动态规划算法的入门

原创 王国栋 LOA算法学习笔记 2021-02-06 13:06

01 前言

在学习完动态规划之后，总感觉以后能很容易的解决各种各样的动态规划问题了，没想到在遇上新题还是半知半解，有许多细节问题考虑不周全，要想真正做到融会贯通，必须要形成自己的思考逻辑。

作为一个算法小白，以下是我的理解：动态规划就是利用已知的**记忆答案**来解决新问题，这样可以避免重复计算，而这些**记忆答案**则需要用数组来保存。下面我们就讲一讲解决动态规划问题的步骤：

1.确定dp状态--dp状态的确定要遵循两个原则，一是**最优子结构**，二是**无后效性**，也就是说我们将原问题划分为多个子问题时，大规模子问题的最优解仅仅与小规模子问题最优解有关，而与小规模子问题最优解是如何得到的没有关系。

2.求出状态转移方程--确定dp状态后，根据dp状态确定转移方程。

3.列出状态的初始值。

4.求解题目要求的结果。

动态规划的题目一般遵循以下特点：

1.求某个方案的总数

2.求最大值/最小值

3.求是否可行

下面我们分别以三个问题来说明这三类题目，题目的难度依次递增。

02 问题一

2.1 问题描述

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？
来源：*https://leetcode-cn.com/problems/unique-paths/*

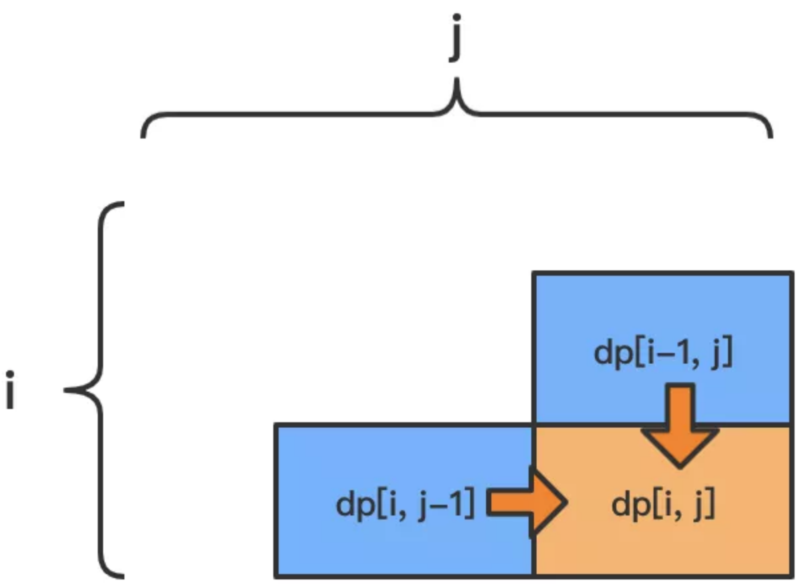

图1

2.2 问题分析

1）确定dp状态：
dp[i,j]：表示从原点到(i, j)的路径数，其中0≤i<n，0≤j<n。
2）dp方程：假设我们现在已经到(i,j)的位置，由图1，很容易得到，到达(i,j)的路径有两种：
从(i-1,j)向下走一步，
从(i, j-1)向右走一步，
因此，我们写出dp方程：dp[i,j]=dp[i-1,j]+dp[i,j-1]
3）初始化：我们可以看出，在第一行，机器人只能向右走；第一列，机器人只能向下走。因此第一行和第一列的边界条件都要设置为1。
4）最终要求的答案就是dp[m-1][n-1]。

2.3 代码实现

```
class Solution {
    public int uniquePaths(int m, int n) {
        int[][] dp = new int[m][n];
        //将第一列都设为1
        for (int i = 0; i < m; ++i) {
            dp[i][0] = 1;
        }
        //将第一行都设为1
        for (int j = 0; j < n; ++j) {
            dp[0][j] = 1;
        }
        for (int i = 1; i < m; ++i) {
            for (int j = 1; j < n; ++j) {
                dp[i][j] = dp[i - 1][j] + dp[i][j - 1];
            }
        }
        return dp[m - 1][n - 1];
    }
}
```

2.4 复杂度

时间复杂度：O(mn)。
空间复杂度：O(mn)，需要一个m×n的数组来储存所有的状态。

03 问题二

3.1 问题描述

给定不同面额的硬币 coins 和一个总金额 amount。编写一个函数来计算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。

Example 1:
Input: coins = [1,2,5], amount = 11
Output: 3
Explanation: 11 = 5 + 5 + 1
来源：*https://leetcode-cn.com/problems/coin-change/*

