《算法设计与分析》

课程实验报告

专业：计算机科学与技术

班级：2021211307

姓名：陈朴炎

学号：2021211138

目录

1 作业内容

1.1 作业 1.1 最长公共子序列

利用“附件 1.最长公共子序列输入数据-2023“中给出的字符串 A，B，C，D 分别找出下列两两字符串间的最长公共子串，并输出结果：

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

说明：

- 产生由 10 个数字{0,1,2,3,4,5,6,7,8,9}组成的长度在 800-1000 之间（也可以更长）的序列 A1，C1
- 产生由 10 个符号{),!,@,#,$,%,^,&,*,(}组成的长度在 800-1000 之间（也可以更长）的序列 B1，D1
- 将由 26 个英文字母和符号“+”组成的字符串 an+algorithm+is+any+welldefined+computational+procedure+that+takes+some+values+as+input+and+produces+some+values+as+output 中的各个字母和符号“+”在保持原有前后顺序的前提下插入到字符串 A1，B1，C1，D1 中，得到字符串 A，B，C，D

注：在 C、D 中，An+algorithm…

由 26 个英文字母和+组成的字符串中的各个符号插入到 A1，B1，C1，D1 中后，任意 2 个符号间应当有数字隔开。例如：1a27n4+498a3l9g76o,不要出现“1an4+498al9g76o”。

1.2 作业 1.2 最长递减子序列

利用最长公共子序列求解下列最长递减子序列问题：

- 给定由 n 个整数$a_1, a_2, \ldots, a_n$构成的序列，在这个序列中随意删除一些元素后可得到一个子序列$a_i, a_j, a_k, \ldots, a_m$，其中 1≤i≤m≤n，并且$a_i \geq a_j \geq a_k \geq \cdots \geq a_m$，则称序列$a_i, a_j, a_k, \ldots, a_m$为原序列的一个递减子序列，长度最长的递减子序列即为原序列的最长递减子序列。
- 例如，序列{1, 7, 2, 3, 6, 5}，它的一个最长递减子序列为{7, 6, 5}

要求：

- 利用“附件 2.最大子段和输入数据-序列 1-2023”、“附件 2.最大子段和输入数据-序列 2-2023”，求这两个序列中的最长递减子序列

1.3 作业 2 最大子段和

针对“附件 2.最大子段和输入数据-序列 1-2023”、“附件 2.最大子段和输入数据-序列 2-2023”中给出的序列 1、序列 2，分别计算其最大子段和

- 序列 1：长度在 300-500 之间，由(-400,2023)内的数字组成
- 序列 2：长度在 100-200 之间，由(-307,2023)内的数字组成
- 要求
- 1．指出最大子段和在原序列中的位置
- 2．给出最大子段和具体值

1.4 评判标准

（1）针对上述典型问题，编程实现算法（二选一），程序能够针对一种输入正常

运行，给出正确结果，并提交代码及实验报告。

（2）算法程序能够面对多种输入和边界条件稳定运行，输出正确结果，算法性能符合预期，并且按期提交代码及实验报告。

（3）在满足 2 的前提下，能够观察对比不同规模的输入数据下，算法的运行时间和空间占用的变化，分析算法时间和空间复杂性。

（4）在满足 2、3 的前提下，提出改进算法性能的思路及方法，并付诸实现。

2 最长公共子序列问题

2.1 算法设计

2.1.1 算法设计思想

（1）判断最优子结构性质

设序列 X(m) = {x1, x2, …, xm}和 Y(n) = {y1, y2, …, yn}的最长公共子序列为 Z(k) = {z1, z2, … zk}，问题规模为<m, n>。分析以下三种不同情况：

