20180326_LCS 最长公共子序列

```
1- 最长公共子序列问题不要求连续

2- 最长公共子串是要求连续的
```

第一部分、什么是动态规划算法

ok，咱们先来了解下什么是动态规划算法。

动态规划一般也只能应用于有最优子结构的问题。最优子结构的意思是局部最优解能决定全局最优解(对有些问题这个要求并不能完全满足，故有时需要引入一定的近似)。简单地说，问题能够分解成子问题来解决。

动态规划算法分以下4个步骤：

1. 描述最优解的结构
2. 递归定义最优解的值
3. 按自底向上的方式计算最优解的值　//此3步构成动态规划解的基础。
4. 由计算出的结果构造一个最优解。　//此步如果只要求计算最优解的值时，可省略。

好，接下来，咱们讨论适合采用动态规划方法的最优化问题的俩个要素：最优子结构性质，和子问题重叠性质。

- 最优子结构

如果问题的**最优解所包含的子问题的解也是最优的，我们就称该问题具有最优子结构性质（即满足最优化原理）。意思就是，总问题包含很多个子问题，而这些子问题的解也是最优的。**

- 重叠子问题

子问题重叠性质是指在用递归算法自顶向下对问题进行求解时，每次产生的子问题并不总是新问题，**有些子问题会被重复计算多次。动态规划算法正是利用了这种子问题的重叠性质，对每一个子问题只计算一次，然后将其计算结果保存在一个表格中，当再次需要计算已经计算过的子问题时，只是在表格中简单地查看一下结果**，从而获得较高的效率。

第二部分、动态规划算法解LCS问题

下面，咱们运用此动态规划算法解此LCS问题。有一点必须声明的是，LCS问题即最长公共子序列问题，它**不要求所求得的字符在所给的字符串中是连续的**（例如：输入两个字符串BDCABA和ABCBDAB，字符串BCBA和BDAB都是是它们的最长公共子序列，则输出它们的长度4，并打印任意一个子序列）。

ok，咱们马上进入面试题第56题的求解，即运用经典的动态规划算法：

2.0、LCS问题描述

56.最长公共子序列。 题目：如果字符串一的所有字符按其在字符串中的顺序出现在另外一个字符串二中， 则字符串一称之为字符串二的子串。

注意，并不要求子串（字符串一）的字符必须连续出现在字符串二中。 请编写一个函数，**输入两个字符串，求它们的最长公共子串，并打印出最长公共子串。** 例如：输入两个字符串BDCABA和ABCBDAB，字符串BCBA和BDAB都是是它们的最长公共子序列，则输出它们的长度4，并打印任意一个子序列。

分析：求最长公共子序列（Longest Common Subsequence, LCS）是一道非常经典的动态规划题，因此一些重视算法的公司像MicroStrategy都把它当作面试题。

事实上，最长公共子序列问题也有最优子结构性质。

记:

> Xi=＜x1，…，xi＞即X序列的前i个字符 (1≤i≤m)（前缀）
>
> Yj=＜y1，…，yj＞即Y序列的前j个字符 (1≤j≤n)（前缀）

假定Z=＜z1，…，zk＞∈LCS(X , Y)。

<u>分最后一个元素是否相同</u>

- 若**xm=yn**（最后一个字符相同），则不难用反证法证明：该字符必是X与Y的任一最长公共子序列Z（设长度为k）的最后一个字符，即有zk = xm = yn 且显然有Zk-1∈LCS(Xm-1 , Yn-1)即Z的前缀**Zk-1是Xm-1与Yn-1的最长公共子序列**。此时，问题化归成求Xm-1与Yn-1的LCS（*LCS(X , Y)的长度等于LCS(Xm-1 , Yn-1)*的长度加1）。
- 若**xm≠yn**，则亦不难用反证法证明：要么Z∈LCS(Xm-1, Y)，要么Z∈LCS(X , Yn-1)。由于zk≠xm与zk≠yn其中至少有一个必成立，若zk≠xm则有Z∈LCS(Xm-1 , Y)，类似的，若zk≠yn 则有Z∈LCS(X , Yn-1)。此时，问题化归成求Xm-1与Y的LCS及X与Yn-1的LCS。LCS(X , Y)的长度为：max{LCS(Xm-1 , Y)的长度, LCS(X , Yn-1)的长度}。

