目录

湖南工商大学 算法导论 课程教案

授课题目（教学章、节或主题） **课时安排: 2学时**

第十一讲：动态规划 **授课时间 :第十一周周一第1、2节**

教学内容（包括基本内容、重点、难点）：

基本内容： （1）基本概念：动态规划，适用问题；

（2）钢条切割问题；

（3）矩阵链乘法问题;

（4）最长公共子序列问题.

教学重点、难点：重点为动态规划原理、递归算法设计

教学媒体的选择：本章使用大数据分析软件Jupyter教学，Jupyter集课件、Python程序运行、HTML网页制作、Pdf文档生成、Latex文档编译于一身，是算法导论课程教学的最佳选择。

板书设计：黑板分为上下两块，第一块基本定义，推导证明以及例子放在第二块。第一块 整个课堂不擦洗，以便学生随时看到算法流程图以及基本算法理论等内容。

课程过程设计： （1）讲解基本算法理论； （2）举例说明； （3）程序设计与编译； （4）对本课堂进行总 结、讨论； （5）布置作业与实验报告

第十一讲 动态规划

一. 基本概念

动态规划（Dynamic programming）：是一种在数学、计算机科学和经济学中使用的，通过把原问题分解为相对简单的子问题的方式求解复杂问题的方法。 动态规划算法是通过拆分问题，定义问题状态和状态之间的关系，使得问题能够以递推（或者说分治）的方式去解决。

什么是动态规划

动态规划（Dynamic Programming）对于子问题重叠的情况特别有效，因为它将子问题的解保存在表格中，当需要某个子问题的解时，直接取值即可，从而避免重复计算！

动态规划是一种灵活的方法，不存在一种万能的动态规划算法可以解决各类最优化问题（每种算法都有它的缺陷）。所以除了要对基本概念和方法正确理解外，必须具体问题具体分析处理，用灵活的方法建立数学模型，用创造性的技巧去求解。

基本策略

基本思想与分治法类似，也是将待求解的问题分解为若干个子问题（阶段），按顺序求解子阶段，前一子问题的解，为后一子问题的求解提供了有用的信息。在求解任一子问题时，列出各种可能的局部解，通过决策保留那些有可能达到最优的局部解，丢弃其他局部解。依次解决各子问题，最后一个子问题就是初始问题的解。

动态规划中的子问题往往不是相互独立的（即子问题重叠）。在求解的过程中，许多子问题的解被反复地使用。为了避免重复计算，动态规划算法采用了填表来保存子问题解的方法。

适用问题

那么什么样的问题适合用动态规划的方法来解决呢?

适合用动态规划来解决的问题，都具有下面三个特点：**最优化原理、无后效性、有重叠子问题。**

（1）最优化原理：如果问题的最优解所包含的子问题的解也是最优的，就称该问题具有最优子结构，即满足最优化原理。

（2）无后效性：即某阶段状态一旦确定，就不受这个状态以后决策的影响。也就是说，某状态以后的过程不会影响以前的状态，只与当前状态有关。

（3）有重叠子问题：即子问题之间是不独立的，一个子问题在下一阶段决策中可能被多次使用到。（该性质并不是动态规划适用的必要条件，但是如果没有这条性质，动态规划算法同其他算法相比就不具备优势。

这类问题的求解步骤通常如下：

初始状态→|决策1|→|决策2|→…→|决策n|→结束状态

（1）划分：按照问题的特征，把问题分为若干阶段。注意：划分后的阶段一定是有序的或者可排序的

（2）确定状态和状态变量：将问题发展到各个阶段时所处的各种不同的客观情况表现出来。状态的选择要满足无后续性

（3）确定决策并写出状态转移方程：状态转移就是根据上一阶段的决策和状态来导出本阶段的状态。根据相邻两个阶段状态之间的联系来确定决策方法和状态转移方程

（4）边界条件：状态转移方程是一个递推式，因此需要找到递推终止的条件

算法实现

动态规划三要素：

（1）问题的阶段

（2）每个阶段的状态

（3）相邻两个阶段之间的递推关系

整个求解过程可以用一张最优决策表来描述，最优决策表是一张二维表（行：决策阶段，列：问题的状态）表格需要填写的数据一般对应此问题的在某个阶段某个状态下的最优值（如最短路径，最长公共子序列，最大价值等），填表的过程就是根据递推关系，最后根据整个表格的数据通过简单的取舍或者运算求得问题的最优解。

