0.贪心总结.md

贪心问题的核心在于不用考虑全局只用考虑当前最优，当前最优能反推到全局最优。特殊情况需要配合排序，双次遍历来解决

- 跳跃问题
  - [ ] 45 跳跃游戏II
  - [ ] 55 跳跃游戏
- 股票问题
  - [ ] 121 买卖股票的最佳时机
  - [ ] 122 买卖股票的最佳时机II
- 排序+贪心
  - 区间问题： 排序只是为了保证每次只用对一个边界进行判断，因为另一个边界是有序的
    - [ ] 56 合并区间
    - [ ] 435 无重叠区间
    - [ ] 452 用最少数量的箭引爆气球
  - 技巧：排序是为了保证影响因素的单一
    - [ ] 406 根据身高重建队列
    - [ ] 455 分发饼干
- 双侧条件+两次遍历解法：每次遍历只满足一侧条件
  - [ ] 135 分发糖果
- Greedy Search思想
  - [ ] 53 最大子序和
  - [ ] 134 加油站：注意delta>0意味着一定有可以到达的路径
  - [ ] 179 最大数: 根据结果判断排序顺序
  - [ ] 376 摆动序列：左区间包含0，用来处理第一个位置
  - [ ] 605 种花问题：如何计算每个区间能种的花的数量，以及左右边界的处理
  - [ ] 674 最长连续递增序列
  - [ ] 738: 单调递增的数字：找到递减的位置，当前位置-1，把后面的所有数字都变成9999就是最大的递增数据
  - [ ] 763 划分字母区间：只用考虑右边界！碰到最远右边界之后才需要更新
  - [ ] 860 柠檬水找零：没啥说的遍历就完了

1005. K 次取反后最大化的数组和.md

给定一个整数数组 A，我们只能用以下方法修改该数组：我们选择某个索引 i 并将 A[i] 替换为 -A[i]，然后总共重复这个过程 K 次。（我们可以多次选择同一个索引 i。）

以这种方式修改数组后，返回数组可能的最大和。

示例 1：

输入：A = [4,2,3], K = 1
输出：5
解释：选择索引 (1,) ，然后 A 变为 [4,-2,3]。

示例 2：

输入：A = [3,-1,0,2], K = 3
输出：6
解释：选择索引 (1, 2, 2) ，然后 A 变为 [3,1,0,2]。

示例 3：

输入：A = [2,-3,-1,5,-4], K = 2
输出：13
解释：选择索引 (1, 4) ，然后 A 变为 [2,3,-1,5,4]。

提示：

```
1 <= A.length <= 10000
1 <= K <= 10000
-100 <= A[i] <= 100
```

```python
class Solution:
    def largestSumAfterKNegations(self, nums: List[int], k: int) -> int:
        nums.sort()
        total = 0
        for i in range(len(nums)):
            if nums[i]<0 and k>0:
                k-=1
                nums[i] = - nums[i]

        total = sum(nums)
        if k%2==1:
            minn = min(nums)
            return total - 2* minn
        else:
            return total
```

Tips

贪心，先sort(nums)，然后从最小开始遍历，在K允许的范围内把最大的负数都变成正数。遍历完一遍如果K>0，这时有两种情况。如果K是偶数这时可以直接求和返回，因为可以任意把一个元素变成负再变回正。如果k是奇数，这时选择nums中最小的数不停的反转最后得到负数

121. 买卖股票的最佳时机.md

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

你只能选择 某一天 买入这只股票，并选择在 未来的某一个不同的日子 卖出该股票。设计一个算法来计算你所能获取的最大利润。

返回你可以从这笔交易中获取的最大利润。如果你不能获取任何利润，返回 0 。

示例 1：

输入：[7,1,5,3,6,4]
输出：5
解释：在第 2 天（股票价格 = 1）的时候买入，在第 5 天（股票价格 = 6）的时候卖出，最大利润 = 6-1 = 5 。
注意利润不能是 7-1 = 6, 因为卖出价格需要大于买入价格；同时，你不能在买入前卖出股票。

