编辑距离

概念：字符串的编辑距离，edit distance，是指把 字符串 A 转成 字符串 B 所需要的最少操作次数，这里的字串操作包括：

1） Insert a character
2） Delete a character
3） Replace a character

一、问题分析：假设需要计算字符串 A 的字符串 B 的编辑距离，它们的长度分别为 length_A、length_B。则 edit 的距离我们考虑是 由字符串 A 转成 字符串 B。我们把转换这个过程量化称 C[length_A , length_B]，表示由长度为 length_A 的字符串 A 转换称 长度为 length_B 的字符串 B 所需要的操作次数，会有两种情况。

a） 首先考虑两个字符串的最后一个字符：（**实例情况**）
编辑距离会有以下几种情况：

A） 当两个字符串的**最后一个字符相等时**，则我们只需要计算
C[length_A – 1 , length_B -1]

B） 当两个字符串的**最后一个字符不相等时**，则我们可以将字符串 A 的最后一个字符转换成字符串 B 的最后一个字符，那么这个时候，我们计算的状态就变成了下式：
C[length_A – 1 , length_B -1] + 1

(其中 1 是指最后一个字符的转换操作)

C） 当两个字符串的**最后一个字符不相等时**，则我们可以将字符串 B 的最后一个字符插入到字符串 A 的最后一个位置，那么这个时候，我们计算的状态就变成了下式：
C[length_A , length_B -1] + 1

（其中 1 是指对 A 最后一个字符的插入操作，它们的长度差距是因为 B 的最后一个字符插入到 A 的最后一个位置，导致 A 的长度加 1，这个时候，A 的长度变成了 length_A + 1 ，但是这个时候的 A 和 B 的最后一个字符是一样的，所以，我们只要在插入操作过后，继续对它们去除最后一个字符后的字符串计算编辑距离即可）

D） 当两个字符串的**最后一个字符不相等时**，则我们可以将字符串 A 的最后一个字符删掉，那么这个时候，我们计算的状态就变成了下式：
C[length_A - 1 , length_B] + 1

（其中 1 是指对 A 最后一个字符的删除操作，它们的长度差距是因为 A 的最后一个字符已经删掉，导致 A 的长度减 1，这个时候，A 的长度变成了 length_A - 1 ，所以，我们只要在删除操作过后，继续对它们去除最后一个字符后的字符串计算编辑距离即可）

我们其实可以观察， A） B）两种情况在更新以后的计算形式是一致的，所以我们是否可以通过它们之间的不同来对这两种情况进行归纳成一种情况呢，如果可以的话，综合完的新的情况（A、B 结合）、C、D 的最小值不就是我们要计算的了吗？

b) Next，考虑 A 的第 i 个字符，B 的第 j 个字符：（**一般情况**）

我们这个时候，默认 A 的前 i-1 个字符已经和 B 的前 j-1 个字符相同了。我们根据 a) 中的情况推广到一般情况，则类似于 a）中的情况分析，有以下情况：

A) 当 A 的第 i 个字符和 B 的第 j 个字符相等时，那就是 A[i] = B[j]，那么我们的计算方式就会变成：

C[A[i+1 … length_A] , B[j+1…length_B]]

B) 当 A 的第 i 个字符和 B 的第 j 个字符不相等时，那就是 A[i] != B[j]，那么我们可以将 A[i]替换成 B[j]（替换操作），操作后的计算就会变成：

C[A[i+1 … length_A] , B[j+1…length_B]] + 1

(1 为替换操作)

C) 当 A 的第 i 个字符和 B 的第 j 个字符不相等时，那就是 A[i] != B[j]，那么我们可以将 A[i]删掉（删除操作），操作后的计算就会变成：

C[A[i+1 … length_A] , B[j …length_B]] + 1

（1 为删除操作）

D) 当 A 的第 i 个字符和 B 的第 j 个字符不相等时，那就是 A[i] != B[j]，那么我们可以将 B[j]插入到 A 的第 i 个位置，使得 A[i]=B[j]（插入操作），操作后的计算就会变成：

C[A[i … length_A] , B[j+1…length_B]] + 1

（1 为插入操作）

问题分析的总结：

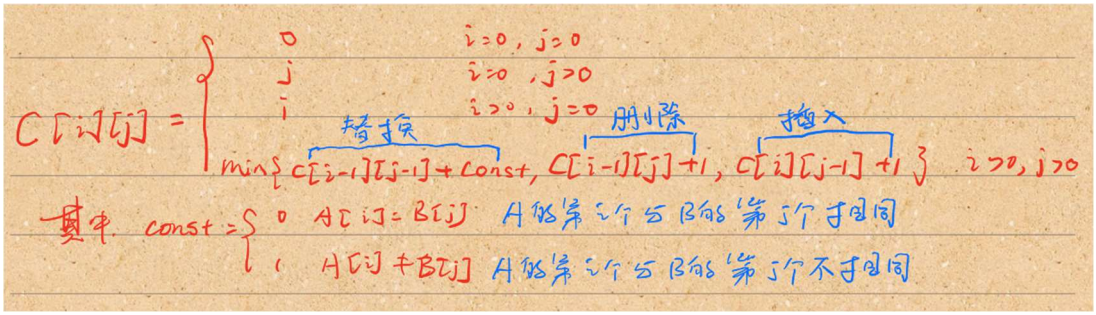
1、 我们分析动态规划的问题，往往希望能够找到当前问题的子问题，找到子问题的形式，从而从子问题中看是否会有最优子结构。当我们找到这个问题具有最优子结构后，我们就可以通过这个最优子结构写出我们问题的递归形式，当递归形式写出后，我们就可以从递归形式中找到动态规划的解决方法，因为往往递归形式就是我们的动态规划转移方程。

