2019-2020 学年第一学期算法设计与分析

上机实验报告

实验名称：最长公共子序列

学　号	秦敏浩	姓　名	17122490	评　分	
专　业	计算机科学与技术	实验类型	综合	任课教师	岳晓冬
完成日期	2019.10.4	实验学时	2		

一、实验问题描述

1、实验内容

最长公共子序列：给定 2 个序列 X、Y，求 X 和 Y 的最长公共子序列 Z。如有潜能，写程序找出所有的最长公共子序列。

2、实验目的

- 完成最长公共子序列问题
- 理解动态规划的要点和性质，写出动态方程
- 找出全部最长公共子序列，比较两种建图方法的优劣
- 改进书中代码

3、实验环境

操作系统：Linux/Windows

编译环境：GNU G++14

二、算法设计与分析

1、算法基本思想

考虑到任何一个字符串和空串的 LCS 长度为 0。设现已知道 X-1 串和 Y-1 串的 LCS、X 串和 Y-1 串的 LCS、X-1 串和 Y 串的 LCS，则此时 X 串和 Y 串的 LCS 就可以通过以上三者得到——若 X 串和 Y 串末尾字母一致，则为 LCS(X-1,Y-1)+1，否则在 LCS(X,Y-1)和 LCS(X-1,Y)中取大者即可。

考虑在每一次转移时记录路径，正向建边或反向建边形成 DAG 图，即可通过回溯求出所有最长公共子序列。两者在效率上会有不同，这在之后会进行讨论。此外，在记录路径时可采用二进制压位的方法，可以简化书中代码。

2、算法设计与求解步骤

a,b：输入的字符串

from[x][y]：记录 LCS(X,Y)的更新来源

dp[x][y]：记录 LCS(X,Y)

- 空串和任何串的 LCS 大小为 0，记录 LCS[0,y]=LCS[x,0]=0
- 建立两重循环，外层扩展 X 串长度 x，内层扩展 Y 串长度 y
- 根据转移方程完成转移，记录更新来源
- 根据更新来源，回溯所有最长公共子序列，并输出

3、算法分析

动态规划部分：由于每个数组单元计算耗费为 O(1)，因此总复杂度为 O(mn)，其中 m、n 为两个串的长度。动态方程为：

$$dp[i,j]=\begin{cases}0 & i=0 \text{ or } j=0\\ dp[i-1,j-1]+1 & i,j>0;X_i=Y_j\\ \max\left(dp[i,j-1],dp[i-1,j]\right) & i,j>0;X_i\neq Y_j\end{cases}$$

寻找一个 LCS 串：如果从[0,0]开始寻找，则复杂度为 O(mn)；如果从[m,n]开始回溯，则复杂度为 O(m+n)

寻找全部 LCS 串：欧拉拆分函数：约 $O\left(e^{\sqrt{\frac{20}{3}(m+n-2)}}\right)$

简单说明：考虑到最坏情况下 LCS 长度为 0，m*n 的矩阵除边界外所有节点都有向上或向左两个转移方向。此时所有递归路径边界与 m*n 的 ferres 图像定义一致。原问题等价于求 m 个小于等于 n 的自然数的组合方案数。即等价于将整数 m+n 拆分成 n 个正整数无序排列方案数。如果从[0,0]开始寻找，则复杂度更高。

4、算法程序实现

篇幅原因，见附录或附件。

三、调试与运行

输入：

```
2

7 6

A B C B D A B

B D C A B A

8 9

b a a b a b a b

a b a b b a b b a
```

输出：

```
2
7 6
A B C B D A B
B D C A B A

Case 1
LCS(X, Y) = 4

B C B A
B C A B
B D A B
B D A B

  0  0  0  0  0  0  0
  0  0  0  0  1  1  1
  0  1  1  1  1  2  2
  0  1  1  2  2  2  2
  0  1  1  2  2  3  3
  0  1  2  2  2  3  3
  0  1  2  2  3  3  4
  0  1  2  2  3  4  4

000 010 010 010 010 010 010
001 011 011 011 100 010 110
001 100 010 010 011 100 010
001 001 011 100 010 011 011
001 101 011 001 011 100 010
001 001 100 011 011 001 011
001 001 001 011 100 011 100
001 101 001 011 001 100 011
```

```
8 9
b a a b a b a b
a b a b b a b b a

Case 2
LCS(X, Y) = 6

b a b a b a
b a b a b a
a a b a b a
b a b a b a
b a b a b a
b a b a b a
a a b a b a
b a a b b a
b a b a b a
b a b a b a
a a b a b a
b a b a b a
b a b a b a
b a b a b a
a a b a b a
a b a b a b
a b a b a b
b a b b a b
b a b b a b
a a b b a b
a b a b a b
a b a b a b
b a b a b b
b a b a b b
a a b a b b
b a b a b b
b a b a b b
b a b a b b
a a b a b b
a b a b a b
a b a b a b
b a b b a b
b a b b a b
a a b b a b
a b a b a b
a b a b a b

  0  0  0  0  0  0  0  0  0  0
  0  0  1  1  1  1  1  1  1  1
  0  1  1  2  2  2  2  2  2  2
  0  1  1  2  2  2  3  3  3  3
  0  1  2  2  3  3  3  4  4  4
  0  1  2  3  3  3  4  4  4  5
  0  1  2  3  4  4  4  5  5  5
  0  1  2  3  4  4  5  5  5  6
  0  1  2  3  4  5  5  6  6  6

000 010 010 010 010 010 010 010 010 010
001 011 100 010 110 110 010 110 110 010
001 100 011 100 010 010 110 010 010 110
001 101 011 101 011 011 100 010 010 110
001 001 100 011 100 110 011 100 110 010
001 101 001 100 011 011 100 011 011 100
001 001 101 001 100 110 011 100 110 011
001 101 001 101 001 011 100 011 011 100
001 001 101 001 101 100 011 100 110 011
```