- 情况 1：若 $x_m=y_n$，则 $z_k=x_m=y_n$，且 $Z(k-1)=\{z_1, z_2,\ldots,z_{k-1}\}$ 是 $X(m-1)=\{x_1,x_2,\ldots,x_{m-1}\}$和 $Y(n-1)=\{y_1,y_2,\ldots,y_{n-1}\}$的最长公共子序列。
- 情况 2：若 $x_m\neq y_n$，$z_k=y_n$，则 Z(k)是 X(m-1)和 Y(n)的的最长公共子序列。
- 情况 3：若 $x_m\neq y_n$且 $z_k\neq y_n$，则 Z(k)是 X(m)和 Y(n-1)的最长公共子序列。

上述 3 种情况下，原问题的最优解包括了子问题的最优解。由此可见，2 个序列的最长公共子序列包含了这 2 个序列的前缀最长公共子序列

因此， 最长公共子序列问题具有最优子结构性质。因此，根据小规模子问题

<m-1，n-1><m-1，n><m，n-1>的解，可以自下而上构造问题<m，n>的解。

（2）问题递归结构/状态方程

使用一个二维数组 c 来辅助算法实现，c[i][j]表示序列 X(i) 和 Y(j)的最长公共子序列的长度。

1． 边界条件：X(i)={x_1,x_2,…,x_i}，Y(j)={y_1,y_2,…,y_j}，当 i=0 或 j=0 时，即其中 1 个序列为空，则空序列是 X_i 和 Y_j 的最长公共子序列，故此时 C[i][j]=0

2．其它情况下，由最优子结构性质可建立递归关系如下：

$$c[i][j]=\begin{cases}0 & i=0,或j=0\\ c[i-1][j-1]+1 & i,j>0;x_i=y_j\\ \max\{c[i][j-1],c[i-1][j]\} & i,j>0;x_i\neq y_j\end{cases}$$

（3）算法设计

对 X(m)和 Y(n)，总共有 **ϑ(mn)**个不同的子问题，用动态规划算法自底向上地计算最长公共子序列，避免重复计算子问题，以提高算法效率。

考虑到在求最长公共子串的时候，需要得到不同位置的公共子串是由哪种情况的来的，因此需要一个二维数组 b，b[i][j]用来记录 c[i][j]的值是由哪个子问题得到的（3 种情况之一），用于后续构造最长公共子序列。

首先，我们要得到数组 c 和数组 b。步骤如下：

（1）初始化 c 数组，当 Y[j]为空时，最小子问题<i， 0>为 0；当 X[i]为空时，最小子问题<0，j>为 0。

（2）需要两重循环，自下而上，计算子问题{X[i]，Y[j]}

（3）第一重循环，i 从 1 开始，一直到 m 结束，表示从左往右一步一步计算

对于 X 串前 i 个字符和 Y 串的最长公共子序列。

（4）第二重循环，j 从 1 开始，一直到 n 结束，表示对应于每个 X 的前缀，都计算它们对应于 Y 的最长公共子序列。并且判断如下情况：

如果发现 X[i]==Y[j]，那么对应于 X[i]和 Y[j]的最长公共子串 Z[k]长度就是 Z[k-1]+1。并且这是情况 1，将 b[i][j]赋值为 1。

如果 X[i]≠Y[j]，就要判断下面两种情况：

1.c[i-1][j]>=c[i][j-1]， 就将 c[i][j]赋值为 c[i-1][j]，b[i][j]赋值为 2。

2.c[i-1][j] <c[i][j-1]，就将 c[i][j]赋值为 c[i][j-1]， b[i][j]赋值为 3。

（5）得到数组 c 和 b 后，我们可以求出最长公共子序列的值，步骤如下：

1.查看 b[i][j]的值，进行判断

2.如果为 1，那么说明是情况 1，X_i 和 Y_j 的最长公共子序列由 X_{i-1} 和 Y_{j-1} 的最长公共子序列，在尾部加上 x_i 得到。

3.如果为 2，说明是情况 2，X_i 和 Y_j 的最长公共子序列与 X_{i-1} 和 Y_j 的的最长公共子序列相同，那么需要查看 b[i-1][j]的值，并跳到 2。

