0.动态规划总结.md

- 常规问题
- [ ] 62 不同路径: $dp[i][j] = dp[i-1][j] + dp[i][j-1]$
- [ ] 63 不同路径 II : 同上，碰到障碍保持0
- [ ] 64 最小路径和：$dp[i][j] = min(dp[i-1][j], dp[i][j-1]) + grid[i]$[j]
- [ ] 70 爬楼梯：$dp[n] = dp[n-1] + dp[n-2]$，长度是n
- [ ] 343 整数拆分: $dp[i] = max(dp[i-j]*j, (i-j)*j)$，拆分or不拆分
- [ ] 509 斐波那契数 : $dp[n] = dp[n-1] + dp[n-2]$, 长度是n+1
- [ ] 746 使用最小花费爬楼梯:$dp[i] = min(dp[i-2], dp[i-1]) + cost[i]$, 最后返回$min(dp[-1], dp[-2])$
- [ ] 198 打家劫舍 $dp[i] = max(dp[i-1], dp[i-2] + nums[i-1])$
- [ ] 213 打家劫舍 II: 首尾相连，去除首/尾做两次计算
- [ ] 337 打家劫舍 III：dfs返回打劫/不打劫当前节点的两个rob数值
- 0/1背包：不能重复使用元素倒序背包

问题可以被分成： 求和，计数，是否存在组合，最大/最小组合数，组合数/排列数

- [ ] 416 分割等和子集：求和，全0初始化，内外皆可，$dp[j] = max(dp[j], dp[j-n] + n)$
- [ ] 474 一和零：计数，全0初始化，背包内物品外，$dp[i] = dp[i-cost] + 1$
- [ ] 494 目标和：组合数=target， dp[0]=1，背包内物品外，$dp[i] += dp[i-n]$
- [ ] 1049 最后一块石头的重量 II：求和，全0初始化，内外皆可，$dp[i] = max(dp[i], dp[i-s] + s)$
- 完全背包：可以重复使用元素正序背包
- [ ] 139 单词拆分：是否存在排列，dp[0]=True，背包外物品内，value=True, cost=单词匹配 dp[i]=dp[i] or (dp[i-cost] & is_s)
- [ ] 279 完全平方数：最小组合数，dp[0]=0（剩余位置最大化），物品外背包内，$dp[i] = min(dp[i], dp[i-num**2] + 1)$
- [ ] 322 零钱兑换：最小组合数，dp[0]=0(剩余位置最大化)，物品外背包内，$dp[i] = min(dp[i-coin] + 1, dp[i])$
- [ ] 377 组合总和 IV：组合数=target，dp[0]=1，物品外背包内，$dp[i] += dp[i-n]$
- [ ] 518 零钱兑换 II：组合数=target，dp[0]=1，物品外背包内，$dp[i] += dp[i-c]$
- 买卖股票
- [ ] 122 买卖股票的最佳时机 II：无数次
- [ ] 123 买卖股票的最佳时机 III: 2次
- [ ] 188 买卖股票的最佳时机 IV：k次
- [ ] 309 最佳买卖股票时机含冷冻期：把不持有状态区分成T，T-1,<T-2
- [ ] 714 买卖股票的最佳时机含手续费：持有收益考虑fee
- 子序列

- [ ] 72 编辑距离: 和583相比增加替换操作，相同$dp[i][j] = dp[i-1][j-1]$, 不同$dp[i][j] = min(dp[i][j-1], dp[i-1][j], dp[i-1][j-1]) + 1$
- [ ] 97 交错字符串: $dp[i][j] = (dp[i-1][j] and s1[i-1] == s3[pos]) or (dp[i][j-1] and s2[j-1] == s3[pos])$
- [ ] 115 不同的子序列：相同$dp[i][j] = dp[i-1][j] + dp[i-1][j-1]$, 不同$dp[i][j] = dp[i-1][j]$
- [ ] 300 最长递增子序列: $dp[i] = max(dp[i], dp[j] + 1)$,注意初始化都是1，且需要全局maxlen
- [ ] 392 判断子序列:只有删除操作，相同$dp[i][j] = dp[i-1][j-1] + 1$, 不同$dp[i][j] = dp[i][j-1]$, 判断最后长度是否为s
- [ ] 516 最长回文子序列：相同$dp[i][j] = dp[i+1][j-1] + 2$，不同$dp[i][j] = max(dp[i+1][j], dp[i][j-1])$, 左下到右上遍历
- [ ] 583 两个字符串的删除操作：和392相比是双向删除，注意初始化是序列长度，相同$dp[i][j] = dp[i-1][j-1]$, 不同$dp[i][j] = min(dp[i][j-1], dp[i-1][j]) + 1$
- [ ] 718 最长重复子数组：连续子序列$dp[i][j] = dp[i-1][j-1] + 1$，需要全局maxlen
- [ ] 1035 不相交的线：和1143相同
- [ ] 1143 最长公共子序列：不连续的子序列，相同$dp[i][j] = dp[i-1][j-1] + 1$，不同$dp[i][j] = max(dp[i-1][j], dp[i][j-1])$，最大长度一定是最后一位

1035. 不相交的线.md

在两条独立的水平线上按给定的顺序写下 nums1 和 nums2 中的整数。

现在，可以绘制一些连接两个数字 nums1[i] 和 nums2[j] 的直线，这些直线需要同时满足满足：

```
 nums1[i] == nums2[j]
且绘制的直线不与任何其他连线（非水平线）相交。
```

请注意，连线即使在端点也不能相交：每个数字只能属于一条连线。

以这种方法绘制线条，并返回可以绘制的最大连线数。

示例 1：

输入：nums1 = [1,4,2], nums2 = [1,2,4]
输出：2
解释：可以画出两条不交叉的线，如上图所示。
但无法画出第三条不相交的直线，因为从 nums1[1]=4 到 nums2[2]=4 的直线将与从 nums1[2]=2 到 nums2[1]=2 的直线相交。

示例 2：

输入：nums1 = [2,5,1,2,5], nums2 = [10,5,2,1,5,2]
输出：3

示例 3：

输入：nums1 = [1,3,7,1,7,5], nums2 = [1,9,2,5,1]
输出：2

提示：

```
1 <= nums1.length <= 500
1 <= nums2.length <= 500
1 <= nums1[i], nums2[i] <= 2000
```

```
class Solution:
    def maxUncrossedLines(self, nums1: List[int], nums2: List[int]) -> int:
        l1 = len(nums1)
        l2 = len(nums2)
        dp = [[0] * (l2+1) for i in range(l1+1)]
        max_len = 0
        for i in range(1, l1+1):
            for j in range(1, l2+1):
                if nums1[i-1]==nums2[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]
```

Tips

和1143最长公共子序列，因为线不想交，其实就是不改变在原始数组中的顺序

1049. 最后一块石头的重量 II.md

有一堆石头，用整数数组 stones 表示。其中 stones[i] 表示第 i 块石头的重量。

每一回合，从中选出任意两块石头，然后将它们一起粉碎。假设石头的重量分别为 x 和 y，且 x <= y。那么粉碎的可能结果如下：

```
如果 x == y，那么两块石头都会被完全粉碎；
如果 x != y，那么重量为 x 的石头将会完全粉碎，而重量为 y 的石头新重量为 y-x。
```

最后，最多只会剩下一块 石头。返回此石头 最小的可能重量 。如果没有石头剩下，就返回 0。

示例 1：

输入：stones = [2,7,4,1,8,1]
输出：1
解释：
组合 2 和 4，得到 2，所以数组转化为 [2,7,1,8,1]，
组合 7 和 8，得到 1，所以数组转化为 [2,1,1,1]，
组合 2 和 1，得到 1，所以数组转化为 [1,1,1]，
组合 1 和 1，得到 0，所以数组转化为 [1]，这就是最优值。

