从若干实例探讨算法的思考模式

七月算法 邹博

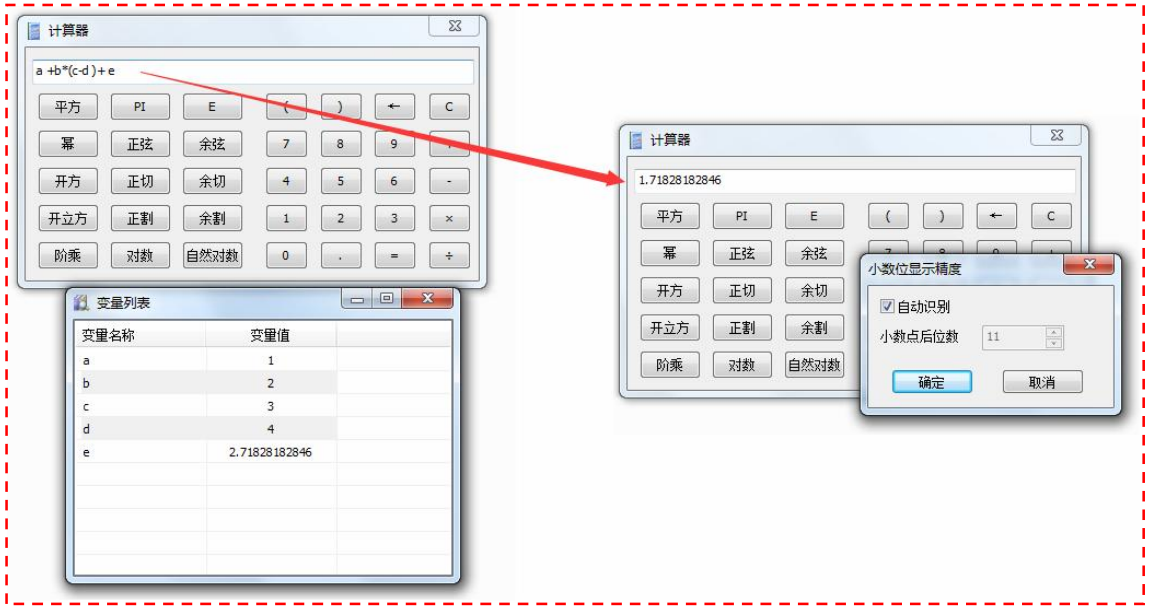
2015年4月7日

总论

- 算法包罗万象
  - 推理、逻辑、“机智”
  - 演绎、归纳、类别
- 算法是脑力的游戏
- 合理运用算法，能够获得更高的效率

字符串表达式的计算

- a+b*(c-d)+e
- 朴素算法
- 逆波兰表达式
  - 栈的典型应用



最大连续子数组

- 给定一个数组A[0,…,n-1]，求A的连续子数组，使得该子数组的和最大。
- 例如
  - 数组： 1, -2, 3, 10, -4, 7, 2, -5，
  - 最大子数组：3, 10, -4, 7, 2

最大连续子数组的解法

- 暴力法
- 分治法
- 分析法
- 动态规划法

暴力法

- 直接求解A[i,…j]的值：
- $0 \leqslant i < n$
- $i \leqslant j < n$
- i,i+1…,j-1,j的最大长度为n
- 因此：时间复杂度O(n^3)

暴力法Code

```
int MaxSubArray(int* A, int n)
{
    int maxSum = a[0];
    int currSum;
    for (int i = 0; i < n; i++)
    {
        for (int j = i; j < n; j++)
        {
            currSum = 0;
            for (int k = i; k <= j; k++)
            {
                currSum += A[k];
            }
            if (currSum > maxSum)
                maxSum = currSum;
        }
    }
    return maxSum;
}
```

分治法

- 将数组从中间分开，那么最大子数组要么完全在左半边数组，要么完全在右半边数组，要么跨立在分界点上。
- 完全在左数组、右数组递归解决。
- 跨立在分界点上：实际上是左数组的最大后缀和右数组的最大前缀的和。因此，从分界点向前扫，向后扫即可。