例如：$f(n,m)=\max\{f(n-1,m),\ f(n-1,m-w[n])+P(n,m)\}$

二. 钢条切割

1. 钢条切割问题：

一段长为 n 的钢条和一个价格表 $p_i(i = 1, 2, 3, 4, \ldots, n)$，求切割方案，使得销售收益 R_n 最大。

长度	1	2	3	4	5	6	7	8	9	10
价格	1	5	8	9	10	17	17	20	24	30

长度为 n 的钢条有 2^{n-1} 种切割方案（因为在距离钢条左端 $i(i = 1, 2, , 3, \ldots, n-1)$ 处，我们总是可以选择切割或者不切割）

设一个最优切割方案将钢条切为 k（$1 \leq k \leq n$）段，

最优切割方案为：$n = i_1 + i_2 + i_3 + \cdots + i_k$

切割的长度分别为：i_1、i_2、i_3、$\ldots$、i_k，

得到的最大收益：$R_n = p_{i1} + p_{i2} + p_{i3} + \cdots + p_{ik}$

2. 问题分析

首先可以将钢条分割为 i 和 $n-i$ 两段，求这两段的最优收益 R_i 和 R_{n-i}（每种方案的最优收益为两段的最优收益之和）。由于无法确定哪种方案（i 取何值时）会获得最优收益，我们必须考虑所有的 i，选取其中收益最大者。若直接出售钢条会获得最大收益，可以选择不做任何切割。最优切割收益公式：$R_n = \max(p_n, R_1 + R_{n-1}, R_2 + R_{n-2}, \cdots, R_{n-1} + R_1)$。当完成首次切割后，我们可以将两段钢条（$i$ 和 $n-i$）看成两个独立的钢条切割问题实例，通过组合两个相关子问题的最优解，构成原问题的最优解。

一种相似但更为简单的递归求解方法：将长度为 n 的钢条分解为左边开始一段，以及剩余部分继续分解的结果。简化后的公式：

$$R_n = \max_{1 \leq i \leq n} (p_i + R_{n-i})$$

3. 编程

方法1：自顶向下递归实现

缺点：输入规模大时，程序运行时间会变得相当长

原因：反复求解相同的子问题

```
#伪代码
Cut-Rod(p, n)
1    if n==0
2        return 0
3    q = -∞
4    for i = 1 to n
5        q = max(q, p[i] + Cut-Rod(p, n-i))
6    return q
```

In [1]:

```python
#钢条切割问题Python代码自顶向下递归实现

#  自顶向下递归算法实现

#  R[n] = max{1≤i≤n, (pi+R[n-i])}  时间复杂度为：2^n(指数函数)

def CutRod(p, n):  # 函数返回：切割长度为 n 的钢条所得的最大收益
    if n == 0:
        return 0
    q = -1
    for i in range(1, n+1):
        q = max(q, p[i] + CutRod(p, n-i))

        '''
        tmp = p[i] + CutRod(p, n-i)
        if q < tmp:
            q = tmp
        '''
    return q
p=[0,1,5,8,9,10,17,17,20,24,30]    # 价格表，下标为对应的钢条长度，如当钢条长度
print("最大收益为：",CutRod(p,4))  #  最大收益为：  10

```

最大收益为： 10

方法2：带备忘录的自顶向下递归

特点：对每个子问题只求解一次，并将结果存储下来（随后再次遇到此问题的解，只需查找保存的结果，不必重新计算），用内存空间来节省计算时间

```
#伪代码
mem-cut-rod(p, n)
1    let r[0…n] be a new array
2    for i=0 to n
3        r[i] = -∞
4    return mem-cut-rod-aux(p, n, r)

mem-cut-rod-aux(p, n, r)
1    if r[n] >= 0
2        return r[n]
3    if n == 0
4        q = 0
5    else
6        q = -∞
7        for i=1 to n
8            q = max(q, p[i] + mem-cut-rod-aux(p, n-i, r))
9    r[n] = q
10   return q
```

