编辑距离

1、基础概念

编辑距离（Minimum Edit Distance，MED），由俄罗斯科学家 Vladimir Levenshtein 在1965年提出，也因此而得名 Levenshtein Distance。

在信息论、语言学和计算机科学领域，Levenshtein Distance 是用来度量两个序列相似程度的指标。通俗地来讲，编辑距离指的是在两个单词<w_1,w_2>之间，由其中一个单词w_1转换为另一个单词w_2所需要的最少单字符编辑操作次数。

在这里定义的单字符编辑操作有且仅有三种：

插入（Insertion）
删除（Deletion）
替换（Substitution）

譬如，"kitten" 和 "sitting" 这两个单词，由 "kitten" 转换为 "sitting" 需要的最少单字符编辑操作有：

1.kitten → sitten (substitution of "s" for "k") 将k替换为s
2.sitten → sittin (substitution of "i" for "e") 将e替换为i
3.sittin → sitting (insertion of "g" at the end) 在末尾插入g

好，概念有了，让我们看看如何用代码来实现。

2、四种操作

我们现在给定两个字符串s1,s2，分别为“mleast”和“alast”，以及动态规划表DP[i][j]，其中i,j分别表示当前s1和s2所比较的两个字母的位置。我们将通过这个例子来理解这个算法。

a、跳过操作

我们首先看两个字符串的最后一个字母，他们都是t，很容易想到，如果两个字母相同的话，那么这两个字母是不会对结果产生任何影响的，所以**编辑距离(mleast,alast)** = **编辑距离(mleas,alas)**，这个操作就是跳过操作。用代码表示就是**dp[i][j]=dp[i-1][j-1]**。

在这个例子中，后三个字母完全相同，那么我们直接跳到“mle”和“al”这步判断，这个时候，两个字符串的末尾字母是不同的。对于这样的情况，我们保持s2不变，对于s1有三种操作可以选择：

b、删除操作

删除操作是最明显的答案，在这里我们只需要将“mle”中的“e”删除就能得到与第二个字符串类似的结果，删除后s1的索引i前移，继续和s2比较。则有**编辑距离(mle,al)** = **编辑距离(ml,al)+1**。用代码表示就是**dp[i][j] = dp[i-1][j] + 1**。

c、插入操作

在这里，我们也可以同样选择插入操作来完成，即将“mle”改变成“mlel”,这个时候，两个位置的字符就匹配了。我们需要前移s2的j索引，继续与s1比较。即编辑距离(mle,al) = 编辑距离(mlel,al)+1。用代码表示就是dp[i][j] = dp[i][j-1] + 1。

d、替换操作

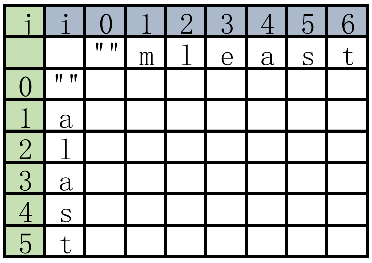
最后一种最直接的方式就是替换操作。就是把“mle”中的“e”替换为“l”，即“mle”变更为“mll”。这样两个字符串在当前位置字符就一样了，回归成为了第一种状态。即**编辑距离(mlel,al) = 编辑距离(mle,a)+1。用代码表示就是dp[i][j] = dp[i-1][j-1] + 1。**

现在，我们回到最初的阶段，对于“mle”和“al”这两个单词，我们到底该选择上面的哪个操作才是最优解呢？很简单，选择操作数最小的一个，即**min(dp[i-1][j] + 1,dp[i][j-1] + 1,dp[i-1][j-1] + 1)**。

3、填表

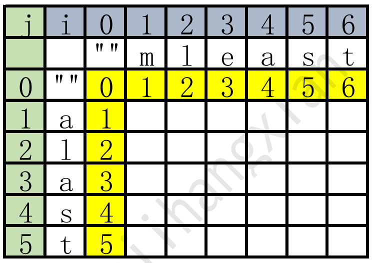
通过上文的描述，我们对基础概念以及核心算法有了一个大致的了解。那么，现在我们将通过填表，来深入的学习。

首先，建立一个动态规划表，横纵坐标分别是两个字母，但是请注意要在每个字目前留个空字符的位置。具体如下图所示。



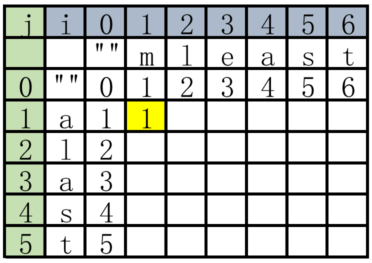
j	i	0	1	2	3	4	5	6
		""	m	l	e	a	s	t
0	""							
1	a							
2	l							
3	a							
4	s							
5	t							

