

实验二：动态规划算法设计

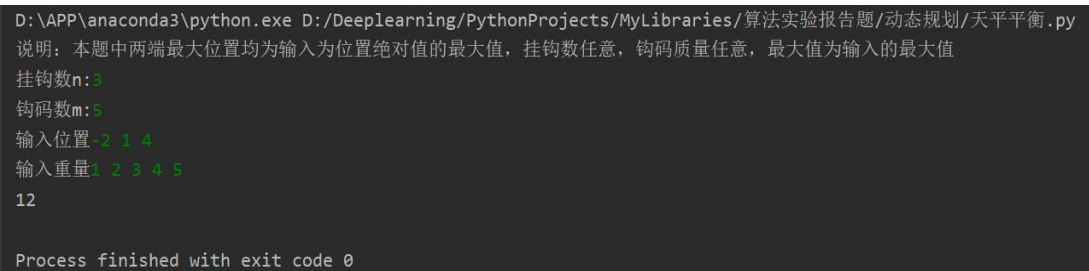
学号	202018526	姓名	高树林	成绩	
友情提示	1. 算法描述及代码实现与网络或他人雷同者，均按 0 分计算； 2. 要求算法描述明确、代码清晰、格式美观； 3. 纸质版与电子版同时提交（电子版命名格式： 完整学号-姓名-实验 X-实验名称 ，其中 $X \in \{1,2,3,4\}$ ，不得省略“-”，如：2088166-乔峰-实验 1-分治与递归策略）； 4. 电子版中代码需格式化处理，方便查看（ http://www.codeinword.com/ ）。				
实验目的	1. 掌握动态规划解决问题的一般过程。 2. 通过使用动态规划算法求解给定问题，学会使用动态规划解决实际问题。				
实验内容	1. 天平平衡问题：已知一个天平左右两端共有 n 个挂钩，且有 m 个不同质量的钩码，求将钩码全部挂到钩子上使天平平衡的方法的总数。试设计求解该问题的动态规划算法。 2. 数塔问题：对于诸如下图的数塔，若从顶层走到底层，每一步只能走到相邻的结点，求经过的结点的数字之和最大的路径。试设计求解该问题的动态规划算法。				
算法描述	1. 天平平衡问题解题思路或算法思想： 动态规划思想就是遍历所有的状态，然后确定所求状态。在本题中，可以先遍历所有挂钩，将钩码全部挂到挂钩上的所有可能全部遍历出来，最后找到平衡的状态出现几次就是最后的结果。具体实现是设置一个平衡度 j 为代建 dp 二维列表的列，最大质量和最大挂钩位置都设置为离中心点最远的位置，将中心点位置设为 0，则向左为负，向右为正值。钩码质量和挂钩位置任意。 2. 数塔问题解题思路或算法思想 数塔问题的关键点在于全部列举最后比较。首先由于当前节点的子节点最终会与其做加和运算，因此可以选出与当前节点的子节点是哪一个，这里相当与剪枝，能够降低算法的复杂度。从此处向上迭代，直到第一行，此时第一个元素就是所求的值。这是一种自底向上的算法。具体实现为：在第 i 行时，比较 $dp[i][j]$ 和 $dp[i][j+1]$ 的值，找出较大者与上一行的 $dp[i-1][j]$ 做加和，这样决策，问题便将了一阶。反复迭代到最后一步，最后的 $dp[0][0]$ 就是最大的结果。 之后找到元素的位置则采用的是自顶向下的算法。使 dp 和 dp_new 的相对位置做差，结果赋给 cur ，如果 $cur == dp[i][j]$ 则说明下一步应该是 $dp_new[i][j]$ ；如果 $cur == dp[i][j+1]$ 则说明下一步应该是 $dp_new[i][j+1]$ 。依次循环下去就能找到路径。				
程序及运行结果（附图）	1. 天平平衡问题 代码： <pre> 1. def Findways(n, m, c, g, dp): 2. for i in range(1, m + 1):#最外层遍历是遍历钩码 3. for j in range(40 * abs(max(c)) * max(g)):#规定平衡度，将此设为二维表的列 4. if dp[i - 1][j]: # 在表中可自取 5. for k in range(1, n + 1):#遍历挂钩 6. dp[i][j + c[k] * g[i]] = dp[i][j + c[k] * g[i]] + dp[i - 1][j]# 7. 填表 8. return dp[m][20 * abs(max(c)) * max(g)] </pre>				

```

9.
10. if __name__ == "__main__":
11.     print("说明：本题中两端最大位置均为输入为位置绝对值的最大值，挂钩数任意，钩码质量任意，最大值为输入的最大值")
12.     n = int(input("挂钩数 n:"))
13.     m = int(input("钩码数 m:"))
14.     c = [0] + list(map(int, input("输入位置").split(' '))) + [0] * (19 - n)
15.     g = [0] + list(map(int, input("输入重量").split(' ')))
16.     g = g + [0] * (max(g) - m)
17.     dp = [[0] * 500 * abs(max(c)) * max(g) for _ in range(25)]
18.     dp[0][20 * abs(max(c)) * max(g)] = 1
19.     print(Findways(n, m, c, g, dp))

