



In [8]:

```
1  ## 快速排序
2  quick_sort = lambda array: array if len(array) <= 1 else quick_sort([item for item in array[
3      [array[0]]+quick_sort([item for item in array[1:] if item > array[0]])
4
5  # 快速排序 O(logn)
6  def quick_sort(array):
7      if len(array) < 2:
8          return array
9      mid = array[len(array)//2]
10     left,right = [],[]
11     array.remove(mid)
12     for item in array:
13         if item >= mid:
14             right.append(item)
15         else:
16             left.append(item)
17     return quick_sort(left)+[mid]+quick_sort(right)
18
19 # 冒泡排序O(n^2)
20 def bs(L):
21     for i in range(len(L)):
22         for j in range(i, len(L)):
23             if L[i] > L[j]:
24                 L[i],L[j] = L[j],L[i]
25     return L
26
27 # 插入排序 O(n^2)-容易超时
28 def insertionSort(arr):
29     for i in range(len(arr)):
30         preIndex = i-1
31         current = arr[i] # 跟前面的比较
32         while preIndex >= 0 and arr[preIndex] > current:
33             arr[preIndex+1] = arr[preIndex]
34             preIndex-=1
35         arr[preIndex+1] = current # 此时满足条件
36     return arr
37
38
39 # 归并排序
40 def sortArray(self, nums):
41     temp = [0]*len(nums)
42     def mergeSort(l,r):
43         if l >= r : return
44         mid = (l+r) // 2
45         mergeSort(l,mid)
46         mergeSort(mid+1,r)
47         i,j = l,mid+1
48         k = 0
49         while i <= mid and j <= r:
50             if nums[i] < nums[j]:
51                 temp[k] = nums[i]
52                 i += 1
53             else:
54                 temp[k] = nums[j]
55                 j += 1
56             k += 1
57         while i <= mid:
```

2021/7/18

ToBeACoder

```
58         temp[k] = nums[i]
59         i += 1
60         k += 1
61     while j <= r:
62         temp[k] = nums[j]
63         j += 1
64         k += 1
65     nums[l:r+1] = temp[r-l+1:]
66 mergeSort(0, len(nums)-1)
67 return nums
```

动态规划

二维 dp . $dp[i][j]$
计算来自前一行

$i-1$

i

$i+1$

- 维时 dp 相当于 2 维的堆叠和
不断刷新.

$i-1$

i

$i+1$

$\square = \square$,
所以前值先不可改变!

如果 $\square \neq \square$
改变了, 则不能代
表前行了!

所以要后序.

In [2]:

```
1  ## 0-1背包问题，理论篇
2  ## 二维
3  W = 4
4  wt = [2, 1, 3]
5  val = [4, 2, 3]
6  def knapsack2(W, wt, val):
7      N = len(wt)
8      dp = [[0]*(W+1) for _ in range(N+1)] # base case 已经初始化，填充多了一行
9      # dp[i][j] 表示： 从0到i中选择，背包当前容量为j时的价值
10     for i in range(1, N+1):
11         for w in range(1, W+1):
12             if w < wt[i-1]: # wt和val要取i-1才能对应上
13                 dp[i][w] = dp[i-1][w] # 如果到一维情况，那么表示不变dp[w]=dp[w]
14             else:
15                 dp[i][w] = max(dp[i-1][w], dp[i-1][w-wt[i-1]]+val[i-1])
16     return dp[N][W]
17
18 ## 一维
19 ## dp 表示二维数组dp的i-1行，一维二维中dp[i]行只用到了dp[i-1]行的东西，所以可复用来更新
20 ## 而必须后序更新是避免重复运用
21
22 def knapsack1(W, wt, val):
23     N = len(wt)
24     dp = [0]*(W+1) # base case 已经初始化，填充多了一行
25     # dp[j] 表示： 背包当前容量为j时的价值
26     for i in range(N): # 从0开始
27         for w in range(W, wt[i]-1, -1): # W-W-1-...-wt[i]
28             dp[w] = max(dp[w], dp[w-wt[i]]+val[i]) # 覆盖
29     return dp[-1]
30 knapsack2(W, wt, val) # 取第一个和第二个
```

Out[2]:

6

In []:

```

1  ### 416 等分割子集
2  def canPartition( nums):
3      # 二维
4      sumnums = sum(nums)
5      if sumnums %2 != 0: return False # 先判断再处理
6      n = len(nums)
7      sumnums = int(sumnums/2)
8      dp = [[False]*(sumnums+1) for _ in range(n+1)]
9      for i in range(n+1): dp[i][0] = True # 背包装满了
10     for i in range(1,n+1):
11         for j in range(1,sumnums+1):
12             if j < nums[i-1] :
13                 # 放不下
14                 dp[i][j] = dp[i-1][j]
15             else:
16                 # 放得下, 但是放还是不放
17                 dp[i][j] = dp[i-1][j] or dp[i-1][j-nums[i-1]]
18     return dp[n][-1]
19
20
21     # 一维
22     sumnums = sum(nums)
23     n = len(nums)
24     w = sumnums//2
25     if sumnums % 2 != 0:
26         return False
27     dp = [False] * (w+1)
28     dp[0] = True
29     for num in nums:
30         for i in range(num,w+1)[:::-1]:
31             dp[i] = dp[i] or dp[i - num] # 不断更新在不同w情况下的dp
32     return dp[w] # 定义为是否能放下该重量
33
34
35     ## 套模板, 更容易理解
36     # 背包大小sum//2, 物品的价值=物品的重量
37     dp = [0]*(w+1)
38     for i in range(n):
39         for j in range(w, nums[i]-1, -1):
40             # num[i]会大过背包重量, 但是这是不会有任何操作
41             dp[j] = max(dp[j], dp[j-nums[i]]+nums[i])
42     return dp[w] == w
43
44 # 1049 最后一块石头
45 def lastStoneWeightII(stones):
46     n = len(stones)
47     target = sum(stones)//2
48     dp = [0]*(target+1)
49     for i in range(n):
50         for j in range(target, stones[i]-1, -1):
51             dp[j] = max(dp[j], dp[j-stones[i]]+stones[i])
52     # (sum(stones)-dp[-1])-dp[-1] (一堆<平均, 一堆>平均, 具体为多少未知)
53     return sum(stones)-2*dp[-1]
54
55 # 494 目标和 改变数组元素的+-
56 def findTargetSumWays(nums, target):
57     # 公式推导可得, 不变的排前, 变负的排后

```

```

58 # left-right=target, left+right=sum, left=(target+sum)/2
59 # 回溯, 不去重的组合求和III, 不去重相对于考虑了位置——超时
60 # 和全排列不同——不要求全部加入(拿捏不准要多少个元素)。
61 # 和组合求和III(有重复, 但不可复用自己(同), 但是不考虑顺序)不同, 要求考虑位置。
62 # 动态规划
63 # 背包容量target, nums[i]即重量和价值, 重量和<=target
64 Sum = sum(nums)
65 if target > Sum: return 0
66 target, m = (target+Sum)//2, (target+Sum)%2
67 if m != 0: return 0
68 dp = [0]*(target+1)
69 dp[0] = 1
70 for i in range(len(nums)):
71     for j in range(target, nums[i]-1, -1):
72         dp[j] += dp[j-nums[i]]
73 return dp[-1]
74
75 # 474 0和1, 不可复用
76 def findMaxForm(strs, m, n):
77     # 二维背包问题, n:1行m:0列
78     if len(strs) == 0:
79         return 0
80     dp = [[0] * (m+1) for _ in range(n+1)]
81     for str_ in strs:
82         # 外物
83         count_0 = str_.count('0')
84         count_1 = str_.count('1')
85         # for i in range(start, end, 步长)
86         # 0-1背包放反序, 内包
87         for i in range(n, count_1-1, -1):
88             for j in range(m, count_0-1, -1):
89                 dp[i][j] = max(dp[i][j], dp[i-count_1][j-count_0]+1)
90     return dp[n][m]
91
92
93
94 # 完全背包
95 # 518 零钱兑换II
96 def change(amount, coins):
97     # 完全背包
98     dp = [0]*(amount+1)
99     dp[0] = 1
100     for i in range(len(coins)):
101         for j in range(coins[i], amount+1):
102             dp[j] += dp[j-coins[i]]
103     return dp[-1]
104
105 # 377 组合总和IV
106 def combinationSum4(nums, target):
107     dp = [0]*(target+1)
108     dp[0]=1
109     for i in range(1, target+1): # 背包
110         for j in range(len(nums)): # 物
111             if i >= nums[j]:
112                 dp[i] += dp[i-nums[j]]
113     return dp[-1]
114
115 # 70爬楼梯 排列数
116 def climbStairs(self, n: int) -> int:
117     dp = [0]*(n + 1)
118     dp[0] = 1