4.如果为 3，说明是情况 3，X_i 和 Y_j 的最长公共子序列与 X_i 和 Y_{j-1} 的的最长公共子序列相同，那么需要查看 b[i][j-1]的值，并跳到 2。

2.1.2 基础算法代码

```
1.  void LCS(int i, int j, string x, int **b){
2.      if(i==0 || j==0)
3.          return;
```

```
    // 第一种情况下，最长公共子序列由 x[i-1]y[j-1]的解由 x[i]构成
    if(b[i][j] == 1){
        LCS(i-1, j-1, x, b);
        cout << x[i];
    }else if(b[i][j] == 2){
        LCS(i-1, j, x, b);
    }else{
        LCS(i, j-1, x, b);
    }
}

void LCSLengthPro(int m, int n, string x, string y, int **c){
    int i, j;
    // Y[j]空时，最小子问题<i, 0>
    for(i = 1; i <= m; i++)
        c[i][0] = 0;

    // X[i]空时，最小子问题<0, j>
    for(i = 1;i <= n; i++)
        c[0][i] = 0;
    for(i = 1; i <= m; i++){
        for(j = 1; j <= n; j++){
            if(x[i] == y[j]){
                c[i][j] = c[i-1][j-1] + 1;
            }else if(c[i-1][j] >= c[i][j-1]){
                c[i][j] = c[i-1][j];
            }else{
                c[i][j] = c[i][j-1];
            }
        }
    }
}
```

2.1.3 复杂性分析

在 LCSLength 函数中，每个数组单元的计算耗时 O(1)，算法耗时 O(mn)，由于需要两个二维数组，所以所需的空间复杂性为 O(mn)。

在 LCS 函数中，每次递归调用使 i 或 j 减一，算法计算的时间复杂性为

O(m+n)。

2.2 算法改进

2.2.1 算法改进思想

在算法 LCSLength 和 LCS 中，可进一步将数组 b 省去

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]这 3 个数组元素的值所确定。对于给定的数组元素 c[i][j]，可以不借助于数组 b 而仅借助于 c 本身在 O(1)时间内确定 c[i][j]的值是由 c[i-1][j-1],c[i-1][j]和 c[i][j-1]中哪一个值所确定的。

2.2.2 算法改进实现代码

```
1.  int checkBIJ(int ** c, int i, int j){
2.      if(i == 0||j == 0){
3.          cout << "checkBIJ : 下标出错" <<endl;
4.          exit(1);
5.      }
6.      if(c[i][j] == c[i-1][j-1] + 1){
7.          return 1;
8.      }
9.      if(c[i-1][j] >= c[i][j-1]){
10.         return 2;
11.     }
12.     return 3;
13. }
14. void LCSLengthPro(int m, int n, string x, string y, int **c){
15.     int i, j;
16.     // Y[j]空时，最小子问题<i, 0>
17.     for(i = 1; i <= m; i++)
18.         c[i][0] = 0;
19.
20.     // X[i]空时，最小子问题<0, j>
21.     for(i = 1;i <= n; i++)
```

```
        c[0][i] = 0;
    for(i = 1; i <= m; i++){
        for(j = 1; j <= n; j++){
            if(x[i] == y[j]){
                c[i][j] = c[i-1][j-1] + 1;
            }else if(c[i-1][j] >= c[i][j-1]){
                c[i][j] = c[i-1][j];
            }else{
                c[i][j] = c[i][j-1];
            }
        }
    }
}

void LCSpro(int i, int j, string x, int **c){
    if(i ==0 || j== 0){
        return;
    }
    // 第一种情况下，最长公共子序列由 x[i-1]y[j-1]的解和 x[i]构成
    if(checkBIJ(c, i, j) == 1){
        LCSpro(i-1, j-1, x, c);
        cout << x[i];
    }else if(checkBIJ(c, i, j) == 2){
        LCSpro(i-1, j, x, c);
    }else{
        LCSpro(i, j-1, x, c);
    }
}
```

