动态规划常见的面试问题总结

目录

一、硬币问题

题型 1：假设有 1 元， 3 元， 5 元的硬币若干（无限） ， 现在需要凑出 11 元，问如何组合才能使硬币的数量最少？

（1）思考过程（参考硬币问题）

假设一个函数 dp(i) 表示需要凑出 i 的总价值需要的最少硬币数量，那么我们不难想到：

当 i = 0 时， dp(0) = 0。因为不要凑钱了嘛，当然也不需要任何硬币了。这一步很关键！
当 i = 1 时， dp(1) = 1。
当 i = 2 时，因为我们并没有 2 元的硬币，所以只能拿 1 元的硬币来凑, dp(2) = 2。
当 i = 3 时，我们可以在第 3 步的基础上加上 1 个 1 元硬币，得到 3 这个结果。但其实我们有 3 元硬币，所以这一步的最优结果不是建立在第 3 步的结果上得来的，而是应该建立在第 1 步上，加上 1 个 3 元硬币，得 dp(3) = 1。
依此类推……

可以看出，除了第 1 步，其他往后的结果都是建立在它之前得到的某一步的最优解上，加上 1 个硬币得到，因此可以得出：

d(i) = d(j) + 1， j < i。通俗地讲，如果我们需要凑出 i 元，就在凑出 j 的结果上再加上某一个硬币就行了。

那这里我们加上的是哪个硬币呢。嗯，其实很简单，把每个硬币试一下就行了：

假设最后加上的是 1 元硬币，那 dp(i) = dp(j) + 1 = dp(i - 1) + 1。

假设最后加上的是 3 元硬币，那 dp(i) = dp(j) + 1 = dp(i - 3) + 1。
假设最后加上的是 5 元硬币，那 dp(i) = dp(j) + 1 = dp(i - 5) + 1。

因此，分别计算出 dp(i - 1) + 1，dp(i - 3) + 1，dp(i - 5) + 1 的值，取其中的最小值，即为最优解 d(i)，状态转移方程：

dp[i] = min{dp[i - coins[j]] + 1}，其中 i >= coins[j], 0 <= j < coins.length

换一种表达方式：给定总金额为 A 的一张纸币，现要兑换成面额分别为 a1,a2,....,an 的硬币，且希望所得到的硬币个数最少。

（2）Python 代码

```python
# 动态规划思想  dp 方程式如下
# dp[0] = 0
# dp[i] = min{dp[i - coins[j]] + 1}, 且 其中 i >= coins[j], 0 <= j < coins.length
# 回溯法，输出可找的硬币方案
# path[i] 表示经过本次兑换后所剩下的面值，即 i - path[i] 可得到本次兑换的硬币值。


def changeCoins(coins, n):
    if n < 0: return None
    dp, path = [0] * (n + 1), [0] * (n + 1)  # 初始化
    for i in range(1, n + 1):
        minNum = i  # 初始化当前硬币最优值
        for c in coins:  # 扫描一遍硬币列表，选择一个最优值
            if i >= c and minNum > dp[i - c] + 1:
                minNum, path[i] = dp[i - c] + 1, i - c
        dp[i] = minNum  # 更新当前硬币最优值

    print('最少硬币数:', dp[-1])
    print('可找的硬币', end=': ')
    while path[n] != 0:
        print(n - path[n], end=' ')
        n = path[n]
    print(n, end=' ')


if __name__ == '__main__':
    coins, n = [1, 3, 5], 11  # 输入可换的硬币种类，总金额 n
    changeCoins(coins, n)
```

题型 2 ：有数量不限的硬币， 币值为 25 分、 10 分、 5 分和 1 分， 请编写代码计算 n 分有几种表示法。

（1）求解思路

当只有 1 分的硬币时， n 从 1 到 n 分别有多少种表示方法；
当有 1 分和 5 分的硬币时， n 从 1 到 n 分别有多少种表示方法；
依次类推， 直到我们将 1 分、5 分、 10 分和 25 分的硬币全部使用完， 思想类似于 0-1 背包问题。

用数组 coins[i] = {1,5,10,25}表示各种币值， 假设 ways[i][j]代表能用前 i 种硬币来表示 j 分的方法数目。此时可以得到一张二维表 ways[i][j]，其中横坐标表示前 i 种表示币值， j 表示硬币的总值。当增加一种新的硬币币值时， 有两种情况：

若不加入此种币值： ways[i][j]=ways[i-1][j]；
若加入此种币值： 加入该枚硬币之前的方法数为 ways[i][j-coins[i]]，那么加入该枚硬币之后构成 j 分的方法数也为 ways[i][j-coins[i]]。因此当增加一种新的币值时， j 分的表示方法数为 ways[i][j]=ways[i-1][j]+ways[i][j-coins[i]]。

（2）Python 代码

二维表的形式：

```python
def changeCoins2(coins, n):
    len1 = len(coins)
    if len1 == 0 and n < 0:
        return None
    ways = [[0] * (n+1) for row in range(len1)]
    for i in range(len1):
        ways[i][0] = 1  # 第 1 行初始化为 1
    for j in range(1, n+1):
        ways[0][j] = 1  # 第 1 列初始化为 1
    for i in range(1, len1):
        for j in range(1, n+1):
            if j>=coins[i]:
                ways[i][j] = ways[i - 1][j] + ways[i][j - coins[i]]
            else:
                ways[i][j] = ways[i - 1][j]

    print('\n 假设有数量不限的硬币, 币值为{0}, 则{1}分共有{2}种表示法
 '.format(coins, n, ways[len1 - 1][n]))


```

```python
if __name__ == '__main__':
    coins, n = [1, 5, 10, 25], 10  # 输入可换的硬币种类，总金额 n
    changeCoins2(coins, n)
```