示例 2：

输入：prices = [7,6,4,3,1]
输出：0
解释：在这种情况下, 没有交易完成, 所以最大利润为 0。

提示：

```
1 <= prices.length <= 105
0 <= prices[i] <= 104
```

1. 贪心算法

虽然所有股票题都可以用动态规划来做，但是其实每个问题都有更合适的独立解法。只能买卖一次的问题适合用贪心来做，找左侧的最小值和右侧的最大值，diff就是最大收益。实现方式是不断更新左侧的最小值，然后用当前值更新maxprfix

```python
class Solution:
    def maxProfit(self, prices: List[int]) -> int:
        profit =0
        minprice =prices[0]
        for i in range(1, len(prices)):
            minprice= min(minprice, prices[i])
            profit = max(profit, prices[i]-minprice)


        return profit
```

2. 动态规划

这里动态规划的内存占用和耗时都更差，不过可以把所有股票问题用统一的逻辑来解决

这里有两个状态，$dp[i][0]$持有股票的现金账户（可以是当前买入，或者之前买入持有），和$dp[i][1]$不持有股票的看金账户（不持有可以是未买入或者已经卖出。

状态转移

- 只能买卖一次，持有现金要么是延续之前持有，要么是当天买入成本
  $dp[i][0] = max(dp[i-1][0], -prices[i])$
- 不持有现金，要么是延续之前卖出收益，要么是T-1的成本当天买入收益
  $dp[i][1] = max(dp[i-1][1], dp[i-1][0] + prices[i])$

初始化

- $dp[i][0] = -prices[0]$
- $dp[i][]$

```python

class Solution:
    def maxProfit(self, prices: List[int]) -> int:
        l = len(prices)
        dp0 = [0] * l
        dp1 = [0] * l
        dp0[0] = -prices[0]

        for i in range(1,l):
            dp0[i] = max(dp0[i-1], -prices[i])
            dp1[i] = max(dp1[i-1], dp0[i-1] + prices[i])
        return dp1[-1]
```

122. 买卖股票的最佳时机 II.md

给定一个数组 prices ，其中 prices[i] 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: prices = [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 。

示例 2:

输入: prices = [1,2,3,4,5]
输出: 4
解释: 在第 1 天（股票价格 = 1）的时候买入，在第 5 天 （股票价格 = 5）的时候卖出, 这笔交易所能获得利润 = 5-1 = 4 。
　注意你不能在第 1 天和第 2 天接连购买股票，之后再将它们卖出。因为这样属于同时参与了多笔交易，你必须在再次购买前出售掉之前的股票。

示例 3:

输入: prices = [7,6,4,3,1]
输出: 0
解释: 在这种情况下, 没有交易完成, 所以最大利润为 0。

提示：

```
1 <= prices.length <= 3 * 104
0 <= prices[i] <= 104
```

1. 贪心算法

贪心，每一步只用考虑当前价格和下一个价格，如果当前低，则买入，再下一刻卖出。很容易想歪，想到什么找局部min，局部max，然后求diff之类的，其实只要低就买，高就跳过就好

```python
class Solution:
    def maxProfit(self, prices: List[int]) -> int:
        profit = 0
        for i in range(len(prices)-1):

            if prices[i+1]-prices[i]>0:
                profit += prices[i+1]-prices[i]
        return profit
```

2. 动态规划

这里动态规划的内存占用和耗时都更差，不过可以把所有股票问题用统一的逻辑来解决

这里有两个状态，$dp[i][0]$持有股票的现金账户（可以是当前买入，或者之前买入持有），和$dp[i][1]$不持有股票的看金账户（不持有可以是未买入或者已经卖出。

状态转移

- 买卖多次，持有现金要么是延续之前持有，要么在T-1卖出后再买入
  $dp[i][0] = max(dp[i-1][0], dp[i-1][1] - prices[i])$
- 不持有现金，要么是延续之前卖出收益，要么是T-1的成本当天买入收益
  $dp[i][1] = max(dp[i-1][1], dp[i-1][0] + prices[i])$