主过程 mem-cut-rod(p, n)将辅助数组 $r[0...n]$ 初始化为负无穷，然后调用辅助过程mem-cut-rod-aux(最初cut-rob引入备忘机制的版本)。伪代码解读:

首先检查所需值是否已知（第1行）;
如果是，则第2行直接返回保存的值;
否则第3~8行用通常方法计算所需值q;
第9行将q存入r[n]中;
第10行返回;

In [2]:

```
#钢条切割Python代码2
# 带备忘录的自顶向下法 -- 每个子问题只求解一次，并将之存放在数组 r 中，以备用

def MemorizedCutRod(p, n):
    r=[-1]*(n+1)                          #  数组初始化
    def MemorizedCutRodAux(p, n, r):
        if r[n] >= 0:
            return r[n]
        q = -1
        if n == 0:
            q = 0
        else:
            for i in range(1, n + 1):
                q = max(q, p[i] + MemorizedCutRodAux(p, n - i, r))
        r[n] = q
        return q
    return MemorizedCutRodAux(p, n, r),r
print("最大收益为：",MemorizedCutRod(p, 5))  # 最大收益为：  (13, [0, 1, 5, 8,
```

最大收益为： (13, [0, 1, 5, 8, 10, 13])

方法3：自底向上法（动态规划）

特点：任何子问题的求解都依赖于 更小子问题 的求解。对子问题规模进行排序，按由小到大的问题进行求解。当求解某个子问题时，他所依赖的更小的子问题都已求解完毕，结果已经保存。

优点：时间复杂度在三种方法中最好

```
bottom-up-cut-rod(p, n)
1    let r[0…n] be a new array
2    r[0] = 0
3    for j=1 to n
4        q = -∞
5        for i=1 to j
6            q = max(q, p[i] + r[j-i])
7        r[j] = q
8    return r[n]
```

自底向上版本采用子问题的自然顺序：若i<j，则规模为i的子问题比规模为j的子问题“更小”。因此，过程依次求解规模为j=0,1,…,n的子问题。伪代码详解：

第1行创建一个新数组r[0...n]来保存子问题的解；
第2行将r[0]初始化为0，因为长度为0的钢条没有收益；
第3~6行对j=1...n按升序求解每个规模为j的子问题。求解方法与cut-rod采用的方法相同，只是现在直接访问数组元素r[j-i]来获取规模为j-i的子问题的解，而不必进行递归调用；
第7行将规模为 j 的子问题的解存入r[j]；
第8行返回r[n],即最优解

bottom-up-cut-rod 主体是嵌套的双层循环，内层循环（5~6行）的迭代次数构成一个等差数列和，不难分析时间为 n^2 . mem-cut-rod 运行时间也是 n^2 ，其分析略难：当求解一个之前已经计算出结果的子问题时，递归调用会立即返回，即每个子问题只求解一次，而它求解了规模为 $0，1，。。。，n$ 的子问题；为求解规模为n的子问题，第6~7行的循环会迭代n次；因此进行的所有递归调用执行此for循环的迭代次数也是一个等差数列，其和也是 n^2 。

In [5]:

```
#钢条切割Python代码3
# 自底向上

def BottomUpCutRod(p, n):
    r = [0]*(n+1)
    for i in range(1, n+1):
        if n == 0:
            return 0
        q =0
        for j in range(1, i+1):
            q = max(q, p[j]+r[i-j])
            r[i] = q
    return r[n],r
p=[0,1,5,8,9,10,17,17,20,24,30]
print(BottomUpCutRod(p, 10))   #  (30, [0, 1, 5, 8, 10, 13, 17, 18, 22, 25, 30
```

```
(30, [0, 1, 5, 8, 10, 13, 17, 18, 22, 25, 30])
```