分治Code

```
double MaxAddSub(double *a, int from, int to)
{
    if(to == from)
        return a[from];

    int middle = (from+to)/2;
    double m1 = MaxAddSub(from, middle);
    double m2 = MaxAddSub(middle+1, to);

    int i,left = a[middle], now = a[middle];
    for(i = middle-1; i >= from; --i)
    {
        now += a[i];
        left = max(now, left);
    }
    int right = a[middle+1];
    now = a[middle+1];
    for(i = middle+2; i <= to; ++i)
    {
        now += a[i];
        right = max(now, right);
    }
    double m3 = left+right;
    return max(m1,m2,m3);
}
```

分治法算法复杂度分析

- 算法的递推关系：T(n)=2*T(n/2) + cn
  - c为常数
- 若n=2^k,则有，

  T(n) =2(2T(n/4)+cn/2)+cn

  = 4T(n/4) + 2cn =4(2T(n/8) +cn/4) + 2cn

  = …

  =2kT(1)+kcn

  =an+cnlogn　//k=logn//
- 若2^k<n<2^(k+1)，则T(2^k) ≤T(n)≤T(2^(k+1))
- 所以得：T(n) = O(nlogn)

分析法（逻辑推理的算法应用）

- 前缀和p[i] = a[0] + a[1] + ...+a[i]
- s[i,j]=p[j]-p[i-1](定义p[-1] = 0)
- 算法过程
- 1. 求i前缀p[i]：
  - 遍历i：0≤i ≤n-1
  - p[i]=p[i-1]+A[i]
- 2. 计算p[i]-p[j]
  - 遍历i：0≤i ≤n-1，最小值m的初值取0(P[-1]=0)，然后遍历p[0…i-1]，更新m
  - p[i]-m即为以A[i]结尾的数组中最大的子数组
- 3. 在第2步中，可顺手记录p[i]-m的最大值。

- 1、2步都是线性的，因此，时间复杂度O(n)。

进一步的分析

- 记S[i]为以A[i]结尾的数组中和最大的子数组
- 则：S[i+1] = max(S[i]+A[i+1], A[i+1])
- S[0]=A[0]
- 遍历i：0≤i ≤n-1
- 动态规划：最优子问题
- 时间复杂度：O(n)

动态规划伪代码

```
result = a[0]
sum = a[0]

for i: 1 to LENGTH[a]-1
   if sum > 0
      sum += a[i]
   else
      sum = a[i]

   if sum > result
      result = sum

return result
```

查找旋转数组的最小值

- Suppose a sorted array is rotated at some pivot unknown to you beforehand.(i.e., 0 1 2 4 5 6 7 might become 4 5 6 7 0 1 2).Find the minimum element.You may assume no duplicate exists in the array.
- 假定一个排序数组以某个未知元素为支点做了旋转，如：原数组0 1 2 4 5 6 7旋转后得到4 5 6 7 0 1 2。请找出旋转后数组的最小值。假定数组中没有重复数字。

分析

- 旋转之后的数组实际上可以划分成两个有序的子数组：前面子数组的大小都大于后面子数组中的元素；
  - 4 5 6 7 0 1 2
  - 注意到实际上最小的元素就是两个子数组的分界线。

思路 4 5 6 7 0 1 2

- 用两个指针low，high分别指向数组的第一个元素和最后一个元素。如果是正常的排序数组（元素间不重复），第一个元素肯定小于最后一个元素。
- 计算中间位置mid = (low+high)/2;
  - 若：A[mid]>A[low]，则A[low,low+1….mid-1,mid]是递增序列，最小元素位于子数组A[mid+1,mid+2,…high]中。因此，做赋值low=mid+1；
  - 若：A[mid]<A[low]，则A[low,low+1….mid-1,mid]不是递增序列，即：中间元素该子数组中，做赋值high=mid。
  - 注：对偶地，若考察A[mid]与A[high]的关系，能够得到相似的结论。

代码

```
int FindMin(int* num, int size)
{
    int low = 0;
    int high = size - 1;
    int mid;
    while(low < high)
    {
        mid = (high + low) / 2;
        if (num[mid] < num[high])    //最小值在左半部分
            high = mid;
        else if (num[mid] > num[high])  //最小值在右半部分
            low = mid + 1;
    }
    return num[low];
}
```

