[i][1];
        }
    }

    return count;
}

/**
 * 测试函数，验证算法正确性
 */
int main() {
    // 测试用例 1: 基本情况 - 有重叠区间
    vector<vector<int>> intervals1 = {{1, 2}, {2, 3}, {3, 4}, {1, 3}};
    int result1 = eraseOverlapIntervals(intervals1);
```

```
cout << "测试用例 1:" << endl;
cout << "区间数组: [";
for (int i = 0; i < intervals1.size(); i++) {
    cout << "[" << intervals1[i][0] << "," << intervals1[i][1] << "]";
    if (i < intervals1.size() - 1) cout << ", ";
}
cout << "]" << endl;
cout << "需要移除的区间数量: " << result1 << endl;
cout << "期望输出: 1" << endl << endl;

// 测试用例 2: 基本情况 - 无重叠区间
vector<vector<int>> intervals2 = {{1, 2}, {2, 3}};
int result2 = eraseOverlapIntervals(intervals2);
cout << "测试用例 2:" << endl;
cout << "区间数组: [";
for (int i = 0; i < intervals2.size(); i++) {
    cout << "[" << intervals2[i][0] << "," << intervals2[i][1] << "]";
    if (i < intervals2.size() - 1) cout << ", ";
}
cout << "]" << endl;
cout << "需要移除的区间数量: " << result2 << endl;
cout << "期望输出: 0" << endl << endl;

// 测试用例 3: 复杂情况 - 多个重叠
vector<vector<int>> intervals3 = {{1, 100}, {11, 22}, {1, 11}, {2, 12}};
int result3 = eraseOverlapIntervals(intervals3);
cout << "测试用例 3:" << endl;
cout << "区间数组: [";
for (int i = 0; i < intervals3.size(); i++) {
    cout << "[" << intervals3[i][0] << "," << intervals3[i][1] << "]";
    if (i < intervals3.size() - 1) cout << ", ";
}
cout << "]" << endl;
cout << "需要移除的区间数量: " << result3 << endl;
cout << "期望输出: 3" << endl << endl;

// 测试用例 4: 边界情况 - 单元素数组
vector<vector<int>> intervals4 = {{1, 2}};
int result4 = eraseOverlapIntervals(intervals4);
cout << "测试用例 4:" << endl;
cout << "区间数组: [";
cout << "[" << intervals4[0][0] << "," << intervals4[0][1] << "]";
cout << "]" << endl;
```

```
    cout << "需要移除的区间数量: " << result4 << endl;
    cout << "期望输出: 0" << endl << endl;

    // 测试用例 5: 边界情况 - 空数组
    vector<vector<int>> intervals5 = {};
    int result5 = eraseOverlapIntervals(intervals5);
    cout << "测试用例 5:" << endl;
    cout << "区间数组: []" << endl;
    cout << "需要移除的区间数量: " << result5 << endl;
    cout << "期望输出: 0" << endl << endl;

    // 测试用例 6: 复杂情况 - 完全重叠
    vector<vector<int>> intervals6 = {{1, 10}, {2, 9}, {3, 8}, {4, 7}};
    int result6 = eraseOverlapIntervals(intervals6);
    cout << "测试用例 6:" << endl;
    cout << "区间数组: [";
    for (int i = 0; i < intervals6.size(); i++) {
        cout << "[" << intervals6[i][0] << "," << intervals6[i][1] << "]";
        if (i < intervals6.size() - 1) cout << ", ";
    }
    cout << "]" << endl;
    cout << "需要移除的区间数量: " << result6 << endl;
    cout << "期望输出: 3" << endl;

    return 0;
}

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

文件: Code13_NonOverlappingIntervals.java
==========================================
package class093;

import java.util.Arrays;
import java.util.Comparator;

/**
 * 无重叠区间（Non-overlapping Intervals）
 * 题目来源：LeetCode 435
 * 题目链接：https://leetcode.cn/problems/non-overlapping-intervals/
 *
 * 问题描述：
 * 给定一个区间的集合，找到需要移除区间的最小数量，使剩余区间互不重叠。
```

```
 *
 * 算法思路：
 * 使用贪心策略，按照区间结束时间排序：
 * 1. 将区间按照结束时间从小到大排序
 * 2. 遍历排序后的区间，记录当前选择的最后一个区间的结束时间
 * 3. 如果当前区间的开始时间大于等于前一个区间的结束时间，说明不重叠，选择该区间
 * 4. 否则，需要移除该区间，计数加 1
 *
 * 时间复杂度：O(n log n) - 排序的时间复杂度
 * 空间复杂度：O(1) - 只使用了常数额外空间
 *
 * 是否最优解：是。这是该问题的最优解法。
 *
 * 适用场景：
 * 1. 区间调度问题
 * 2. 最大不重叠区间选择
 *
 * 异常处理：
 * 1. 处理空数组情况
 * 2. 处理单元素数组
 *
 * 工程化考量：
 * 1. 输入验证：检查数组是否为空
 * 2. 边界条件：处理单元素和双元素数组
 * 3. 性能优化：使用快速排序提高效率
 *
 * 相关题目：
 * 1. LeetCode 452. 用最少数量的箭引爆气球 - 类似区间问题
 * 2. LeetCode 56. 合并区间 - 区间合并问题
 * 3. LeetCode 252. 会议室 - 区间重叠判断
 * 4. 牛客网 NC135 买票需要多少时间 - 队列模拟相关
 * 5. LintCode 391. 数飞机 - 区间调度相关
 * 6. HackerRank - Jim and the Orders - 贪心调度问题
 * 7. CodeChef - TACHSTCK - 区间配对问题
 * 8. AtCoder ABC104C - All Green - 动态规划相关
 * 9. Codeforces 1363C - Game On Leaves - 博弈论相关
 * 10. POJ 3169 - Layout - 差分约束系统
 */
public class Code13_NonOverlappingIntervals {

    /**
     * 计算需要移除的最小区间数量
     *
```

```
     * @param intervals 区间数组，每个区间包含开始和结束时间
     * @return 需要移除的最小区间数量
     */
    public static int eraseOverlapIntervals(int[][] intervals) {
        // 边界条件检查
        if (intervals == null || intervals.length == 0) {
            return 0;
        }

        int n = intervals.length;
        if (n == 1) {
            return 0; // 只有一个区间，不需要移除
        }

        // 按照区间结束时间排序
        Arrays.sort(intervals, new Comparator<int[]>() {
            @Override
            public int compare(int[] a, int[] b) {
                return Integer.compare(a[1], b[1]);
            }
        });

        int count = 0; // 需要移除的区间数量
        int end = intervals[0][1]; // 当前选择的最后一个区间的结束时间

        for (int i = 1; i < n; i++) {
            // 如果当前区间的开始时间小于前一个区间的结束时间，说明重叠
            if (intervals[i][0] < end) {
                count++; // 需要移除当前区间
            } else {
                // 不重叠，更新结束时间
                end = intervals[i][1];
            }
        }

        return count;
    }

    /**
     * 测试函数，验证算法正确性
     */
    public static void main(String[] args) {
        // 测试用例 1: 基本情况 - 有重叠区间
```

```
int[][] intervals1 = {{1, 2}, {2, 3}, {3, 4}, {1, 3}};
int result1 = eraseOverlapIntervals(intervals1);
System.out.println("测试用例 1:");
System.out.print("区间数组: [");
for (int i = 0; i < intervals1.length; i++) {
    System.out.print("[" + intervals1[i][0] + "," + intervals1[i][1] + "]");
    if (i < intervals1.length - 1) System.out.print(", ");
}
System.out.println("]");
System.out.println("需要移除的区间数量: " + result1);
System.out.println("期望输出: 1");
System.out.println();

// 测试用例 2: 基本情况 - 无重叠区间
int[][] intervals2 = {{1, 2}, {2, 3}};
int result2 = eraseOverlapIntervals(intervals2);
System.out.println("测试用例 2:");
System.out.print("区间数组: [");
for (int i = 0; i < intervals2.length; i++) {
    System.out.print("[" + intervals2[i][0] + "," + intervals2[i][1] + "]");
    if (i < intervals2.length - 1) System.out.print(", ");
}
System.out.println("]");
System.out.println("需要移除的区间数量: " + result2);
System.out.println("期望输出: 0");
System.out.println();

// 测试用例 3: 复杂情况 - 多个重叠
int[][] intervals3 = {{1, 100}, {11, 22}, {1, 11}, {2, 12}};
int result3 = eraseOverlapIntervals(intervals3);
System.out.println("测试用例 3:");
System.out.print("区间数组: [");
for (int i = 0; i < intervals3.length; i++) {
    System.out.print("[" + intervals3[i][0] + "," + intervals3[i][1] + "]");
    if (i < intervals3.length - 1) System.out.print(", ");
}
System.out.println("]");
System.out.println("需要移除的区间数量: " + result3);
System.out.println("期望输出: 3");
System.out.println();

// 测试用例 4: 边界情况 - 单元素数组
int[][] intervals4 = {{1, 2}};
```

```
        int result4 = eraseOverlapIntervals(intervals4);
        System.out.println("测试用例 4:");
        System.out.print("区间数组: [");
        System.out.print("[" + intervals4[0][0] + "," + intervals4[0][1] + "]");
        System.out.println("]");
        System.out.println("需要移除的区间数量: " + result4);
        System.out.println("期望输出: 0");
        System.out.println();

        // 测试用例 5: 边界情况 - 空数组
        int[][] intervals5 = {};
        int result5 = eraseOverlapIntervals(intervals5);
        System.out.println("测试用例 5:");
        System.out.println("区间数组: []");
        System.out.println("需要移除的区间数量: " + result5);
        System.out.println("期望输出: 0");
        System.out.println();

        // 测试用例 6: 复杂情况 - 完全重叠
        int[][] intervals6 = {{1, 10}, {2, 9}, {3, 8}, {4, 7}};
        int result6 = eraseOverlapIntervals(intervals6);
        System.out.println("测试用例 6:");
        System.out.print("区间数组: [");
        for (int i = 0; i < intervals6.length; i++) {
            System.out.print("[" + intervals6[i][0] + "," + intervals6[i][1] + "]");
            if (i < intervals6.length - 1) System.out.print(", ");
        }
        System.out.println("]");
        System.out.println("需要移除的区间数量: " + result6);
        System.out.println("期望输出: 3");
    }
}
```

==========

文件: Code13_NonOverlappingIntervals.py

==========

```
# 无重叠区间（Non-overlapping Intervals）
# 题目来源：LeetCode 435
# 题目链接：https://leetcode.cn/problems/non-overlapping-intervals/
#
# 问题描述：
# 给定一个区间的集合，找到需要移除区间的最小数量，使剩余区间互不重叠。
```

```
#
# 算法思路：
# 使用贪心策略，按照区间结束时间排序：
# 1. 将区间按照结束时间从小到大排序
# 2. 遍历排序后的区间，记录当前选择的最后一个区间的结束时间
# 3. 如果当前区间的开始时间大于等于前一个区间的结束时间，说明不重叠，选择该区间
# 4. 否则，需要移除该区间，计数加 1
#
# 时间复杂度：O(n log n) - 排序的时间复杂度
# 空间复杂度：O(1) - 只使用了常数额外空间
#
# 是否最优解：是。这是该问题的最优解法。
#
# 适用场景：
# 1. 区间调度问题
# 2. 最大不重叠区间选择
#
# 异常处理：
# 1. 处理空数组情况
# 2. 处理单元素数组
#
# 工程化考量：
# 1. 输入验证：检查数组是否为空
# 2. 边界条件：处理单元素和双元素数组
# 3. 性能优化：使用内置排序提高效率
#
# 相关题目：
# 1. LeetCode 452. 用最少数量的箭引爆气球 - 类似区间问题
# 2. LeetCode 56. 合并区间 - 区间合并问题
# 3. LeetCode 252. 会议室 - 区间重叠判断
# 4. 牛客网 NC135 买票需要多少时间 - 队列模拟相关
# 5. LintCode 391. 数飞机 - 区间调度相关
# 6. HackerRank - Jim and the Orders - 贪心调度问题
# 7. CodeChef - TACHSTCK - 区间配对问题
# 8. AtCoder ABC104C - All Green - 动态规划相关
# 9. Codeforces 1363C - Game On Leaves - 博弈论相关
# 10. POJ 3169 - Layout - 差分约束系统

class Solution:
    """
    计算需要移除的最小区间数量

    Args:
```

```
        intervals: List[List[int]] - 区间数组，每个区间包含开始和结束时间

    Returns:
        int - 需要移除的最小区间数量
    """
    def eraseOverlapIntervals(self, intervals):
        # 边界条件检查
        if not intervals:
            return 0

        n = len(intervals)
        if n == 1:
            return 0  # 只有一个区间，不需要移除

        # 按照区间结束时间排序
        intervals.sort(key=lambda x: x[1])

        count = 0  # 需要移除的区间数量
        end = intervals[0][1]  # 当前选择的最后一个区间的结束时间

        for i in range(1, n):
            # 如果当前区间的开始时间小于前一个区间的结束时间，说明重叠
            if intervals[i][0] < end:
                count += 1  # 需要移除当前区间
            else:
                # 不重叠，更新结束时间
                end = intervals[i][1]

        return count


def main():
    solution = Solution()

    # 测试用例 1: 基本情况 - 有重叠区间
    intervals1 = [[1, 2], [2, 3], [3, 4], [1, 3]]
    result1 = solution.eraseOverlapIntervals(intervals1)
    print("测试用例 1:")
    print(f"区间数组: {intervals1}")
    print(f"需要移除的区间数量: {result1}")
    print("期望输出: 1")
    print()
```

```
# 测试用例 2: 基本情况 - 无重叠区间
intervals2 = [[1, 2], [2, 3]]
result2 = solution.eraseOverlapIntervals(intervals2)
print("测试用例 2:")
print(f"区间数组: {intervals2}")
print(f"需要移除的区间数量: {result2}")
print("期望输出: 0")
print()

# 测试用例 3: 复杂情况 - 多个重叠
intervals3 = [[1, 100], [11, 22], [1, 11], [2, 12]]
result3 = solution.eraseOverlapIntervals(intervals3)
print("测试用例 3:")
print(f"区间数组: {intervals3}")
print(f"需要移除的区间数量: {result3}")
print("期望输出: 3")
print()

# 测试用例 4: 边界情况 - 单元素数组
intervals4 = [[1, 2]]
result4 = solution.eraseOverlapIntervals(intervals4)
print("测试用例 4:")
print(f"区间数组: {intervals4}")
print(f"需要移除的区间数量: {result4}")
print("期望输出: 0")
print()

# 测试用例 5: 边界情况 - 空数组
intervals5 = []
result5 = solution.eraseOverlapIntervals(intervals5)
print("测试用例 5:")
print(f"区间数组: {intervals5}")
print(f"需要移除的区间数量: {result5}")
print("期望输出: 0")
print()

# 测试用例 6: 复杂情况 - 完全重叠
intervals6 = [[1, 10], [2, 9], [3, 8], [4, 7]]
result6 = solution.eraseOverlapIntervals(intervals6)
print("测试用例 6:")
print(f"区间数组: {intervals6}")
print(f"需要移除的区间数量: {result6}")
print("期望输出: 3")
```

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

==========

文件: Code14_MinimumArrowsToBurstBalloons.cpp

==========

```
// 用最少数量的箭引爆气球（Minimum Number of Arrows to Burst Balloons）
// 题目来源：LeetCode 452
// 题目链接：https://leetcode.cn/problems/minimum-number-of-arrows-to-burst-balloons/
//
// 问题描述：
// 在二维空间中有许多球形的气球，每个气球在水平方向上的直径范围是[xstart, xend]。
// 用最少数量的箭引爆所有气球。一支箭可以垂直向上射出，在 xstart 和 xend 之间穿过气球。
// 只要箭的 x 坐标在气球的直径范围内，气球就会被引爆。
//
// 算法思路：
// 使用贪心策略，按照气球结束坐标排序：
// 1. 将气球按照结束坐标从小到大排序
// 2. 遍历排序后的气球，记录当前箭的位置
// 3. 如果当前气球的开始坐标大于箭的位置，说明需要新的箭
// 4. 否则，当前箭可以引爆这个气球
//
// 时间复杂度：O(n log n) - 排序的时间复杂度
// 空间复杂度：O(1) - 只使用了常数额外空间
//
// 是否最优解：是。这是该问题的最优解法。
//
// 适用场景：
// 1. 区间覆盖问题
// 2. 最小点覆盖区间问题
//
// 异常处理：
// 1. 处理空数组情况
// 2. 处理单元素数组
//
// 工程化考量：
// 1. 输入验证：检查数组是否为空
// 2. 边界条件：处理单元素和双元素数组
// 3. 性能优化：使用快速排序提高效率
//
```