当然，二维表未免过于复杂，我们可以用一张一维表，即用一维数组 ways[j]来记录 j 分的表示方法数。改进的代码实现如下：

```python
def changeCoins2(coins, n):
    len1 = len(coins)
    if len1 == 0 and n < 0:
        return None
    ways = [0] * (n+1)  # 初始化
    ways[0] = 1

    for i in range(len1):
        for j in range(coins[i], n+1):
            # 保证 n 小于等于 100000，为了防止溢出，请将答案 Mod 1000000007
            ways[j] = (ways[j] + ways[j - coins[i]])%1000000007

    print('\n 假设有数量不限的硬币, 币值为{0}, 则{1}分共有{2}种表示法'.format(coins, n, ways[n]))

if __name__ == '__main__':
    coins, n = [1, 5, 10, 25], 10  # 输入可换的硬币种类，总金额 n
    changeCoins2(coins, n)
```

二、爬楼梯问题（青蛙跳台问题）

题型 1：一只青蛙一次可以跳上 1 级台阶,也可以跳上 2 级。求该青蛙跳上一个 n 级的台阶总共有多少种跳法（先后次序不同算不同的结果）。

（1）分析过程，参考 https://www.aliyun.com/jiaocheng/520413.html

假设 f(n)表示一只青蛙跳上一个 n 级台阶总共的跳法总数，则不难可得：

当 n = 0 时，f(0) = 0;
当 n = 1 时， f(1) = 1;
当 n = 2 时， f(2) = 1+1 = 2，表示一种跳法是跳两次 1 级台阶，另一种跳法是跳一次 2 级台阶；

依次递推，得到递推公式为：$f(n) = f(n - 1) + f(n - 2)$, $n>=3$。

因此，这个题的本质就是斐波那契数列!!！但又不完全是!!！我们知道，这个数列可以用递归函数来表示，也可以用迭代来进行计算，前者属于自顶向下的模式（简洁明了），后者属于自底向上的模式（简单高效），面试过程中建议两者皆会！实际工程中常用迭代的方法！

（2）Python 代码

A、递归求解：

```
def jumpFloor(number):
    # write code here
    if number <= 0: return 0
    if number == 1: return 1
    if number == 2: return 2
    if number >= 3:
        return jumpFloor(number - 1) + jumpFloor(number - 2)

print(jumpFloor(2))
```

B、迭代求解：

```
# -*- coding:utf-8 -*-
class Solution:
    def jumpFloor(self, number):
        # write code here
        if number<=0: return 0
        if number==1: return 1
        if number==2: return 2
        jumpFloor1,jumpFloor2 = 1,2
        if number>=3:
            for i in range(3,number+1):
                res = jumpFloor1+jumpFloor2
                jumpFloor1,jumpFloor2 = jumpFloor2,res
        return res
```

当然，如果整理后，还可以写出更简洁的代码，参考 Python 用时最短

```
# -*- coding:utf-8 -*-
class Solution:
    def jumpFloor(self, number):
```

```
        # write code here
        a = 1
        b = 1
        for i in range(number):
            a,b = b,a+b
        return a
```

小结：如果我们变化一下，一只青蛙一次可以跳上 1 级台阶，也可以跳上 2 级，也可以跳上 3 级。求该青蛙跳上一个 n 级的台阶总共有多少种跳法（先后次序不同算不同的结果）。

推导方式同上：

当 n = 0 时，f(0) = 0；
当 n = 1 时， f(1) = 1；
当 n = 2 时， f(2) = 1+1 = 2，表示一种跳法是跳两次 1 级台阶，另一种跳法是跳一次 2 级台阶；
当 n = 3 时， f(3) = 1+1+1+1 = 4，表示一种是跳三次 1 级台阶，一种是先跳 1 级再跳 2 级台阶，一种是先跳 2 级再跳 1 级台阶，还有一种是直接跳 3 级台阶；
依次递推，得到递推公式为：f(n) = f(n - 1) + f(n - 2) + f(n - 3)，n >= 4。
编程的话类似处理，两种方法，迭代为佳!!!

题型 2：一只青蛙一次可以跳上 1 级台阶,也可以跳上 2 级……它也可以跳上 n 级。求该青蛙跳上一个 n 级的台阶总共有多少种跳法。

（1）分析过程

假设 f(n)表示一只青蛙跳上一个 n 级台阶总共的跳法总数，则不难可得：

当 n = 0 时，f(0) = 0；
当 n = 1 时，f(1) = f(0) + 1 = 1；
当 n = 2 时，f(2) = f(0) + f(1) + 1 = 2；
当 n = 3 时，f(3) = f(0) + f(1) + f(2) + 1 = 4；
依次类推，得到：

f(n) = f(0) + f(1) + f(2) + ⋯ + f(n - 1) + 1, n >= 1
f(n - 1) = f(0) + f(1) + f(2) + ⋯ + f(n - 2) + 1, n >= 2
整理可得：f(n) = 2*f(n - 1)，n >= 2，且 f(1) = 1。这就是我们高中所学的等比数列通项公式。不难得出，n>=1。

（2）Python 代码

```python
# -*- coding:utf-8 -*-
class Solution:
    def jumpFloorII(self, number):
        # write code here
        if number<=0: return 0
        if number>=1: return pow(2,number-1) #数学归纳法得出结论
```

注意：这里如果不用内置函数 pow()，用 2**(number - 1)，时间效率会低几十倍!!!