零子数组

- 求对于长度为N的数组A，求子数组的和接近0的子数组，要求时间复杂度O(NlogN)。

算法流程

- 申请同样长度的空间sum[0…N-1]，sum[i]是A的前i项和。
  - Trick：定义sum[-1] = 0
- 显然有：$\sum_{k=i}^{j} A_k = sum(j) - sum(i-1)$
- 算法：
- 对sum[0…N-1]排序，然后计算sum相邻元素的差，最小值记为min1。
  - min1：在A中任意取两个集合，各自元素的和求差的最小值
- 因为sum[-1]=0，sum[0…N-1]的最小值记为min2。
  - min2：在A中任意取一个集合求元素和的最小值
- min1和min2的更小者，即为所求。

要说明的两个问题

- sum本身的计算和相邻元素差的计算，都是O(N)，sum的排序是O(NlogN)，因此，总时间复杂度：O(NlogN)
- 强调：除了计算sum相邻元素的差的最小值，别忘了sum自身的最小值。
  - 一个对应A[i…j]，一个对应A[0…j]

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。

暴力求解：穷举法

- 假定字符串X，Y的长度分别为m，n；
- X的一个子序列即下标序列{1, 2, …, m}的严格递增子序列，因此，X共有2^m个不同子序列；同理，Y有2^n个不同子序列，从而穷举搜索法需要指数时间$O(2^m \cdot 2^n)$；
- 对X的每一个子序列，检查它是否也是Y的子序列，从而确定它是否为X和Y的公共子序列，并且在检查过程中选出最长的公共子序列；
- 显然，不可取。

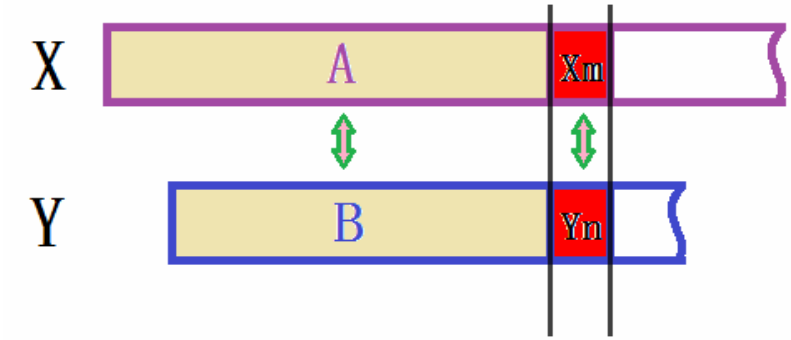
LCS的记号

- 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；
- Xi=<x1，…，xi>即X序列的前i个字符 (1≤i≤m)(Xi不妨读作“字符串X的i前缀”)
- Yj=<y1，…，yj>即Y序列的前j个字符 (1≤j≤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=<z1，…，zk>。
  - 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∈ LCS(X，Y)．

LCS解法的探索： $x_m=y_n$

- 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$x_m(=y_n)$。
  - $z_k=x_m=y_n$
  - $LCS(X_m , Y_n) = LCS(X_{m-1} , Y_{n-1}) + x_m$

结尾字符相等，则$LCS(X_m,Y_n)=LCS(X_{m-1},Y_{n-1})+x_m$



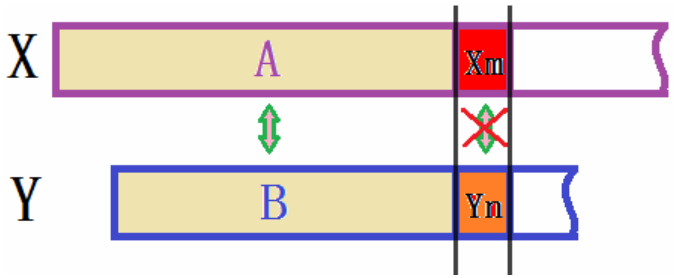
- 记$LCS(X_m,Y_n)=W+x_m$，则W是X_{m-1}的子序列；同理，W是Y_{n-1}的子序列；因此，W是X_{m-1}和Y_{n-1}的公共子序列。
  - 反证：若W不是X_{m-1}和Y_{n-1}的最长公共子序列，不妨记$LCS(X_{m-1},Y_{n-1})=W'$，且$|W'|>|W|$；那么，将W换成W'，得到更长的$LCS(X_m,Y_n)=W'x_m$，与题设矛盾。