2.2.3 改进后的复杂度分析

每次通过 c 数组来判断原先 b 数组的值只需要 O(1)的时间，而 b 数组被省去了。虽说时间复杂度和空间复杂度数量级没改变还是 O(n*n)和 O(n*n)，但是空间复杂度大大降低了。

而如果不想求最长公共子序列的值，而是只求最长公共子序列的长度，算法的空间需求还可以大大减少。

在计算 c[i][j]时，只用到数组 c 的第 i 行和第 i-1 行。因此，用 2 行的数组空间就可以计算出最长公共子序列的长度。进一步的分析还可将空间需求减至 O(min(m,n))

2.3 程序实现

```
1.  // 编码：GBK
2.  #include <iostream>
3.  #include <fstream>
4.  #include <vector>
5.  #include <string>
6.
7.  using namespace std;
8.
9.  void newArray(int **&array, size_t rows, size_t cols){
10.     array = new int *[rows];
11.     for(size_t i = 0; i < rows; i++){
12.         array[i] = new int[cols];
13.     }
14. }
15.
16. int checkBIJ(int ** c, int i, int j){
17.     if(i == 0||j == 0){
18.         cout << "checkBIJ : 下标出错" <<endl;
19.         exit(1);
20.     }
21.     if(c[i][j] == c[i-1][j-1] + 1){
22.         return 1;
23.     }
24.     if(c[i-1][j] >= c[i][j-1]){
25.         return 2;
26.     }
27.     return 3;
28. }
29.
30. void LCSLength(int m, int n, string x, string y, int **c, int **b){
31.     int i, j;
32.     // Y[j]空时，最小子问题<i, 0>
33.     for(i = 1; i <= m; i++)
34.         c[i][0] = 0;
```

```
35.
36.     // X[i]空时，最小子问题<0, j>
37.     for(i = 1;i <= n; i++)
38.         c[0][i] = 0;
39.     for(i = 1; i <= m; i++){
40.         for(j = 1; j <= n; j++){
41.             if(x[i] == y[j]){
42.                 c[i][j] = c[i-1][j-1] + 1;
43.                 b[i][j] = 1;
44.             }else if(c[i-1][j] >= c[i][j-1]){
45.                 c[i][j] = c[i-1][j];
46.                 b[i][j] = 2;
47.             }else{
48.                 c[i][j] = c[i][j-1];
49.                 b[i][j] = 3;
50.             }
51.         }
52.     }
53. }
54.
55. void LCS(int i, int j, string x, int **b){
56.     if(i==0 || j==0)
57.         return;
58.     // 第一种情况下，最长公共子序列由 x[i-1]y[j-1]的解由 x[i]构成
59.     if(b[i][j] == 1){
60.         LCS(i-1, j-1, x, b);
61.         cout << x[i];
62.     }else if(b[i][j] == 2){
63.         LCS(i-1, j, x, b);
64.     }else{
65.         LCS(i, j-1, x, b);
66.     }
67. }
68.
69. void LCSLengthPro(int m, int n, string x, string y, int **c){
70.     int i, j;
71.     // Y[j]空时，最小子问题<i, 0>
72.     for(i = 1; i <= m; i++)
73.         c[i][0] = 0;
74.
75.     // X[i]空时，最小子问题<0, j>
76.     for(i = 1;i <= n; i++)
```

```
77.         c[0][i] = 0;
78.     for(i = 1; i <= m; i++){
79.         for(j = 1; j <= n; j++){
80.             if(x[i] == y[j]){
81.                 c[i][j] = c[i-1][j-1] + 1;
82.             }else if(c[i-1][j] >= c[i][j-1]){
83.                 c[i][j] = c[i-1][j];
84.             }else{
85.                 c[i][j] = c[i][j-1];
86.             }
87.         }
88.     }
89. }
90.
91. void LCSpro(int i, int j, string x, int **c){
92.     if(i ==0 || j== 0){
93.         return;
94.     }