```

```

119     m = 2
120     for j in range(n + 1):
121         # 遍历背包
122         for step in range(1, m + 1):
123             # 遍历物品
124             if j >= step:
125                 dp[j] += dp[j - step]
126     return dp[n]
127
128 # 322 零钱兑换I, 完全背包, 摆列组合都可以
129 def coinChange(coins, amount):
130     dp = [amount+1]*(amount+1)
131     dp[0] = 0
132     # for i in range(1, amount+1):
133     #     for coin in coins:
134     #         if i < coin: continue
135     #         dp[i] = min(dp[i], dp[i-coin]+1) # 记得加一
136     # return dp[-1] if dp[-1] <= amount else -1
137     for coin in coins:
138         for i in range(coin, amount+1):
139             # 少了一行判断
140             dp[i] = min(dp[i], dp[i-coin]+1) # 记得加一
141     return dp[-1] if dp[-1] <= amount else -1
142
143 # 279 完全平方数
144 def numSquares(n):
145     # 完全背包, 正序
146     # 最少个数, 组合排列都可以
147     nums = [i**2 for i in range(1, n + 1) if i**2 <= n] # 构造物
148     dp = [10**4]*(n+1)
149     dp[0] = 0
150     for i in range(len(nums)):
151         for j in range(nums[i], n+1):
152             dp[j] = min(dp[j], dp[j-nums[i]]+1)
153     return dp[-1]
154
155 # 单词拆分——
156 def wordBreak(s, wordDict):
157     n = len(s)
158     dp = [False]*(n+1)
159     dp[0] = True
160     for i in range(1, n+1):
161         for word in wordDict:
162             lenw = len(word)
163             if lenw <= i and word == s[i-lenw:i]:
164                 dp[i] = dp[i] or dp[i-lenw]
165     return dp[n]
166
167
168 # 198 打家劫舍I
169 def rob(self, nums: List[int]) -> int:
170     if nums == []:
171         return 0
172     n = len(nums)
173     dp = [0]*(n+2)
174     for i in range(2, n+2):
175         dp[i] = max(dp[i-1], dp[i-2] + nums[i-2])
176     return dp[-1]
177
178 # 213 打家劫舍II
179 def rob(self, nums: List[int]) -> int:

```

```
180     if nums == []:
181         return 0
182     if len(nums)==1:
183         return nums[0]
184     def robsub(nums):
185         if nums == []:
186             return 0
187         n = len(nums)
188         dp = [0]*(n+2)
189         for i in range(2,n+2):
190             dp[i] = max(dp[i-1], dp[i-2] + nums[i-2])
191         return dp[-1]
192     return max( robsub(nums[1:]), robsub(nums[:-1]) )
193
194 # 337 打家劫舍III
195 class Solution:
196     # rember
197     rob_dict = {}
198     def rob(self, root: TreeNode) -> int:
199         if not root:
200             # root == None
201             return 0
202         if root in self.rob_dict.keys():
203             return self.rob_dict[root]
204         if not root.left and not root.right:
205             # 到达这个点在犹豫，没有跳过，那么直接取是最大的
206             self.rob_dict[root] = root.val
207             return root.val
208         # 如果都不是，那么必然存在左子树或右子树
209         get = root.val
210         if root.right:
211             get += self.rob(root.right.right)+self.rob(root.right.left)
212         if root.left:
213             get += self.rob(root.left.right)+self.rob(root.left.left)
214         not_get = self.rob(root.left)+self.rob(root.right)
215         res = max(get, not_get)
216         self.rob_dict[root] = res
217         return res
```

In [3]:

```
1  ### 多重背包理论篇-->转成0-1背包问题
2  def multi_pack1(weight, values, nums, bag_weight):
3      # 将物品展开, 将nums消除掉
4      for i in range(len(nums)):
5          while nums[i]:
6              weight.append(weight[i])
7              value.append(value[i])
8              nums[i] -= 1
9      dp = [0]*(bag_weight+1)
10     for i in range(len(weight)):
11         for j in range(bag_weight, weight[i]-1, -1):
12             dp[j] = max(dp[j], dp[j-weight[i]]+value[i])
13     print(" ".join(map(str, dp)))
14
15     weight = [1, 3, 4]
16     values = [15, 20, 30]
17     nums = [2, 3, 2]
18     bag_weight = 10
19     multi_pack1(weight, values, nums, bag_weight)
```