初始化

- $dp[i][0] = -prices[0]$
- $dp[i][1] = 0$

```python
class Solution:
    def maxProfit(self, prices: List[int]) -> int:
        l = len(prices)
        dp0 = [0] * l
        dp1 = [0] * l
        dp0[0] = -prices[0]

        for i in range(1,l):
            dp0[i] = max(dp0[i-1], dp1[i-1]-prices[i])
            dp1[i] = max(dp1[i-1], dp0[i-1] + prices[i])
        return dp1[-1]
```

134. 加油站.md

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。

```python
class Solution:
    def canCompleteCircuit(self, gas: List[int], cost: List[int]) -> int:
        totalsum = 0
        cursum = 0
        index =0
        for i in range(len(gas)):
            cur = gas[i]-cost[i]
            totalsum+=cur
            cursum +=cur
            if cursum<0:
```

```
                index =i+1
                cursum = 0

        if totalsum<0:
            return -1
        else:
            return index
```

Tips

贪心算法，1个变量统计total(gas-cost)如果这个值>0，则一定存在可以达到的路径，否则直接返回-1。如果存在路径，每步就可以用贪心计算，加入当前gas-cost如果cursum<0，则之前的路径肯定不可用，这时开始重新计数cursum归0，开始点变为下一位。这里不太容易想到为啥之前cursum<0，起始点一定在当前位置之后，因为totalsum>=0，所以前面cursum<0，后面一定存在cursum>0

135. 分发糖果.md

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。
评分更高的孩子必须比他两侧的邻位孩子获得更多的糖果。
```

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。

```python
class Solution:
    def candy(self, ratings: List[int]) -> int:
        n=len(ratings)
        ans = [1] * n
        for i in range(1,n):
            if ratings[i]>ratings[i-1]:
                ans[i] = ans[i-1]+1

        for i in range(n-2,-1,-1):
            if ratings[i]>ratings[i+1]:
                ans[i] =max(ans[i+1]+1, ans[i])
        return sum(ans)
```

Tips

贪心，永远只看局部最优，力争每次只处理一个规则。

1. 先处理右边大于左边，一次遍历，局部调整方式就是ans[i] = ans[i+1]+1
2. 再反向遍历处理左边大于右边，一次遍历，每一步的调整方式是在满足上一步分发的糖果不变小的基础上进行调整ans[i] = max(ans[i+1]+1, ans[i])

179. 最大数.md

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。

注意：输出结果可能非常大，所以你需要返回一个字符串而不是整数。

示例 1：

输入：nums = [10,2]
输出："210"

示例 2：

输入：nums = [3,30,34,5,9]
输出："9534330"

示例 3：

输入：nums = [1]
输出："1"

示例 4：

输入：nums = [10]
输出："10"

提示：

```
1 <= nums.length <= 100
0 <= nums[i] <= 109
```

```
class Solution:
    def largestNumber(self, nums: List[int]) -> str:
        def cmp(a,b):
            if a+b<b+a:
                return -1
            elif a+b==b+a:
                return 0
            else:
                return 1
        nums = sorted([str(i) for i in nums], key=functools.cmp_to_key(cmp),
reverse=True)
        if nums[0]=='0':
            return '0'
        else:
            return ''.join(nums)
```

Tips

1. 直接对比a和b的大小不好对比，可以通过直接对比拼接以后的结果哪个更大来决定
2. 贪心排序，每两个元素两两交换顺序，通过functools.cmp_to_key来实现

376. 摆动序列.md

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。仅有一个元素或者含两个不等元素的序列也视作摆动序列。

