动态规划是什么?

动态规划（英語：Dynamic programming，简称DP）是一种在数学、管理科学、计算机科学、经济学和生物信息学中使用的，通过把原问题分解为相对简单的子问题的方式求解复杂问题的方法。动态规划常常适用于有重叠子问题[1]和最优子结构性质的问题，动态规划方法所耗时间往往远少于朴素解法。动态规划背后的基本思想非常简单。大致上，若要解一个给定问题，我们需要解其不同部分（即子问题），再根据子问题的解以得出原问题的解。通常许多子问题非常相似，为此动态规划法试图仅仅解决每个子问题一次，从而减少计算量：一旦某个给定子问题的解已经算出，则将其记忆化存储，以便下次需要同一个子问题解之时直接查表。这种做法在重复子问题的数目关于输入的规模呈指數增長时特别有用。（维基百科上对于动态规划的定义）

我的简单理解就是问题的求解可以分成很多子问题，然后将每个子问题的最优解存储起来，下次需要该子问题的解的时候就直接查询调用，避免了很多重复的计算。

动态规划的要素

1、递归+记忆化（将子结果进行存储）==》递推 ：从题目中获取递推公式，并且递推的顺序应该是从下往上的

递推公式为F(n)=F(n-1)+F(n-2)

```
temp[0]=1;
temp[1]=2;
for(int i=2;i<n;i++){
temp[i]=temp[i-1]+temp[i-2];
}
```

2、状态的定义：opt(n),dp[n],fib(n);

3、状态转移方程：opt(n)=best_of(opt(n-1),opt(n-2));

4、最优子结构：子程序A，B的最优解的结合就可以作为向上递推到上一个子问题的解，而不需要关注A,B之前的解；

动态规划的基础篇：https://blog.csdn.net/weixin_44625138/article/details/100188659

实战分析

一、leetcode题目: 爬楼梯

假设你正在爬楼梯。需要 n 阶你才能到达楼顶。

每次你可以爬 1 或 2 个台阶。你有多少种不同的方法可以爬到楼顶呢？

注意：给定 n 是一个正整数。

在此处给出两种解法：

1、解法一：将子问题的解用数组存储起来：(比较简单的动态规划法)

```
public static int climbStairs(int n){
if(n<=2) return n; //先判断传入的参数
int[] temp=new int[n]; //开辟一个数组用来存储值计算过的值
```

temp[0]=1;

temp[1]=2;

for(int i=2;i<n;i++){

temp[i]=temp[i-1]+temp[i-2];

}

return temp[n-1];

}

2、解法二：因为只是需要前两个的值，因此此处直接引用三个变量来进行记录，用来节省空间，而不用另外开辟数组：

public static int climbStairs2(int n){

if(n<=2) return n;

int temp=0;

int temp1=1;

int temp2=2;

for(int i=2;i<n;i++){

temp=temp1+temp2;

temp2=temp1;

temp1=temp;

```
    }
    return temp;

}

```

二、leetcode题目: 三角形最小路径和

给定一个三角形，找出自顶向下的最小路径和。每一步只能移动到下一行中相邻的结点上。

例如，给定三角形：

[

[2],

[3,4],

[6,5,7],

[4,1,8,3]

]

自顶向下的最小路径和为 11（即，2 + 3 + 5 + 1 = 11）

解法一：使用 递归+保存记忆+分治 的解法（此方法在LeetCode上会超时，当三角形的层数比较深的时候会超时）

private static Integer[][] memo;

public static int minimumTotal(List<List<Integer>> triangle) {

```
    int level=triangle.size();
    memo=new Integer[level][level];

    return getmin(0,0,triangle,level);

}

public static int getmin(int row,int y,List<List<Integer>> triangle,int level){
    if(memo[row][y]!=null) {
                return memo[row][y];
     }
     if(row==level-1){
            return memo[row][y]=triangle.get(row).get(y);
     }

    //此处使用到了分治
    int left=getmin(row+1,y,triangle,level);
    int right=getmin(row+1,y+1,triangle,level);
     //下一层的最小值加上本层的值
    return memo[row][y]= Math.min(left,right)+triangle.get(row).get(y);
}

```

解法二：动态规划：

public static int GetMinDp(List<List<Integer>> triangle) {

int row=triangle.size();

```
 int[] temp=new int[row+1];
 for(int i=triangle.size()-1;i>=0;i--) {
         for(int j=0;j<triangle.get(i).size();j++) {
                 temp[j]=triangle.get(i).get(j)+Math.min(temp[j], temp[j+1]);
                 System.out.println(temp[j]);
         }

 }

