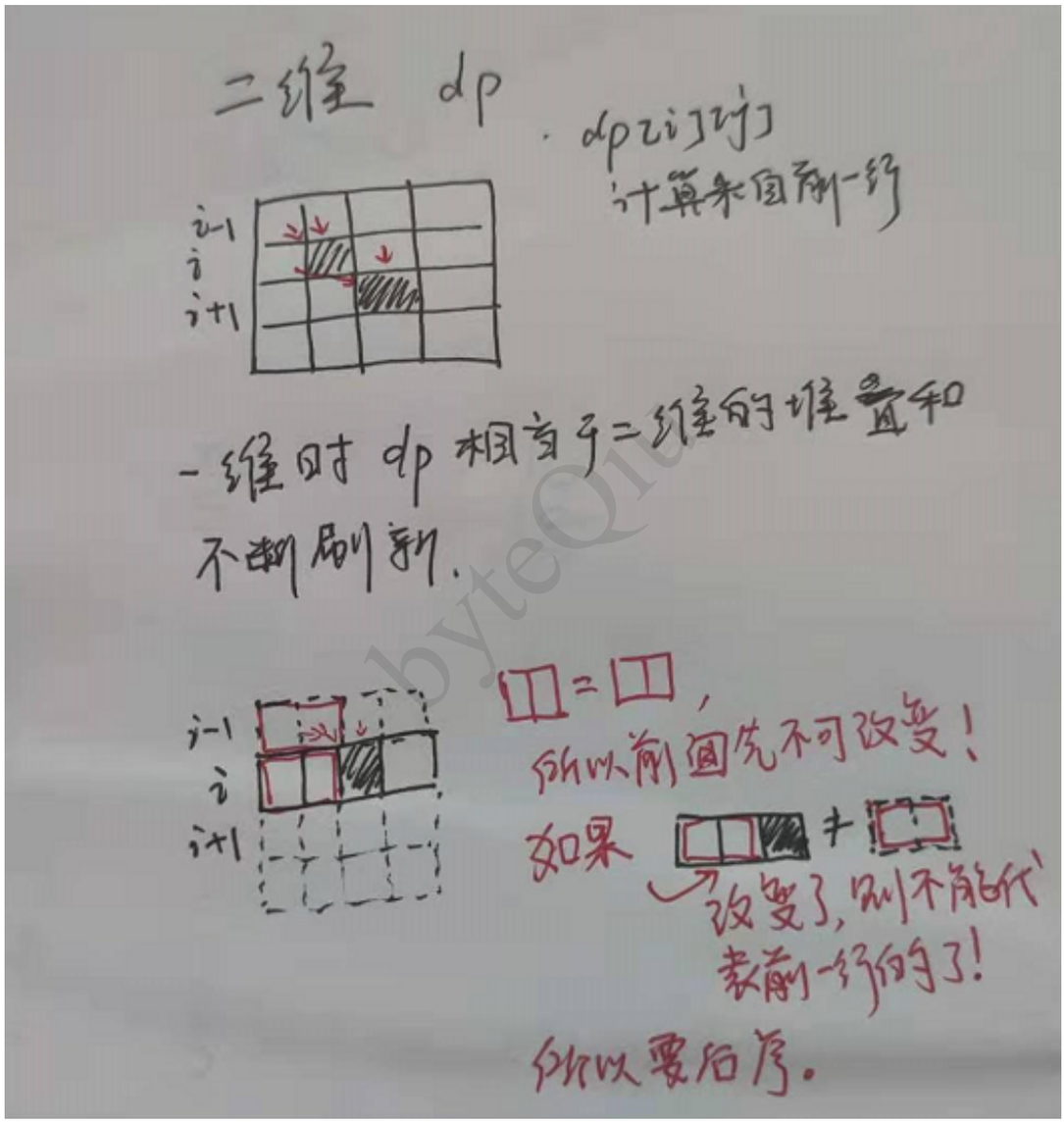
In [8]:

```
## 快速排序
quick_sort = lambda array: array if len(array) <= 1 else quick_sort([item for item in array[
            [array[0]]+quick_sort([item for item in array[1:] if item > array[0]])

# 快速排序 O(logn)
def quick_sort(array):
    if len(array) < 2:
        return array
    mid = array[len(array)//2]
    left,right = [],[]
    array.remove(mid)
    for item in array:
        if item >= mid:
            right.append(item)
        else:
            left.append(item)
    return quick_sort(left)+[mid]+quick_sort(right)

# 冒泡排序O(n^2)
def bs(L):
    for i in range(len(L)):
        for j in range(i,len(L)):
            if L[i] > L[j]:
                L[i],L[j] = L[j],L[i]
    return L

# 插入排序 O(n^2)-容易超时
def insertionSort(arr):
    for i in range(len(arr)):
        preIndex = i-1
        current = arr[i] # 跟前面的比较
        while preIndex >= 0 and arr[preIndex] > current:
            arr[preIndex+1] = arr[preIndex]
            preIndex-=1
        arr[preIndex+1] = current  # 此时满足条件
    return arr


# 归并排序
def sortArray(self, nums):
    temp = [0]*len(nums)
    def mergeSort(l,r):
        if l >= r : return
        mid = (l+r) // 2
        mergeSort(l,mid)
        mergeSort(mid+1,r)
        i,j = l,mid+1
        k = 0
        while i <= mid and j <= r:
            if nums[i] < nums[j]:
                temp[k] = nums[i]
                i += 1
            else:
                temp[k] = nums[j]
                j += 1
            k += 1
        while i <= mid:
```

```
            temp[k] = nums[i]
            i += 1
            k += 1
        while j <= r:
            temp[k] = nums[j]
            j += 1
            k += 1
        nums[l:r+1] = temp[:r-l+1]
    mergeSort(0,len(nums)-1)
    return nums
```