0 15 30 45 45 50 65 75 75 85 95



stock

In []:

```

1 # dp[i][j] 表示第i天利润
2 ##### 121 一次买入，一次卖出，同一天买卖没有收益，k=1
3 def maxProfit(self, prices: List[int]) -> int:
4
5     # dp[i][j] i 表示 天数, j 表示 1持有 or 0出售
6     n = len(prices)
7     dp = [[0]*2 for i in range(n)]
8     dp[0][0] = 0
9     dp[0][1] = -prices[0] # 现金流
10    for i in range(1,n):
11        dp[i][0] = max(dp[i-1][0], dp[i-1][1]+prices[i]) # 手头没有股票: 1. 一直没有 2. 刚卖
12        dp[i][1] = max(dp[i-1][1], -prices[i]) # 手头有股票: 1. 一直有 2. 刚买, 前面没有利润
13    return dp[-1][0]
14
15 ##### 122 多次买卖 k = +inf
16 def maxProfit(self, prices: List[int]) -> int:
17     # 捕获上坡那一段即可, 贪心
18     res = 0
19     for i in range(1, len(prices)):
20         res += max(prices[i]-prices[i-1], 0)
21     return res
22
23     # 动态规划, 0表示不持有, 1表示持有
24     n = len(prices)
25     dp = [[0]*2 for _ in range(n)]
26     # dp[0][0] = 0
27     dp[0][1] = -prices[0]
28     for i in range(1,n):
29         dp[i][0] = max(dp[i-1][0], dp[i-1][1]+prices[i]) # 不持有(以前没有or刚卖)
30         dp[i][1] = max(dp[i-1][1], dp[i-1][0]-prices[i]) # 持有(以前有or刚买)
31     return dp[-1][0]
32
33     # dp 加滚动数组
34     n = len(prices)
35     dp = [[0]*2 for _ in range(2)] # 空间压缩
36     # dp[0][0] = 0
37     dp[0][1] = -prices[0]
38     for i in range(1,n):
39         dp[i%2][0] = max(dp[(i-1)%2][0], dp[(i-1)%2][1]+prices[i]) # 不持有(以前没有or刚卖)
40         dp[i%2][1] = max(dp[(i-1)%2][1], dp[(i-1)%2][0]-prices[i]) # 持有(以前有or刚买)
41     return dp[(n-1)%2][0]
42
43 ##### 123 两次买卖 k=2
44 def maxProfit(self, prices: List[int]) -> int:
45     # 5种状态, 0 没有操作 1 第一次买入 2 第一次卖出 3 第二次买入 4 第二次卖出
46     n = len(prices)
47     if n == 0: return 0
48     dp = [[0]*5 for i in range(n)]
49     dp[0][1] = -prices[0]
50     dp[0][3] = -prices[0] # 在第0天已经做了一次买卖, 但是0收益(同一天买卖了)
51     for i in range(1,n):
52         # dp[i][0] = dp[i-1][0] # 没操作
53         dp[i][1] = max(dp[i-1][0]-prices[i], dp[i-1][1]) # 保持 or 之前没有刚第一次买
54         dp[i][2] = max(dp[i-1][1]+prices[i], dp[i-1][2]) # 保持 or 之前有刚第一次卖
55         dp[i][3] = max(dp[i-1][2]-prices[i], dp[i-1][3]) # 保持 or 之前没有刚第二次买
56         dp[i][4] = max(dp[i-1][3]+prices[i], dp[i-1][4]) # 保持 or 之前有刚第二次卖
57     return dp[-1][4]