95.     // 第一种情况下，最长公共子序列由 x[i-1]y[j-1]的解和 x[i]构成
96.     if(checkBIJ(c, i, j) == 1){
97.         LCSpro(i-1, j-1, x, c);
98.         cout << x[i];
99.     }else if(checkBIJ(c, i, j) == 2){
100.             LCSpro(i-1, j, x, c);
101.        }else{
102.            LCSpro(i, j-1, x, c);
103.        }
104.    }
105.
106.    int main(){
107.        vector <string> strings;
108.        ifstream inputFile("./附件 1.最长公共子序列输入文件-2023.txt");
109.        if(!inputFile.is_open()){
110.            cerr << "打开附件 1 失败" <<endl;
111.            return 1;
112.        }
113.        string line;
114.        while(getline(inputFile, line)){
115.            strings.push_back(line);
116.        }
117.
118.        inputFile.close();
```

```
119.        // 第 0 个字符是没用的
120.        string A = " "+strings[1], B = " "+strings[4], C = " "+strings[7], D = "
    "+strings[10];
121.
122.        int **bab , **bcd, **bad, **bcb, **cab, **ccd, ** cad, ** ccb;
123.        newArray(bab, A.length()+1, B.length()+1);
124.        newArray(cab, A.length()+1, B.length()+1);
125.
126.        newArray(bcd, C.length()+1, B.length()+1);
127.        newArray(ccd, C.length()+1, B.length()+1);
128.
129.        newArray(bad, A.length()+1, D.length()+1);
130.        newArray(cad, A.length()+1, D.length()+1);
131.
132.        newArray(bcb, C.length()+1, B.length()+1);
133.        newArray(ccb, C.length()+1, B.length()+1);
134.
135.        LCSLength(A.length(), B.length(), A, B, cab, bab);
136.        LCSLength(A.length(), D.length(), A, D, cad, bad);
137.        LCSLength(C.length(), D.length(), C, D, ccd, bcd);
138.        LCSLength(C.length(), B.length(), C, B, ccb, bcb);
139.        cout << "A-B: ";
140.        LCS(A.length(), B.length(), A, bab);
141.        cout << endl << endl << "A-D: ";
142.        LCS(A.length(), D.length(), A, bad);
143.        cout << endl << endl << "C-D: ";
144.        LCS(C.length(), D.length(), C, bcd);
145.        cout << endl << endl << "C-B: ";
146.        LCS(C.length(), B.length(), C, bcb);
147.
148.        cout << endl << endl << "====================算法改进之后，输出如下：
    ====================" << endl;
149.        LCSLengthPro(A.length(), B.length(), A, B, cab);
150.        LCSLengthPro(A.length(), D.length(), A, D, cad);
151.        LCSLengthPro(C.length(), D.length(), C, D, ccd);
152.        LCSLengthPro(C.length(), B.length(), C, B, ccb);
153.        cout << "A-B: ";
154.        LCSpro(A.length(), B.length(), A, cab);
155.        cout << endl << endl << "A-D: ";
156.        LCSpro(A.length(), D.length(), A, cad);
157.        cout << endl << endl << "C-D: ";
158.        LCSpro(C.length(), D.length(), C, ccd);
```

```
159.        cout << endl << endl << "C-B: ";
160.        LCSpro(C.length(), B.length(), C, ccb);
161.        return 0;
162.    }
```