```
// 相关题目：
// 1. LeetCode 435. 无重叠区间 - 类似区间问题
// 2. LeetCode 56. 合并区间 - 区间合并问题
// 3. LeetCode 252. 会议室 - 区间重叠判断
// 4. 牛客网 NC135 买票需要多少时间 - 队列模拟相关
// 5. LintCode 391. 数飞机 - 区间调度相关
// 6. HackerRank - Jim and the Orders - 贪心调度问题
// 7. CodeChef - TACHSTCK - 区间配对问题
// 8. AtCoder ABC104C - All Green - 动态规划相关
// 9. Codeforces 1363C - Game On Leaves - 博弈论相关
// 10. POJ 3169 - Layout - 差分约束系统

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

/**
 * 计算引爆所有气球所需的最少箭数
 *
 * @param points 气球直径范围数组，每个元素是[xstart, xend]
 * @return 最少需要的箭数
 */
int findMinArrowShots(vector<vector<int>>& points) {
    // 边界条件检查
    if (points.empty()) {
        return 0;
    }

    int n = points.size();
    if (n == 1) {
        return 1; // 只有一个气球，需要一支箭
    }

    // 按照气球结束坐标排序
    sort(points.begin(), points.end(), [](const vector<int>& a, const vector<int>& b) {
        return a[1] < b[1];
    });

    int arrows = 1; // 至少需要一支箭
    int arrowPos = points[0][1]; // 第一支箭的位置
```

```
    for (int i = 1; i < n; i++) {
        // 如果当前气球的开始坐标大于箭的位置，需要新的箭
        if (points[i][0] > arrowPos) {
            arrows++;
            arrowPos = points[i][1]; // 更新箭的位置
        }
        // 否则，当前箭可以引爆这个气球，继续使用同一支箭
    }

    return arrows;
}

/**
 * 测试函数，验证算法正确性
 */
int main() {
    // 测试用例 1: 基本情况 - 有重叠气球
    vector<vector<int>> points1 = {{10, 16}, {2, 8}, {1, 6}, {7, 12}};
    int result1 = findMinArrowShots(points1);
    cout << "测试用例 1:" << endl;
    cout << "气球范围: [";
    for (int i = 0; i < points1.size(); i++) {
        cout << "[" << points1[i][0] << "," << points1[i][1] << "]";
        if (i < points1.size() - 1) cout << ", ";
    }
    cout << "]" << endl;
    cout << "最少箭数: " << result1 << endl;
    cout << "期望输出: 2" << endl << endl;

    // 测试用例 2: 基本情况 - 无重叠气球
    vector<vector<int>> points2 = {{1, 2}, {3, 4}, {5, 6}, {7, 8}};
    int result2 = findMinArrowShots(points2);
    cout << "测试用例 2:" << endl;
    cout << "气球范围: [";
    for (int i = 0; i < points2.size(); i++) {
        cout << "[" << points2[i][0] << "," << points2[i][1] << "]";
        if (i < points2.size() - 1) cout << ", ";
    }
    cout << "]" << endl;
    cout << "最少箭数: " << result2 << endl;
    cout << "期望输出: 4" << endl << endl;

    // 测试用例 3: 复杂情况 - 完全重叠
```

```
vector<vector<int>> points3 = {{1, 2}, {2, 3}, {3, 4}, {4, 5}};
int result3 = findMinArrowShots(points3);
cout << "测试用例 3:" << endl;
cout << "气球范围: [";
for (int i = 0; i < points3.size(); i++) {
    cout << "[" << points3[i][0] << "," << points3[i][1] << "]";
    if (i < points3.size() - 1) cout << ", ";
}
cout << "]" << endl;
cout << "最少箭数: " << result3 << endl;
cout << "期望输出: 2" << endl << endl;

// 测试用例 4: 边界情况 - 单元素数组
vector<vector<int>> points4 = {{1, 2}};
int result4 = findMinArrowShots(points4);
cout << "测试用例 4:" << endl;
cout << "气球范围: [";
cout << "[" << points4[0][0] << "," << points4[0][1] << "]";
cout << "]" << endl;
cout << "最少箭数: " << result4 << endl;
cout << "期望输出: 1" << endl << endl;

// 测试用例 5: 边界情况 - 空数组
vector<vector<int>> points5 = {};
int result5 = findMinArrowShots(points5);
cout << "测试用例 5:" << endl;
cout << "气球范围: []" << endl;
cout << "最少箭数: " << result5 << endl;
cout << "期望输出: 0" << endl << endl;

// 测试用例 6: 复杂情况 - 大数测试
vector<vector<int>> points6 = {{INT_MIN, INT_MIN + 1}, {INT_MAX - 1, INT_MAX}};
int result6 = findMinArrowShots(points6);
cout << "测试用例 6:" << endl;
cout << "气球范围: [";
for (int i = 0; i < points6.size(); i++) {
    cout << "[" << points6[i][0] << "," << points6[i][1] << "]";
    if (i < points6.size() - 1) cout << ", ";
}
cout << "]" << endl;
cout << "最少箭数: " << result6 << endl;
cout << "期望输出: 2" << endl;
```

```
    return 0;
}

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

文件: Code14_MinimumArrowsToBurstBalloons.java
==========================================
package class093;

import java.util.Arrays;
import java.util.Comparator;

/**
 * 用最少数量的箭引爆气球（Minimum Number of Arrows to Burst Balloons）
 * 题目来源：LeetCode 452
 * 题目链接：https://leetcode.cn/problems/minimum-number-of-arrows-to-burst-balloons/
 *
 * 问题描述：
 * 在二维空间中有许多球形的气球，每个气球在水平方向上的直径范围是[xstart, xend]。
 * 用最少数量的箭引爆所有气球。一支箭可以垂直向上射出，在 xstart 和 xend 之间穿过气球。
 * 只要箭的 x 坐标在气球的直径范围内，气球就会被引爆。
 *
 * 算法思路：
 * 使用贪心策略，按照气球结束坐标排序：
 * 1. 将气球按照结束坐标从小到大排序
 * 2. 遍历排序后的气球，记录当前箭的位置
 * 3. 如果当前气球的开始坐标大于箭的位置，说明需要新的箭
 * 4. 否则，当前箭可以引爆这个气球
 *
 * 时间复杂度：O(n log n) - 排序的时间复杂度
 * 空间复杂度：O(1) - 只使用了常数额外空间
 *
 * 是否最优解：是。这是该问题的最优解法。
 *
 * 适用场景：
 * 1. 区间覆盖问题
 * 2. 最小点覆盖区间问题
 *
 * 异常处理：
 * 1. 处理空数组情况
 * 2. 处理单元素数组
 *
 * 工程化考量：
```

```
 * 1. 输入验证：检查数组是否为空
 * 2. 边界条件：处理单元素和双元素数组
 * 3. 性能优化：使用快速排序提高效率
 *
 * 相关题目：
 * 1. LeetCode 435. 无重叠区间 - 类似区间问题
 * 2. LeetCode 56. 合并区间 - 区间合并问题
 * 3. LeetCode 252. 会议室 - 区间重叠判断
 * 4. 牛客网 NC135 买票需要多少时间 - 队列模拟相关
 * 5. LintCode 391. 数飞机 - 区间调度相关
 * 6. HackerRank - Jim and the Orders - 贪心调度问题
 * 7. CodeChef - TACHSTCK - 区间配对问题
 * 8. AtCoder ABC104C - All Green - 动态规划相关
 * 9. Codeforces 1363C - Game On Leaves - 博弈论相关
 * 10. POJ 3169 - Layout - 差分约束系统
 */
public class Code14_MinimumArrowsToBurstBalloons {

    /**
     * 计算引爆所有气球所需的最少箭数
     *
     * @param points 气球直径范围数组，每个元素是[xstart, xend]
     * @return 最少需要的箭数
     */
    public static int findMinArrowShots(int[][] points) {
        // 边界条件检查
        if (points == null || points.length == 0) {
            return 0;
        }

        int n = points.length;
        if (n == 1) {
            return 1; // 只有一个气球，需要一支箭
        }

        // 按照气球结束坐标排序（注意使用 Integer.compare 避免溢出）
        Arrays.sort(points, new Comparator<int[]>() {
            @Override
            public int compare(int[] a, int[] b) {
                return Integer.compare(a[1], b[1]);
            }
        });
```

```
    int arrows = 1; // 至少需要一支箭
    int arrowPos = points[0][1]; // 第一支箭的位置

    for (int i = 1; i < n; i++) {
        // 如果当前气球的开始坐标大于箭的位置，需要新的箭
        if (points[i][0] > arrowPos) {
            arrows++;
            arrowPos = points[i][1]; // 更新箭的位置
        }
        // 否则，当前箭可以引爆这个气球，继续使用同一支箭
    }

    return arrows;
}

/**
 * 测试函数，验证算法正确性
 */
public static void main(String[] args) {
    // 测试用例 1: 基本情况 - 有重叠气球
    int[][] points1 = {{10, 16}, {2, 8}, {1, 6}, {7, 12}};
    int result1 = findMinArrowShots(points1);
    System.out.println("测试用例 1:");
    System.out.print("气球范围: [");
    for (int i = 0; i < points1.length; i++) {
        System.out.print("[" + points1[i][0] + "," + points1[i][1] + "]");
        if (i < points1.length - 1) System.out.print(", ");
    }
    System.out.println("]");
    System.out.println("最少箭数: " + result1);
    System.out.println("期望输出: 2");
    System.out.println();

    // 测试用例 2: 基本情况 - 无重叠气球
    int[][] points2 = {{1, 2}, {3, 4}, {5, 6}, {7, 8}};
    int result2 = findMinArrowShots(points2);
    System.out.println("测试用例 2:");
    System.out.print("气球范围: [");
    for (int i = 0; i < points2.length; i++) {
        System.out.print("[" + points2[i][0] + "," + points2[i][1] + "]");
        if (i < points2.length - 1) System.out.print(", ");
    }
    System.out.println("]");
```

```
System.out.println("最少箭数: " + result2);
System.out.println("期望输出: 4");
System.out.println();

// 测试用例 3: 复杂情况 - 完全重叠
int[][] points3 = {{1, 2}, {2, 3}, {3, 4}, {4, 5}};
int result3 = findMinArrowShots(points3);
System.out.println("测试用例 3:");
System.out.print("气球范围: [");
for (int i = 0; i < points3.length; i++) {
    System.out.print("[" + points3[i][0] + "," + points3[i][1] + "]");
    if (i < points3.length - 1) System.out.print(", ");
}
System.out.println("]");
System.out.println("最少箭数: " + result3);
System.out.println("期望输出: 2");
System.out.println();

// 测试用例 4: 边界情况 - 单元素数组
int[][] points4 = {{1, 2}};
int result4 = findMinArrowShots(points4);
System.out.println("测试用例 4:");
System.out.print("气球范围: [");
System.out.print("[" + points4[0][0] + "," + points4[0][1] + "]");
System.out.println("]");
System.out.println("最少箭数: " + result4);
System.out.println("期望输出: 1");
System.out.println();

// 测试用例 5: 边界情况 - 空数组
int[][] points5 = {};
int result5 = findMinArrowShots(points5);
System.out.println("测试用例 5:");
System.out.println("气球范围: []");
System.out.println("最少箭数: " + result5);
System.out.println("期望输出: 0");
System.out.println();

// 测试用例 6: 复杂情况 - 大数测试
int[][] points6 = {{-2147483646, -2147483645}, {2147483646, 2147483647}};
int result6 = findMinArrowShots(points6);
System.out.println("测试用例 6:");
System.out.print("气球范围: [");
```

```
        for (int i = 0; i < points6.length; i++) {
            System.out.print("[" + points6[i][0] + "," + points6[i][1] + "]");
            if (i < points6.length - 1) System.out.print(", ");
        }
        System.out.println("]");
        System.out.println("最少箭数: " + result6);
        System.out.println("期望输出: 2");
    }
}

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

文件: Code14_MinimumArrowsToBurstBalloons.py
==========================================
# 用最少数量的箭引爆气球（Minimum Number of Arrows to Burst Balloons）
# 题目来源：LeetCode 452
# 题目链接：https://leetcode.cn/problems/minimum-number-of-arrows-to-burst-balloons/
#
# 问题描述：
# 在二维空间中有许多球形的气球，每个气球在水平方向上的直径范围是[xstart, xend]。
# 用最少数量的箭引爆所有气球。一支箭可以垂直向上射出，在 xstart 和 xend 之间穿过气球。
# 只要箭的 x 坐标在气球的直径范围内，气球就会被引爆。
#
# 算法思路：
# 使用贪心策略，按照气球结束坐标排序：
# 1. 将气球按照结束坐标从小到大排序
# 2. 遍历排序后的气球，记录当前箭的位置
# 3. 如果当前气球的开始坐标大于箭的位置，说明需要新的箭
# 4. 否则，当前箭可以引爆这个气球
#
# 时间复杂度：O(n log n) - 排序的时间复杂度
# 空间复杂度：O(1) - 只使用了常数额外空间
#
# 是否最优解：是。这是该问题的最优解法。
#
# 适用场景：
# 1. 区间覆盖问题
# 2. 最小点覆盖区间问题
#
# 异常处理：
# 1. 处理空数组情况
# 2. 处理单元素数组
#
```

```
# 工程化考量：
# 1. 输入验证：检查数组是否为空
# 2. 边界条件：处理单元素和双元素数组
# 3. 性能优化：使用内置排序提高效率
#
# 相关题目：
# 1. LeetCode 435. 无重叠区间 - 类似区间问题
# 2. LeetCode 56. 合并区间 - 区间合并问题
# 3. LeetCode 252. 会议室 - 区间重叠判断
# 4. 牛客网 NC135 买票需要多少时间 - 队列模拟相关
# 5. LintCode 391. 数飞机 - 区间调度相关
# 6. HackerRank - Jim and the Orders - 贪心调度问题
# 7. CodeChef - TACHSTCK - 区间配对问题
# 8. AtCoder ABC104C - All Green - 动态规划相关
# 9. Codeforces 1363C - Game On Leaves - 博弈论相关
# 10. POJ 3169 - Layout - 差分约束系统

class Solution:
    """
    计算引爆所有气球所需的最少箭数

    Args:
        points: List[List[int]] - 气球直径范围数组，每个元素是[xstart, xend]

    Returns:
        int - 最少需要的箭数
    """
    def findMinArrowShots(self, points):
        # 边界条件检查
        if not points:
            return 0

        n = len(points)
        if n == 1:
            return 1  # 只有一个气球，需要一支箭

        # 按照气球结束坐标排序
        points.sort(key=lambda x: x[1])

        arrows = 1  # 至少需要一支箭
        arrow_pos = points[0][1]  # 第一支箭的位置

        for i in range(1, n):
```

```
            # 如果当前气球的开始坐标大于箭的位置，需要新的箭
            if points[i][0] > arrow_pos:
                arrows += 1
                arrow_pos = points[i][1]  # 更新箭的位置
            # 否则，当前箭可以引爆这个气球，继续使用同一支箭

        return arrows


def main():
    solution = Solution()

    # 测试用例 1: 基本情况 - 有重叠气球
    points1 = [[10, 16], [2, 8], [1, 6], [7, 12]]
    result1 = solution.findMinArrowShots(points1)
    print("测试用例 1:")
    print(f"气球范围: {points1}")
    print(f"最少箭数: {result1}")
    print("期望输出: 2")
    print()

    # 测试用例 2: 基本情况 - 无重叠气球
    points2 = [[1, 2], [3, 4], [5, 6], [7, 8]]
    result2 = solution.findMinArrowShots(points2)
    print("测试用例 2:")
    print(f"气球范围: {points2}")
    print(f"最少箭数: {result2}")
    print("期望输出: 4")
    print()

    # 测试用例 3: 复杂情况 - 完全重叠
    points3 = [[1, 2], [2, 3], [3, 4], [4, 5]]
    result3 = solution.findMinArrowShots(points3)
    print("测试用例 3:")
    print(f"气球范围: {points3}")
    print(f"最少箭数: {result3}")
    print("期望输出: 2")
    print()

    # 测试用例 4: 边界情况 - 单元素数组
    points4 = [[1, 2]]
    result4 = solution.findMinArrowShots(points4)
    print("测试用例 4:")
```

```
    print(f"气球范围: {points4}")
    print(f"最少箭数: {result4}")
    print("期望输出: 1")
    print()

    # 测试用例 5: 边界情况 - 空数组
    points5 = []
    result5 = solution.findMinArrowShots(points5)
    print("测试用例 5:")
    print(f"气球范围: {points5}")
    print(f"最少箭数: {result5}")
    print("期望输出: 0")
    print()

    # 测试用例 6: 复杂情况 - 大数测试
    points6 = [[-2147483646, -2147483645], [2147483646, 2147483647]]
    result6 = solution.findMinArrowShots(points6)
    print("测试用例 6:")
    print(f"气球范围: {points6}")
    print(f"最少箭数: {result6}")
    print("期望输出: 2")


if __name__ == "__main__":
    main()

