最小编辑代价

【题目】

给定两个字符串 str1 和 str2，再给定三个整数 ic、dc 和 rc，分别代表插入、删除和替换一个字符的代价，返回将 str1 编辑成 str2 的最小代价。

【举例】

str1="abc"，str2="adc"，ic=5，dc=3，rc=2。

从"abc"编辑成"adc"，把'b'替换成'd'是代价最小的，所以返回 2。

str1="abc"，str2="adc"，ic=5，dc=3，rc=100。

从"abc"编辑成"adc"，先删除'b'，然后插入'd'是代价最小的，所以返回 8。

str1="abc"，str2="abc"，ic=5，dc=3，rc=2。

不用编辑了，本来就是一样的字符串，所以返回 0。

【难度】

校 ★★★☆

【解答】

如果 str1 的长度为 M，str2 的长度为 N，经典动态规划的方法可以达到时间复杂度为 $O(M\times N)$，额外空间复杂度为 $O(M\times N)$。如果结合空间压缩的技巧，可以把额外空间复杂度减至 $O(\min\{M,N\})$。

先来介绍经典动态规划的方法。首先生成大小为$(M+1)\times(N+1)$的矩阵 dp，dp[i][j]的值代表 str1[0..i-1]编辑成 str2[0..j-1]的最小代价。举个例子，str1="ab12cd3"，str2="abcdf"，ic=5，dc=3，rc=2。dp 是一个 8×6 的矩阵，最终计算结果如下。

	' '	'a'	'b'	'c'	'd'	'f'
' '	0	5	10	15	20	25
'a'	3	0	5	10	15	20
'b'	6	3	0	5	10	15
'1'	9	6	3	2	7	12

'2'	12	9	6	5	4	9
'c'	15	12	9	6	7	6
'd'	18	15	12	9	6	9
'3'	21	18	15	12	9	8

下面具体说明 dp 矩阵每个位置的值是如何计算的。

1．dp[0][0]=0，表示 str1 空的子串编辑成 str2 空的子串的代价为 0。

2．矩阵 dp 第一列即 dp[0..M-1][0]。dp[i][0]表示 str1[0..i-1]编辑成空串的最小代价，毫无疑问，是把 str1[0..i-1]所有的字符删掉的代价，所以 dp[i][0]=dc*i。

3．矩阵 dp 第一行即 dp[0][0..N-1]。dp[0][j]表示空串编辑成 str2[0..j-1]的最小代价，毫无疑问，是在空串里插入 str2[0..j-1]所有字符的代价，所以 dp[0][j]=ic*j。

4．其他位置按照从左到右，再从上到下来计算，dp[i][j]的值只可能来自以下四种情况。

- str1[0..i-1]可以先编辑成 str1[0..i-2]，也就是删除字符 str1[i-1]，然后由 str1[0..i-2]编辑成 str2[0..j-1]，dp[i-1][j]表示 str1[0..i-2]编辑成 str2[0..j-1]的最小代价，那么 dp[i][j]可能等于 dc+dp[i-1][j]。
- str1[0..i-1]可以先编辑成 str2[0..j-2]，然后将 str2[0..j-2]插入字符 str2[j-1]，编辑成 str2[0..j-1]，dp[i][j-1]表示 str1[0..i-1]编辑成 str2[0..j-2]的最小代价，那么 dp[i][j]可能等于 dp[i][j-1]+ic。
- 如果 str1[i-1]!=str2[j-1]。先把 str1[0..i-1]中 str1[0..i-2]的部分变成 str2[0..j-2]，然后把字符 str1[i-1]替换成 str2[j-1]，这样 str1[0..i-1]就编辑成 str2[0..j-1]了。dp[i-1][j-1]表示 str1[0..i-2]编辑成 str2[0..i-2]的最小代价，那么 dp[i][j]可能等于 dp[i-1][j-1]+rc。
- 如果 str1[i-1]==str2[j-1]。先把 str1[0..i-1]中 str1[0..i-2]的部分变成 str2[0..j-2]，因为此时字符 str1[i-1]等于 str2[j-1]，所以 str1[0..i-1]已经编辑成 str2[0..j-1]了。dp[i-1][j-1]表示 str1[0..i-2]编辑成 str2[0..i-2]的最小代价，那么 dp[i][j]可能等于 dp[i-1][j-1]。