2、 我们分析问题的思路，大脑往往是比较喜欢具体的形式，那就是从一些具体的实例出发，弄懂问题的形式，然后将问题的形式推向一般化。上述的问题分析已经充分展示出这一思考路线。

二、建立动态规划方程

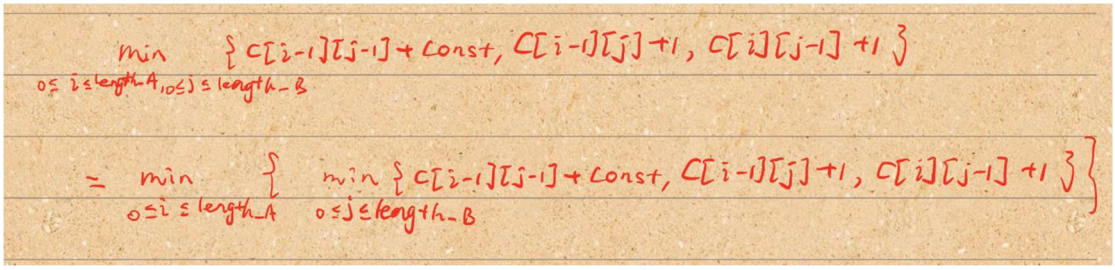
为了统一形式，我们沿用上面的做法，我们用 C[i][j]表示字符串 A 和字符串 B 的编辑距离，它表示 A 字符串从第 0 个字符到第 i 个字符与 B 字符串从第 0 个字符到第 j 个字符的编辑距离。其中第 0 个字符表示该字符串为空串，所以我们可以得到这样一个结论：C[0][j]=j，因为这种情况指的是 A 为空串，B 为长度为 j 的串，那么我们只需要将 B 中相应的串插入、改成 A 中对应的字符即可，我们可以发现，无论哪一种，消耗都是 j；同理，C[i][0]=i，分析同理，不再赘述。

结合问题分析和上述分析，我们可以得到这个问题的递归方程：



三、动态规划的填表的过程（我们将以一个简单的实例进行演示，这其实也是算法设计时很重要的一步）

我们观察递归式的最后一项就应该明白，我们的 i 、 j 的变化，是一个固定，另一个变化的过程，这也是填表的过程。其实我们的最后一项写详细一些为下式：



从上式中，我们明显可以看见，这其实就是两重循环的过程。但要记住我们动态规划比较重要的一点，同时也是我们选择动态规划的目的：我们的计算过程就是填表的过程，并且我们在计算一个大问题时，会用到之前的小问题，那就是说，其实在我们填表的过程中，我们要尽量使我们的操作运用到之前在表中填的值。下面，我们将以一个简单的实例给出相应的操作。

比如：get 和 greet 的编辑距离。我们由上述的操作可以明白，我们需要做出相应的表，如下所示：

第一行的填表过程中，其实就是 C[0][j] = j 的过程，这个也是上述递归式的顺序

	（空串）	g	r	e	e	t
（空串）	0	1	2	3	4	5
g						
e						
t						

第二行的填表过程中，(**说明 x=?y，当 x=y 时，该表示为 0，反之为 1**)

比如 C[1][0] = 1 ,

C[1][1] = min{C[0][0] + g =?g , C[1][0] + 1 , C[0][1] + 1} 而这里所用到的值，我们其实之前就已经填好，不需要重新计算，这正是动态规划的魅力所在，经过计算 C[1][1] = 0。

同理，C[1][2] = min{C[0][1] + g=?r , C[0][2] + 1 , C[1][1] + 1} ， 经过计算 C[1][2] = 1

C[1][3] = min{C[0][2] + g=?e , C[0][3] + 1 , C[1][2] + 1} ， 经过计算 C[1][3] = 2

C[1][4] = min{C[0][3] + g=?e , C[0][4] + 1 , C[1][3] + 1} ， 经过计算 C[1][4] = 3

C[1][5] = min{C[0][4] + g=?t , C[0][5] + 1 , C[1][4] + 1} ， 经过计算 C[1][5] = 4

将上述结果填入表格即有以下表格：

	（空串）	g	r	e	e	t
（空串）	0	1	2	3	4	5
g	1	0	1	2	3	4
e						
t						

类似上述操作，直到将表格填满……
通过计算会有以下表格：

	（空串）	g	r	e	e	t
（空串）	0	1	2	3	4	5
g	1	0	1	2	3	4
e	2	1	1	1	2	3
t	3	2	2	2	2	2

其实，我们在求解的过程中能够找到最优解（也就是如何变化）这样是最好，但是，我们这个问题存在的最优解不唯一，很难用相应的数据结果进行记录。上述就是整个 edit distance 的动态规划过程。

简单的伪码描述为：（其实已经离实现不远了）

```
Function editDistance(x , y , len_x + 1 , len_y + 1):   // +1 是考虑到字符给出一个空串的位置
    Define a list like: distance_list [len_x + 1][len_y + 1]
    For i <- 0 to len_x -1:
        For j <-0 to len_y -1:
            If i equal to 0:   // 这就是当 x 为空串的情况
                Distance_list[i][j] = j

            Else if j equal to 0:   // 这就是当 y 为空串的情况
                Distance_list[i][j] = i

            Else:
                Delete = distance_list[i – 1][j] + 1   // 删除
                Insert = distance_list[i][j – 1] + 1   // 插入
                Const = (x[i - 1] == y[j -1]) //减 1 是因为，字符串的 index 是从 0 开始的
                Replace = distance_list[i - 1][j - 1] + const   //替换
                Distance_list[i][j] = min{delete , insert , replace} // 选择三种情况中的最小
    The distance_list[len_x + 1][len_y + 1] is the edit distance
    END
```