最长递增子序列

【题目】

给定数组arr，返回arr的最长递增子序列。

【举例】

arr=[2,1,5,3,6,4,8,9,7]，返回的最长递增子序列为[1,3,4,8,9]。

【要求】

如果arr长度为N，请实现时间复杂度为O(N*logN)的方法。

【难度】

校 ★★★☆

【解答】

先来介绍时间复杂度为O(N^2)的方法，具体过程如下：

1，生成长度为N的数组dp，dp[i]表示在以arr[i]这个数结尾的情况下，arr[0..i]中的最大递增子序列长度。

2，对于第一个数arr[0]来说令dp[0]=1，接下来从左到右依次算出以每个位置的数结尾的情况下，最长递增子序列长度。

3，假设计算到位置i，求以arr[i]结尾情况下的最长递增子序列长度即dp[i]。如果最长递增子序列以arr[i]结尾，那么在arr[0..i-1]中所有比arr[i]小的数都可以作为倒数第二个数。在这么多倒数第二个数的选择中，以哪个数结尾的最大递增子序列更大，就选那个数作为倒数第二个数，所以dp[i]=Max{dp[j]+1(0<=j<i，arr[j]<arr[i])}。如果arr[0..i-1]中所有的数都不比arr[i]小，令dp[i]=1即可，说明以arr[i]结尾情况下的最长递增子序列只包含arr[i]。

按照步骤1~3可以计算出dp数组，具体过程请参看如下代码中的getdp1方法。

```
public int[] getdp1(int[] arr) {
        int[] dp = new int[arr.length];
        for (int i = 0; i < arr.length; i++) {
                dp[i] = 1;
                for (int j = 0; j < i; j++) {
                        if (arr[i] > arr[j]) {
                                dp[i] = Math.max(dp[i], dp[j] + 1);
                        }
                }
        }
        return dp;
}
```

接下来解释如何根据求出的dp数组得到最长递增子序列，以题目的例子来说明，arr=[2,1,5,3,6,4,8,9,7]，求出的数组dp=[1,1,2,2,3,3,4,5,4]。

1，遍历dp数组，找到最大值以及位置。在本例中最大值为5，位置为7。说明最终的最长递增子序列的长度为5，并且应该以arr[7]这个数(arr[7]==9)结尾。

2，从arr数组的位置7开始从右向左遍历。如果对于某一个位置i，既有arr[i]<arr[7]又有dp[i]==dp[7]-1，说明arr[i]可以作为最长递增子序列的倒数第二个数。在本例中，arr[6]<arr[7]并且dp[6]==dp[7]-1，所以8应该作为最长递增子序列的倒数第二个数。

3，从arr数组的位置6开始继续向左遍历，按照同样的过程找到倒数第三个数。在本例中，位置5满足arr[5]<arr[6]并且dp[5]==dp[6]-1，同时位置4也满足。选arr[5]或者arr[4]作为倒数第三个数都可以。

4，重复这样的过程，直到所有的数都找出来。

dp数组包含了每一步决策的信息，其实根据dp数组找出最长递增子序列的过程就是从某一个位置开始逆序还原出决策路径的过程。具体过程请参看如下代码中的generateLIS方法。

```
public int[] generateLIS(int[] arr, int[] dp) {
        int len = 0;
```

```
		int index = 0;
		for (int i = 0; i < dp.length; i++) {
			if (dp[i] > len) {
				len = dp[i];
				index = i;
			}
		}
		int[] lis = new int[len];
		lis[--len] = arr[index];
		for (int i = index; i >= 0; i--) {
			if (arr[i] < arr[index] && dp[i] == dp[index] - 1) {
				lis[--len] = arr[i];
				index = i;
			}
		}
		return lis;
	}
```

整个过程的主方法参看如下代码中的lis1方法。

```
	public int[] lis1(int[] arr) {
		if (arr == null || arr.length == 0) {
			return null;
		}
		int[] dp = getdp1(arr);
		return generateLIS(arr, dp);
	}
```

很明显，计算dp数组的过程时间复杂度为O(N^2)，根据dp数组得到最长递增子序列的过程时间复杂度O(N)，所以整个过程的时间复杂度为O(N^2)。如果让时间复杂度达到O(N*logN)，只要让计算dp数组的过程达到时间复杂度O(N*logN)即可，之后根据dp数组生成最长递增子序列的过程是一样的。