由于上述当**xm≠yn**的情况中，求LCS(Xm-1 , Y)的长度与LCS(X , Yn-1)的长度，这两个问题不是相互独立的：**两者都需要求LCS(Xm-1，Yn-1)的长度**。另外两个序列的LCS中包含了两个序列的前缀的LCS，故问题具有最优子结构性质考虑用动态规划法。

也就是说，解决这个LCS问题，你要求三个方面的东西：

```
1、LCS（Xm-1，Yn-1）+1;
2、LCS（Xm-1，Y），LCS（X，Yn-1）;
3、max{LCS（Xm-1，Y），LCS（X，Yn-1）}。
```

2.1、最长公共子序列的结构

最长公共子序列的结构有如下表示：

设序列X=和Y=的一个最长公共子序列Z=，则：

1. 若xm=yn，则zk=xm=yn且Zk-1是Xm-1和Yn-1的最长公共子序列；
2. 若xm≠yn且zk≠xm ，则Z是Xm-1和Y的最长公共子序列；
3. 若xm≠yn且zk≠yn ，则Z是X和Yn-1的最长公共子序列。

其中Xm-1=，Yn-1=，Zk-1=。

2.2、子问题的递归结构

由最长公共子序列问题的最优子结构性质可知，要找出X=和Y=的最长公共子序列，可按以下方式递归地进行：

- 当xm=yn时，找出Xm-1和Yn-1的最长公共子序列，然后在其尾部加上xm(=yn)即可得X和Y的一个最长公共子序列。
- 当xm≠yn时，必须解两个子问题，即找出Xm-1和Y的一个最长公共子序列及X和Yn-1的一个最长公共子序列。这两个公共子序列中**较长者即为X和Y的一个最长公共子序列。**

由此递归结构容易看到最长公共子序列问题具有子问题重叠性质。例如，在计算X和Y的最长公共子序列时，可能要计算出X和Yn-1及Xm-1和Y的最长公共子序列。而这两个子问题都包含一个公共子问题，即计算Xm-1和Yn-1的最长公共子序列。

与矩阵连乘积最优计算次序问题类似，我们来建立子问题的最优值的递归关系。

用c[i,j]记录序列Xi和Yj的最长公共子序列的长度。其中Xi=，Yj=。当i=0或j=0时，空序列是Xi和Yj的最长公共子序列，故c[i,j]=0。其他情况下，由定理可建立递归关系如下： 【i 和 j 表示索引的序号】

$$c[i,j]=\begin{cases}0 & \text{if } i=0 \text{ or } j=0,\\ c[i-1,j-1]+1 & \text{if } i,j>0 \text{ and } x_i=y_j,\\ \max(c[i,j-1],c[i-1,j]) & \text{if } i,j>0 \text{ and } x_i\neq y_j.\end{cases}$$

2.3、计算最优值

直接利用上节节末的递归式，我们将很容易就能写出一个计算c[i,j]的递归算法，但其计算时间是随输入长度指数增长的。由于在所考虑的子问题空间中，总共只有$\theta(m*n)$个不同的子问题，因此，用动态规划算法自底向上地计算最优值能提高算法的效率。

计算最长公共子序列长度的动态规划算法LCS_LENGTH(X,Y)以序列X=和Y=作为输入。输出两个数组c[0..m ,0..n]和b[1..m ,1..n]。其中：

- c[i,j]存储Xi与Yj的最长公共子序列的长度，
- b[i,j]记录指示c[i,j]的值是由哪一个子问题的解达到的，这在构造最长公共子序列时要用到。
- 最后，X和Y的最长公共子序列的长度记录于c[m,n]中。

伪代码如下：

```
Procedure LCS_LENGTH(X,Y);
begin
  m:=length[X];
  n:=length[Y];
  for i:=1 to m do c[i,0]:=0;
  for j:=1 to n do c[0,j]:=0;
  for i:=1 to m do
    for j:=1 to n do
      if x[i]=y[j] then
        begin
          c[i,j]:=c[i-1,j-1]+1;
          b[i,j]:="↖";
        end
      else if c[i-1,j]≥c[i,j-1] then
        begin
          c[i,j]:=c[i-1,j];
          b[i,j]:="↑";
```

```
        end
      else
        begin
          c[i,j]:=c[i,j-1];
          b[i,j]:="←"
        end;
  return(c,b);
end;
```