三、装箱问题与背包问题

题型：有一个箱子容量为 V（正整数，0<=V<=20000），同时有 n 个物品（0<n<=30），每个物品有一个体积（正整数）要求 n 个物品中，任取若干个装入箱内，使箱子的剩余空间为最小。

输入描述：

一个整数 v，表示箱子容量
一个整数 n，表示有 n 个物品
接下来 n 个整数，分别表示这 n 个物品的各自体积
输出描述：

一个整数，表示箱子剩余空间。
样例输入：

24
6
8
3
12
7
9
7

样例输出：

0

（1）分析过程

属于背包型动态规划，相当于背包容量和背包中物品价值二者相等的一般背包问题（貌似也称为伪背包问题）。通过转化思想即求：在总体积为 V 的情况下，可以得到的最大价值，最

后再用总体积减去最大价值时所占空间就是剩下的最少空间。

假设每个物品 i 的体积为 Vi，i=1,2,⋯,n，dp[i][j]表示前 i 件物品装入体积为 j 的箱子，箱子总共所占的最大体积。一共 n 件物品，那么 dp[n][V]就是前 n 件物品选择部分装入体积为 V 的箱子后，箱子所占的最大体积。

当当前输入的物品体积大于箱子容量剩余空间 j 时，即 Vi > j，则不装箱，得到：dp[i][j] = dp[i - 1][j]；
当当前输入的物品体积小于等于箱子容量剩余空间 j 时，即 Vi <= j，则需要考虑装与不装两种状态，取体积最大的那一个：dp[i][j] = max(dp[i - 1][j]，dp[i - 1][j - t] + t)，t = Vi。

以上思路是二维表的情况，若改为一维表呢？对于每一个物品 i，都存在放入箱子和不放入箱子两种情况。当前箱子容量剩余 j 时，若 i 放入，则为 dp[j - a[i]] + a[i]；若 i 不放入，则为 dp[i]；因此，状态转移方程为：dp[j] = max(dp[j], dp[j - a[i]] + a[i])。

（2）Python 代码

二维表情况：

```python
def solveBinPacking(V, arr):
    len1 = len(arr)
    if V<=0 and len1 == 0:
        return None
    dp = [[0]*(V+1) for row in range(len1+1)] # 初始化
    for i in range(1, len1 + 1):
        t = arr[i-1]
        for j in range(1, V+1):
            if j>= t:
                dp[i][j] = max(dp[i-1][j], dp[i-1][j-t] + t)
            else:
                dp[i][j] = dp[i-1][j]
    return V-dp[len1][V]


if __name__ == '__main__':
    V = int(input()) # 最大体积
    n = int(input()) # 物品数量
    arr = []
    for i in range(n):
        tmp = int(input())
        arr.append(tmp)

```

```
    print(solveBinPacking(V, arr))
```

一维表情况：

```
def solveBinPacking(V, arr):
    len1 = len(arr)
    if V<=0 and len1 == 0:
        return None
    dp = [0]*(V+1)
    for i in range(len1):
        for j in range(arr[i], V+1):
            dp[j] = max(dp[j], dp[j - arr[i]] + arr[i])
    return V-dp[V]

if __name__ == '__main__':
    V = int(input()) # 最大体积
    n = int(input()) # 物品数量
    arr = []
    for i in range(n):
        tmp = int(input())
        arr.append(tmp)

    print(solveBinPacking(V, arr))
```

四、最大递增子序列问题（最长上升子序列问题）

题目：最长上升子序列问题（LIS），给定 n 个整数 A1,A2,⋯,AnA1,A2,⋯,An，按从左到右的顺序选出尽量多的整数，组成一个上升子序列。 例如序列 1, 6, 2, 3, 7, 5，可以选出上升子序列 1, 2, 3, 5，也可以选出 1, 6, 7，但前者更长。选出的上升子序列中相邻元素不能相等。

子序列可以理解为：删除 0 个或多个数，其他数的顺序不变，数学定义为：已知序列 U_1，U_2，⋯，U_n，其中 U_i<U_(i+1)，且 A[U_i]<A[U_(i+1)])。常见考题为：对于一个数字序列，请设计一个算法，返回该序列的最大上升子序列的长度。

输入描述及样例（给定一个数字序列）：

[2, 1, 4, 3, 1, 5, 6]

输出描述及样例（最长上升子序列的长度）：

4

（1）分析过程

假设 dp[i]表示以标识为 i 的元素为递增序列结尾元素的最长递增子序列的长度，由于这里的递增序列不要求严格相邻，因 此 arr[i]需要和每一个 arr[j] (i > j) 比较，该方法的算法复杂度为 O(N^2)：

若存在 arr[i] > arr[j]，说明第 i 个元素可以接在第 j 个元素后面作为新的递增序列的结尾，即 dp[i] = max(dp[j])+1 = max(dp[j] + 1)；
若存在 arr[i] <= arr[j]，说明第 i 个元素比前面所有的数都小，此时以 i 元素作为结尾的递增序列长度为 1，即 dp[i] = 1；
最后，取出 dp 中最大的值就是最长的递增子序列的长度。
因此，状态转移方程为：当 arr[i] <= arr[j] 且 j < i 时，dp[i] = max{1, dp[j] + 1}。

哎呀，感觉有点懵逼，举个实际例子分析一下：

以一个例子为例：2 3 1 5
（1）对于 2，最长递增子序列为 1
（2）对于 3，最长递增子序列为 2
（3）对于 1，最长递增子序列为 2,3，但该处因为相当于和前面的断开了，所以应该定义此处的最长递增子序列为 1
（4）对于 5，如果和前面的 1 连接，最长递增子序列为 1,5，长度为 2；如果和前面的 3 连接，最长递增子序列为 2,3,5，长度为 3
综上所述，最长递增子序列为 2,3,5，长度为 3

（2）Python 代码

A、算法复杂度为 O(N^2)的代码：

```python
def lengthOfLIS(arr):
    len1 = len(arr)
    if len1==0:
        return None
    dp = [0]*len1
    dp[0] = 1
    for i in range(1, len1):
        maxValue = 0
        for j in range(i):
```

```python
            if arr[i]>arr[j]:
                if maxValue < dp[j] + 1:
                    maxValue = dp[j] + 1
            else:
                if maxValue < 1:
                    maxValue = 1
        dp[i] = maxValue
    print(max(dp))

if __name__ == '__main__':
    arr = [int(i) for i in input().split()]
    lengthOfLIS(arr)
```

