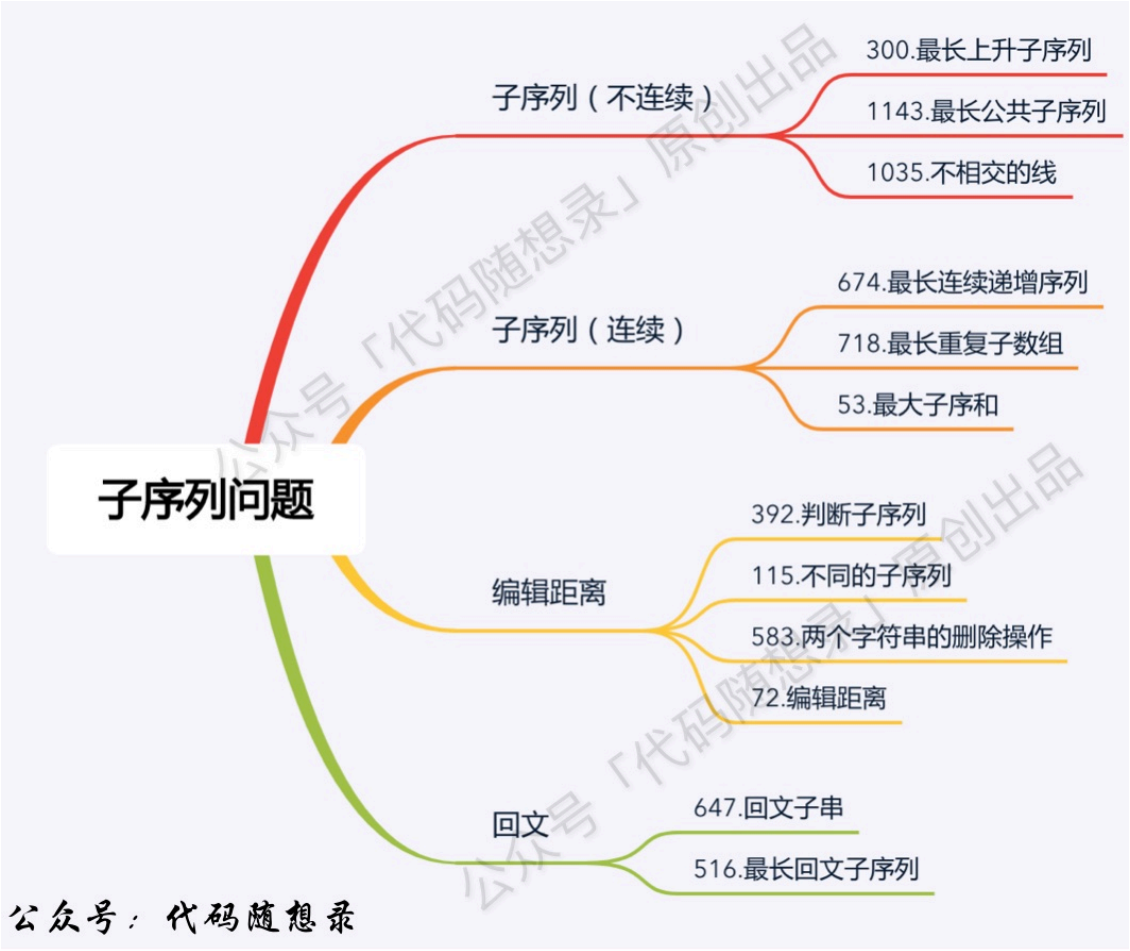
DP

子序列问题(substrings)



子序列（不连续）

300.Longest Increasing Subsequence(非常经典)

Q:Given an integer array `nums`, return the length of the longest strictly increasing subsequence.

A: dp[i]表示i位置最长的递增序列。位置i的最长升序子序列等于j从0到i-1各个位置的最长升序子序列 + 1 的最大值。所以：**if (nums[i] > nums[j]) dp[i] = max(dp[i], dp[j] + 1)**;

```python
class Solution:
    def lengthOfLIS(self, nums: List[int]) -> int:
        if len(nums) <= 1:
            return len(nums)
        dp = [1] * len(nums)
        result = 0
        for i in range(1, len(nums)):
            for j in range(0, i):
                if nums[i] > nums[j]:
                    dp[i] = max(dp[i], dp[j] + 1)
            result = max(result, dp[i]) #取长的子序列
        return result
```

354.Russian Doll Envelopes

本质：LIS的2d情况

方法：`envelopes.sort(key= lambda x: (x[0],-x[1]))`

难点：在二维数组中，对第一个元素升序排序，第二个元素在第一个元素相同的情况下降序。

https://docs.python.org/3/howto/sorting.html

sort 与 sorted 区别：

sort 是应用在 list 上的方法，sorted 可以对所有可迭代的对象进行排序操作。

list 的 sort 方法返回的是对已经存在的列表进行操作，无返回值，而内建函数 sorted 方法返回的是一个新的 list，而不是在原来的基础上进行的操作。

```python
# 复杂度较高 有些测试用例过不了
class Solution:
    def maxEnvelopes(self, envelopes: List[List[int]]) -> int:
        if not envelopes:
            return 0

        envelopes.sort(key= lambda x: (x[0],-x[1])) #一个升序 一个降序

        n = len(envelopes)
        f = [1] * n
        for i in range(n):
            for j in range(i):
                if envelopes[j][1] < envelopes[i][1]:
                    f[i] = max(f[i], f[j] + 1)

        return max(f)
```

1143. 最长公共子序列（母题，很多题目出自这题,非常非常重要！！！！！）

Q：Given two strings `text1` and `text2`, return *the length of their longest* ***common subsequence***. If there is no **common subsequence**, return `0`.

