122

给定一个数组，它的第 i 个元素是一支给定股票第 i 天的价格。

设计一个算法来计算你所能获取的最大利润。你可以尽可能地完成更多的交易（多次买卖一支股票）。

注意：你不能同时参与多笔交易（你必须在再次购买前出售掉之前的股票）。

示例 1:

输入: [7,1,5,3,6,4]

输出: 7

解释: 在第 2 天（股票价格 = 1）的时候买入，在第 3 天（股票价格 = 5）的时候卖出, 这笔交易所能获得利润 = 5-1 = 4 。

  随后，在第 4 天（股票价格 = 3）的时候买入，在第 5 天（股票价格 = 6）的时候卖出, 这笔交易所能获得利润 = 6-3 = 3 。

题解：只要后一天比前一天高，就记录其差价，可以理解为当天卖出去之后又买回来，一天一天的交易。

class Solution:

    def maxProfit(self, prices: List[int]) -> int:

        res = 0

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

            x = max(0, prices[i] - prices[i-1])

            res += x

        return res

55

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。

示例 1:

输入: [2,3,1,1,4]

输出: true

解释: 我们可以先跳 1 步，从位置 0 到达 位置 1, 然后再从位置 1 跳 3 步到达最后一个位置。

示例 2:

输入: [3,2,1,0,4]

输出: false

解释: 无论怎样，你总会到达索引为 3 的位置。但该位置的最大跳跃长度是 0 ， 所以你永远不可能到达最后一个位置。

题解：维护一个最大的能跳到的索引值，如果这个值超过了数组长度，则表示能跳到最后一个，如果到达某一个点，这个点只能往前跳0个并且当前最远距离<=该索引位置，所以就不能继续往后跳，失败。

class Solution:

    def canJump(self, nums: List[int]) -> bool:

        if len(nums) == 0 or len(nums) == 1:

            return True

        maxN = 0

        # r = [0 for \_ in range(len(nums)+1)]

        for i in range(len(nums)):

            maxN = max(maxN, i+nums[i]+1)

            if maxN >= len(nums):

                return True

            if nums[i] == 0 and i+1 >= maxN:

                return False

        return True

题解2：使用数组记录能达到的所有点，标为true，然后遍历，只要有一个点为false，就是不能达到。

class Solution {

public:

bool canJump(vector<int>& nums) {

const int len = nums.size();

if (len == 1) return true;

bool ans[len] = {false};

for (int i = 0;i < len-1;i++){

for (int j = 1;j <= nums[i];j++){

if (i+j < len)

ans[i+j] = true;

}

}

for (int i = 1;i < len;i++){

if (ans[i] == false) return false;

}

return true;

}

};

45

给定一个非负整数数组，你最初位于数组的第一个位置。

数组中的每个元素代表你在该位置可以跳跃的最大长度。

你的目标是使用最少的跳跃次数到达数组的最后一个位置。

示例:

输入: [2,3,1,1,4]

输出: 2

解释: 跳到最后一个位置的最小跳跃数是 2。

  从下标为 0 跳到下标为 1 的位置，跳 1 步，然后跳 3 步到达数组的最后一个位置。

说明:

假设你总是可以到达数组的最后一个位置。

题解：题目保证总能到达最后一个位置，所以只需要在每次跳跃中找到该次跳跃中所能达到的最远距离。假设当前位置为cur\_pos，那么就是找到cur\_pos+1到cur\_pos+nums[cur\_pos]中的最大值，就是下次跳跃该达到的点。如果本次跳跃已经可以达到最后一个位置，返回结果。

class Solution:

    def jump(self, nums: List[int]) -> int:

        step = 0

        l = len(nums)

        if l == 1 or l == 0:

            return 0

        cur\_pos = 0

        while cur\_pos < l-1:

            step += 1

            tmp = cur\_pos + nums[cur\_pos]

            if cur\_pos + nums[cur\_pos] >= l-1:

                return step

            maxN = 0

            for i in range(cur\_pos+1, tmp+1):

                if i + nums[i] > maxN:

                    maxN = i + nums[i]

                    cur\_pos = i

        return step

134

在一条环路上有 N 个加油站，其中第 i 个加油站有汽油 gas[i] 升。

你有一辆油箱容量无限的的汽车，从第 i 个加油站开往第 i+1 个加油站需要消耗汽油 cost[i] 升。你从其中的一个加油站出发，开始时油箱为空。

如果你可以绕环路行驶一周，则返回出发时加油站的编号，否则返回 -1。

说明:

如果题目有解，该答案即为唯一答案。

输入数组均为非空数组，且长度相同。

输入数组中的元素均为非负数。

示例 1:

输入:

gas = [1,2,3,4,5]

cost = [3,4,5,1,2]

输出: 3

解释:

从 3 号加油站(索引为 3 处)出发，可获得 4 升汽油。此时油箱有 = 0 + 4 = 4 升汽油

开往 4 号加油站，此时油箱有 4 - 1 + 5 = 8 升汽油

开往 0 号加油站，此时油箱有 8 - 2 + 1 = 7 升汽油

开往 1 号加油站，此时油箱有 7 - 3 + 2 = 6 升汽油

开往 2 号加油站，此时油箱有 6 - 4 + 3 = 5 升汽油

开往 3 号加油站，你需要消耗 5 升汽油，正好足够你返回到 3 号加油站。

因此，3 可为起始索引。

示例 2:

输入:

gas = [2,3,4]

cost = [3,4,3]