举例： $x_m=y_n$

	1	2	3	4	5	6	7
X	B	D	C	A	B	A	
Y	A	B	C	B	D	A	B

- 对于上面的字符串X和Y：
- $x_3=y_3$=‘C’， 则： LCS(BDC,ABC)=LCS(BD,AB)+‘C’
- $x_5=y_4$=‘B’， 则： LCS(BDCAB,ABCB)=LCS(BDCA,ABC)+‘B’

LCS解法的探索：$x_m \neq y_n$



- 若$x_m \neq y_n$，则：要么$LCS(X_m,Y_n)=LCS(X_{m-1}, Y_n)$，要么$LCS(X_m,Y_n)=LCS(X_m, Y_{n-1})$。
- 证明：令$Z_k=LCS(X_m,Y_n)$；由于$x_m \neq y_n$则$z_k \neq x_m$与$z_k \neq y_n$至少有一个必然成立，不妨假定$z_k \neq x_m$（$z_k \neq y_n$的分析与之类似）。
  - 因为$z_k \neq x_m$，则最长公共子序列Z_k是X_{m-1}和Y_n得到的，即：$Z_k=LCS(X_{m-1},Y_n)$
  - 同理，若$z_k \neq y_n$，则$Z_k=LCS(X_m,Y_{n-1})$
- 即，若$x_m \neq y_n$，则：
  - $LCS(X_m,Y_n)= \max\{LCS(X_{m-1},Y_n),LCS(X_m,Y_{n-1})\}$

举例： $x_m \neq y_n$

	1	2	3	4	5	6	7
X	B	D	C	A	B	A	
Y	A	B	C	B	D	A	B

- 对于字符串X和Y：
- $x_2 \neq y_2$， 则： LCS(BD,AB)=max{ LCS(BD,A), LCS(B, AB) }
- $x_4 \neq y_5$， 则： LCS(BDCA,ABCBD)=

  max{ LCS(BDCA,ABCB), LCS(BDC,ABCBD) }

LCS分析总结

$$LCS(X_m, Y_n) = \begin{cases} LCS(X_{m-1}, Y_{n-1}) + x_m & \text{当} x_m = y_n \\ \max\{LCS(X_{m-1}, Y_n), LCS(X_m, Y_{n-1})\} & \text{当} x_m \neq y_n \end{cases}$$

- 显然，属于动态规划问题。

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Xi和Yj的最长公共子序列的长度。
  - 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。

$$c(i,j)=\begin{cases}0 & \text{当}i=0\text{或者}j=0\\ c(i-1,j-1)+1 & \text{当}i>0,j>0,\text{且}x_i=y_j\\ \max\{c(i-1,j),c(i,j-1)\} & \text{当}i>0,j>0,\text{且}x_i\neq y_j\end{cases}$$

算法中的数据结构：方向变量

- 使用二维数据B[m,n]，其中，b[i,j]标记c[i,j]的值是由哪一个子问题的解达到的。即c[i,j]是由c[i-1,j-1]+1或者c[i-1,j]或者c[i,j-1]的哪一个得到的。取值范围为Left，Top，LeftTop三种情况。

实例

- X=< A，B，C，B，D，A，B >
- Y=< B，D，C，A，B，A >

i	j	0	1	2	3	4	5	6
	y_j		B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
2	B	0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
3	C	0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
4	B	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
5	D	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
6	A	0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
7	B	0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