由算法LCS_LENGTH计算得到的数组b可用于快速构造序列X=和Y=的最长公共子序列。首先从b[m,n]开始，沿着其中的箭头所指的方向在数组b中搜索。

- 当b[i,j]中遇到"↖"时（*意味着xi=yi是LCS的一个元素*），表示Xi与Yj的最长公共子序列是由Xi-1与Yj-1的最长公共子序列在尾部加上xi得到的子序列；
- 当b[i,j]中遇到"↑"时，表示Xi与Yj的最长公共子序列和Xi-1与Yj的最长公共子序列相同；
- 当b[i,j]中遇到"←"时，表示Xi与Yj的最长公共子序列和Xi与Yj-1的最长公共子序列相同。

这种方法是按照反序来找LCS的每一个元素的。由于每个数组单元的计算耗费*O*(1)时间，***算法LCS_LENGTH耗时O(mn)。***

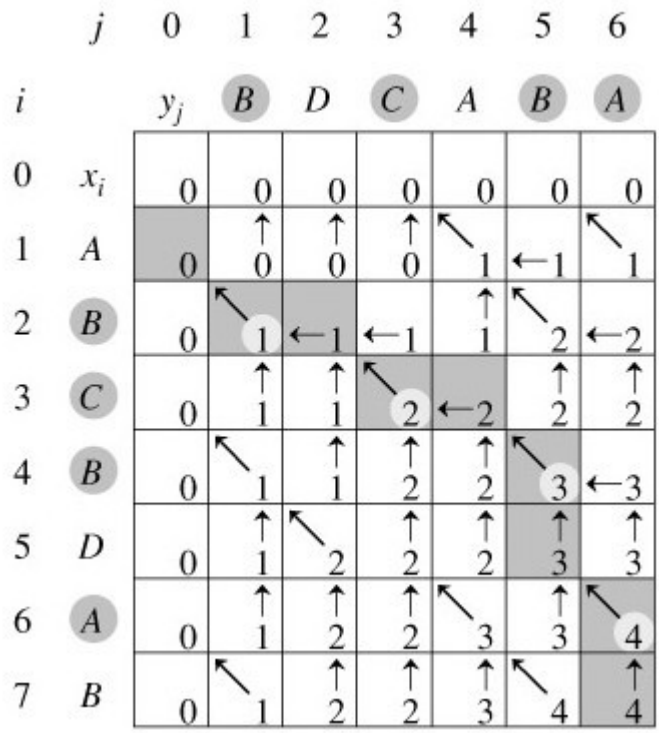
2.4、构造最长公共子序列

下面的算法`LCS(b,X,i,j)`实现根据b的内容打印出Xi与Yj的最长公共子序列。通过算法的调用`LCS(b,X,length[X],length[Y])`，便可打印出序列X和Y的最长公共子序列。

```
Procedure LCS(b,X,i,j);
begin
  if i=0 or j=0 then return;
  if b[i,j]="↖" then
    begin
      LCS(b,X,i-1,j-1);
      print(x[i]); {打印x[i]}
    end
  else if b[i,j]="↑" then LCS(b,X,i-1,j)
                     else LCS(b,X,i,j-1);
end;
```

在算法LCS中，每一次的递归调用使i或j减1，因此算法的计算时间为*O*(m+n)。

例如，设所给的两个序列为`X=<A，B，C，B，D，A，B>`和`Y=<B，D，C，A，B，A>`。由算法LCS_LENGTH和LCS计算出的结果如下图所示：



我来说明下此图（参考算法导论）。在序列X={A，B，C，B，D，A，B}和 Y={B，D，C，A，B，A}上，由LCS_LENGTH计算出的表c和b。第i行和第j列中的方块包含了c[i，j]的值以及指向b[i，j]的箭头。在c[7,6]的项4，表的右下角为X和Y的一个`LCS<B，C，B，A>`的长度。对于i，j>0，项c[i，j]仅依赖于是否有xi=yi，及项c[i-1，j]和c[i，j-1]的值，这几个项都在c[i，j]之前计算。为了**重构一个LCS的元素，从右下角开始跟踪b[i，j]的箭头**即可，这条路径标示为阴影，这条路径上的**每一个“↖”对应于一个使xi=yi为一个LCS的成员的项（高亮标示）**。