A: i 指向text1, j指向text2，dp[i][j]指当前位置最长公共子序列数量。很多子串类题目在写的时候都会初始化列表的长度+1，是因为腾出空间给字符串不存在的时候初始化为0. 当这样写的时候，比较两个字符串中的字母是否相等，就要写成 `text1[i-1] == text2[j-1]`. 而最后return的是dp[m][n].

递推式易得：

```
class Solution:
    def longestCommonSubsequence(self, text1: str, text2: str) -> int:
        m = len(text1)
        n = len(text2)
        dp = [[0 for _ in range(n+1)]for _ in range(m+1)]
        for i in range(1,m+1):
            for j in range(1,n+1):
                if(text1[i-1]!=text2[j-1]):
                    dp[i][j] = max(dp[i-1][j],dp[i][j-1])
                else:
                    dp[i][j] = dp[i-1][j-1]+1
        return dp[m][n]
```

1035.Uncrossed Lines

A: 可以转换为1143.代码完全一样。

91. Decode Ways

类似题目有：70. Climbing Stairs

分析：

- 本题利用动态规划比较容易解决，但需要注意分情况讨论
  - dp[i] 为 str[0..i] 的译码方法总数
  - 分情况讨论：（建立最优子结构）
    - **若 s[i] = '0'**，那么若 s[i-1] = '1' or '2'，则 dp[i] = dp[i-2]；否则 return 0
      - 解释：s[i-1]+s[i] 唯一被译码，不增加情况
    - **若 s[i-1] = '1'**, 则 dp[i] = dp[i-1] + dp[i-2]
      - 解释：s[i-1]与s[i]分开译码，为dp[i-1]；合并译码，为dp[i-2]
    - **若s[i-1]='2'** and '1'<=s[i]<='6'，则 dp[i] = dp[i-1] + dp[i-2]
      - 解释：同上
  - 由分析可知，dp[i] 仅可能与前两项有关，故可以用单变量代替 dp[] 数组，将空间复杂度从O（n）降到 O（1）

```python
class Solution:
    def numDecodings(self, s: str) -> int:
        if len(s)<1:
            return 0
        dp = [0 for _ in range(len(s)+1)]
        dp[0] = 1
        dp[1] = 1 if s[0] != '0' else 0

        for i in range(2,len(s)+1):
            if s[i-1] != '0':
                dp[i] += dp[i-1]
            if s[i - 2] != '0' and s[i-2:i] < '27': #千万不能用elif
                dp[i] += dp[i-2]
        return dp[len(s)]
```

子序列（连续）

674.Longest Continuous Increasing Subsequence

A：很简单。初始化一个每个元素为0的dp数组。如果该元素大于前一个元素（因为是连续的，跟前一个元素比较就可以，如果是不连续的，就是300题，要跟前面所有元素比较一次），就+1，否则等于1. 注意要初始化dp的第一个元素为1，然后从i=1开始遍历.

```python
class Solution:
    def findLengthOfLCIS(self, nums: List[int]) -> int:
        dp = [0 for _ in range(len(nums))]
        dp[0]=1
        for i in range(1,len(nums)):
            if(nums[i]>nums[i-1]):
                dp[i] = dp[i-1]+1
            else:
                dp[i] = 1
        return max(dp)
```

718. Maximum Length of Repeated Subarray

Q:Given two integer arrays `nums1` and `nums2`, return *the maximum length of a subarray that appears in* ***both*** *arrays*.

A: 和上一题几乎没有区别。只是上一题是一个数组，本题是两个数组。只需要用i，j分别遍历就好。

```python
class Solution:
    def findLength(self, nums1: List[int], nums2: List[int]) -> int:
        dp = [[0] * (len(nums2)+1) for _ in range(len(nums1)+1)]
        result = 0
        for i in range(1, len(nums1)+1):
            for j in range(1, len(nums2)+1):
                if (nums1[i-1] == nums2[j-1]):
                    dp[i][j] = dp[i-1][j-1] + 1
                result = max(result, dp[i][j])
        return result
```

53.Maximum Subarray

Q：Given an integer array `nums`, find the contiguous subarray (containing at least one number) which has the largest sum and return *its sum*.（找到拥有最大和的连续子序列）

A：**dp[i]：包括下标i之前的最大连续子序列和为dp[i]。**

- dp[i - 1] + nums[i]，即：nums[i]加入当前连续子序列和
- nums[i]，即：从头开始计算当前连续子序列和

  上面两个取最大值，就是dp[i]能去到的最大，然后再跟全局变量比较，如果大于全局变量则取代它作为全局最大值。

```python
class Solution:
    def maxSubArray(self, nums: List[int]) -> int:
        n = len(nums)
        dp = [0 for _ in range(n)]
        dp[0] = nums[0]
        global_max = dp[0]
        for i in range(1,n):
            dp[i] = max(dp[i-1]+nums[i],nums[i])
            global_max = max(dp[i],global_max)
        return global_max
```