计算LCS长度

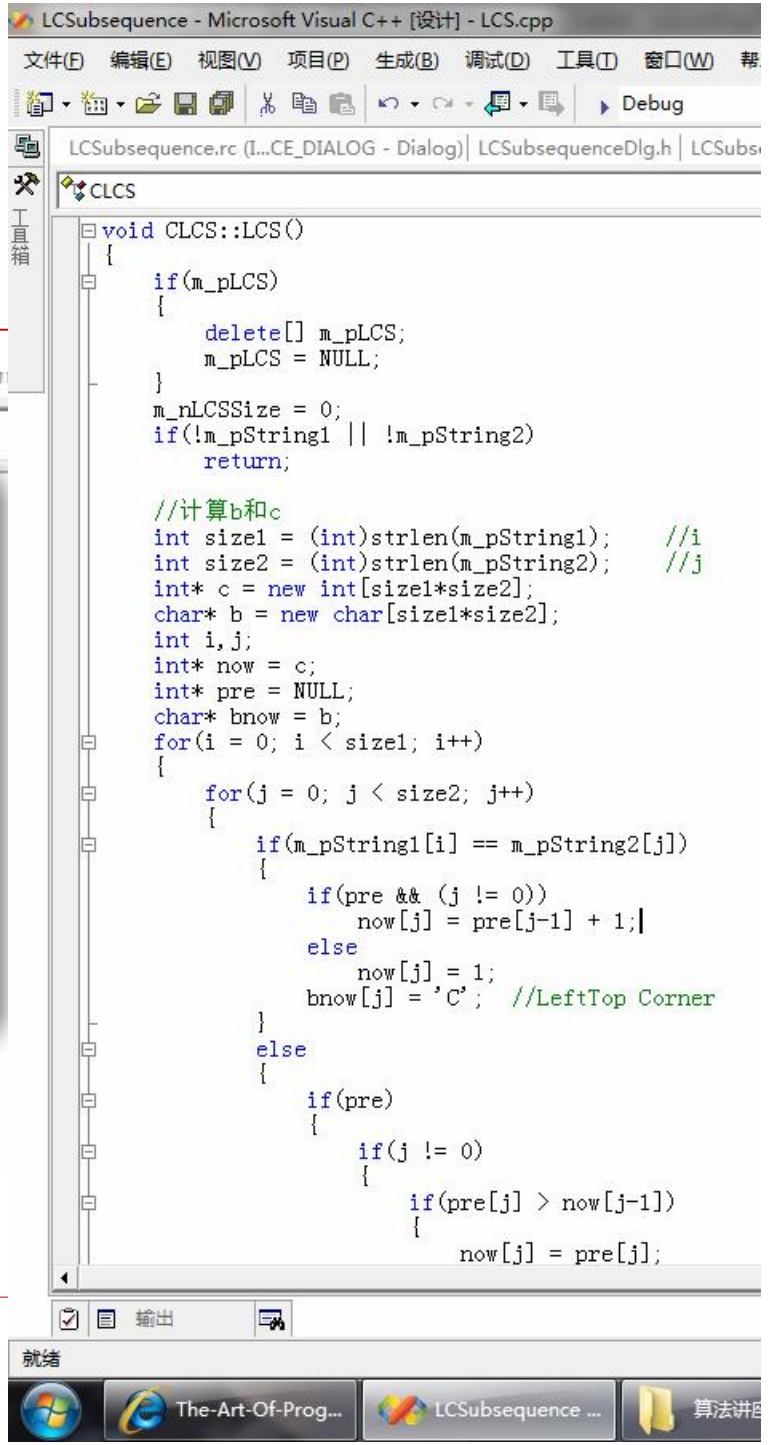
```
Procedure LCS_LENGTH(X,Y);
begin
  m:=length[X];
  n:=length[Y];
  for i:=1 to m do c[i,0]:=0;
  for j:=1 to n do c[0,j]:=0;
  for i:=1 to m do
    for j:=1 to n do
      if x[i]=y[j] then
        begin
          c[i,j]:=c[i-1,j-1]+1;
          b[i,j]:="↖";
        end
      else if c[i-1,j]≥c[i,j-1]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

根据b提供的方向，构造最长公共子序列