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

文件: Code15_MaximumSubarray.cpp
================================================
// 最大子数组和（Maximum Subarray）
// 题目来源：LeetCode 53
// 题目链接：https://leetcode.cn/problems/maximum-subarray/
//
// 问题描述：
// 给定一个整数数组 nums，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），返回其最大和。
//
// 算法思路：
// 使用 Kadane 算法（贪心策略）：
// 1. 遍历数组，维护当前子数组和和最大子数组和
// 2. 如果当前子数组和为负数，重置为当前元素值（因为负数会减小后续和）
// 3. 否则，继续累加当前元素
// 4. 每次更新最大子数组和
```

```
//
// 时间复杂度：O(n) - 只需遍历数组一次
// 空间复杂度：O(1) - 只使用了常数额外空间
//
// 是否最优解：是。Kadane 算法是该问题的最优解法。
//
// 适用场景：
// 1. 连续子数组和问题
// 2. 最大收益问题
//
// 异常处理：
// 1. 处理空数组情况
// 2. 处理全负数数组
//
// 工程化考量：
// 1. 输入验证：检查数组是否为空
// 2. 边界条件：处理单元素数组
// 3. 性能优化：一次遍历完成计算
//
// 相关题目：
// 1. LeetCode 152. 乘积最大子数组 - 类似问题，但需要处理负数
// 2. LeetCode 121. 买卖股票的最佳时机 - 最大差值问题
// 3. LeetCode 918. 环形子数组的最大和 - 环形数组版本
// 4. 牛客网 NC140 排序 - 各种排序算法实现
// 5. LintCode 41. 最大子数组 - 与本题相同
// 6. HackerRank - Maximum Subarray Sum - 类似问题
// 7. CodeChef - MAXSUBA - 最大子数组问题
// 8. AtCoder ABC139D - ModSum - 数学相关
// 9. Codeforces 1370C - Number Game - 博弈论相关
// 10. POJ 2479 - Maximum sum - 双最大子数组和

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

// 简单的输出函数，避免 iostream 依赖
void printArray(const vector<int>& nums) {
    cout << "[";
    for (int i = 0; i < nums.size(); i++) {
        cout << nums[i];
        if (i < nums.size() - 1) cout << ", ";
    }
```

```
    cout << "]";
}

/**
 * 计算最大子数组和
 *
 * @param nums 整数数组
 * @return 最大子数组和
 */
int maxSubArray(vector<int>& nums) {
    // 边界条件检查
    if (nums.empty()) {
        return 0;
    }

    int n = nums.size();
    if (n == 1) {
        return nums[0]; // 只有一个元素，直接返回
    }

    int maxSum = nums[0]; // 最大子数组和
    int currentSum = nums[0]; // 当前子数组和

    for (int i = 1; i < n; i++) {
        // 如果当前子数组和为负数，重置为当前元素值
        if (currentSum < 0) {
            currentSum = nums[i];
        } else {
            // 否则，继续累加当前元素
            currentSum += nums[i];
        }

        // 更新最大子数组和
        if (currentSum > maxSum) {
            maxSum = currentSum;
        }
    }

    return maxSum;
}

/**
 * 测试函数，验证算法正确性
```

```
 */
int main() {
    // 测试用例 1: 基本情况 - 正数数组
    vector<int> nums1 = {-2, 1, -3, 4, -1, 2, 1, -5, 4};
    int result1 = maxSubArray(nums1);
    cout << "测试用例 1:" << endl;
    cout << "数组: [";
    for (int i = 0; i < nums1.size(); i++) {
        cout << nums1[i];
        if (i < nums1.size() - 1) cout << ", ";
    }
    cout << "]" << endl;
    cout << "最大子数组和: " << result1 << endl;
    cout << "期望输出: 6" << endl << endl;

    // 测试用例 2: 基本情况 - 全正数数组
    vector<int> nums2 = {1, 2, 3, 4, 5};
    int result2 = maxSubArray(nums2);
    cout << "测试用例 2:" << endl;
    cout << "数组: [";
    for (int i = 0; i < nums2.size(); i++) {
        cout << nums2[i];
        if (i < nums2.size() - 1) cout << ", ";
    }
    cout << "]" << endl;
    cout << "最大子数组和: " << result2 << endl;
    cout << "期望输出: 15" << endl << endl;

    // 测试用例 3: 基本情况 - 全负数数组
    vector<int> nums3 = {-2, -3, -1, -5};
    int result3 = maxSubArray(nums3);
    cout << "测试用例 3:" << endl;
    cout << "数组: [";
    for (int i = 0; i < nums3.size(); i++) {
        cout << nums3[i];
        if (i < nums3.size() - 1) cout << ", ";
    }
    cout << "]" << endl;
    cout << "最大子数组和: " << result3 << endl;
    cout << "期望输出: -1" << endl << endl;

    // 测试用例 4: 边界情况 - 单元素数组
    vector<int> nums4 = {5};
```

```
    int result4 = maxSubArray(nums4);
    cout << "测试用例 4:" << endl;
    cout << "数组: [";
    cout << nums4[0];
    cout << "]" << endl;
    cout << "最大子数组和: " << result4 << endl;
    cout << "期望输出: 5" << endl << endl;

    // 测试用例 5: 边界情况 - 空数组
    vector<int> nums5 = {};
    int result5 = maxSubArray(nums5);
    cout << "测试用例 5:" << endl;
    cout << "数组: []" << endl;
    cout << "最大子数组和: " << result5 << endl;
    cout << "期望输出: 0" << endl << endl;

    // 测试用例 6: 复杂情况 - 混合数组
    vector<int> nums6 = {8, -19, 5, -4, 20};
    int result6 = maxSubArray(nums6);
    cout << "测试用例 6:" << endl;
    cout << "数组: [";
    for (int i = 0; i < nums6.size(); i++) {
        cout << nums6[i];
        if (i < nums6.size() - 1) cout << ", ";
    }
    cout << "]" << endl;
    cout << "最大子数组和: " << result6 << endl;
    cout << "期望输出: 21" << endl;

    return 0;
}

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

文件: Code15_MaximumSubarray.java
==========================================
package class093;

/**
 * 最大子数组和（Maximum Subarray）
 * 题目来源：LeetCode 53
 * 题目链接：https://leetcode.cn/problems/maximum-subarray/
 *
```

```
 * 问题描述：
 * 给定一个整数数组 nums，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），返回其最大和。
 *
 * 算法思路：
 * 使用 Kadane 算法（贪心策略）：
 * 1. 遍历数组，维护当前子数组和和最大子数组和
 * 2. 如果当前子数组和为负数，重置为当前元素值（因为负数会减小后续和）
 * 3. 否则，继续累加当前元素
 * 4. 每次更新最大子数组和
 *
 * 时间复杂度：O(n) - 只需遍历数组一次
 * 空间复杂度：O(1) - 只使用了常数额外空间
 *
 * 是否最优解：是。Kadane 算法是该问题的最优解法。
 *
 * 适用场景：
 * 1. 连续子数组和问题
 * 2. 最大收益问题
 *
 * 异常处理：
 * 1. 处理空数组情况
 * 2. 处理全负数数组
 *
 * 工程化考量：
 * 1. 输入验证：检查数组是否为空
 * 2. 边界条件：处理单元素数组
 * 3. 性能优化：一次遍历完成计算
 *
 * 相关题目：
 * 1. LeetCode 152. 乘积最大子数组 - 类似问题，但需要处理负数
 * 2. LeetCode 121. 买卖股票的最佳时机 - 最大差值问题
 * 3. LeetCode 918. 环形子数组的最大和 - 环形数组版本
 * 4. 牛客网 NC140 排序 - 各种排序算法实现
 * 5. LintCode 41. 最大子数组 - 与本题相同
 * 6. HackerRank - Maximum Subarray Sum - 类似问题
 * 7. CodeChef - MAXSUBA - 最大子数组问题
 * 8. AtCoder ABC139D - ModSum - 数学相关
 * 9. Codeforces 1370C - Number Game - 博弈论相关
 * 10. POJ 2479 - Maximum sum - 双最大子数组和
 */
public class Code15_MaximumSubarray {

    /**
```

```
     * 计算最大子数组和
     *
     * @param nums 整数数组
     * @return 最大子数组和
     */
    public static int maxSubArray(int[] nums) {
        // 边界条件检查
        if (nums == null || nums.length == 0) {
            return 0;
        }

        int n = nums.length;
        if (n == 1) {
            return nums[0]; // 只有一个元素，直接返回
        }

        int maxSum = nums[0]; // 最大子数组和
        int currentSum = nums[0]; // 当前子数组和

        for (int i = 1; i < n; i++) {
            // 如果当前子数组和为负数，重置为当前元素值
            if (currentSum < 0) {
                currentSum = nums[i];
            } else {
                // 否则，继续累加当前元素
                currentSum += nums[i];
            }

            // 更新最大子数组和
            if (currentSum > maxSum) {
                maxSum = currentSum;
            }
        }

        return maxSum;
    }

    /**
     * 测试函数，验证算法正确性
     */
    public static void main(String[] args) {
        // 测试用例 1: 基本情况 - 正数数组
        int[] nums1 = {-2, 1, -3, 4, -1, 2, 1, -5, 4};
```

```
int result1 = maxSubArray(nums1);
System.out.println("测试用例 1:");
System.out.print("数组: [");
for (int i = 0; i < nums1.length; i++) {
    System.out.print(nums1[i]);
    if (i < nums1.length - 1) System.out.print(", ");
}
System.out.println("]");
System.out.println("最大子数组和: " + result1);
System.out.println("期望输出: 6");
System.out.println();

// 测试用例 2: 基本情况 - 全正数数组
int[] nums2 = {1, 2, 3, 4, 5};
int result2 = maxSubArray(nums2);
System.out.println("测试用例 2:");
System.out.print("数组: [");
for (int i = 0; i < nums2.length; i++) {
    System.out.print(nums2[i]);
    if (i < nums2.length - 1) System.out.print(", ");
}
System.out.println("]");
System.out.println("最大子数组和: " + result2);
System.out.println("期望输出: 15");
System.out.println();

// 测试用例 3: 基本情况 - 全负数数组
int[] nums3 = {-2, -3, -1, -5};
int result3 = maxSubArray(nums3);
System.out.println("测试用例 3:");
System.out.print("数组: [");
for (int i = 0; i < nums3.length; i++) {
    System.out.print(nums3[i]);
    if (i < nums3.length - 1) System.out.print(", ");
}
System.out.println("]");
System.out.println("最大子数组和: " + result3);
System.out.println("期望输出: -1");
System.out.println();

// 测试用例 4: 边界情况 - 单元素数组
int[] nums4 = {5};
int result4 = maxSubArray(nums4);
```

```
        System.out.println("测试用例 4:");
        System.out.print("数组: [");
        System.out.print(nums4[0]);
        System.out.println("]");
        System.out.println("最大子数组和: " + result4);
        System.out.println("期望输出: 5");
        System.out.println();

        // 测试用例 5: 边界情况 - 空数组
        int[] nums5 = {};
        int result5 = maxSubArray(nums5);
        System.out.println("测试用例 5:");
        System.out.println("数组: []");
        System.out.println("最大子数组和: " + result5);
        System.out.println("期望输出: 0");
        System.out.println();

        // 测试用例 6: 复杂情况 - 混合数组
        int[] nums6 = {8, -19, 5, -4, 20};
        int result6 = maxSubArray(nums6);
        System.out.println("测试用例 6:");
        System.out.print("数组: [");
        for (int i = 0; i < nums6.length; i++) {
            System.out.print(nums6[i]);
            if (i < nums6.length - 1) System.out.print(", ");
        }
        System.out.println("]");
        System.out.println("最大子数组和: " + result6);
        System.out.println("期望输出: 21");
    }
}
```

==========

文件: Code15_MaximumSubarray.py

==========

```
# 最大子数组和（Maximum Subarray）
# 题目来源：LeetCode 53
# 题目链接：https://leetcode.cn/problems/maximum-subarray/
#
# 问题描述：
# 给定一个整数数组 nums，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），返回其最大和。
#
```

```
# 算法思路：
# 使用 Kadane 算法（贪心策略）：
# 1. 遍历数组，维护当前子数组和和最大子数组和
# 2. 如果当前子数组和为负数，重置为当前元素值（因为负数会减小后续和）
# 3. 否则，继续累加当前元素
# 4. 每次更新最大子数组和
#
# 时间复杂度：O(n) - 只需遍历数组一次
# 空间复杂度：O(1) - 只使用了常数额外空间
#
# 是否最优解：是。Kadane 算法是该问题的最优解法。
#
# 适用场景：
# 1. 连续子数组和问题
# 2. 最大收益问题
#
# 异常处理：
# 1. 处理空数组情况
# 2. 处理全负数数组
#
# 工程化考量：
# 1. 输入验证：检查数组是否为空
# 2. 边界条件：处理单元素数组
# 3. 性能优化：一次遍历完成计算
#
# 相关题目：
# 1. LeetCode 152. 乘积最大子数组 - 类似问题，但需要处理负数
# 2. LeetCode 121. 买卖股票的最佳时机 - 最大差值问题
# 3. LeetCode 918. 环形子数组的最大和 - 环形数组版本
# 4. 牛客网 NC140 排序 - 各种排序算法实现
# 5. LintCode 41. 最大子数组 - 与本题相同
# 6. HackerRank - Maximum Subarray Sum - 类似问题
# 7. CodeChef - MAXSUBA - 最大子数组问题
# 8. AtCoder ABC139D - ModSum - 数学相关
# 9. Codeforces 1370C - Number Game - 博弈论相关
# 10. POJ 2479 - Maximum sum - 双最大子数组和

class Solution:
    """
    计算最大子数组和

    Args:
        nums: List[int] - 整数数组
```

```
    Returns:
        int - 最大子数组和
    """
    def maxSubArray(self, nums):
        # 边界条件检查
        if not nums:
            return 0

        n = len(nums)
        if n == 1:
            return nums[0]  # 只有一个元素，直接返回

        max_sum = nums[0]  # 最大子数组和
        current_sum = nums[0]  # 当前子数组和

        for i in range(1, n):
            # 如果当前子数组和为负数，重置为当前元素值
            if current_sum < 0:
                current_sum = nums[i]
            else:
                # 否则，继续累加当前元素
                current_sum += nums[i]

            # 更新最大子数组和
            if current_sum > max_sum:
                max_sum = current_sum

        return max_sum


def main():
    solution = Solution()

    # 测试用例 1: 基本情况 - 正数数组
    nums1 = [-2, 1, -3, 4, -1, 2, 1, -5, 4]
    result1 = solution.maxSubArray(nums1)
    print("测试用例 1:")
    print(f"数组: {nums1}")
    print(f"最大子数组和: {result1}")
    print("期望输出: 6")
    print()
```

```
# 测试用例 2: 基本情况 - 全正数数组
nums2 = [1, 2, 3, 4, 5]
result2 = solution.maxSubArray(nums2)
print("测试用例 2:")
print(f"数组: {nums2}")
print(f"最大子数组和: {result2}")
print("期望输出: 15")
print()

# 测试用例 3: 基本情况 - 全负数数组
nums3 = [-2, -3, -1, -5]
result3 = solution.maxSubArray(nums3)
print("测试用例 3:")
print(f"数组: {nums3}")
print(f"最大子数组和: {result3}")
print("期望输出: -1")
print()

# 测试用例 4: 边界情况 - 单元素数组
nums4 = [5]
result4 = solution.maxSubArray(nums4)
print("测试用例 4:")
print(f"数组: {nums4}")
print(f"最大子数组和: {result4}")
print("期望输出: 5")
print()

# 测试用例 5: 边界情况 - 空数组
nums5 = []
result5 = solution.maxSubArray(nums5)
print("测试用例 5:")
print(f"数组: {nums5}")
print(f"最大子数组和: {result5}")
print("期望输出: 0")
print()

# 测试用例 6: 复杂情况 - 混合数组
nums6 = [8, -19, 5, -4, 20]
result6 = solution.maxSubArray(nums6)
print("测试用例 6:")
print(f"数组: {nums6}")
print(f"最大子数组和: {result6}")
print("期望输出: 21")
```

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

==========

文件: Code16_MergeIntervals.cpp

==========

```
// 合并区间（Merge Intervals）
// 题目来源：LeetCode 56
// 题目链接：https://leetcode.cn/problems/merge-intervals/
//
// 问题描述：
// 以数组 intervals 表示若干个区间的集合，请合并所有重叠的区间，并返回一个不重叠的区间数组。
//
// 算法思路：
// 使用贪心策略，按照区间开始时间排序：
// 1. 将区间按照开始时间从小到大排序
// 2. 遍历排序后的区间，维护当前合并区间
// 3. 如果当前区间与合并区间重叠，更新合并区间的结束时间
// 4. 否则，将当前合并区间加入结果，开始新的合并区间
//
// 时间复杂度：O(n log n) - 排序的时间复杂度
// 空间复杂度：O(n) - 需要存储结果区间
//
// 是否最优解：是。这是该问题的最优解法。
//
// 适用场景：
// 1. 区间合并问题
// 2. 重叠区间处理
//
// 异常处理：
// 1. 处理空数组情况
// 2. 处理单元素数组
//
// 工程化考量：
// 1. 输入验证：检查数组是否为空
// 2. 边界条件：处理单元素和双元素数组
// 3. 性能优化：使用快速排序提高效率
//
// 相关题目：
// 1. LeetCode 57. 插入区间 - 区间插入问题
```