```

```

58
59 ##### 188 k次买卖——思路来源于123 k = any int
60 def maxProfit(self, k: int, prices: List[int]) -> int:
61     # 观察两次买卖
62
63     n = len(prices)
64     if n == 0: return 0
65     dp = [[0]*(2*k+1) for i in range(n)]
66     for j in range(1, 2*k, 2):
67         dp[0][j] = -prices[0] # 在第0天已经做了一次买卖，但是0收益(同一天买卖了)
68     for i in range(1, n):
69         for j in range(1, 2*k, 2):
70             dp[i][j] = max(dp[i-1][j-1]-prices[i], dp[i-1][j]) # 保持 or 之前没有刚第一次买
71             dp[i][j+1] = max(dp[i-1][j]+prices[i], dp[i-1][j+1]) # 保持 or 之前有刚第一次卖
72             # dp[i][3] = max(dp[i-1][2]-prices[i], dp[i-1][3]) # 保持 or 之前没有刚第二次买
73             # dp[i][4] = max(dp[i-1][3]+prices[i], dp[i-1][4]) # 保持 or 之前有刚第二次卖
74     return dp[-1][2*k]
75
76 ##### 309 k = +inf and cold -- 把122中的dp[i-1][0]-prices[i]改成dp[i-2][0]-prices[i]即可
77 def maxProfit(self, prices: List[int]) -> int:
78     n = len(prices)
79     dp = [[0]*2 for _ in range(n)]
80     dp[0][1] = -prices[0]
81     for i in range(1, n):
82         dp[i][0] = max(dp[i-1][0], dp[i-1][1]+prices[i]) # 不持有(以前没有or刚卖)
83         dp[i][1] = max(dp[i-1][1], dp[i-2][0]-prices[i]) # 持有(以前有or刚买)
84     return dp[-1][0]
85
86
87 ##### 714
88 def maxProfit(self, prices: List[int], fee: int) -> int:
89     n = len(prices)
90     dp = [[0]*2 for _ in range(n)]
91     dp[0][1] = -prices[0]
92     for i in range(1, n):
93         dp[i][0] = max(dp[i-1][0], dp[i-1][1]+prices[i]-fee) # 122中多减去一个手续费
94         dp[i][1] = max(dp[i-1][1], dp[i-1][0]-prices[i])
95     return dp[-1][0]

```

子序列

In []:

```

1 ##### 300 最长递增子序列
2 def lengthOfLIS(self, nums: List[int]) -> int:
3     n = len(nums)
4     dp = [1]*n #dp[i] 表示以i结尾的最长子序列长度
5     for i in range(1,n):
6         for j in range(i):
7             if nums[i] > nums[j]: # 符合递增, 找前面比他小的数的dp+1
8                 dp[i] = max(dp[i], dp[j]+1) # 一直找一直找
9     return max(dp)
10
11 ##### 674 最长递增子串
12 def findLengthOfLCIS(self, nums: List[int]) -> int:
13     n = len(nums)
14     dp = [1]*n #dp[i] 表示以i结尾的最长子串长度
15     for i in range(1,n):
16         if nums[i] > nums[i-1]:
17             dp[i] = dp[i-1]+1
18     return max(dp)
19
20 ##### 718 最长重复子串
21 def findLength(self, nums1: List[int], nums2: List[int]) -> int:
22     # 子串
23     n = len(nums1)
24     m = len(nums2)
25     res = 0
26     dp = [[0]*(m+1) for _ in range(n+1)] # 以i-1为结尾的nums1, 以j-1为结尾的nums2的最长重复子
27     for i in range(1,n+1):
28         for j in range(1,m+1):
29             if nums1[i-1] == nums2[j-1]: # 错位对齐
30                 dp[i][j] = dp[i-1][j-1]+1 # 不相等即为0
31             res = max(res, dp[i][j])
32     return res
33
34 ##### 1143 最长公共子序列
35 def longestCommonSubsequence(self, text1: str, text2: str) -> int:
36     m = len(text1)
37     n = len(text2)
38     if n == 0 or m == 0:
39         return 0
40     dp = [[0]*(n+1) for _ in range(m+1)] # 定义二维dp table # 以i-1为结尾的text1, 以j-1为结
41     for i in range(1,m+1): # i, j对应谁要对应上
42         for j in range(1,n+1):
43             if text1[i-1] == text2[j-1]:
44                 dp[i][j] = dp[i-1][j-1]+1
45             else:
46                 dp[i][j] = max(dp[i-1][j], dp[i][j-1])
47     return dp[-1][-1]
48
49
50 ### 53 最大子序号, 贪心的另一种解法
51 def maxSubArray(self, nums: List[int]) -> int:
52     n = len(nums)
53     dp = [0]*n
54     dp[0] = nums[0]
55     for i in range(1,n):
56         dp[i] = max(dp[i-1]+nums[i], nums[i])
57     return max(dp)