152.Maximum Product Subarray

是上一道题目的乘法版本，但是差异挺大。本题需要用2个变量来记录类似dp[i-1]的东西，因为一旦当前变量是负数，之前最小的就会变成最大，而之前最大的就会变成最小，所以我们要准备两个变量记录两种结果。（有点抽象）

```python
class Solution:
    def maxProduct(self, nums: List[int]) -> int:
        res = -inf
        imax = 1
        imin = 1
        for i in range(len(nums)):
```

```
            #出现负数 最大的变最小 最小的变最大
            if nums[i]<0:
                tmp = imax
                imax = imin
                imin = tmp
            imax = max(imax*nums[i],nums[i])
            imin = min(imin*nums[i],nums[i])
            res = max(res,imax)
        return res
```

编辑距离系列(edit distance)

392. **Is Subsequence**

Q:Given two strings `s` and `t`, return `true` *if* `s` *is a* ***subsequence*** *of* `t`*, or* `false` *otherwise.*

A: 转化成1143（最长公共子序列）. 如果s和t的最长公共子序列长度等于s（短的那条），就return True.

```
class Solution:
    def isSubsequence(self, s: str, t: str) -> bool:
        m = len(s)
        n = len(t)
        dp = [[0] * (n + 1) for _ in range(m + 1)]
        for i in range(1,m+1):
            for j in range(1,n+1):
                if(s[i-1]==t[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] == m
```

115.Distinct Subsequences(经典题目)

Q:Given two strings `s` and `t`, return *the number of distinct subsequences of* `s` *which equals* `t`.

Ex：

```
Input: s = "rabbbit", t = "rabbit"
Output: 3
Explanation:
As shown below, there are 3 ways you can generate "rabbit" from S.
rabbbit
rabbbit
rabbbit
```

A：遇到相同的字母有两种选择：匹配或者跳过。这两种可能性的和就是所有可能得个数。当遇到不同字母时，就只能跳过。还有一个要注意的是初始化问题。分析如下：

从递推公式dp[i][j] = dp[i - 1][j - 1] + dp[i - 1][j]; 和 dp[i][j] = dp[i - 1][j]; 中可以看出dp[i][0] 和dp[0][j]是一定要初始化的。

每次当初始化的时候，都要回顾一下dp[i][j]的定义，不要凭感觉初始化。i->s, j->t,问题是从s中找等于t的子串个数。

dp[i][0]表示什么呢?

dp[i][0] 表示：以i-1为结尾的s可以随便删除元素，出现空字符串的个数。

那么dp[i][0]一定都是1，因为也就是把以i-1为结尾的s，删除所有元素，出现空字符串的个数就是1。

再来看dp[0][j]，dp[0][j]：空字符串s可以随便删除元素，出现以j-1为结尾的字符串t的个数。

那么dp[0][j]一定都是0，s如论如何也变成不了t。

最后就要看一个特殊位置了，即：dp[0][0] 应该是多少。

dp[0][0]应该是1，空字符串s，可以删除0个元素，变成空字符串t。

```python
class Solution:
    def numDistinct(self, s: str, t: str) -> int:
        m = len(s)
        n = len(t)
        dp = [[0 for _ in range(n+1)]for _ in range(m+1)]
        for i in range(m+1):
            dp[i][0] = 1
        '''
        for j in range(n+1):
            dp[0][j] = 0
        '''
        for i in range(1,m+1):
            for j in range(1,n+1):
```

```
                if(s[i-1]!=t[j-1]):
                    dp[i][j] = dp[i-1][j] #不匹配 j往后走一个
                else:
                    dp[i][j] = dp[i-1][j-1] + dp[i-1][j] #匹配 + 不匹配
        return dp[m][n]
```

583. **Delete Operation for Two Strings**

Q：Given two strings `word1` and `word2`, return *the minimum number of **steps** required to make* `word1` *and* `word2` *the same*.

A:转化为1143(最长公共子序列lcs).要删除的个数就是word1-lcs + word2-lcs.

```
class Solution:
    def minDistance(self, word1: str, word2: str) -> int:
        m, n = len(word1), len(word2)
        dp = [[0] * (n + 1) for _ in range(m + 1)]
        for i in range(1, m + 1):
            for j in range(1, n + 1):
                if word1[i - 1] == word2[j - 1]:
                    dp[i][j] = dp[i - 1][j - 1] + 1
                else:
                    dp[i][j] = max(dp[i - 1][j], dp[i][j - 1])

        lcs = dp[m][n]
        return m - lcs + n - lcs
```

72. **Edit Distance**

Q：Given two strings `word1` and `word2`, return *the minimum number of operations required to convert* `word1` *to* `word2`.

You have the following three operations permitted on a word:

- Insert a character（增）
- Delete a character（删）
- Replace a character（改）

A： 这题和上一题的区别是本题只能操作word1，这就比较好弄了。

一样：
dp[i][j] = dp[i-1][j-1]

不一样：
replace： dp[i][j] = dp[i-1][j-1]+1
delete: dp[i][j] = dp[i-1][j]
insert: word1增加一个变成word2其实就是等于word2删除一个变成word1: dp[i][j] = dp[i][j-1]
这三个里面取最小就行