```
// 2. LeetCode 252. 会议室 - 区间重叠判断
// 3. LeetCode 253. 会议室 II - 区间重叠计数
// 4. 牛客网 NC135 买票需要多少时间 - 队列模拟相关
// 5. LintCode 391. 数飞机 - 区间调度相关
// 6. HackerRank - Jim and the Orders - 贪心调度问题
// 7. CodeChef - TACHSTCK - 区间配对问题
// 8. AtCoder ABC104C - All Green - 动态规划相关
// 9. Codeforces 1363C - Game On Leaves - 博弈论相关
// 10. POJ 3169 - Layout - 差分约束系统

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

/**
 * 合并重叠区间
 *
 * @param intervals 区间数组，每个区间包含开始和结束时间
 * @return 合并后的不重叠区间数组
 */
vector<vector<int>> merge(vector<vector<int>>& intervals) {
    // 边界条件检查
    if (intervals.empty()) {
        return {};
    }

    int n = intervals.size();
    if (n == 1) {
        return intervals; // 只有一个区间，直接返回
    }

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

    vector<vector<int>> result;
    vector<int> currentInterval = intervals[0];
    result.push_back(currentInterval);

    for (int i = 1; i < n; i++) {
        int currentEnd = currentInterval[1];
        int nextStart = intervals[i][0];
```

```
        int nextEnd = intervals[i][1];

        // 如果当前区间与下一个区间重叠
        if (currentEnd >= nextStart) {
            // 合并区间，取较大的结束时间
            currentInterval[1] = max(currentEnd, nextEnd);
            result.back()[1] = currentInterval[1]; // 更新结果中的最后一个区间
        } else {
            // 不重叠，开始新的合并区间
            currentInterval = intervals[i];
            result.push_back(currentInterval);
        }
    }

    return result;
}

// 测试函数
int main() {
    // 测试用例 1: 基本情况 - 有重叠区间
    vector<vector<int>> intervals1 = {{1, 3}, {2, 6}, {8, 10}, {15, 18}};
    vector<vector<int>> result1 = merge(intervals1);

    // 测试用例 2: 基本情况 - 无重叠区间
    vector<vector<int>> intervals2 = {{1, 4}, {5, 8}, {9, 12}};
    vector<vector<int>> result2 = merge(intervals2);

    // 测试用例 3: 复杂情况 - 完全重叠
    vector<vector<int>> intervals3 = {{1, 4}, {2, 3}, {3, 5}};
    vector<vector<int>> result3 = merge(intervals3);

    // 测试用例 4: 边界情况 - 单元素数组
    vector<vector<int>> intervals4 = {{1, 3}};
    vector<vector<int>> result4 = merge(intervals4);

    // 测试用例 5: 边界情况 - 空数组
    vector<vector<int>> intervals5 = {};
    vector<vector<int>> result5 = merge(intervals5);

    // 测试用例 6: 复杂情况 - 包含区间
    vector<vector<int>> intervals6 = {{1, 10}, {2, 3}, {4, 5}, {6, 7}, {8, 9}};
    vector<vector<int>> result6 = merge(intervals6);
```

```
    return 0;
}
```

==========

文件: Code16_MergeIntervals.java
==========

```
package class093;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.Comparator;
import java.util.List;

/**
 * 合并区间（Merge Intervals）
 * 题目来源：LeetCode 56
 * 题目链接：https://leetcode.cn/problems/merge-intervals/
 *
 * 问题描述：
 * 以数组 intervals 表示若干个区间的集合，请合并所有重叠的区间，并返回一个不重叠的区间数组。
 *
 * 算法思路：
 * 使用贪心策略，按照区间开始时间排序：
 * 1. 将区间按照开始时间从小到大排序
 * 2. 遍历排序后的区间，维护当前合并区间
 * 3. 如果当前区间与合并区间重叠，更新合并区间的结束时间
 * 4. 否则，将当前合并区间加入结果，开始新的合并区间
 *
 * 时间复杂度：O(n log n) - 排序的时间复杂度
 * 空间复杂度：O(n) - 需要存储结果区间
 *
 * 是否最优解：是。这是该问题的最优解法。
 *
 * 适用场景：
 * 1. 区间合并问题
 * 2. 重叠区间处理
 *
 * 异常处理：
 * 1. 处理空数组情况
 * 2. 处理单元素数组
 *
 * 工程化考量：
```

```
 * 1. 输入验证：检查数组是否为空
 * 2. 边界条件：处理单元素和双元素数组
 * 3. 性能优化：使用快速排序提高效率
 *
 * 相关题目：
 * 1. LeetCode 57. 插入区间 - 区间插入问题
 * 2. LeetCode 252. 会议室 - 区间重叠判断
 * 3. LeetCode 253. 会议室 II - 区间重叠计数
 * 4. 牛客网 NC135 买票需要多少时间 - 队列模拟相关
 * 5. LintCode 391. 数飞机 - 区间调度相关
 * 6. HackerRank - Jim and the Orders - 贪心调度问题
 * 7. CodeChef - TACHSTCK - 区间配对问题
 * 8. AtCoder ABC104C - All Green - 动态规划相关
 * 9. Codeforces 1363C - Game On Leaves - 博弈论相关
 * 10. POJ 3169 - Layout - 差分约束系统
 */
public class Code16_MergeIntervals {

    /**
     * 合并重叠区间
     *
     * @param intervals 区间数组，每个区间包含开始和结束时间
     * @return 合并后的不重叠区间数组
     */
    public static int[][] merge(int[][] intervals) {
        // 边界条件检查
        if (intervals == null || intervals.length == 0) {
            return new int[0][];
        }

        int n = intervals.length;
        if (n == 1) {
            return intervals; // 只有一个区间，直接返回
        }

        // 按照区间开始时间排序
        Arrays.sort(intervals, new Comparator<int[]>() {
            @Override
            public int compare(int[] a, int[] b) {
                return Integer.compare(a[0], b[0]);
            }
        });
```

```
    List<int[]> result = new ArrayList<>();
    int[] currentInterval = intervals[0];
    result.add(currentInterval);

    for (int i = 1; i < n; i++) {
        int currentEnd = currentInterval[1];
        int nextStart = intervals[i][0];
        int nextEnd = intervals[i][1];

        // 如果当前区间与下一个区间重叠
        if (currentEnd >= nextStart) {
            // 合并区间，取较大的结束时间
            currentInterval[1] = Math.max(currentEnd, nextEnd);
        } else {
            // 不重叠，开始新的合并区间
            currentInterval = intervals[i];
            result.add(currentInterval);
        }
    }

    return result.toArray(new int[result.size()][]);
}

/**
 * 测试函数，验证算法正确性
 */
public static void main(String[] args) {
    // 测试用例 1: 基本情况 - 有重叠区间
    int[][] intervals1 = {{1, 3}, {2, 6}, {8, 10}, {15, 18}};
    int[][] result1 = merge(intervals1);
    System.out.println("测试用例 1:");
    System.out.print("输入区间: [");
    for (int i = 0; i < intervals1.length; i++) {
        System.out.print("[" + intervals1[i][0] + "," + intervals1[i][1] + "]");
        if (i < intervals1.length - 1) System.out.print(", ");
    }
    System.out.println("]");
    System.out.print("合并结果: [");
    for (int i = 0; i < result1.length; i++) {
        System.out.print("[" + result1[i][0] + "," + result1[i][1] + "]");
        if (i < result1.length - 1) System.out.print(", ");
    }
    System.out.println("]");
```

```
System.out.println("期望输出: [[1,6],[8,10],[15,18]]");
System.out.println();

// 测试用例 2: 基本情况 - 无重叠区间
int[][] intervals2 = {{1, 4}, {5, 8}, {9, 12}};
int[][] result2 = merge(intervals2);
System.out.println("测试用例 2:");
System.out.print("输入区间: [");
for (int i = 0; i < intervals2.length; i++) {
    System.out.print("[" + intervals2[i][0] + "," + intervals2[i][1] + "]");
    if (i < intervals2.length - 1) System.out.print(", ");
}
System.out.println("]");
System.out.print("合并结果: [");
for (int i = 0; i < result2.length; i++) {
    System.out.print("[" + result2[i][0] + "," + result2[i][1] + "]");
    if (i < result2.length - 1) System.out.print(", ");
}
System.out.println("]");
System.out.println("期望输出: [[1,4],[5,8],[9,12]]");
System.out.println();

// 测试用例 3: 复杂情况 - 完全重叠
int[][] intervals3 = {{1, 4}, {2, 3}, {3, 5}};
int[][] result3 = merge(intervals3);
System.out.println("测试用例 3:");
System.out.print("输入区间: [");
for (int i = 0; i < intervals3.length; i++) {
    System.out.print("[" + intervals3[i][0] + "," + intervals3[i][1] + "]");
    if (i < intervals3.length - 1) System.out.print(", ");
}
System.out.println("]");
System.out.print("合并结果: [");
for (int i = 0; i < result3.length; i++) {
    System.out.print("[" + result3[i][0] + "," + result3[i][1] + "]");
    if (i < result3.length - 1) System.out.print(", ");
}
System.out.println("]");
System.out.println("期望输出: [[1,5]]");
System.out.println();

// 测试用例 4: 边界情况 - 单元素数组
int[][] intervals4 = {{1, 3}};
```

```
        int[][] result4 = merge(intervals4);
        System.out.println("测试用例 4:");
        System.out.print("输入区间: [");
        System.out.print("[" + intervals4[0][0] + "," + intervals4[0][1] + "]");
        System.out.println("]");
        System.out.print("合并结果: [");
        System.out.print("[" + result4[0][0] + "," + result4[0][1] + "]");
        System.out.println("]");
        System.out.println("期望输出: [[1,3]]");
        System.out.println();

        // 测试用例 5: 边界情况 - 空数组
        int[][] intervals5 = {};
        int[][] result5 = merge(intervals5);
        System.out.println("测试用例 5:");
        System.out.println("输入区间: []");
        System.out.println("合并结果: []");
        System.out.println("期望输出: []");
        System.out.println();

        // 测试用例 6: 复杂情况 - 包含区间
        int[][] intervals6 = {{1, 10}, {2, 3}, {4, 5}, {6, 7}, {8, 9}};
        int[][] result6 = merge(intervals6);
        System.out.println("测试用例 6:");
        System.out.print("输入区间: [");
        for (int i = 0; i < intervals6.length; i++) {
            System.out.print("[" + intervals6[i][0] + "," + intervals6[i][1] + "]");
            if (i < intervals6.length - 1) System.out.print(", ");
        }
        System.out.println("]");
        System.out.print("合并结果: [");
        for (int i = 0; i < result6.length; i++) {
            System.out.print("[" + result6[i][0] + "," + result6[i][1] + "]");
            if (i < result6.length - 1) System.out.print(", ");
        }
        System.out.println("]");
        System.out.println("期望输出: [[1,10]]");
    }
}
```

==========

文件: Code16_MergeIntervals.py

```
==========================================
# 合并区间（Merge Intervals）
# 题目来源：LeetCode 56
# 题目链接：https://leetcode.cn/problems/merge-intervals/
#
# 问题描述：
# 以数组 intervals 表示若干个区间的集合，请合并所有重叠的区间，并返回一个不重叠的区间数组。
#
# 算法思路：
# 使用贪心策略，按照区间开始时间排序：
# 1. 将区间按照开始时间从小到大排序
# 2. 遍历排序后的区间，维护当前合并区间
# 3. 如果当前区间与合并区间重叠，更新合并区间的结束时间
# 4. 否则，将当前合并区间加入结果，开始新的合并区间
#
# 时间复杂度：O(n log n) - 排序的时间复杂度
# 空间复杂度：O(n) - 需要存储结果区间
#
# 是否最优解：是。这是该问题的最优解法。
#
# 适用场景：
# 1. 区间合并问题
# 2. 重叠区间处理
#
# 异常处理：
# 1. 处理空数组情况
# 2. 处理单元素数组
#
# 工程化考量：
# 1. 输入验证：检查数组是否为空
# 2. 边界条件：处理单元素和双元素数组
# 3. 性能优化：使用内置排序提高效率
#
# 相关题目：
# 1. LeetCode 57. 插入区间 - 区间插入问题
# 2. LeetCode 252. 会议室 - 区间重叠判断
# 3. LeetCode 253. 会议室 II - 区间重叠计数
# 4. 牛客网 NC135 买票需要多少时间 - 队列模拟相关
# 5. LintCode 391. 数飞机 - 区间调度相关
# 6. HackerRank - Jim and the Orders - 贪心调度问题
# 7. CodeChef - TACHSTCK - 区间配对问题
# 8. AtCoder ABC104C - All Green - 动态规划相关
# 9. Codeforces 1363C - Game On Leaves - 博弈论相关
```

10. POJ 3169 - Layout - 差分约束系统

```
class Solution:
    """
    合并重叠区间

    Args:
        intervals: List[List[int]] - 区间数组，每个区间包含开始和结束时间

    Returns:
        List[List[int]] - 合并后的不重叠区间数组
    """
    def merge(self, intervals):
        # 边界条件检查
        if not intervals:
            return []

        n = len(intervals)
        if n == 1:
            return intervals  # 只有一个区间，直接返回

        # 按照区间开始时间排序
        intervals.sort(key=lambda x: x[0])

        result = []
        current_interval = intervals[0]
        result.append(current_interval)

        for i in range(1, n):
            current_end = current_interval[1]
            next_start = intervals[i][0]
            next_end = intervals[i][1]

            # 如果当前区间与下一个区间重叠
            if current_end >= next_start:
                # 合并区间，取较大的结束时间
                current_interval[1] = max(current_end, next_end)
                result[-1][1] = current_interval[1]  # 更新结果中的最后一个区间
            else:
                # 不重叠，开始新的合并区间
                current_interval = intervals[i]
                result.append(current_interval)
```

```
        return result


def main():
    solution = Solution()

    # 测试用例 1: 基本情况 - 有重叠区间
    intervals1 = [[1, 3], [2, 6], [8, 10], [15, 18]]
    result1 = solution.merge(intervals1)
    print("测试用例 1:")
    print(f"输入区间: {intervals1}")
    print(f"合并结果: {result1}")
    print("期望输出: [[1, 6], [8, 10], [15, 18]]")
    print()

    # 测试用例 2: 基本情况 - 无重叠区间
    intervals2 = [[1, 4], [5, 8], [9, 12]]
    result2 = solution.merge(intervals2)
    print("测试用例 2:")
    print(f"输入区间: {intervals2}")
    print(f"合并结果: {result2}")
    print("期望输出: [[1, 4], [5, 8], [9, 12]]")
    print()

    # 测试用例 3: 复杂情况 - 完全重叠
    intervals3 = [[1, 4], [2, 3], [3, 5]]
    result3 = solution.merge(intervals3)
    print("测试用例 3:")
    print(f"输入区间: {intervals3}")
    print(f"合并结果: {result3}")
    print("期望输出: [[1, 5]]")
    print()

    # 测试用例 4: 边界情况 - 单元素数组
    intervals4 = [[1, 3]]
    result4 = solution.merge(intervals4)
    print("测试用例 4:")
    print(f"输入区间: {intervals4}")
    print(f"合并结果: {result4}")
    print("期望输出: [[1, 3]]")
    print()

    # 测试用例 5: 边界情况 - 空数组
```

```
    intervals5 = []
    result5 = solution.merge(intervals5)
    print("测试用例 5:")
    print(f"输入区间: {intervals5}")
    print(f"合并结果: {result5}")
    print("期望输出: []")
    print()

    # 测试用例 6: 复杂情况 - 包含区间
    intervals6 = [[1, 10], [2, 3], [4, 5], [6, 7], [8, 9]]
    result6 = solution.merge(intervals6)
    print("测试用例 6:")
    print(f"输入区间: {intervals6}")
    print(f"合并结果: {result6}")
    print("期望输出: [[1, 10]]")


if __name__ == "__main__":
    main()

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

文件: Code17_QueueReconstructionByHeight.cpp
================================================
// 根据身高重建队列（Queue Reconstruction by Height）
// 题目来源：LeetCode 406
// 题目链接：https://leetcode.cn/problems/queue-reconstruction-by-height/
//
// 问题描述：
// 假设有打乱顺序的一群人站成一个队列，每个人由一个整数对(h, k)表示，
// 其中 h 是这个人的身高，k 是排在这个人前面且身高大于或等于 h 的人数。
// 请重建这个队列，使其满足上述要求。
//
// 算法思路：
// 使用贪心策略，按照身高降序、k 值升序排序：
// 1. 将人群按照身高降序、k 值升序排序
// 2. 按照排序后的顺序，将每个人插入到结果队列的第 k 个位置
// 3. 这样能保证前面身高更高的人先被放置，后面插入的人不会影响前面人的 k 值
//
// 时间复杂度：O(n²) - 插入操作的时间复杂度
// 空间复杂度：O(n) - 需要存储结果队列
//
// 是否最优解：是。这是该问题的最优解法。
```