所以根据上述图所示的结果，程序将最终输出：“B C B A”，或**“B D A B” ?????**

可能还是有读者对上面的图看的不是很清楚，下面，我再通过对最大子序列，最长公共子串与最长公共子序列的比较来阐述相关问题：

- **最大子序列**：最大子序列是要找出由数组成的一维数组中**和最大的连续子序列**。比如{5,-3,4,2}的最大子序列就是{5,-3,4,2}，它的和是8,达到最大；而{5,-6,4,2}的最大子序列是{4,2}，它的和是6。你已经看出来了，找最大子序列的方法很简单，**只要前i项的和还没有小于0那么子序列就一直向后扩展，否则丢弃之前的子序列开始新的子序列，**同时我们要记下各个子序列的和，最后找到和最大的子序列。更多请参看：程序员编程艺术第七章、求连续子数组的最大和。
- **最长公共子串**：找两个字符串的最长公共子串，这个子串要求在原字符串中是**连续的。其实这又是一个序贯决策问题，可以用动态规划来求解。我们采用一个二维矩阵来记录中间的结果**。这个二维矩阵怎么构造呢？直接举个例子吧："bab"和"caba"(当然我们现在一眼就可以看出来最长公共子串是"ba"或"ab")

```
    b   a   b
c   0   0   0
a   0   1   0
b   1   0   1
a   0   1   0
```

我们看矩阵的**斜对角线最长的那个就能找出最长公共子串**。

不过在二维矩阵上找最长的由1组成的斜对角线也是件麻烦费时的事，下面改进：**当要在矩阵是填1时让它等于其左上角元素加1。**

```
    b   a   b
c   0   0   0
a   0   1   0
b   1   0   2
a   0   2   0
```

这样矩阵中的**最大元素就是最长公共子串的长度**。

在构造这个二维矩阵的过程中**由于得出矩阵的某一行后其上一行就没用了，所以实际上在程序中可以用一维数组来代替这个矩阵**。

$$c[i,j]=\begin{cases}0 & \text{if } i=0 \text{ or } j=0,\\ c[i-1,j-1]+1 & \text{if } i,j>0 \text{ and } x_i=y_j,\\ \max(c[i,j-1],c[i-1,j]) & \text{if } i,j>0 \text{ and } x_i\neq y_j.\end{cases}$$

将第三个式子改成0 即可。

- **最长公共子序列LCS问题**：最长公共子序列与最长公共子串的区别在于最长公共子序列不要求在原字符串中是连续的，比如ADE和ABCDE的最长公共子序列是ADE。

  我们用动态规划的方法来思考这个问题如是求解。首先要找到状态转移方程：

  等号约定，C1是S1的最右侧字符，C2是S2的最右侧字符，S1'是从S1中去除C1的部分，S2'是从S2中去除C2的部分。

  LCS(S1,S2)等于：

> （1）LCS（S1，S2'）
>
> （2）LCS（S1'，S2）
>
> （3）如果C1不等于C2：LCS（S1'，S2'）；如果C1等于C2：LCS（S1'，S2'）+C1；

边界终止条件：如果S1和S2都是空串，则结果也是空串。

下面我们同样要构建一个矩阵来存储动态规划过程中子问题的解。这个矩阵中的每个数字代表了该行和该列之前的LCS的长度。与上面刚刚分析出的状态转移议程相对应，矩阵中每个格子里的数字应该这么填，它等于以下3项的最大值：

> （1）上面一个格子里的数字
>
> （2）左边一个格子里的数字
>
> （3）左上角那个格子里的数字（如果C1不等于C2）；左上角那个格子里的数字+1（如果C1等于C2）

举个例子：

```
      G   C   T   A
  0   0   0   0   0
G 0   1   1   1   1
B 0   1   1   1   1
```

