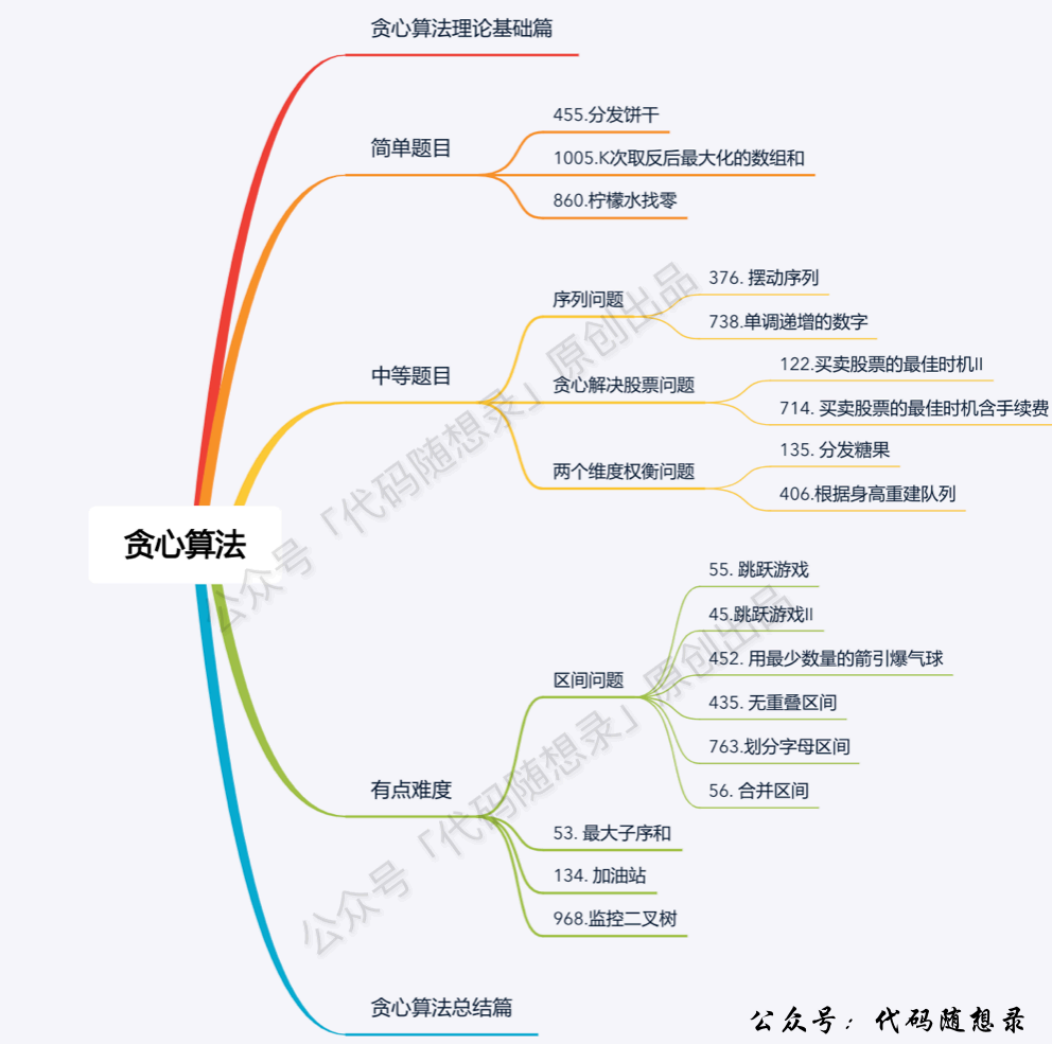
贪心算法



贪心的本质是选择每一阶段的局部最优，从而达到全局最优。

这么说有点抽象，来举一个例子：

例如，有一堆钞票，你可以拿走十张，如果想达到最大的金额，你要怎么拿?