时间复杂度O(N*logN)生成dp数组的过程是利用二分查找来进行的优化。先生成一个长度为N的数组ends，初始时ends[0]=arr[0]，其他位置上的值为0。生成整型变量right，初始时right=0。在从左到右遍历arr数组的过程中，求解dp[i]的过程需要使用ends数组和right变量，所以这里解释一下其含义。遍历的过程中，ends[0..right]为有效区，ends[right+1..N-1]为无效区。对于有效区上的位置b，如果有ends[b]==c，则表示遍历到目前为止，在所有长度为b+1的递增序列中，最小的结尾数是c。无效区的位置则没有意义。
比如arr=[2,1,5,3,6,4,8,9,7]，初始时dp[0]=1，ends[0]=2，right=0。ends[0..0]为有效区，ends[0]==2的含义是，在遍历过arr[0]之后，所有长度为1的递增序列中(此时只有[2])，最小的结尾数是2。之后的遍历继续用这个例子来说明求解过程。
1，遍历到arr[1]==1。ends有效区=ends[0..0]=[2]，在有效区中找到最左边的大于等于arr[1]的数。发现是ends[0]，表示以arr[1]结尾的最长递增序列只有arr[1]，所以令dp[1]=1。然后令ends[0]=1，因为遍历到目前为止，在所有长度为1的递增序列中，最小的结尾数是1而不再是2了。
2，遍历到arr[2]==5。ends有效区=ends[0..0]=[1]，在有效区中找到最左边的大于等于arr[2]的数。发现没有这样的数，表示以arr[2]结尾的最长递增序列长度=ends有效区长度+1，所以令dp[2]=2。ends整个有效区都没有比arr[2]更大的数，说明发现了比ends有效区长度更长的递增序列，于是把有效区扩大，ends有效区=ends[0..1]=[1,5]。
3，遍历到arr[3]==3。ends有效区=ends[0..1]=[1,5]，在有效区中用二分法找到最左边的大于等于arr[3]的数。发现是ends[1]，表示以arr[3]结尾的最长递增序列长度为2，所以令dp[3]=2。然后令ends[1]=3，因为遍历到目前为止，在所有长度为2的递增序列中，最小的结尾数是3而不再是5了。
4，遍历到arr[4]==6。ends有效区=ends[0..1]=[1,3]，在有效区中用二分法找到最左边的大于等于arr[4]的数。发现没有这样的数，表示以arr[4]结尾的最长递增序列长度=ends有效区长度+1，所以令dp[4]=3。ends整个有效区都没有比arr[4]更大的数，说明发现了比ends有效区长度更长的递增序列，于是把有效区扩大，ends有效区=ends[0..2]=[1,3,6]。

5，遍历到arr[5]==4。ends有效区=ends[0..2]=[1,3,6]，在有效区中用二分法找到最左边的大于等于arr[5]的数。发现是ends[2]，表示以arr[5]结尾的最长递增序列长度为3，所以令dp[5]=3。然后令ends[2]=4，表示在所有长度为3的递增序列中，最小的结尾数变为4。
6，遍历到arr[6]==8。ends有效区=ends[0..2]=[1,3,4]，在有效区中用二分法找到最左边的大于等于arr[6]的数。发现没有这样的数，表示以arr[6]结尾的最长递增序列长度=ends有效区长度+1，所以令dp[6]=4。ends整个有效区都没有比arr[6]更大的数，说明发现了比ends有效区长度更长的递增序列，于是把有效区扩大，ends有效区=ends[0..3]=[1,3,4,8]。
7，遍历到arr[7]==9。ends有效区=ends[0..3]=[1,3,4,8]，在有效区中用二分法找到最左边的大于等于arr[7]的数。发现没有这样的数，表示以arr[7]结尾的最长递增序列长度=ends有效区长度+1，所以令dp[7]=5。ends整个有效区都没有比arr[7]更大的数，于是把有效区扩大，ends有效区=ends[0..5]=[1,3,4,8,9]。
8，遍历到arr[8]==7。ends有效区=ends[0..5]=[1,3,4,8,9]，在有效区中用二分法找到最左边的大于等于arr[8]的数。发现是ends[3]，表示以arr[8]结尾的最长递增序列长度为4，所以令dp[8]=4。然后令ends[3]=7，表示在所有长度为4的递增序列中，最小的结尾数变为7。
具体过程请参看如下代码中的getdp2方法。

```
public int[] getdp2(int[] arr) {
        int[] dp = new int[arr.length];
        int[] ends = new int[arr.length];
        ends[0] = arr[0];
        dp[0] = 1;
        int right = 0;
        int l = 0;
        int r = 0;
        int m = 0;
        for (int i = 1; i < arr.length; i++) {
                l = 0;
                r = right;
                while (l <= r) {
                        m = (l + r) / 2;
                        if (arr[i] > ends[m]) {
                                l = m + 1;
                        } else {
                                r = m - 1;
                        }
                }
                right = Math.max(right, l);
                ends[l] = arr[i];
                dp[i] = l + 1;
        }
        return dp;
}
```