还要注意初始化问题：
虽然都是从word1->word2，但本题初始化不同于115.Distinct Subsequences。 如果word2为空串，则dp[i]的值应该为当前word1的长度，如果word1为空串，则dp[j]的值应该为当前word2的长度

```python
class Solution:
    def minDistance(self, word1: str, word2: str) -> int:
        m = len(word1)
        n = len(word2)
        dp = [[0 for _ in range(n+1)] for _ in range(m+1)]
        #初始化 如果都为，则要操作的数量的数据的长度
        for i in range(m+1):
            dp[i][0] = i

        for j in range(n+1):
            dp[0][j] = j

        for i in range(1,m+1):
            for j in range(1,n+1):
                if(word1[i-1]==word2[j-1]):
                    dp[i][j] = dp[i-1][j-1]
                else:
                    # insert: dp[i-1][j]+1
                    # delete: dp[i][j-1]+1 #word1删除一个元素等于word2插入一个元素
                    # replace : dp[i-1][j-1]+1
                    dp[i][j] = min(dp[i-1][j],dp[i][j-1],dp[i-1][j-1])+1
        return dp[-1][-1]
```

回文（Palindromic）

647. **Palindromic Substrings(经典题目)**

Q：Given a string `s`, return *the number of **palindromic substrings** in it*.A string is a **palindrome** when it reads the same backward as forward.

A：回文的题目，肯定都是一个指针在左边一个在右边或者是从中间扩散到两边。

当只有一个字符时，比如 a 自然是一个回文串。
当有两个字符时，如果是相等的，比如 aa，也是一个回文串。
当有三个及以上字符时，比如 ababa 这个字符记作串 1，把两边的 a 去掉，也就是 bab 记作串 2， 可以看出只要串2是一个回文串，那么左右各多了一个 a 的串 1 必定也是回文串。所以当 s[i]==s[j] 时，自然要看 dp[i+1][j-1] 是不是一个回文串。

```
class Solution:
    def countSubstrings(self, s: str) -> int:
        dp = [[False] * len(s) for _ in range(len(s))]
        result = 0
        for i in range(len(s)-1, -1, -1): #注意遍历顺序
            for j in range(i, len(s)):
                if s[i] == s[j]:
                    if j - i <= 1:
                        result += 1
                        dp[i][j] = True
                    elif dp[i+1][j-1]==True:
                        result += 1
                        dp[i][j] = True
        return result
```

516. Longest Palindromic Subsequence

Q：上一题是回文子串的个数，这一题是最长回文子串。

A：遍历的方式完全一样，因为要求最长，递归式就是之前求最长的模版稍微修改一下(1143).值得注意的是，当两个元素相等的时候，回文子串的长度是+2. 初始化的时候，对角线先初始化为1，因为是相同的

还有一点，返回时，应该返回右上角的元素而不是右下角。为什么？递归方程如下：

```
dp[i][j] = dp[i + 1][j - 1] + 2 #s[i+1]==s[j-1]
dp[i][j] = max(dp[i + 1][j], dp[i][j - 1])  #otherwise
```

dp[i][j] 由dp[i+1][j-1]，dp[i + 1][j],dp[i][j - 1]推出，i+1指下一行元素，本行元素由下一行元素推出，所以我们必须先更新下一行，所以对i是从下到上遍历，最终返回的元素是右上角。

```
class Solution:
    def longestPalindromeSubseq(self, s: str) -> int:

        n = len(s)
        dp = [[0 for _ in range(n)]for _ in range(n)] #并没有n+1

        #元素相等
        for i in range(n):
            dp[i][i] = 1
```

```
        #i right->left
        #j left->right
        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] #并不是返回右下角，而是返回右上角
```

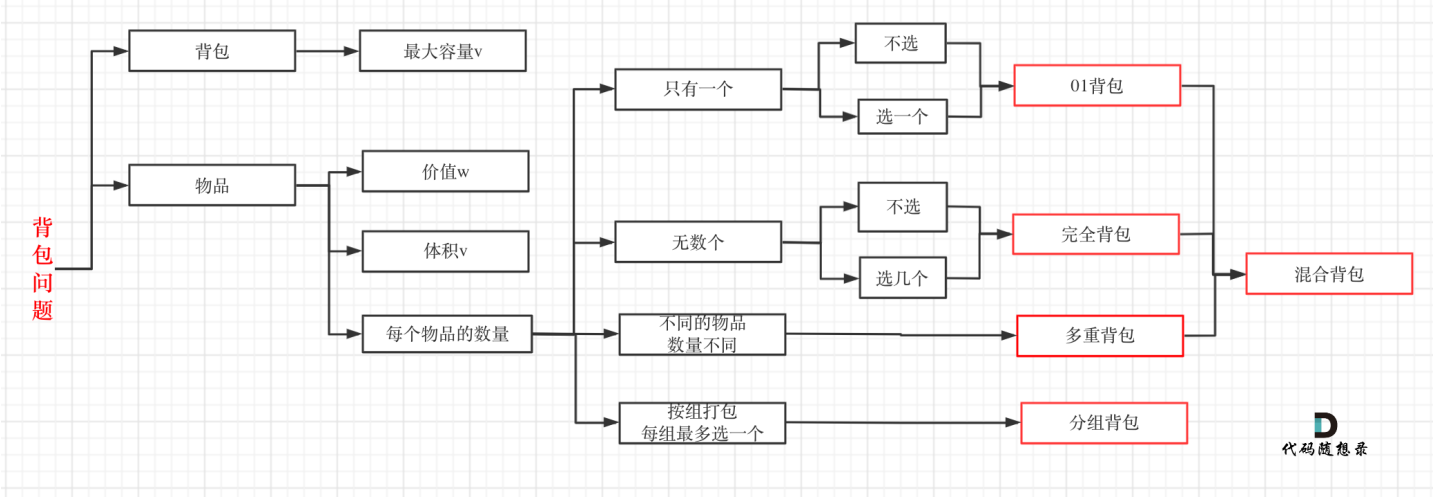
517. Longest Palindromic Substring

```
class Solution:
    def longestPalindrome(self, s: str) -> str:
        #检查是否回文
        #记录长度
        #循环不断更新长度
        length = 0
        left = 0
        right = 0
        dp = [[False]* len(s) for _ in range(len(s))]
        for i in range(len(s),-1,-1):
            for j in range(i,len(s)):
                if s[i] == s[j]:
                    if j-i<=1 or dp[i+1][j-1]:
                        dp[i][j] = True
                    if dp[i][j]==True and j-i+1 > length:
                        length = j-i+1
                        left = i
                        right = j
        return s[left:right+1]
```

