1. 经典算法题记录（动态规划）

1.1. 矩阵相关

1.1.1. 矩形面积

1.1.1.1. 柱状图中最大矩形（84，困难）

> **题目：** 给定 n 个非负整数，用来表示柱状图中各个柱子的高度。每个柱子彼此相邻，且宽度为 1 。求在该柱状图中，能够勾勒出来的矩形的最大面积。

> **暴力法**:

> **思路1**：索引i和j围成的矩形面积为s=(j-i+1)*min(heights[i:j+1])

```
class Solution:
    def largestRectangleArea(self, heights: List[int]) -> int:
        #暴力法
        if not heights:return 0
        res = 0
        for i in range(0,len(heights)):
            Min = heights[i]
```

```
            for j in range(i,len(heights)):
                Min = min(Min,heights[j])
                res = max(res,(j-i+1)*Min)
                #print(i,j,Min)
        return res
#改进：第二次循环可以将j移到最后一个大于Min的位置，避免多余计算。
```

> **思路2**：以索引i为中心向左右扩展，找到左右最后一个大于heights[i]的索引left和right，此时矩形面积为s=heights[i]*(right-left+1).矩形意义是以heights[i]为高能围成的最大矩形。

> **进阶**：

> > **单调栈**：单调栈是一种特殊的栈，栈内元素保持递增或者递减。 单调栈有两个性质： 1.满足从栈顶到栈底的元素具有严格的单调性 2.满足栈的后进先出特性越靠近栈底的元素越早进栈 利用单调栈，可以找到从左/右遍历第一个比它小/大的元素的位置，在某些问题中可以将时间复杂度从O(N^2)降低为O(N)
> > 维护逻辑：元素大于栈顶进栈，小于则栈顶出栈直到栈顶小于该元素

```
#stack存元素的索引
class Solution(object):
    def largestRectangleArea(self, heights):
        heights.append(-1)
        stack = []
        maxArea = 0
        for i in range(len(heights)):
            while stack and heights[stack[-1]] > heights[i]:
                height = heights[stack.pop()]
                width = i if not stack else i-stack[-1]-1
                maxArea = max(maxArea,height*width)
            stack.append(i)
            #print(stack)
        return maxArea
```

1.1.1.2. 矩阵中最大长方形（85，困难）

> **题目**：给定一个仅包含 0 和 1 的二维二进制矩阵，找出只包含 1 的最大矩形，并返回其面积。

> 思路：对每一行求出向上的连续为1的柱状图，使用题目84的解法即可。

```
class Solution:

    def largestRectangleArea(self, heights):
        heights.append(-1)
        stack = []
        maxArea = 0
        for i in range(len(heights)):
            while stack and heights[stack[-1]] > heights[i]:
                height = heights[stack.pop()]
```

```
                width = i if not stack else i-stack[-1]-1
                maxArea = max(maxArea,height*width)
            stack.append(i)
            #print(stack)
        return maxArea

    #柱状图采取递归的方式求
    def maximalRectangle(self, matrix: List[List[str]]) -> int:
        if not matrix:return 0
        last = [int(i) for i in matrix[0]]
        res = self.largestRectangleArea(last)
        for i in range(1,len(matrix)):
            heights = [int(i)+j if int(i) > 0 else 0 for i,j in
zip(matrix[i],last)]
            print(heights)
            res = max(res,self.largestRectangleArea(heights))
            last = heights
        return res
```

1.1.1.3. 矩阵中最大正方形（221，中等）

> **题目**：在一个由 0 和 1 组成的二维矩阵内，找到只包含 1 的最大正方形，并返回其面积。

> 思路1：沿用最大长方形的思路，面积计算修改。

```
class Solution:

    def largestRectangleArea(self, heights):
        heights.append(-1)
        stack = []
        maxArea = 0
        for i in range(len(heights)):
            while stack and heights[stack[-1]] > heights[i]:
                height = heights[stack.pop()]
                width = i if not stack else i-stack[-1]-1
                #修改面积计算公式
                maxArea = max(maxArea,min(height,width)**2)
            stack.append(i)
            #print(stack)
        return maxArea

    #柱状图采取递归的方式求
    def maximalSquare(self, matrix: List[List[str]]) -> int:
        if not matrix:return 0
        last = [int(i) for i in matrix[0]]
        res = self.largestRectangleArea(last)
        for i in range(1,len(matrix)):
            heights = [int(i)+j if int(i) > 0 else 0 for i,j in
zip(matrix[i],last)]
            print(heights)
            res = max(res,self.largestRectangleArea(heights))
```

```
            last = heights
        return res
```