时间复杂度O(N*logN)方法的整个过程请参看如下代码中的lis2方法。

```
public int[] lis2(int[] arr) {
        if (arr == null || arr.length == 0) {
                return null;
        }
        int[] dp = getdp2(arr);
        return generateLIS(arr, dp);
}
```

最长公共子序列问题

【题目】

给定两个字符串str1和str2，返回两个字符串的最长公共子序列。

【举例】

str1=“1A2C3D4B56”，str2=“B1D23CA45B6A”。

“123456”或者“12C4B6”都是最长公共子序列，返回哪一个都行。

【难度】

尉 ★★☆☆

【解答】

本题是非常经典的动态规划问题，先来介绍求解动态规划表的过程。如果str1的长度为M，str2的长度为N，生成大小为M*N的矩阵dp，行数为M，列数为N。dp[i][j]的含义是str1[0..i]与str2[0..j]的最长公共子序列的长度。从左到右，再从上到下计算矩阵dp：

1，矩阵dp第一列即dp[0..M-1][0]，dp[i][0]的含义是str1[0..i]与str2[0]的最长公共子序列长度。str2[0]只有一个字符，所以dp[i][0]最大为1。如果str1[i]==str2[0]，令dp[i][0]=1，一旦dp[i][0]被设置为1，之后的dp[i+1..M-1][0]也都为1。比如str1[0..M-1]=“ABCDE”，str2[0]=“B”。str1[0]为“A”与str2[0]不相等，所以dp[0][0]=0。str1[1]为“B”与str2[0]相等，所以str1[0..1]与str2[0]的最长公共子序列为“B”，令dp[1][0]=1。之后的dp[2..4][0]肯定都是1，因为str[0..2]、str[0..3]和str[0..4]与str2[0]的最长公共子序列肯定有“B”。

2，矩阵dp第一行即dp[0][0..N-1]与步骤1同理，如果str1[0]==str2[j]，则令dp[0][j]=1，一旦dp[0][j]被设置为1，之后的dp[0][j+1..N-1]也都为1。

3，对于其他位置(i,j)，dp[i][j]的值只可能来自以下三种情况：

1）可能是dp[i-1][j]，代表str1[0..i-1]与str2[0..j]的最长公共子序列长度。比如str1=“A1BC2”，str2=“AB34C”。str1[0..3]即“A1BC”与str2[0..4]即“AB34C”的最长公共子序列为“ABC”，即dp[3][4]为3。str1[0..4]即“A1BC2”与str2[0..4]即“AB34C”的最长公共子序列也是“ABC”，所以dp[4][4]也为3。

2）可能是dp[i][j-1]，代表str1[0..i]与str2[0..j-1]的最长公共子序列长度。比如str1=“A1B2C”，str2=“AB3C4”。str1[0..4]即“A1B2C”与str2[0..3]即“AB3C”的最长公共子序列为“ABC”，即dp[4][3]为3。str1[0..4]即“A1B2C”与str2[0..4]即“AB3C4”的最长公共子序列也是“ABC”，所以dp[4][4]也为3。

3）如果str1[i]==str2[j]，还可能是dp[i-1][j-1]+1。比如str1=“ABCD”，str2=“ABCD”。str1[0..2]即“ABC”与str2[0..2]即“ABC”的最长公共子序列为“ABC”，即dp[2][2]为3。因为str1[3]==str2[3]==“D”，所以str1[0..3]与str2[0..3]的最长公共子序列是“ABCD”。