示例 2：

输入：stones = [31,26,33,21,40]
输出：5

示例 3：

输入：stones = [1,2]
输出：1

提示：

```
1 <= stones.length <= 30
1 <= stones[i] <= 100
```

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

Tips

1. 0/1背包问题，和416题解法一模一样

1143. 最长公共子序列.md

给定两个字符串 text1 和 text2，返回这两个字符串的最长 公共子序列 的长度。如果不存在 公共子序列 ，返回 0 。

一个字符串的 子序列 是指这样一个新的字符串：它是由原字符串在不改变字符的相对顺序的情况下删除某些字符（也可以不删除任何字符）后组成的新字符串。

```
例如，"ace" 是 "abcde" 的子序列，但 "aec" 不是 "abcde" 的子序列。
```

两个字符串的 公共子序列 是这两个字符串所共同拥有的子序列。

示例 1：

输入：text1 = "abcde", text2 = "ace"
输出：3
解释：最长公共子序列是 "ace" ，它的长度为 3 。

示例 2：

输入：text1 = "abc", text2 = "abc"
输出：3
解释：最长公共子序列是 "abc" ，它的长度为 3 。

示例 3：

输入：text1 = "abc", text2 = "def"
输出：0
解释：两个字符串没有公共子序列，返回 0 。

提示：

```
1 <= text1.length, text2.length <= 1000
text1 和 text2 仅由小写英文字符组成。
```

```
class Solution:
    def longestCommonSubsequence(self, text1: str, text2: str) -> int:
        max_len = 0
        l1 = len(text1)
        l2 = len(text2)
        dp = [[0] *(l2+1)for i in range(l1+1)]
        for i in range(1, l1+1):
            for j in range(1, l2+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]
```

Tips

和718题不同的时这里不要求连续，所以会多一个状态转移，并且因为当前状态可以一直向后传递所以也不需要max_len来记录遍历过的最大长度，数组最后一个元素就是最大长度

- $dp[i][j] = max(dp[i-1][j], dp[i][j-1])$

115. 不同的子序列.md

####

给定一个字符串 `s` 和一个字符串 `t` ，计算在 `s` 的子序列中 `t` 出现的个数。

字符串的一个 **子序列** 是指，通过删除一些（也可以不删除）字符且不干扰剩余字符相对位置所组成的新字符串。（例如，`"ACE"` 是 `"ABCDE"` 的一个子序列，而 `"AEC"` 不是）

题目数据保证答案符合 32 位带符号整数范围。

示例 1：

```
输入：s = "rabbbit", t = "rabbit"
输出：3
解释：
如下图所示, 有 3 种可以从 s 中得到 "rabbit" 的方案。
rabbbit
rabbbit
rabbbit
```

示例 2：

```
输入：s = "babgbag", t = "bag"
输出：5
解释：
如下图所示, 有 5 种可以从 s 中得到 "bag" 的方案。
babgbag
babgbag
babgbag
babgbag
babgbag
```

提示：

- `0 <= s.length, t.length <= 1000`
- `s` 和 `t` 由英文字母组成

```python
class Solution:
    def numDistinct(self, s: str, t: str) -> int:
        ls = len(s)
        lt = len(t)
        dp = [[1]+[0] *(lt)for i in range(ls+1)]
        for i in range(1, ls+1):
            for j in range(1, lt+1):
                if s[i-1]==t[j-1]:
                    dp[i][j] = dp[i-1][j-1] + dp[i-1][j]
                else:
                    dp[i][j] = dp[i-1][j]
        return dp[-1][-1]
```

Tips

动态规划

- $dp[i][j]$含义是$s[i-1]$中出现$t[j-1]$的次数
- 初始化dp[i][0]=1
- 状态转移：
  - s[j-1]=t[j-1]：$dp[i][j] = dp[i-1][j-1] + dp[i-1][j]$ 这里是继承j-2和i-2的状态，以及j-1和i-2的状态（继承之前就匹配当前字符的状态
  - s[j-1]!=t[j-1]：$dp[i][j] = dp[i-1][j-1]$

1155. 掷骰子的N种方法.md

这里有 d 个一样的骰子，每个骰子上都有 f 个面，分别标号为 1, 2, ..., f。

我们约定：掷骰子的得到总点数为各骰子面朝上的数字的总和。

如果需要掷出的总点数为 target，请你计算出有多少种不同的组合情况（所有的组合情况总共有 f^d 种），模 10^9 + 7 后返回。

示例 1：

输入：d = 1, f = 6, target = 3
输出：1

示例 2：

输入：d = 2, f = 6, target = 7
输出：6

示例 3：

输入：d = 2, f = 5, target = 10
输出：1

示例 4:

输入：d = 1, f = 2, target = 3
输出：0

示例 5:

输入：d = 30, f = 30, target = 500
输出：222616187

提示：

```
1 <= d, f <= 30
1 <= target <= 1000
```

123. 买卖股票的最佳时机 III.md

给定一个数组，它的第 i 个元素是一支给定的股票在第 i 天的价格。

设计一个算法来计算你所能获取的最大利润。你最多可以完成 两笔 交易。

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 = [3,3,5,0,0,3,1,4]
输出：6
解释：在第 4 天（股票价格 = 0）的时候买入，在第 6 天（股票价格 = 3）的时候卖出，这笔交易所能获得利润 = 3-0 = 3 。
　　随后，在第 7 天（股票价格 = 1）的时候买入，在第 8 天 （股票价格 = 4）的时候卖出，这笔交易所能获得利润 = 4-1 = 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：在这个情况下, 没有交易完成, 所以最大利润为 0。

示例 4:

输入：prices = [1]
输出：0

提示：

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

```
class Solution:
    def maxProfit(self, prices: List[int]) -> int:
        l = len(prices)
        dp1 = [0] * l
        dp2 =[0] * l
        dp3 =[0] * l
        dp4 =[0] * l
        dp1[0] = -prices[0]
        dp3[0] = -prices[0]
        for i in range(1,l):
            dp1[i] = max(dp1[i-1], -prices[i])
            dp2[i] = max(dp2[i-1], dp1[i-1]+prices[i])
            dp3[i] = max(dp3[i-1], dp2[i-1]-prices[i])
            dp4[i] = max(dp4[i-1], dp3[i-1]+prices[i])
        return dp4[-1]
```

Tips

1. 4个状态，第一次买入/卖出，第二次买入/卖出持有现金
2. 每一步状态转移
3. dp1[i] = max(dp1[i-1], -prices[i])
   dp2[i] = max(dp2[i-1], dp1[i-1]+prices[i])
   dp3[i] = max(dp3[i-1], dp2[i-1]-prices[i])
   dp4[i] = max(dp4[i-1], dp3[i-1]+prices[i])
4. 初始化，买入状态的初始化永远是第一日成本

139. 单词拆分.md

给定一个非空字符串 s 和一个包含非空单词的列表 wordDict，判定 s 是否可以被空格拆分为一个或多个在字典中出现的单词。

说明：

```
拆分时可以重复使用字典中的单词。
你可以假设字典中没有重复的单词。
```

示例 1：

输入: s = "leetcode", wordDict = ["leet", "code"]
输出: true
解释: 返回 true 因为 "leetcode" 可以被拆分成 "leet code"。

示例 2：

输入: s = "applepenapple", wordDict = ["apple", "pen"]
输出: true
解释: 返回 true 因为 "applepenapple" 可以被拆分成 "apple pen apple"。
  注意你可以重复使用字典中的单词。

示例 3：

输入: s = "catsandog", wordDict = ["cats", "dog", "sand", "and", "cat"]
输出: false

```
class Solution:
    def wordBreak(self, s: str, wordDict: List[str]) -> bool:
        ls = len(s)
        dp = [False]* (ls+1)
        dp[0]=True
        wordDict = set(wordDict)
        for i in range(1,len(s)+1):
            for word in wordDict:
                lw = len(word)
                if i < lw:
                    continue
                else:
                    if s[i-lw:i] == word and dp[i-lw]:
                        dp[i]=True
                        break
        return dp[-1]
```