或者

```python
def lengthOfLIS(nums):
    if nums == []:
        return 0
    N = len(nums)
    Dp = [1] * N
    for i in range(N - 1):
        for j in range(0, i + 1):
            if nums[i + 1] > nums[j]:
                Dp[i + 1] = max(Dp[i + 1], Dp[j] + 1)
    print(max(Dp))

if __name__ == '__main__':
    arr = [int(i) for i in input().split()]
    lengthOfLIS(arr)
```

B、算法复杂度为 O(NlogN)的代码

参考：https://blog.csdn.net/zhangyx_xyz/article/details/50949957

五、最长公共子序列问题(LCS)

问题：字符序列的子序列是指从给定字符序列中随意地（不一定连续）去掉若干个字符（可

能一个也不去掉）后所形成的字符序列。令给定的字符序列 X = “x0，x1，…，xm-1”，序列 Y = “y0，y1，…，yk-1”是 X 的子序列，存在 X 的一个严格递增下标序列 <i0，i1，…，ik-1>，使得对所有的 j=0，1，…，k-1，有 xij = yj。例如，X=“ABCBDAB”，Y=“BCDB”是 X 的一个子序列。

子序列的基本概念，可参考：https://blog.csdn.net/lz161530245/article/details/76943991

（1）分析过程，参考：https://blog.csdn.net/wangdd_199326/article/details/76464333

假设序列 A = [B, D, C, A, B, A]，序列 B = [A, B, C, B, D, A, B]。M 与 N 分别表示序列 A 与 B 的长度。我们来看看怎么得到最长公共子序列 LCS = [B, C, B, A]。这里需要说明的是最长公共子序列的答案并不唯一，但是最长公共子序列的长度唯一，因此一般求得都是长度!!! 假设 dp[i][j]表示 A 序列中前 i 个字符与 B 序列中前 j 个字符的最大公共子序列长度，那么：

当 i = 0 或 j = 0 时，dp[0][0] = 0;
当 i > 0，j > 0 且 A[i] = B[j] 时，dp[i][j] = dp[i - 1][j - 1] + 1;
当 i > 0，j > 0 且 A[i] ！ = B[j] 时，dp[i][j] = max(dp[i - 1][j]， dp[i][j - 1]);

因此，最后的最长公共子序列长度为：dp[M] [N]。动态规划求解的时间复杂度为 O(M*N)，空间复杂度也为 O(M*N)。但是在面试的时候，面试官其实更希望面试者能求着具体的最长公共子序列，而不仅仅是求其长度。原因是，工作中这种工程问题，长度求出来是没有任何用的!!! 求出所有的公共子序列才是工作中应具备的能力。

（2）Python 代码

动态规划思想：

```python
def lengthOfLongestCommonSubsequence(arrA, arrB):
    if arrA == [] or arrA == []:
        return 0
    M, N = len(arrA), len(arrB)
    dp = [[0]*(N + 1) for row in range(M + 1)]
    for i in range(1, M + 1):
        for j in range(1, N + 1):
            if arrA[i - 1] == arrB[j - 1]:
                dp[i][j] = dp[i-1][j-1] + 1
            else:
                dp[i][j] = max(dp[i-1][j], dp[i][j-1])
    print(dp[M][N])

if __name__ == '__main__':
    arrA = [i for i in input().split()]
```

```python
    arrB = [i for i in input().split()]
    lengthOfLongestCommonSubsequence(arrA, arrB)
```

如果需要求出具体的最长公共子序列，可以参考：Python 中最长的公共子序列

```python
def LongestCommonSubsequence(s1, s2):
    matrix = [["" for x in range(len(s2))] for x in range(len(s1))]
    for i in range(len(s1)):
        for j in range(len(s2)):
            if s1[i] == s2[j]:
                if i == 0 or j == 0:
                    matrix[i][j] = s1[i]
                else:
                    matrix[i][j] = matrix[i-1][j-1] + s1[i]
            else:
                matrix[i][j] = max(matrix[i-1][j], matrix[i][j-1])

    cs = matrix[-1][-1]
    return len(cs), cs

if __name__ == "__main__":
    s1 = "1234ABCD"
    s2 = "ABCD1234"
    print("s1 与 s2 的最长公共子序列：", LongestCommonSubsequence(s1, s2))
```

s1 与 s2 的最长公共子序列： (4, 'ABCD')
请注意：长度唯一，但最长公共子序列却不一定唯一。例如：

s1 = "1234ABCDabcd"
s2 = "abcdABCD1234"
大家都能很明显地看出来，公共子序列是["1234", "abcd", "ABCD"]。那怎么全部求出来？