> 思路2：直接动态规划
>
> 定义dp矩阵，dp[i][j]表示以第i行第j列为右下角的正方形的最大边长。则有递推关系如下
> `if matrix[i][j] == 0:dp[i][j] = 0`
> `if matrix[i][j] != 0:dp[i][j]=1+min(dp[i-1][j],dp[i][j-1],dp[i-1][j-1])`
> 注：补充全零的行列，便于计算。

```
#java version
class Solution {

    public int maximalSquare(char[][] matrix) {
        int m = matrix.length;
        if(m < 1) return 0;
        int n = matrix[0].length;
        int max = 0;
        int[][] dp = new int[m+1][n+1];

        for(int i = 1; i <= m; ++i) {
            for(int j = 1; j <= n; ++j) {
                if(matrix[i-1][j-1] == '1') {
                    dp[i][j] = 1 + Math.min(dp[i-1][j-1], Math.min(dp[i-1]
[j], dp[i][j-1]));
                    max = Math.max(max, dp[i][j]);
                }
            }
        }

        return max*max;
    }
}
```

1.1.2. 最短路径（带权重）

1.1.2.1. 地下城游戏

> **题目**：一些恶魔抓住了公主（P）并将她关在了地下城的右下角。地下城是由 M x N 个房间组成的二维网格。我们英勇的骑士（K）最初被安置在左上角的房间里，他必须穿过地下城并通过对抗恶魔来拯救公主。骑士的初始健康点数为一个正整数。如果他的健康点数在某一时刻降至 0 或以下，他会立即死亡。有些房间由恶魔守卫，因此骑士在进入这些房间时会失去健康点数（若房间里的值为负整数，则表示骑士将损失健康点数）；其他房间要么是空的（房间里的值为 0），要么包含增加骑士健康点数的魔法球（若房间里的值为正整数，则表示骑士将增加健康点数）。为了尽快到达公主，骑士决定每次只向右或向下移动一步。

1.2. 字符串相关

1.2.1. 最小公共子串

1.2.2. 编辑距离（**189，困难**）

> **题目**：给定两个单词 word1 和 word2，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

> **思路**：动态规划。建立dp矩阵，`dp[i][j]`表示`string1[:i]`和`string2[:j]`的编辑距离。则有如下递推关系：
> `dp[i][j] = min(dp[i-1][j]+1,dp[i][j-1]+1,dp[i-1][j-1]+(string1[i-1] != string2[j-1]))`
> **矩阵初始化**：`dp[0][j]=j`,`dp[i][0]=i`
> **解释**：在计算`string1[:i]`和`string2[:j]`的编辑距离时，对其末尾有如下四种处理方式，对应递推式右边的四种情况。
>
> > 1.删掉`string1[i-1]`
> > 2.删掉`string2[j-1]`
> > 3.同时删掉`string1[i-1]`和`string2[j-1]`
> > 4.同时保留`string1[i-1]`和`string2[j-1]`,此时有`string1[i-1] == string2[j-1]`

```
class Solution:
    def minDistance(self, word1: str, word2: str) -> int:

        if not word1:
            return len(word2)
        if not word2:
            return len(word1)

        dp = [[0]*(len(word2)+1) for _ in range(len(word1)+1)]

        for i in range(len(word1)):
            dp[i+1][0] = i+1
        for j in range(len(word2)):
            dp[0][j+1] = j+1
        #print(dp)
        for i in range(1,len(word1)+1):
            for j in range(1,len(word2)+1):
                #print(i,j)
                dp[i][j] = min(dp[i-1][j-1]+(word1[i-1]!=word2[j-1]),dp[i-1][j]+1,dp[i][j-1]+1)
        return dp[-1][-1]
```

1.2.3. 两个字符串的删除操作（**583，中等**）

> **题目**：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。

> **思路**：和编辑距离的思路一致。唯一不同的是，在`string1[i-1] != string2[j-1]`时，删除操作的步数加2。

```
#略
```

1.2.4. 回文系列

1. 最长回文子串（**5，中等**）

> **题目**：给定一个字符串 s，找到 s 中最长的回文子串

> **思路**：s前加上标记字符#，建立dp数组，`dp[i]`指以`s[dp[i-1]]`为结尾的最长回文子串。则有如下递推关系：
> `if s[i] == s[i-1-dp[i-1]]:dp[i]=2+dp[i-1]`
> `if s[i] != s[i-1-dp[i-1]]:dp[i]=max(i-j+1，j从i-1-dp[i-1]取到i使得s[j:i+1]是回文串)`
> 注：第二种情况j的取值不能超出以s[i-1]为结尾的最长回文子串的范围，否则最长的定义相悖。

