动态规划

动态规划（英语：Dynamic programming，简称 DP）是一种在数学、管理科学、计算机科学、经济学和生物信息学中使用的，通过把原问题分解为相对简单的子问题的方式求解复杂问题的方法。

动态规划常常适用于有重叠子问题和最优子结构性质的问题，动态规划方法所耗时间往往远少于朴素解法。

动态规划背后的基本思想非常简单。大致上，若要解一个给定问题，我们需要解其不同部分（即子问题），再根据子问题的解以得出原问题的解。动态规划往往用于优化递归问题，例如斐波那契数列，如果运用递归的方式来求解会重复计算很多相同的子问题，利用动态规划的思想可以减少计算量。

通常许多子问题非常相似，为此动态规划法试图仅仅解决每个子问题一次，具有天然剪枝的功能，从而减少计算量：一旦某个给定子问题的解已经算出，则将其记忆化存储，以便下次需要同一个子问题解之时直接查表。这种做法在重复子问题的数目关于输入的规模呈指数增长时特别有用。

Leetcode 第 674 题：最长连续递增序列

给定一个未经排序的整数数组，找到最长且连续的的递增序列，并返回该序列的长度。

示例 1:

输入: [1,3,5,4,7]

输出: 3

解释: 最长连续递增序列是 [1,3,5], 长度为 3。

尽管 [1,3,5,7] 也是升序的子序列, 但它不是连续的，因为 5 和 7 在原数组里被 4 隔开。

示例 2:

输入: [2,2,2,2,2]

输出: 1

解释: 最长连续递增序列是 [2], 长度为 1。

注意：数组长度不会超过 10000。

方法一：动态规划（时间复杂度：O(N)，空间复杂度：O(N)）

解题思路：以数组每个元素为最后的元素，求其最长连续递增序列。递推关系式如下：

$$dp[i]=dp[i-1]+1, \text{ if } nums[i]>nums[i-1]$$

$$dp[i]=1, \text{ otherwise}$$

代码如下：

```
class Solution:
    def findLengthOfLCIS(self, nums: List[int]) -> int:
        #dp
        n=len(nums)
        if n<=1:
            return n
```

```
        dp=[1]*n
        for i in range(1,n):
            if nums[i]>nums[i-1]:
                dp[i]=dp[i-1]+1
        return max(dp)
```

Leetcode 第 300 题：最长连续递增序列
给定一个无序的整数数组，找到其中最长上升子序列的长度。

示例:
输入: [10,9,2,5,3,7,101,18]
输出: 4
解释: 最长的上升子序列是 [2,3,7,101]，它的长度是 4。

解题思路：
跟上题思路雷同，以数组每个元素当成最后的元素找其最长上升子序列。递推关系如下：
dp[i]=max(dp[j]+1) where 0<=j<i and nums[j]<nums[i]
代码如下：

```
class Solution:
    def lengthOfLIS(self, nums: List[int]) -> int:
        n=len(nums)
        if n<=1:
            return n
        dp=[1]*n
        for i in range(1,n):
            max_v=1
            for j in range(i):
                if nums[i]>nums[j]:
                    max_v=max(dp[j]+1,max_v)
            dp[i]=max_v
```

```
        return max(dp)
```

Leetcode 第 516 题：最长回文字序列

给定一个字符串 s ，找到其中最长的回文子序列，并返回该序列的长度。可以假设 s 的最大长度为 1000 。

示例 1:
输入:"bbbab"
输出:4
一个可能的最长回文子序列为 "bbbb"。

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

解题思路：
要求一个字符串 s 的最长回文序列长度 L(s)，则可以看 s[0]与 s[n-1]的关系，若 s[0]==s[n-1], 则 L(s)=L(s[1:n-1])+2,否则 L(s)=max(L(s[:n-1]),L(s[1:n-2]))
代码如下：

```
lass Solution:
    def longestPalindromeSubseq(self, s: str) -> int:
        #dp
        n=len(s)
        if n<=1:
            return n
        dp=[[0]*n for _ in range(n)]
        max_l=1
        for i in range(n):
            dp[i][i]=1
```

```
        for i in range(n-1):
            if s[i]==s[i+1]:
                dp[i][i+1]=2
                max_l=2
            else:
                dp[i][i+1]=1
        for l in range(3,n+1):
            for i in range(n-l+1):
                j=i+l-1
                if s[i]==s[j]:
                    dp[i][j]=dp[i+1][j-1]+2
                    max_l=max(max_l,dp[i][j])
                else:
                    dp[i][j]=max(dp[i+1][j],dp[i][j-1])
                    max_l=max(max_l,dp[i][j])
        return max_l
```