背包问题

背包问题模版性很强，只要判断出来是背包问题和其类型，基本可以套模版解决。

分类及解法：



背包问题大体的解题模板是两层循环，分别遍历物品nums和背包容量target，然后写转移方程，
根据背包的分类我们确定物品和容量遍历的先后顺序，根据问题的分类我们确定状态转移方程的写法

首先是背包分类的模板：
1、0/1背包（物品只能挑1次或0次）：外循环nums（物品）,内循环target（重量）,target倒序且target>=nums[i];
2、完全背包（物品可以重复挑选）：外循环nums,内循环target,target正序且target>=nums[i];
3、组合背包：外循环target,内循环nums,target正序且target>=nums[i];

- **如果求组合数就是外层for循环遍历物品，内层for遍历背包。**
- **如果求排列数就是外层for遍历背包，内层for循环遍历物品。**

然后是问题分类的模板：
1、最值问题: dp[j] = max/min(dp[j], dp[j-nums[i]]+1)或dp[j] = max/min(dp[j], dp[j-num[i]]+nums[i]);
2、存在问题(bool)：dp[j]=dp[j]||dp[j-num[i]];
3、不考虑顺序组合问题（有几种方法）：dp[j]+=dp[j-num[i]];

Note：

1. 注意target的取值，很多题目target的隐形取值为target/2

2. **背包的初始化一般都是target+1,如果求最大值，则初始化为0；如果求最小值，则初始成题目给定的最大范围的值。**

Q：有n件物品和一个最多能背重量为w 的背包。第i件物品的重量是weight[i]，得到的价值是value[i] 。**每件物品只能用一次**，求解将哪些物品装入背包里物品价值总和最大。

A：

二维数组解法

dp的定义：dp[i][j] 表示从下标为[0-i]的物品里任意取，放进容量为j的背包，价值总和最大是多少

递推公式：`dp[i][j] = max(dp[i - 1][j], dp[i - 1][j - weight[i]] + value[i])`

打表：具体做法是：画一个 len 行，target + 1 列的表格。这里 len 是物品的个数，target 是背包的容量。len 行表示一个一个物品考虑，target + 1多出来的那 1 列，表示背包容量从 0 开始考虑。很多时候，我们需要考虑这个容量为 0 的数值。

初始化：

- dp[0][j]，即：i为0，存放编号0的物品的时候，各个容量的背包所能存放的最大价值。
- dp[i][0],即背包重量为0，物品价值当然初始化为0.

```
rows, cols = len(weight), bag_size + 1
dp = [[0 for _ in range(cols)] for _ in range(rows)]

# 初始化dp数组.
for i in range(rows):
  dp[i][0] = 0
first_item_weight, first_item_value = weight[0], value[0]
for j in range(1, cols):
  if first_item_weight <= j:
    dp[0][j] = first_item_value
```

遍历顺序：先遍历物品还是重量?

主要代码：

```
# 更新dp数组: 先遍历物品, 再遍历背包.
for i in range(1, len(weight)):
  cur_weight, cur_val = weight[i], value[i]
  for j in range(1, cols):
    if cur_weight > j: # 说明背包装不下当前物品.
      dp[i][j] = dp[i - 1][j] # 所以不装当前物品.
    else:
    # 定义dp数组: dp[i][j] 前i个物品里，放进容量为j的背包，价值总和最大是多少。
      dp[i][j] = max(dp[i - 1][j], dp[i - 1][j - cur_weight]+ cur_val)
```

一维数组解法（滚动数组）

条件：上一层可以重复利用，直接拷贝到当前层。但是还是要用i和j遍历。

dp[j]表示：容量为j的背包，所背的物品价值可以最大为dp[j]。

递推公式：`dp[j] = max(dp[j], dp[j - weight[i]] + value[i])`

初始化：全为0即可,bag_weight是背包的容量 `dp = [0] * (bag_weight + 1)`

遍历顺序！：二维dp遍历的时候，背包容量是从小到大，而一维dp遍历的时候，背包是从大到小。倒序遍历的目的是为了只让物品i被放入1次。倒序遍历的原因是，本质上还是一个对二维数组的遍历，并且右下角的值依赖上一层左上角的值，因此需要保证左边的值仍然是上一层的，从右向左覆盖。**(看例子，已经画图给出)**

主要代码:

```
# 先遍历物品, 再遍历背包容量
 for i in range(len(weight)):
     for j in range(bag_weight, weight[i] - 1, -1):
        #bag_weight >= weight[i] -> bag_weight > weight[i]-1
        # 递归公式
         dp[j] = max(dp[j], dp[j - weight[i]] + value[i])
```

0-1 背包

416. **Partition Equal Subset Sum**

Q:Given a **non-empty** array `nums` containing **only positive integers**, find if the array can be partitioned into two subsets such that the sum of elements in both subsets is equal.