这三个可能的值中，选最大的作为dp[i][j]的值。具体过程请参看如下代码中的getdp方法。

```
public int[][] getdp(char[] str1, char[] str2) {
        int[][] dp = new int[str1.length][str2.length];
        dp[0][0] = str1[0] == str2[0] ? 1 : 0;
        for (int i = 1; i < str1.length; i++) {
            dp[i][0] = Math.max(dp[i - 1][0], str1[i] == str2[0] ? 1 : 0);
        }
        for (int j = 1; j < str2.length; j++) {
            dp[0][j] = Math.max(dp[0][j - 1], str1[0] == str2[j] ? 1 : 0);
        }
        for (int i = 1; i < str1.length; i++) {
            for (int j = 1; j < str2.length; j++) {
                dp[i][j] = Math.max(dp[i - 1][j], dp[i][j - 1]);
                if (str1[i] == str2[j]) {
                    dp[i][j] = Math.max(dp[i][j], dp[i - 1][j - 1] + 1);
                }
            }
        }
        return dp;
}
```

dp矩阵中最右下角的值代表str1整体和str2整体的最长公共子序列的长度。通过整个dp矩阵的状态，可以得到最长公共子序列。具体方法如下：
1，从矩阵的右下角开始，有三种移动方式，向上、向左、向左上。假设移动的过程中，i表示此时的行数，j表示此时的列数，同时用一个变量res来表示最长公共子序列。
2，如果dp[i][j]大于dp[i-1][j]和dp[i][j-1]，说明之前在计算dp[i][j]的时候，一定是选择了决策dp[i-1][j-1]+1，可以确定str1[i]等于str2[j]，并且这个字符一定属于最长公共子序列，把这个字符放进res，然后向左上方移动。
3，如果dp[i][j]等于dp[i-1][j]，说明之前在计算dp[i][j]的时候，dp[i-1][j-1]+1这个决策不是必须选择的决策，向上方移动即可。
4，如果dp[i][j]等于dp[i][j-1]，与步骤3同理，向左方移动。
5，如果dp[i][j]同时等于dp[i-1][j]和dp[i][j-1]，向上还是向下无所谓，选择其中一个即可，反正不会错过必须选择的字符。
也就是说通过dp求解最长公共子序列的过程，就是还原出当时是如何求解dp的过程，来自哪个策略就朝哪个方向移动。全部过程请参看如下代码中的lcse方法。

```
public String lcse(String str1, String str2) {
    if (str1 == null || str2 == null || str1.equals("") || str2.equals("")) {
       return "";
    }
    char[] chs1 = str1.toCharArray();
    char[] chs2 = str2.toCharArray();
    int[][] dp = getdp(chs1, chs2);
    int m = chs1.length - 1;
    int n = chs2.length - 1;
    char[] res = new char[dp[m][n]];
    int index = res.length - 1;
    while (index >= 0) {
       if (n > 0 && dp[m][n] == dp[m][n - 1]) {
           n--;
       } else if (m > 0 && dp[m][n] == dp[m - 1][n]) {
           m--;
       } else {
           res[index--] = chs1[m];
           m--;
           n--;
       }
    }
    return String.valueOf(res);
}
```

计算dp矩阵中的某个位置就是简单比较相关的3个位置的值而已，所以时间复杂度为O(1)，动态规划表dp的大小为M*N，所以计算dp矩阵的时间复杂度为O(M*N)。通过dp得到最长公共子序列的过程为O(M+N)，因为向左最多移动N个位置，向上最多移动M个位置，所以总的时间复杂度为O(M*N)，额外空间复杂度为O(M*N)。如果题目不要求返回最长公共子序列，只想求最长公共子序列的长度，那么可以用空间压缩的方法将额外空间复杂度减小为O(min{M,N})，有兴趣的读者请阅读本书“矩阵的最小路径和”问题，这里不再详述。

最长公共子串问题

【题目】

给定两个字符串str1和str2，返回两个字符串的最长公共子串。

【举例】

str1=“1AB2345CD”，str2=“12345EF”。返回“2345”。

【要求】

如果str1长度为M，str2长度为N，实现时间复杂度为O(M*N)，额外空间复杂度为O(1)的方法。

【难度】

校 ★★★☆

【解答】

经典动态规划的方法可以做到时间复杂度O(M*N)，额外空间复杂度O(M*N)，经过优化之后的实现可以把额外空间复杂度从O(M*N)降至O(1)，我们先来介绍经典方法。