输出: -1

解释:

你不能从 0 号或 1 号加油站出发，因为没有足够的汽油可以让你行驶到下一个加油站。

我们从 2 号加油站出发，可以获得 4 升汽油。 此时油箱有 = 0 + 4 = 4 升汽油

开往 0 号加油站，此时油箱有 4 - 3 + 2 = 3 升汽油

开往 1 号加油站，此时油箱有 3 - 3 + 3 = 3 升汽油

你无法返回 2 号加油站，因为返程需要消耗 4 升汽油，但是你的油箱只有 3 升汽油。

因此，无论怎样，你都不可能绕环路行驶一周。

题解：首先判断总油量是否大于总消耗，因为总油量小于总消耗时，怎么开都不可能绕一周。当存在解的时候解是唯一的。如果使用暴力搜索的方法会超时，哪怕是添加第一站的判断条件都超时。所以可以想一下，什么样的轨迹可以实现目的。因为油箱的容量无限，所以应该在经过加油站的时候，尽可能多的往油箱里面剩油。根据这个思路，应该找到第一站开始剩余油量最多的站点，通过gas-cost可以得到每一个站点的油耗，大于0表示剩余，所以对于新的gas-cost序列，可以求从索引0开始的最小的连续和，表示0-该点所需剩余油是最多的，并且保证有唯一解，那么从该站的下一站作为开始点，就可以保证油箱剩余油量是最多的，符合贪心策略。

class Solution:

    def canCompleteCircuit(self, gas: List[int], cost: List[int]) -> int:

        gas\_sum = sum(gas)

        cost\_sum = sum(cost)

        if gas\_sum < cost\_sum:

            return -1

        l = len(gas)

        sub = [gas[i]-cost[i] for i in range(l)]

        min = 1e8

        start = -1

        cur\_sum = 0

        for i in range(l):

            cur\_sum += sub[i]

            if cur\_sum < min:

                min = cur\_sum

                start = i

        return (start+1) % l

135

老师想给孩子们分发糖果，有 N 个孩子站成了一条直线，老师会根据每个孩子的表现，预先给他们评分。

你需要按照以下要求，帮助老师给这些孩子分发糖果：

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。

那么这样下来，老师至少需要准备多少颗糖果呢？

示例 1:

输入: [1,0,2]

输出: 5

解释: 你可以分别给这三个孩子分发 2、1、2 颗糖果。

示例 2:

输入: [1,2,2]

输出: 4

解释: 你可以分别给这三个孩子分发 1、2、1 颗糖果。

第三个孩子只得到 1 颗糖果，这已满足上述两个条件。

题解：假设A在B的左边相邻位置，如果rating[A]>rating[B]，那么A的糖果应该比B的糖果多1，称为右规则，如果rating[A]<rating[B]，那么A应该比B少1，称为左规则，如果相等，则糖果也相等。所以构建的糖果序列应该同时满足左规则和右规则。那么初始化糖果序列为1，从第二个开始，如果大于左侧则+1，这样的糖果序列满足左规则；再从倒数第二个开始，大于右侧则+1，这样的糖果序列满足右规则。对于满足左规则和右规则的糖果序列，每个位置取两个序列的最大值，便可以保证同时满足左规则和右规则，并且是最小的解。

class Solution:

    def candy(self, ratings: List[int]) -> int:

        left = [1 for i in range(len(ratings))]

        right = left[:]

        for i in range(1, len(ratings), 1):

            if ratings[i] > ratings[i-1]:

                left[i] = left[i-1] + 1

        cnt = left[len(ratings)-1]

        for i in range(len(ratings)-2, -1, -1):

            if ratings[i] > ratings[i+1]:

                right[i] = right[i+1] + 1

            cnt += max(left[i], right[i])

        print(left)

        print(right)

        return cnt

正确性：将问题拆分成两个子问题：1.如果评分比左侧高，那么糖果比左侧多；2.如果评分比右侧高，那么糖果比右侧多。然后将子问题合并，每个位置取最大值发糖，因为问题1和2要么是+1，要么是初始值，最大值可以保证同时满足两个子问题。如果是最小值，则只能满足一个问题。

题解2：糖果序列的结果只取决于局部范围内的rating，这个局部范围是两个相邻最低点的区间，以及区间内的最高点。最低点糖果数为1，最高点的取值应该是左侧最低点到最高点的递增序列个数left和最高点到右侧最低点的递减序列个数right中的最大值，较小的另一半则是从反方向开始加，每次值不同就加1。或者可以使用另一种方法，先按递增的数量发糖果，每有一个递增的数字就多加1，然后统计递减序列个数，首先按照递减序列的个数按照等差序列加糖果，并且如果递减序列个数大于递增序列个数，说明递减序列使用递增的糖是不够的，需要额外增加二者差值的糖。如果循环结束，递减序列个数大于0，则和上述方法一样的添加糖果。综上，这个方法主要看重递增序列，然后补全递减序列。class Solution:

    def candy(self, ratings: List[int]) -> int:

        res = 1

        pre = 1

        des\_num = 0

        for i in range(1, len(ratings)):

            if ratings[i] >= ratings[i-1]:

                if des\_num > 0:

                    res += ((1+des\_num)\*des\_num) // 2

                    if pre <= des\_num:

                        res += (des\_num - pre + 1)

                    pre = 1

                    des\_num = 0

                if ratings[i] == ratings[i-1]:

                    pre = 1

                    # continue

                else:

                    pre += 1

                res += pre

            else:

                des\_num += 1

        if des\_num > 0:

            res += ((1 + des\_num) \* des\_num) // 2

            if pre <= des\_num:

                res += (des\_num - pre + 1)

        return res

316

给定一个仅包含小写字母的字符串，去除字符串中重复的字母，使得每个字母只出现一次。需保证返回结果的字典序最小（要求不能打乱其他字符的相对位置）。

示例 1:

输入: "bcabc"

输出: "abc"

示例 2:

输入: "cbacdcbc"

输出: "acdb"

题解：要保证每个字母只出现一次并且字典序最小，最简单的想法是，当前字符如果还没有加到结果里，就把他加进去，如果这个字符已经在结果里，判断是否需要修改字符串结果，如果这个字符大于结果的最后一个字符，那么把这个字符加到结果后面删掉前面出现的字符，会得到更小字典序的结果。

利用栈来实现，当字符c不在栈中，如果c小于栈顶并且栈顶字符会在后续继续出现，那么就需要弹出栈顶，因为栈顶元素可以在后面遍历时加上，而且字典序小于之前的结果。比如bcabc，当前栈为bc，当入a时，bc都大于a并且都会在后续出现，所以弹出bc后以更小的a为首，得到了更小的字典序，并且bc会在后续加上，保证出现一次。

class Solution:

    def removeDuplicateLetters(self, s: str) -> str:

        stack = ['0']

        for i in range(len(s)):

            if s[i] not in stack:

                while s[i] < stack[-1] and stack[-1] in s[i+1:]:

                    stack.pop()

                stack.append(s[i])

        return "".join(stack[1:])

392

给定字符串 s 和 t ，判断 s 是否为 t 的子序列。

你可以认为 s 和 t 中仅包含英文小写字母。字符串 t 可能会很长（长度 ~= 500,000），而 s 是个短字符串（长度 <=100）。

字符串的一个子序列是原始字符串删除一些（也可以不删除）字符而不改变剩余字符相对位置形成的新字符串。（例如，"ace"是"abcde"的一个子序列，而"aec"不是）。

示例 1:

s = "abc", t = "ahbgdc"

返回 true.

示例 2:

s = "axc", t = "ahbgdc"

返回 false.

后续挑战 :

如果有大量输入的 S，称作S1, S2, ... , Sk 其中 k >= 10亿，你需要依次检查它们是否为 T 的子序列。在这种情况下，你会怎样改变代码？

class Solution:

    def isSubsequence(self, s: str, t: str) -> bool:

        i = 0

        j = 0

        while i < len(s) and j < len(t):

            if s[i] == t[j]:

                i += 1

                j += 1

            else:

                j += 1

        if i == len(s):

            return True

        return False

455

假设你是一位很棒的家长，想要给你的孩子们一些小饼干。但是，每个孩子最多只能给一块饼干。对每个孩子 i ，都有一个胃口值 gi ，这是能让孩子们满足胃口的饼干的最小尺寸；并且每块饼干 j ，都有一个尺寸 sj 。如果 sj >= gi ，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。

注意：

你可以假设胃口值为正。

一个小朋友最多只能拥有一块饼干。

示例 1:

输入: [1,2,3], [1,1]

输出: 1

解释:

你有三个孩子和两块小饼干，3个孩子的胃口值分别是：1,2,3。

虽然你有两块小饼干，由于他们的尺寸都是1，你只能让胃口值是1的孩子满足。

所以你应该输出1。

题解：最小的s去满足最小的g。

class Solution:

    def findContentChildren(self, g: List[int], s: List[int]) -> int:

        g.sort()

        s.sort()

        res = 0

        i, j = 0, 0

        while i < len(s) and j < len(g):

            if s[i] >= g[j]:

                i += 1

                j += 1

                res += 1

            else:

                i += 1

        return res

376

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。第一个差（如果存在的话）可能是正数或负数。少于两个元素的序列也是摆动序列。

例如， [1,7,4,9,2,5] 是一个摆动序列，因为差值 (6,-3,5,-7,3) 是正负交替出现的。相反, [1,4,7,2,5] 和 [1,7,4,5,5] 不是摆动序列，第一个序列是因为它的前两个差值都是正数，第二个序列是因为它的最后一个差值为零。

给定一个整数序列，返回作为摆动序列的最长子序列的长度。 通过从原始序列中删除一些（也可以不删除）元素来获得子序列，剩下的元素保持其原始顺序。

示例 1:

输入: [1,7,4,9,2,5]

输出: 6

解释: 整个序列均为摆动序列。

题解：寻找数字序列中的递增或递减序列，如果存在长序列，只能算一个数字的。比如1234只能算12。所以就是寻找这种递增或递减的数量，可以使用前缀标记的方法，设pre=0，如果Si>Si+1，cur=1，否则-1.然后比较pre和cur是否一致，如果相同，则表示和前面的序列性质一样，要么递增要么递减，只有不同是才会拐点，就有了摆动。然后记录pre=cur记录上一个数字的变化。

class Solution:

    def wiggleMaxLength(self, nums: List[int]) -> int:

        l = len(nums)

        if l < 2:

            return l

        res = 1

        pre = 0

        for i in range(1, l, 1):

            if nums[i] == nums[i-1]:

                continue

            elif nums[i] > nums[i-1]:

                cur = 1

            else:

                cur = -1

            res += pre != cur

            pre = cur

        return res

321

给定长度分别为 m 和 n 的两个数组，其元素由 0-9 构成，表示两个自然数各位上的数字。现在从这两个数组中选出 k (k <= m + n) 个数字拼接成一个新的数，要求从同一个数组中取出的数字保持其在原数组中的相对顺序。