Leetcode 第 198 题：打家劫舍

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。

解题思路：解决改问题可假设有 dp[i]为偷到第 i 家的最高金额，由于不能偷相邻两家，所以 dp[i]=max(dp[i-1],dp[i-2]+nums[i])

代码如下：

```
class Solution:
    def rob(self, nums: List[int]) -> int:
        n=len(nums)
```

```
        if n==0:return 0
        if n==1:return nums[0]
        dp=[0]*(n+1)
        dp[1]=nums[0]
        for i in range(2,n+1):
            dp[i]=max(dp[i-2]+nums[i-1],dp[i-1])
        return dp[-1]
```

Leetcode 第 213 题：打家劫舍

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，能够偷窃到的最高金额。

示例 1:
输入: [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: [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 。

解题思路：

由于这排房子排成一个圆圈，第一间和最后一间是连在一起的，所以两间不能同时偷，因此可以分为两种情况，一种考虑偷第一家，第二种考虑不偷第一家，两种情况的最大值即为结果。

代码如下：

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

Leetcode 第 72 题：编辑距离

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：

1）插入一个字符 2）删除一个字符 3）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')

解题思路：
假设 dp(i,j)为 word1[:i]和 word2[:j]的最短编辑距离，那 dp[i][j]分为如下几种情况：
若 word1[i]==word2[j],则 dp(i,j)=dp(i-1,j-1)
若 word1[i]!=word2[j],则需分以下几种情况：
第一：word1[:i-1]添加一个字母 则 dp(i,j)有可能为 dp(i-1,j)+1
第二：Word2[:j-1]添加一个字母 则 dp(i,j)有可能为 dp(i,j-1)+1
第三：将 word1[i]替换成 word2[j],或者反过来 则 dp(i,j)有可能为 dp(i-1,j-1)+1
第四：word1 或者 word2 删除一个元素 则 dp(i,j)有可能为 dp(i-1,j)+1 或 dp(i,j-1)+1
总结起来，就是 dp(i,j)=min(dp(i-1,j)+1,dp(i,j-1)+1,dp(i-1,j-1)+1)

代码如下：

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

Leetcode 第 5 题：最长回文子串
给定一个字符串 s，找到 s 中最长的回文子串。你可以假设 s 的最大长度为 1000。

示例 1：
输入: "babad"
输出: "bab"
注意: "aba" 也是一个有效答案。

示例 2：
输入: "cbbd"
输出: "bb"

解题思路：假定 f(i,j)(只有 0，1 两个取值,0 表示不是回文串，1 表示是回文串)为 s[i:j+1]是否一个回文串，如果 s[i]==s[j]且 f(i+1,j-1)==1，则 f(i,j)==1，如此一来，从长度遍历，每遍历一次更新一次最大长度。

代码如下：

```
class Solution:
    def longestPalindrome(self, s: str) -> str:
        n=len(s)
        if n==0:return ""
        if n==1:return s[0]
        dp=[[0]*n for _ in range(n)]
        max_l=s[0]
        for i in range(n):
            dp[i][i]=1
        for i in range(n-1):
            if s[i]==s[i+1]:
                dp[i][i+1]=1
                max_l=s[i:i+2]
```

```
        for l in range(3,n+1):
            for i in range(n-l+1):
                j=i+l-1
                if s[i]==s[j] and dp[i+1][j-1]==1:
                    dp[i][j]=1
                    max_l=s[i:j+1]
        return max_l
```

题目来源：LeetCode