Tips

1. 完全背包问题，物品的wordDict，weight是单词本身，value也是单词本身，求安特定顺序是否能装满背包
2. dp初始化,dp[0]=True, 其余都是False
3. 每一步

$$dp[i] = dp[i - len(word)] and \left(s[(i - len(word)) : i] == word\right)$$

4. 其实是一个隐形的排列问题，因为只能按照字符串的顺序来拼接wordDict，所以必须外层遍历背包，内层遍历物品。因为是完全背包，遍历背包从前向后

188. 买卖股票的最佳时机 IV.md

给定一个整数数组 prices ，它的第 i 个元素 prices[i] 是一支给定的股票在第 i 天的价格。

设计一个算法来计算你所能获取的最大利润。你最多可以完成 k 笔交易。

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：k = 2, prices = [2,4,1]
输出：2
解释：在第 1 天 (股票价格 = 2) 的时候买入，在第 2 天 (股票价格 = 4) 的时候卖出，这笔交易所能获得利润 = 4-2 = 2 。

示例 2：

输入：k = 2, prices = [3,2,6,5,0,3]
输出：7
解释：在第 2 天 (股票价格 = 2) 的时候买入，在第 3 天 (股票价格 = 6) 的时候卖出, 这笔交易所能获得利润 = 6-2 = 4 。
随后，在第 5 天 (股票价格 = 0) 的时候买入，在第 6 天 (股票价格 = 3) 的时候卖出, 这笔交易所能获得利润 = 3-0 = 3 。

提示：

```
0 <= k <= 100
0 <= prices.length <= 1000
0 <= prices[i] <= 1000
```

```python
class Solution:
    def maxProfit(self, k: int, prices: List[int]) -> int:
        if not prices:
            return 0
        #Initialize
        l = len(prices)
        dp = [[0]*l for i in range(2*k+1)]
        for i in range(1, 2*k+1,2):
            dp[i][0] = -prices[0]

        for i in range(1, l ):
            for j in range(1,2*k+1,2):
                dp[j][i] = max(dp[j][i-1], dp[j-1][i-1]-prices[i])
                dp[j+1][i] = max(dp[j+1][i-1], dp[j][i-1]+prices[i])
        return dp[2*k][-1]
```

Tips

其实两次以下都用了偷懒的写法，就是省略了无操作的状态，所以第一次买入持有现金的dp，状态转移是如下的

- dp1[i] = max(dp1[i-1], -prices[i])

但到K次买卖我们需要用数组来实现的时候，需要一个无操作状态，来保证所有状态转移是相同的

```
dp[j][i] = max(dp[j][i-1], dp[j-1][i-1]-prices[i])
dp[j+1][i] = max(dp[j+1][i-1], dp[j][i-1]+prices[i])
```

198. 打家劫舍.md

你是一个专业的小偷，计划偷窃沿街的房屋。每间房内都藏有一定的现金，影响你偷窃的唯一制约因素就是相邻的房屋装有相互连通的防盗系统，如果两间相邻的房屋在同一晚上被小偷闯入，系统会自动报警。

给定一个代表每个房屋存放金额的非负整数数组，计算你 不触动警报装置的情况下 ，一夜之内能够偷窃到的最高金额。

示例 1：

输入：[1,2,3,1]
输出：4
解释：偷窃 1 号房屋 (金额 = 1) ，然后偷窃 3 号房屋 (金额 = 3)。
　　偷窃到的最高金额 = 1 + 3 = 4 。

示例 2：

输入：[2,7,9,3,1]
输出：12
解释：偷窃 1 号房屋 (金额 = 2), 偷窃 3 号房屋 (金额 = 9)，接着偷窃 5 号房屋 (金额 = 1)。
　　偷窃到的最高金额 = 2 + 9 + 1 = 12 。

提示：

```
1 <= nums.length <= 100
0 <= nums[i] <= 400
```

```python
class Solution:
    def rob(self, nums: List[int]) -> int:
        l = len(nums)
        if l<=1:
            return nums[-1]
        dp = [0] * (l+1)
        dp[1] = nums[0]
        for i in range(2, l+1):
            dp[i] = max(dp[i-1], dp[i-2]+nums[i-1])
        return dp[-1]
```

Tips

1. 初始化这里有一点需要注意的，就是初始长度是len(num)+1，这里是针对只有两个元素的nums，如果只初始len(nums)就需要对dp[1]进行特殊处理了。dp[0]=0,dp[1]=nums[0]
2. 每一步，这里因为加入了dp[0]，所以有一步错位

$$dp[i] = max(dp[i-1], dp[i-2] + nums[i-1])$$

3. 递归顺序，因为dp[i]由前两个元素推导出来所以一定是向前遍历

213. 打家劫舍 II.md

你是一个专业的小偷，计划偷窃沿街的房屋，每间房内都藏有一定的现金。这个地方所有的房屋都 围成一圈 ，这意味着第一个房屋和最后一个房屋是紧挨着的。同时，相邻的房屋装有相互连通的防盗系统，如果两间相邻的房屋在同一晚上被小偷闯入，系统会自动报警 。

给定一个代表每个房屋存放金额的非负整数数组，计算你 在不触动警报装置的情况下 ，今晚能够偷窃到的最高金额。

示例 1：

输入：nums = [2,3,2]
输出：3
解释：你不能先偷窃 1 号房屋（金额 = 2），然后偷窃 3 号房屋（金额 = 2）, 因为他们是相邻的。

示例 2：

输入：nums = [1,2,3,1]
输出：4
解释：你可以先偷窃 1 号房屋（金额 = 1），然后偷窃 3 号房屋（金额 = 3）。
　偷窃到的最高金额 = 1 + 3 = 4 。

示例 3：

输入：nums = [0]
输出：0

提示：

```
1 <= nums.length <= 100
0 <= nums[i] <= 1000
```

```python
class Solution:
    def rob(self, nums: List[int]) -> int:
        def helper(nums):
            l = len(nums)
            dp = [0] * (l+1)
            dp[1] = nums[0]
            for i in range(2, l+1):
                dp[i] = max(dp[i-1], dp[i-2]+ nums[i-1])
            return dp[-1]

        if len(nums)==1:
            return nums[0]
        res1 = helper(nums[1:])
        res2 = helper(nums[:-1])
        return max(res1, res2)
```

1. 和打家劫舍1的动态转一部分是一样的，只不过因为连成环，所以出现两种选择，不打劫第一个，和不打劫最后一个，剩余的部分都不会连成环所以就和第一题相同了

279. 完全平方数.md

给定正整数 n，找到若干个完全平方数（比如 1, 4, 9, 16, ...）使得它们的和等于 n。你需要让组成和的完全平方数的个数最少。

给你一个整数 n ，返回和为 n 的完全平方数的 最少数量 。

完全平方数 是一个整数，其值等于另一个整数的平方；换句话说，其值等于一个整数自乘的积。例如，1、4、9 和 16 都是完全平方数，而 3 和 11 不是。

示例 1：

输入：n = 12
输出：3
解释：12 = 4 + 4 + 4

示例 2：

输入：n = 13
输出：2
解释：13 = 4 + 9

提示：

```
1 <= n <= 104
```

```python
class Solution:
    def numSquares(self, n: int) -> int:
        dp = [n+1] *(n+1)
        dp[0] = 0
        for num in range(1, int(n**0.5)+1):
            for i in range(num**2, n+1):
                dp[i] = min(dp[i], dp[i-num**2]+1)
        return dp[-1]
```