三. 矩阵链乘法

矩阵连乘法问题：

(1)问题描述

给定n个矩阵的链 $< A_1, A_2, \dots, A_n >$，其中 $i = 1, 2, \cdots, n$，矩阵 A_i 的维数为 $p_{i-1} \times p_i$。求一个完全“括号化方案”，使得计算乘积

$$A_1 A_2 \cdots A_n$$

所需的标量乘法次数最小。例如，如果矩阵链为 $< A_1, A_2, A_3, A_4 >$，则有共5种完全括号化的矩阵乘积链：

$$(A_1(A_2(A_3A_4)))$$
$$(A_1((A_2A_3)A_4))$$
$$((A_1A_2)(A_3A_4))$$
$$((A_1(A_2A_3))A_4)$$
$$(((A_1A_2)A_3)A_4)$$

(2)最优括号化方案的结构特征

用记号 $A_{i,j}$ 表示 $A_i A_{i+1} \cdots A_j$ 通过加括号后得到的一个最优计算模式，且恰好在 A_k 与 A_{k+1} 之间分开。则“前缀”子链 $A_i A_{i+1} \cdots A_k$ 必是一个最优的括号化子方案，记为 $A_{i,k}$；同理“后缀”子链 $A_{k+1} A_{k+2} \cdots A_j$ 也必是一个最优的括号化子方案，记为 $A_{k+1,j}$。

(3)一个递归求解的方案

对于矩阵链乘法问题，我们将所有对于 $1 \le i \le j \le n$ 确定 $A_i A_{i+1} \cdots A_j$ 的最小代价括号方案作为子问题。令 $m[i, j]$ 表示计算矩阵 $A_{i,j}$ 所需要的标量乘法的次数最小值，则最优解就是计算 $A_{i \dots n}$ 所需的最低代价就是 $m[1, n]$

递归定义 $m[i\ ,\ j]$:

- 对于 $i = j$ 的情况下，显然有 $m = 0$，不需要做任何标量乘法运算。所以，对于所有的 $i = 1、2.....n$，$m[i\ ,\ i] = 0.$
- 当 $i < j$ 的情况，就按照最优括号化方案的结构特征进行计算 $m[i, j]$。假设最优括号化方案的分割点在矩阵 A_k 和 A_{k+1} 之间，那么 m 的值就是 $A_{i \dots k}$ 和 $A_{k+1 \dots j}$ 的代价加上两者量程的代价的最小值。即
$$m[i, j] = m[i, k] + m[k + 1, j] + p_{i-1} p_k p_j$$
该公式的假设是最优分割点是已知的，但是实际上不知道。然而，k 只有 $j - i$ 中情况取值。由于最优分割点k必定在 i~j 内取得，只需要检查所有可能的情况，找到最优解即可。

可以得出一个递归公式

$$m[i, j] = \begin{cases} 0 & i = j \\ \min_{i \le k \le j} \{m[i, k] + m[k + 1, j] + p_{i-1} p_k p_j\} & i < j \end{cases}$$

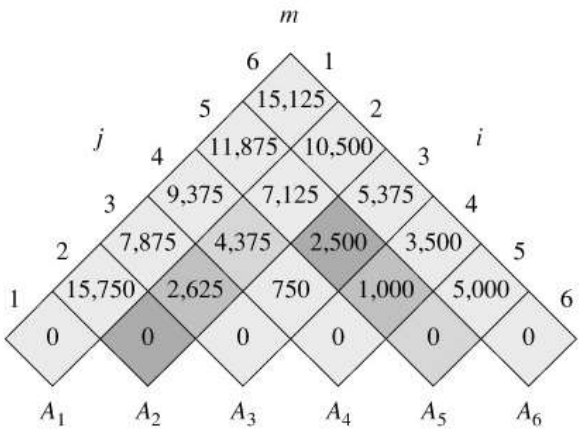
m 只是给出了子问题最优解的代价，但是并未给出构造最优解的足够信息(即分割点的位置信息)。所以，在此基础之上，我们使用一个二维数组 $s[i,j]$ 来保存 $A_i A_{i+1} \cdots A_j$ 的分割点位置k。