```
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;
```

算法实现Demo


LCSubsequence
字符串1
BCDABAB
字符串2
CBACBAABA
最长公共子序列
CABAB
计算
关闭
Index = size1*size2-1;
SSize = c[nIndex];
S = new char[m_nLCSSize+1];
S[m_nLCSSize] = 0;


LCSubsequence - Microsoft Visual C++ [设计] - LCS.cpp
文件(F) 编辑(E) 视图(V) 项目(P) 生成(B) 调试(D) 工具(T) 窗口(W)
Debug
LCSubsequence.rc (I...CE_DIALOG - Dialog) LCSubsequenceDlg.h
CLCS
void CLCS::LCS()
{
if(m_pLCS)
{
delete[] m_pLCS;
m_pLCS = NULL;
}
m_nLCSSize = 0;
if(!m_pString1 || !m_pString2)
return;
//计算b和c
int size1 = (int)strlen(m_pString1); //i
int size2 = (int)strlen(m_pString2); //j
int* c = new int[size1*size2];
char* b = new char[size1*size2];
int i,j;
int* now = c;
int* pre = NULL;
char* bnow = b;
for(i = 0; i < size1; i++)
{
for(j = 0; j < size2; j++)
{
if(m_pString1[i] == m_pString2[j])
{
if(pre && (j != 0))
now[j] = pre[j-1] + 1;
else
now[j] = 1;
bnow[j] = 'C'; //LeftTop Corner
}
else
{
if(pre)
{
if(j != 0)
{
if(pre[j] > now[j-1])
{
now[j] = pre[j];
输出
就绪
The-Art-Of-Prog...
LCSubsequence ...
算法讲

进一步思考的问题

- 方向数组b是完全可以省略的：
  - 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，因此，在计算中，可以临时判断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。
- 若只计算LCS的长度，则空间复杂度为min(m, n)。
  - 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。

最大公共子序列的多解性：求所有的LCS

$$LCS(X_m,Y_n)=\begin{cases} LCS(X_{m-1},Y_{n-1})+x_m & \text{当} x_m = y_n \\ \max\{LCS(X_{m-1},Y_n),LCS(X_m,Y_{n-1})\} & \text{当} x_m \neq y_n \end{cases}$$

- 当$x_m \neq y_n$时：

  若$LCS(X_{m-1},Y_n)=LCS(X_m,Y_{n-1})$，会导致多解：有多个最长公共子序列，并且它们的长度相等。
- B的取值范围从1,2,3扩展到1,2,3,4
- 广度优先遍历

		Yj	A	B	C	D	C	D	A	B
		0	1	2	3	4	5	6	7	8
Xi	0	0(0)	0(0)	0(0)	0(0)	0(0)	0(0)	0(0)	0(0)	0(0)
B	1	0(0)	0(4)	1(1)	1(3)	1(3)	1(3)	1(3)	1(3)	1(1)
A	2	0(0)	1(1)	1(4)	1(4)	1(4)	1(4)	1(4)	2(1)	2(3)
D	3	0(0)	1(2)	1(4)	1(4)	2(1)	2(3)	2(1)	2(4)	2(4)
C	4	0(0)	1(2)	1(4)	2(1)	2(4)	3(1)	3(3)	3(3)	3(3)
D	5	0(0)	1(2)	1(4)	2(2)	3(1)	3(4)	4(1)	4(3)	4(3)
C	6	0(0)	1(2)	1(4)	2(1)	3(2)	4(1)	4(4)	4(4)	4(4)
B	7	0(0)	1(2)	2(1)	2(4)	3(2)	4(2)	4(4)	4(4)	5(1)
A	8	0(0)	1(1)	2(2)	2(4)	3(2)	4(2)	4(4)	5(1)	5(4)

LCS的应用：最长递增子序列LIS

- Longest Increasing Subsequence
- 给定一个长度为N的数组，找出一个最长的单调递增子序列。
- 例如：给定数组{5，6，7，1，2，8}，则其最长的单调递增子序列为{5，6，7，8}，长度为4。
  - 分析：其实此LIS问题可以转换成最长公子序列问题，为什么呢？

使用LCS解LIS问题

- 原数组为A {5，6，7，1，2，8}
- 排序后：A’{1，2，5，6，7，8}
- 因为，原数组A的子序列顺序保持不变，而且排序后A’本身就是递增的，这样，就保证了两序列的最长公共子序列的递增特性。如此，若想求数组A的最长递增子序列，其实就是求数组A与它的排序数组A’的最长公共子序列。
  - 此外，本题也可以直接使用动态规划来求解

LCS的应用：最长递增子序列LIS

- Longest Increasing Subsequence
- 给定一个长度为N的数组，找出一个最长的单调递增子序列。
- 例如：给定数组{5，6，7，1，2，8}，则其最长的单调递增子序列为{5，6，7，8}，长度为4。
  - 分析：其实此LIS问题可以转换成最长公子序列问题，为什么呢？

使用LCS解LIS问题

- 原数组为A {5，6，7，1，2，8}
- 排序后：A’{1，2，5，6，7，8}
- 因为，原数组A的子序列顺序保持不变，而且排序后A’本身就是递增的，这样，就保证了两序列的最长公共子序列的递增特性。如此，若想求数组A的最长递增子序列，其实就是求数组A与它的排序数组A’的最长公共子序列。
  - 此外，本题也可以直接使用动态规划来求解

附：LIS的动态规化算法

- 设长度为N的数组为{a0，a1, a2, ...an-1}，则假定以aj结尾的数组序列的最长递增子序列长度为L(j)，则L(j)={ max(L(i))+1, i<j且a[i]<a[j] }。也就是说，我们需要遍历在j之前的所有位置i(从0到j-1)，找出满足条件a[i]<a[j]的L(i)，求出max(L(i))+1即为L(j)的值。最后，我们遍历所有的L(j)（从0到N-1），找出最大值即为最大递增子序列。时间复杂度为$O(N^2)$。