Tips

1. 完全背包求最小物品数的问题，value=num，weight=num，背包大小是n
2. dp初始化，min问题的初始化都是用大于背包大小的值来初始化，dp[0]=0
3. 每一步

$$dp[i] = min(dp[i], dp[i - num] + 1)$$

4. 因为是完全背包，遍历背包从前向后。求min的问题，先遍历背包还是先遍历物品都无所谓

300. 最长递增子序列.md

给你一个整数数组 nums ，找到其中最长严格递增子序列的长度。

子序列是由数组派生而来的序列，删除（或不删除）数组中的元素而不改变其余元素的顺序。例如，[3,6,2,7] 是数组 [0,3,1,6,2,2,7] 的子序列。

示例 1：

输入：nums = [10,9,2,5,3,7,101,18]
输出：4
解释：最长递增子序列是 [2,3,7,101]，因此长度为 4 。

示例 2：

输入：nums = [0,1,0,3,2,3]
输出：4

示例 3：

输入：nums = [7,7,7,7,7,7,7]
输出：1

提示：

```
1 <= nums.length <= 2500
-104 <= nums[i] <= 104
```

进阶：

```
你可以设计时间复杂度为 O(n2) 的解决方案吗?
你能将算法的时间复杂度降低到 O(n log(n)) 吗?
```

1. O(n^2)时间复杂度

```python
class Solution:
    def lengthOfLIS(self, nums: List[int]) -> int:
        if not nums:
            return 0
        l = len(nums)
        dp = [1] * l
        for i in range(1, l):
            for j in range(0, i):
                if nums[i] > nums[j]:
                    dp[i]= max(dp[i],dp[j]+1)
        return max(dp)
```

Tips

- 状态转移：每个元素i，都和之前的所有元素进行对比，如果nums[i]>nums[j]对dp[j]++
- 初始化：因为是子序列长度，所以dp初始化都是1！！
- 注意因为状态会overwrite所以需要每一步都判断max

2. O(nlogn)时间复杂度，肯定需要二分查找了

```

```

309. 最佳买卖股票时机含冷冻期.md

给定一个整数数组，其中第 i 个元素代表了第 i 天的股票价格 。

设计一个算法计算出最大利润。在满足以下约束条件下，你可以尽可能地完成更多的交易（多次买卖一支股票）:

```
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）。
卖出股票后，你无法在第二天买入股票 (即冷冻期为 1 天)。
```

示例:

输入: [1,2,3,0,2]
输出: 3
解释: 对应的交易状态为: [买入, 卖出, 冷冻期, 买入, 卖出]

```python
class Solution:
    def maxProfit(self, prices: List[int]) -> int:
        l = len(prices)
        dp1 = [0] * l # 持有现金
        dp2 = [0] * l # 不持有，当天卖出现金 【T=0】
        dp3 = [0] * l # 不持有，今天为冻结期 【T=-1】
        dp4 = [0] * l # 不持有，已度过冷冻期 【T<-1】
        dp1[0] = -prices[0]

        for i in range(1,l):
            dp1[i] = max(dp1[i-1], max(dp4[i-1],dp3[i-1]) - prices[i])
            dp2[i] = dp1[i-1]+prices[i]
            dp3[i] = dp2[i-1]
            dp4[i] = max(dp4[i-1], dp3[i-1])
            print(dp1[i],dp2[i],dp3[i],dp4[i])
        return max(dp4[-1],dp3[-1],dp2[-1])
```

Tips

zhe

322. 零钱兑换.md

给你一个整数数组 `coins` ，表示不同面额的硬币；以及一个整数 `amount` ，表示总金额。

计算并返回可以凑成总金额所需的 **最少的硬币个数** 。如果没有任何一种硬币组合能组成总金额，返回 `-1` 。

你可以认为每种硬币的数量是无限的。

示例 1：

```
输入: coins = [1, 2, 5], amount = 11
输出: 3
解释: 11 = 5 + 5 + 1
```

示例 2:

```
输入: coins = [2], amount = 3
输出: -1
```

示例 3:

```
输入: coins = [1], amount = 0
输出: 0
```

示例 4:

```
输入: coins = [1], amount = 1
输出: 1
```

示例 5:

```
输入: coins = [1], amount = 2
输出: 2
```

提示:

- `1 <= coins.length <= 12`
- `1 <= coins[i] <= 231 - 1`
- `0 <= amount <= 104`

```python
class Solution:
    def coinChange(self, coins: List[int], amount: int) -> int:
        dp = [amount+1] * (amount+1)
        dp[0] = 0
        for c in coins:
            for i in range(c,amount+1):
                dp[i] = min(dp[i-c]+1, dp[i])
        if dp[-1]>amount:
            return -1
        else:
            return dp[-1]
```

Tips

1. 完全背包中的最小化问题，value=coin，weight=coin，背包weight=amount，求装满背包所需的最少物品
2. dp初始化，dp[0]=0,其余=amount+1，为了在求min的过程中不覆盖原值，必须用一个大于所有可能只的值
3. 每一步计算

$$dp[i] = min(dp[i - coin] + 1, dp[i])$$

4. 遍历顺序，因为是完全背包，所以遍历dp时从前向后，使得每个coin可以使用多次。因为没有组合/排列的要求，所以遍历物品和背包的顺序谁先谁后都可以

337. 打家劫舍 III.md

在上次打劫完一条街道之后和一圈房屋后，小偷又发现了一个新的可行窃的地区。这个地区只有一个入口，我们称之为"根"。 除了"根"之外，每栋房子有且只有一个"父"房子与之相连。一番侦察之后，聪明的小偷意识到"这个地方的所有房屋的排列类似于一棵二叉树"。 如果两个直接相连的房子在同一天晚上被打劫，房屋将自动报警。

计算在不触动警报的情况下，小偷一晚能够盗取的最高金额。

示例 1:

输入: [3,2,3,null,3,null,1]

```
 3
/ \
```

2 3
\ \
3 1

输出: 7
解释: 小偷一晚能够盗取的最高金额 = 3 + 3 + 1 = 7.

示例 2:

输入: [3,4,5,1,3,null,1]

```
 3
/ \
```

4 5
/\ \
1 3 1

输出: 9
解释: 小偷一晚能够盗取的最高金额 = 4 + 5 = 9.

```
# Definition for a binary tree node.
# class TreeNode:
#     def __init__(self, val=0, left=None, right=None):
#         self.val = val
```

```python
#         self.left = left
#         self.right = right
class Solution:
    def rob(self, root: TreeNode) -> int:
        def dfs(root):
            if not root:
                return (0,0)
            left = dfs(root.left)
            right = dfs(root.right)
            rob_current = root.val + left[1] + right[1]
            not_rob_current = max(left[0],left[1]) + max(right[0], right[1])
            return rob_current, not_rob_current

        res = dfs(root)
        return max(res[0], res[1])
```

Tips

1. 每个节点都可以选择投当前节点，则一定不偷子节点，不偷当前节点，则可以选择偷/或者不偷子节点
2. 递归顺序因为每个parent需要用到children状态所以是后序遍历，递归的入参就是每个节点，但是出参因为需要区分偷/不偷当前节点所有会有两个出参
3. 每一步的状态转移都是
    1. rob_current = root.val + not_rob_left + not_rob_right
    2. Not_rob_current = max(rob_left, not_rob_left) + max(rob_right, not_rob_right)

343. 整数拆分.md

给定一个正整数 n，将其拆分为至少两个正整数的和，并使这些整数的乘积最大化。 返回你可以获得的最大乘积。

示例 1:

输入: 2
输出: 1
解释: 2 = 1 + 1, 1 × 1 = 1。

示例 2:

输入: 10
输出: 36
解释: 10 = 3 + 3 + 4, 3 × 3 × 4 = 36。

说明: 你可以假设 n 不小于 2 且不大于 58。

```python
class Solution:
    def integerBreak(self, n: int) -> int:
        dp = [0] * (n+1)
        dp[2] = 1
        for i in range(3,n+1):
            for j in range(1,i-1):
                dp[i] =max(dp[i], max(dp[i-j] * j, (i-j) * j))
                print(dp)
        return dp[-1]
```

