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 $O(M\times N)$，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为 $O(M\times N)$，额外空间复杂度为 $O(M\times N)$，经过优化之后的实现可以把额外空间复杂度从 $O(M\times N)$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 $M\times N$ 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]当作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'）当作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'）当作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]的意义后，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。

- 如果 str1[i]!=str2[j]，说明在必须把 str1[i]和 str2[j]当作公共子串最后一个字符是不可能的，令 dp[i][j]=0。
- 如果 str1[i]==str2[j]，说明 str1[i]和 str2[j]可以作为公共子串的最后一个字符，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 |

实际上可进行值的传递，即=max(左边和上边)，
再考虑相等的情况=斜左上+1，
这样可以直接通过dp[行数][列数]得到最大公共子串长度。
算路径也是一样，找到变化的地方(即不同于左边和上边)。
但这种方式不符合dp的定义，计算量也增大了。

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 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 $M \times N$ 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，只需要一个变量就可以计算出所有位置的值，如图 4-1 所示。

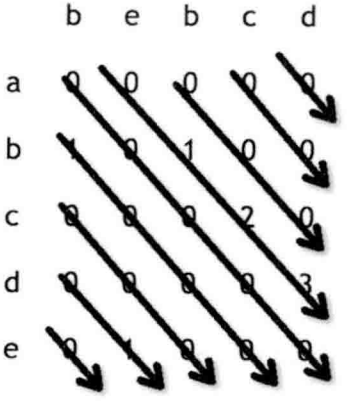


图 4-1

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) {                    依次计算每条斜线
        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);
}
```