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

        #判断回文串
        def is_palindromic(string):
            start = 0
            end = len(string)-1
            while start < end:
                if string[start] != string[end]:
                    return False
                start += 1
                end -= 1
            return True

        s = '#' + s
        dp = [0]*(len(s))

        for i in range(1,len(s)):
            if s[i] == s[i-1-dp[i-1]]:
                dp[i] = 2 + dp[i-1]
            else:
                for j in range(i-1-dp[i-1],i+1):
                    if is_palindromic(s[j:i+1]):
                        dp[i] = i-j+1
                        break
        Max = max(dp)
        for i in range(len(dp)):
            if dp[i] == Max:
                return s[i-Max+1:i+1]
```

2. 最长回文子序列（**516，中等**）

> **题目**：给定一个字符串s，找到其中最长的回文子序列。

> **思路**：建立dp矩阵，`dp[i][j]`表示`s[i:j+1]`的最长回文序列长度，则递推关系如下：
> `dp[i][j] = max(dp[i+1][j],dp[i][j-1],dp[i+1][j-1]+2*(s[i] == s[j]))`
> 注：dp应该依对角线更新。

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

        if not s:return 0
        if s == s[::-1]:return len(s)

        df = [[0]*len(s) for _ in range(len(s))]

        for i in range(len(s)):
            df[i][i] = 1

        for margin in range(1,len(s)):
            for i in range(len(s)-margin):
                j = i + margin
                #print(i,j)
                df[i][j] = max(df[i+1][j],df[i][j-1],df[i+1][j-1]+2*(s[i]
== s[j]))

        return df[0][len(s)-1]
```

3. 回文子串数目（**647，中等**）

> **题目**：给定一个字符串，你的任务是计算这个字符串中有多少个回文子串。具有不同开始位置或结束位置的子串，即使是由相同的字符组成，也会被计为是不同的子串。

> **思路**：建立dp数组，`dp[i]`表示`s[:i+1]`的回文子串数目。则有：
> `dp[i] = dp[i-1] + #(以s[i]结尾的回文串数目)`。

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

        def is_palindrome(s):
            start = 0
            end = len(s) - 1
            while start < end:
                if s[start] != s[end]:
                    return False
                start += 1
                end -= 1
            return True

        df = [0]*len(s)
        df[0] = 1
        #记录从头开始一直相同的序列长度
        Len = 0
        for i in range(len(s)):
```

```
            if s[i] != s[0]:
                break
            Len += 1
        #print(Len)

        for i in range(1,len(s)):
            #特殊情况
            if i <= Len-1:
                df[i] = (i+1)*(i+2)//2
                continue
            df[i] = df[i-1]
            for j in range(i+1):
                if is_palindrome(s[j:i+1]):
                    df[i] += 1
        return dp[-1]
```

1.3. 二叉树相关

1.4. 图相关

1.5. 背包问题

1.6. 斐波拉其数列原型

1.7. 其他动态规划

1.7.1. 买卖股票问题

1.7.1.1. 一次买卖股票最大利润（121，简单）

> **题目**：给定一个数组，它的第 i 个元素是一支给定股票第 i 天的价格。如果你最多只允许完成一笔交易（即买入和卖出一支股票），设计一个算法来计算你所能获取的最大利润。注意你不能在买入股票前卖出股票。

> **思路**：维护历史最小价格，当前价格大于最小价格时，更新结果。

```
#一次买进卖出的最大利润
def maxProfitByOne(prices):
    profit = 0
    #维护历史最小价格
    min_stack = []
    for i in prices:
        if not min_stack or min_stack[-1] >= i:
            min_stack.append(i)
        else:
            profit = max(profit,i-min_stack[-1])
    return profit
```

1.7.1.2. 尽可能买卖股票最大利润（122，简单）

> **题目**：给定一个数组，它的第 i 个元素是一支给定股票第 i 天的价格。设计一个算法来计算你所能获取的最大利润。你可以尽可能地完成更多的交易（多次买卖一支股票）。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）。

> **思路**：对所有连续递增子数组首尾进行计算即可。简化计算方式：`profit += max(0,prices[j]-prices[j-1])`。

```
#略
```

1.7.1.3. 两次买进卖出的最大利润（123，困难）

> **题目**：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）。

> **思路1**：借助买卖一次股票的最大利润，暴力将prices数组拆成两部分。

```
class Solution:
    def maxProfit(self, prices: List[int]) -> int:

        #一次买进卖出的最大利润
        def maxProfitByOne(prices):
            profit = 0
            #维护历史最小价格
            min_stack = []
            for i in prices:
                if not min_stack or min_stack[-1] >= i:
                    min_stack.append(i)
                else:
                    profit = max(profit,i-min_stack[-1])
            return profit

        res = 0
        for i in range(len(prices)):
            res =
max(res,maxProfitByOne(prices[:i])+maxProfitByOne(prices[i:]))

        return res
```

> **思路2**：维护四个状态`min_stack,max_stack,dp1,dp2`。`min_stack[i]`表示前i个元素的最小值，`max_stack[i]`表示后i个元素的最大值，`dp1[i]`表示对`prices[:i]`一次买入卖出的最大利润，`dp2[i]`表示对`prices[-i-1:]`一次买入卖出的最大利润。更新模式如下：
> `dp1[i] = max(dp1[i-1],prices[i-1]-min_stack[i-1])`
> `dp2[i] = max(dp2[i-1],max_stack[i-1]-prices[::-1][i-1])`