```
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。
```

子序列 可以通过从原始序列中删除一些（也可以不删除）元素来获得，剩下的元素保持其原始顺序。

给你一个整数数组 nums ，返回 nums 中作为 摆动序列 的 最长子序列的长度 。

示例 1：

输入：nums = [1,7,4,9,2,5]
输出：6
解释：整个序列均为摆动序列，各元素之间的差值为 (6, -3, 5, -7, 3) 。

示例 2：

输入：nums = [1,17,5,10,13,15,10,5,16,8]
输出：7
解释：这个序列包含几个长度为 7 摆动序列。
其中一个是 [1, 17, 10, 13, 10, 16, 8] ，各元素之间的差值为 (16, -7, 3, -3, 6, -8) 。

示例 3：

输入：nums = [1,2,3,4,5,6,7,8,9]
输出：2

提示：

```
1 <= nums.length <= 1000
0 <= nums[i] <= 1000
```

进阶：你能否用 `O(n)` 时间复杂度完成此题?

```
class Solution:
    def wiggleMaxLength(self, nums: List[int]) -> int:
        prediff, curdiff, counter = 0,0,1
        for i in range(len(nums)-1):
            curdiff = nums[i+1]-nums[i]
            if (curdiff>0 and prediff <=0) or (curdiff<0 and prediff>=0):
                counter+=1
                prediff = curdiff
        return counter
```

Tips

贪心算法，只用考虑当前状态是否前一个diff和当前diff的方向不同。prediff包含=0的状态，是为了左边界而设。
如果只有两个元素[2,5]此时答案为2，可以变成[2,2,5],也就是prediff=0，curdiff=4

406. 根据身高重建队列.md

假设有打乱顺序的一群人站成一个队列，数组 people 表示队列中一些人的属性（不一定按顺序）。每个 people[i] = [hi, ki] 表示第 i 个人的身高为 hi ，前面 正好 有 ki 个身高大于或等于 hi 的人。

请你重新构造并返回输入数组 people 所表示的队列。返回的队列应该格式化为数组 queue ，其中 queue[j] = [hj, kj] 是队列中第 j 个人的属性（queue[0] 是排在队列前面的人）。

示例 1：

输入：people = [[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]]
解释：
编号为 0 的人身高为 5 ，没有身高更高或者相同的人排在他前面。
编号为 1 的人身高为 7 ，没有身高更高或者相同的人排在他前面。
编号为 2 的人身高为 5 ，有 2 个身高更高或者相同的人排在他前面，即编号为 0 和 1 的人。
编号为 3 的人身高为 6 ，有 1 个身高更高或者相同的人排在他前面，即编号为 1 的人。
编号为 4 的人身高为 4 ，有 4 个身高更高或者相同的人排在他前面，即编号为 0、1、2、3 的人。
编号为 5 的人身高为 7 ，有 1 个身高更高或者相同的人排在他前面，即编号为 1 的人。
因此 [[5,0],[7,0],[5,2],[6,1],[4,4],[7,1]] 是重新构造后的队列。

示例 2：

输入：people = [[6,0],[5,0],[4,0],[3,2],[2,2],[1,4]]
输出：[[4,0],[5,0],[2,2],[3,2],[1,4],[6,0]]

提示：

```
1 <= people.length <= 2000
0 <= hi <= 106
0 <= ki < people.length
题目数据确保队列可以被重建
```

```python
class Solution:
    def reconstructQueue(self, people: List[List[int]]) -> List[List[int]]:
        people = sorted(people, key=lambda x:(-x[0],x[1])) #按身高倒排，按index正派
        ans = list()
        for p in people:
            ans.insert(p[1],p)
        return ans
```

Tips

贪心问题，祝勇考虑局部状态，满足每个people的局部最优既为全局最优

这题非常巧妙在于，如果按身高倒排，相同身高按index正排。在向最终结果中插入当前people时，只有在它前面插入的people会有影响，而前方的影响完全反应在当前people的index中，在它后面的poeple都和当前people的index无关。所以只需向前遍历，并考虑当前状态即可。

435. 无重叠区间.md

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

解释: 你不需要移除任何区间，因为它们已经是无重叠的了。