Tips

1. 时间复杂度O(n^2), 空间复杂度O(n)
2. dp[i]的含义是拆分成出至少两个正整数的乘积的最大值，所以dp[2]=1
3. 状态转移为遍历小于i-1，的所有数值计算拆分或者不拆分两个数字的乘积的最大值，然后更新当前状态，每个当前状态都需要遍历之前的i-1个数值菜鞥得到，所以不能只保存两个状态需要保留全部状态

$$dp[i] = max(dp[i-j]*j, (i-j)*j)$$

4. 初始化，这里选择直接初始化dp[2]=1, 前面的0和1是哦否初始化都没有安息，因为在计算中可以让j从1开始也就可以避开dp[0]和dp[1]

377. 组合总和 IV.md

给你一个由 不同 整数组成的数组 nums ，和一个目标整数 target 。请你从 nums 中找出并返回总和为 target 的元素组合的个数。

题目数据保证答案符合 32 位整数范围。

示例 1：

输入：nums = [1,2,3], target = 4
输出：7
解释：
所有可能的组合为：
(1, 1, 1, 1)
(1, 1, 2)
(1, 2, 1)
(1, 3)
(2, 1, 1)
(2, 2)
(3, 1)
请注意，顺序不同的序列被视作不同的组合。

示例 2：

输入：nums = [9], target = 3
输出：0

提示：

```
1 <= nums.length <= 200
1 <= nums[i] <= 1000
nums 中的所有元素 互不相同
1 <= target <= 1000
```

1. nums都是正数

```
class Solution:
    def combinationSum4(self, nums: List[int], target: int) -> int:
        dp = [0] * (target+1)
        dp[0]=1
        for i in range(target+1):
            for n in nums:
                if i-n>=0:
                    dp[i] +=dp[i-n]
        return dp[-1]
```

Tips

1. 完全背包中求排列数，value=num, weight=num, 背包weight=target,求装满背包的组合数
2. dp初始化，dp[0]=1， 其余为0
3. 每步计算

$$dp[i] += dp[i - num]$$

4. 遍历顺序，因为是完全背包，遍历背包从前向后，因为是求排列数，所以外层遍历背包，内层遍历物品

2. 如果存在负数

392. 判断子序列.md

给定字符串 s 和 t ，判断 s 是否为 t 的子序列。

字符串的一个子序列是原始字符串删除一些（也可以不删除）字符而不改变剩余字符相对位置形成的新字符串。（例如，"ace"是"abcde"的一个子序列，而"aec"不是）。

进阶：

如果有大量输入的 S，称作 S1, S2, ... , Sk 其中 k >= 10亿，你需要依次检查它们是否为 T 的子序列。在这种情况下，你会怎样改变代码？

致谢：

特别感谢 @pbrother 添加此问题并且创建所有测试用例。

示例 1：

输入：s = "abc", t = "ahbgdc"
输出：true

示例 2：

输入：s = "axc", t = "ahbgdc"
输出：false

提示：

```
0 <= s.length <= 100
0 <= t.length <= 10^4
两个字符串都只由小写字符组成。
```

1. 双指针解法，逐个向前遍历判断是否相同

```python
class Solution:
    def isSubsequence(self, s: str, t: str) -> bool:
        i = 0
        j = 0
        n1 = len(s)
        n2 = len(t)
        while (i <n1) and (j < n2):
            if s[i]==t[j]:
                i+=1
                j+=1
            else:
                j+=1
        if i < n1:
            return False
        else:
            return True
```

2. 动态规划解法

- 初始化都是0

- $dp[i][j]$是$s[i-1]t[j-1]$ 匹配上的字符个数
- 转移
  - $s[i-1]==t[j-1]:dp[i][j]=dp[i-1][j-1]+1$
  - $s[i-1]!=t[j-1]:dp[i][j]=dp[i][j-1]$ 删除t[j-1]，继承t[j-2]的匹配结果
- 结果判断$dp[-1][-1]==len()$

416. 分割等和子集.md

给你一个 只包含正整数 的 非空 数组 nums 。请你判断是否可以将这个数组分割成两个子集，使得两个子集的元素和相等。

示例 1：

输入：nums = [1,5,11,5]
输出：true
解释：数组可以分割成 [1, 5, 5] 和 [11] 。

示例 2：

输入：nums = [1,2,3,5]
输出：false
解释：数组不能分割成两个元素和相等的子集。

提示：

```
1 <= nums.length <= 200
1 <= nums[i] <= 100
```

```python
class Solution:
    def canPartition(self, nums: List[int]) -> bool:
        total = sum(nums)
        if total %2 ==1:
            return False
        l = total//2
        dp = [0] * (l+1)
        for n in nums:
            for j in range(l,n-1,-1):
                dp[j] = max(dp[j], dp[j-n]+n)
        return dp[-1]==l
```

Tips

1. 01背包问题，weight是数值本身，value也是数值本身
2. dp的长度是total+1, 初始化都是0，每一步迭代

$$dp[i] = max(dp[i], dp[i - num] + num)$$

3. 传统背包求和问题，内外层遍历的顺序可以互换不会影响结果
4. 所有01问题遍历dp的方向都是从后向前，避免同一个元素被使用多次

474. 一和零.md

给你一个二进制字符串数组 strs 和两个整数 m 和 n 。

请你找出并返回 strs 的最大子集的长度，该子集中 最多 有 m 个 0 和 n 个 1 。

如果 x 的所有元素也是 y 的元素，集合 x 是集合 y 的 子集 。

示例 1：

输入：strs = ["10", "0001", "111001", "1", "0"], m = 5, n = 3
输出：4
解释：最多有 5 个 0 和 3 个 1 的最大子集是 {"10","0001","1","0"} ，因此答案是 4 。
其他满足题意但较小的子集包括 {"0001","1"} 和 {"10","1","0"} 。{"111001"} 不满足题意，因为它含 4 个 1 ，大于 n 的值 3 。

示例 2：

输入：strs = ["10", "0", "1"], m = 1, n = 1
输出：2
解释：最大的子集是 {"0", "1"} ，所以答案是 2 。

提示：

```
1 <= strs.length <= 600
1 <= strs[i].length <= 100
strs[i] 仅由 '0' 和 '1' 组成
1 <= m, n <= 100
```

```python
class Solution:
    def findMaxForm(self, strs: List[str], m: int, n: int) -> int:
        dp = [[0] * (n+1) for i in range(m+1)]
        for s in strs:
            m0 = s.count('0')
            n1 = s.count('1')
            for i in range(m,m0-1,-1):
                for j in range(n,n1-1,-1):
                    dp[i][j] = max(dp[i][j], dp[i-m0][j-n1]+1)
        return dp[-1][-1]
```

Tips

1. 二维0/1背包问题，weight是二维的（0的个数，1的个数）,value是1

494. 目标和.md

给你一个整数数组 nums 和一个整数 target 。

向数组中的每个整数前添加 '+' 或 '-' ，然后串联起所有整数，可以构造一个 表达式 ：

```
例如，nums = [2, 1] ，可以在 2 之前添加 '+' ，在 1 之前添加 '-' ，然后串联起来得到表达式 "+2-1"
。
```

返回可以通过上述方法构造的、运算结果等于 target 的不同 表达式 的数目。

示例 1：

输入：nums = [1,1,1,1,1], target = 3
输出：5
解释：一共有 5 种方法让最终目标和为 3 。
-1 + 1 + 1 + 1 + 1 = 3
+1 - 1 + 1 + 1 + 1 = 3
+1 + 1 - 1 + 1 + 1 = 3
+1 + 1 + 1 - 1 + 1 = 3
+1 + 1 + 1 + 1 - 1 = 3