5．以上四种可能的值中，选最小值作为 dp[i][j]的值。dp 最右下角的值就是最终结果。

具体过程请参看如下代码中的 minCost1 方法。

```
public int minCost1(String str1, String str2, int ic, int dc, int rc) {
        if (str1 == null || str2 == null) {
                return 0;
        }
        char[] chs1 = str1.toCharArray();
        char[] chs2 = str2.toCharArray();
        int row = chs1.length + 1;
```

```
        int col = chs2.length + 1;
        int[][] dp = new int[row][col];
        for (int i = 1; i < row; i++) {
                dp[i][0] = dc * i;
        }
        for (int j = 1; j < col; j++) {
                dp[0][j] = ic * j;
        }
        for (int i = 1; i < row; i++) {
                for (int j = 1; j < col; j++) {
                        if (chs1[i - 1] == chs2[j - 1]) {
                                dp[i][j] = dp[i - 1][j - 1];
                        } else {
                                dp[i][j] = dp[i - 1][j - 1] + rc;
                        }
                        dp[i][j] = Math.min(dp[i][j], dp[i][j - 1] + ic);
                        dp[i][j] = Math.min(dp[i][j], dp[i - 1][j] + dc);
                }
        }
        return dp[row - 1][col - 1];
}
```

经典动态规划方法结合空间压缩的方法。空间压缩的原理请读者参考本书“矩阵的最小路径和”问题，这里不再详述。但是本题空间压缩的方法有一点特殊。在“矩阵的最小路径和”问题中，dp[i][j]依赖两个位置的值 dp[i-1][j]和 dp[i][j-1]，滚动数组从左到右更新是没有问题的，因为在求 dp[j]的时候，dp[j]没有更新之前相当于 dp[i-1][j]的值，dp[j-1]的值又已经更新过相当于 dp[i][j-1]的值。而本题 dp[i][j]依赖 dp[i-1][j]、dp[i][j-1]和 dp[i-1][j-1]的值，所以滚动数组从左到右更新时，还需要一个变量来保存 dp[j-1]没更新之前的值，也就是左上角的 dp[i-1][j-1]。

理解了上述过程后，就不难发现该过程确实只用了一个 dp 数组，但 dp 长度等于 str2 的长度加 1（即 N+1），而不是 $O(\min\{M,N\})$。所以还要把 str1 和 str2 中长度较短的一个作为列对应的字符串，长度较长的作为行对应的字符串。上面介绍的动态规划方法都是把 str2 作为列对应的字符串，如果 str1 做了列对应的字符串，把插入代价 ic 和删除代价 dc 交换一下即可。

具体过程请参看如下代码中的 minCost2 方法。

```
public int minCost2(String str1, String str2, int ic, int dc, int rc) {
        if (str1 == null || str2 == null) {
                return 0;
        }
        char[] chs1 = str1.toCharArray();
        char[] chs2 = str2.toCharArray();
        char[] longs = chs1.length >= chs2.length ? chs1 : chs2;
```

```
		char[] shorts = chs1.length < chs2.length ? chs1 : chs2;
		if (chs1.length < chs2.length) { // str2 较长就交换 ic 和 dc 的值
			int tmp = ic;
			ic = dc;
			dc = tmp;
		}
		int[] dp = new int[shorts.length + 1];
		for (int i = 1; i <= shorts.length; i++) {
			dp[i] = ic * i;
		}
		for (int i = 1; i <= longs.length; i++) {
			int pre = dp[0]; // pre 表示左上角的值
			dp[0] = dc * i;
			for (int j = 1; j <= shorts.length; j++) {
				int tmp = dp[j]; // dp[j]没更新前先保存下来
				if (longs[i - 1] == shorts[j - 1]) {
					dp[j] = pre;
				} else {
					dp[j] = pre + rc;
				}
				dp[j] = Math.min(dp[j], dp[j - 1] + ic);
				dp[j] = Math.min(dp[j], tmp + dc);
				pre = tmp; // pre 变成 dp[j]没更新前的值
			}
		}
		return dp[shorts.length];
	}
```

字符串的交错组成

【题目】

给定三个字符串 str1、str2 和 aim，如果 aim 包含且仅包含来自 str1 和 str2 的所有字符，而且在 aim 中属于 str1 的字符之间保持原来在 str1 中的顺序，属于 str2 的字符之间保持原来在 str2 中的顺序，那么称 aim 是 str1 和 str2 的交错组成。实现一个函数，判断 aim 是否是 str1 和 str2 交错组成。

【举例】

str1="AB"，str2="12"。那么"AB12"、"A1B2"、"A12B"、"1A2B"和"1AB2"等都是 str1 和 str2 的交错组成。