动态规划理论基础

1、概述

动态规划英文名称位Dynamic Programming，简称DP。如果某一问题有很多重叠子问题，那么使用动态规划师最有效的解决办法。

动态规划不同于其他的算法，它的重点在于“动态”两个字，其核心的含义是指每一个状态都是前序状态的计算推导而来。

2、基本解题步骤

动态规划的核心是状态转移方程，很多初学者的在学习的时候，往往是去背诵一些模板似的转移方程，例如01背包、完全背包，看起来似乎是学会了，但是题目稍加变形，完全就摸不到头脑，这是因为你并没有掌握真正的方法。

下面，就让我们一起来严肃的研究一下动态规划题目的基本解题步骤。

简单来说，解决一道动态规划问题，我们需要严格的按照以下五步进行：

1、确定dp数组及其下标的含义。

2、推导状态转移方程。

3、初始化DP数组。

4、确定遍历顺序。

5、举例推导DP数组。

在后续的例题中，我们都将严格按照以上五个步骤，来拆解习题进行深入学习。

3、问题排查

在写出了一个动态规划之后，我们该如何进行排查，也是一件非常重要的事情。在这里我们也将这个步骤流程化。

1、举例推导递推公式，即将我们的递推公式带入实际的数据中进行验证

2、打印dp数组的日志，观察dp数组的数据与预期是否一致。

3、反复核验dp数组的初始值与起止点，确保每个数值都是准确并有意义的。

下面，我们将以一道非常简单的真题为基础，正式开始学习动态规划。

4、真题讲解

力扣509：斐波那契数列

题目描述

斐波那契数，通常用 F(n) 表示，形成的序列称为 斐波那契数列 。该数列由 0 和 1 开始，后面的每一项数字都是前面两项数字的和。也就是：

F(0) = 0，F(1) = 1
F(n) = F(n - 1) + F(n - 2)，其中 n > 1

给你 n ，请计算 F(n) 。

示例 1：

输入：2
输出：1
解释：F(2) = F(1) + F(0) = 1 + 0 = 1

示例 2：

输入：3
输出：2
解释：F(3) = F(2) + F(1) = 1 + 1 = 2

解析

这是一道非常简单的入门练习题，可以用来练习非常多的基本功，例如递归、递推等等。今天，我们将通过这道简单的题目，来学习动态规划。

首先，根据前文的基本解题步骤，我们逐一完成。

1、确定dp数组及其下标的含义。
dp[i]：第i个数的斐波那契数的值

2、推导状态转移方程。
dp[i] = dp[i-1] + dp[i-2]，即第i个数，为前两个数字之和。

3、初始化DP数组。
dp[0] = 0;
dp[1] = 1;

4、确定遍历顺序。
根据状态转移方程，我们很容易发现，后面的数字是依赖前面的数字的。因此，我们应该从前向后进行遍历。

5、举例推导DP数组。
根据转移方程，我们推导前10个数字应该如下：
0 1 1 2 3 5 8 13 21 34 55

如果我们的代码没有通过，那么，我们就应该打印一下我们的dp数组，观察实际的数据是否与上面的预期一致。

编码

方案1

```
#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    //初始化数组前两位
    int dp[2] = {0, 1};

    int fib(int n) {
        //边界条件
        if (n <= 1) {
            return n;
        }
        //从前向后遍历
        for (int i = 2; i <= n; ++i) {
            //后一个数字等于前两个数字之和
            int sum = dp[0] + dp[1];
            //计算出的结果向前偏移
            dp[0] = dp[1];
            dp[1] = sum;
        }
        return dp[1];
    }
};


int main() {
    Solution s{};
    int n;
    cin >> n;
    cout << s.fib(n);
    return 0;
}
```

通过进一步的分析，我们发现实际上只需要使用数组中两个数字即可计算出全部的结果，所以，我们可以对这段程序进行优化。

优化后的代码如下：

方案2

```
class Solution {
public:
    //初始化数组前两位
    int dp[2] = {0, 1};

    int fib(int n) {
        //边界条件
```

```
        if (n <= 1) {
            return n;
        }
        //从前向后遍历
        for (int i = 2; i <= n; ++i) {
            //后一个数字等于前两个数字之和
            int sum = dp[0] + dp[1];
            //计算出的结果向前偏移
            dp[0] = dp[1];
            dp[1] = sum;
        }
        return dp[1];
    }
};
```