首先需要生成动态规划表。生成大小为M*N的矩阵dp，行数为M，列数为N。dp[i][j]的含义是，在必须把str1[i]和str2[j]当做公共子串最后一个字符的情况下，公共子串最长能有多长。比如，str1=“A1234B”，str2=“CD1234”，dp[3][4]的含义是在必须把str1[3]即’3’和str2[4]即’3’当做公共子串最后一个字符的情况下，公共子串最长能有多长。这种情况下的最长公共子串为“123”，所以dp[3][4]为3。再比如，str1=“A12E4B”，str2=“CD12F4”，dp[3][4]的含义是在必须把str1[3]即’E’和str2[4]即’F’当做公共子串最后一个字符的情况下，公共子串最长能有多长。这种情况下根本不能构成公共子串，所以dp[3][4]为0。介绍了dp[i][j]的意义，接下来介绍dp[i][j]怎么求。具体过程如下：

1，矩阵dp第一列即dp[0..M-1][0]。对于某一个位置(i,0)来说，如果str1[i]==str2[0]，令dp[i][0]=1。否则令dp[i][0]=0。比如str1=“ABAC”，str2[0]=“A”。dp矩阵第一列上的值依次为dp[0][0]=1，dp[1][0]=0，dp[2][0]=1，dp[3][0]=0。

2，矩阵dp第一行即dp[0][0..N-1]与步骤1同理。对于某一个位置(0,j)来说，如果str1[0]==str2[j]，令dp[0][j]=1。否则令dp[0][j]=0。

3，其他位置按照从左到右再从上到下来计算，dp[i][j]的值只可能有两种情况。

1）如果str1[i]!=str2[j]，说明在必须把str1[i]和str2[j]当做公共子串最后一个字符是不可能的，令dp[i][j]=0。

2）如果str1[i]==str2[j]，说明str1[i]和str2[j]可以作为公共子串最后一个字符，从最后一个字符向左能扩多大的长度呢？就是dp[i-1][j-1]的值，所以令dp[i][j]=dp[i-1][j-1]+1。

如果str1=“abcde”，str2=“bebcd”。计算的dp矩阵如下：

		b	e	b	c	d
	a	0	0	0	0	0
	b	1	0	1	0	0
	c	0	0	0	2	0
	d	0	0	0	0	3
	e	0	1	0	0	0

计算dp矩阵的具体过程请参看如下代码中的getdp方法。

```
public int[][] getdp(char[] str1, char[] str2) {
        int[][] dp = new int[str1.length][str2.length];
        for (int i = 0; i < str1.length; i++) {
                if (str1[i] == str2[0]) {
                        dp[i][0] = 1;
                }
```

```
        }
        for (int j = 1; j < str2.length; j++) {
            if (str1[0] == str2[j]) {
                dp[0][j] = 1;
            }
        }
        for (int i = 1; i < str1.length; i++) {
            for (int j = 1; j < str2.length; j++) {
                if (str1[i] == str2[j]) {
                    dp[i][j] = dp[i - 1][j - 1] + 1;
                }
            }
        }
        return dp;
    }
```

生成动态规划表dp之后，得到最长公共子串是非常容易的。比如上边生成的dp中，最大值是dp[3][4]==3，说明最长公共子串的长度为3。最长公共子串的最后一个字符是str1[3]，当然也是str2[4]，因为两个字符一样。那么最长公共子串为从str1[3]开始向左一共3个字节的子串即str1[1..3]，当然也是str2[2..4]。总之，遍历dp找到最大值及其位置，最长公共子串自然可以得到。具体过程请参看如下代码中的lcst1方法，也是整个过程的主方法。

```
    public String lcst1(String str1, String str2) {
        if (str1 == null || str2 == null || str1.equals("") || str2.equals("")) {
            return "";
        }
        char[] chs1 = str1.toCharArray();
        char[] chs2 = str2.toCharArray();
        int[][] dp = getdp(chs1, chs2);
        int end = 0;
        int max = 0;
        for (int i = 0; i < chs1.length; i++) {
            for (int j = 0; j < chs2.length; j++) {
                if (dp[i][j] > max) {
                    end = i;
                    max = dp[i][j];
                }
            }
        }
        return str1.substring(end - max + 1, end + 1);
    }
```