求满足该条件的最大数。结果返回一个表示该最大数的长度为 k 的数组。

说明: 请尽可能地优化你算法的时间和空间复杂度。

示例 1:

输入:

nums1 = [3, 4, 6, 5]

nums2 = [9, 1, 2, 5, 8, 3]

k = 5

输出:

[9, 8, 6, 5, 3]

示例 2:

输入:

nums1 = [6, 7]

nums2 = [6, 0, 4]

k = 5

输出:

[6, 7, 6, 0, 4]

示例 3:

输入:

nums1 = [3, 9]

nums2 = [8, 9]

k = 3

输出:

[9, 8, 9]

题解：从数组中找到k个数字合并为一个新的数组形成最大数，并且保持相对顺序。没有说明这道题的数据范围，没想到可以直接使用宽度搜索，或者优化的暴力搜索。首先要找出k个数字，所以是从nums1中找到i个数字并且从nums2中找到k-i个数字，并且保持顺序。从数组中找i个数字形成最大数组可以使用栈，为了确保最终有i个数字，最多弹出次数为len(nums)-i，所以如果没超过最多弹出次数，并且当前的数字比栈顶更大，就应该弹出栈并放入更大的数字，这样可以得到更大的序列结果。其次是合并，合并过程使用归并排序的合并，因为得到的两个数组都是保持相对顺序的数组。最终遍历i从0到k，在保证选择数字个数小于数组长度的情况下，选择最大数组的子序列。

class Solution:

    def maxNumber(self, nums1: List[int], nums2: List[int], k: int) -> List[int]:

        def pick(nums, k):

            if k <= 0:

                return []

            res = []

            maxN = len(nums) - k

            for num in nums:

                while maxN > 0 and res and res[-1] < num:

                    res.pop()

                    maxN -= 1

                res.append(num)

            return res[:k]

        def merge(nums1, nums2):

            res = []

            while nums1 and nums2:

                if nums1 > nums2:

                    res.append(nums1.pop(0))

                else:

                    res.append(nums2.pop(0))

            res.extend(nums1 if nums1 else nums2)

            return res

        maxRes = []

        for i in range(k+1):

            if i <= len(nums1) and k-i <= len(nums2):

                maxRes = max(maxRes, merge(pick(nums1.copy(), i),pick(nums2.copy(), k-i)))

        return maxRes

330

给定一个已排序的正整数数组 nums，和一个正整数 n 。从 [1, n] 区间内选取任意个数字补充到 nums 中，使得 [1, n] 区间内的任何数字都可以用 nums 中某几个数字的和来表示。请输出满足上述要求的最少需要补充的数字个数。

示例 1:

输入: nums = [1,3], n = 6

输出: 1

解释:

根据 nums 里现有的组合 [1], [3], [1,3]，可以得出 1, 3, 4。

现在如果我们将 2 添加到 nums 中， 组合变为: [1], [2], [3], [1,3], [2,3], [1,2,3]。

其和可以表示数字 1, 2, 3, 4, 5, 6，能够覆盖 [1, 6] 区间里所有的数。

所以我们最少需要添加一个数字。

示例 2:

输入: nums = [1,5,10], n = 20

输出: 2

解释: 我们需要添加 [2, 4]。

示例 3:

输入: nums = [1,2,2], n = 5

输出: 0

题解：首先应该计算现有的数组能形成的和，然后遍历1-n，如果keys中存在该值，那么就是有和，继续寻找，如果没有，就是没有这个值，向数组中添加该值，然后遍历keys给所有已有的值加上新加的值，并且从这个值的下一个继续遍历。

上面方法求和复杂度太高，有其他的求覆盖范围的方法，设数组能表示的范围是[1,miss),因为miss没有覆盖到，所以添加miss到数组，这样覆盖范围新增了[miss,miss+miss)。所以如果nums[i]<miss，范围应该扩展到[1, nums[i]+miss)，否则扩展到[1,miss+miss)。

此外，最小数组能覆盖最大的和，应该是2的等比序列。1+2=3, 1+2+4=7。也就是2^n-1都能覆盖到。

以[1,2,3,8]和n=80为例。



Miss=1时，范围到1，迭代加1

Miss=2，范围到2，跌代加1

Miss=4，范围到4，迭代加1

Miss=7，此时nums[i]>miss，表示当前miss覆盖不到，[1,2,3]无法表示7，所以数组加上7，范围变成了14,此时miss=14

然后Nums[i]<miss，范围到22，miss=nums[i]+miss

然后i超过了数组长度，再也不能覆盖更大范围，只能通过添加数字的方式。每次加miss直到miss大于等于n，完成覆盖目的。

While miss <= n:

If nums[i]<=miss:

Miss += nums[i++]

Else:

Miss \*= 2

Res += 1

循环条件的<=n表示miss是达不到的最小值，miss==n也无法覆盖到n。nums[i]==miss时，刚好通过nums[i]加和可以得到miss+nums[i]-1的值。