```

```

58
59 ### 392 判断子序列，有顺序，不能用hashmap表
60 def isSubsequence(self, s: str, t: str) -> bool:
61     # 双指针
62     p1, p2 = 0, 0
63     while p1 < len(s) and p2 < len(t):
64         if s[p1] == t[p2]:
65             p1 += 1
66             p2 += 1
67     if p1 == len(s): return True # p1 最终会跳到len(s)
68     return False
69
70 # 动态规划解法
71 n = len(s)
72 m = len(t)
73 dp = [[0]*(m+1) for _ in range(n+1)]
74 for i in range(1, n+1):
75     for j in range(1, m+1):
76         if s[i-1] == t[j-1]:
77             dp[i][j] = dp[i-1][j-1] + 1
78         else:
79             dp[i][j] = dp[i][j-1]
80     return dp[-1][-1] == n
81
82
83 ### 115 不同的序列
84 def numDistinct(self, s: str, t: str) -> int:
85     # t 是字典
86     n = len(s)
87     m = len(t)
88     dp = [[0]*(m+1) for _ in range(n+1)]
89     for i in range(n+1):
90         dp[i][0] = 1
91     for j in range(1, m+1):
92         dp[0][j] = 0
93     for i in range(1, n+1):
94         for j in range(1, m+1):
95             if s[i-1] == t[j-1]:
96                 # 删除这一个可能存在匹配个数，不加入
97                 dp[i][j] = dp[i-1][j-1] + dp[i-1][j]
98             else:
99                 # 不加入这一个
100                 dp[i][j] = dp[i-1][j]
101     return dp[-1][-1]
102
103 ##### 583 两个字符串的删除
104 def minDistance(self, word1: str, word2: str) -> int:
105     n = len(word1)
106     m = len(word2)
107     dp = [[0]*(m+1) for _ in range(n+1)]
108     for i in range(n+1):
109         dp[i][0] = i
110     for j in range(1, m+1):
111         dp[0][j] = j
112     for i in range(1, n+1):
113         for j in range(1, m+1):
114             if word1[i-1] == word2[j-1]:
115                 dp[i][j] = dp[i-1][j-1]
116             else:
117                 dp[i][j] = min(dp[i-1][j]+1, dp[i][j-1]+1, dp[i-1][j-1]+2) # 下标-1 为删除操作
118     return dp[-1][-1]

```

```

119
120 ##### 72 编辑距离
121 def minDistance(self, word1: str, word2: str) -> int:
122     n = len(word1)
123     m = len(word2)
124     dp = [[0]*(m+1) for _ in range(n+1)]
125     for i in range(n+1):
126         dp[i][0] = i
127     for j in range(m+1):
128         dp[0][j] = j
129     for i in range(1,n+1):
130         for j in range(1,m+1):
131             if word1[i-1] == word2[j-1]: # word和dp错位一格
132                 dp[i][j] = dp[i-1][j-1]
133             else:
134                 dp[i][j] = min(dp[i-1][j]+1, dp[i][j-1]+1, dp[i-1][j-1]+1)
135     return dp[-1][-1]
136
137 ##### 647 回文子串
138 def countSubstrings(self, s: str) -> int:
139     result = 0
140     n = len(s)
141     dp = [[False]*n for _ in range(n)]
142     for i in range(n-1,-1,-1):
143         for j in range(i,n):
144             if s[i] == s[j]:
145                 if j-i <= 1:
146                     dp[i][j] = True
147                     result += 1
148                 elif dp[i+1][j-1]:
149                     dp[i][j] = True
150                     result += 1
151     return result
152
153 ##### 516 最长回文子串
154 def longestPalindromeSubseq(self, s: str) -> int:
155     n = len(s)
156     dp = [[0]*n for _ in range(n)]
157     for i in range(n):
158         dp[i][i] = 1
159     for i in range(n-1,-1,-1):
160         for j in range(i+1,n):
161             if s[i] == s[j]:
162                 dp[i][j] = dp[i+1][j-1]+2
163             else:
164                 dp[i][j] = max(dp[i+1][j], dp[i][j-1])
165     return dp[0][-1]
166
167 ##### 高楼扔鸡蛋
168 def superEggDrop(self, k: int, n: int) -> int:
169     # 单纯备忘录超出时间限制
170     memo = {}
171     def dp(k,n):
172         if k == 1: return n
173         if n == 0: return 0
174         if (k,n) in memo.keys():
175             return memo[(k,n)]
176         res = float('inf')
177         # for i in range(1,n+1):
178         #     # 这种属于线性搜索, 让i 从0 开始到n