A：问题转化成能不能捡一堆硬币使得总和为列表总元素之和的一半。如果不能直接返回 False，可以的话直接调用0-1背包模版即可。

```python
class Solution:
    def canPartition(self, nums: List[int]) -> bool:
        sum_nums = sum(nums)
        if (sum_nums%2==1): return False
        target = sum(nums)//2 # 背包体积
        dp = [0]* (target+1) #有答案是10001，其实只要target+1即可
        for i in range(len(nums)):
            for j in range(target,nums[i]-1,-1):
                dp[j] = max(dp[j], dp[j-nums[i]]+nums[i])
        return target == dp[target]
```

049. **Last Stone Weight II**

Q：You are given an array of integers `stones` where `stones[i]` is the weight of the `ith` stone.

We are playing a game with the stones. On each turn, we choose any two stones and smash them together. Suppose the stones have weights `x` and `y` with `x <= y`. The result of this smash is:

- If `x == y`, both stones are destroyed, and
- If `x != y`, the stone of weight `x` is destroyed, and the stone of weight `y` has new weight `y - x`.

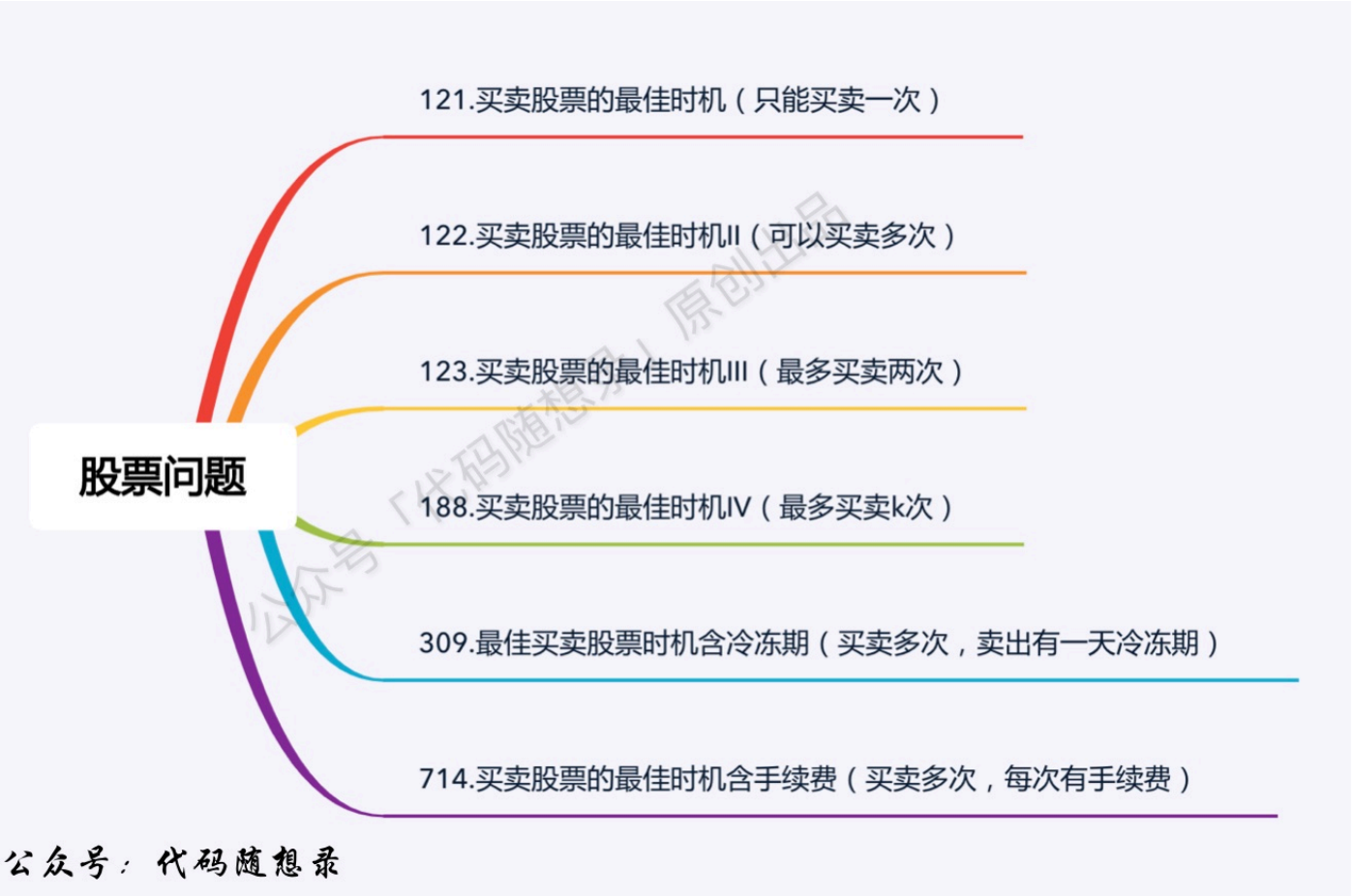
At the end of the game, there is **at most one** stone left.

Return *the smallest possible weight of the left stone*. If there are no stones left, return `0`.