```
class Solution:
    def maxProfit(self, prices: List[int]) -> int:
```

```
        if len(prices) <= 1:
            return 0
        #可以同时更新
        #min_stack和max_stack计算
        min_stack = []
        max_stack = []
        for i in prices:
            if not min_stack or i < min_stack[-1]:
                min_stack.append(i)
            else:
                min_stack.append(min_stack[-1])
        for i in prices[::-1]:
            if not max_stack or i > max_stack[-1]:
                max_stack.append(i)
            else:
                max_stack.append(max_stack[-1])

        #dp1和dp2计算
        dp1 = [0]*(len(prices)+1)
        dp2 = [0]*(len(prices)+1)
        for i in range(2,len(prices)+1):
            dp1[i] = max(dp1[i-1],prices[i-1]-min_stack[i-1])

        for j in range(2,len(prices)+1):
            dp2[j] = max(dp2[j-1],max_stack[j-1]-prices[-j])

        return max([i+j for i,j in zip(dp1,dp2[::-1])])
```

> **思路3**：巧妙定义dp变量。对于任意一天考虑四个变量: `fstBuy`: 在该天第一次买入股票可获得的最大收益 `fstSell`: 在该天第一次卖出股票可获得的最大收益 `secBuy`: 在该天第二次买入股票可获得的最大收益 `secSell`: 在该天第二次卖出股票可获得的最大收益 分别对四个变量进行相应的更新, 最后`secSell`就是最大 收益值`(secSell >= fstSell)`

```
#java version
class Solution {
    public int maxProfit(int[] prices) {

        int fstBuy = Integer.MIN_VALUE, fstSell = 0;
        int secBuy = Integer.MIN_VALUE, secSell = 0;
        for(int p : prices) {
            fstBuy = Math.max(fstBuy, -p);
            fstSell = Math.max(fstSell, fstBuy + p);
            secBuy = Math.max(secBuy, fstSell - p);
            secSell = Math.max(secSell, secBuy + p);
        }
        return secSell;
    }
}
```

1.7.1.4. 最多k次买入卖出的最大利润（188，困难）

> **题目**：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 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）。

> **思路1**：暴力递归

```
class Solution:
    def maxProfit(self, k: int, prices: List[int]) -> int:

        if len(prices) <= 1:return 0

        self.res = 0

        def helper(profit,prices,k):
            #print(profit,prices,k)
            #简单情况
            if len(prices) <= 1 or k == 0:
                self.res = max(self.res,profit)
                return
            #找到第一个非递减的
            for j in range(1,len(prices)):
                if prices[j] > prices[j-1]:
                    break
            Min = prices[j-1]
            #可能不取
            helper(profit,prices[j:],k)
            for i in range(j,len(prices)):
                if prices[i] > Min:
                    helper(profit+prices[i]-Min,prices[i+1:],k-1)

        helper(0,prices,k)
        return self.res
```

> **思路2**：巧妙的dp。dp数组的定义如下。当k大于等于数组长度一半时, 问题退化为贪心问题此时采用尽可能买卖股票最大利润的贪心方法解决可以大幅提升时间性能, 对于其他的k,可以采用两次买进卖出的最大利润的方法来解决, 两次买进卖出的最大利润[思路3]中定义了两次买入和卖出时最大收益的变量, 在这里就是k组这样的变量, `t[i][0]`和`t[i][1]`分别表示第i笔交易买入和卖出时各自的最大收益。

```
#java version
class Solution {
    public int maxProfit(int k, int[] prices) {

        #简单情况
        if(k < 1) return 0;
        #贪心情况
        if(k >= prices.length/2) return greedy(prices);
        #一般情况
        int[][] t = new int[k][2];
```

```
        for(int i = 0; i < k; ++i)
            t[i][0] = Integer.MIN_VALUE;
        for(int p : prices) {
            t[0][0] = Math.max(t[0][0], -p);
            t[0][1] = Math.max(t[0][1], t[0][0] + p);
            for(int i = 1; i < k; ++i) {
                t[i][0] = Math.max(t[i][0], t[i-1][1] - p);
                t[i][1] = Math.max(t[i][1], t[i][0] + p);
            }
        }
        return t[k-1][1];
    }
    #贪心情况
    private int greedy(int[] prices) {
        int max = 0;
        for(int i = 1; i < prices.length; ++i) {
            if(prices[i] > prices[i-1])
                max += prices[i] - prices[i-1];
        }
        return max;
    }
}
```