2.4 输出结果

```
ter3_dynamicProgramming\LCS\" ; if ($?) { g++ LCS_seq.cpp -o LCS_seq } ; if ($?) { .\LCS_seq }
A-B: an+algorithm+is+any+welldefined+computational+procedure+that+takes+some+values+as+input+and+p
roduces+some+values+as+output20212113xx2023

A-D: an+algorithm+is+any+welldefined+computational+procedure+that+takes+some+values+as+input+and+p
roduces+some+values+as+output20212113xx2023

C-D: an+algorithm+is+any+welldefined+computational+procedure+that+takes+some+values+as+input+and+p
roduces+some+values+as+output20212113xx2023

C-B: an+algorithm+is+any+welldefined+computational+procedure+that+takes+some+values+as+input+and+p
roduces+some+values+as+output20212113xx2023

====================算法改进之后，输出如下：====================
A-B: an+algorithm+is+any+welldefined+computational+procedure+that+takes+some+values+as+input+and+p
roduces+some+values+as+output20212113xx2023

A-D: an+algorithm+is+any+welldefined+computational+procedure+that+takes+some+values+as+input+and+p
roduces+some+values+as+output20212113xx2023

C-D: an+algorithm+is+any+welldefined+computational+procedure+that+takes+some+values+as+input+and+p
roduces+some+values+as+output20212113xx2023

C-B: an+algorithm+is+any+welldefined+computational+procedure+that+takes+some+values+as+input+and+p
roduces+some+values+as+output20212113xx2023
PS F:\hupt-homework\algorithm\Chapter3_dynamicProgramming\LCS>
```

图 2-1 最长公共子序列输出结果图

可以看到，改进前和改进后的算法输出结果都一样，测试的 4 个字符串的最长公共子串都相同。

3 最长递减子序列问题

3.1 算法设计

3.1.1 算法设计思想

（1）判断最优子结构性质

设序列 $\mathbf{X(m)}=\{x_1,x_2,\ldots,x_m\}$最长递减子序列为 $\mathbf{Z(k)}=\{z_1,z_2,\ldots,z_k\}$ ，问题规模< m >，考察两种情况下问题结构：

1. Z(k) == X(m)，那么说明 Z(k-1)是 X(m-1)的最长递减子序列。

2. Z(k) ≠ X(m)，那么说明 Z(k)是 X(m)的最长递减子序列。

由此可见，一个序列的最长递减子序列包含了这个序列的前缀子问题的最长递减子序列。因此，最长递减子序列问题具有最优子结构性质。

因此根据小规模子问题< m-1 >的解，可以自下而上构造问题< m >的解

（2）问题递归结构/状态方程

使用一个一维数组 c 来辅助算法实现，c[i]表示序列 X 的前缀 X(i)最长递减子序列的长度。

边界条件：c 里每一个元素的初始化为 1，因为对于数据数组里的每个元素都能构成一个长度为 1 的递减子序列。

状态方程：c[i] = max(c[j]) + 1,其中 0≤j<i 且 data[j] < data[i]。

（3）算法设计

（1）对每一个 c[i]，初始化为 1；对每一个 s[i]，初始化为-1。

（2）对数据数组从左往右扫描，判断其前缀 X[i]的最长递减子序列。

（3）第二层循环，j 从 0 到 i，如果发现 data[j] >= data[i] 并且 c[j]+1 > c[i]，那么就把 c[i]赋值为更大的 c[j]+1，并且将 s[i]指向 j，代表该前缀的最长递减子序列的倒数第二个数据在 data 的 j 下标处。

（4）找到最大的 c[i]，并找到所对应的下标 s[i]。

（5）通过下标一步一步回溯得到最长的递减子序列。

3.1.2 算法基础代码

```
1.  void LDS(vector<int>data){
```

```
    int n = data.size();
    vector<int> c(n, 1);// 初始化全为 1
    vector<int> s(n, -1);
    for(int i = 1; i < n ;i++){
        for(int j = 0; j < i; j++){
            if(data[j] >= data[i] && c[j]+1 > c[i]){
                c[i] = c[j]+1;
                s[i] = j;
            }
        }
    }
    int maxLen = *max_element(c.begin(), c.end());
    int i = distance(c.begin(), max_element(c.begin(), c.end()));
    vector<int> ldsSeq;
    while(i!=-1){
        ldsSeq.push_back(data[i]);
        i = s[i];
    }
    reverse(ldsSeq.begin(), ldsSeq.end());
    cout << "最长递减子序列的长度是: " << maxLen << endl;
    for(auto item : ldsSeq){
        cout << item << "\t";
    }
    cout << endl;
}
```