示例 2：

输入：nums = [1], target = 1
输出：1

提示：

```
1 <= nums.length <= 20
0 <= nums[i] <= 1000
0 <= sum(nums[i]) <= 1000
-1000 <= target <= 1000
```

```python
class Solution:
    def findTargetSumWays(self, nums: List[int], target: int) -> int:
        val = sum(nums)-target
        if val%2==1 or val <0:
            return 0
        val = val//2
        dp = [0] * (val+1)
        dp[0]=1
        for n in nums:
            for i in range(val, n-1,-1):
                dp[i]+=dp[i-n]
        return dp[-1]
```

Tips

1. 转换为从nums中挑选部分个元素，使得两部分的元素之差为target.A+B=sum(nums),A-B=target,B = (sum(nums)-target)//2
2. 非传统背包问题，组合计数类的背包问题，通用迭代方案如下，初始化一般dp[0]=1，因为需要有初始值进行累加

$$dp[i] += dp[i - weight]$$

509. 斐波那契数.md

斐波那契数，通常用 F(n) 表示，形成的序列称为 斐波那契数列 。该数列由 0 和 1 开始，后面的每一项数字都是前面两项数字的和。也就是：

F(0) = 0，F(1) = 1
F(n) = F(n - 1) + F(n - 2)，其中 n > 1

给你 n ，请计算 F(n) 。

示例 1：

输入：2
输出：1
解释：F(2) = F(1) + F(0) = 1 + 0 = 1

示例 2：

输入：3
输出：2
解释：F(3) = F(2) + F(1) = 1 + 1 = 2

示例 3：

输入：4
输出：3
解释：F(4) = F(3) + F(2) = 2 + 1 = 3

提示：

```
0 <= n <= 30
```

1. 常规递归解法，找到每一步的状态转移对于斐波那契数列来说dp[n]=dp[n-1]+dp[n-2]，以及最终的停止条件。计算相对低效因为再递归过程中会出现相同n被计算很多次的情况。时间复杂度O(n)

```
class Solution:
    def fib(self, n: int) -> int:
        def helper(n):
            if n==0:
                return 0
            if n==1:
                return 1
            return helper(n-1)+ helper(n-2)
        return helper(n)
```

2. 加入dic来保存曾经计算过的dp[n]

```
class Solution:
    def fib(self, n: int) -> int:
        dic = {}
        def helper(n):
            if n==0:
                return 0
            if n==1:
                return 1
            if n in dic:
                return dic[n]
            val = helper(n-1)+ helper(n-2)
            dic[n] = val
            return val
        return helper(n)
```

3. 动态规划

```
class Solution:
    def fib(self, n: int) -> int:
        dp= [0,1]
        if n<=1:
            return dp[n]
        for i in range(2,n+1):
            dp.append(dp[i-1]+dp[i-2])
        return dp[-1]
```

默认状态下动态规划的时间复杂度都是O(n), n是dp的长度，空间复杂度是O(1)/O(n)取决于用哪种写法

516. 最长回文子序列.md

给你一个字符串 s ，找出其中最长的回文子序列，并返回该序列的长度。

子序列定义为：不改变剩余字符顺序的情况下，删除某些字符或者不删除任何字符形成的一个序列。

示例 1：

输入：s = "bbbab"
输出：4
解释：一个可能的最长回文子序列为 "bbbb" 。

示例 2：

输入：s = "cbbd"
输出：2
解释：一个可能的最长回文子序列为 "bb" 。

提示：

```
1 <= s.length <= 1000
s 仅由小写英文字母组成
```

```python
class Solution:
    def longestPalindromeSubseq(self, s: str) -> int:
        l = len(s)
        dp = [[0] * l for _ in range(l)]
        for i in range(l):
            dp[i][i] = 1
        for i in range(l-1,-1,-1):
            for j in range(i+1,l):
                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]
```

Tips

1. $dp[i][j]$是【i，j】之间回文子串的长度
2. 状态转移
    1. s[i]==s[j]: $dp[i][j] = dp[i+1][j-1] + 2$
    2. s[i]!=s[j]: 向里各收缩一位继承最长的内部子串长度 $dp[i][j] = max(dp[i+1][j], dp[i][j-1])$
3. 因为以上不等的状态需要继承内部状态，所以需要把i==j的部分先设成1, 对角线确实都是回文子串
4. 根据dp状态转移是从左下到忧伤，所以必须从bottom到top进行遍历，只用遍历半矩阵。最大值在右上角

518. 零钱兑换 II.md

给你一个整数数组 coins 表示不同面额的硬币，另给一个整数 amount 表示总金额。

请你计算并返回可以凑成总金额的硬币组合数。如果任何硬币组合都无法凑出总金额，返回 0 。

假设每一种面额的硬币有无限个。

题目数据保证结果符合 32 位带符号整数。

示例 1：

输入：amount = 5, coins = [1, 2, 5]
输出：4
解释：有四种方式可以凑成总金额：
5=5
5=2+2+1
5=2+1+1+1
5=1+1+1+1+1

示例 2：

输入：amount = 3, coins = [2]
输出：0
解释：只用面额 2 的硬币不能凑成总金额 3 。

示例 3：

输入：amount = 10, coins = [10]
输出：1

提示：

```
1 <= coins.length <= 300
1 <= coins[i] <= 5000
coins 中的所有值 互不相同
0 <= amount <= 5000
```

```python
class Solution:
    def change(self, amount: int, coins: List[int]) -> int:
        dp = [0] * (amount+1)
        dp[0] = 1
        for c in coins:
            for i in range(c,amount+1):
                dp[i] += dp[i-c]
        return dp[-1]
```

Tips

1. 完全背包求组合数, value=coin，weight=coin，背包大小是amount，求装满背包的组合数
2. dp初始化，dp[0]=1，因为需要在每步迭代中累加所以需要初始值，其余为0
3. 每步dp

$$dp[i] += dp[i-c]$$

4. 遍历顺序，因为是完全背包所以dp的遍历从前向后，因为是组合数而非排列数，所以外层遍历物品，内层遍历背包

583. 两个字符串的删除操作.md

给定两个单词 word1 和 word2，找到使得 word1 和 word2 相同所需的最小步数，每步可以删除任意一个字符串中的一个字符。

示例：

输入: "sea", "eat"
输出: 2
解释: 第一步将"sea"变为"ea"，第二步将"eat"变为"ea"

提示：

```
给定单词的长度不超过500。
给定单词中的字符只含有小写字母。
```

```
class Solution:
    def minDistance(self, word1: str, word2: str) -> int:
        l1 = len(word1)
        l2 = len(word2)
        dp = [[0] * (l2+1) for i in range(l1+1)]
        for i in range(l1+1):
            dp[i][0] = i
        for i in range(l2+1):
            dp[0][i] = i

        for i in range(1,l1+1):
            for j in range(1,l2+1):
                if word1[i-1] == word2[j-1]:
                    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]+2)
        return dp[-1][-1]
```

Tips

- $dp[i][j]$的含义是word1[i-1],word[j-1]需要删除的最少字符
- 初始化，完全按照含义,$dp[i][0]$需要删除的字符就是i个
- 状态转移：
    - 如果相同，直接继承i-1，j-1的状态
    - 如果不同，可以选择删除i-1，删除j-1或者两个都删除，取最小值

62. 不同路径.md

一个机器人位于一个 m x n 网格的左上角 （起始点在下图中标记为 “Start” ）。

机器人每次只能向下或者向右移动一步。机器人试图达到网格的右下角（在下图中标记为 “Finish” ）。

问总共有多少条不同的路径?