A：问题转换为把石头分成尽量平衡的两组->target = target/2,后面就是模版。注意最后返回的是sum_weight - 2*dp[target].

```python
class Solution:
    def lastStoneWeightII(self, stones: List[int]) -> int:
        sum_weight = sum(stones)
        target = sum_weight//2
        dp = [0]*(target+1)
        for i in range(len(stones)):
            for j in range(target,stones[i]-1,-1):
                dp[j] = max(dp[j],dp[j-stones[i]]+stones[i])
        return sum_weight - 2*dp[target]
```

股票问题



此类问题只有一个套路：状态机！

dp[m][n]: 其中m代表第几天，n代表状态。

121. 只能买卖一次

#dp[m][0]: 第m天处于卖出状态: 有可能是前一天已经卖出，有可能是今天刚卖出
#dp[m][1]: 第m天处于买入状态：有可能是前一天已经买入，有可能刚买入

```
#dp[m][0]: 第m天处于卖出状态
#dp[m][1]: 第m天处于卖入状态

class Solution:
    def maxProfit(self, prices: List[int]) -> int:
        dp = [[0]*2 for _ in range(len(prices))]

        #初始化
        dp[0][0] = -prices[0]
        dp[0][1] = 0
```

```
        #转移方程
        for i in range(1,len(prices)):
            #dp[i][0] 第i天的状态是持有的
            #dp[i][1] 第i天的状态是未持有的
            dp[i][0] = max(dp[i-1][0],-prices[i]) #前一天持有/今天刚持有.. 的收益
            #为什么是+?  因为第二个状态是前一天持有，今天卖。其中前一天持有是个负数。
            dp[i][1] = max(dp[i-1][1],dp[i-1][0]+prices[i])

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

122. 可以买卖多次：同121

#dp[i-1][1]-prices[i] 指今天买入的又卖出去

```
class Solution:
    def maxProfit(self, prices: List[int]) -> int:
        # 可以多次买卖
        #索引编号分析

        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][1]-prices[i] 今天买入的又卖出去
            dp[i][1] = max(dp[i-1][1],dp[i-1][0]+prices[i])
        return dp[len(prices)-1][1]
```

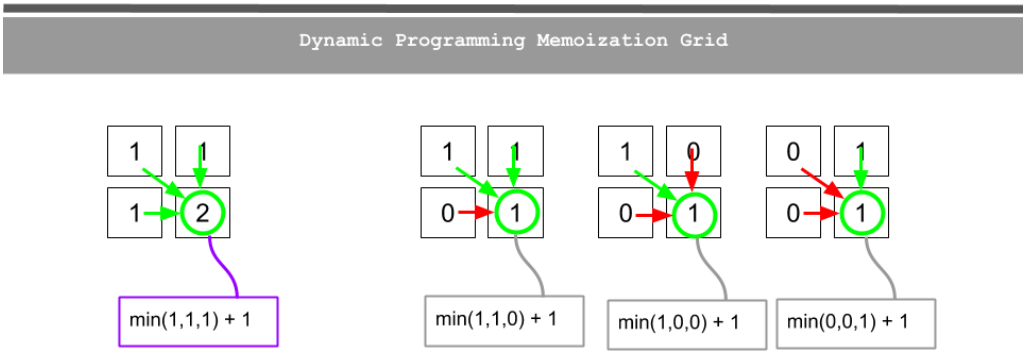
其他

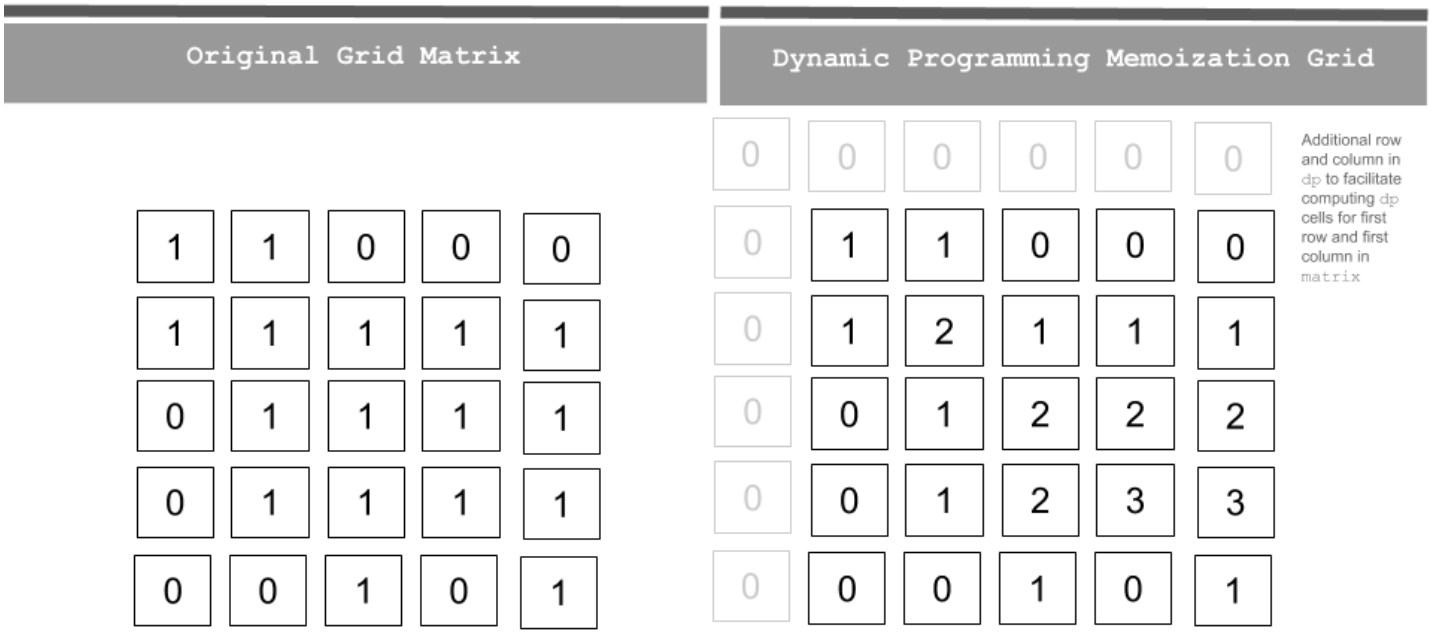
221. Maximal Square

dp[i][j]:以i，j为右下角的最大正方形边长

什么时候需要初始化dp数组需要多一行和一列？？？ 因为在状态转移方程中需要用到上一行和左边一列的数据。对于第一行和第一列的数据来说他们没有。如果我们没有初始化多一行的话，他们需要自行判断。但如果初始多一样，他们能直接在循环中判断，不需要再另行判断。

- We are going to use dynamic programming to memoize the combo elements.
- Whenever we see '1' in `matrix`, are going to look the surrounding elements.
  - We are going to get the minimum value of all surrounding elements in `dp` grid.Then, we are going to add 1. WHY?
    - If this 1 is a part of a combo chain, this will increase the square size.
    - If this 1 is NOT a part of a combo chain, at least it is a cell of size 1x1 on its own.





- Have a `max_side` variable updated as you process each cell
- Return the value of `max_side * max_side` when the loops are done