```cpp
#include <iostream>
using namespace std;
#define len(a) (sizeof(a) / sizeof(a[0])) //数组长度
int lis(int arr[], int len)
{
    int longest[len];
    for (int i=0; i<len; i++)
        longest[i] = 1;

    for (int j=1; j<len; j++) {
        for (int i=0; i<j; i++) {
            if (arr[j]>arr[i] && longest[j]<longest[i]+1){//注意longest[j]<longest[i]+1这个条件，不能省略
                longest[j] = longest[i] + 1; //计算以arr[j]结尾的序列的最长递增子序列长度
            }
        }
    }

    int max = 0;
    for (int j=0; j<len; j++) {
        cout << "longest[" << j << "]=" << longest[j] << endl;
        if (longest[j] > max) max = longest[j];  //从longest[j]中找出最大值
    }
    return max;
}

int main()
{
    int arr[] = {1, 4, 5, 6, 2, 3, 8}; //测试数组
    int ret = lis(arr, len(arr));
    cout << "max increment substring len=" << ret << endl;
    return 0;
}
```

LCS与字符串编辑距离

- 字符串编辑距离（Edit Distance），是俄罗斯科学家Vladimir Levenshtein在1965年提出的概念，又称Levenshtein距离，是指两个字符串之间，由一个转成另一个所需的最少编辑操作次数。许可的编辑操作包括
  - 将一个字符替换成另一个字符
  - 插入一个字符
  - 删除一个字符
- 如何求两个字符串的编辑距离？

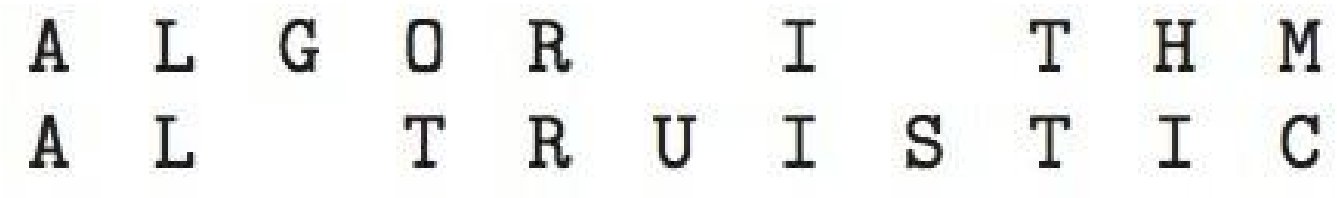
- 去年9月26日百度一二面试题，10月9日腾讯面试题第1小题，10月13日百度2013校招北京站笔试题第二大题第3小题，及去年10月15日2013年Google校招笔试最后一道大题皆是考察的这个字符串编辑距离问题。

编辑距离的应用

- DNA分析
  - 基因学的一个主要主题就是比较DNA序列并尝试找出两个序列的公共部分。如果两个DNA序列有类似的公共子序列，那么这些两个序列很可能是同源的。在比对两个序列时，不仅要考虑完全匹配的字符，还要考虑一个序列中的空格或间隙（或者，相反地，要考虑另一个序列中的插入部分）和不匹配，这两个方面都可能意味着突变（mutation）。在序列比对中，需要找到最优的比对（最优比对大致是指要将匹配的数量最大化，将空格和不匹配的数量最小化）。如果要更正式些，可以确定一个分数，为匹配的字符添加分数、为空格和不匹配的字符减去分数。
- 拼写纠错（Spell Correction）& 拼写检查（Spell Checker）
  - 将每个词与词典中的词条比较，英文单词往往需要做词干提取等规范化处理，如果一个词在词典中不存在，就被认为是一个错误，然后试图提示N个最可能要输入的词——拼写建议。常用的提示单词的算法就是列出词典中与原词具有最小编辑距离的词条。
- 命名实体抽取（Named Entity Extraction）&实体共指（Entity Coreference）
  - 由于实体的命名往往没有规律，如品牌名，且可能存在多种变形、拼写形式，如“IBM”和“IBM Inc.”，这样导致基于词典完全匹配的命名实体识别方法召回率较低，为此，我们可以使用编辑距离由完全匹配泛化到模糊匹配。
- 字符串核函数（String Kernel）
  - 最小编辑距离作为字符串之间的相似度计算函数，用于SVM等机器学习算法中。

字符串编辑的分析

- 字符串“ALGORITHM”是如何变成字符串“ALTRUISTIC”的



递推关系

- 枚举字符串S和T最后一个字符s[i]、t[j]对应的四种情况：(字符-字符)（字符-空白）（空白-字符）（空白-空白）；显然的是，（空白-空白）必然是多余的编辑操作。
- S + 空白
- T + 字符X
  - S变成T，最后，在S的末尾插入“字符X”
  - dp[i,j] = dp[i,j-1] + 1
- S + 字符X
- T + 字符Y
  - S变成T，最后，在X修改成Y
  - dp[i,j] = dp[i-1,j-1] + (X==Y ? 0 : 1)
- S + 字符X
- T + 空白
  - S变成T，X被删除
  - dp[i,j] = dp[i-1,j] + 1

编辑距离