```
//
// 适用场景：
// 1. 队列重建问题
// 2. 带约束的排序问题
//
// 异常处理：
// 1. 处理空数组情况
// 2. 处理单元素数组
//
// 工程化考量：
// 1. 输入验证：检查数组是否为空
// 2. 边界条件：处理单元素和双元素数组
// 3. 性能优化：使用链表提高插入效率
//
// 相关题目：
// 1. LeetCode 135. 分发糖果 - 双向约束问题
// 2. LeetCode 56. 合并区间 - 区间合并问题
// 3. LeetCode 252. 会议室 - 区间重叠判断
// 4. 牛客网 NC140 排序 - 各种排序算法实现
// 5. LintCode 391. 数飞机 - 区间调度相关
// 6. HackerRank - Jim and the Orders - 贪心调度问题
// 7. CodeChef - TACHSTCK - 区间配对问题
// 8. AtCoder ABC104C - All Green - 动态规划相关
// 9. Codeforces 1363C - Game On Leaves - 博弈论相关
// 10. POJ 3169 - Layout - 差分约束系统

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

/**
 * 重建队列
 *
 * @param people 人群数组，每个元素是[h, k]
 * @return 重建后的队列
 */
vector<vector<int>> reconstructQueue(vector<vector<int>>& people) {
    // 边界条件检查
    if (people.empty()) {
        return {};
    }
```

```
    int n = people.size();
    if (n == 1) {
        return people; // 只有一个人，直接返回
    }

    // 按照身高降序、k 值升序排序
    sort(people.begin(), people.end(), [](const vector<int>& a, const vector<int>& b) {
        if (a[0] == b[0]) {
            return a[1] < b[1]; // 身高相同，按 k 值升序
        }
        return a[0] > b[0]; // 身高降序
    });

    // 使用链表提高插入效率
    list<vector<int>> resultList;

    for (const auto& person : people) {
        auto it = resultList.begin();
        advance(it, person[1]); // 移动到第 k 个位置
        resultList.insert(it, person);
    }

    // 将链表转换为向量
    vector<vector<int>> result(resultList.begin(), resultList.end());
    return result;
}

// 测试函数
int main() {
    // 测试用例 1: 基本情况
    vector<vector<int>> people1 = {{7, 0}, {4, 4}, {7, 1}, {5, 0}, {6, 1}, {5, 2}};
    vector<vector<int>> result1 = reconstructQueue(people1);

    // 测试用例 2: 简单情况
    vector<vector<int>> people2 = {{6, 0}, {5, 0}, {4, 0}, {3, 2}, {2, 2}, {1, 4}};
    vector<vector<int>> result2 = reconstructQueue(people2);

    // 测试用例 3: 边界情况 - 单元素数组
    vector<vector<int>> people3 = {{5, 0}};
    vector<vector<int>> result3 = reconstructQueue(people3);

    // 测试用例 4: 边界情况 - 空数组
    vector<vector<int>> people4 = {};
```

```
    vector<vector<int>> result4 = reconstructQueue(people4);

    // 测试用例 5: 复杂情况 - 相同身高
    vector<vector<int>> people5 = {{5, 2}, {5, 0}, {5, 1}, {4, 0}, {4, 1}};
    vector<vector<int>> result5 = reconstructQueue(people5);

    return 0;
}
```

==========

文件: Code17_QueueReconstructionByHeight.java

==========

```
package class093;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.Comparator;
import java.util.List;

/**
 * 根据身高重建队列（Queue Reconstruction by Height）
 * 题目来源：LeetCode 406
 * 题目链接：https://leetcode.cn/problems/queue-reconstruction-by-height/
 *
 * 问题描述：
 * 假设有打乱顺序的一群人站成一个队列，每个人由一个整数对(h, k)表示，
 * 其中 h 是这个人的身高，k 是排在这个人前面且身高大于或等于 h 的人数。
 * 请重建这个队列，使其满足上述要求。
 *
 * 算法思路：
 * 使用贪心策略，按照身高降序、k 值升序排序：
 * 1. 将人群按照身高降序、k 值升序排序
 * 2. 按照排序后的顺序，将每个人插入到结果队列的第 k 个位置
 * 3. 这样能保证前面身高更高的人先被放置，后面插入的人不会影响前面人的 k 值
 *
 * 时间复杂度：O(n²) - 插入操作的时间复杂度
 * 空间复杂度：O(n) - 需要存储结果队列
 *
 * 是否最优解：是。这是该问题的最优解法。
 *
 * 适用场景：
 * 1. 队列重建问题
```

```
 * 2. 带约束的排序问题
 *
 * 异常处理：
 * 1. 处理空数组情况
 * 2. 处理单元素数组
 *
 * 工程化考量：
 * 1. 输入验证：检查数组是否为空
 * 2. 边界条件：处理单元素和双元素数组
 * 3. 性能优化：使用链表提高插入效率
 *
 * 相关题目：
 * 1. LeetCode 135. 分发糖果 - 双向约束问题
 * 2. LeetCode 56. 合并区间 - 区间合并问题
 * 3. LeetCode 252. 会议室 - 区间重叠判断
 * 4. 牛客网 NC140 排序 - 各种排序算法实现
 * 5. LintCode 391. 数飞机 - 区间调度相关
 * 6. HackerRank - Jim and the Orders - 贪心调度问题
 * 7. CodeChef - TACHSTCK - 区间配对问题
 * 8. AtCoder ABC104C - All Green - 动态规划相关
 * 9. Codeforces 1363C - Game On Leaves - 博弈论相关
 * 10. POJ 3169 - Layout - 差分约束系统
 */
public class Code17_QueueReconstructionByHeight {

    /**
     * 重建队列
     *
     * @param people 人群数组，每个元素是[h, k]
     * @return 重建后的队列
     */
    public static int[][] reconstructQueue(int[][] people) {
        // 边界条件检查
        if (people == null || people.length == 0) {
            return new int[0][];
        }

        int n = people.length;
        if (n == 1) {
            return people; // 只有一个人，直接返回
        }

        // 按照身高降序、k 值升序排序
```

```
    Arrays.sort(people, new Comparator<int[]>() {
        @Override
        public int compare(int[] a, int[] b) {
            if (a[0] == b[0]) {
                return Integer.compare(a[1], b[1]); // 身高相同，按 k 值升序
            }
            return Integer.compare(b[0], a[0]); // 身高降序
        }
    });

    // 使用链表提高插入效率
    List<int[]> result = new ArrayList<>();

    for (int[] person : people) {
        // 将每个人插入到第 k 个位置
        result.add(person[1], person);
    }

    return result.toArray(new int[result.size()][]);
}

/**
 * 测试函数，验证算法正确性
 */
public static void main(String[] args) {
    // 测试用例 1: 基本情况
    int[][] people1 = {{7, 0}, {4, 4}, {7, 1}, {5, 0}, {6, 1}, {5, 2}};
    int[][] result1 = reconstructQueue(people1);
    System.out.println("测试用例 1:");
    System.out.print("输入人群: [");
    for (int i = 0; i < people1.length; i++) {
        System.out.print("[" + people1[i][0] + "," + people1[i][1] + "]");
        if (i < people1.length - 1) System.out.print(", ");
    }
    System.out.println("]");
    System.out.print("重建队列: [");
    for (int i = 0; i < result1.length; i++) {
        System.out.print("[" + result1[i][0] + "," + result1[i][1] + "]");
        if (i < result1.length - 1) System.out.print(", ");
    }
    System.out.println("]");
    System.out.println("期望输出: [[5,0],[7,0],[5,2],[6,1],[4,4],[7,1]]");
    System.out.println();
```

```
// 测试用例 2: 简单情况
int[][] people2 = {{6, 0}, {5, 0}, {4, 0}, {3, 2}, {2, 2}, {1, 4}};
int[][] result2 = reconstructQueue(people2);
System.out.println("测试用例 2:");
System.out.print("输入人群: [");
for (int i = 0; i < people2.length; i++) {
    System.out.print("[" + people2[i][0] + "," + people2[i][1] + "]");
    if (i < people2.length - 1) System.out.print(", ");
}
System.out.println("]");
System.out.print("重建队列: [");
for (int i = 0; i < result2.length; i++) {
    System.out.print("[" + result2[i][0] + "," + result2[i][1] + "]");
    if (i < result2.length - 1) System.out.print(", ");
}
System.out.println("]");
System.out.println("期望输出: [[4,0],[5,0],[2,2],[3,2],[1,4],[6,0]]");
System.out.println();

// 测试用例 3: 边界情况 - 单元素数组
int[][] people3 = {{5, 0}};
int[][] result3 = reconstructQueue(people3);
System.out.println("测试用例 3:");
System.out.print("输入人群: [");
System.out.print("[" + people3[0][0] + "," + people3[0][1] + "]");
System.out.println("]");
System.out.print("重建队列: [");
System.out.print("[" + result3[0][0] + "," + result3[0][1] + "]");
System.out.println("]");
System.out.println("期望输出: [[5,0]]");
System.out.println();

// 测试用例 4: 边界情况 - 空数组
int[][] people4 = {};
int[][] result4 = reconstructQueue(people4);
System.out.println("测试用例 4:");
System.out.println("输入人群: []");
System.out.println("重建队列: []");
System.out.println("期望输出: []");
System.out.println();

// 测试用例 5: 复杂情况 - 相同身高
```

```
        int[][] people5 = {{5, 2}, {5, 0}, {5, 1}, {4, 0}, {4, 1}};
        int[][] result5 = reconstructQueue(people5);
        System.out.println("测试用例 5:");
        System.out.print("输入人群: [");
        for (int i = 0; i < people5.length; i++) {
            System.out.print("[" + people5[i][0] + "," + people5[i][1] + "]");
            if (i < people5.length - 1) System.out.print(", ");
        }
        System.out.println("]");
        System.out.print("重建队列: [");
        for (int i = 0; i < result5.length; i++) {
            System.out.print("[" + result5[i][0] + "," + result5[i][1] + "]");
            if (i < result5.length - 1) System.out.print(", ");
        }
        System.out.println("]");
        System.out.println("期望输出: [[4,0],[5,0],[5,1],[4,1],[5,2]]");
    }
}
```

==========

文件: Code17_QueueReconstructionByHeight.py

==========

```
# 根据身高重建队列（Queue Reconstruction by Height）
# 题目来源：LeetCode 406
# 题目链接：https://leetcode.cn/problems/queue-reconstruction-by-height/
#
# 问题描述：
# 假设有打乱顺序的一群人站成一个队列，每个人由一个整数对(h, k)表示，
# 其中 h 是这个人的身高，k 是排在这个人前面且身高大于或等于 h 的人数。
# 请重建这个队列，使其满足上述要求。
#
# 算法思路：
# 使用贪心策略，按照身高降序、k 值升序排序：
# 1. 将人群按照身高降序、k 值升序排序
# 2. 按照排序后的顺序，将每个人插入到结果队列的第 k 个位置
# 3. 这样能保证前面身高更高的人先被放置，后面插入的人不会影响前面人的 k 值
#
# 时间复杂度：O(n²) - 插入操作的时间复杂度
# 空间复杂度：O(n) - 需要存储结果队列
#
# 是否最优解：是。这是该问题的最优解法。
#
```

```
# 适用场景：
# 1. 队列重建问题
# 2. 带约束的排序问题
#
# 异常处理：
# 1. 处理空数组情况
# 2. 处理单元素数组
#
# 工程化考量：
# 1. 输入验证：检查数组是否为空
# 2. 边界条件：处理单元素和双元素数组
# 3. 性能优化：使用列表提高插入效率
#
# 相关题目：
# 1. LeetCode 135. 分发糖果 - 双向约束问题
# 2. LeetCode 56. 合并区间 - 区间合并问题
# 3. LeetCode 252. 会议室 - 区间重叠判断
# 4. 牛客网 NC140 排序 - 各种排序算法实现
# 5. LintCode 391. 数飞机 - 区间调度相关
# 6. HackerRank - Jim and the Orders - 贪心调度问题
# 7. CodeChef - TACHSTCK - 区间配对问题
# 8. AtCoder ABC104C - All Green - 动态规划相关
# 9. Codeforces 1363C - Game On Leaves - 博弈论相关
# 10. POJ 3169 - Layout - 差分约束系统

class Solution:
    """
    重建队列

    Args:
        people: List[List[int]] - 人群数组，每个元素是[h, k]

    Returns:
        List[List[int]] - 重建后的队列
    """
    def reconstructQueue(self, people):
        # 边界条件检查
        if not people:
            return []

        n = len(people)
        if n == 1:
            return people  # 只有一个人，直接返回
```

```
        # 按照身高降序、k 值升序排序
        people.sort(key=lambda x: (-x[0], x[1]))

        # 使用列表存储结果
        result = []

        for person in people:
            # 将每个人插入到第 k 个位置
            result.insert(person[1], person)

        return result


def main():
    solution = Solution()

    # 测试用例 1: 基本情况
    people1 = [[7, 0], [4, 4], [7, 1], [5, 0], [6, 1], [5, 2]]
    result1 = solution.reconstructQueue(people1)
    print("测试用例 1:")
    print(f"输入人群: {people1}")
    print(f"重建队列: {result1}")
    print("期望输出: [[5,0],[7,0],[5,2],[6,1],[4,4],[7,1]]")
    print()

    # 测试用例 2: 简单情况
    people2 = [[6, 0], [5, 0], [4, 0], [3, 2], [2, 2], [1, 4]]
    result2 = solution.reconstructQueue(people2)
    print("测试用例 2:")
    print(f"输入人群: {people2}")
    print(f"重建队列: {result2}")
    print("期望输出: [[4,0],[5,0],[2,2],[3,2],[1,4],[6,0]]")
    print()

    # 测试用例 3: 边界情况 - 单元素数组
    people3 = [[5, 0]]
    result3 = solution.reconstructQueue(people3)
    print("测试用例 3:")
    print(f"输入人群: {people3}")
    print(f"重建队列: {result3}")
    print("期望输出: [[5,0]]")
    print()
```

```
    # 测试用例 4: 边界情况 - 空数组
    people4 = []
    result4 = solution.reconstructQueue(people4)
    print("测试用例 4:")
    print(f"输入人群: {people4}")
    print(f"重建队列: {result4}")
    print("期望输出: []")
    print()

    # 测试用例 5: 复杂情况 - 相同身高
    people5 = [[5, 2], [5, 0], [5, 1], [4, 0], [4, 1]]
    result5 = solution.reconstructQueue(people5)
    print("测试用例 5:")
    print(f"输入人群: {people5}")
    print(f"重建队列: {result5}")
    print("期望输出: [[4,0],[5,0],[5,1],[4,1],[5,2]]")


if __name__ == "__main__":
    main()

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

文件: Code18_MinimumRefuelingStops.cpp
==============================================
// 最低加油次数（Minimum Number of Refueling Stops）
// 题目来源：LeetCode 871
// 题目链接：https://leetcode.cn/problems/minimum-number-of-refueling-stops/
//
// 问题描述：
// 汽车从起点出发驶向目的地，该目的地距离起点 target 英里。
// 沿途有加油站，每个 station[i]代表一个加油站，位于距离起点 station[i][0]英里处，有 station[i][1]
升汽油。
// 假设汽车油箱的容量是无限的，其中最初有 startFuel 升燃料。
// 它每行驶 1 英里就会用掉 1 升汽油。
// 当汽车到达加油站时，它可能停下来加油，将所有汽油从加油站转移到汽车中。
// 为了到达目的地，汽车所必要的最低加油次数是多少？如果无法到达目的地，则返回-1。
//
// 算法思路：
// 使用贪心策略，结合最大堆：
// 1. 遍历加油站，维护当前能到达的最远位置
// 2. 当油量不足以到达下一个加油站时，从经过的加油站中选择油量最大的进行加油
```

```
// 3. 使用最大堆来存储经过的加油站的油量
// 4. 每次加油后更新当前油量和加油次数
//
// 时间复杂度：O(n log n) - 堆操作的时间复杂度
// 空间复杂度：O(n) - 最大堆的空间
//
// 是否最优解：是。这是该问题的最优解法。
//
// 适用场景：
// 1. 路径规划问题
// 2. 资源调度问题
//
// 异常处理：
// 1. 处理无法到达目的地的情况
// 2. 处理边界条件
//
// 工程化考量：
// 1. 输入验证：检查参数是否合法
// 2. 边界条件：处理起点就是目的地的情况
// 3. 性能优化：使用堆提高效率
//
// 相关题目：
// 1. LeetCode 134. 加油站 - 环路加油问题
// 2. LeetCode 45. 跳跃游戏 II - 最少跳跃次数
// 3. LeetCode 55. 跳跃游戏 - 可达性判断
// 4. 牛客网 NC48 跳跃游戏 - 与 LeetCode 55 相同
// 5. LintCode 117. 跳跃游戏 II - 与 LeetCode 45 相同
// 6. HackerRank - Jumping on the Clouds - 简化版跳跃游戏
// 7. CodeChef - JUMP - 类似跳跃游戏的变种
// 8. AtCoder ABC161D - Lunlun Number - BFS 搜索相关
// 9. Codeforces 1324C - Frog Jumps - 贪心跳跃问题
// 10. POJ 1700 - Crossing River - 经典过河问题