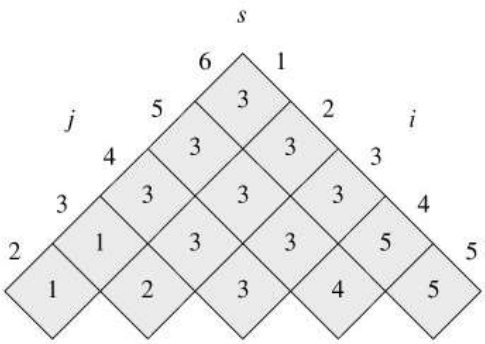
(4)采用自底向上表格方法来代替上述递归公式算法来计算最优代价。

过程中假定矩阵 A_i 的规模为 $p_{i-1} \times p_i$，输入是一个序列 $p =< p_0, p_1, \cdots, p_n >$，长度为 $p.length = n + 1$.其中使用一个辅助表m 来记录代价 $m[i,j]$，另一个表 s 来记录分割点的位置信息，以便于构造出最优解。

```
MAXTRIX_CHAIN_ORDER(p)
 2   n = length[p]-1;
 3   for i=1 to n
 4       do m[i][i] = 0;
 5   for t = 2 to n  //t is the chain length
 6        do for i=1 to n-t+1
 7                       j=i+t-1;
 8                       m[i][j] = MAXLIMIT;
 9                       for k=i to j-1
10                              q = m[i][k] + m[k+1][i] + p_i-1p_kp_j;
11                              if q < m[i][j]
12                                 then m[i][j] = q;
13                                      s[i][j] = k;
14   return m and s;
```

MATRIX_CHAIN_ORDER具有循环嵌套，深度为3层，运行时间为 $O(n^3)$。如果采用递归进行实现，则需要指数级时间$\Omega(2^n)$，因为中间有些重复计算.





(5)构造一个最优解

第三步中已经计算出来最小代价，并保存了相关的记录信息。因此只需对s表格进行递归调用展开既可以得到一个最优解。书中给出了伪代码，摘录如下：

```
PRINT_OPTIMAL_PARENS(s,i,j)
    if i== j
        then print "Ai"
    else
        print "(";
        PRINT_OPTIMAL_PARENS(s,i,s[i][j]);
        PRINT_OPTIMAL_PARENS(s,s[i][j]+1,j);
        print")";
```

矩阵链 Python 代码：

In [6]:

```
p = [30, 35, 15, 5, 10, 20, 25]
def matrix_chain_order(p):
    n = len(p) - 1    # 矩阵个数
    m = [[0 for i in range(n)] for j in range(n)]
    s = [[0 for i in range(n)] for j in range(n)] # 用来记录最优解的括号位置
    for l in range(1, n): # 控制列，从左往右
        for i in range(l-1, -1, -1):  # 控制行,从下往上
            m[i][l] = float('inf') # 保存要填充格子的最优值
            for k in range(i, l):  # 控制分割点
                q = m[i][k] + m[k+1][l] + p[i]*p[k+1]*p[l+1]
                if q < m[i][l]:
                    m[i][l] = q
                    s[i][l] = k
    return m, s

def print_option_parens(s, i, j):
    if i == j:
        print('A'+str(i+1), end='')
    else:
        print('(', end='')
        print_option_parens(s, i, s[i][j])
        print_option_parens(s, s[i][j]+1, j)
        print(')', end='')

r, s = matrix_chain_order(p)
print_option_parens(s, 0, 5)
```

```
((A1(A2A3))((A4A5)A6))
```

四. 最长公共子序列

1. 最长公共子序列(LCS)的定义

最长公共子序列，即Longest Common Subsequence，LCS。一个序列S任意删除若干个字符得到新序列T，则T叫做S的子序列； 两个序列X和Y的公共子序列中，长度最长的那个，定义为X和Y的最长公共子序列。

- 字符串13455 与245576的最长公共子序列为455;
- 字符串acdfg与adfc的最长公共子序列为adf

注意区别最长公共子串(Longest Common Substring)最长公共字串要求连续

LCS的意义

求两个序列中最长的公共子序列算法，广泛的应用在图形相似处理、媒体流的相似比较、计算生物学方面。生物学家常常利用该算法进行基因序列比对，由此推测序列的结构、功能和演化过程。

LCS可以描述两段文字之间的“相似度”，即它们的雷同程度，从而能够用来辨别抄袭。另一方面，对一段文字进行修改之后，计算改动前后文字的最长公共子序列，将除此子序列外的部分提取出来，这种方法判断修改的部分，往往十分准确。简而言之，百度知道、百度百科都用得上。