> T 0 1 1 2 2
>
> A 0 1 1 2 3

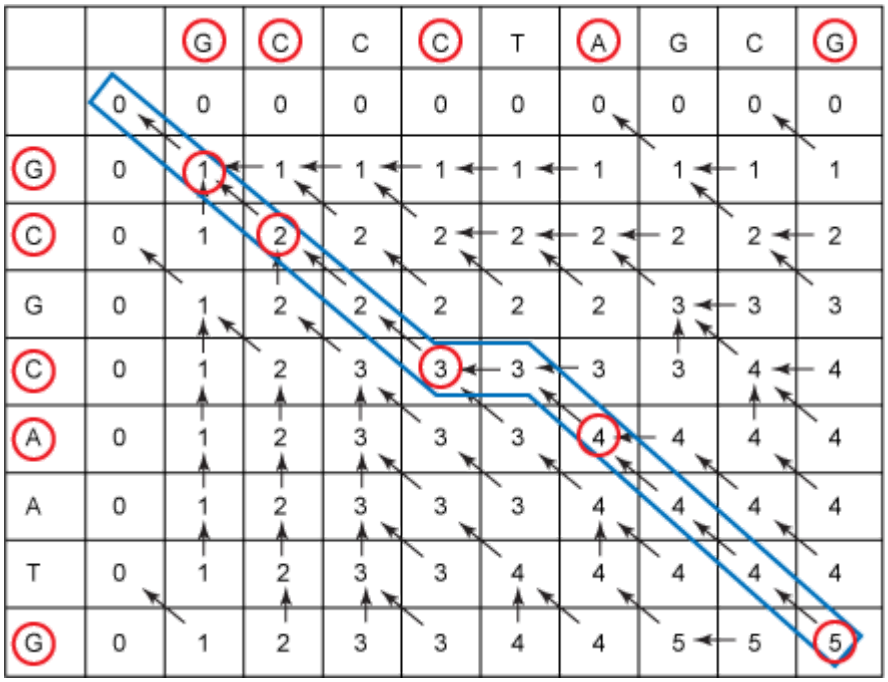
填写最后一个数字时，它应该是下面三个的最大者：

> （1）上边的数字2
>
> （2）左边的数字2
>
> （3）左上角的数字2+1=3,因为此时C1==C2

所以最终结果是3。

在填写过程中我们还是**记录下当前单元格的数字来自于哪个单元格，以方便最后我们回溯找出最长公共子串。有时候左上、左、上三者中有多个同时达到最大，那么任取其中之一，但是在整个过程中你必须遵循固定的优先标准**。在我的代码中优先级别是 **左上>左>上。**

下图给出了回溯法找出LCS的过程：



		G	C	C	C	T	A	G	C	G
	0	0	0	0	0	0	0	0	0	0
G	0	1	1	1	1	1	1	1	1	1
C	0	1	2	2	2	2	2	2	2	2
G	0	1	2	2	2	2	2	3	3	3
C	0	1	2	3	3	3	3	3	4	4
A	0	1	2	3	3	3	4	4	4	4
A	0	1	2	3	3	3	4	4	4	4
T	0	1	2	3	3	4	4	4	4	4
G	0	1	2	3	3	4	4	5	5	5

2.5、算法的改进

对于一个具体问题，按照一般的算法设计策略设计出的算法，往往在算法的时间和空间需求上还可以改进。这种改进，通常是利用具体问题的一些特殊性。

例如，在算法LCS_LENGTH和LCS中，可进一步将数组b省去。事实上，数组元素c[i,j]的值仅由c[i-1,j-1]，c[i-1,j]和c[i,j-1]三个值之一确定，而数组元素b[i,j]也只是用来指示c[i,j]究竟由哪个值确定。因此，在算法LCS中，我们可以不借助于数组b而借助于数组c本身临时判断c[i,j]的值是由c[i-1,j-1]，c[i-1,j]和c[i,j-1]中哪一个数值元素所确定，代价是$O(1)$时间。既然b对于算法LCS不是必要的，那么算法LCS_LENGTH便不必保存它。这一来，可节省$\theta(mn)$的空间，而LCS_LENGTH和LCS所需要的时间分别仍然是$O(mn)$和$O(m+n)$。不过，由于数组c仍需要$O(mn)$的空间，因此这里所作的改进，只是在空间复杂性的常数因子上的改进。

另外，如果只需要计算最长公共子序列的长度，则算法的空间需求还可大大减少。事实上，在计算c[i,j]时，只用到数组c的第i行和第i-1行。因此，只要用2行的数组空间就可以计算出最长公共子序列的长度。更进一步的分析还可将空间需求减至min(m, n)。