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

/**
 * 计算最低加油次数
 *
 * @param target 目的地距离
 * @param startFuel 初始油量
```

```
 * @param stations 加油站数组，每个元素是[位置, 油量]
 * @return 最低加油次数，无法到达返回-1
 */
int minRefuelStops(int target, int startFuel, vector<vector<int>>& stations) {
    // 边界条件检查
    if (startFuel >= target) {
        return 0; // 初始油量足够到达目的地
    }

    if (stations.empty()) {
        return startFuel >= target ? 0 : -1; // 没有加油站，检查初始油量是否足够
    }

    // 最大堆，存储经过的加油站的油量
    priority_queue<int> maxHeap;

    int currentFuel = startFuel; // 当前油量
    int currentPosition = 0;    // 当前位置
    int refuelCount = 0;        // 加油次数
    int stationIndex = 0;        // 加油站索引

    while (currentPosition + currentFuel < target) {
        int nextPosition = currentPosition + currentFuel; // 当前能到达的最远位置

        // 将能到达的加油站加入最大堆
        while (stationIndex < stations.size() && stations[stationIndex][0] <= nextPosition) {
            maxHeap.push(stations[stationIndex][1]);
            stationIndex++;
        }

        // 如果没有加油站可加油，且无法到达目的地
        if (maxHeap.empty()) {
            return -1;
        }

        // 选择油量最大的加油站加油
        int maxFuel = maxHeap.top();
        maxHeap.pop();
        currentFuel = currentFuel - (stations[stationIndex - 1][0] - currentPosition) + maxFuel;
        currentPosition = stations[stationIndex - 1][0];
        refuelCount++;

        // 如果加油后能到达目的地
```

```
        if (currentPosition + currentFuel >= target) {
            return refuelCount;
        }
    }

    return refuelCount;
}

// 测试函数
int main() {
    // 测试用例 1: 基本情况 - 需要加油
    int target1 = 100;
    int startFuel1 = 10;
    vector<vector<int>> stations1 = {{10, 60}, {20, 30}, {30, 30}, {60, 40}};
    int result1 = minRefuelStops(target1, startFuel1, stations1);

    // 测试用例 2: 基本情况 - 无法到达
    int target2 = 100;
    int startFuel2 = 10;
    vector<vector<int>> stations2 = {{20, 20}};
    int result2 = minRefuelStops(target2, startFuel2, stations2);

    // 测试用例 3: 边界情况 - 不需要加油
    int target3 = 50;
    int startFuel3 = 60;
    vector<vector<int>> stations3 = {{10, 20}, {20, 30}};
    int result3 = minRefuelStops(target3, startFuel3, stations3);

    // 测试用例 4: 边界情况 - 没有加油站
    int target4 = 50;
    int startFuel4 = 40;
    vector<vector<int>> stations4 = {};
    int result4 = minRefuelStops(target4, startFuel4, stations4);

    // 测试用例 5: 复杂情况 - 多个加油站
    int target5 = 100;
    int startFuel5 = 20;
    vector<vector<int>> stations5 = {{10, 10}, {20, 20}, {30, 30}, {40, 40}, {50, 50}};
    int result5 = minRefuelStops(target5, startFuel5, stations5);

    return 0;
}
```

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

文件: Code18_MinimumRefuelingStops.java
==========================================
package class093;

import java.util.Collections;
import java.util.PriorityQueue;

/**
 * 最低加油次数（Minimum Number of Refueling Stops）
 * 题目来源：LeetCode 871
 * 题目链接：https://leetcode.cn/problems/minimum-number-of-refueling-stops/
 *
 * 问题描述：
 * 汽车从起点出发驶向目的地，该目的地距离起点 target 英里。
 * 沿途有加油站，每个 station[i]代表一个加油站，位于距离起点 station[i][0]英里处，有 station[i][1]升汽油。
 * 假设汽车油箱的容量是无限的，其中最初有 startFuel 升燃料。
 * 它每行驶 1 英里就会用掉 1 升汽油。
 * 当汽车到达加油站时，它可能停下来加油，将所有汽油从加油站转移到汽车中。
 * 为了到达目的地，汽车所必要的最低加油次数是多少？如果无法到达目的地，则返回-1。
 *
 * 算法思路：
 * 使用贪心策略，结合最大堆：
 * 1. 遍历加油站，维护当前能到达的最远位置
 * 2. 当油量不足以到达下一个加油站时，从经过的加油站中选择油量最大的进行加油
 * 3. 使用最大堆来存储经过的加油站的油量
 * 4. 每次加油后更新当前油量和加油次数
 *
 * 时间复杂度：O(n log n) - 堆操作的时间复杂度
 * 空间复杂度：O(n) - 最大堆的空间
 *
 * 是否最优解：是。这是该问题的最优解法。
 *
 * 适用场景：
 * 1. 路径规划问题
 * 2. 资源调度问题
 *
 * 异常处理：
 * 1. 处理无法到达目的地的情况
 * 2. 处理边界条件
 *
```

```
 * 工程化考量：
 * 1. 输入验证：检查参数是否合法
 * 2. 边界条件：处理起点就是目的地的情况
 * 3. 性能优化：使用堆提高效率
 *
 * 相关题目：
 * 1. LeetCode 134. 加油站 - 环路加油问题
 * 2. LeetCode 45. 跳跃游戏 II - 最少跳跃次数
 * 3. LeetCode 55. 跳跃游戏 - 可达性判断
 * 4. 牛客网 NC48 跳跃游戏 - 与LeetCode 55相同
 * 5. LintCode 117. 跳跃游戏 II - 与LeetCode 45相同
 * 6. HackerRank - Jumping on the Clouds - 简化版跳跃游戏
 * 7. CodeChef - JUMP - 类似跳跃游戏的变种
 * 8. AtCoder ABC161D - Lunlun Number - BFS搜索相关
 * 9. Codeforces 1324C - Frog Jumps - 贪心跳跃问题
 * 10. POJ 1700 - Crossing River - 经典过河问题
 */
public class Code18_MinimumRefuelingStops {

    /**
     * 计算最低加油次数
     *
     * @param target 目的地距离
     * @param startFuel 初始油量
     * @param stations 加油站数组，每个元素是[位置, 油量]
     * @return 最低加油次数，无法到达返回-1
     */
    public static int minRefuelStops(int target, int startFuel, int[][] stations) {
        // 边界条件检查
        if (startFuel >= target) {
            return 0; // 初始油量足够到达目的地
        }

        if (stations == null || stations.length == 0) {
            return startFuel >= target ? 0 : -1; // 没有加油站，检查初始油量是否足够
        }

        // 最大堆，存储经过的加油站的油量
        PriorityQueue<Integer> maxHeap = new PriorityQueue<>(Collections.reverseOrder());

        int currentFuel = startFuel; // 当前油量
        int currentPosition = 0;    // 当前位置
        int refuelCount = 0;        // 加油次数
```

```
        int stationIndex = 0;        // 加油站索引

        while (currentPosition + currentFuel < target) {
            int nextPosition = currentPosition + currentFuel; // 当前能到达的最远位置

            // 将能到达的加油站加入最大堆
            while (stationIndex < stations.length && stations[stationIndex][0] <= nextPosition) {
                maxHeap.offer(stations[stationIndex][1]);
                stationIndex++;
            }

            // 如果没有加油站可加油，且无法到达目的地
            if (maxHeap.isEmpty()) {
                return -1;
            }

            // 选择油量最大的加油站加油
            int maxFuel = maxHeap.poll();
            currentFuel = currentFuel - (stations[stationIndex - 1][0] - currentPosition) +
maxFuel;
            currentPosition = stations[stationIndex - 1][0];
            refuelCount++;

            // 如果加油后能到达目的地
            if (currentPosition + currentFuel >= target) {
                return refuelCount;
            }
        }

        return refuelCount;
    }

    /**
     * 测试函数，验证算法正确性
     */
    public static void main(String[] args) {
        // 测试用例 1: 基本情况 - 需要加油
        int target1 = 100;
        int startFuel1 = 10;
        int[][] stations1 = {{10, 60}, {20, 30}, {30, 30}, {60, 40}};
        int result1 = minRefuelStops(target1, startFuel1, stations1);
        System.out.println("测试用例 1:");
        System.out.println("目的地距离: " + target1);
```

```
System.out.println("初始油量: " + startFuel1);
System.out.print("加油站: [");
for (int i = 0; i < stations1.length; i++) {
    System.out.print("[" + stations1[i][0] + "," + stations1[i][1] + "]");
    if (i < stations1.length - 1) System.out.print(", ");
}
System.out.println("]");
System.out.println("最低加油次数: " + result1);
System.out.println("期望输出: 2");
System.out.println();

// 测试用例 2: 基本情况 - 无法到达
int target2 = 100;
int startFuel2 = 10;
int[][] stations2 = {{20, 20}};
int result2 = minRefuelStops(target2, startFuel2, stations2);
System.out.println("测试用例 2:");
System.out.println("目的地距离: " + target2);
System.out.println("初始油量: " + startFuel2);
System.out.print("加油站: [");
for (int i = 0; i < stations2.length; i++) {
    System.out.print("[" + stations2[i][0] + "," + stations2[i][1] + "]");
    if (i < stations2.length - 1) System.out.print(", ");
}
System.out.println("]");
System.out.println("最低加油次数: " + result2);
System.out.println("期望输出: -1");
System.out.println();

// 测试用例 3: 边界情况 - 不需要加油
int target3 = 50;
int startFuel3 = 60;
int[][] stations3 = {{10, 20}, {20, 30}};
int result3 = minRefuelStops(target3, startFuel3, stations3);
System.out.println("测试用例 3:");
System.out.println("目的地距离: " + target3);
System.out.println("初始油量: " + startFuel3);
System.out.print("加油站: [");
for (int i = 0; i < stations3.length; i++) {
    System.out.print("[" + stations3[i][0] + "," + stations3[i][1] + "]");
    if (i < stations3.length - 1) System.out.print(", ");
}
System.out.println("]");
```

```
        System.out.println("最低加油次数: " + result3);
        System.out.println("期望输出: 0");
        System.out.println();

        // 测试用例 4: 边界情况 - 没有加油站
        int target4 = 50;
        int startFuel4 = 40;
        int[][] stations4 = {};
        int result4 = minRefuelStops(target4, startFuel4, stations4);
        System.out.println("测试用例 4:");
        System.out.println("目的地距离: " + target4);
        System.out.println("初始油量: " + startFuel4);
        System.out.println("加油站: []");
        System.out.println("最低加油次数: " + result4);
        System.out.println("期望输出: -1");
        System.out.println();

        // 测试用例 5: 复杂情况 - 多个加油站
        int target5 = 100;
        int startFuel5 = 20;
        int[][] stations5 = {{10, 10}, {20, 20}, {30, 30}, {40, 40}, {50, 50}};
        int result5 = minRefuelStops(target5, startFuel5, stations5);
        System.out.println("测试用例 5:");
        System.out.println("目的地距离: " + target5);
        System.out.println("初始油量: " + startFuel5);
        System.out.print("加油站: [");
        for (int i = 0; i < stations5.length; i++) {
            System.out.print("[" + stations5[i][0] + "," + stations5[i][1] + "]");
            if (i < stations5.length - 1) System.out.print(", ");
        }
        System.out.println("]");
        System.out.println("最低加油次数: " + result5);
        System.out.println("期望输出: 3");
    }
}

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

文件: Code18_MinimumRefuelingStops.py
==============================================
# 最低加油次数（Minimum Number of Refueling Stops）
# 题目来源：LeetCode 871
# 题目链接：https://leetcode.cn/problems/minimum-number-of-refueling-stops/
```

#
问题描述：
汽车从起点出发驶向目的地，该目的地距离起点 target 英里。
沿途有加油站，每个 station[i]代表一个加油站，位于距离起点 station[i][0]英里处，有 station[i][1]升汽油。
假设汽车油箱的容量是无限的，其中最初有 startFuel 升燃料。
它每行驶 1 英里就会用掉 1 升汽油。
当汽车到达加油站时，它可能停下来加油，将所有汽油从加油站转移到汽车中。
为了到达目的地，汽车所必要的最低加油次数是多少？如果无法到达目的地，则返回-1。
#
算法思路：
使用贪心策略，结合最大堆：
1. 遍历加油站，维护当前能到达的最远位置
2. 当油量不足以到达下一个加油站时，从经过的加油站中选择油量最大的进行加油
3. 使用最大堆来存储经过的加油站的油量
4. 每次加油后更新当前油量和加油次数
#
时间复杂度：O(n log n) - 堆操作的时间复杂度
空间复杂度：O(n) - 最大堆的空间
#
是否最优解：是。这是该问题的最优解法。
#
适用场景：
1. 路径规划问题
2. 资源调度问题
#
异常处理：
1. 处理无法到达目的地的情况
2. 处理边界条件
#
工程化考量：
1. 输入验证：检查参数是否合法
2. 边界条件：处理起点就是目的地的情况
3. 性能优化：使用堆提高效率
#
相关题目：
1. LeetCode 134. 加油站 - 环路加油问题
2. LeetCode 45. 跳跃游戏 II - 最少跳跃次数
3. LeetCode 55. 跳跃游戏 - 可达性判断
4. 牛客网 NC48 跳跃游戏 - 与 LeetCode 55 相同
5. LintCode 117. 跳跃游戏 II - 与 LeetCode 45 相同
6. HackerRank - Jumping on the Clouds - 简化版跳跃游戏
7. CodeChef - JUMP - 类似跳跃游戏的变种

```
# 8. AtCoder ABC161D - Lunlun Number - BFS搜索相关
# 9. Codeforces 1324C - Frog Jumps - 贪心跳跃问题
# 10. POJ 1700 - Crossing River - 经典过河问题

import heapq

class Solution:
    """
    计算最低加油次数

    Args:
        target: int - 目的地距离
        startFuel: int - 初始油量
        stations: List[List[int]] - 加油站数组，每个元素是[位置, 油量]

    Returns:
        int - 最低加油次数，无法到达返回-1
    """
    def minRefuelStops(self, target, startFuel, stations):
        # 边界条件检查
        if startFuel >= target:
            return 0  # 初始油量足够到达目的地

        if not stations:
            return 0 if startFuel >= target else -1  # 没有加油站，检查初始油量是否足够

        # 最大堆（使用最小堆的负数来模拟最大堆）
        max_heap = []

        current_fuel = startFuel  # 当前油量
        current_position = 0      # 当前位置
        refuel_count = 0          # 加油次数
        station_index = 0         # 加油站索引

        while current_position + current_fuel < target:
            next_position = current_position + current_fuel  # 当前能到达的最远位置

            # 将能到达的加油站加入最大堆
            while station_index < len(stations) and stations[station_index][0] <= next_position:
                heapq.heappush(max_heap, -stations[station_index][1])  # 使用负数模拟最大堆
                station_index += 1

            # 如果没有加油站可加油，且无法到达目的地
```

```
            if not max_heap:
                return -1

            # 选择油量最大的加油站加油
            max_fuel = -heapq.heappop(max_heap)
            current_fuel = current_fuel - (stations[station_index - 1][0] - current_position) +
max_fuel
            current_position = stations[station_index - 1][0]
            refuel_count += 1

            # 如果加油后能到达目的地
            if current_position + current_fuel >= target:
                return refuel_count

        return refuel_count


def main():
    solution = Solution()

    # 测试用例 1: 基本情况 - 需要加油
    target1 = 100
    start_fuel1 = 10
    stations1 = [[10, 60], [20, 30], [30, 30], [60, 40]]
    result1 = solution.minRefuelStops(target1, start_fuel1, stations1)
    print("测试用例 1:")
    print(f"目的地距离: {target1}")
    print(f"初始油量: {start_fuel1}")
    print(f"加油站: {stations1}")
    print(f"最低加油次数: {result1}")
    print("期望输出: 2")
    print()

    # 测试用例 2: 基本情况 - 无法到达
    target2 = 100
    start_fuel2 = 10
    stations2 = [[20, 20]]
    result2 = solution.minRefuelStops(target2, start_fuel2, stations2)
    print("测试用例 2:")
    print(f"目的地距离: {target2}")
    print(f"初始油量: {start_fuel2}")
    print(f"加油站: {stations2}")
    print(f"最低加油次数: {result2}")
```

```
    print("期望输出: -1")
    print()

    # 测试用例 3: 边界情况 - 不需要加油
    target3 = 50
    start_fuel3 = 60
    stations3 = [[10, 20], [20, 30]]
    result3 = solution.minRefuelStops(target3, start_fuel3, stations3)
    print("测试用例 3:")
    print(f"目的地距离: {target3}")
    print(f"初始油量: {start_fuel3}")
    print(f"加油站: {stations3}")
    print(f"最低加油次数: {result3}")
    print("期望输出: 0")
    print()

    # 测试用例 4: 边界情况 - 没有加油站
    target4 = 50
    start_fuel4 = 40
    stations4 = []
    result4 = solution.minRefuelStops(target4, start_fuel4, stations4)
    print("测试用例 4:")
    print(f"目的地距离: {target4}")
    print(f"初始油量: {start_fuel4}")
    print(f"加油站: {stations4}")
    print(f"最低加油次数: {result4}")
    print("期望输出: -1")
    print()

    # 测试用例 5: 复杂情况 - 多个加油站
    target5 = 100
    start_fuel5 = 20
    stations5 = [[10, 10], [20, 20], [30, 30], [40, 40], [50, 50]]
    result5 = solution.minRefuelStops(target5, start_fuel5, stations5)
    print("测试用例 5:")
    print(f"目的地距离: {target5}")
    print(f"初始油量: {start_fuel5}")
    print(f"加油站: {stations5}")
    print(f"最低加油次数: {result5}")
    print("期望输出: 3")


if __name__ == "__main__":
```

main()