解决方案：[python] 获得所有的最长公共子序列，原作者的代码优化结果如下：

```python
class LCS_naive:
    """
    最长公共子序列：
        通过动态规划，得到矩阵 D，
        并从矩阵 D 中读出一个最长公共子序列
        不支持所有的最长公共子序列
    """

    def __init__(self, str1, str2):
```

```
        self.matrix = [[]]
        self.str1 = str1
        self.str2 = str2
        self.len1 = len(str1)
        self.len2 = len(str2)
        self.matrix = [[0 for i in range(self.len2 + 1)] for j in range(self.len1 + 1)]


    def _get_matrix(self):
        """通过动态规划，构建矩阵"""
        for i in range(self.len1):
            for j in range(self.len2):
                if self.str1[i] == self.str2[j]:
                    self.matrix[i + 1][j + 1] = self.matrix[i][j] + 1
                else:
                    self.matrix[i + 1][j + 1] = max(self.matrix[i][j + 1], self.matrix[i + 1][j])

    def _matrix_show(self, matrix):
        """展示通过动态规划所构建的矩阵"""
        print("-------matrix-------")
        print(" ", " ", end=" ")
        for ch in self.str2:
            print(ch, end=" ")
        print()
        for i in range(len(matrix)):
            if i > 0:
                print(self.str1[i - 1], end=" ")
            else:
                print(" ", end=" ")
            for j in range(len(matrix[i])):
                print(matrix[i][j], end=" ")
            print()
        print("--------------------")

    def _get_one_lcs_from_matrix(self):
        i = len(self.matrix) - 1
        if i == 0:
            print("matrix is too small")
            return
        j = len(self.matrix[0]) - 1
        res = []
        while not (i == 0 or j == 0):
```

```
            if self.str1[i - 1] == self.str2[j - 1]:
                res.append(self.str1[i - 1])
                i -= 1
                j -= 1
            else:
                if self.matrix[i - 1][j] > self.matrix[i][j - 1]:
                    i = i - 1
                else:
                    j = j - 1
        return "".join(res[::-1])

    def get_lcs(self):
        self._get_matrix()
        self._matrix_show(self.matrix)
        lcs = self._get_one_lcs_from_matrix()
        print("最长公共子序列: ", lcs)



class LCS(LCS_naive):
    """
    继承自 LCS_naive
    增加获取所有 LCS 的支持
    """
    def __init__(self, s1, s2):
        LCS_naive.__init__(self, s1, s2)
        self.LCS = []

    def _get_all_lcs_from_matrix(self):
        self._pre_travesal(self.len1, self.len2, [])

    def _pre_travesal(self, i, j, lcs_ted):
        if i == 0 or j == 0:
            self.LCS.append("".join(lcs_ted[::-1]))
            # print("".join(lcs_ted[::-1]))
            return
        if self.str1[i - 1] == self.str2[j - 1]:
            lcs_ted.append(self.str1[i - 1])
            self._pre_travesal(i - 1, j - 1, lcs_ted)
        else:
            if self.matrix[i - 1][j] > self.matrix[i][j - 1]:
                self._pre_travesal(i - 1, j, lcs_ted)
            elif self.matrix[i - 1][j] < self.matrix[i][j - 1]:
                self._pre_travesal(i, j - 1, lcs_ted)
```

```
            else:
                ###### 分支
                self._pre_travesal(i - 1, j, lcs_ted[:])
                self._pre_travesal(i, j - 1, lcs_ted)


    # 注释掉只有一种结果，不注释得到多个结果
    def get_lcs(self):
        self._get_matrix()
        self._matrix_show(self.matrix)
        self._get_all_lcs_from_matrix()
        print("所有的最长公共子序列:", list(set(self.LCS)))


if __name__ == "__main__":
    lcs = LCS("1234ABCDabcd", "abcdABCD1234")
    lcs.get_lcs()
```

另外，回溯输出最长公共子序列过程：

算法分析：
由于每次调用至少向上或向左（或向上向左同时）移动一步，故最多调用(m + n)次就会遇到 i = 0 或 j = 0 的情况，此时开始返回。返回时与递归调用时方向相反，步数相同，故回溯法算法时间复杂度为 Θ(m + n)。

补充：相信很多人看到这个图想到了棋盘问题！详细介绍可见博客：https://blog.csdn.net/tomcmd/article/details/47906787。只不过，假设棋盘问题是求从左上角点 A，走到右下角点 B 的路径总数，此时，初始化二维表的时候，第一行与第一列设置为 1 即可。

棋盘问题面试经历（题型总结）：C(m , n) = m! /[n!(m-n)!] 以下结论前提是，左上角 A，右下角 B 均已在棋盘上（啥玩意儿？就是让你从 A 走到 B，这里容易混淆！）。

A、一个 m*n 的网格（左上到右下的最短路径长度为 m + n -1），问从左下角到右上角的最短路径有多少种？（等价于问从左下角到右上角的走法有多少种？）要求每次只能向下或向右移动一格

答案：从 m+n 步中选出 m - 1 步向下或 n - 1 步向右，因此为 result = f (m , n) = C(m + n - 2 , m - 1) = C(m + n - 2 , n - 1) 种

B、一个 m*n 的网格，中间有个位置 P 标记上“X”不能走，问从左下角到右上角的走法有多少种？（等价于问从左下角到右上角的最短路径有多少种？）要求每次只能向下或向右移动一格

答案：假设有一个点 P 不能走，且位置为(x , y)，$1 <= x <= m$ ，$1 <= y <= n$，那么分三步骤：

（1）如果没有点 P 时，先求 f (m , n)；

（2）考虑点 P，计算 $f(x, y) * f(m - x + 1, n - y + 1)$

（3）最终结果为：$res = f(m, n) - f(x, y) * f(m - x + 1, n - y + 1)$

$$= C(m + n - 2, n - 1) - C(x + y - 2, x - 1) * C(m + n - x - y, m - x)$$

$$= C(m + n - 2, n - 1) - C(x + y - 2, y - 1) * C(m + n - x - y, n - y)$$

注意：棋盘问题一定要注意审题，有的是 C(m + n , m)，为什么？因为起始点，终点不在棋盘上！

题目 1：在如下 4*5 的矩阵中，请计算从左下角 A 移动到右上角 B 一共有_____种走法？。要求每次只能向上或向右移动一格，并且不能经过 P(3 , 3)。

答案：17 = C(7, 3) -C (4 , 2)*C(3 , 1) = 35 - 6x3

题目 2：现有一 5×6 的矩形网格，问从矩形最右上角一点到最左下角一点有几种路径?