```

```
180     # res = min(res,max(dp(k,n-i),dp(k-1,i-1))+1) # res = min(res,max(碎, 没碎)+1)
181     # 替换为二分搜索,就是将i->(0,n) 变成->(mid)
182     lo = 1; hi = n
183     while lo <= hi:
184         # 通过循环不断的改变mid
185         mid = lo+(hi-lo)//2
186         broken = dp(k-1,mid-1)
187         not_broken = dp(k,n-mid)
188         if broken > not_broken:
189             hi = mid - 1 # mid相当于i
190             res = min(res,broken+1)
191         else:
192             lo = mid + 1 # mid相当于i
193             res = min(res,not_broken+1)
194     memo[(k,n)]=res
195     return res
196 return dp(k,n)
197
198 # 换定义
199 dp = [[0]*(n+1) for _ in range(k+1)]
200 m = 0
201 while dp[k][m] < n:
202     m += 1
203     for i in range(1,k+1):
204         dp[i][m] = dp[i][m-1] + dp[i-1][m-1] + 1
205 return m
```

In []:

```
1 ##### 739 每日温度
2 def dailyTemperatures(self, temperatures: List[int]) -> List[int]:
3     #单调栈
4     n = len(temperatures)
5     res = [0]*n # 存放结果
6     stack = [] # 存放索引而非元素值stack
7     for i in range(n-1,-1,-1):
8         while s and temperatures[stack[-1]] <= temperatures[i]:
9             stack.pop()
10        res[i] = 0 if not stack else stack[-1]-i
11        stack.append(i)
12    return res
13
14 ##### 下一个更大的数
15 def nextGreaterElements(self, nums: List[int]) -> List[int]:
16     n = len(nums)
17     res = [0]*n
18     stack = []
19     for i in range(n-1,-1,-1):
20         while stack and stack[-1] <= nums[i]:
21             stack.pop()
22         res[i] = -1 if not stack else stack[-1]
23         stack.append(nums[i])
24    return res
25
26 ##### 503 循环数组，下一个更大的元素
27 def nextGreaterElements(self, nums: List[int]) -> List[int]:
28     n = len(nums)
29     res = [0]*n
30     stack = []
31     for i in range(2*n-1,-1,-1):
32         while stack and stack[-1] <= nums[i%n]:
33             stack.pop()
34         res[i%n] = -1 if not stack else stack[-1]
35         stack.append(nums[i%n])
36    return res
37
38
39 ##### 42 接雨水
40 def trap(self, height: List[int]) -> int:
41     # 暴力求解 (319/320) O(N^2)
42     n = len(height)
43     res = 0
44     for i in range(1,n-1):
45         leftmax, rightmax = 0, 0
46         for j in range(i, n):
47             rightmax = max(rightmax, height[j])
48         for j in range(i, -1, -1):
49             leftmax = max(leftmax, height[j])
50         res += min(leftmax, rightmax) - height[i]
51    return res
52
53 # 暴力+备忘录 (dp全部算完) O(N)
54 n = len(height)
55 if n == 0: return 0
56 res = 0
57 leftmax = [0]*n
```

```
58     rightmax = [0]*n
59     leftmax[0] = height[0]
60     rightmax[-1] = height[-1]
61     for i in range(1,n):
62         leftmax[i] = max(height[i],leftmax[i-1])
63     for i in range(n-2,-1,-1):
64         rightmax[i] = max(height[i],rightmax[i+1])
65     for i in range(1,n-1):
66         res += min(leftmax[i],rightmax[i])-height[i]
67     return res
68
69     # 双指针
70     if not height: return 0
71     n = len(height)
72     left,right = 0,n-1
73     res = 0
74     leftmax,rightmax = height[0],height[-1]
75     while left <= right:
76         leftmax = max(leftmax,height[left])
77         rightmax = max(rightmax,height[right])
78         if leftmax < rightmax:
79             # 管小不管大, 可能存在更大但不在于
80             res += leftmax - height[left]
81             left += 1
82         else:
83             res += rightmax - height[right]
84             right -= 1
85     return res
```