```python
class Solution:
    def eraseOverlapIntervals(self, intervals: List[List[int]]) -> int:
        intervals = sorted(intervals, key = lambda x: x[1])
        rm_counter =0
        right = intervals[0][1]
        for i in intervals[1:]:
            if i[0]< right:
                rm_counter+=1
            else:
                right = i[1]
        return rm_counter
```

Tips

区间问题需要借助排序+贪心来解决。最大屋重叠区间就是每一步都保留右边界最小的一个，給剩下的区间尽可能多的地方。所以先按右边界升序sort，然后依次遍历统计需要移除的区间数

45. 跳跃游戏 II.md

给你一个非负整数数组 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。

你的目标是使用最少的跳跃次数到达数组的最后一个位置。

假设你总是可以到达数组的最后一个位置。

示例 1:

输入: nums = [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。

示例 2:

输入: nums = [2,3,0,1,4]
输出: 2

提示:

```
1 <= nums.length <= 104
0 <= nums[i] <= 1000
```

```python
class Solution:
    def jump(self, nums: List[int]) -> int:
        counter = 0
        longest = 0
        reach = 0
        for i in range(len(nums)):
            if longest < i:
                counter +=1
                longest =reach
            reach = max(reach, i+nums[i])
        return counter
```

Tips

贪心算法每一步都计算当前步所能到达的最远距离，当跨过上一次能覆盖的最远距离后counter+1。

这时又两种情况，上次的最远距离依旧是当前的最远距离这里自然+1，另一种是在中间存在更远的跳跃，reach>longest，这时也会自然+1

452. 用最少数量的箭引爆气球.md

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，所以纵坐标并不重要，因此只要知道开始和结束的横坐标就足够了。开始坐标总是小于结束坐标。

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

给你一个数组 points ，其中 points [i] = [xstart,xend] ，返回引爆所有气球所必须射出的最小弓箭数。

示例 1：

输入：points = [[10,16],[2,8],[1,6],[7,12]]
输出：2
解释：对于该样例，x = 6 可以射爆 [2,8],[1,6] 两个气球，以及 x = 11 射爆另外两个气球

示例 2：

输入：points = [[1,2],[3,4],[5,6],[7,8]]
输出：4

示例 3：

输入：points = [[1,2],[2,3],[3,4],[4,5]]
输出：2

示例 4：

输入：points = [[1,2]]
输出：1

示例 5：

输入：points = [[2,3],[2,3]]
输出：1

提示：

```
1 <= points.length <= 104
points[i].length == 2
-231 <= xstart < xend <= 231 - 1
```

```python
class Solution:
    def findMinArrowShots(self, points: List[List[int]]) -> int:
        points = sorted(points, key=lambda x:x[0])
        boundary =points[0][1]
        counter = 1
        for p in points[1:]:
            if p[0] > boundary:
                boundary = p[1]
                counter+=1
            else:
                boundary = min(boundary, p[1])
        return counter
```

Tips

同样是贪心中的区间问题，最小箭 = len-最大重叠数。和无重叠区间刚好相反，这题是如何让区间尽可能多的重叠，所以按左边界升序排列，把start离得近的放在一起。每一步只需要判断当前start是否在当前boundary之内，并用当前的end更新boundary，如果超出+1，并得到新的boundary。因为已经对左边界进行了排序，所以只用对右边界进行更新

455. 分发饼干.md

假设你是一位很棒的家长，想要给你的孩子们一些小饼干。但是，每个孩子最多只能给一块饼干。

对每个孩子 i，都有一个胃口值 g[i]，这是能让孩子们满足胃口的饼干的最小尺寸；并且每块饼干 j，都有一个尺寸 s[j] 。如果 s[j] >= g[i]，我们可以将这个饼干 j 分配给孩子 i ，这个孩子会得到满足。你的目标是尽可能满足越多数量的孩子，并输出这个最大数值。

示例 1:

输入: g = [1,2,3], s = [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。

示例 2:

输入: g = [1,2], s = [1,2,3]
输出: 2
解释:
你有两个孩子和三块小饼干，2个孩子的胃口值分别是1,2。
你拥有的饼干数量和尺寸都足以让所有孩子满足。
所以你应该输出2.

提示：