经典动态规划的方法需要大小为M*N的dp矩阵，但实际上是可以减小至O(1)的，因为我们注意到计算每一个dp[i][j]的时候，最多只需要其左上方dp[i-1][j-1]的值，所以按照斜线方向来计算所有的值的话，只需要一个变量就可以计算出所有位置的值，如图1-1。

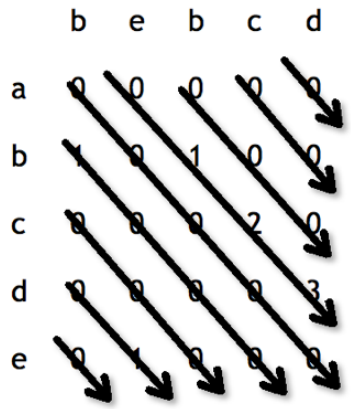

图1-1

每一条斜线在计算之前生成整型变量len，len表示左上方位置的值，初始时len=0。从斜线最左上的位置开始向右下方依次计算每个位置的值，假设计算到位置(i,j)，此时len表示位置(i-1,j-1)的值。如果str1[i]==str2[j]，那么位置(i,j)的值为len+1，如果str1[i]!=str2[j]，那么位置(i,j)的值为0。计算后将len更新成位置(i,j)的值，然后计算下一个位置，即(i+1,j+1)位置的值。依次计算下去就可以得到斜线上每个位置的值，然后再算下一条斜线。用全局变量max记录所有位置的值中的最大值。最大值出现时，用全局变量end记录其位置即可。具体过程请参看如下代码中的lcst2方法。

```
public String lcst2(String str1, String str2) {
    if (str1 == null || str2 == null || str1.equals("") || str2.equals("")) {
        return "";
    }
    char[] chs1 = str1.toCharArray();
    char[] chs2 = str2.toCharArray();
    int row = 0; // 斜线开始位置的行
    int col = chs2.length - 1; // 斜线开始位置的列
    int max = 0; // 记录最大长度
    int end = 0; // 最大长度更新时，记录子串的结尾位置
    while (row < chs1.length) {
        int i = row;
        int j = col;
        int len = 0;
        // 从(i,j)开始向右下方遍历
        while (i < chs1.length && j < chs2.length) {
                if (chs1[i] != chs2[j]) {
                        len = 0;
                } else {
                        len++;
                }
                // 记录最大值，以及结束字符的位置
                if (len > max) {
                        end = i;
                        max = len;
                }
                i++;
                j++;
        }
        if (col > 0) { // 斜线开始位置的列先向左移动
                col--;
        } else { // 列移动到最左之后，行向下移动
                row++;
        }
    }
    return str1.substring(end - max + 1, end + 1);
}
```

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O(M*N)，额外空间复杂度O(M*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)*(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]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。

1）str1[0..i-1]可以先编辑成str1[0..i-2]，也就是删除字符str1[i-1]，然后再由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]。

2）str1[0..i-1]可以先编辑成str2[0..j-2]，然后str2[0..j-2]再插入字符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。

3）如果str1[i-1]!=str2[j-1]。先把str1[0..i-1]中str1[0..i-2]的部分可以先变成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。

4）如果str1[i-1]==str2[j-1]。先把str1[0..i-1]中str1[0..i-2]的部分可以先变成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]的值又已经更新过了相当于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，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})啊。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”等等都是str1和str2交错组成。

【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O(M*N)，额外空间复杂度O(M*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aim如果是str1和str2的交错组成，aim的长度一定是M+N，否则直接返回false。然后生成大小为(M+1)*(N+1)布尔类型的矩阵dp，dp[i][j]的值代表aim[0..i+j-1]能否被str1[0..i-1]和str2[0..j-1]交错组成。计算dp矩阵的时候，是从左到右再从上到下的计算的，dp[M][N]也就是dp矩阵中最右下角的值，就表示aim整体能否被str1整体和str2整体交错组成，也就是最终结果。具体说明dp矩阵每个位置的值是如何计算。