所以贪心法找出的思路是对的，但是怎么去实现，以及怎么样的扩展思路也很重要。

本题最初的找出所有数字和的方向，也可以暴力实现，但是通过找当前数组索引的覆盖范围，逐步扩大，复杂度要低很多。从最初的点的搜索，扩展到区间的判断更简单。

class Solution:

    def minPatches(self, nums: List[int], n: int) -> int:

        p, i = 0, 0

        miss = 1

        while miss <= n:

            if i < len(nums) and nums[i] <= miss:

                miss = nums[i] + miss

                i += 1

            else:

                miss \*= 2

                p += 1

        return p

406

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。 编写一个算法来重建这个队列。

注意：

总人数少于1100人。

示例

输入:

[[7,0], [4,4], [7,1], [5,0], [6,1], [5,2]]

输出:

[[5,0], [7,0], [5,2], [6,1], [4,4], [7,1]]

题解：重新排序，使得排好序的满足在这个人前面且身高大于等于h的人数刚好为k，所以很容易想到解决方案。首先，按照k升序排列，如果k相同则按照h升序排列。遍历排好序的人群，如果k=0说明前面没人比自己高，顺序加入结果中，如果k!=0，则说明前面有比自己高的，并且h是升序排列，小h加入到队列中不影响大h在队伍中的k值，所以只需要顺序遍历，统计比这个人高的数量，等到数量和k一样时，插入到该位置，如果没有插入，说明人数不够，就加到最后一个。

class Solution:

    def reconstructQueue(self, people: List[List[int]]) -> List[List[int]]:

        import functools

        people.sort(key=functools.cmp\_to\_key(cmp))

        res = []

        for i in range(len(people)):

            if people[i][1] == 0:

                res.append(people[i])

            else:

                low = 0

                j = 0

                while j < i:

                    if res[j][0] >= people[i][0]:

                        low += 1

                    if low > people[i][1]:

                        res.insert(j, people[i])

                        break

                    j += 1

                if j == i:

                    res.append(people[i])

        return res

def cmp(a, b):

    if a[1] == b[1]:

        if a[0] > b[0]:

            return 1

        else:

            return -1

    elif a[1] > b[1]:

        return 1

    else:

        return -1

435

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。

注意:

可以认为区间的终点总是大于它的起点。

区间 [1,2] 和 [2,3] 的边界相互“接触”，但没有相互重叠。

示例 1:

输入: [ [1,2], [2,3], [3,4], [1,3] ]

输出: 1

解释: 移除 [1,3] 后，剩下的区间没有重叠。

示例 2:

输入: [ [1,2], [1,2], [1,2] ]

输出: 2

解释: 你需要移除两个 [1,2] 来使剩下的区间没有重叠。

示例 3:

输入: [ [1,2], [2,3] ]

输出: 0

题解：要删除所有重复区间，很容易想到对区间进行以开始时间升序的排列，然后比较每个区间和前一个区间是否冲突，如果冲突，则必须从中删除一个，因为要删掉最小的区间数量，所以从冲突的区间中选择结束时间最小的，这样留给后续区间的空间更大。

class Solution:

    def eraseOverlapIntervals(self, intervals: List[List[int]]) -> int:

        import functools

        intervals.sort(key=functools.cmp\_to\_key(cmp))

        print(intervals)

        res = 0

        i = 1

        while i < len(intervals):

            if intervals[i][0] < intervals[i-1][1]:

                if intervals[i][1] >= intervals[i-1][1]:

                    intervals.pop(i)

                else:

                    intervals.pop(i-1)

                res += 1

            else:

                i += 1

        return res

def cmp(a, b):

    if a[0] == b[0]:

        if a[1] > b[1]:

            return 1

        else:

            return -1

    elif a[0] > b[0]:

        return 1

    else:

        return -1

452

在二维空间中有许多球形的气球。对于每个气球，提供的输入是水平方向上，气球直径的开始和结束坐标。由于它是水平的，所以y坐标并不重要，因此只要知道开始和结束的x坐标就足够了。开始坐标总是小于结束坐标。平面内最多存在104个气球。

一支弓箭可以沿着x轴从不同点完全垂直地射出。在坐标x处射出一支箭，若有一个气球的直径的开始和结束坐标为 xstart，xend， 且满足  xstart ≤ x ≤ xend，则该气球会被引爆。可以射出的弓箭的数量没有限制。 弓箭一旦被射出之后，可以无限地前进。我们想找到使得所有气球全部被引爆，所需的弓箭的最小数量。

Example:

输入:

[[10,16], [2,8], [1,6], [7,12]]

输出:

2

解释:

对于该样例，我们可以在x = 6（射爆[2,8],[1,6]两个气球）和 x = 11（射爆另外两个气球）。

题解：题目要求是找出最小的重叠区间数量，若没有重叠区间，则该区间是单个区间。首先对区间序列开始时间升序排序，可以维护一个重叠区间指针start、end，如果新的区间new.start<start and new.end>end，那么不变，如果new.start>start and new.end>end, start=new.start，如果new.start>start and new.end<end, start=new.start, end=new.end，如果new.start<start and new.end<end, end=new.end；否则没有重叠存在，弓箭+1，start=new.start, end=new.end.

class Solution:

    def findMinArrowShots(self, points: List[List[int]]) -> int:

        if len(points) < 2:

            return len(points)

        points.sort(key=lambda x:x[0])

        print(points)

        res = 0

        start = points[0][0]

        end = points[0][1]

        for i in range(1, len(points)):

            if points[i][0] > end:

                res += 1

                start = points[i][0]

                end = points[i][1]

            elif points[i][0] < start:

                if points[i][1] < end:

                    end = points[i][1]

            else:

                start = points[i][0]

                if points[i][1] < end:

                    end = points[i][1]

        return res+1

402

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。

注意:

num 的长度小于 10002 且 ≥ k。

num 不会包含任何前导零。

示例 1 :