```

```
10    return temp[0];
11
```

}

解法分析：

1、确定状态方程为temp[j];

2、确定状态转移方程为：temp[j]=triangle.get(i).get(j)+Math.min(temp[j], temp[j+1]);

triangle.get(i).get(j)相当于（i,j）的值 temp[j]相当于i+1层的第j个数的数值，temp[j+1]相当于i+1的第j+1个数的数值, 由于只能向下或者向后移动，因此对于triangle.get(i).get(j)来说，下一层的值对应的列为j或者j+1；由于上一层的值只需要用到下一层的值，因此tem[j]的值可以被覆盖，从下往上一直递推。到了最顶层，只有一列，j此时为0，因此最后的值为 temp[0]

三、leetcode题目：编辑距离

给定两个单词 word1 和 word2，计算出将 word1 转换成 word2 所使用的最少操作数 。

你可以对一个单词进行如下三种操作：

插入一个字符

删除一个字符

替换一个字符

入: word1 = "horse", word2 = "ros"

输出: 3

解释:

horse -> rorse (将 'h' 替换为 'r')

rorse -> rose (删除 'r')

rose -> ros (删除 'e')

动态规划的题解如下：

class Solution {

public int minDistance(String word1, String word2) {

//先进行参数的判断

if(word1==null || word2==null|| word1.equals(word2)) return 0;

```
1      int m=word1.length();
2      int n=word2.length();
3      if(m*n==0) return m+n;
4
5      //确定状态方程，dp[i][j] 表示 word1 的前 i 个字母和 word2 的前 j 个字母之间的编辑距离
6      int[][] dp=new int[m+1][n+1];
7
8      //处理边界,一个空串和一个非空串的编辑距离为 dp[i][0] = i 和 dp[0][j] = j
9      for(int i=0;i<m+1;i++){
10         dp[i][0]=i;
```

```
    }

     for(int i=0;i<n+1;i++){
        dp[0][i]=i;
    }

    //开始进行双层循环遍历；

    for(int i=1;i<m+1;i++){
        for(int j=1;j<n+1;j++){
            int insert=dp[i-1][j]+1;     //如果是插入的操作
            int delete=dp[i][j-1]+1;     //如果是删除操作
            int replace=dp[i-1][j-1];    //如果是替换操作
            if (word1.charAt(i - 1) != word2.charAt(j - 1))  //不相等的话表示对于i-1处的字符需要进行操作
                replace += 1;
             dp[i][j] = Math.min(insert, Math.min(delete, replace));    //最后获取最小值
        }
    }
    return dp[m][n];


}

}
```

四、Leetcode题目：零钱替换

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。

示例 1:

输入: coins = [1, 2, 5], amount = 11

输出: 3

解释: 11 = 5 + 5 + 1

解法：动态递归，自下向上

步骤：

1、确定状态方程：dp[i] (金额为i时的硬币数量的最小值)

2、确定状态转移方程为：dp[i]=min(dp[i-coins[j]])+1 ,coins[j]表示最后一枚硬币的面值，dp[i-coins[j]]则表示除去最后一枚硬币的最小的数量；

3、初始化dp[0]=0，并把其他的值赋值为max;

复杂度分析

时间复杂度：O(Sn)O(Sn)。在每个步骤中，算法在 nn 个硬币中查找计算下一个 F(i)F(i)，其中 1\leq i\leq s1≤i≤s。因此，总的来说，迭代次数是 SnSn。

空间复杂度：O(S)O(S)，dp 使用的空间。

class Solution {

public int coinChange(int[] coins, int amount) {

if(coins==null || coins.length==0 || amount<0) return -1;

if(amount==0) return 0;

int max=amount+1;

```
     //dp表示金额为s时的数量最小值

    int[] dp=new int[max];

     //数组中填充数值，其实就是初始化；

     Arrays.fill(dp, max);
    //边界值,当金额为0时，数量值为0；
    dp[0]=0;
    for(int i=1;i<=amount;i++){
        for(int j=0;j<coins.length;j++){
            //此处一定要进行判断，否则数组越界
            if(coins[j]<=i){
            dp[i]=Math.min(dp[i],dp[i-coins[j]]+1);
                }
        }
    }

```

//dp[amount] > amount 表明最后值无法被更新，也就是找不到组合的值

```
    return dp[amount] > amount ? -1 :dp[amount];

}
```

}

原文链接：https://blog.csdn.net/weixin_44625138/article/details/100529596