动态规划



In [2]:

```python
## 0-1背包问题，理论篇
## 二维
W = 4
wt = [2,1,3]
val = [4,2,3]
def knapsack2(W,wt,val):
    N = len(wt)
    dp = [[0]*(W+1) for _ in range(N+1)]  # base case 已经初始化，填充多了一行
    # dp[i][j] 表示： 从0到i中选择，背包当前容量为j时的价值
    for i in range(1,N+1):
        for w in range(1,W+1):
            if w < wt[i-1]:  # wt和val要取i-1才能对应上
                dp[i][w] = dp[i-1][w]  # 如果到一维情况，那么表示不变dp[w]=dp[w]
            else:
                dp[i][w] = max(dp[i-1][w],dp[i-1][w-wt[i-1]]+val[i-1])
    return dp[N][W]

## 一维
## 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
## 而必须后序更新是避免重复运用

def knapsack1(W,wt,val):
    N = len(wt)
    dp = [0]*(W+1)  # base case 已经初始化，填充多了一行
    # dp[j] 表示： 背包当前容量为j时的价值
    for i in range(N): # 从0开始
        for w in range(W,wt[i]-1,-1):  # W-W-1-...-wt[i]
                dp[w] = max(dp[w],dp[w-wt[i]]+val[i]) # 覆盖
    return dp[-1]
knapsack2(W,wt,val)  # 取第一个和第二个
```


Out[2]:

6

In [ ]:

```python
### 416 等分割子集
def canPartition( nums):
    # 二维
    sumnums = sum(nums)
    if sumnums %2 != 0: return False  # 先判断再处理
    n = len(nums)
    sumnums = int(sumnums/2)
    dp = [[False]*(sumnums+1) for _ in range(n+1)]
    for i in range(n+1): dp[i][0] = True  # 背包装满了
    for i in range(1,n+1):
        for j in range(1,sumnums+1):
            if j < nums[i-1] :
                # 放不下
                dp[i][j] = dp[i-1][j]
            else:
                # 放得下，但是放还是不放
                dp[i][j] = dp[i-1][j] or dp[i-1][j-nums[i-1]]
    return dp[n][-1]


    # 一维
    sumnums = sum(nums)
    n = len(nums)
    w = sumnums//2
    if sumnums % 2 != 0:
        return False
    dp = [False] * (w+1)
    dp[0] = True
    for num in nums:
        for i in range(num,w+1)[::-1]:
            dp[i] = dp[i] or dp[i - num]   # 不断更新在不同w情况下的dp
    return dp[w]   # 定义为是否能放下该重量


    ## 套模板,更容易理解
    # 背包大小sum//2,物品的价值=物品的重量
    dp = [0]*(w+1)
    for i in range(n):
        for j in range(w,nums[i]-1,-1):
            # num[i]会大过背包重量,但是这是不会有任何操作
            dp[j] = max(dp[j],dp[j-nums[i]]+nums[i])
    return dp[w] == w

# 1049 最后一块石头
def lastStoneWeightII(stones):
    n = len(stones)
    target = sum(stones)//2
    dp = [0]*(target+1)
    for i in range(n):
        for j in range(target,stones[i]-1,-1):
            dp[j] = max(dp[j],dp[j-stones[i]]+stones[i])
    # (sum(stones)-dp[-1])-dp[-1]  （一堆<平均，一堆>平均，具体为多少未知）
    return sum(stpnes)-2*dp[-1]

# 494 目标和 改变数组元素的+-
def findTargetSumWays(nums, target):
    # 公式推导可得,不变的排前，变负的排后
```

```
    # left-right=target,left+right=sum,left=(target+sum)/2
    # 回溯,不去重的组合求和III，不去重相对于考虑了位置——超时
    # 和全排列不同——不要求全部加入(拿捏不准要多少个元素)。
    # 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。
    # 动态规划
    # 背包容量target,nums[i]即重量和价值，重量和<=target
    Sum = sum(nums)
    if target > Sum: return 0
    target,m = (target+Sum)//2,(target+Sum)%2
    if m != 0: return 0
    dp = [0]*(target+1)
    dp[0] = 1
    for i in range(len(nums)):
        for j in range(target,nums[i]-1,-1):
            dp[j] += dp[j-nums[i]]
    return dp[-1]

# 474 0和1，不可复用
def findMaxForm(strs, m, n):
    # 二维背包问题,n:1行m:0列
    if len(strs) == 0:
        return 0
    dp = [[0] * (m+1) for _ in range(n+1)]
    for str_ in strs:
        # 外物
        count_0 = str_.count('0')
        count_1 = str_.count('1')
        # for i in range(start,end,步长)
        # 0-1背包放反序 ， 内包
        for i in range(n,count_1-1,-1):
            for j in range(m,count_0-1,-1):
                dp[i][j] = max(dp[i][j],dp[i-count_1][j-count_0]+1)
    return dp[n][m]



# 完全背包
# 518 零钱兑换II
def change(amount, coins):
    # 完全背包
    dp = [0]*(amount+1)
    dp[0] = 1
    for i in range(len(coins)):
        for j in range(coins[i],amount+1):
            dp[j] += dp[j-coins[i]]
    return dp[-1]

# 377 组合总和IV
def combinationSum4(nums, target):
    dp = [0]*(target+1)
    dp[0]=1
    for i in range(1,target+1): # 背包
        for j in range(len(nums)):  # 物
            if i >= nums[j]:
                dp[i] += dp[i-nums[j]]
    return dp[-1]

# 70爬楼梯 排列数
def climbStairs(self, n: int) -> int:
    dp = [0]*(n + 1)
    dp[0] = 1
```

```
    m = 2
    for j in range(n + 1):
        # 遍历背包
        for step in range(1, m + 1):
            # 遍历物品
            if j >= step:
                dp[j] += dp[j - step]
    return dp[n]

# 322 零钱兑换I，完全背包，摆列组合都可以
def coinChange(coins, amount):
    dp = [amount+1]*(amount+1)
    dp[0] = 0
    # for i in range(1,amount+1):
    #      for coin in coins:
    #         if i < coin: continue
    #         dp[i] = min(dp[i],dp[i-coin]+1)   # 记得加一
    # return dp[-1] if dp[-1] <= amount else -1
    for coin in coins:
        for i in range(coin,amount+1):
            # 少了一行判断
            dp[i] = min(dp[i],dp[i-coin]+1)   # 记得加一
    return dp[-1] if dp[-1] <= amount else -1

# 279 完全平方数
def numSquares(n):
    # 完全背包，正序
    # 最少个数，组合排列都可以
    nums = [i**2 for i in range(1, n + 1) if i**2 <= n]  # 构造物
    dp = [10**4]*(n+1)
    dp[0] = 0
    for i in range(len(nums)):
        for j in range(nums[i],n+1):
            dp[j] = min(dp[j],dp[j-nums[i]]+1)
    return dp[-1]

# 单词拆分——
def wordBreak(s, wordDict):
    n = len(s)
    dp = [False]*(n+1)
    dp[0] = True
    for i in range(1,n+1):
        for word in wordDict:
            lenw = len(word)
            if lenw <= i and word == s[i-lenw:i]:
                dp[i] = dp[i] or dp[i-lenw]
    return dp[n]


# 198 打家劫舍I
def rob(self, nums: List[int]) -> int:
    if nums == []:
        return 0
    n = len(nums)
    dp = [0]*(n+2)
    for i in range(2,n+2):
        dp[i] = max(dp[i-1],dp[i-2] + nums[i-2])
    return dp[-1]

# 213 打家劫舍II
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]:

```python
### 多重背包理论篇-->转成0-1背包问题
def multi_pack1(weight,values,nums,bag_weight):
    # 将物品展开,将nums消除掉
    for i in range(len(nums)):
        while nums[i]:
            weight.append(weight[i])
            value.append(value[i])
            nums[i] -= 1
    dp = [0]*(bag_weight+1)
    for i in range(len(weight)):
        for j in range(bag_weight,weight[i]-1,-1):
            dp[j] = max(dp[j],dp[j-weight[i]]+value[i])
    print(" ".join(map(str, dp)))

weight = [1, 3, 4]
values = [15, 20, 30]
nums = [2, 3, 2]
bag_weight = 10
multi_pack1(weight,values,nums,bag_weight)
```

```
0 15 30 45 45 50 65 75 75 85 95
```

Stock

In [ ]:

```python
# dp[i][j] 表示第i天利润
#### 121 一次买入，一次卖出，同一天买卖没有收益，k=1
 def maxProfit(self, prices: List[int]) -> int:

    # dp[i][j]  i 表示 天数, j 表示 1持有 or 0出售
    n = len(prices)
    dp = [[0]*2 for i in range(n)]
    dp[0][0] = 0
    dp[0][1] = -prices[0]  # 现金流
    for i in range(1,n):
        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
        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
    return dp[-1][0]

#### 122 多次买卖 k = +inf
def maxProfit(self, prices: List[int]) -> int:
    # 捕获上坡那一段即可，贪心
    res = 0
    for i in range(1,len(prices)):
        res += max(prices[i]-prices[i-1],0)
    return res

    # 动态规划, 0表示不持有，1表示持有
    n = len(prices)
    dp = [[0]*2 for _ in range(n)]
#     dp[0][0] = 0
    dp[0][1] = -prices[0]
    for i in range(1,n):
        dp[i][0] = max(dp[i-1][0],dp[i-1][1]+prices[i])  # 不持有(以前没有or刚卖)
        dp[i][1] = max(dp[i-1][1],dp[i-1][0]-prices[i])  # 持有(以前有or刚买)
    return dp[-1][0]

    # dp 加滚动数组
    n = len(prices)
    dp = [[0]*2 for _ in range(2)] # 空间压缩
#     dp[0][0] = 0
    dp[0][1] = -prices[0]
    for i in range(1,n):
        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)
        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)
    return dp[(n-1)%2][0]