```
1 <= g.length <= 3 * 104
0 <= s.length <= 3 * 104
1 <= g[i], s[j] <= 231 - 1
```

```python
class Solution:
    def findContentChildren(self, g: List[int], s: List[int]) -> int:
        g.sort()
        s.sort()
        counter = 0
        i =0
        j = 0
        ng = len(g)
        ns = len(s)
        while i<ng and j< ns:
            if g[i]<=s[j]:
                counter+=1
                i+=1
                j+=1
            else:
                j+=1
        return counter
```

Tips

贪心算法，sort一下然后每个人都给比他需要的大又最小的饼干

53. 最大子序和.md

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。

示例 1：

输入：nums = [-2,1,-3,4,-1,2,1,-5,4]
输出：6
解释：连续子数组 [4,-1,2,1] 的和最大，为 6 。

示例 2：

输入：nums = [1]
输出：1

示例 3：

输入：nums = [0]
输出：0

示例 4：

输入：nums = [-1]
输出：-1

示例 5：

输入：nums = [-100000]
输出：-100000

提示：

```
1 <= nums.length <= 105
-104 <= nums[i] <= 104
```

进阶：如果你已经实现复杂度为 O(n) 的解法，尝试使用更为精妙的 分治法 求解。

1. 贪心

```python
class Solution:
    def maxSubArray(self, nums: List[int]) -> int:
        cur = nums[0]
        maxn = nums[0]
        for i in nums[1:]:
            if cur >0:
                cur += i
            else:
                cur = i
            maxn = max(maxn, cur)
        return maxn
```

Tips

贪心，每一步都判断之前的cursum是否<0，如果是重新开始，否则累加

2.

55. 跳跃游戏.md

给定一个非负整数数组 nums ，你最初位于数组的 第一个下标 。

数组中的每个元素代表你在该位置可以跳跃的最大长度。

判断你是否能够到达最后一个下标。

示例 1：

输入：nums = [2,3,1,1,4]
输出：true
解释：可以先跳 1 步，从下标 0 到达下标 1, 然后再从下标 1 跳 3 步到达最后一个下标。

示例 2：

输入：nums = [3,2,1,0,4]
输出：false
解释：无论怎样，总会到达下标为 3 的位置。但该下标的最大跳跃长度是 0 ， 所以永远不可能到达最后一个下标。

提示：

```
1 <= nums.length <= 3 * 104
0 <= nums[i] <= 105
```

```python
class Solution:
    def canJump(self, nums: List[int]) -> bool:
        dist=0
        n = len(nums) -1
        for i,k in enumerate(nums):
            print(i,dist)
            if dist<i:
                return False
            if dist>=n:
                return True
            dist = max(dist, i+k)
        return False
```

Tips

贪心算法，每一步都判断之前的最远距离是否能cover当前位置，以及是否已经cover最远位置，并更新最远距离

1. 默认都按从0开始，所以最后一位是len(nums)-1,>=既为True

56. 合并区间.md

以数组 intervals 表示若干个区间的集合，其中单个区间为 intervals[i] = [starti, endi] 。请你合并所有重叠的区间，并返回一个不重叠的区间数组，该数组需恰好覆盖输入中的所有区间。

示例 1：

输入：intervals = [[1,3],[2,6],[8,10],[15,18]]
输出：[[1,6],[8,10],[15,18]]
解释：区间 [1,3] 和 [2,6] 重叠, 将它们合并为 [1,6].

示例 2：

输入：intervals = [[1,4],[4,5]]
输出：[[1,5]]
解释：区间 [1,4] 和 [4,5] 可被视为重叠区间。

提示：

```
1 <= intervals.length <= 104
intervals[i].length == 2
0 <= starti <= endi <= 104
```

