One Step Offer 算法第五讲

Dynamic Programming 动态规划 1

动态规划Dynamic Programming

动态规划过程是：每次决策依赖于当前状态，又随即引起状态的转移一个决策序列就是在变化的状态中产生出 **查找多重叠子问题情况下的最优解**

动态规划和其它遍历算法（如**深/广度优先搜索**）都是将原问题拆成多个子问题然后求解，他们之间最本质的区别是，动态规划**保存子问题的解，避免重复计算**

解决动态规划问题的关键是找到**状态转移方程** 有些问题还会涉及到**状态记录**

动态规划的适用情况

1. 最优化原理：如果问题的最优解所包含的子问题的解也是最优的，就称该问题具有最优子结构，即满足最优化原理。
2. 无后效性：即某阶段状态一旦确定，就不受这个状态以后决策的影响。也就是说，某状态以后的过程不会影响以前的状态，只与当前状态有关。
3. 有重叠子问题：即子问题之间是不独立的，一个子问题在下一阶段决策中可能被多次使用到。（该性质并不是动态规划适用的必要条件，但是如果没有这条性质，动态规划算法同其他算法相比就不具备优势）

动态规划 的维度

一维动态规划：典型的有斐波那契数列，寻找等差数列 -- 一般为Easy/Medium难度

二维动态规划：通常是两个及以上维度的路径最优解问题，也是面试中最常见的问题-- 大多数为Medium / Hard难度的题

三维动态规划：通常是比较复杂的多维度问题 -- 大多数为Hard

动态规划的分析过程

(1) **划分阶段**：按照问题的时间或空间特征，把问题分为若干个阶段。在划分阶段时，注意划分后的阶段一定要是有序的或者是可排序的，否则问题就无法求解。

(2) **确定状态和状态变量**：将问题发展到各个阶段时所处于的各种客观情况用不同的状态表示出来。当然，状态的选择要满足无后效性。

(3) **确定决策并写出状态转移方程**：因为决策和状态转移有着天然的联系，状态转移就是根据上一阶段的状态和决策来导出本阶段的状态。所以如果确定了决策，状态转移方程也就可写出。但事实上常常是反过来做，根据相邻两个阶段的状态之间的关系来确定决策方法和状态转移方程。

(4) **寻找边界条件**：给出的状态转移方程是一个递推式，需要一个递推的终止条件或边界条件。

动态规划的解题步骤

（1）分析最优解的性质，并刻画其结构特征。

（2）递归的定义最优解。

（3）以自底向上或自顶向下的记忆化方式（备忘录法）计算出最优值

（4）根据计算最优值时得到的信息，构造问题的最优解

例题一：Climibing Stairs

给定 *n* 节台阶，每次可以走一步或走两步，求一共有多少种方式可以走完这些台阶。

Input: 3

Output: 3

解释：1+1+1, 1+2, 2+1

问题解析

典型的斐波那契问题：到达第i阶的方法数=第i-1阶方法数 + 第i-2阶方法数

```
int climbStairs(int n) {
    if (n <= 2) return n;
    vector<int> dp(n + 1, 1);
    for (int i = 2; i <= n; ++i) {
        dp[i] = dp[i-1] + dp[i-2];
    }
    return dp[n];
}
```

时间复杂度：O(n)
空间复杂度：O(n)

```
int climbStairs(int n) {
    if (n <= 2) return n;
    int pre2 = 1, pre1 = 2, cur;
    for (int i = 2; i < n; ++i) {
        cur = pre1 + pre2;
        pre2 = pre1;
        pre1 = cur;
    }
    return cur;
}
```

时间复杂度：O(n)
空间复杂度：O(1)

例题二: Arithmetic Slices

给定一个数组，求这个数组中连续且等差的子数组一共多少个？ 子数组的长度最少为3

Input: nums = [1,2,3,4]

Output: 3

[1,2,3] [2,3,4] [1,2,3,4]

解题思路

等差数列：num[i] - num[i-1] = num[i-1] - num[i-2]

dp 数组的定义通常为以 i 结尾的，而等差子数组可以在任意一个位置终结，因此此题在最后需要对 dp 数组求和。

问题解答

```
public int numberOfArithmeticSlices(int[] A) {
        int[] dp = new int[A.length];
        int sum = 0;
        for (int i = 2; i < dp.length; i++) {
            if (A[i] - A[i - 1] == A[i - 1] - A[i - 2]) {
                dp[i] = 1 + dp[i - 1];
                sum += dp[i];
            }
        }
        return sum;
    }
```

二维DP - Minimum Path Sum

给定一个 *m*×*n* 大小的非负整数矩阵，求从左上角开始到右下角结束的、经过的数字的和最小的路径。每次只能向右或者向下移动。

Input:
[[1,3,1],
[1,5,1],
[4,2,1]]
Output: 7

解释：最短路径为
1 -> 3 -> 1-> 1 -> 1
sum = 7

问题分析

我们可以定义一个同样是二维的 dp 数组，其中 dp[i][j] 表示从左上角开始到 (i, j) 位置的最优路径的数字和。因为每次只能向下或者向右移动，我们可以很容易得到状态转移方程 dp[i][j] =min(dp[i-1][j], dp[i][j-1]) + grid[i][j]，其中 grid 表示原数组。