==========

文件: Code19_TaskScheduler.cpp
==========
```
// 任务调度器（Task Scheduler）
// 题目来源：LeetCode 621
// 题目链接：https://leetcode.cn/problems/task-scheduler/
//
// 问题描述：
// 给定一个用字符数组表示的 CPU 需要执行的任务列表。其中包含使用大写的 A-Z 字母表示的 26 种不同种类的任务。
// 任务可以以任意顺序执行，并且每个任务都可以在 1 个单位时间内执行完。
// CPU 在任何一个单位时间内都可以执行一个任务，或者在待命状态。
// 然而，两个相同种类的任务之间必须有长度为 n 的冷却时间，因此至少有连续 n 个单位时间内 CPU 在执行不同的任务，或者在待命状态。
// 你需要计算完成所有任务所需要的最短时间。
//
// 算法思路：
// 使用贪心策略，基于任务频率：
// 1. 统计每个任务的频率
// 2. 找到出现次数最多的任务，假设出现次数为 maxCount
// 3. 计算至少需要的时间：(maxCount - 1) * (n + 1) + 出现次数为 maxCount 的任务个数
// 4. 如果计算结果小于任务总数，说明可以安排得更紧凑，返回任务总数
//
// 时间复杂度：O(n) - 统计频率的时间复杂度
// 空间复杂度：O(1) - 只需要常数空间存储频率
//
// 是否最优解：是。这是该问题的最优解法。
//
// 适用场景：
// 1. 任务调度问题
// 2. 资源分配问题
//
// 异常处理：
// 1. 处理空数组情况
// 2. 处理 n=0 的情况
//
// 工程化考量：
// 1. 输入验证：检查参数是否合法
// 2. 边界条件：处理单任务情况
// 3. 性能优化：使用数组统计频率
```

```
//
// 相关题目：
// 1. LeetCode 767. 重构字符串 - 类似的任务安排问题
// 2. LeetCode 358. 重排字符串 k 距离 apart - 更一般的任务调度
// 3. LeetCode 253. 会议室 II - 区间调度问题
// 4. 牛客网 NC140 排序 - 各种排序算法实现
// 5. LintCode 945. 任务计划 - 类似问题
// 6. HackerRank - Task Scheduling - 任务调度问题
// 7. CodeChef - TASKS - 任务分配问题
// 8. AtCoder ABC131D - Megalomania - 任务截止时间问题
// 9. Codeforces 1324C - Frog Jumps - 贪心跳跃问题
// 10. POJ 1700 - Crossing River - 经典过河问题

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

/**
 * 计算完成任务的最短时间
 *
 * @param tasks 任务数组
 * @param n 冷却时间
 * @return 最短完成时间
 */
int leastInterval(vector<char>& tasks, int n) {
    // 边界条件检查
    if (tasks.empty()) {
        return 0;
    }

    if (n == 0) {
        return tasks.size(); // 没有冷却时间，直接按顺序执行
    }

    // 统计任务频率
    vector<int> frequency(26, 0);
    for (char task : tasks) {
        frequency[task - 'A']++;
    }

    // 找到最大频率
    int maxFrequency = 0;
```

```
    for (int freq : frequency) {
        maxFrequency = max(maxFrequency, freq);
    }

    // 统计出现最大频率的任务个数
    int maxCount = 0;
    for (int freq : frequency) {
        if (freq == maxFrequency) {
            maxCount++;
        }
    }

    // 计算最短时间
    int result = (maxFrequency - 1) * (n + 1) + maxCount;

    // 如果计算结果小于任务总数，说明可以安排得更紧凑
    return max(result, (int)tasks.size());
}

// 测试函数
int main() {
    // 测试用例 1: 基本情况
    vector<char> tasks1 = {'A', 'A', 'A', 'B', 'B', 'B'};
    int n1 = 2;
    int result1 = leastInterval(tasks1, n1);

    // 测试用例 2: 简单情况
    vector<char> tasks2 = {'A', 'A', 'A', 'B', 'B', 'B'};
    int n2 = 0;
    int result2 = leastInterval(tasks2, n2);

    // 测试用例 3: 复杂情况
    vector<char> tasks3 = {'A', 'A', 'A', 'A', 'A', 'A', 'B', 'C', 'D', 'E', 'F', 'G'};
    int n3 = 2;
    int result3 = leastInterval(tasks3, n3);

    // 测试用例 4: 边界情况 - 单任务
    vector<char> tasks4 = {'A'};
    int n4 = 3;
    int result4 = leastInterval(tasks4, n4);

    // 测试用例 5: 边界情况 - 空数组
    vector<char> tasks5 = {};
```

```
    int n5 = 5;
    int result5 = leastInterval(tasks5, n5);

    // 测试用例 6: 复杂情况 - 多个相同频率
    vector<char> tasks6 = {'A', 'A', 'A', 'B', 'B', 'B', 'C', 'C', 'C', 'D', 'D', 'E'};
    int n6 = 2;
    int result6 = leastInterval(tasks6, n6);

    return 0;
}
```

==========

文件: Code19_TaskScheduler.java

==========

```
package class093;

import java.util.*;

/**
 * 任务调度器（Task Scheduler）
 * 题目来源：LeetCode 621
 * 题目链接：https://leetcode.cn/problems/task-scheduler/
 *
 * 问题描述：
 * 给定一个用字符数组表示的 CPU 需要执行的任务列表。其中包含使用大写的 A-Z 字母表示的 26 种不同种类的任务。
 * 任务可以以任意顺序执行，并且每个任务都可以在 1 个单位时间内执行完。
 * CPU 在任何一个单位时间内都可以执行一个任务，或者在待命状态。
 * 然而，两个相同种类的任务之间必须有长度为 n 的冷却时间，因此至少有连续 n 个单位时间内 CPU 在执行不同的任务，或者在待命状态。
 * 你需要计算完成所有任务所需要的最短时间。
 *
 * 算法思路：
 * 使用贪心策略，基于任务频率：
 * 1. 统计每个任务的频率
 * 2. 找到出现次数最多的任务，假设出现次数为 maxCount
 * 3. 计算至少需要的时间：(maxCount - 1) * (n + 1) + 出现次数为 maxCount 的任务个数
 * 4. 如果计算结果小于任务总数，说明可以安排得更紧凑，返回任务总数
 *
 * 时间复杂度：O(n) - 统计频率的时间复杂度
 * 空间复杂度：O(1) - 只需要常数空间存储频率
 *
```

 * 是否最优解：是。这是该问题的最优解法。
 *
 * 适用场景：
 * 1. 任务调度问题
 * 2. 资源分配问题
 *
 * 异常处理：
 * 1. 处理空数组情况
 * 2. 处理 n=0 的情况
 *
 * 工程化考量：
 * 1. 输入验证：检查参数是否合法
 * 2. 边界条件：处理单任务情况
 * 3. 性能优化：使用数组统计频率
 *
 * 相关题目：
 * 1. LeetCode 767. 重构字符串 - 类似的任务安排问题
 * 2. LeetCode 358. 重排字符串 k 距离 apart - 更一般的任务调度
 * 3. LeetCode 253. 会议室 II - 区间调度问题
 * 4. 牛客网 NC140 排序 - 各种排序算法实现
 * 5. LintCode 945. 任务计划 - 类似问题
 * 6. HackerRank - Task Scheduling - 任务调度问题
 * 7. CodeChef - TASKS - 任务分配问题
 * 8. AtCoder ABC131D - Megalomania - 任务截止时间问题
 * 9. Codeforces 1324C - Frog Jumps - 贪心跳跃问题
 * 10. POJ 1700 - Crossing River - 经典过河问题
 */
public class Code19_TaskScheduler {

    /**
     * 计算完成任务的最短时间
     *
     * @param tasks 任务数组
     * @param n 冷却时间
     * @return 最短完成时间
     */
    public static int leastInterval(char[] tasks, int n) {
        // 边界条件检查
        if (tasks == null || tasks.length == 0) {
            return 0;
        }

        if (n == 0) {

```
        return tasks.length; // 没有冷却时间，直接按顺序执行
    }

    // 统计任务频率
    int[] frequency = new int[26];
    for (char task : tasks) {
        frequency[task - 'A']++;
    }

    // 找到最大频率
    int maxFrequency = 0;
    for (int freq : frequency) {
        maxFrequency = Math.max(maxFrequency, freq);
    }

    // 统计出现最大频率的任务个数
    int maxCount = 0;
    for (int freq : frequency) {
        if (freq == maxFrequency) {
            maxCount++;
        }
    }

    // 计算最短时间
    int result = (maxFrequency - 1) * (n + 1) + maxCount;

    // 如果计算结果小于任务总数，说明可以安排得更紧凑
    return Math.max(result, tasks.length);
}

/**
 * 测试函数，验证算法正确性
 */
public static void main(String[] args) {
    // 测试用例 1: 基本情况
    char[] tasks1 = {'A', 'A', 'A', 'B', 'B', 'B'};
    int n1 = 2;
    int result1 = leastInterval(tasks1, n1);
    System.out.println("测试用例 1:");
    System.out.println("任务数组: " + Arrays.toString(tasks1));
    System.out.println("冷却时间: " + n1);
    System.out.println("最短完成时间: " + result1);
    System.out.println("期望输出: 8");
```

```
System.out.println();

// 测试用例 2: 简单情况
char[] tasks2 = {'A', 'A', 'A', 'B', 'B', 'B'};
int n2 = 0;
int result2 = leastInterval(tasks2, n2);
System.out.println("测试用例 2:");
System.out.println("任务数组: " + Arrays.toString(tasks2));
System.out.println("冷却时间: " + n2);
System.out.println("最短完成时间: " + result2);
System.out.println("期望输出: 6");
System.out.println();

// 测试用例 3: 复杂情况
char[] tasks3 = {'A', 'A', 'A', 'A', 'A', 'A', 'B', 'C', 'D', 'E', 'F', 'G'};
int n3 = 2;
int result3 = leastInterval(tasks3, n3);
System.out.println("测试用例 3:");
System.out.println("任务数组: " + Arrays.toString(tasks3));
System.out.println("冷却时间: " + n3);
System.out.println("最短完成时间: " + result3);
System.out.println("期望输出: 16");
System.out.println();

// 测试用例 4: 边界情况 - 单任务
char[] tasks4 = {'A'};
int n4 = 3;
int result4 = leastInterval(tasks4, n4);
System.out.println("测试用例 4:");
System.out.println("任务数组: " + Arrays.toString(tasks4));
System.out.println("冷却时间: " + n4);
System.out.println("最短完成时间: " + result4);
System.out.println("期望输出: 1");
System.out.println();

// 测试用例 5: 边界情况 - 空数组
char[] tasks5 = {};
int n5 = 5;
int result5 = leastInterval(tasks5, n5);
System.out.println("测试用例 5:");
System.out.println("任务数组: " + Arrays.toString(tasks5));
System.out.println("冷却时间: " + n5);
System.out.println("最短完成时间: " + result5);
```

```
        System.out.println("期望输出: 0");
        System.out.println();

        // 测试用例 6: 复杂情况 - 多个相同频率
        char[] tasks6 = {'A', 'A', 'A', 'B', 'B', 'B', 'C', 'C', 'C', 'D', 'D', 'E'};
        int n6 = 2;
        int result6 = leastInterval(tasks6, n6);
        System.out.println("测试用例 6:");
        System.out.println("任务数组: " + Arrays.toString(tasks6));
        System.out.println("冷却时间: " + n6);
        System.out.println("最短完成时间: " + result6);
        System.out.println("期望输出: 12");
    }
}
```

==========

文件: Code19_TaskScheduler.py

==========

```
# 任务调度器（Task Scheduler）
# 题目来源：LeetCode 621
# 题目链接：https://leetcode.cn/problems/task-scheduler/
#
# 问题描述：
# 给定一个用字符数组表示的 CPU 需要执行的任务列表。其中包含使用大写的 A-Z 字母表示的 26 种不同种类的任务。
# 任务可以以任意顺序执行，并且每个任务都可以在 1 个单位时间内执行完。
# CPU 在任何一个单位时间内都可以执行一个任务，或者在待命状态。
# 然而，两个相同种类的任务之间必须有长度为 n 的冷却时间，因此至少有连续 n 个单位时间内 CPU 在执行不同的任务，或者在待命状态。
# 你需要计算完成所有任务所需要的最短时间。
#
# 算法思路：
# 使用贪心策略，基于任务频率：
# 1. 统计每个任务的频率
# 2. 找到出现次数最多的任务，假设出现次数为 maxCount
# 3. 计算至少需要的时间：(maxCount - 1) * (n + 1) + 出现次数为 maxCount 的任务个数
# 4. 如果计算结果小于任务总数，说明可以安排得更紧凑，返回任务总数
#
# 时间复杂度：O(n) - 统计频率的时间复杂度
# 空间复杂度：O(1) - 只需要常数空间存储频率
#
# 是否最优解：是。这是该问题的最优解法。
```

```
#
# 适用场景：
# 1. 任务调度问题
# 2. 资源分配问题
#
# 异常处理：
# 1. 处理空数组情况
# 2. 处理 n=0 的情况
#
# 工程化考量：
# 1. 输入验证：检查参数是否合法
# 2. 边界条件：处理单任务情况
# 3. 性能优化：使用数组统计频率
#
# 相关题目：
# 1. LeetCode 767. 重构字符串 - 类似的任务安排问题
# 2. LeetCode 358. 重排字符串 k 距离 apart - 更一般的任务调度
# 3. LeetCode 253. 会议室 II - 区间调度问题
# 4. 牛客网 NC140 排序 - 各种排序算法实现
# 5. LintCode 945. 任务计划 - 类似问题
# 6. HackerRank - Task Scheduling - 任务调度问题
# 7. CodeChef - TASKS - 任务分配问题
# 8. AtCoder ABC131D - Megalomania - 任务截止时间问题
# 9. Codeforces 1324C - Frog Jumps - 贪心跳跃问题
# 10. POJ 1700 - Crossing River - 经典过河问题

from typing import List

class Solution:
    """
    计算完成任务的最短时间

    Args:
        tasks: List[str] - 任务数组
        n: int - 冷却时间

    Returns:
        int - 最短完成时间
    """
    def leastInterval(self, tasks: List[str], n: int) -> int:
        # 边界条件检查
        if not tasks:
            return 0
```

```
        if n == 0:
            return len(tasks)  # 没有冷却时间，直接按顺序执行

        # 统计任务频率
        frequency = [0] * 26
        for task in tasks:
            frequency[ord(task) - ord('A')] += 1

        # 找到最大频率
        max_frequency = max(frequency)

        # 统计出现最大频率的任务个数
        max_count = frequency.count(max_frequency)

        # 计算最短时间
        result = (max_frequency - 1) * (n + 1) + max_count

        # 如果计算结果小于任务总数，说明可以安排得更紧凑
        return max(result, len(tasks))


def main():
    solution = Solution()

    # 测试用例 1: 基本情况
    tasks1 = ['A', 'A', 'A', 'B', 'B', 'B']
    n1 = 2
    result1 = solution.leastInterval(tasks1, n1)
    print("测试用例 1:")
    print(f"任务数组: {tasks1}")
    print(f"冷却时间: {n1}")
    print(f"最短完成时间: {result1}")
    print("期望输出: 8")
    print()

    # 测试用例 2: 简单情况
    tasks2 = ['A', 'A', 'A', 'B', 'B', 'B']
    n2 = 0
    result2 = solution.leastInterval(tasks2, n2)
    print("测试用例 2:")
    print(f"任务数组: {tasks2}")
    print(f"冷却时间: {n2}")
```

```
print(f"最短完成时间: {result2}")
print("期望输出: 6")
print()

# 测试用例 3: 复杂情况
tasks3 = ['A', 'A', 'A', 'A', 'A', 'A', 'B', 'C', 'D', 'E', 'F', 'G']
n3 = 2
result3 = solution.leastInterval(tasks3, n3)
print("测试用例 3:")
print(f"任务数组: {tasks3}")
print(f"冷却时间: {n3}")
print(f"最短完成时间: {result3}")
print("期望输出: 16")
print()

# 测试用例 4: 边界情况 - 单任务
tasks4 = ['A']
n4 = 3
result4 = solution.leastInterval(tasks4, n4)
print("测试用例 4:")
print(f"任务数组: {tasks4}")
print(f"冷却时间: {n4}")
print(f"最短完成时间: {result4}")
print("期望输出: 1")
print()

# 测试用例 5: 边界情况 - 空数组
tasks5 = []
n5 = 5
result5 = solution.leastInterval(tasks5, n5)
print("测试用例 5:")
print(f"任务数组: {tasks5}")
print(f"冷却时间: {n5}")
print(f"最短完成时间: {result5}")
print("期望输出: 0")
print()

# 测试用例 6: 复杂情况 - 多个相同频率
tasks6 = ['A', 'A', 'A', 'B', 'B', 'B', 'C', 'C', 'C', 'D', 'D', 'E']
n6 = 2
result6 = solution.leastInterval(tasks6, n6)
print("测试用例 6:")
print(f"任务数组: {tasks6}")
```

```
    print(f"冷却时间: {n6}")
    print(f"最短完成时间: {result6}")
    print("期望输出: 12")


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

==========

文件: Code20_RemoveKDigits.cpp

==========