```python
class Solution:
    def merge(self, intervals: List[List[int]]) -> List[List[int]]:
        intervals = sorted(intervals, key = lambda x:x[0])
        res = []
        pre = intervals[0]
        for i in intervals[1:]:
            if i[0]<= pre[1]:
                pre[1] = max(pre[1],i[1])
            else:
                res.append(pre)
                pre = i
        res.append(pre)
        return res
```

Tips

贪心算法，碰到区间问题必要先sort。合并过程中之需要考虑当前的区间和上一个区间是否存在重合，如果存在则更新右边界，如果不重合就放入结果。不要忘记最后一个结果的放入

605. 种花问题.md

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。

示例 1：

输入：flowerbed = [1,0,0,0,1], n = 1
输出：true

示例 2：

输入：flowerbed = [1,0,0,0,1], n = 2
输出：false

提示：

```
1 <= flowerbed.length <= 2 * 104
flowerbed[i] 为 0 或 1
flowerbed 中不存在相邻的两朵花
0 <= n <= flowerbed.length
```

```python
class Solution:
    def canPlaceFlowers(self, flowerbed: List[int], n: int) -> bool:
        counter =0
        prepos = -1
        l = len(flowerbed)
        for i in range(l):
            if flowerbed[i]==1:
                if prepos<0:
                    counter+=(i-prepos-1)//2
                else:
                    counter+= (i-prepos-2)//2 #-2因为左右不能碰到所以长度缩减2
                prepos = i
            print(prepos, counter )

        if prepos<0:
            counter +=  (l-prepos)//2
        else:
            counter += (l-prepos-1)//2 #右边界无碍所以
        print(counter)
        if counter >=n:
            return True
        else:
            return False
```

Tips

1. Greedy的思想，每次碰到1就把和上一个1之间的位置种上最多的树
2.

674. 最长连续递增序列.md

给定一个未经排序的整数数组，找到最长且 连续递增的子序列，并返回该序列的长度。

连续递增的子序列 可以由两个下标 l 和 r（l < r）确定，如果对于每个 l <= i < r，都有 nums[i] < nums[i + 1] ，那么子序列 [nums[l], nums[l + 1], ..., nums[r - 1], nums[r]] 就是连续递增子序列。

示例 1：

输入：nums = [1,3,5,4,7]
输出：3
解释：最长连续递增序列是 [1,3,5], 长度为3。
尽管 [1,3,5,7] 也是升序的子序列, 但它不是连续的，因为 5 和 7 在原数组里被 4 隔开。

示例 2：

输入：nums = [2,2,2,2,2]
输出：1
解释：最长连续递增序列是 [2], 长度为1。

提示：

```
1 <= nums.length <= 104
-109 <= nums[i] <= 109
```

1. 贪心算法

```python
class Solution:
    def findLengthOfLCIS(self, nums: List[int]) -> int:
        ans = 1
        cur = 1
        for i,n in enumerate(nums[1:]):
            print(i, nums[i],n)
            if n > nums[i]:
                cur+=1
            else:
                ans = max(ans, cur)
                cur = 1
        ans = max(ans, cur)
        return ans
```

2. 动态规划

```python
class Solution:
    def findLengthOfLCIS(self, nums: List[int]) -> int:
        max_len = 1
        l = len(nums)
        dp = [1] * len(nums)
        for i in range(1, l):
            if nums[i] > nums[i-1]:
                dp[i] = dp[i-1] +1
            max_len = max(max_len, dp[i])
        return max_len
```

Tips

- 最长连续序列类问题的初始化都是1
- 状态转移如果存在连续递增的两个值 dp[i] = max(dp[i], dp[i-1] +1)
-

738. 单调递增的数字.md

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。

（当且仅当每个相邻位数上的数字 x 和 y 满足 x <= y 时，我们称这个整数是单调递增的。）

示例 1:

输入: N = 10
输出: 9

示例 2:

输入: N = 1234
输出: 1234

示例 3:

输入: N = 332
输出: 299

说明: N 是在 [0, 10^9] 范围内的一个整数。