答案：126

棋盘问题的代码实现：

A、递归+动态规划

B、分析过程

六、最长公共子串问题

题目：给定两个字符串,求出它们的最长公共子串（连续）

（1）分析过程

这个题其实与最长公共子序列很像，唯一的区别就是这里要求连续的！假设字符串 A = “1AB2345CD”，字符串 B = “12345EF”，M 与 N 分别是字符串 A 与 B 的长度，最长公共子串为“2345”。假设 dp[i][j]表示 A 串中的前 i 个字符与 B 串中的前 j 个字符的最长公共子串的长度，那么：

当 i = 0 或 j = 0 时，dp[0][0] = 0；
当 i > 0，j > 0 且 A[i] = B[j] 时，dp[i][j] = dp[i - 1][j - 1] + 1；
当 i > 0，j > 0 且 A[i] ！ = B[j] 时，dp[i][j] = 0；

因此，最后的最长公共子串长度为：max(dp)，即 dp 中长度最大的值就是最长公共子串的长度。动态规划求解的时间复杂度为 O(M*N)，空间复杂度也为 O(M*N)。面试的时候，面试官其实更希望面试者能求着具体的最长公共子串，而不仅仅是求其长度。

请注意：长度唯一，但最长公共子串却不一定唯一。实际工程项目中，求出所有的最长公共子串的实用性远大于求出最长公共子串长度。出于不同的需求，这里罗列了求 LCS 的长度，单个 LCS，以及所有 LCS 的 python 代码。

（2）Python 代码 -- 求最长公共子串的长度

```python
def lengthOfLongestCommonSubstring(arrA, arrB):
    M, N = len(arrA), len(arrB)
    if M == 0  or N == 0:
        return 0
    maxValue = 0
    dp = [[0]*(N + 1) for row in range(M + 1)]
    for i in range(1, M + 1):
        for j in range(1, N + 1):
            if arrA[i - 1] == arrB[j - 1]:
                dp[i][j] = dp[i-1][j-1] + 1
            else:
                dp[i][j] = 0
            if maxValue < dp[i][j]:
                maxValue = dp[i][j]
    print(maxValue)

if __name__ == '__main__':
    arrA = input()
    arrB = input()
    lengthOfLongestCommonSubstring(arrA, arrB)
```

（3）Python 代码 -- 求具体的最长公共子串

```python
import time

# 方法一
def findLongestCommonSubstring(s1, s2):
    lcs, temp = [], []
    for i in range(len(s1)):
```

```python
        for j in range(len(s2)):
            if s1[i] == s2[j]:
                k, z = i, j
                while s1[k] == s2[z]:
                    temp += s1[k]
                    if k+1 < len(s1) and z+1 < len(s2):
                        k, z = k + 1, z + 1
                    else:
                        break
                if len(temp) > len(lcs):
                    lcs = temp
                temp = []
    return "".join(lcs)

# 方法二
def find_LongestCommonSubstring(s1, s2):
    matrix = [[0 for _ in range(len(s2) + 1)] for _ in range(len(s1) + 1)]
    max_lens, substr_end_index = 0, 0
    for i in range(len(s1)):
        for j in range(len(s2)):
            if s1[i] == s2[j]:
                matrix[i+1][j+1] = matrix[i][j] + 1
                if matrix[i+1][j+1] > max_lens:
                    max_lens, substr_end_index = matrix[i+1][j+1], i + 1
    substr = s1[substr_end_index - max_lens : substr_end_index]
    return substr


if __name__ == '__main__':
    s1 = "1AB2345CD"
    s2 = "12345EF"

    # start_time1 = time.time()
    print("s1 与 s2 的最长公共子串为:", findLongestCommonSubstring(s1, s2))
    # print("耗时:{0} ms!".format(round(1000*(time.time() - start_time1)), 3))

    # start_time2 = time.time()
    print("s1 与 s2 的最长公共子串为:", find_LongestCommonSubstring(s1, s2))
    # print("耗时:{0} ms!".format(round(1000 * (time.time() - start_time2)), 3))
```

s1 与 s2 的最长公共子串为: 2345

s1 与 s2 的最长公共子串为: 2345

那么问题来了，这个最长公共子串不唯一怎么办？

例子：A = "1234ASD5678", B = "A1234fgs5678sa" 。以上代码的运行结果：

s1 与 s2 的最长公共子串为: 1234
s1 与 s2 的最长公共子串为: 1234

说明代码存在逻辑问题。也就是无法求出所有的最长公共子串，修改后的版本如下：

```python
def findLongestCommonSubstring(s1, s2):
    mylcs, lcs, temp = [], [], []
    for i in range(len(s1)):
        for j in range(len(s2)):
            if s1[i] == s2[j]:
                k, z = i, j
                while s1[k] == s2[z]:
                    temp += s1[k]
                    if k+1 < len(s1) and z+1 < len(s2):
                        k, z = k + 1, z + 1
                    else:
                        break
                if len(temp) >= len(lcs):
                    lcs = temp
                    mylcs.append("".join(lcs))
                temp = []
    return mylcs


if __name__ == '__main__':
    s1 = "1234ASD5678"
    s2 = "A1234fgs5678sa"

    print("s1 与 s2 的最长公共子串为:", findLongestCommonSubstring(s1, s2))
```

七、最大连续子序列求和问题（最大子串求和问题）——Max Sum

问题：给定 K 个整数的序列{N1, N2, ……, Nk}，其中任意连续子序列可表示为 {Ni, Ni+1, ……, Nj}，其中 1 <= i <= j <= k。最大连续子序列是所有连续子序列中元素和最大的一个，例如给定序列{-2，11，-4，3，-5，-2}，其最大连续子序列为{11，-4，13}，最大连续子序列和为 20。

重点参考博客：https://www.cnblogs.com/conw/p/5896155.html。
Python 编程代码参考：https://blog.csdn.net/yangfengyougu/article/details/81807950