指定每次拿最大的，最终结果就是拿走最大数额的钱。

每次拿最大的就是局部最优，最后拿走最大数额的钱就是推出全局最优。

再举一个例子如果是 有一堆盒子，你有一个背包体积为n，如何把背包尽可能装满，如果还每次选最大的盒子，就不行了。

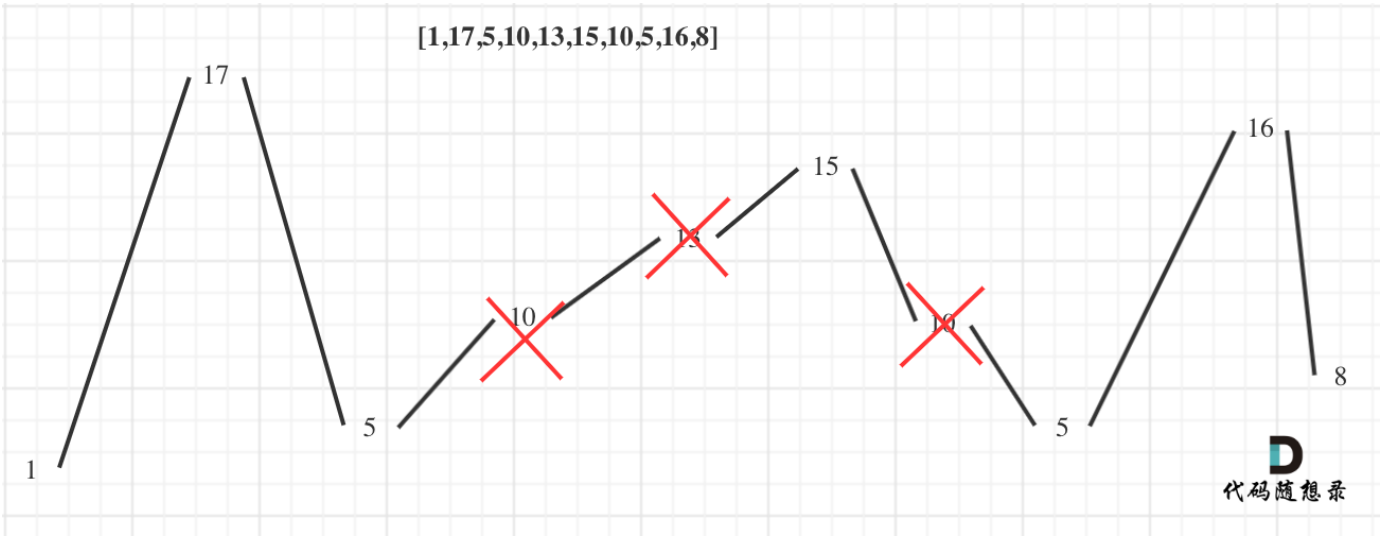
无固定套路 只能靠积累

455. Assign Cookies

技巧：要遍历饼干和小孩。但是只用一次for循环遍历小孩，用指针来遍历饼干。

```
class Solution:
    def findContentChildren(self, g: List[int], s: List[int]) -> int:
        #贪心算法 先用大饼干喂饱大胃口
        g.sort()
        s.sort()
        num = 0
        start = len(s)-1
        for i in range(len(g)-1,-1,-1):
            if start>=0 and g[i]<=s[start]:
                start -= 1
                num += 1
        return num
```

376. Wiggle Subsequence



保留波峰的元素，删掉不处于波峰的元素。

局部最优：删除单调坡度上的节点（不包括单调坡度两端的节点），那么这个坡度就可以有两个局部峰值。

整体最优：整个序列有最多的局部峰值，从而达到最长摆动序列。

如何判断是波峰：当前元素-前一个元素，后一个元素-当前元素，若符号不一样则为波峰。比如17,17-1=16,5-17=-12，符号不同->17为波峰；比如10： 10-5=5， 13-10=3，符号相同->10不为波峰。

```
#错误
def wiggleMaxLength(self, nums: List[int]) -> int:
        pre = 0
        cur = 0
        cnt = 1
```

```
        for i in range(1,len(nums)-1):
            pre = nums[i]-nums[i-1]
            cur = nums[i+1]-nums[i]
            if (pre>0 and cur<=0) or (pre<0 and cur>=0):
                cnt += 1
        return cnt


#正确
class Solution:
    def wiggleMaxLength(self, nums: List[int]) -> int:
        if len(nums)<=1:
            return len(nums)
        pre = 0
        cur = 0
        cnt = 1
        for i in range(1,len(nums)):
            cur = nums[i]-nums[i-1]
            if (pre>=0 and cur<0) or (pre<=0 and cur>0):
                cnt += 1
                pre = cur
        return cnt
```

本题也可以使用动态规划来解决，其实类似于股票问题，用状态机思想解决。

```
class Solution:
    def wiggleMaxLength(self, nums: List[int]) -> int:
        # 0 i 作为波峰的最大长度
        # 1 i 作为波谷的最大长度
        # dp是一个列表，列表中每个元素是长度为 2 的列表
        dp = []
        for i in range(len(nums)):
            # 初始为[1, 1]
            dp.append([1, 1])
            for j in range(i):
                # nums[i] 为波谷
                if nums[j] > nums[i]:
                    dp[i][1] = max(dp[i][1], dp[j][0] + 1)
                # nums[i] 为波峰
                if nums[j] < nums[i]:
                    dp[i][0] = max(dp[i][0], dp[j][1] + 1)
        return max(dp[-1][0], dp[-1][1])
```