```

截图：



```

D:\APP\anaconda3\python.exe D:/Deeplearning/PythonProjects/MyLibraries/算法实验报告题/动态规划/天平平衡.py
说明：本题中两端最大位置均为输入为位置绝对值的最大值，挂钩数任意，钩码质量任意，最大值为输入的最大值
挂钩数n:3
钩码数m:5
输入位置-2 1 4
输入重量1 2 3 4 5
12
Process finished with exit code 0

```

1. 数塔问题

代码：

```

2. import copy
3.
4.
5. def findways(m, dp):
6.     dp_new = copy.deepcopy(dp) # 深拷贝，因为 dp 要后续修改，但是又会用到原来的值
7.     temp = -float('inf') # 因为要求最大，因此初始值要设为负无穷
8.     for i in range(m - 1, -1, -1):
9.         for j in range(i + 1):
10.            temp = max(dp[i][j], dp[i][j + 1]) # 找到第 i 行相邻两个元素中最大的两个元素
11.            dp[i - 1][j] = temp + dp_new[i - 1][j] # 将最大的元素和他的父节点相加，进入循环迭代，最后一次迭代出最大值
12.            print('%d' % dp_new[0][0], end='')
13.            j = 0
14.            for i in range(1, m):
15.                cur = dp[i-1][j] - dp_new[i-1][j]
16.                if cur == dp[i][j + 1]:
17.                    j += 1
18.                print('路径为: ', '->%d' % dp_new[i][j], end='')
19.            print('')
20.            return temp
21.

```

```

22.
23. if __name__ == '__main__':
24.     m = int(input('输入数塔深度: '))
25.     dp = [[0] * (m + 1) for _ in range(m + 1)] # 防止第 12 行出现越界情况, 因此在这里需
        要设置规模比原规模大 1
26.     for i in range(m):
27.         a = list(map(int, input('第%d 层: ' % i).split(' ')))
28.         dp[i] = a + [0] * (m - len(a) + 1)
29.     print('最大值为: ', findways(m, dp))

```

截图:

```

D:\APP\anaconda3\python.exe D:/Deeplearning/Pytho
输入数塔深度: 5
第0层: 10
第1层: 9 7
第2层: 8 3 7
第3层: 21 32 65 78
第4层: 2 5 2 8 1
路径为: 10->7->7->78->8
最大值为: 110

```

总结

动态规划题目在算法里面算是比较难的问题了, 但是该算法很实用, 能通过对所有结果进行比较与规划从而得到全局最优解。动态规划的最终归宿一般是一个二维表 **dp**, 在建表的过程中一般该步与上一步或者前几步的值是有联系的, 因此计算当前值的手只需要在 **dp** 表中找, 通过下标查询的时间复杂度为 $O(1)$, 因此大大降低了问题解决的时间复杂度。一般来说动态规划不是一步写出来的, 一般会经历以下几个步骤: 用递归调用方法先写出第一版, 第二版则是通过把第一版的递归项用 **dp** 表格缓存起来, 以便后续调用。第三版一般是通过分析第二版的表格生成关系直接将表格写出来, 这是最终版本的动态规划。上述两个代码都是第三版最终版代码。因此对于一道具体的题目现在我的理解是可以先举出几个比较简单的例子来分析一下, 进而扩展到整体。因为动态规划在解决问题的过程中就是利用开始的初值以后枚举各种情况从而确定下一阶段的最优情况并记录其值, 正是因为这个 **dp** 用的数组才有了特殊意义, 这样推下去就得到了最终的最优解。目前为止, 自己感觉这种记录中间阶段最优值得方法是 **dp** 的重要部分, 基因这个它才能把问题分阶段, 因为这个才去除了冗余, 刚开始是一一枚举, 但是随着每个最优解被记录下来, 越往后省去的运算越多 (从数字塔问题中明显看得出)。对于一个很大的问题根据 **dp** 是状态转换方程一步一步往前推, 现在自己还不能完全适应, 因为问题的解决过程是自下而上的, 由易到难的, 这种推理方式与其刚好相反。所以还是得多看点题, 写完代码后仔细看看程序是怎么运行的, 以渐渐适应这种方式。

对于一个问题还是比较容易确定是否采用动态规划的, 因为它是用来求最优值的, 一旦题目中有和最优相关的词语就得考虑一下它了。但是动态规划定义虽然简单, 用起来却是另一回事。即使知道要用动态规划也不一定真正根据题目应用好, 由于现在自己做看题目很少 (也就

	背包，最大字段，最长公共子序列，最大矩阵连乘积等)，仅仅能做那几道相似的题目。有人说：“动态规划看不 100 道题就不算入门。” 所以这里还要花很长时间。
--	---