我们很容易想到的就是一个空串在变成含有指定数量n的字符时，必然会经过n次操作。所以最初的两列分别填写如下，我们用黄色标记本次操作 ：

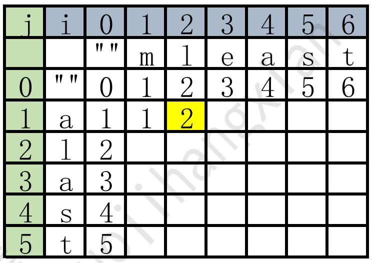


j	i	0	1	2	3	4	5	6
		""	m	l	e	a	s	t
0	""	0	1	2	3	4	5	6
1	a	1						
2	l	2						
3	a	3						
4	s	4						
5	t	5						

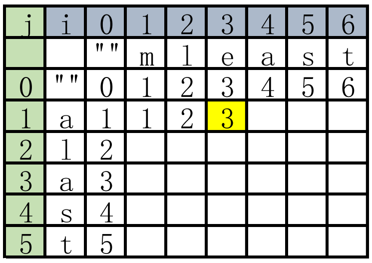
接下来，我们来填写第一个字母a这一行。根据上面的公式，在不相等的时候我们分别尝试dp[i-1][j]+1 、dp[i][j-1]+1和dp[i-1][j-1]+1,然后最小值，我们用黄色代表填入值，则有：



j	i	0	1	2	3	4	5	6
		""	m	l	e	a	s	t
0	""	0	1	2	3	4	5	6
1	a	1	1					
2	l	2						
3	a	3						
4	s	4						
5	t	5						



j	i	0	1	2	3	4	5	6
		""	m	l	e	a	s	t
0	""	0	1	2	3	4	5	6
1	a	1	1	2				
2	l	2						
3	a	3						
4	s	4						
5	t	5						



j	i	0	1	2	3	4	5	6
		""	m	l	e	a	s	t
0	""	0	1	2	3	4	5	6
1	a	1	1	2	3			
2	l	2						
3	a	3						
4	s	4						
5	t	5						

这时，我们遇到了两个相同的字母，根据上文推倒的规则，这时应采取跳过操作，即dp[i][j]等于dp[i-1][j-1]

j	i	0	1	2	3	4	5	6
		""	m	l	e	a	s	t
0	""	0	1	2	3	4	5	6
1	a	1	1	2	3	3		
2	l	2						
3	a	3						
4	s	4						
5	t	5						

我们继续填写，直到完成全部表格。

j	i	0	1	2	3	4	5	6
		""	m	l	e	a	s	t
0	""	0	1	2	3	4	5	6
1	a	1	1	2	3	3	4	5
2	l	2	2	1	2	3	4	5
3	a	3	3	3	2	2	3	4
4	s	4	4	4	3	3	2	3
5	t	5	5	5	4	4	3	2

通过填表发现，我们最多只需要两步，就可以把“alast”变成“mleast”。事实上也是如此：
第一步：在la中间插入e，则alast就变成了aleast。
第二步：把a改成m，就得到了mleast。

更神奇的是，如果你想把mleast变成alast也是两步就能完成：
第一步：删除e，则mleast就变成了mlast。
第二部：把m改成a就得到了alast。

4、编写代码

将前文的核心内容进行逻辑化，就能编写出我们的代码。

主要步骤如下：

a、建立动态规划表dp
b、初始化dp表的空字符串状态
c、计算填表

核心动态转移方程：
如果两个字符相等：则dp[i][j] = dp[i-1][j-1]
否则：dp[i][j] = min(dp[i-1][j]+1,dp[i][j-1]+1,dp[i-1][j-1]+1)

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

using namespace std;
#define N 2001  //字符串最大长度为2000
//动态规划表，记录当前从a[i]到b[j]需要的操作数
//i是纵向字符串的索引 j是横向字符串的索引
int dp[N][N];
char a[N], b[N]; //待检测字符串

int main() {
```

```
    //读入两个待处理的字符，并计算长度  a是纵向字符串 b是横向字符串
   cin >> a >> b;
    int lena = strlen(a);
    int lenb = strlen(b);
    //空串想要达到指定字符的样子，必然需要指定的次数
   for (int i = 1; i <= lena; i++) {
        dp[i][0] = i;
    }
    for (int i = 1; i <= lenb; i++) {
        dp[0][i] = i;
    }
    for (int i = 1; i <= lena; i++) {
        for (int j = 1; j <= lenb; j++) {
            //两个字符相等时，同时减1
            if (a[i - 1] == b[j - 1]) {
                dp[i][j] = dp[i - 1][j - 1];
            }
                //反之则比较三个操作，选择最小值
          else {
                int value = min(dp[i - 1][j], dp[i][j - 1]);
                dp[i][j] = min(value, dp[i - 1][j - 1]) + 1;
            }
        }
    }
    //结果存储在最后一个格子中
   printf("%d\n", dp[lena][lenb]);
    return 0;
}
```