122.Best Time to Buy and Sell Stock II

之前讨论过，可以用动态规划状态机来解决

```
class Solution:
    def maxProfit(self, prices: List[int]) -> int:
        # 可以多次买卖
        # dp[i][0]:持有状态
        # dp[i][1]:不持有状态

        dp = [[0 for _ in range(2)]for _ in range(len(prices))]
        dp[0][0] = -prices[0]
        dp[0][1] = 0
        for i in range(1,len(prices)):
            dp[i][0] = max(dp[i-1][0],dp[i-1][1]-prices[i])
            dp[i][1] = max(dp[i-1][1],dp[i-1][0]+prices[i])
        return dp[len(prices)-1][1]
```

现在尝试用贪心算法:**只收集正利润**

```
class Solution:
    def maxProfit(self, prices: List[int]) -> int:
        #贪心算法 只收集正利润
        res = 0
        for i in range(1,len(prices)):
            #if(prices[i]-prices[i-1]>0):
            #   res += prices[i]-prices[i-1]
            res += max(prices[i]-prices[i-1],0)
        return res
```

860. Lemonade Change

```
class Solution:
    def lemonadeChange(self, bills: List[int]) -> bool:
        #碰到10 找一张5
        #碰到20 优先找10 + 5

        cnt_5 = 0
        cnt_10 = 0
        cnt_20 = 0
        for i in range(len(bills)):
            if bills[i] == 5:
                cnt_5 += 1
```

```
            elif bills[i] == 10:
                if cnt_5 < 1:
                    return False
                cnt_5 -= 1
                cnt_10 += 1
            elif bills[i] == 20:
                if cnt_10 > 0 and cnt_5 > 0:
                    cnt_5 -= 1
                    cnt_10 -= 1
                elif cnt_10 == 0 and cnt_5 >= 3:
                    cnt_5 -= 3
                else:
                    return False
        return True
```

\1005. Maximize Sum Of Array After K Negations

```
class Solution:
    def largestSumAfterKNegations(self, nums: List[int], k: int) -> int:
        # 优先将绝对值大的负的变成正的
        # 其次把负的变成正的
        # 操作完以后，要是有0就在0上操作
        # 要是k剩单数，就在最小的正数上操作
        # 要是k剩双数，就随意在一个数上操作
        nums.sort()
        for i in range(len(nums)):
            if k > 0 and nums[i] < 0:
                nums[i] *= -1
                k -= 1
        # k没用完
        while k > 0:
            min_num_idx = nums.index(min(nums))
            nums[min_num_idx] *= -1
            k -= 1

        return sum(nums)
```

\406. Queue Reconstruction by Height

题目没看懂

\135. Candy（重要）

要是围城一个环怎么做？https://github.com/lwlpyl/ByteDance/blob/master/test190316_03.c

```
class Solution:
    def candy(self, ratings: List[int]) -> int:
        nums = [1] * len(ratings)

        #左规则
        for i in range(1,len(nums)):
            if ratings[i] > ratings[i-1]:
                nums[i] = nums[i-1] + 1

        #右规则
        for j in range(len(nums)-2,-1,-1):
            if ratings[j] > ratings[j+1]:
                nums[j] = max(nums[j], nums[j+1]+1)


        return sum(nums)
```

区间问题

55. Jump Game

i每次移动只能在cover的范围内移动，每移动一个元素，cover得到该元素数值（新的覆盖范围）的补充，让i继续移动下去。

而cover每次只取 max(该元素数值补充后的范围, cover本身范围)。

如果cover大于等于了终点下标，直接return true就可以了。

```python
class Solution:
    def canJump(self, nums: List[int]) -> bool:
        cover = 0
        i = 0
        while i<=cover:
            cover = max(cover,i+nums[i])
            if cover >= len(nums)-1:
                return True
            i += 1
        return False
```

45. Jump Game II

1. 维护几个变量：当前所能达到的最远位置 `end`，下一步所能跳到的最远位置 `max_pos`，最少跳跃次数 `setps`。
2. 遍历数组 `nums` 的前 `len(nums) - 1` 个元素：
    1. 每次更新第 `i` 位置下一步所能跳到的最远位置 `max_pos`。
    2. 如果索引 `i` 到达了 `end` 边界，则：更新 `end` 为新的当前位置 `max_pos`，并令步数 `setps` 加 `1`。
3. 最终返回跳跃次数 `steps`。