输入: num = "1432219", k = 3

输出: "1219"

解释: 移除掉三个数字 4, 3, 和 2 形成一个新的最小的数字 1219。

示例 2 :

输入: num = "10200", k = 1

输出: "200"

解释: 移掉首位的 1 剩下的数字为 200. 注意输出不能有任何前导零。

示例 3 :

输入: num = "10", k = 2

输出: "0"

解释: 从原数字移除所有的数字，剩余为空就是0。

题解：从数字字符串中移除k个数字字符得到最小的数字。对于数字的大小来说，左边越小数字越小，也就是说如果两个数字相邻并且左边更大，那么删除左边可以得到更小的结果，也就是相邻逆序对删除左边元素，这个很好想到，重点是对后续的处理。

1、如果删除了所有逆序对之后，删除的数量不够k，因为逆序对均被删除，剩下的字符串应该是一个不递减的序列，所以只需要从末尾开始删除剩余的元素。

这样就得到了最终结果，在结果中删除所有前导0即可。

可以看出，在数组的操作中，要记录当前字符和前面遍历过的没有删掉的字符，所以需要使用栈结构来存储，如果cur>=栈顶，则当前元素入栈；否则栈顶弹出并且直到找到一个不大于当前元素的数字，弹出的字符个数就是删掉的字符个数，然后该元素入栈。最后得到的栈顺序拿出就是最终的序列，如果删掉的元素个数不够k，只需要删掉末尾的k-m个元素。

想法不是很难，关键是栈结构的使用，没有想到，下次需要记录当前元素和遍历过的之前的元素做对比并且不确定对比数量时，就需要用栈，并且应该第一个想到栈。

class Solution:

    def removeKdigits(self, num: str, k: int) -> str:

        stack = []

        for d in num:

            while k and stack and stack[-1] > d:

                stack.pop()

                k -= 1

            stack.append(d)

        final = stack[:-k] if k else stack

        return "".join(final).lstrip("0") or "0"

502

假设 力扣（LeetCode）即将开始其 IPO。为了以更高的价格将股票卖给风险投资公司，力扣 希望在 IPO 之前开展一些项目以增加其资本。 由于资源有限，它只能在 IPO 之前完成最多 k 个不同的项目。帮助 力扣 设计完成最多 k 个不同项目后得到最大总资本的方式。

给定若干个项目。对于每个项目 i，它都有一个纯利润 Pi，并且需要最小的资本 Ci 来启动相应的项目。最初，你有 W 资本。当你完成一个项目时，你将获得纯利润，且利润将被添加到你的总资本中。

总而言之，从给定项目中选择最多 k 个不同项目的列表，以最大化最终资本，并输出最终可获得的最多资本。

示例 1:

输入: k=2, W=0, Profits=[1,2,3], Capital=[0,1,1].

输出: 4

解释:

由于你的初始资本为 0，你尽可以从 0 号项目开始。

在完成后，你将获得 1 的利润，你的总资本将变为 1。

此时你可以选择开始 1 号或 2 号项目。

由于你最多可以选择两个项目，所以你需要完成 2 号项目以获得最大的资本。

因此，输出最后最大化的资本，为 0 + 1 + 3 = 4。

注意:

假设所有输入数字都是非负整数。

表示利润和资本的数组的长度不超过 50000。

答案保证在 32 位有符号整数范围内。

题解：（暴力，超时）想法是对项目按照投资成本排序，每次使用二分找到小于等于已有成本的项目，在所有可以满足的项目中找到利润最大的项目投入，持续添加，但是在最后一个测试用例k=50000时超时了。

class Solution:

    def findMaximizedCapital(self, k: int, W: int, Profits: List[int], Capital: List[int]) -> int:

        if len(Profits) == 0:

            return W

        com = [[Capital[i], Profits[i]] for i in range(len(Profits))]

        import functools

        com.sort(key=functools.cmp\_to\_key(cmp))

        while k > 0 and com:

            index = self.find(com, W)

            if index == -1:

                break

            maxN = -1

            ii = -1

            for i in range(index+1):

                if com[i][1] > maxN:

                    ii = i

                    maxN = com[i][1]

            W += com[ii][1]

            k -= 1

            com.pop(ii)

        return W

    def find(self, nums, k):

        l = 0

        r = len(nums)-1

        while l < r:

            m = l + (r-l+1) // 2

            if nums[m][0] <= k:

                l = m

            else:

                r = m-1

        if nums[l][0] <= k:

            return l

        return -1

def cmp(a, b):

    if a[0] == b[0]:

        if a[1] > b[1]:

            return 1

        else:

            return -1

    elif a[0] < b[0]:

        return -1

    else:

        return 1

题解2：设输入项目数量为n,上述算法复杂度为O(k\*lgn\*n)，并且常数很大，k=50000时肯定超出。所以要想办法在找满足条件的项目以及其中利润最大的算法中进行优化。如果使用最大堆、最小堆的话，可以每次很快从中取出利润最大、并且成本满足的项目。