第三部分、最长公共子序列问题代码

ok，最后给出此面试第56题的代码，参考代码如下，请君自看：

```
// LCS.cpp : 定义控制台应用程序的入口点。
//

//copyright@zhedahht
//updated@2011.12.13 July
#include "stdafx.h"
#include "string.h"
#include <iostream>
using namespace std;

// directions of LCS generation
enum decreaseDir {kInit = 0, kLeft, kUp, kLeftUp};

void LCS_Print(int **LCS_direction,
    char* pStr1, char* pStr2,
    size_t row, size_t col);

// Get the length of two strings' LCSs, and print one of the LCSs
// Input: pStr1         - the first string
//        pStr2         - the second string
// Output: the length of two strings' LCSs
int LCS(char* pStr1, char* pStr2)
{
    if(!pStr1 || !pStr2)
        return 0;

    size_t length1 = strlen(pStr1);
    size_t length2 = strlen(pStr2);
    if(!length1 || !length2)
        return 0;

    size_t i, j;

    // initiate the length matrix
    int **LCS_length;
    LCS_length = (int**)(new int[length1]);
    for(i = 0; i < length1; ++ i)
        LCS_length[i] = (int*)new int[length2];

    for(i = 0; i < length1; ++ i)
        for(j = 0; j < length2; ++ j)
            LCS_length[i][j] = 0;

    // initiate the direction matrix
    int **LCS_direction;
    LCS_direction = (int**)(new int[length1]);
    for( i = 0; i < length1; ++ i)
        LCS_direction[i] = (int*)new int[length2];
```

```
    for(i = 0; i < length1; ++ i)
        for(j = 0; j < length2; ++ j)
            LCS_direction[i][j] = kInit;

    for(i = 0; i < length1; ++ i)
    {
        for(j = 0; j < length2; ++ j)
        {
            //之前此处的代码有问题，现在订正如下:
            if(i == 0 || j == 0)
            {
                if(pStr1[i] == pStr2[j])
                {
                    LCS_length[i][j] = 1;
                    LCS_direction[i][j] = kLeftUp;
                }
                else
                {
                    if(i > 0)
                    {
                        LCS_length[i][j] = LCS_length[i - 1][j];
                        LCS_direction[i][j] = kUp;
                    }
                    if(j > 0)
                    {
                        LCS_length[i][j] = LCS_length[i][j - 1];
                        LCS_direction[i][j] = kLeft;
                    }
                }
            }
            // a char of LCS is found,
            // it comes from the left up entry in the direction matrix
            else if(pStr1[i] == pStr2[j])
            {
                LCS_length[i][j] = LCS_length[i - 1][j - 1] + 1;
                LCS_direction[i][j] = kLeftUp;
            }
            // it comes from the up entry in the direction matrix
            else if(LCS_length[i - 1][j] > LCS_length[i][j - 1])
            {
                LCS_length[i][j] = LCS_length[i - 1][j];
                LCS_direction[i][j] = kUp;
            }
            // it comes from the left entry in the direction matrix
            else
            {
                LCS_length[i][j] = LCS_length[i][j - 1];
                LCS_direction[i][j] = kLeft;
            }
        }
    }


    LCS_Print(LCS_direction, pStr1, pStr2, length1 - 1, length2 - 1); //调用下面的
```

```
LCS_Pring 打印出所求子串。
    return LCS_length[length1 - 1][length2 - 1];                    //返回长度。
}

// Print a LCS for two strings
// Input: LCS_direction - a 2d matrix which records the direction of
//                        LCS generation
//        pStr1         - the first string
//        pStr2         - the second string
//        row           - the row index in the matrix LCS_direction
//        col           - the column index in the matrix LCS_direction
void LCS_Print(int **LCS_direction,
    char* pStr1, char* pStr2,
    size_t row, size_t col)
{
    if(pStr1 == NULL || pStr2 == NULL)
        return;

    size_t length1 = strlen(pStr1);
    size_t length2 = strlen(pStr2);

    if(length1 == 0 || length2 == 0 || !(row < length1 && col < length2))
        return;

    // kLeftUp implies a char in the LCS is found
    if(LCS_direction[row][col] == kLeftUp)
    {
        if(row > 0 && col > 0)
            LCS_Print(LCS_direction, pStr1, pStr2, row - 1, col - 1);

        // print the char
        printf("%c", pStr1[row]);
    }
    else if(LCS_direction[row][col] == kLeft)
    {
        // move to the left entry in the direction matrix
        if(col > 0)
            LCS_Print(LCS_direction, pStr1, pStr2, row, col - 1);
    }
    else if(LCS_direction[row][col] == kUp)
    {
        // move to the up entry in the direction matrix
        if(row > 0)
            LCS_Print(LCS_direction, pStr1, pStr2, row - 1, col);
    }
}

int _tmain(int argc, _TCHAR* argv[])
{
    char* pStr1="abcde";
    char* pStr2="acde";
    LCS(pStr1,pStr2);

    printf("\n");
```

```
    system("pause");
    return 0;
}
```

