

状态转移方程

动态规划中当前的状态往往依赖于前一阶段的状态和前一阶段的决策结果。例如我们知道了第 i 个阶段的状态 S_i 以及决策 U_i ，那么第 $i+1$ 阶段的状态 S_{i+1} 也就确定了。所以解决动态规划问题的关键就是确定状态转移方程，一旦状态转移方程确定了，那么我们就可以根据方程式进行编码。

在前面的文章《动态规划-开篇》讲到了如何设计一个动态规划算法，有以下四个步骤：

- 1、刻画一个最优解的结构特征。
- 2、递归地定义最优解的值。
- 3、计算最优解的值，通常采用自底向上的方法。
- 4、利用计算出的信息构造一个最优解。

对于确定状态转移方程就在第一步和第二步中，首先要确定问题的决策对象，接着对决策对象划分阶段并确定各个阶段的状态变量，最后建立各阶段的状态变量的转移方程。

例如用 $dp[i]$ 表示以序列中第 i 个数字结尾的最长递增子序列长度和最长公共子序列中用 $dp[i][j]$ 表示的两个字符串中前 i 、 j 个字符的最长公共子序列，我们就是通过这两个数字量的不断求解最终得到答案的。这个数字量就被我们称为状态。状态是描述问题当前状况的一个数字量。首先，它是数字的，是可以被抽象出来保存在内存中的。其次，它可以完全的表示一个状态的特征，而不需要其他任何的辅助信息。最后，也是状态最重要的特点，状态间的转移完全依赖于各个状态本身，如最长递增子序列中， $dp[x]$ 的值由 $dp[i](i < x)$ 的值确定。若我们在分析动态规划问题的时候能够找到这样一个符合以上所有条件的状态，那么多半这个问题是可以被正确解出的。所以说，解动态规划问题的关键，就是寻找一个好的状态。

总结

下面对这几天的学习总结一下，将我遇到的各种模型的状态转移方程汇总如下：

1、最长公共子串

假设两个字符串为 $str1$ 和 $str2$ ，它们的长度分别为 n 和 m 。 $dp[i][j]$ 表示 $str1$ 中前 i 个字符与 $str2$ 中前 j 个字符分别组成的两个前缀字符串的最长公共长度。这样就把长度为 n 的 $str1$ 和长度为 m 的 $str2$ 划分成长度为 i 和长度为 j 的子问题进行求解。状态转移方程如下：

1. $dp[0][j] = 0; (0 \leq j \leq m)$
2. $dp[i][0] = 0; (0 \leq i \leq n)$
3. $dp[i][j] = dp[i-1][j-1] + 1; (str1[i] == str2[j])$
4. $dp[i][j] = 0; (str1[i] != str2[j])$

因为最长公共子串要求必须在原串中是连续的，所以一旦某处出现不匹配的情况，此处的值就重置为0。

详细代码请看最长公共子串。

2、最长公共子序列

区分一下，最长公共子序列不同于最长公共子串，序列是保持子序列字符串的下标在 $str1$ 和 $str2$ 中的下标顺序是递增的，该字符串在原串中并不一定是连续的。同样的我们可以假设 $dp[i][j]$ 表示为字符串 $str1$ 的前 i 个字符和字符串 $str2$ 的前 j 个字符的最长公共子序列的长度。状态转移方程如下：

1. $dp[0][j] = 0; (0 \leq j \leq m)$
2. $dp[i][0] = 0; (0 \leq i \leq n)$
3. $dp[i][j] = dp[i-1][j-1] + 1; (str1[i-1] == str2[j-1])$
4. $dp[i][j] = \max\{dp[i][j-1], dp[i-1][j]\}; (str1[i-1] != str2[j-1])$

详细代码请看最长公共子序列。

3、最长递增子序列（最长递减子序列）

因为两者的思路都是一样的，所以只给出最长递增子序列的状态转移方程。假设有序列 $\{a_1, a_2, \dots, a_n\}$ ，我们求其最长递增子序列长度。按照递推求解的思想，我们用 $F[i]$ 代表若递增子序列以 a_i 结束时它的最长长度。当 i 较小，我们容易直接得出其值，如 $F[1] = 1$ 。那么，如何由已经求得的 $F[i]$ 值推得后面的值呢？假设， $F[1]$ 到 $F[x-1]$ 的值都已经确定，注意到，以 a_x 结尾的递增子序列，除了长度为1的情况，其它情况中， a_x 都是紧跟在一个由 $a_i(i < x)$ 组成递增子序列之后。要求以 a_x 结尾的最长递增子序列长度，我们依次比较 a_x 与其之前所有的 $a_i(i < x)$ ，若 a_i 小于 a_x ，则说明 a_x 可以跟在以 a_i 结尾的递增子序列之后，形成一个新的递增子序列。又因为以 a_i 结尾的递增子序列最长长度已经求得，那么在这种情况下，由

以 a_i 结尾的最长递增子序列再加上 a_x 得到的新的序列，其长度也可以确定，取所有这些长度的最大值，我们即能得到 $F[x]$ 的值。特殊的，当没有 $a_i (i < x)$ 小于 a_x ，那么以 a_x 结尾的递增子序列最长长度为1。即 $F[x] = \max\{1, F[i] + 1 | a_i < a_x \text{ \&\& } i < x\}$ 。