构造projects保存未展开的项目，available记录可投资的项目。按照启动资金排序，每次选取项目时，统计可以投资的项目，然后选择最大的利润，这样寻找最大利润的复杂度为O(lgN)，比上述题解更优。遍历放入可以投资的所有项目，在每次总资本增加后，只加入新的项目，并且使用最小堆维护项目资金，每次选择利润最高的项目即可。

from heapq import nlargest, heappop, heappush

class Solution:

    def findMaximizedCapital(self, k: int, W: int, Profits: List[int], Capital: List[int]) -> int:

        if len(Profits) == 0:

            return W

        com = [[Capital[i], Profits[i]] for i in range(len(Profits))]

        com.sort(key=lambda x:-x[0])

        from heapq import heappush, heappop

        ava = []

        while k > 0:

            while com and com[-1][0] <= W:

                heappush(ava, -com.pop()[1])

            if ava:

                W -= heappop(ava)

                k -= 1

            else:

                break

        return W

上述算法复杂度：while的k次循环，然后堆排序最多有n个元素，复杂度为nlgn，一共为O(k\*n\*lgn)。运行时间好过题解1的原因是，运行过程中，每次加入的项目是已经堆排序维护好的，而题解1每次都需要再从所有可投资项目中再寻找一次。或者题解1在找到最大的利润之后，对剩余可投资项目排序，取出最大，然后再找时，对第一个二分找到的index和第二个二分找到的index之间的项目排序，取最大值和第一区间最大值比较。但是复杂度为O(k\*lgn\*nlgn)还是更大。不如维护堆的成本更合适。

题解3：题解2的复杂度是由于维护堆排序需要lgn的复杂度，如果不维护，将已经投资的项目启动资金标记为inf表示不可用，则每次遍历找到可以投资的项目中利润最高的项目，复杂度为O(n\*min(n,k))

from heapq import nlargest, heappop, heappush

class Solution:

    def findMaximizedCapital(self, k: int, W: int, Profits: List[int], Capital: List[int]) -> int:

        from heapq import nlargest

        if W >= max(Capital):

            return W + sum(nlargest(k, Profits))

        n = len(Profits)

        for i in range(min(n ,k)):

            idx = -1

            for j in range(n):

                if Capital[j] <= W:

                    if idx == -1 or Profits[idx] < Profits[j]:

                        idx = j

            if idx == -1:

                break

            W += Profits[idx]

            Capital[idx] = float("inf")

        return W

判断过程发现，如果不添加W>=max()仍旧会超时。这样在题解1中添加这个条件后，发现通过样例。等新加账号后写到题解里，应该是测试样例有问题，复杂度考察不足。

44

给定一个字符串 (s) 和一个字符模式 (p) ，实现一个支持 '?' 和 '\*' 的通配符匹配。

'?' 可以匹配任何单个字符。

'\*' 可以匹配任意字符串（包括空字符串）。

两个字符串完全匹配才算匹配成功。

说明:

s 可能为空，且只包含从 a-z 的小写字母。

p 可能为空，且只包含从 a-z 的小写字母，以及字符 ? 和 \*。

示例 1:

输入:

s = "aa"

p = "a"

输出: false

解释: "a" 无法匹配 "aa" 整个字符串。

示例 2:

输入:

s = "aa"

p = "\*"

输出: true

解释: '\*' 可以匹配任意字符串。

示例 3:

输入:

s = "cb"

p = "?a"

输出: false

解释: '?' 可以匹配 'c', 但第二个 'a' 无法匹配 'b'。

示例 4:

输入:

s = "adceb"

p = "\*a\*b"

输出: true

解释: 第一个 '\*' 可以匹配空字符串, 第二个 '\*' 可以匹配字符串 "dce".

示例 5:

输入:

s = "acdcb"

p = "a\*c?b"

输入: false

题解：贪心法，问题的关键在于\*的处理。考虑如下情况，s[i]和p[j]不匹配，但是p[j]之前出现过\*，那么就可以修改p中\*的匹配范围，比如之前这个\*只匹配s[k]，现在发现不匹配那么就可以修改\*的匹配范围为s[k]-s[k+1]，直到s,p下一个字符是匹配的。也就是说，一旦发现不匹配，回溯\*的匹配位置延长匹配范围。并在最后处理掉多余\*。

使用双指针iS,jS记录\*在s、p的匹配位置。

If s[i] == p[j] or p[j] == ‘?’: 当前字符匹配，继续前进，并且此前出现的\*已经无法修改匹配范围。

I++, j++

Elif p[j] == \*: p中字符为\*，记录下匹配到的s和p中的位置，用来不匹配时回溯。

iS = i //s中不匹配的第一个位置

jS = j //p中第一个不确定匹配s字符的\*的位置

j++

elif iS>0: //表示当前s和p的字符不匹配，但是此前已经用过p中的\*来匹配过之前s，这时就可以通过回溯的方式，修改\*对于s的匹配范围。

I = iS+1

J = jS + 1

Else:

Return False

class Solution:

    def isMatch(self, s: str, p: str) -> bool:

        i, j, iS, jS, sL, pL = 0, 0, -1, -1, len(s), len(p)

        while i < sL:

            if j < pL and (s[i] == p[j] or p[j] == '?'):

                i += 1

                j += 1

            elif j < pL and p[j] == '\*':

                iS = i

                jS = j

                j += 1

            elif iS >= 0:

                i = iS + 1

                iS += 1

                j = jS + 1

            else:

                return False

        while j < pL and p[j] == '\*':

            j += 1

        return j == pL

714

给定一个整数数组 prices，其中第 i 个元素代表了第 i 天的股票价格 ；非负整数 fee 代表了交易股票的手续费用。

你可以无限次地完成交易，但是你每次交易都需要付手续费。如果你已经购买了一个股票，在卖出它之前你就不能再继续购买股票了。

返回获得利润的最大值。

示例 1:

输入: prices = [1, 3, 2, 8, 4, 9], fee = 2

输出: 8

解释: 能够达到的最大利润:

在此处买入 prices[0] = 1

在此处卖出 prices[3] = 8

在此处买入 prices[4] = 4

在此处卖出 prices[5] = 9

总利润: ((8 - 1) - 2) + ((9 - 4) - 2) = 8.

