动态规划 Dynamic Programming

Notebook:	程序设计与算法		
Created:	02/10/2020 11:36 pm	**Updated:**	11/10/2020 6:56 pm
Author:	Jie Zhong		

递归到动规的一般转化方法
递归函数有n个参数，就定义一个n维数组，数组的下标是递归函数参数的取值范围，数组元素的值是递归函数的返回值。这样就可以从边界开始逐步填充数组，相当于计算递归函数值的逆过程。

动规解题的一般思路

1. 将原问题分解为子问题
2. 确定状态
    - 在用动态规划解题时，往往将和子问题相关的各个变量的一组取值称之为一个"状态"。
    - 一个"状态"对应于一个或多个子问题，某个状态下的值，就是这个"状态"所对应的子问题的解。
    - 所有"状态"的集合，构成问题的"状态空间"。整个问题的时间复杂度 = 状态数目 * 计算每个状态所需的时间
3. 确定一些初始状态（边界状态）的值
4. 确定状态转移方程
    - 定义出什么是"状态"，以及在该"状态"下的"值"之后，就要找出不同的状态之间如何迁移，求出另一个状态的值。状态迁移可以用递推公式表示，此递推公式也可被称作为"状态转移方程"

能用动规解决的问题的特点

1. 问题具有最优子结构性质。如果问题的最优解包含的子问题的解也是最优的，我们就称该问题具有最优子结构性质
2. 无后效性。当前的若干个状态值一旦确定，则此后过程的演变就只和这若干个状态的值有关，和之前是采用哪种手段或经过哪条路径演变到当前的状态没有关系。

递归

- 优点：直观；
- 缺点：
    - 可能因为递归层数太深导致爆栈，函数调用带来额外时间开销。
    - 可能包含重复计算，增加时间复杂度

动规（递推）

- 效率高，有可能使用滚动数组节省空间

*** 参考：***
icourse163，程序设计与算法（二）算法基础，第六周 动态规划（一）

实战题：
Leetcode - 买卖股票系列问题

- 题解

    - https://leetcode-cn.com/problems/best-time-to-buy-and-sell-stock-iii/solution/yi-ge-tong-yong-fang-fa-tuan-mie-6-dao-gu-piao-wen/
- 国际站题解
    - https://leetcode.com/problems/best-time-to-buy-and-sell-stock-iv/discuss/54117/Clean-Java-DP-solution-with-comment
    - https://leetcode.com/problems/best-time-to-buy-and-sell-stock-iv/discuss/54114/Easy-understanding-and-can-be-easily-modified-to-different-situations-Java-Solution

Leetcode - 32. 最长有效括号

- 动规
    - dp[i]是以第i个字符结尾的最长的有效括号
    - 要组成有效括号，最后一个字符必须是')'
        - 判断i前面一个字符：
            1. '(' - dp[i] = (i >= 2 ? dp[i - 2] : 0) + 2
            2. ')' - 如果和i-1对应的位置是'(', 则 dp[i] = dp[i - 1] + 2 + (i - dp[i - 1] >= 2 ? dp[i - dp[i - 1] - 2]: 0)
- 栈
    - 题解：
        1. 始终保持栈底元素为当前已经遍历的元素中"最后一个没有被匹配的右括号的下标"
        2. 栈里其他元素维护左括号的下标

Leetcode - 72. 编辑距离

```
package com.nowcoder.jie;
import java.util.Scanner;

public class Huawei52a {

    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        while (in.hasNext()) {
            String strA = in.next();
            String strB = in.next();
            int ic = 1;
            int dc = 1;
            int rc = 1;
            int cost = strEditCost(strA, strB, ic, dc, rc);
            System.out.println(cost);
        }
        in.close();
    }

    public static int strEditCost(String strA, String strB, int ic, int dc,
int rc) {
        /*
         * 字符串之间的距离，编辑距离，将strA编辑成strB所需的最小代价 编辑操作包括
插入一个字符、删除一个字符、替换一个字符
         * 分别对应的代价是ic、dc、rc，insert cost、delete cost、replace cost
         * strA[x-1]代表strA的第x个字符，注意下标是从0开始的,strB[y-1]代表strB的
第y个字符 定义一个代价矩阵为(N+1)*(M+1)，
         * M, N 表示strA strB的长度 dp[x][y]表示strA的前x个字符串编辑成 strB的前
y个字符所花费的代价
         * dp[x][y]是下面几种值的最小值： 1、dp[x][y] = dp[x-1][y] + dc
         * dp[x-1][y]将strA的前x-1个字符编辑成strB的前y个字符的代价已知，
         * 那么将将strA的前x个字符编辑成strB的前y个字符的代价dp[x][y]就是dp[x-1]
[y] + dc
         * 相当于strA的前x-1个字符编辑成strB的前y个字符，现在变成了strA的前x个字
符，增加了一个字符，要加上删除代价
         * 2、dp[x][y] = dp[x][y-1] + ic dp[x][y-1]将strA的前x个字符编辑成strB
的前y-1个字符的代价已知，
         * 现在变为strB的前y个字符，相应的在strA前x个操作代价的基础上插入一个字符
3、dp[x][y] = dp[x-1][y-1]
         * dp[x-1][y-1]将strA的前x-1个字符编辑成strB的前y-1个字符的代价已知，
         * strA的第x个字符和strB的第y个字符相同，strA[x-1] == strB[y-1]，没有引
入操作 4、dp[x][y] =
```

```
         * dp[x-1][y-1] + rc strA的第x个字符和strB的第y个字符不相同，strA[x-1]
! = strB[y-1],
         * 在strA的前x-1个字符编辑成strB的前y-1个字符的代价已知的情况下,
         * 计算在strA的前x字符编辑成strB的前y个字符的代价需要加上替换一个字符的代
价
         */
        int m = strA.length();
        int n = strB.length();
        int[][] dp = new int[m + 1][n + 1];
        for (int i = 1; i <= n; i++)
            dp[0][i] = i * ic;
        for (int i = 1; i <= m; i++)
            dp[i][0] = i * dc;
        for (int x = 1; x <= m; x++) {
            for (int y = 1; y <= n; y++) {
                int cost1 = dp[x - 1][y] + dc;
                int cost2 = dp[x][y - 1] + ic;
                int cost3 = 0;
                if (strA.charAt(x - 1) == strB.charAt(y - 1))
                    cost3 = dp[x - 1][y - 1];
                else
                    cost3 = dp[x - 1][y - 1] + rc;
                dp[x][y] = Math.min(cost1, cost2);
                dp[x][y] = Math.min(dp[x][y], cost3);
            }
        }
        return dp[m][n];
    }
}
```