示例 1：

输入：m = 3, n = 7
输出：28

示例 2：

输入：m = 3, n = 2
输出：3
解释：
从左上角开始，总共有 3 条路径可以到达右下角。

1. 向右 -> 向下 -> 向下
2. 向下 -> 向下 -> 向右
3. 向下 -> 向右 -> 向下

示例 3：

输入：m = 7, n = 3
输出：28

示例 4：

输入：m = 3, n = 3
输出：6

提示：

```
1 <= m, n <= 100
题目数据保证答案小于等于 2 * 109
```

1. 二维矩阵

```
class Solution:
    def uniquePaths(self, m: int, n: int) -> int:
        dp = [[1]+[0]*(n-1) for _ in range(m)]
        dp[0] = [1]*n
        for i in range(1,m):
            for j in range(1,n):
                dp[i][j] = dp[i-1][j]+ dp[i][j-1]
        return dp[-1][-1]
```

2. 一维矩阵: 和一维dp可以简化成2个元素一样，二维dp，在这个问题里因为也是逐行更新的所以只用保留一行即可

```
class Solution:
    def uniquePaths(self, m: int, n: int) -> int:
        dp = [1] * n
        for i in range(1,m):
            for j in range(1,n):
                dp[j] += dp[j-1]
        return dp[-1]
```

Tips

1. 初始状态：第一行和第一列都是1
2. 转移状态

$$dp[i][j] = dp[i-1][j] + dp[i][j-1]$$

63. 不同路径 II.md

一个机器人位于一个 m x n 网格的左上角 （起始点在下图中标记为“Start” ）。

机器人每次只能向下或者向右移动一步。机器人试图达到网格的右下角（在下图中标记为“Finish”）。

现在考虑网格中有障碍物。那么从左上角到右下角将会有多少条不同的路径？

网格中的障碍物和空位置分别用 1 和 0 来表示。

示例 1：

输入：obstacleGrid = [[0,0,0],[0,1,0],[0,0,0]]
输出：2
解释：
3x3 网格的正中间有一个障碍物。
从左上角到右下角一共有 2 条不同的路径：

1. 向右 -> 向右 -> 向下 -> 向下
2. 向下 -> 向下 -> 向右 -> 向右

示例 2：

输入：obstacleGrid = [[0,1],[0,0]]
输出：1

提示：

```
m == obstacleGrid.length
n == obstacleGrid[i].length
1 <= m, n <= 100
obstacleGrid[i][j] 为 0 或 1
```

```python
class Solution:
    def uniquePathsWithObstacles(self, obstacleGrid: List[List[int]]) -> int:
        m = len(obstacleGrid)
        n = len(obstacleGrid[0])
        dp = [[0]*n for _ in range(m)]
        for i in range(m):
            if not obstacleGrid[i][0]:
                dp[i][0]=1
            else:
                break
        for i in range(n):
            if not obstacleGrid[0][i]:
                dp[0][i]=1
            else:
                break
        for i in range(1,m):
            for j in range(1,n):
                if not obstacleGrid[i][j]:
                    dp[i][j] = dp[i-1][j] + dp[i][j-1]
        return dp[-1][-1]
```

Tips

1. 转移状态好理解，只有当当前位置没有障碍的时候才更新状态，否则保持原始状态
2. 不过需要注意初始状态的设置，行和列的初始化，当位置i出现障碍的时候，后面的所有位置都是0

64. 最小路径和.md

给定一个包含非负整数的 m x n 网格 grid ，请找出一条从左上角到右下角的路径，使得路径上的数字总和为最小。

说明：每次只能向下或者向右移动一步。

示例 1：

输入：grid = [[1,3,1],[1,5,1],[4,2,1]]
输出：7
解释：因为路径 1→3→1→1→1 的总和最小。

示例 2：

输入：grid = [[1,2,3],[4,5,6]]
输出：12

提示：

```
m == grid.length
n == grid[i].length
1 <= m, n <= 200
0 <= grid[i][j] <= 100
```

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

Tips

和前两道到路径问题差不多，区别只是对数字进行累加

70. 爬楼梯.md

假设你正在爬楼梯。需要 n 阶你才能到达楼顶。

每次你可以爬 1 或 2 个台阶。你有多少种不同的方法可以爬到楼顶呢？

注意：给定 n 是一个正整数。

示例 1：

输入： 2
输出： 2
解释： 有两种方法可以爬到楼顶。

1. 1 阶 + 1 阶
2. 2 阶

示例 2：

输入： 3
输出： 3
解释： 有三种方法可以爬到楼顶。

1. 1 阶 + 1 阶 + 1 阶
2. 1 阶 + 2 阶
3. 2 阶 + 1 阶

解法1.哈哈递归毫无意外超时， 因为递归会重复计算中间的节点

```
class Solution:
    def climbStairs(self, n: int) -> int:
        def helper(n):
            if n<=2:
                return n
            else:
                return helper(n-1) + helper(n-2)
        return helper(n)
```

解法2. 动态规划从bottom到top进行DP

```
class Solution:
    def climbStairs(self, n: int) -> int:
        if n<=2:
            return n
        dp = [1,2]
        for i in range(2, n):
            dp.append(dp[i-1]+dp[i-2])
        return dp[n-1]
```

解法3. 动态规划其实只用保留两个状态即可

```
class Solution:
    def climbStairs(self, n: int) -> int:
        if n<=2:
            return n
        dp = [1,2]
        for i in range(2, n):
            dp[0], dp[1] = dp[1], dp[0]+dp[1]
        return dp[1
```

Tips

1. 比较明显的动态规划问题，于是定义转移矩阵，以及初始条件
2. dp(1)=1 dp(2)=2

3. dp(n) = dp(n-1)+dp(n-2) 其实就是斐波那且数列

714. 买卖股票的最佳时机含手续费.md

给定一个整数数组 prices，其中第 i 个元素代表了第 i 天的股票价格 ；整数 fee 代表了交易股票的手续费用。

你可以无限次地完成交易，但是你每笔交易都需要付手续费。如果你已经购买了一个股票，在卖出它之前你就不能再继续购买股票了。

返回获得利润的最大值。

注意：这里的一笔交易指买入持有并卖出股票的整个过程，每笔交易你只需要为支付一次手续费。

示例 1：

输入：prices = [1, 3, 2, 8, 4, 9], fee = 2
输出：8
解释：能够达到的最大利润:
在此处买入 prices[0] = 1
在此处卖出 prices[3] = 8
在此处买入 prices[4] = 4
在此处卖出 prices[5] = 9
总利润: ((8 - 1) - 2) + ((9 - 4) - 2) = 8

示例 2：

输入：prices = [1,3,7,5,10,3], fee = 3
输出：6

提示：

```
1 <= prices.length <= 5 * 104
1 <= prices[i] < 5 * 104
0 <= fee < 5 * 104
```

1. 贪心算法

和122题的差别在与这里每一次买卖都要付出手续费，于是唯一一种不同的情况就是如果价格是2，8，9。在前一道题之需要买卖两次就可以，这里我们需要只买卖一次。如果想要使用贪心（只对当前状态进行判断），需要一点小技巧。最初的买入价格是price+fee，在2买入，在8卖出以后，新更新的买入价格先更新为不加手续费的当前价格也就是8，

如果下一日价格<8-fee，则从新把买入价格设置为带手续费的

如果下一次价格>8，则继续统计profit，这里其实类比在8没有卖出，而是在9卖出。切判断是否卖出的时机放在了T+1

```python
class Solution:
    def maxProfit(self, prices: List[int], fee: int) -> int:
        profit = 0
        buy = prices[0] + fee
        for p in prices[1:]:
            if p + fee < buy:
                buy = p+fee
            elif p > buy:
                profit += p-buy
                buy = p
        return profit
```

2. 动态规划

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 $dp[i][1] = max(dp[i-1][1], dp[i-1][0] + prices[i] - fee)$

初始化

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

```python
class Solution:
    def maxProfit(self, prices: List[int], fee: 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]-fee)
        return dp1[-1]
```