LCS的记号

- 字符串 X，长度为 m，从1开始数;
- 字符串 Y，长度为 n，从1开始数;
- $X_i =< x_1, x_2, \cdots, x_i >$ 即 X 序列的前 i 个字符 $(1 \le i \le m)$ (X_i 不妨读作“字符串 X 的 i 前缀”)
- $Y_j =< y_1, y_2, \cdots, y_j >$即 Y 序列的前 j 个字符 $(1 \le j \le n)$ (字符串 Y 的 j 前缀);
- $LCS(X,Y)$ 为字符串 X 和 Y 的最长公共子序列，即为 $Z =< z_1, z_2, \cdots, z_k >$。

注：不严格的表述。事实上，X和Y的可能存在多个子串，长度相同并且最大，因此，$LCS(X,Y)$ 严格的说，是个字符串集合。即：$Z \in LCS(X,Y)$.

LCS解法的探索：

- 若 $x_m = y_n$ (最后一个字符相同)，则：X_m 与 Y_n 的最长公共子序列 Z_k 的最后一个字符必定为 x_m，即 $z_k = x_m = y_n$.
$$LCS(X_m, Y_n) = LCS(X_{m-1}, Y_{n-1}) + x_m$$
- 若 $x_m \neq y_n$，则:
$$LCS(X_m, Y_n) = \max\{LCS(X_{m-1}, Y_n),\ LCS(X_m, Y_{n-1})\}$$

算法中的数据结构：长度数组

使用二维数组 $C[m,n]$

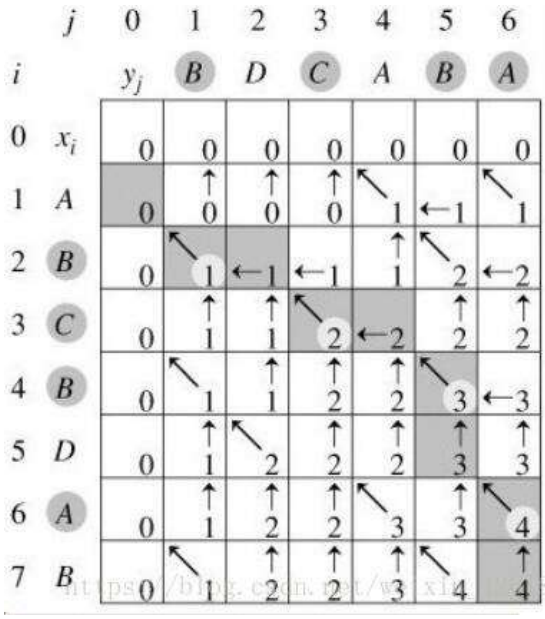
- $c[i,j]$ 记录序列 X_i 和 Y_j 的最长公共子序列的长度;
- 当 $i=0$ 或 $j=0$ 时，空序列是 X_i 和 Y_j 的最长公共子序列，故 $c[i,j]=0$.

2. LCS 的例子

求 X=< A，B，C，B，D，A，B > 和 Y=< B，D，C，A，B，A >的最长公共子序列。

```
#LCS 伪代码
LCS_LENGTH(X,Y)
    m = length(X);
    n = length(Y);
    for i = 1 to m
      c[i][0] = 0;
    for j=1 to n
      c[0][j] = 0;
    for i=1 to m
       for j=1 to n
           if x[i] = y[j]
              then c[i][j] = c[i-1][j-1]+1;
                   b[i][j] = '\';
           else if c[i-1][j] >= c[i][j-1]
                  then c[i][j] = c[i-1][j];
                       b[i][j] = '|';
                  else
                       c[i][j] = c[i][j-1];
                       b[i][j] = '-';
return c and b
```

由伪代码可以看出LCS_LENGTH运行时间为 $\Theta(mn)$。



构造一个LCS

根据第三步中保存的表 b 构建一个LCS序列。从 $b[m][n]$ 开始，当遇到'\'时，表示$x_i = y_j$，是LCS中的一个元素。通过递归即可求出LCS的序列元素。书中给出了伪代码如下所示：

```
PRINT_LCS(b,X,i,j)
    if i==0 or j==0
        then return
    if b[i][j] == '\'
        then PRINT_LCS(b,X,i-1,j-1)
             print X[i]
     else if b[i][j] == '|'
                then PRINT_LCS(b,X,i-1,j)
             else PRINT_LSC(b,X,i,j-1)
```