（1）时间复杂度为 O(N^3)的解法——穷举

思想：穷举求出所有连续子序列的序列和，再求最大！

```python
def MaxSubSequence(arr):
    if arr == []:
        return None
    M = len(arr)
    MaxSum = 0
    for i in range(M):
        for j in range(i, M):
            tmpSum = 0
            for k in range(i, j+1):
                tmpSum += arr[k]
            if tmpSum > MaxSum:
                MaxSum = tmpSum
    print(MaxSum)

if __name__ == '__main__':
    arr = [int(i) for i in input().split()]
    MaxSubSequence(arr)
```

（2）时间复杂度为 O(N^2)的解法——穷举法的优化，去除内层循环

```python
def MaxSubSequence(arr):
    if arr == []:
        return None
    M = len(arr)
```

```python
    MaxSum = 0
    for i in range(M):
        tmpSum = 0
        for j in range(i, M):
            tmpSum += arr[j]
            if tmpSum > MaxSum:
                MaxSum = tmpSum
    print(MaxSum)

if __name__ == '__main__':
    arr = [int(i) for i in input().split()]
    MaxSubSequence(arr)
```

（3）时间复杂度为 O(NlogN)的解法——分治法

思想：首先，我们可以把整个序列平均分成左右两部分，答案则会在以下三种情况中：

所求序列完全包含在左半部分的序列中。
所求序列完全包含在右半部分的序列中。
所求序列刚好横跨分割点，即左右序列各占一部分。

前两种情况和大问题一样，只是规模小了些，如果三个子问题都能解决，那么答案就是三个结果的最大值。 以分割点为起点向左的最大连续序列和、以分割点为起点向右的最大连续序列和，这两个结果的和就是第三种情况的答案。

因为已知起点，所以这两个结果都能在 O(N)的时间复杂度能算出来。递归不断减小问题的规模，直到序列长度为 1 的时候，那答案就是序列中那个数字。

代码为：

```python
def maxsum(nums):
    if len(nums) == 1:
        return nums[0]
    #分组
    center = len(nums)//2
    left_nums = nums[0:center]
    right_nums = nums[center:len(nums)]

    #分别求左右序列最大子序列和
    left_maxsum = maxsum3(left_nums)
    right_maxsum = maxsum3(right_nums)

    #求左序列最大和(包括最后一个元素)
```

```
    left_sum = 0
    left_max= left_nums[len(left_nums)-1]
    i = len(left_nums)-1
    while i >= 0:
        left_sum += left_nums[i]

        if left_sum > left_max:
            left_max = left_sum
        i -= 1

    #求右序列最大和(包括第一个元素)
    right_sum =0
    right_max = right_nums[0]
    i = 0
    while i < len(right_nums):
        right_sum += right_nums[i]
        if right_sum > right_max:
            right_max = right_sum
        i += 1

    l = [left_maxsum,right_maxsum,left_max + right_max]
    return max(l)

if __name__ == '__main__':
    arr = [int(i) for i in input().split()]
    res = maxsum(arr)
    print(res)
```

（4）时间复杂度为 O(N)的解法——动态规划（面试常考!）

例如：序列 A = {-2, 11, -4, 3, -5, -2}，其最大连续子序列为{11, -4, 13}，最大连续子序列和为 20。

假设 dp[i] 表示以 A[i] 为子序列末端的最大连续和，因为 dp[i]要求必须以 A[i]结尾的连续序列，那么只有两种情况：

最大连续序列只有一个元素，即以 A[i]开始，以 A[i]结尾 ，最大和就是 A[i]本身
最大和的连续序列有多个元素，即以 A[p]开始（p 小于 i），以 A[i]结尾，最大和是 dp[i - 1] + A[i]
因此状态转移方程为：dp[i] = max (dp[i -1] + A[i]，A[i])

最后，连续子序列的和为：maxsub[n] = max (dp [i])，1 <= i <= n

Python 代码为:

```python
def maxsum(nums):
    if len(nums) == 1:# 判断序列长度，若为 1，直接返回
        return nums[0]
    dp = res = nums[0]
    for i in range(1,len(nums)):
        dp = max(nums[i],dp + nums[i])
        res = max(dp,res)
    print(res)


if __name__ == '__main__':
    arr = [int(i) for i in input().split()]
    maxsum(arr)
```

八、股票收益最大化（一次交易、多次交易与最多两次交易）

问题 1：假设把某股票的价格按照时间先后顺序存储在数组中，请问买卖该股票一次可获得的最大利润是多少？

例如，一只股票在某些时间节点的价格为{9,11,8,5,7,12,16,14}。如果我们能在价格为 5 的时候买入并在价格为 16 时卖出，则能获得最大的利润为 11。规定无论如何买，都会亏，即是一个从大到小排序的数组，此时返回 0，如，arr = [4, 3, 2, 1]，输出为 0。

分析思路：（记录当前最小值和最大差值）

给定一个数组 arr，初始化最小值 minPrice = arr[0]，最大利润 maxPrice = arr[1] - arr[0]；遍历数组，若求最大利润 maxPrice，即是计算当前的最小值 minPrice 后面的数字减去 minPrice，得到的一个最大的差值 diffPrice，如果 diffPrice 大于 maxPrice，则 maxPrice = diffPrice。
最后，判断 maxPrice，若 maxPrice>=0，输出即可，若 maxPrice<0，则 maxPrice = 0。

Python 代码：（时间复杂度为 O(N)，空间复杂度为 O(1)）

```python
# 股票收益最大化问题总结
def BestStock_1_time(arr):
    len1 = len(arr)
    if len1 < 2:
```

```
        return 0
    minPrice = arr[0]
    maxPrice = arr[1] - arr[0]

    for i in range(2, len1):
        if arr[i - 1] < minPrice:
            minPrice = arr[i - 1]
        Diff = arr[i] - minPrice
        if Diff > maxPrice:
            maxPrice = Diff
    if maxPrice < 0:
        maxPrice = 0
    return maxPrice

if __name__ == '__main__':
    try:
        while True:
            arr = [int(i) for i in input().split()]
            print(BestStock_1_time(arr))
    except:
        pass
```