```
// 移掉 K 位数字（Remove K Digits）
// 题目来源：LeetCode 402
// 题目链接：https://leetcode.cn/problems/remove-k-digits/
//
// 问题描述：
// 给定一个以字符串表示的非负整数 num，移除这个数中的 k 位数字，使得剩下的数字最小。
//
// 算法思路：
// 使用贪心策略，结合单调栈：
// 1. 遍历数字字符串，维护一个单调递增栈
// 2. 当遇到比栈顶小的数字时，弹出栈顶元素（移除数字）
// 3. 直到移除 k 个数字或栈为空
// 4. 如果遍历完成后还没有移除 k 个数字，从栈顶继续移除
// 5. 处理前导零和空栈情况
//
// 时间复杂度：O(n) - 每个元素最多入栈出栈一次
// 空间复杂度：O(n) - 栈的空间
//
// 是否最优解：是。这是该问题的最优解法。
//
// 适用场景：
// 1. 数字最小化问题
// 2. 字符串处理问题
//
// 异常处理：
// 1. 处理 k 大于等于数字长度的情况
// 2. 处理前导零情况
// 3. 处理空字符串情况
//
// 工程化考量：
// 1. 输入验证：检查参数是否合法
```

```
// 2. 边界条件：处理各种边界情况
// 3. 性能优化：使用字符串构建提高效率
//
// 相关题目：
// 1. LeetCode 321. 拼接最大数 - 类似数字拼接问题
// 2. LeetCode 316. 去除重复字母 - 类似字符处理问题
// 3. LeetCode 738. 单调递增的数字 - 数字单调性问题
// 4. 牛客网 NC140 排序 - 各种排序算法实现
// 5. LintCode 1254. 移除 K 位数字 - 与本题相同
// 6. HackerRank - Largest Permutation - 最大排列问题
// 7. CodeChef - DIGITREM - 数字移除问题
// 8. AtCoder ABC155D - Pairs - 数字配对问题
// 9. Codeforces 1324C - Frog Jumps - 贪心跳跃问题
// 10. POJ 1700 - Crossing River - 经典过河问题

#include <string>
#include <stack>
#include <algorithm>
using namespace std;

/**
 * 移除 k 位数字使得剩下的数字最小
 *
 * @param num 数字字符串
 * @param k 要移除的数字个数
 * @return 移除后的最小数字字符串
 */
string removeKdigits(string num, int k) {
    // 边界条件检查
    if (num.empty() || k >= num.length()) {
        return "0";
    }

    if (k == 0) {
        return num; // 不需要移除任何数字
    }

    stack<char> stack;

    for (int i = 0; i < num.length(); i++) {
        char currentChar = num[i];

        // 当栈不为空，且 k>0，且当前数字小于栈顶数字时，弹出栈顶
```

```
        while (!stack.empty() && k > 0 && currentChar < stack.top()) {
            stack.pop();
            k--;
        }

        // 将当前数字压入栈中
        stack.push(currentChar);
    }

    // 如果还有 k 个数字需要移除，从栈顶移除（因为栈是单调递增的）
    while (k > 0 && !stack.empty()) {
        stack.pop();
        k--;
    }

    // 构建结果字符串
    string result;
    while (!stack.empty()) {
        result += stack.top();
        stack.pop();
    }
    reverse(result.begin(), result.end());

    // 处理前导零
    int startIndex = 0;
    while (startIndex < result.length() && result[startIndex] == '0') {
        startIndex++;
    }

    // 如果所有数字都是 0，返回"0"
    if (startIndex == result.length()) {
        return "0";
    }

    return result.substr(startIndex);
}

// 测试函数
int main() {
    // 测试用例 1: 基本情况
    string num1 = "1432219";
    int k1 = 3;
    string result1 = removeKdigits(num1, k1);
```

```
    // 测试用例 2: 简单情况
    string num2 = "10200";
    int k2 = 1;
    string result2 = removeKdigits(num2, k2);

    // 测试用例 3: 复杂情况
    string num3 = "10";
    int k3 = 2;
    string result3 = removeKdigits(num3, k3);

    // 测试用例 4: 边界情况 - 移除所有数字
    string num4 = "12345";
    int k4 = 5;
    string result4 = removeKdigits(num4, k4);

    // 测试用例 5: 边界情况 - 不移除数字
    string num5 = "12345";
    int k5 = 0;
    string result5 = removeKdigits(num5, k5);

    // 测试用例 6: 复杂情况 - 前导零处理
    string num6 = "100200";
    int k6 = 1;
    string result6 = removeKdigits(num6, k6);

    return 0;
}

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

文件: Code20_RemoveKDigits.java
================================================
package class093;

import java.util.Stack;

/**
 * 移掉 K 位数字（Remove K Digits）
 * 题目来源：LeetCode 402
 * 题目链接：https://leetcode.cn/problems/remove-k-digits/
 *
 * 问题描述：
```

```
 * 给定一个以字符串表示的非负整数 num，移除这个数中的 k 位数字，使得剩下的数字最小。
 *
 * 算法思路：
 * 使用贪心策略，结合单调栈：
 * 1. 遍历数字字符串，维护一个单调递增栈
 * 2. 当遇到比栈顶小的数字时，弹出栈顶元素（移除数字）
 * 3. 直到移除 k 个数字或栈为空
 * 4. 如果遍历完成后还没有移除 k 个数字，从栈顶继续移除
 * 5. 处理前导零和空栈情况
 *
 * 时间复杂度：O(n) - 每个元素最多入栈出栈一次
 * 空间复杂度：O(n) - 栈的空间
 *
 * 是否最优解：是。这是该问题的最优解法。
 *
 * 适用场景：
 * 1. 数字最小化问题
 * 2. 字符串处理问题
 *
 * 异常处理：
 * 1. 处理 k 大于等于数字长度的情况
 * 2. 处理前导零情况
 * 3. 处理空字符串情况
 *
 * 工程化考量：
 * 1. 输入验证：检查参数是否合法
 * 2. 边界条件：处理各种边界情况
 * 3. 性能优化：使用 StringBuilder 提高效率
 *
 * 相关题目：
 * 1. LeetCode 321. 拼接最大数 - 类似数字拼接问题
 * 2. LeetCode 316. 去除重复字母 - 类似字符处理问题
 * 3. LeetCode 738. 单调递增的数字 - 数字单调性问题
 * 4. 牛客网 NC140 排序 - 各种排序算法实现
 * 5. LintCode 1254. 移除 K 位数字 - 与本题相同
 * 6. HackerRank - Largest Permutation - 最大排列问题
 * 7. CodeChef - DIGITREM - 数字移除问题
 * 8. AtCoder ABC155D - Pairs - 数字配对问题
 * 9. Codeforces 1324C - Frog Jumps - 贪心跳跃问题
 * 10. POJ 1700 - Crossing River - 经典过河问题
 */
public class Code20_RemoveKDigits {
```

```
/**
 * 移除 k 位数字使得剩下的数字最小
 *
 * @param num 数字字符串
 * @param k 要移除的数字个数
 * @return 移除后的最小数字字符串
 */
public static String removeKdigits(String num, int k) {
    // 边界条件检查
    if (num == null || num.length() == 0 || k >= num.length()) {
        return "0";
    }

    if (k == 0) {
        return num; // 不需要移除任何数字
    }

    Stack<Character> stack = new Stack<>();

    for (int i = 0; i < num.length(); i++) {
        char currentChar = num.charAt(i);

        // 当栈不为空，且 k>0，且当前数字小于栈顶数字时，弹出栈顶
        while (!stack.isEmpty() && k > 0 && currentChar < stack.peek()) {
            stack.pop();
            k--;
        }

        // 将当前数字压入栈中
        stack.push(currentChar);
    }

    // 如果还有 k 个数字需要移除，从栈顶移除（因为栈是单调递增的）
    while (k > 0 && !stack.isEmpty()) {
        stack.pop();
        k--;
    }

    // 构建结果字符串
    StringBuilder result = new StringBuilder();
    while (!stack.isEmpty()) {
        result.append(stack.pop());
    }
```

```
        result.reverse();

        // 处理前导零
        int startIndex = 0;
        while (startIndex < result.length() && result.charAt(startIndex) == '0') {
            startIndex++;
        }

        // 如果所有数字都是0，返回"0"
        if (startIndex == result.length()) {
            return "0";
        }

        return result.substring(startIndex);
    }

    /**
     * 测试函数，验证算法正确性
     */
    public static void main(String[] args) {
        // 测试用例1: 基本情况
        String num1 = "1432219";
        int k1 = 3;
        String result1 = removeKdigits(num1, k1);
        System.out.println("测试用例1:");
        System.out.println("输入数字: " + num1);
        System.out.println("移除位数: " + k1);
        System.out.println("最小结果: " + result1);
        System.out.println("期望输出: 1219");
        System.out.println();

        // 测试用例2: 简单情况
        String num2 = "10200";
        int k2 = 1;
        String result2 = removeKdigits(num2, k2);
        System.out.println("测试用例2:");
        System.out.println("输入数字: " + num2);
        System.out.println("移除位数: " + k2);
        System.out.println("最小结果: " + result2);
        System.out.println("期望输出: 200");
        System.out.println();

        // 测试用例3: 复杂情况
```

```
        String num3 = "10";
        int k3 = 2;
        String result3 = removeKdigits(num3, k3);
        System.out.println("测试用例 3:");
        System.out.println("输入数字: " + num3);
        System.out.println("移除位数: " + k3);
        System.out.println("最小结果: " + result3);
        System.out.println("期望输出: 0");
        System.out.println();

        // 测试用例 4: 边界情况 - 移除所有数字
        String num4 = "12345";
        int k4 = 5;
        String result4 = removeKdigits(num4, k4);
        System.out.println("测试用例 4:");
        System.out.println("输入数字: " + num4);
        System.out.println("移除位数: " + k4);
        System.out.println("最小结果: " + result4);
        System.out.println("期望输出: 0");
        System.out.println();

        // 测试用例 5: 边界情况 - 不移除数字
        String num5 = "12345";
        int k5 = 0;
        String result5 = removeKdigits(num5, k5);
        System.out.println("测试用例 5:");
        System.out.println("输入数字: " + num5);
        System.out.println("移除位数: " + k5);
        System.out.println("最小结果: " + result5);
        System.out.println("期望输出: 12345");
        System.out.println();

        // 测试用例 6: 复杂情况 - 前导零处理
        String num6 = "100200";
        int k6 = 1;
        String result6 = removeKdigits(num6, k6);
        System.out.println("测试用例 6:");
        System.out.println("输入数字: " + num6);
        System.out.println("移除位数: " + k6);
        System.out.println("最小结果: " + result6);
        System.out.println("期望输出: 200");
    }
}
```

==========

文件: Code20_RemoveKDigits.py
==========
```
# 移掉K位数字（Remove K Digits）
# 题目来源：LeetCode 402
# 题目链接：https://leetcode.cn/problems/remove-k-digits/
#
# 问题描述：
# 给定一个以字符串表示的非负整数num，移除这个数中的k位数字，使得剩下的数字最小。
#
# 算法思路：
# 使用贪心策略，结合单调栈：
# 1. 遍历数字字符串，维护一个单调递增栈
# 2. 当遇到比栈顶小的数字时，弹出栈顶元素（移除数字）
# 3. 直到移除k个数字或栈为空
# 4. 如果遍历完成后还没有移除k个数字，从栈顶继续移除
# 5. 处理前导零和空栈情况
#
# 时间复杂度：O(n) - 每个元素最多入栈出栈一次
# 空间复杂度：O(n) - 栈的空间
#
# 是否最优解：是。这是该问题的最优解法。
#
# 适用场景：
# 1. 数字最小化问题
# 2. 字符串处理问题
#
# 异常处理：
# 1. 处理k大于等于数字长度的情况
# 2. 处理前导零情况
# 3. 处理空字符串情况
#
# 工程化考量：
# 1. 输入验证：检查参数是否合法
# 2. 边界条件：处理各种边界情况
# 3. 性能优化：使用列表模拟栈提高效率
#
# 相关题目：
# 1. LeetCode 321. 拼接最大数 - 类似数字拼接问题
# 2. LeetCode 316. 去除重复字母 - 类似字符处理问题
# 3. LeetCode 738. 单调递增的数字 - 数字单调性问题
```

```
# 4. 牛客网 NC140 排序 - 各种排序算法实现
# 5. LintCode 1254. 移除K位数字 - 与本题相同
# 6. HackerRank - Largest Permutation - 最大排列问题
# 7. CodeChef - DIGITREM - 数字移除问题
# 8. AtCoder ABC155D - Pairs - 数字配对问题
# 9. Codeforces 1324C - Frog Jumps - 贪心跳跃问题
# 10. POJ 1700 - Crossing River - 经典过河问题

class Solution:
    """
    移除k位数字使得剩下的数字最小

    Args:
        num: str - 数字字符串
        k: int - 要移除的数字个数

    Returns:
        str - 移除后的最小数字字符串
    """
    def removeKdigits(self, num: str, k: int) -> str:
        # 边界条件检查
        if not num or k >= len(num):
            return "0"

        if k == 0:
            return num  # 不需要移除任何数字

        # 使用列表模拟栈
        stack = []

        for char in num:
            # 当栈不为空，且k>0，且当前数字小于栈顶数字时，弹出栈顶
            while stack and k > 0 and char < stack[-1]:
                stack.pop()
                k -= 1

            # 将当前数字压入栈中
            stack.append(char)

        # 如果还有k个数字需要移除，从栈顶移除（因为栈是单调递增的）
        if k > 0:
            stack = stack[:-k]
```

```
        # 构建结果字符串
        result = ''.join(stack)

        # 处理前导零
        result = result.lstrip('0')

        # 如果所有数字都是0，返回"0"
        if not result:
            return "0"

        return result


def main():
    solution = Solution()

    # 测试用例1: 基本情况
    num1 = "1432219"
    k1 = 3
    result1 = solution.removeKdigits(num1, k1)
    print("测试用例1:")
    print(f"输入数字: {num1}")
    print(f"移除位数: {k1}")
    print(f"最小结果: {result1}")
    print("期望输出: 1219")
    print()

    # 测试用例2: 简单情况
    num2 = "10200"
    k2 = 1
    result2 = solution.removeKdigits(num2, k2)
    print("测试用例2:")
    print(f"输入数字: {num2}")
    print(f"移除位数: {k2}")
    print(f"最小结果: {result2}")
    print("期望输出: 200")
    print()

    # 测试用例3: 复杂情况
    num3 = "10"
    k3 = 2
    result3 = solution.removeKdigits(num3, k3)
    print("测试用例3:")
```

```
    print(f"输入数字: {num3}")
    print(f"移除位数: {k3}")
    print(f"最小结果: {result3}")
    print("期望输出: 0")
    print()

    # 测试用例 4: 边界情况 - 移除所有数字
    num4 = "12345"
    k4 = 5
    result4 = solution.removeKdigits(num4, k4)
    print("测试用例 4:")
    print(f"输入数字: {num4}")
    print(f"移除位数: {k4}")
    print(f"最小结果: {result4}")
    print("期望输出: 0")
    print()

    # 测试用例 5: 边界情况 - 不移除数字
    num5 = "12345"
    k5 = 0
    result5 = solution.removeKdigits(num5, k5)
    print("测试用例 5:")
    print(f"输入数字: {num5}")
    print(f"移除位数: {k5}")
    print(f"最小结果: {result5}")
    print("期望输出: 12345")
    print()

    # 测试用例 6: 复杂情况 - 前导零处理
    num6 = "100200"
    k6 = 1
    result6 = solution.removeKdigits(num6, k6)
    print("测试用例 6:")
    print(f"输入数字: {num6}")
    print(f"移除位数: {k6}")
    print(f"最小结果: {result6}")
    print("期望输出: 200")


if __name__ == "__main__":
    main()

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