```python
class Solution:
    def monotoneIncreasingDigits(self, n: int) -> int:
        s=list(str(n))
        l = len(s)
        for i in range(l-1, 0, -1):
            if s[i-1]>s[i]:
                s[i-1] = str(int(s[i-1])-1)
                s[i:] = ['9'] * (l-i)
        return int(''.join(s))
```

Tips

为得到单调递增的数字，在碰到每一个不单调的地方，都把当前数值-1，把后面的部分都变成9999

763. 划分字母区间.md

字符串 S 由小写字母组成。我们要把这个字符串划分为尽可能多的片段，同一字母最多出现在一个片段中。返回一个表示每个字符串片段的长度的列表。

示例：

输入：S = "ababcbacadefegdehijhklij"
输出：[9,7,8]
解释：
划分结果为 "ababcbaca", "defegde", "hijhklij"。
每个字母最多出现在一个片段中。
像 "ababcbacadefegde", "hijhklij" 的划分是错误的，因为划分的片段数较少。

提示：

```
S的长度在[1, 500]之间。
S只包含小写字母 'a' 到 'z' 。
```

```python
class Solution:
    def partitionLabels(self, s: str) -> List[int]:
        pos = {}
        for i in range(len(s)):
            pos[s[i]] = i

        left =0
        right = 0
        res = []
```

```
        for i in range(len(s)):
            right = max(right, pos[s[i]])
            if i == right:
                res.append(right-left+1)
                left = i+1
        return res
```

Tips

1. 碰到区间问题，必要先统计位置。在当前位置之需要判断是否更新右边界，以及是否达到了右边界，达到后更新区间和左边界。
2. 最开始以为需要维护左右两个边界，后来发现只要在右边界到达之前有其他字母更新更远的右边界即可

860. 柠檬水找零.md

在柠檬水摊上，每一杯柠檬水的售价为 5 美元。顾客排队购买你的产品，（按账单 bills 支付的顺序）一次购买一杯。

每位顾客只买一杯柠檬水，然后向你付 5 美元、10 美元或 20 美元。你必须给每个顾客正确找零，也就是说净交易是每位顾客向你支付 5 美元。

注意，一开始你手头没有任何零钱。

给你一个整数数组 bills ，其中 bills[i] 是第 i 位顾客付的账。如果你能给每位顾客正确找零，返回 true ，否则返回 false 。

示例 1：

输入：bills = [5,5,5,10,20]
输出：true
解释：
前 3 位顾客那里，我们按顺序收取 3 张 5 美元的钞票。
第 4 位顾客那里，我们收取一张 10 美元的钞票，并返还 5 美元。
第 5 位顾客那里，我们找还一张 10 美元的钞票和一张 5 美元的钞票。
由于所有客户都得到了正确的找零，所以我们输出 true。

示例 2：

输入：bills = [5,5,10,10,20]
输出：false
解释：
前 2 位顾客那里，我们按顺序收取 2 张 5 美元的钞票。
对于接下来的 2 位顾客，我们收取一张 10 美元的钞票，然后返还 5 美元。
对于最后一位顾客，我们无法退回 15 美元，因为我们现在只有两张 10 美元的钞票。
由于不是每位顾客都得到了正确的找零，所以答案是 false。

示例 3：

输入：bills = [5,5,10]
输出：true

示例 4：

输入：bills = [10,10]
输出：false

提示：

```
1 <= bills.length <= 105
bills[i] 不是 5 就是 10 或是 20
```

```python
class Solution:
    def lemonadeChange(self, bills: List[int]) -> bool:
        dic = defaultdict(int)
        for i in bills:
            if i == 5:
                dic[5]+=1
            elif i ==10:
                if dic[5]<1:
                    return False
                else:
                    dic[10]+=1
                    dic[5]-=1
            else:
                if dic[10]>=1 and dic[5]>=1:
                    dic[10]-=1
                    dic[5]-=1
                    dic[20]+=1
                elif dic[5]>=3:
                    dic[20]+=1
                    dic[5]-=3
                else:
                    return False

        return True
```

Tips

贪心算法，在每一步之用判断当前状态下能否找零即可，如果是20有限用10+5，因为10只能用于20找零，而5可以用于更多场景