3.2 问题分析

1）确定dp状态：dp[amount]：凑成amount需要的的最少硬币个数

2）dp方程：

根据上面的例子，凑成amount=11需要的硬币数可能有以下几种情况：

①凑成10元的最少硬币数+1元这枚硬币。

②凑成9元的最少硬币数+2元这枚硬币。

③凑成6元的最少硬币数+5元这枚硬币。

对以上三种情况求最小值即可。即dp[11]=min(dp[10]+1,dp[9]+1,dp[6]+1)

因此，归纳dp方程：dp[amount]=min(dp[amount-coins[i]])

3）初始化：组成0的最少硬币数是0，所以dp[0] = 0。我们在初始化的时候设了一个不可能的值：amount+1，使得新状态的值发生状态转移。

4）最终答案是dp[amount]。

3.3 代码实现

```
public class Solution {


    public int coinChange(int[] coins, int amount) {
        int[] dp = new int[amount + 1];
        Arrays.fill(dp, amount + 1);
        dp[0] = 0;
        for (int i = 1; i <= amount; i++) {
            for (int coin : coins) {
                if (i - coin >= 0 && dp[i - coin] != amount + 1) {
                    dp[i] = Math.min(dp[i], 1 + dp[i - coin]);
                }
            }
        }
        if (dp[amount] == amount + 1) {
            dp[amount] = -1;
        }
        return dp[amount];
    }
}
```

3.4 复杂度

时间复杂度：O(N×amount)，N是硬币的种类数，amount是硬币的面额。

空间复杂度：O(amount)，数组大小是amount。

04 问题三

4.1 问题描述

一只青蛙想要过河。假定河流被等分为 x 个单元格，并且在每一个单元格内都有可能放有一石子（也有可能没有）。青蛙可以跳上石头，但是不可以跳入水中。

给定石子的位置列表（用单元格序号升序表示）， 请判定青蛙能否成功过河（即能否在最后一步跳至最后一个石子上）。 开始时， 青蛙默认已站在第一个石子上，并可以假定它第一步只能跳跃一个单位（即只能从单元格1跳至单元格2）。

如果青蛙上一步跳跃了 k 个单位，那么它接下来的跳跃距离只能选择为 k - 1、k 或 k + 1个单位。 另请注意，青蛙只能向前方（终点的方向）跳跃。

来源：*https://leetcode-cn.com/problems/frog-jump/*

4.2 问题分析

1）确定dp状态：dp[i][k]:表示能否通过前面的某一块**石头j**通过**跳k步**到达当前的**石头i**，其中，1≤j≤i-1, k是i和j石头之间的距离

那么，对于j石头来说，跳到j石头的上一个石头的步数必须是：k-1 or k or k+1

2）dp方程：dp[i][k] = dp[j][k - 1] || dp[j][k] || dp[j][k + 1]

3）初始化：只有1块石头的情况下，输出true；第一步的距离如果不是1，输出false。

4）如果dp[len-1][k]任何一个为true，则为true，否则为false，其中1≤k≤len-1。

4.3 代码实现

```
class Solution {
    public boolean canCross(int[] stones) {
        int len = stones.length;
        if (stones[1] != 1){
            return false;
        }
        boolean[][] dp = new boolean[len][len+1];
        dp[0][0]=true;
        for (int i=1;i<len;i++){
            for (int j=0;j<i;j++){
                int k = stones[i] - stones[j];
                if (k <= j+1){
                    dp[i][k] = dp[j][k-1] || dp[j][k] || dp[j][k+1];
                    if (i==len-1 && dp[i][k]){
                        return true;
                    }
                }
            }
        }
        return false;
    }
}
```

4.4 复杂度

时间复杂度：O(n^2)，嵌套两层循环。

空间复杂度：O(n^2)，使用二维数组，且不能使用滚动数组进行优化。

05 总结

通过以上三道题目，我们大致了解了如何解决动态规划的问题，而难点在于如何准确的定义状态，列出正确的状态转移方程。从课堂内容和课下练习中，我总结出两个方法：

1.将问题化简，从最简单的情况开始考虑，从而定义状态，求得状态转移方程。

2.大胆尝试，我们在定义了状态之后有时很难找到状态方程，不应怕犯错，可以多次假设不同的状态定义方式，直至求出正确的转移方程。

正如文章一开头所说的那样，在理解了动态规划的思想之后，我们还要实际操作一番，形成自己的逻辑思维。以上只是我这个跨考小白学习动态规划的一点经验，不一定适合每个人，希望能给大家提供一些参考，如有错误还望大家指正。