#### 123 两次买卖 k=2
def maxProfit(self, prices: List[int]) -> int:
    # 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
    n = len(prices)
    if n == 0: return 0
    dp = [[0]*5 for i in range(n)]
    dp[0][1] = -prices[0]
    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)
    for i in range(1,n):
        # dp[i][0] = dp[i-1][0]  # 没操作
        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
        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
        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
        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
    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 [ ]:

```python
#### 300 最长递增子序列
def lengthOfLIS(self, nums: List[int]) -> int:
    n = len(nums)
    dp = [1]*n #dp[i] 表示以i结尾的最长子序列长度
    for i in range(1,n):
        for j in range(i):
            if nums[i] > nums[j]: # 符合递增，找前面比他小的数的dp+1
                dp[i] = max(dp[i],dp[j]+1) # 一直找一直找
    return max(dp)

#### 674 最长递增子串
def findLengthOfLCIS(self, nums: List[int]) -> int:
    n = len(nums)
    dp = [1]*n  #dp[i] 表示以i结尾的最长子串长度
    for i in range(n-1):
        if nums[i+1] > nums[i]:
            dp[i+1] = dp[i]+1
    return max(dp)

#### 718 最长重复子串
def findLength(self, nums1: List[int], nums2: List[int]) -> int:
    # 子串
    n = len(nums1)
    m = len(nums2)
    res = 0
    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
    for i in range(1,n+1):
        for j in range(1,m+1):
            if nums1[i-1] == nums2[j-1]: # 错位对齐
                dp[i][j] = dp[i-1][j-1]+1  # 不相等即为0
            res = max(res,dp[i][j])
    return res

#### 1143 最长公共子序列
def longestCommonSubsequence(self, text1: str, text2: str) -> int:
    m = len(text1)
    n = len(text2)
    if n == 0 or m == 0:
        return 0
    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
    for i in range(1,m+1):  # i,j对应谁要对应上
        for j in range(1,n+1):
            if text1[i-1] == text2[j-1]:
                dp[i][j] = dp[i-1][j-1]+1
            else:
                dp[i][j] = max(dp[i-1][j],dp[i][j-1])
    return dp[-1][-1]


### 53 最大子序号，贪心的另一种解法
def maxSubArray(self, nums: List[int]) -> int:
    n = len(nums)
    dp = [0]*n
    dp[0] = nums[0]
    for i in range(1,n):
        dp[i] = max(dp[i-1]+nums[i],nums[i])
    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]
```

```python

#### 72 编辑距离
def minDistance(self, word1: str, word2: str) -> int:
    n = len(word1)
    m = len(word2)
    dp = [[0]*(m+1) for _ in range(n+1)]
    for i in range(n+1):
        dp[i][0] = i
    for j in range(m+1):
        dp[0][j] = j
    for i in range(1,n+1):
        for j in range(1,m+1):
            if word1[i-1] == word2[j-1]:  # word和dp错位一格
                dp[i][j] = dp[i-1][j-1]
            else:
                dp[i][j] = min(dp[i-1][j]+1,dp[i][j-1]+1,dp[i-1][j-1]+1)
    return dp[-1][-1]

#### 647 回文子串
def countSubstrings(self, s: str) -> int:
    result = 0
    n = len(s)
    dp = [[False]*n for _ in range(n)]
    for i in range(n-1,-1,-1):
        for j in range(i,n):
            if s[i] == s[j]:
                if j-i <= 1:
                    dp[i][j] = True
                    result += 1
                elif dp[i+1][j-1]:
                    dp[i][j] = True
                    result += 1
    return result

#### 516 最长回文子串
def longestPalindromeSubseq(self, s: str) -> int:
    n = len(s)
    dp = [[0]*n for _ in range(n)]
    for i in range(n):
        dp[i][i] = 1
    for i in range(n-1,-1,-1):
        for j in range(i+1,n):
            if s[i] == s[j]:
                dp[i][j] = dp[i+1][j-1]+2
            else:
                dp[i][j] = max(dp[i+1][j],dp[i][j-1])
    return dp[0][-1]


#### 高楼扔鸡蛋
def superEggDrop(self, k: int, n: int) -> int:
    # 单纯备忘录超出时间限制
    memo = {}
    def dp(k,n):
        if k == 1: return n
        if n == 0: return 0
        if (k,n) in memo.keys():
            return memo[(k,n)]
        res = float('inf')
        # for i in range(1,n+1):
        #     # 这种属于线性搜索，让i 从0 开始到n
```

```
        #     res = min(res,max(dp(k,n-i),dp(k-1,i-1))+1)  # res = min(res,max(碎, 没碎)+1)
        # 替换为二分搜索,就是将i->(0,n) 变成->(mid)
        lo = 1; hi = n
        while lo <= hi:
            # 通过循环不断的改变mid
            mid = lo+(hi-lo)//2
            broken = dp(k-1,mid-1)
            not_broken = dp(k,n-mid)
            if broken > not_broken:
                hi = mid -1  # mid相当于i
                res = min(res,broken+1)
            else:
                lo = mid + 1  # mid相当于i
                res = min(res,not_broken+1)
        memo[(k,n)]=res
        return res
    return dp(k,n)

    # 换定义
    dp = [[0]*(n+1) for _ in range(k+1)]
    m = 0
    while dp[k][m] < n:
        m += 1
        for i in range(1,k+1):
            dp[i][m] = dp[i][m - 1] + dp[i - 1][m - 1] + 1
    return m
```