编辑距离算法（动态规划的应用）

一、问题来源

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

1、插入一个字符

2、删除一个字符

3、替换一个字符

4、什么都不用做

二、思路

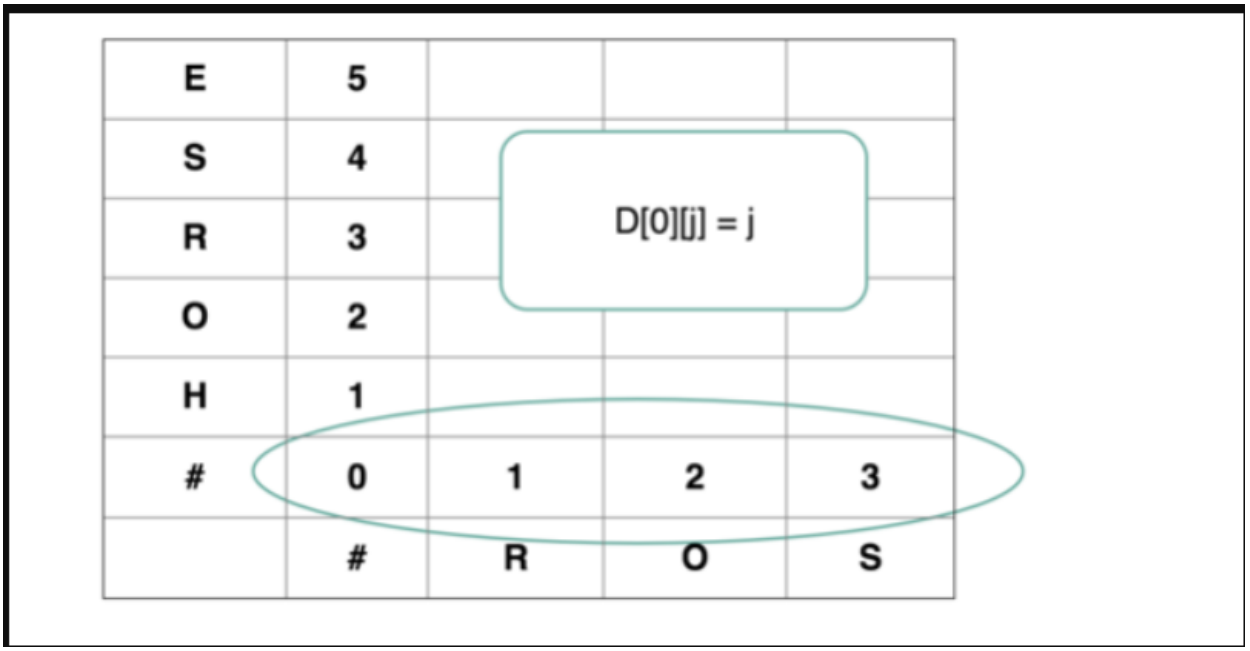
解决两个字符串的动态规划问题，一般都是用两个指针 `i,j` 分别指向两个字符串的最后，然后一步步往前走，缩小问题的规模。

```
def minDistance(s1, s2) -> int:
    def dp(i, j):
        # base case
        if i == -1: return j + 1
        if j == -1: return i + 1

        if s1[i] == s2[j]:
            return dp(i - 1, j - 1)  # 啥都不做
        else:
            return min(
                dp(i, j - 1) + 1,    # 插入
                dp(i - 1, j) + 1,    # 删除
                dp(i - 1, j - 1) + 1 # 替换
            )

    # i，j 初始化指向最后一个索引
    return dp(len(s1) - 1, len(s2) - 1)
```

容易出现重叠路径，因此使用动态规划记录最小编辑距离，用个dp表保存编辑距离

```
dp[i-1][j-1]        # 存储 s1[0...i] 和 s2[0...j] 的最小编辑距离
```

先初始化表格：



三、代码详解

```
int minDistance(String s1, String s2) {
    int m = s1.length(), n = s2.length();
    int[][] dp = new int[m + 1][n + 1];
    // base case
    for (int i = 1; i <= m; i++)
        dp[i][0] = i;
    for (int j = 1; j <= n; j++)
        dp[0][j] = j;
    // 自底向上求解
    for (int i = 1; i <= m; i++)
        for (int j = 1; j <= n; j++)
            if (s1.charAt(i-1) == s2.charAt(j-1))
                dp[i][j] = dp[i - 1][j - 1];        //如果字符相同，i,j直接后移一位
            else
                dp[i][j] = min(
                    dp[i - 1][j] + 1,    //删除
                    dp[i][j - 1] + 1,    //添加
                    dp[i-1][j-1] + 1     //替换
                );
    // 储存着整个 s1 和 s2 的最小编辑距离
    return dp[m][n];
}

int min(int a, int b, int c) {
    return Math.min(a, Math.min(b, c));
}
```

2、数组实现

```
class Solution {
    public boolean oneEditAway(String first, String second) {
        int fl = first.length();
        int sl = second.length();
        if(Math.abs(fl - sl) > 1)
            return false;
        char[] arr1 = first.toCharArray();
        char[] arr2 = second.toCharArray();
        int i = 0;
        int j = 0;
        int count = 0;
        while(i < fl && j < sl){
            if(arr1[i] == arr2[j]){
                i++;
                j++;
            }else{
                if(i + 1 < fl && arr1[i+1] == arr2[j])
                    i++;
                else if(j + 1 < sl && arr1[i] == arr2[j+1])
                    j++;
                else{
                    i++;
                    j++;
                }
                count++;
            }
            if(count > 1)
                return false;
        }
        return count + (fl-i) + (sl-j) > 1 ? false : true;
    }
}
```

执行用时 :3 ms, 在所有 Java 提交中击败了34.37%的用户

内存消耗 :38.4 MB, 在所有 Java 提交中击败了100.00%的用户