Python代码1实现：

In [9]:

```
def LCS(s1, s2):
    size1 = len(s1) + 1
    size2 = len(s2) + 1
    # 程序多加一行，一列，方便后面代码编写
    chess = [[["", 0] for j in list(range(size2))] for i in list(range(size1)
    for i in list(range(1, size1)):
        chess[i][0][0] = s1[i - 1]
    for j in list(range(1, size2)):
        chess[0][j][0] = s2[j - 1]
    print("初始化数据：")
    print(chess)
    for i in list(range(1, size1)):
        for j in list(range(1, size2)):
            if s1[i - 1] == s2[j - 1]:
                chess[i][j] = ['↖', chess[i - 1][j - 1][1] + 1]
            elif chess[i][j - 1][1] > chess[i - 1][j][1]:
                chess[i][j] = ['←', chess[i][j - 1][1]]
            else:
                chess[i][j] = ['↑', chess[i - 1][j][1]]
    print("计算结果：")
    print(chess)
    i = size1 - 1
    j = size2 - 1
    s3 = []
    while i > 0 and j > 0:
        if chess[i][j][0] == '↖':
            s3.append(chess[i][0][0])
            i -= 1
            j -= 1
        if chess[i][j][0] == '←':
            j -= 1
        if chess[i][j][0] == '↑':
            i -= 1
    s3.reverse()
    print("最长公共子序列：%s" % ''.join(s3))

LCS("ABCBDAB", "BDCABA")
```

初始化数据：
[[['', 0], ['B', 0], ['D', 0], ['C', 0], ['A', 0], ['B', 0],
['A', 0]], [['A', 0], ['', 0], ['', 0], ['', 0], ['', 0], ['',
0], ['', 0]], [['B', 0], ['', 0], ['', 0], ['', 0], ['', 0],
['', 0], ['', 0]], [['C', 0], ['', 0], ['', 0], ['', 0], ['',

```
0], ['', 0], ['', 0]], [['B', 0], ['', 0], ['', 0], ['', 0],
['', 0], ['', 0], ['', 0]], [['D', 0], ['', 0], ['', 0], ['',
0], ['', 0], ['', 0], ['', 0]], [['A', 0], ['', 0], ['', 0],
['', 0], ['', 0], ['', 0], ['', 0]], [['B', 0], ['', 0], ['',
0], ['', 0], ['', 0], ['', 0], ['', 0]]]
```

计算结果：

```
[[['', 0], ['B', 0], ['D', 0], ['C', 0], ['A', 0], ['B', 0],
['A', 0]], [['A', 0], ['↑', 0], ['↑', 0], ['↑', 0], ['↖',
1], ['←', 1], ['↖', 1]], [['B', 0], ['↖', 1], ['←', 1],
['←', 1], ['↑', 1], ['↖', 2], ['←', 2]], [['C', 0], ['↑',
1], ['↑', 1], ['↖', 2], ['←', 2], ['↑', 2], ['↑', 2]],
[['B', 0], ['↖', 1], ['↑', 1], ['↑', 2], ['↑', 2], ['↖',
3], ['←', 3]], [['D', 0], ['↑', 1], ['↖', 2], ['↑', 2],
['↑', 2], ['↑', 3], ['↑', 3]], [['A', 0], ['↑', 1], ['↑',
2], ['↑', 2], ['↖', 3], ['↑', 3], ['↖', 4]], [['B', 0],
['↖', 1], ['↑', 2], ['↑', 2], ['↑', 3], ['↖', 4], ['↑',
4]]]
```