718. 最长重复子数组.md

给两个整数数组 A 和 B ，返回两个数组中公共的、长度最长的子数组的长度。

示例：

输入：
A: [1,2,3,2,1]
B: [3,2,1,4,7]
输出：3
解释：
长度最长的公共子数组是 [3, 2, 1] 。

提示：

```
1 <= len(A), len(B) <= 1000
0 <= A[i], B[i] < 100
```

1. 二维dp

```python
class Solution:
    def findLength(self, nums1: List[int], nums2: List[int]) -> int:
        l1 = len(nums1)
        l2 = len(nums2)
        dp = [[0] * (l2+1) for i in range(l1+1)]
        maxlen = 0
        for i in range(1,l1+1):
            for j in range(1,l2+1):
                if nums1[i-1]==nums2[j-1]:
                    dp[i][j] = dp[i-1][j-1]+1
                maxlen = max(maxlen, dp[i][j])
        return maxlen
```

2. 滚动数组

```python
class Solution:
    def findLength(self, nums1: List[int], nums2: List[int]) -> int:
        l1 = len(nums1)+1
        l2 = len(nums2)+1
        dp = [0] * l2
        max_len = 0
        for i in range(1, l1):
            for j in range(l2-1, 0, -1):
                if nums1[i-1] ==nums2[j-1]:
```

```
                    dp[j] = dp[j-1]+ 1 # 这里要反向遍历，因为如果正向有两个相邻元素=nums[i]会overwrite
                    max_len = max(max_len , dp[j])
                else:
                    dp[j] = 0

        return max_len
```

Tips

1. 这里可以把二维dp数组压缩到一维，因为二维计算的状态转移是$dp[i][j]=dp[i-1][j-1]$只依赖上一个对角线元素，所以可以压缩到一维$dp[j]=dp[j-1]+1$
2. 滚动数组有两点注意，一个是反向遍历避免overwrite，一个是遇到不等的部分需要置0

72. 编辑距离.md

给你两个单词 word1 和 word2，请你计算出将 word1 转换成 word2 所使用的最少操作数 。

你可以对一个单词进行如下三种操作：

```
插入一个字符
删除一个字符
替换一个字符
```

示例 1：

输入：word1 = "horse", word2 = "ros"
输出：3
解释：
horse -> rorse (将 'h' 替换为 'r')
rorse -> rose (删除 'r')
rose -> ros (删除 'e')

示例 2：

输入：word1 = "intention", word2 = "execution"
输出：5
解释：
intention -> inention (删除 't')
inention -> enention (将 'i' 替换为 'e')
enention -> exention (将 'n' 替换为 'x')
exention -> exection (将 'n' 替换为 'c')
exection -> execution (插入 'u')

提示：

```
0 <= word1.length, word2.length <= 500
word1 和 word2 由小写英文字母组成
```

```python
class Solution:
    def minDistance(self, word1: str, word2: str) -> int:
        l1 = len(word1)
        l2 = len(word2)
        dp = [[0]*(l2+1) for i in range(l1+1)]
        for i in range(1,l1+1):
            dp[i][0] =i
        for i in range(1,l2+1):
            dp[0][i] =i
        for i in range(1,l1+1):
            for j in range(1,l2+1):
                if word1[i-1]==word2[j-1]:
                    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]
```

Tips

- $dp[i][j]$的含义是word1[i-1],word[j-1]需要进行的最少增删换操作
- 初始化，完全按照含义,$dp[i][0]$需要删除的字符就是i个
- 状态转移：
    - 如果相同，直接继承i-1，j-1的状态
    - 如果不同，可以选择删除i-1，删除j-1或者替换继承i-2,j-2

746. 使用最小花费爬楼梯.md

数组的每个下标作为一个阶梯，第 i 个阶梯对应着一个非负数的体力花费值 cost[i]（下标从 0 开始）。

每当你爬上一个阶梯你都要花费对应的体力值，一旦支付了相应的体力值，你就可以选择向上爬一个阶梯或者爬两个阶梯。

请你找出达到楼层顶部的最低花费。在开始时，你可以选择从下标为 0 或 1 的元素作为初始阶梯。

示例 1：

输入：cost = [10, 15, 20]
输出：15
解释：最低花费是从 cost[1] 开始，然后走两步即可到阶梯顶，一共花费 15 。

示例 2：

输入：cost = [1, 100, 1, 1, 1, 100, 1, 1, 100, 1]
输出：6
解释：最低花费方式是从 cost[0] 开始，逐个经过那些 1 ，跳过 cost[3] ，一共花费 6 。

提示：

```
cost 的长度范围是 [2, 1000]。
cost[i] 将会是一个整型数据，范围为 [0, 999] 。
```

```python
class Solution:
    def minCostClimbingStairs(self, cost: List[int]) -> int:
        dp = [cost[0],cost[1]]
        for i in range(2,len(cost)):
            dp[0],dp[1] = dp[1], min(dp[0],dp[1]) + cost[i]
        return min(dp)
```

Tips

动态规划

1. Dp(i) = min(dp[i-1],dp[i-2]) + cost[i] ， 状态转移有一点trick是选择上一个能达到的最小cost，加上当前cost
2. 初始状态，因为可以从0或者1开始所以分别是前两个cost
3. 最终结果因为cost是对应在台阶上的，所以只要可以到达倒数1/2个台阶就能1/2步跨上顶楼，所以cost失去最后两个台阶的min

97. 交错字符串.md

给定三个字符串 s1、s2、s3，请你帮忙验证 s3 是否是由 s1 和 s2 交错 组成的。

两个字符串 s 和 t 交错 的定义与过程如下，其中每个字符串都会被分割成若干 非空 子字符串：

```
s = s1 + s2 + ... + sn
t = t1 + t2 + ... + tm
|n - m| <= 1
交错 是 s1 + t1 + s2 + t2 + s3 + t3 + ... 或者 t1 + s1 + t2 + s2 + t3 + s3 + ...
```

提示：a + b 意味着字符串 a 和 b 连接。

示例 1：

输入：s1 = "aabcc", s2 = "dbbca", s3 = "aadbbcbcac"
输出：true

示例 2：

输入：s1 = "aabcc", s2 = "dbbca", s3 = "aadbbbaccc"
输出：false

示例 3：

输入：s1 = "", s2 = "", s3 = ""
输出：true

提示：

```
0 <= s1.length, s2.length <= 100
0 <= s3.length <= 200
s1、s2、和 s3 都由小写英文字母组成
```

1. 时间复杂度O(mn), 空间复杂度O(mn)
2. $dp[i][j]$的含义是，s1[:i] + s2[:j]可以构成s3[i+j]的部分

常规动态规划解法，类似路径寻找。先出实话dp[0][0]=True，然后分别处理边界i=0和j=0.之后每一层的状态判断为

$$dp[i][j] = dp[i-1][j] and s1[i-1] == s3[i+j-1]$$
$$dp[i][j] = dp[i][j-1] and s2[j-1] == s3[i+j-1]$$

```
class Solution:
    def isInterleave(self, s1: str, s2: str, s3: str) -> bool:
        n1 = len(s1)
        n2 = len(s2)
        n3 = len(s3)
        if n1+n2!=n3:
            return False
        dp = [[False]*(n2+1) for i in range(n1+1)]
        dp[0][0]= True
        for i in range(1,n1+1):
            dp[i][0] = dp[i-1][0] and s1[i-1]==s3[i-1]
        for j in range(1,n2+1):
            dp[0][j] = dp[0][j-1] and s2[j-1]==s3[j-1]

        for i in range(1,n1+1):
            for j in range(1,n2+1):
                p = i+j-1
                dp[i][j] = (dp[i-1][j] and s1[i-1]==s3[p]) or  (dp[i][j-1] and s2[j-
1]==s3[p])
```

```
        return dp[n1][n2]
```