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

本题也可以使用dp算法，很像题目300，不过常规的dp会超时

```python
class Solution:
    def jump(self, nums: List[int]) -> int:
        dp = [inf] * len(nums)
        dp[0] = 0
        for i in range(1,len(nums)):
            for j in range(0,i):
                if j + nums[j] >= i:
                    dp[i] = min(dp[i],dp[j]+1)
        return dp[len(nums)-1]
```

使用贪心思想修改一下上述的dp

因为本题的数据规模为 10^4，而思路 1 的时间复杂度是 $O(n^2)$，所以就超时了。那么我们有什么方法可以优化一下，减少一下时间复杂度吗?

上文提到，在满足 `j + nums[j] >= i` 的情况下，`dp[i] = min(dp[i], dp[j] + 1)`。

通过观察可以发现，`dp[i]` 是单调递增的，也就是说 `dp[i - 1] <= dp[i] <= dp[i + 1]`。

举个例子，比如跳到下标 `i` 最少需要 `5` 步，即 `dp[i] = 5`，那么必然不可能出现少于 `5` 步就能跳到下标 `i + 1` 的情况，跳到下标 `i + 1` 至少需要 `5` 步或者更多步。

既然 `dp[i]` 是单调递增的，那么在更新 `dp[i]` 时，我们找到最早可以跳到 `i` 的点 `j`，从该点更新 `dp[i]`。即找到满足 `j + nums[j] >= i` 的第一个 `j`，使得 `dp[i] = dp[j] + 1`。

而查找第一个 `j` 的过程可以通过使用一个指针变量 `j` 从前向后迭代查找。

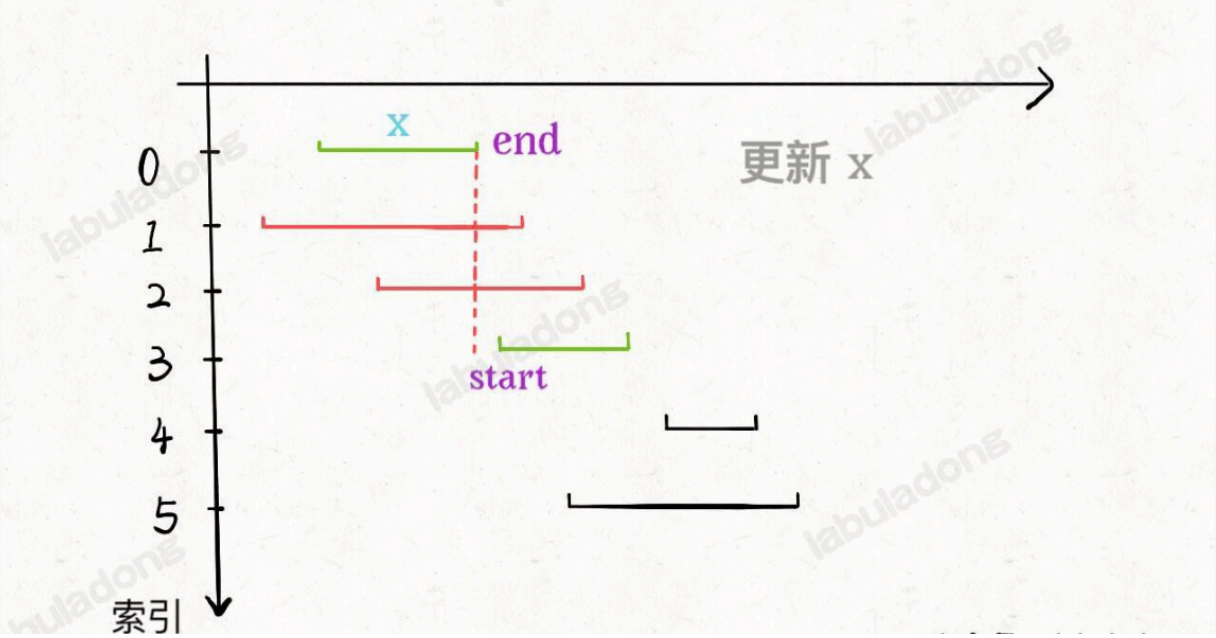
最后，将最终结果 `dp[size - 1]` 返回即可。

```python
class Solution:
    def jump(self, nums: List[int]) -> int:
        dp = [inf] * len(nums)
        dp[0] = 0
        j = 0
        for i in range(1,len(nums)):
            while j + nums[j] < i:
                j += 1
            dp[i] = dp[j]+1

        return dp[len(nums)-1]
```