注意:

0 < prices.length <= 50000.

0 < prices[i] < 50000.

0 <= fee < 50000.

题解：初始题解，对于每天的股票，记录到栈中。如果今天股价高于昨天，如果栈中元素为1，说明应该买入，如果元素为2，则之前的最高价应该不卖，所以弹出，存入今天（最高价）；如果今天股价低于昨天，如果栈中元素为2，那么应该卖出股票，算得利润和0的最大值（利润0表示不买这个股票），如果元素为1，说明是递减序列，则应该弹出并放入股价更低的今天的价格。循环结束若栈中元素有2个，计算利润。

但是这个程序只通过了25个样例。

if len(prices) == 0:

            return 0

        stack = []

        res = 0

        stack.append(prices[0])

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

            if prices[i] > prices[i-1]:

                if len(stack) != 1:

                    stack.pop()

                stack.append(prices[i])

            else:

                if len(stack) == 2:

                    # if prices[i-1] - prices[i] >= fee:

                    if stack[1] - prices[i] >= fee or prices[i-1] - prices[i] >= fee:

                        res += max(stack[1] - stack[0] - fee, 0)

                        stack.pop()

                        stack.pop()

                        stack.append(prices[i])

                else:

                    stack.pop()

                    stack.append(prices[i])

        if len(stack) == 2:

            res += max(0, stack[1] - stack[0] - fee)

        return res

提示考虑每天的股票，我们持有股票或不持有股票的利润的最大值。假设cash表示不持有股票的最大利润，hold表示持有股票的最大利润。第0天时，cash=0，不买入,hold=-prices[0]，表示买了第一天的股票。然后更新cash和hold，cash=max(cash, hold+prices[i]-fee)，=cash表示没有买，另一项表示将持有的股票卖出得到的利润，卖出后不再持有。Hold=max(hold, cash-prices[i])，hold表示不卖，另一项表示买入当天的股票，买入后变为持有状态，最后返回cash。

cash, hold = 0, -prices[0]

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

            cash = max(cash, hold+prices[i]-fee)

            hold = max(hold, cash-prices[i])

        return cash

621

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。

示例 1：

输入: tasks = ["A","A","A","B","B","B"], n = 2

输出: 8

执行顺序: A -> B -> (待命) -> A -> B -> (待命) -> A -> B.

注：

任务的总个数为 [1, 10000]。

n 的取值范围为 [0, 100]。

题解：最初的想法是，对不同种类的任务计数，并且降序排列，因为出现次数最多的任务才是决定的关键。从第一个开始，在不到n的连续时间内插入不同的任务，如果没有可用任务就用待命，在到达n之后马上插入与第一个相同的任务（如果有），如果没有就判断第二个、第三个，并且在后面继续插入不同任务到n。转化之后可以将步骤分为n个时间片，在每个时间片插入一个没有使用过的任务。首先对出现次数降序排列，然后第一个使用出现最多的任务，然后递减使用（如果有），使用完但是还没有到n就补等待，time++。直到把出现次数最多的任务用完。

或者用优先队列实现，每次取出n个任务，数量减1，如果不够就不用管。直到队列为空就排任务完成。

class Solution:

    def leastInterval(self, tasks: List[str], n: int) -> int:

        cnt = [0 for \_ in range(26)]

        for c in tasks:

            cnt[ord(c)-ord('A')] += 1

        cnt.sort()

        time = 0

        while cnt[-1] > 0:

            i = 0

            while i <= n:

                if cnt[-1] == 0:

                    break

                if i < 26 and cnt[25-i] > 0:

                    cnt[25-i] -= 1

                time += 1

                i += 1

            cnt.sort()

        return time

题解2：使用桶来放任务，设桶大小为n+1，相同的任务不能放入同一个桶中，对于重复任务，只能放入不同桶中，所以桶的个数就是重复次数最多的任务的个数。一个桶占用时间为n+1，无论是否放满，因为放不满也需要待命。最后一个桶不需要放慢，时间为里面的任务个数。所以时间=(桶的个数-1)\*(n+1)+最后一个桶的任务数。

最后一个桶的任务数是任务次数=最多次数的任务个数，因为次数最多的都需要放到最后一个桶才能完成。如果任务很多，桶不够用时，比如AAABBCCDDEE最多次数为3，有3个桶，但是3个放不下所有任务，但是通过任务顺序的调整，总能在任务总个数时间内完成，因为最多的都不需要冷却，次数少的更不需要了。所以此时时间=任务个数。

from collections import Counter

class Solution:

def leastInterval(self, tasks: List[str], n: int) -> int:

ct = Counter(tasks)

nbucket = ct.most\_common(1)[0][1]

last\_bucket\_size = list(ct.values()).count(nbucket)

res = (nbucket - 1) \* (n + 1) + last\_bucket\_size

return max(res, len(tasks))