最长公共子序列：BCBA

Python代码2示例： 获取最长公共子序列，可辨别显示多组相同长度的不同最长公共子序列

In [17]:

```
class LCS():
    # 读入待匹配的两个字符串
    def input(self, x, y):
        if type(x) != str or type(y) != str:
            print('input error')
            return None
        self.x = x
        self.y = y

    # 生成最长公共子序列矩阵
    def Compute_LCS(self):
        xlength = len(self.x)
        ylength = len(self.y)
        self.direction_list = [None] * xlength          #这个二维列表存着回溯方
        for i in range(xlength):
            self.direction_list[i] = [None] * ylength
        self.lcslength_list = [None] * (xlength + 1)          #这个二维列表存着
        for j in range(xlength + 1):
            self.lcslength_list[j] = [None] * (ylength + 1)
        for i in range(0, xlength + 1):                  #二维列表第一列设置为0
            self.lcslength_list[i][0] = 0
        for j in range(0, ylength + 1):                  #二维列表第一行设置为0
            self.lcslength_list[0][j] = 0
        #下面是进行回溯方向和长度表的赋值
        for i in range(1, xlength + 1):
            for j in range(1, ylength + 1):
                if self.x[i - 1] == self.y[j - 1]:
                    self.lcslength_list[i][j] = self.lcslength_list[i - 1][j -
                    self.direction_list[i - 1][j - 1] = 0      #左上
                elif self.lcslength_list[i - 1][j] > self.lcslength_list[i][j
                    self.lcslength_list[i][j] = self.lcslength_list[i - 1][j]
                    self.direction_list[i - 1][j - 1] = 1      #上
                elif self.lcslength_list[i - 1][j] < self.lcslength_list[i][j
                    self.lcslength_list[i][j] = self.lcslength_list[i][j - 1]
                    self.direction_list[i - 1][j - 1] = -1   #左
                else:
                    self.lcslength_list[i][j] = self.lcslength_list[i - 1][j]
                    self.direction_list[i - 1][j - 1] = 2    #左或上
        self.lcslength = self.lcslength_list[-1][-1]
        # print(self.direction_list)
        # print(self.lcslength_list)
        return self.direction_list, self.lcslength_list

    # 生成最长公共子序列
    def printLCS(self, curlen, i, j, s):
        if i == 0 or j == 0:
```

```
            return None
        if self.direction_list[i - 1][j - 1] == 0:
            if curlen == self.lcslength:
                s += self.x[i - 1]
                for i in range(len(s)-1, -1, -1):
                    print(s[i],end="")
                print('\n')
            elif curlen < self.lcslength:
                s += self.x[i - 1]
                self.printLCS(curlen + 1, i - 1, j - 1, s)
        elif self.direction_list[i - 1][j - 1] == 1:
            self.printLCS(curlen, i - 1, j, s)
        elif self.direction_list[i - 1][j - 1] == -1:
            self.printLCS(curlen, i, j - 1, s)
        else:
            self.printLCS(curlen, i - 1, j, s)
            self.printLCS(curlen, i, j - 1, s)

    def returnLCS(self):                #回溯的入口
        self.printLCS(1, len(self.x), len(self.y), '')

if __name__ == '__main__':
    str_a = 'abcbdab'
    str_b = 'bdcaba'
    p = LCS()                           #实例化类
    p.input(str_a, str_b)               #读入待匹配的两个字符串
    p.Compute_LCS()                     #生成最长公共子序列矩阵
    p.returnLCS()                       #生成最长公共子序列

```

bcba

bcab

bdab

引用及参考：

[1]《Python数据结构与算法分析》布拉德利.米勒等著，人民邮电出版社，2019年9月.

[2] https://blog.csdn.net/weixin_42018258/article/details/80670067 (https://blog.csdn.net/weixin_42018258/article/details/80670067)

课后练习

1. 写出钢条切割问题的完整Python代码或 C语言代码。

2. 写出矩阵链乘法问题的完整Python代码或 C语言代码，求矩阵规模序列<5,10,3,12,5,50,6>的矩阵链的最优括号化方案。

3. 写出最长公共子序列问题的完整Python代码或 C语言代码，求<1,0,0,1,0,1,0,1>和<0,1,0,1,1,0,1,1,0>的所有LCS。

讨论、思考题、作业：

参考资料（含参考书、文献等）：算法导论. Thomas H. Cormen等，机械工业出版社，2017.

授课类型（请打√）：理论课□ 讨论课□ 实验课□ 练习课□ 其他□

教学过程设计（请打√）：复习□ 授新课□ 安排讨论□ 布置作业□

教学方式（请打√）：讲授□ 讨论□ 示教□ 指导□ 其他□

教学资源（请打√）：多媒体□ 模型□ 实物□ 挂图□ 音像□ 其他□

填表说明：1、每项页面大小可自行添减；2、教学内容与讨论、思考题、作业部分可合二为一。

In [ ]:

```
1
```