435. Non-overlapping Intervals

可以看看这道题：354.Russian Doll Envelopes

> 区间调度问题是让你计算若干区间中最多有几个互不相交的区间，这道题是区间调度问题的一个简单变体。
>
> 区间调度问题思路可以分为以下三步：
>
> 1、从区间集合 `intvs` 中选择一个区间 `x`，这个 `x` 是在当前所有区间中**结束最早的**（`end` 最小）。
>
> 2、把所有与 `x` 区间相交的区间从区间集合 `intvs` 中删除。
>
> 3、重复步骤 1 和 2，直到 `intvs` 为空为止。之前选出的那些 `x` 就是最大不相交子集。
>
> 
>

```python
class Solution:
    def eraseOverlapIntervals(self, intervals: List[List[int]]) -> int:
        intervals.sort(key = lambda x: x[1]) #按第二维的升序排列
        cnt = 1 #统计有几个不重复的区间
        x_end = intervals[0][1]
        for i in intervals:
            if i[0] >= x_end:
                cnt += 1
                x_end = i[1]
        return len(intervals)-cnt #删除的个数 = 总-不重复的
```

452. Minimum Number of Arrows to Burst Balloons

452. Minimum Number of Arrows to Burst Balloons

基本同上一题，除了把>=变成>

```python
class Solution:
    def findMinArrowShots(self, points: List[List[int]]) -> int:
        points.sort(key = lambda x: x[1])
        cnt = 1
        x_end = points[0][1]
        for i in points:
            if i[0] > x_end:
                cnt += 1
                x_end = i[1]
        return cnt
```

56. Merge Intervals

也是区间调度问题，本题需要按start排序

```python
class Solution:
    def merge(self, intervals: List[List[int]]) -> List[List[int]]:
        intervals.sort(key=lambda x: x[0])
        merged = []
        for interval in intervals:
            # 如果列表为空，或者当前区间与上一区间不重合，直接添加
            if not merged or merged[-1][1] < interval[0]:
                merged.append(interval)
            else:
                # 否则的话，我们就可以与上一区间进行合并
                #因为是对start排序，所以并不知道end谁大，不同于之前的题目
                merged[-1][1] = max(merged[-1][1], interval[1])

        return merged
```

763. Partition Labels

与jump game2 非常相似

由于同一个字母只能出现在同一个片段，显然同一个字母的第一次出现的下标位置和最后一次出现的下标位置必须出现在同一个片段。因此需要遍历字符串，得到每个字母最后一次出现的下标位置。

在得到每个字母最后一次出现的下标位置之后，可以使用贪心的方法将字符串划分为尽可能多的片段，具体做法如下。

- 从左到右遍历字符串，遍历的同时维护当前片段的开始下标 $start$ 和结束下标 end，初始时 $start = end = 0$。
- 对于每个访问到的字母 c，得到当前字母的最后一次出现的下标位置 end_c，则当前片段的结束下标一定不会小于 end_c，因此令 $end = \max(end, end_c)$。
- 当访问到下标 end 时，当前片段访问结束，当前片段的下标范围是 $[start, end]$，长度为 $end - start + 1$，将当前片段的长度添加到返回值，然后令 $start = end + 1$，继续寻找下一个片段。
- 重复上述过程，直到遍历完字符串。

上述做法使用贪心的思想寻找每个片段可能的最小结束下标，因此可以保证每个片段的长度一定是符合要求的最短长度，如果取更短的片段，则一定会出现同一个字母出现在多个片段中的情况。由于每次取的片段都是符合要求的最短的片段，因此得到的片段数也是最多的。

由于每个片段访问结束的标志是访问到下标 end，因此对于每个片段，可以保证当前片段中的每个字母都一定在当前片段中，不可能出现在其他片段，可以保证同一个字母只会出现在同一个片段。

```python
class Solution:
    def partitionLabels(self, s: str) -> List[int]:
        res = []
        h = defaultdict(int)
        start, end = 0, 0

        #用哈希表h保存每个元素出现最远的位置
        for i in range(len(s)):
            h[s[i]] = i

        for i in range(len(s)):
            end = max(end,h[s[i]])
            #要是当前位置是当前字母下标最远的位置
            if end == i:
                res.append(end-start+1)
                start = i+1

        return res
```