3.1.3 复杂度分析

时间复杂度：

有两层嵌套循环，外层循环迭代 n 次，内层循环最坏情况下迭代 i 次，其中 i 是当前外层迭代的索引。

max_element 函数： 需要迭代整个数组，最坏情况下迭代 n 次。

reverse 函数： 最坏情况下需要迭代 n/2 次。

总的时间复杂性为 O(n^2)，其中 n 是输入数据的大小。

空间复杂度：

需要额外 O(n) 的空间来存储 c 和 s 数组。

3.2 最长递减子序列另一种实现

3.2.1 逆向扫描算法思路

考虑到最长递减子序列也就相当于找到逆向的最长递增子序列，我们可以在第一层循环 i 从右往左扫描，第二层循环 j 从 i+1 往右扫描。具体思路如下：

（1）初始化 c、s 数组，设置 maxLength 变量存放最长序列长度。

（2）第一层循环，i 从 n-1 开始，一直到 0 结束。并设置一个变量 max，用来存放当前循环的最大子序列长度。

（3）第二层循环，j 从 i+1 开始，一直到 n-1 结束。在该层循环中，每次发现 data[j] <= data[i]，并且 max < c[j]时，都将 c[i] 设置成 c[j]+1，s[i] 指向 j，并让 max 更新。如果发现 maxLength < c[i]，那么更新 maxLength，并将索引值 i 记录下来。

（4）最后通过索引值回溯得到最长递减子序列。

3.2.2 算法实现

```
1.  int LDSpro(vector<int>data, int*c, int*s, int n){
2.      int ret_index = -1;
3.      int maxLength = 0;
4.      // 从后往前找最长递增子序列
5.      for(int i = n-1; i >= 0; i--){
6.          c[i] = 1;
7.          s[i] = i;
8.          int max = 0;
9.          for(int j=i+1; j < n; j++){
10.             if (data[j] <= data[i] && max < c[j]){
11.                 c[i] = c[j] + 1;
```

```
                s[i] = j;
                max = c[j];
                if(maxLength < c[i]){
                    maxLength = c[i];
                    ret_index = i;
                }
            }
        }
    }
    return ret_index;
}

vector<int> getLDSpro(vector<int> data, int *s, int index){
    vector<int> retSeq;
    retSeq.push_back(data[index]);
    while(index != s[index]){
        index = s[index];
        retSeq.push_back(data[index]);
    }
    return retSeq;
}
```