1，dp[0][0]=true。aim为空串时，当然可以被str1为空串和str2为空串交错组成。

2，矩阵dp第一列即dp[0..M-1][0]。dp[i][0]表示aim[0..i-1]能否只被str1[0..i-1]交错组成。如果aim[0..i-1]等于str1[0..i-1]，则令dp[i][0]=true，否则令dp[i][0]=false。

3，矩阵dp第一行即dp[0][0..N-1]。dp[0][j]表示aim[0..j-1]能否只被str2[0..j-1]交错组成。如果aim[0..j-1]等于str1[0..j-1]，则令dp[i][0]=true，否则令dp[i][0]=false。

4，对于其他位置(i,j)，dp[i][j]的值由下面的情况决定。

1）dp[i-1][j]代表aim[0..i+j-2]能否被str1[0..i-2]和str2[0..j-1]交错组成，如果可以，那么如果再有str1[i-1]等于aim[i+j-1]，说明str1[i-1]又可以作为交错组成aim[0..i+j-1]的最后一个字符。令dp[i][j]=true。

2）dp[i][j-1]代表aim[0..i+j-2]能否被str1[0..i-1]和str2[0..j-2]交错组成，如果可以，那么如果再有str2[j-1]等于aim[i+j-1]，说明str1[j-1]又可以作为交错组成aim[0..i+j-1]的最后一个字符。令dp[i][j]=true。

3）如果情况1）和情况2）都不满足，令dp[i][j]=false。

具体过程请参看如下代码中的isCross1方法。

```
public boolean isCross1(String str1, String str2, String aim) {
        if (str1 == null || str2 == null || aim == null) {
                return false;
        }
        char[] ch1 = str1.toCharArray();
        char[] ch2 = str2.toCharArray();
        char[] chaim = aim.toCharArray();
        if (chaim.length != ch1.length + ch2.length) {
                return false;
        }
        boolean[][] dp = new boolean[ch1.length + 1][ch2.length + 1];
        dp[0][0] = true;
        for (int i = 1; i <= ch1.length; i++) {
                if (ch1[i - 1] != chaim[i - 1]) {
                        break;
                }
                dp[i][0] = true;
        }
        for (int j = 1; j <= ch2.length; j++) {
```

```
                if (ch2[j - 1] != chaim[j - 1]) {
                        break;
                }
                dp[0][j] = true;
        }
        for (int i = 1; i <= ch1.length; i++) {
                for (int j = 1; j <= ch2.length; j++) {
                    if ((ch1[i - 1] == chaim[i + j - 1] && dp[i - 1][j])
                        || (ch2[j - 1] == chaim[i + j - 1] && dp[i][j - 1])) {
                            dp[i][j] = true;
                    }
                }
        }
        return dp[ch1.length][ch2.length];
}
```

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。实际进行空间压缩的时候，比较str1和str2哪个长度较小，长度较小的那个作为列对应的字符串，然后生成和较短字符串长度一样的一维数组dp，滚动更新即可。具体请参看如下代码中的isCross2方法。

```
public boolean isCross2(String str1, String str2, String aim) {
        if (str1 == null || str2 == null || aim == null) {
                return false;
        }
        char[] ch1 = str1.toCharArray();
        char[] ch2 = str2.toCharArray();
        char[] chaim = aim.toCharArray();
        if (chaim.length != ch1.length + ch2.length) {
                return false;
        }
        char[] longs = ch1.length >= ch2.length ? ch1 : ch2;
        char[] shorts = ch1.length < ch2.length ? ch1 : ch2;
        boolean[] dp = new boolean[shorts.length + 1];
        dp[0] = true;
        for (int i = 1; i <= shorts.length; i++) {
                if (shorts[i - 1] != chaim[i - 1]) {
                        break;
                }
                dp[i] = true;
        }
        for (int i = 1; i <= longs.length; i++) {
                dp[0] = dp[0] && longs[i - 1] == chaim[i - 1];
                for (int j = 1; j <= shorts.length; j++) {
                    if ((longs[i - 1] == chaim[i + j - 1] && dp[j])
                        || (shorts[j - 1] == chaim[i + j - 1] && dp[j - 1])) {
                        dp[j] = true;
                    } else {
                        dp[j] = false;
                    }
                }
        }
        return dp[shorts.length];
}
```