详细代码请看最长递增子序列。

4、最大子序列和的问题

假设有序列 $\{a_1, a_2, \dots, a_n\}$ ，求子序列的和最大问题，我们用 $dp[i]$ 表示以 a_i 结尾的子序列的最大和。

$dp[1] = a_1; (a_1 \geq 0 \text{ \&\& } i == 1)$

$dp[i] = dp[i-1] + a_i; (a_i \geq 0 \text{ \&\& } i \geq 2)$


$dp[i] = 0; (dp[i-1] + a_i \leq 0 \text{ \&\& } i \geq 2)$

详细代码请看最大子序列的和。

5、数塔问题（动态搜索）

给定一个数组 $data[n][m]$ 构成一个数塔求从最上面走到最低端经过的路径和最大。可以假设 $dp[i][j]$ 表示走到第 i 行第 j 列位置处的最大值，那么可以推出状态转移方程：

$dp[i][j] = \max\{dp[i-1][j-1], dp[i-1][j]\} + data[i][j];$

 [View Code](#)



6、（01）背包问题


这是一个经典的动态规划问题，另外在贪心算法里也有背包问题，至于二者的区别在此就不做介绍了。

假设有 N 件物品和一个容量为 V 的背包。第 i 件物品的体积是 $v[i]$ ，价值是 $c[i]$ ，将哪些物品装入背包可使价值总和最大？

每一种物品都有两种可能即放入背包或者不放入背包。可以用 $dp[i][j]$ 表示第 i 件物品放入容量为 j 的背包所得的最大价值，则状态转移方程可以推出如下：

$dp[i][j] = \max\{dp[i-1][j-v[i]] + c[i], dp[i-1][j]\};$

```
  
  
for (int i = 1; i <= N; i++) //枚举物品  
{  
    for (int j = 0; j <= V; j++) //枚举背包容量  
    {  
        f[i][j] = f[i-1][j];  
        if (j >= v[i])  
        {  
            f[i][j] = Max(f[i-1][j], f[i-1][j-v[i]] + c[i]);  
        }  
    }  
}
```



可以参照动态规划 - 0-1背包问题的算法优化、动态规划-完全背包问题、动态规划-多重背包问题

7、矩阵连乘（矩阵链问题）-参考《算法导论》

例如矩阵链 $\langle A_1, A_2, A_3 \rangle$ ，它们的维数分别为 $10 \times 100, 100 \times 5, 5 \times 50$ ，那么如果顺序相乘即 $((A_1 A_2) A_3)$ ，共需 $10 \times 100 \times 5 + 10 \times 5 \times 50 = 7500$ 次乘法，如果按照 $(A_1 (A_2 A_3))$ 顺序相乘，却需做 $100 \times 5 \times 50 + 10 \times 100 \times 50 = 75000$ 次乘法。两者之间相差了10倍，所以说矩阵链的相乘顺序也决定了计算量的大小。

我们用利用动态规划的方式 $dp[i][j]$ 表示第 i 个矩阵至第 j 个矩阵这段的最优解，还有对于两个矩阵 $A(i, j) * B(j, k)$ 则需要 $i * j * k$ 次乘法，推出状态转移方程：

$dp[i][j] = 0; (i == j, \text{表示只有一个矩阵，计算次数为} 0)$

$dp[i][j] = \min\{dp[i][k] + dp[k+1][j] + p[i-1]*p[k]*p[j]\}; (i < j \ \&\& \ i \leq k < j)$

$dp[1][n]$ 即为最终求解.

```

#define MAXSIZE 100

int dp[MAXSIZE][MAXSIZE]; //存储最小的计算次数
int s[MAXSIZE][MAXSIZE]; //存储断点, 用在输出上面

int i, j, tmp;

for (int l = 2; l <= n; l++) { //j-i的长度, 由于长度为1是相同的矩阵那么为0不用计算
    for (i = 1; i <= n - l + 1; i++) { //由于j-i = l - 1, 那么j的最大值为n, 所以i上限为 n - l + 1;
        j = i + l - 1; //由于j-i = l - 1, 那么j = l + i - 1
        dp[i][j] = dp[i + 1][j] + r[i] * c[i] * c[j]; //初始化, 就是k = i;
        s[i][j] = i;
        for (k = i + 1; k < j; k++) { //循环枚举k i < k < j
            tmp = dp[i][k] + dp[k + 1][j] + r[i] * c[k] * c[j];
            if (dp[i][j] > tmp) {
                dp[i][j] = tmp; //更新为最小值
                s[i][j] = k;
            }
        }
    }
}

//递归调用输出
void output(int i, int j) {
    if (i == j) {
        printf("A%d", i); //当两个相等的时候就不用继续递归就输出A
        return; //返回上一层
    }

    else {
        printf("(");
        output(i, s[i][j]);
        printf(" x ");
        output(s[i][j] + 1, j);
        printf(")");
    }
}
```