四、结果分析

理论分析和测试结果表明了实验的正确性。以下分析对书上代码的优化问题：

回溯时使用二进制优化：书中记录 LCS 更新来源时采用 1,2,3 来记录。这样是无法输出全部最长公共子序列的。一般需要使用 4 种状态进行记录：dp[i,j]由 dp[i-1,j-1]得到；dp[i,j]仅由 dp[i,j-1]得到；dp[i,j]仅由 dp[i-1,j]得到；dp[i,j]仅由 dp[i,j-1]，dp[i-1,j]共同得到。可采用三位二进制来表示这 4 种状态，最高位代表 dp[i-1,j-1]，以此类推。这样在回溯时可以简化代码逻辑。

回溯方向问题：考虑到记录更新来源的本质属于创建一种 DAG 图，包含正向建图和反向建图两种方法。由于反向建图后从[m,n]开始回溯的每条路径都最终到达[0,0]，而正向建图则不然。因此采用反向建图回溯效率会更高。

其他优化：理论上可行的其他优化：寻找全部 LCS 串时进行记忆化操作可以将复杂度减少至 $O\left(m\cdot n\cdot \min\left(m,n\right)\right)$ 即多项式级别；目前代码实现输出**全部最长公共子序列**，采用带状态回溯可以不重复不缺少的找到**本质方案不同的全部最长公共子序列**，采用排序后去重可以找到全部**字符意义上不同的全部最长公共子序列**。

五、本次实验的收获、心得体会

首先，最长公共子序列是一道非常基本而典型的动态规划问题。使用贪心思想决策每一个 dp[i,j]，如果思考清楚很容易推出转移方程。

其次，这是一个开放问题。正如结果分析中所述，题干中对“全部最长公共子序列”的理解有多种不同的方式，由此会产生不同的做法。对于不同的做法，又有着不同的优化方式和写法。

最后，这是一个很有意思的问题。对于不同的需求，不仅代码也会截然不同，甚至还会影响代码时间或空间复杂度。对于寻找全部最长公共子序列问题，可以采用简单回溯问题，此时时间复杂度为指数级，空间复杂度为 O(mn)。若采用带状态的记忆化搜索，则时间复杂度为多项式级，空间复杂度上升为 $O\left(m\cdot n\cdot \min\left(m,n\right)\right)$。此外，对于这些问题的时间复杂度问题也是一个有趣的问题。如之前将问题转化为整数拆分方案数问题，转化过程就比较复杂，而这个问题本身也可以通过动态规划思想进行求解。

附录：代码实现（也可见附件）

```cpp
#include <string>
#include <vector>
#include <iostream>
#include <iomanip>
#include <bitset>
using namespace std;

string ans;
int from[55][55]; // created reversed to get more speed.
char a[55],b[55];         // since there all at most three kinds of froms, use integer to save them
int dp[55][55];

void dfs(int x,int y){ // to find all the LCSs
    if (x==0&&y==0){ // print and return if it get the answer
        for (int i=ans.length()-1;i>=0;--i){ // print in the reversed order
            cout<<ans[i]<<' ';
        }
        cout<<endl;
        return;
    }
    if (from[x][y]&1){            // however not all the froms in the original graph end to [m,n]
        dfs(x-1,y);
    }
    if (from[x][y]&2){
        dfs(x,y-1);
    }
    if (from[x][y]&4){
        ans.push_back(a[x]);
        dfs(x-1,y-1);
        ans.erase(--ans.end()); // to save time
    }
}

int main(){
    int T,kase=0,n,m;
    cin>>T;
    while (T--){
        cin>>m>>n;
        for (int i=0;i<=m;++i){ // initialization
            for (int j=0;j<=n;++j){
                dp[i][j]=0;
            }
        }
        for (int i=1;i<=m;++i){
            from[i][0]=1; // required for tracing back
            cin>>a[i];
        }
        for (int i=1;i<=n;++i){
            from[0][i]=2; // required for tracing back
            cin>>b[i];
        }
        for (int i=1;i<=m;++i){
            for (int j=1;j<=n;++j){
                dp[i][j]=a[i]==b[j]?dp[i-1][j-1]+1:max(dp[i][j-1],dp[i-1][j]);
                from[i][j]=0;
                if (dp[i][j]==dp[i-1][j]){
                    from[i][j]|=1;  // from[i][j]==1: [i,j]->[i-1,j]
                }
                if (dp[i][j]==dp[i][j-1]){
                    from[i][j]|=2; // from[i][j]==-1: [i,j]->[i,j-1]
                }
                if (dp[i][j]==dp[i-1][j-1]+1&&a[i]==b[j]){
                    from[i][j]|=4;  // from[i][j]==0: [i,j]->[i-1,j-1]
                }
            }
        }
        cout<<endl<<"Case "<<++kase<<endl;
        cout<<"LCS(X, Y) = "<<dp[m][n]<<endl<<endl;;
        ans="";
```

```
        dfs(m,n);
        cout<<endl;
        for (int i=0;i<=m;++i){
            for (int j=0;j<=n;++j){
                cout<<setw(3)<<dp[i][j];
            }
            cout<<endl;
        }
        cout<<endl;
        for (int i=0;i<=m;++i){
            for (int j=0;j<=n;++j){
                cout<<setw(4)<<((bitset<3>)from[i][j]);
            }
            cout<<endl;
        }
        cout<<endl;
    }
    return 0;
}
```