```
//dp[i][j]表示源串source[0-i)和目标串target[0-j)的编辑距离
int EditDistance(char *pSource, char *pTarget)
{
    int srcLength = strlen(pSource);
    int targetLength = strlen(pTarget);
    int i, j;
    //边界dp[i][0] = i, dp[0][j] = j
    for (i = 1; i <= srcLength; ++i)
    {
        dp[i][0] = i;
    }
    for (j = 1; j <= targetLength; ++j)
    {
        dp[0][j] = j;
    }
    for (i = 1; i <= srcLength; ++i)
    {
        for (j = 1; j <= targetLength; ++j)
        {
            if (pSource[i - 1] == pTarget[j - 1])
            {
                dp[i][j] = dp[i - 1][j - 1];
            }
            else
            {
                dp[i][j] = 1 + min(dp[i - 1][j],
                   dp[i - 1][j - 1], dp[i][j - 1]);
            }
        }
    }
    return dp[srcLength][targetLength];
}
```

进一步的思考

- 允许交换，算一次变换：如meter/metre
  - 能否写出递推关系式？
  - 还能设计出O(n^2)的算法吗？
- 如果计算字符串的语义距离，怎么考虑？
  - WordNet是由Princeton 大学的心理学家，语言学家和计算机工程师联合设计的一种基于认知语言学的英语词典。它不是光把单词以字母顺序排列，而且按照单词的意义组成一个“单词的网络”。

应用编辑距离处理实际问题的剪枝策略

- 因为一般拼写检查应用只需要给出Top-N的纠正建议即可（如N=10），那么我们可以从词典中按照长度依次为len、len-1、len+1、len-2、len+2、...的词条比较；
- 限定拼写建议词条与当前词条的最小编辑距离不能大于某个阈值；
- 如果搜索到最小编辑距离为x以内的候选词条超过N后，终止处理；
- 缓存常见的拼写错误和建议，提高性能。

进一步思考

- 如果要计算两个已知字符串的最长公共子串的问题，如何求解？

- 借鉴LCS的思想：
- 求字符串A，B的最长公共子串S
- 考察A的i前缀和B的j前缀的最长公共子串S(i,j)：
  - 若A[i]==B[j]，则S(i,j)=S(i-1,j-1)+1

附：实例

- 求字符串bab和caba的最长公共子串

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

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

参考文献

- 余祥宣等，计算机算法基础[M]，华中科技大学出版社，2001
- 刘佳梅.求最长公共子序列问题的一种快速算法.中国科技论文在线[J].2010,11
- 李欣，舒风迪.最长公共子序列问题的改进快速算法.计算机应用研究[J].2000
- 郑翠玲.最长公共子序列算法的分析与实现.武夷学院学报[J],2010,29 卷(2):44~48
- https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/zh/02.04.md(最大子数组)
- https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/zh/05.02.md(字符串编辑距离)

我们在这里

- 更多算法面试题在7|七月算法官网
  - http://www.julyedu.com/
    - 免费视频
    - 直播课程
    - 问答社区

- contact us：微博
  - @七月算法
  - @七月问答

感谢大家

恳请大家批评指正！