```python
#代码
#继承了之前的风格 如1143,675,392等
class Solution:
    def maximalSquare(self, matrix: List[List[str]]) -> int:
        if matrix is None or len(matrix) < 1:
            return 0

        rows = len(matrix)
        cols = len(matrix[0])
```

```
        dp = [[0]*(cols+1) for _ in range(rows+1)]
        max_side = 0

        for r in range(1,rows+1):
            for c in range(1,cols+1):
                if matrix[r-1][c-1] == '1':
                    dp[r][c] = min(dp[r-1][c-1], dp[r-1][c], dp[r][c-1]) + 1
                    max_side = max(max_side, dp[r][c])

        return max_side * max_side
```

329.Longest Increasing Path in a Matrix

dfs 思想，非常好的题目

```
class Solution:
    def longestIncreasingPath(self, matrix: List[List[int]]) -> int:
        row = len(matrix)
        col = len(matrix[0])
        dp = [[0 for _ in range(col)]for _ in range(row)]

        def dfs(i,j):
            if not dp[i][j]:
                val = matrix[i][j]
                #up
                if i!=0 and val > matrix[i-1][j]:
                    up = dfs(i-1,j)
                else:
                    up = 0

                #down
                if i!=row-1 and val > matrix[i+1][j]:
                    down = dfs(i+1,j)
                else:
                    down = 0

                #left
                if j != 0 and val > matrix[i][j-1]:
                    left = dfs(i,j-1)
                else:
                    left = 0

                #right
                if j != col-1 and val > matrix[i][j+1]:
                    right = dfs(i,j+1)
                else:
```

```
                    right = 0
                dp[i][j] = max(up,down,left,right)+1
            return dp[i][j]

        res_path = []
        for i in range(row):
            for j in range(col):
                res_path.append(dfs(i,j))
        return max(res_path)
```

931.Minimum Falling Path Sum

简单的动态规划题目 但是要注意边界。

而且这题明显可以使用一维dp

```
class Solution:
    def minFallingPathSum(self, matrix: List[List[int]]) -> int:
        n = len(matrix[0])
        if n==1:
            return matrix[0][0]
        #这样对于这题没用 因为对于最右边一列，他可以从上一列的右边取，此时已经溢出
        #使用这个只能解决最左边的问题
        dp = [[0 for _ in range(n)]for _ in range(n)]
        for i in range(n):
            dp[0][i] = matrix[0][i]

        for i in range(1,n):
            for j in range(n):
                if j==0:
                    dp[i][j] = matrix[i][j]+min(dp[i-1][j],dp[i-1][j+1])
                elif j==n-1:
                    dp[i][j] = matrix[i][j]+min(dp[i-1][j-1],dp[i-1][j])
                else:
                    dp[i][j] = matrix[i][j]+min(dp[i-1][j-1],dp[i-1][j],dp[i-1][j+1])
        return min(dp[n-1][:])
```

\120. Triangle

从下往上遍历 可以避免边界的讨论！ 最后的dp[0][0]就是结果！！！

空间优化：

```
class Solution:
    def minimumTotal(self, triangle: List[List[int]]) -> int:
        # 可以往下走 i或者i+1
        n = len(triangle)
        dp = [[0 for _ in range(n+1)] for _ in range(n+1)]
        for i in range(n-1,-1,-1):
            for j in range(0,i+1):
                dp[i][j] = min(dp[i+1][j+1],dp[i+1][j])+triangle[i][j]
        return dp[0][0]
```

```
class Solution:
    def minimumTotal(self, triangle: List[List[int]]) -> int:
        n = len(triangle)
        dp = [0] * (n+1)
        for i in range(n-1,-1,-1):
            for j in range(0,i+1):
                dp[j]= min(dp[j+1],dp[j])+triangle[i][j]
        return dp[0]
```

1884. Egg Drop With 2 Eggs and N Floors

更巧妙的方法是用数学