扩展：如果题目改成求两个字符串的最长公共子字符串，应该怎么求？子字符串的定义和子串的定义类似，但要求是连续分布在其他字符串中。

比如输入两个字符串BDCABA和ABCBDAB的最长公共字符串有BD和AB，它们的长度都是2。

第四部分、LCS问题的时间复杂度

算法导论上指出，

1. 最长公共子序列问题的一个一般的算法、**时间复杂度为O（mn）**。然后，Masek和Paterson给出了一个O(mn/lgn)时间内执行的算法，其中n<=m，而且此序列是从一个有限集合中而来。在输入序列中没有出现超过一次的特殊情况中，Szymansk说明这个问题可在O(（n+m）lg（n+m）)内解决。
2. 一篇由Gilbert和Moore撰写的关于可变长度二元编码的早期论文中有这样的应用：在所有的概率pi都是0的情况下构造最优二叉查找树，这篇论文给出一个O（n^3）时间的算法。Hu和Tucker设计了一个算法，它在所有的概率pi都是0的情况下，使用O（n）的时间和O（n）的空间，最后，Knuth把时间降到了O（nlgn）。

关于此动态规划算法更多可参考 算法导论一书第15章 动态规划问题，至于关于此面试第56题的更多，可参考我即将整理上传的答案V04版第41-60题的答案。

补充：一网友提供的关于此最长公共子序列问题的java算法源码，我自行测试了下，正确：

```
import java.util.Random;

public class LCS{
    public static void main(String[] args){

        //设置字符串长度
        int substringLength1 = 20;
        int substringLength2 = 20;  //具体大小可自行 设置

        // 随机生成字符 串
        String x = GetRandomStrings(substringLength1);
        String y = GetRandomStrings(substringLength2);

        Long startTime = System.nanoTime();
        // 构造二维数组记录子问题x[i]和y[i]的LCS的长度
        int[][] opt = new int[substringLength1 + 1][substringLength2 + 1];

        // 动态规划计算所有子问题
        for (int i = substringLength1 - 1; i >= 0; i--){
            for (int j = substringLength2 - 1; j >= 0; j--){
                if (x.charAt(i) == y.charAt(j))
                    opt[i][j] = opt[i + 1][j + 1] + 1;
//参考上文我给的公式。
                else
                    opt[i][j] = Math.max(opt[i + 1][j], opt[i][j + 1]);        //参考上
文我给的公式。
            }

        }
```

理解上段，参考上文我给的公式：

根据上述结论，可得到以下公式，

如果我们记字符串Xi和Yj的LCS的长度为c[i,j]，我们可以递归地求c[i,j]：

$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0, \\ c[i-1, j-1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j, \\ \max(c[i, j-1], c[i-1, j]) & \text{if } i, j > 0 \text{ and } x_i \neq y_j. \end{cases}$$

```
        System.out.println("substring1:"+x);
        System.out.println("substring2:"+y);
        System.out.print("LCS:");

        int i = 0, j = 0;
        while (i < substringLength1 && j < substringLength2){
            if (x.charAt(i) == y.charAt(j)){
                System.out.print(x.charAt(i));
                i++;
                j++;
            } else if (opt[i + 1][j] >= opt[i][j + 1])
                i++;
            else
                j++;
        }
        Long endTime = System.nanoTime();
        System.out.println(" Totle time is " + (endTime - startTime) + " ns");
    }

    //取得定长随机字符串
    public static String GetRandomStrings(int length){
        StringBuffer buffer = new StringBuffer("abcdefghijklmnopqrstuvwxyz");
        StringBuffer sb = new StringBuffer();
        Random r = new Random();
        int range = buffer.length();
        for (int i = 0; i < length; i++){
            sb.append(buffer.charAt(r.nextInt(range)));
        }
        return sb.toString();
    }
}
```

测试结果如下：

```
substring1:akqrshrengxqiyxuloqk
substring2:tdzbujtlqhecaqgwfzbc
LCS:qheq Totle time is 818058 ns
```