C++代码:

```
class Solution {
public:
    int maxProfit(vector<int> &prices) {
        int len = prices.size();
        if (len<2)
            return 0;
        int minPrice = prices[0];
        int maxPrice = prices[1] - prices[0];
        for (int i=2;i<len;i++){
            if (prices[i-1]<minPrice)
                minPrice = prices[i-1];
            int Diff = prices[i] - minPrice;
            if (Diff>maxPrice)
                maxPrice = Diff;
        }
        if (maxPrice<0)
            maxPrice = 0;
        return maxPrice;
    }
```

```
};
```

问题 2：假设把某股票的价格按照时间先后顺序存储在数组中，请问买卖该股票多次可获得的最大利润是多少？

例如:

股票价格 [77, 84, 59, 56, 69, 38, 53, 77, 35, 89]

股票价格差值 [7, -25, -3, 13, -31, 15, 24, -42, 54]
股票增值数 [7, 13, 15, 24, 54]
股票最大收益 113
这样思路很明了，就是求股票价格差值中的所有正数累加和!

Python 代码:（时间复杂度为 O(N)，空间复杂度为 O(N)，方便理解）

```python
def BestStock_n_time(arr):
    len1 = len(arr)
    if len1 < 2:
        return 0

    diffArr = []  # 股票价格差值
    for i in range(len1 - 1):
        diffArr.append(arr[i + 1] - arr[i])
    sum = 0  # 股票最大收益
    for i in range(len(diffArr)):
        if diffArr[i] > 0:
            sum += diffArr[i]
    return sum

if __name__ == '__main__':
    try:
        while True:
            arr = [int(i) for i in input().split()]
            print(BestStock_n_time(arr))
    except:
        pass
```

空间复杂度还可以降为 O(1)，函数为:

```python
def BestStock_n_time(arr):
    len1 = len(arr)
    if len1 < 2:
```

```
        return 0

    sum = 0  # 股票的最大收益
    for i in range(len1 - 1):
        if arr[i + 1] - arr[i] > 0:
            sum += arr[i + 1] - arr[i]
    return sum
C++代码为：

class Solution {
public:
    int maxProfit(vector<int> &prices) {
        int len = prices.size();
        if (len<2)
            return 0;
        vector<int> diffArr;
        for (int i = 0;i < len - 1;i++)
            diffArr.push_back(prices[i+1] - prices[i]);
        int sum = 0;
        for (int i = 0;i < diffArr.size();i++){
            if (diffArr[i]>0)
                sum += diffArr[i];
        }
        return sum;
    }
};
```

问题 3：假设把某股票的价格按照时间先后顺序存储在数组中，请问买卖该股票最多两次可获得的最大利润是多少？

例如，数组 arr = [1, 5 , 2 , 6 , 9 , 10 , 2]，第一次购买价格为 1，第一次卖出价格为 5，第二次购买价格为 2，第二次卖出价格为 10，总共的最大收益为 4 + 8 = 12。

思路 1：分段考虑

以 i 为分界线，前 i 天的最大和 i 天后面的最大，分两段进行每次的一个交易；
两段的最大和，则为最大的利润；

思路 2：动态规划

Buy1 [i] 表示前 i 天做第一笔交易买入股票后剩下的最多的钱；
Sell1 [i] 表示前 i 天做第一笔交易卖出股票后剩下的最多的钱；
Buy2 [i] 表示前 i 天做第二笔交易买入股票后剩下的最多的钱；

Sell2 [i] 表示前 i 天做第二笔交易卖出股票后剩下的最多的钱;

那么存在如下关系:

Buy1 [i] = max { Buy1 [i - 1] , - prices [i] }
Sell1 [i] = max { Sell1 [i - 1] , Buy1 [i - 1] + prices [i] }
Buy2 [i] = max { Buy2 [i - 1] , Sell2 [i - 1] - prices [i] }
Sell2 [i] = max { Sell2 [i - 1] , Buy2 [i - 1] + prices [i] }

最终的输出结果为：Sell2，即为最多两次交易的股票最大收益值。

可以发现上面四个状态都是只与前一个状态有关, 所以可以不使用数组而是使用变量来存储即可。

Python 代码：（时间复杂度为 O(N)，空间复杂度为 O(1)）

```python
from sys import maxsize  # 导入整数最大值
def BestStock_most_2_time(arr):
    buy1, sell1, buy2, sell2 = -maxsize, 0, -maxsize, 0  # 初始化四个变量：整数最小值与 0
    for i in range(len(arr)):
        buy1 = max(buy1, -arr[i])  # 第一次买入
        sell1 = max(sell1, buy1 + arr[i])  # 第一次卖出
        buy2 = max(buy2, sell2 - arr[i])  # 第二次买入
        sell2 = max(sell2, buy2 + arr[i])  # 第二次卖出
    return sell2

if __name__ == '__main__':
    try:
        while True:
            arr = [int(i) for i in input().split()]
            print(BestStock_most_2_time(arr))
    except:
        pass
```

C++代码:

```cpp
class Solution {
public:
    int maxProfit(vector<int>& prices) {
        int buy1 = INT_MIN, sell1 = 0, buy2 = INT_MIN, sell2 = 0;
        for(int i = 0; i < prices.size(); i++) {
            buy1 = max(buy1, -prices[i]);
            sell1 = max(sell1, buy1 + prices[i]);
```

```
            buy2 = max(buy2, sell1 - prices[i]);
            sell2 = max(sell2, buy2 + prices[i]);
        }
        return sell2;
    }
};
```