时间复杂度：O(n^2)

空间复杂度：O(n^2)

答案 - C++

```
int minPathSum(vector<vector<int>>& grid) {
    int m = grid.size(), n = grid[0].size();
    vector<vector<int>> dp(m, vector<int>(n, 0));
    for (int i = 0; i < m; ++i) {
        for (int j = 0; j < n; ++j) {
            if (i == 0 && j == 0) {
                dp[i][j] = grid[i][j];
            } else if (i == 0) {
                dp[i][j] = dp[i][j-1] + grid[i][j];
            } else if (j == 0) {
                dp[i][j] = dp[i-1][j] + grid[i][j];
            } else {
                dp[i][j] = min(dp[i-1][j], dp[i][j-1]) + grid[i][j];
            }
        }
    }
    return dp[m-1][n-1];
}
```

答案 --Java

```java
public int minPathSum(int[][] grid) {
    int m = grid.length;
    int n = grid[0].length;
    int[][] dp = new int[m][n];

    for (int i = 0; i < m; i++) {
       for (int j = 0; j < n; j++) {
          if (i == 0 && j == 0) dp[i][j]  = grid[i][j];
          else if (i == 0) dp[i][j] = dp[i][j-1] + grid[i][j];
          else if (j == 0) dp[i][j] = dp[i-1][j] + grid[i][j];
          else dp[i][j] = Math.min(dp[i-1][j], dp[i][j-1]) + grid[i][j];
       }
    }
    return dp[m-1][n-1];
  }
```

答案 - python

```
def minPathSum(self, grid):
        r, c = len(grid), len(grid[0])
        dp = [grid[0][0] for _ in range(c)]
        for j in range(1, c):
            dp[j] = dp[j-1] + grid[0][j]
        for i in range(1, r):
            dp[0] += grid[i][0]
            for j in range(1, c):
                dp[j] = min(dp[j-1], dp[j]) + grid[i][j]
         return dp[-1]
```

方法二：单维数组

为了提高空间复杂度，我们可不可以使用单维数组？

因为 dp 矩阵的每一个值只和左边和上面的值相关，我们可以使用空间压缩将 dp 数组压缩为一维。对于第 i 行，在遍历到第 j 列的时候，因为第 j-1 列已经更新过了，所以 dp[j-1] 代表 dp[i][j-1]的值；而 dp[j] 待更新，当前存储的值是在第 i-1 行的时候计算的，所以代表 dp[i-1][j] 的值。

C++ 代码实现

```
int minPathSum(vector<vector<int>>& grid) {
    int m = grid.size(), n = grid[0].size();
    vector<int> dp(n, 0);
    for (int i = 0; i < m; ++i) {
        for (int j = 0; j < n; ++j) {
            if (i == 0 && j == 0) {
                dp[j] = grid[i][j];
            } else if (i == 0) {
                dp[j] = dp[j-1] + grid[i][j];
            } else if (j == 0) {
                dp[j] = dp[j] + grid[i][j];
            } else {
                dp[j] = min(dp[j], dp[j-1]) + grid[i][j];
            }
        }
    }
    return dp[n-1];
}
```

例题二 01 matrix

给定一个由 0 和 1 组成的二维矩阵，求每个位置到最近的 0 的距离。

Input:	Output:
[[0,0,0],	[[0,0,0],
[0,1,0],	[0,1,0],
[1,1,1]]	[1,2,1]]

题目分析

一般来说，因为这道题涉及到四个方向上的最近搜索，所以很多人的第一反应可能会是广度优先搜索。但是对于一个大小 O(mn) 的二维数组，对每个位置进行四向搜索，最坏情况的时间复杂度（即全是 1）会达到恐怖的 O(m2n2)。一种办法是使用一个 dp **数组做 memoization**，使得广度优先搜索不会重复遍历相同位置；另一种更简单的方法是，我们从左上到右下进行一次动态搜索，再从右下到左上进行一次动态搜索。两次动态搜索即可完成四个方向上的查找。

C++ 代码实现

```
vector<vector<int>> updateMatrix(vector<vector<int>>& matrix) {
    if (matrix.empty()) return {};
    int n = matrix.size(), m = matrix[0].size();
    vector<vector<int>> dp(n, vector<int>(m, INT_MAX - 1));

    // 从左上到右下做 dp
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < m; ++j) {
            if (matrix[i][j] == 0) {
                dp[i][j] = 0;
            } else {
                if (j > 0) dp[i][j] = min(dp[i][j], dp[i][j-1] + 1);
                if (i > 0) dp[i][j] = min(dp[i][j], dp[i-1][j] + 1);
            }
        }
    }
```

C++ 实现continued

```
// 从右下到左上遍历
for (int i = n - 1; i >= 0; --i) {
    for (int j = m - 1; j >= 0; --j) {
        if (matrix[i][j] != 0) {
            if (j < m - 1) dp[i][j] = min(dp[i][j], dp[i][j+1] + 1);
            if (i < n - 1) dp[i][j] = min(dp[i][j], dp[i+1][j] + 1);
        }
    }
 }
 return dp;
}
```