3.2.3 复杂度分析

时间复杂度：

有两层嵌套循环，外层循环迭代 n 次，内层循环最坏情况下迭代 n 次。总的时间复杂性为 O(n^2)，其中 n 是输入数据的大小。

空间复杂性：

c 和 s 数组： 需要额外 O(n) 的空间来存储 c 和 s 数组。

retSeq 数组： 最坏情况下需要 O(n) 的空间来存储 retSeq 数组。

总的空间复杂性为 O(n)。

3.3 程序实现

```
// 编码 GBK
```

```cpp
#include <iostream>
#include <fstream>
#include <vector>
#include <algorithm>
using namespace std;

void LDSpro1(vector<int>data){
    int n = data.size();
    vector<int> c(n, 1);// 初始化全为 1
    vector<int> s(n, -1);
    for(int i = 1; i < n ;i++){
        for(int j = 0; j < i; j++){
            if(data[j] >= data[i] && c[j]+1 > c[i]){
                c[i] = c[j]+1;
                s[i] = j;
            }
        }
    }
    int maxLen = *max_element(c.begin(), c.end());
    int i = distance(c.begin(), max_element(c.begin(), c.end()));
    vector<int> LDSpro1Seq;
    while(i!=-1){
        LDSpro1Seq.push_back(data[i]);
        i = s[i];
    }
    reverse(LDSpro1Seq.begin(), LDSpro1Seq.end());
    cout << "最长递减子序列的长度是: " << maxLen << endl;
    for(auto item : LDSpro1Seq){
        cout << item << "\t";
    }
    cout << endl;
}

int LDSpro2(vector<int>data, int*c, int*s, int n){
    int ret_index = -1;
    int maxLength = 0;
    // 从后往前找最长递增子序列
    for(int i = n-1; i >= 0; i--){
        c[i] = 1;
        s[i] = i;
        int max = 0;
        for(int j=i+1; j < n; j++){
```

```
            if (data[j] <= data[i] && max < c[j]){
                c[i] = c[j] + 1;
                s[i] = j;
                max = c[j];
                if(maxLength < c[i]){
                    maxLength = c[i];
                    ret_index = i;
                }
            }
        }
    }
    return ret_index;
}

vector<int> getLDSpro2(vector<int> data, int *s, int index){
    vector<int> retSeq;
    retSeq.push_back(data[index]);
    while(index != s[index]){
        index = s[index];
        retSeq.push_back(data[index]);
    }
    return retSeq;
}

int main(){
    vector <int> data1, data2;// 存放数据序列
    ifstream inpuFile("./附件 2.最大子段和输入数据-序列 1-2023.txt");
    if(!inpuFile.is_open()){
        cerr << "打开失败：附件 2，序列 1" << endl;
        return 1;
    }
    int value;
    while(inpuFile >> value){
        data1.push_back(value);
    }
    inpuFile.close();

    ifstream inputFile("./附件 2.最大子段和输入数据-序列 2-2023.txt");
    if(!inputFile.is_open()){
        cerr << "打开失败：附件 2，序列 2" << endl;
        return 1;
    }
```

```
    while(inputFile >> value){
        data2.push_back(value);
    }
    inputFile.close();


    int s1[data1.size()], s2[data2.size()], c1[data1.size()], c2[data2.size()];

    int index1, index2;
    index1 = LDSpro2(data1, c1, s1, data1.size());
    index2 = LDSpro2(data2, c2, s2, data2.size());
    vector<int> seq1 = getLDSpro2(data1, s1, index1);
    vector<int> seq2 = getLDSpro2(data2, s2, index2);
    cout << "序列 1 的最长递减序列长度为: " << seq1.size() << "  序列如下:" << endl;
    for(int i = 0; i < seq1.size(); i++){
            cout << seq1[i] << "\t";
        }
        cout << endl;
        cout << "序列 2 的最长递减序列长度为: " << seq2.size() << "  序列如下:" << endl;
        for(int i = 0; i < seq2.size(); i++){
            cout << seq2[i] << "\t";
        }
        cout << endl;
        LDSpro1(data1);
        LDSpro1(data2);
    }
```

3.4 执行结果

```
PS E:\bupt-homework\algorithm\Chapter3_dynamicProgramming\decrese> cd "e:\bupt-homework\algorithm\Chapter3_dynamicProgramming\decrese\" ; if ($?) { g++ decrease_seq.cpp -o decrease_seq } ; if ($?) { .\decrease_seq }
序列1的最长递减序列长度为: 40  序列如下:
99      99      95      91      89      87      87      79      76      72      65      61      60      56      5
6       51      50      47      36      31      27      27      27      19      3       0       0       0       -
4       -10     -14     -15     -17     -22     -31     -100    -200    -230    -301    -305
序列2的最长递减序列长度为: 30  序列如下:
100     49      47      47      39      38      37      34      34      34      33      28      27      24      2
4       5       -2      -6      -10     -11     -25     -25     -32     -38     -41     -42     -44     -70     -
304     -307
最长递减子序列的长度是: 40
99      99      95      91      89      87      87      79      76      72      65      61      60      56      5
6       51      50      47      36      31      27      27      27      19      3       0       0       0       -
4       -10     -14     -15     -17     -22     -31     -100    -200    -230    -301    -305
最长递减子序列的长度是: 30
100     49      47      47      39      38      37      34      34      34      33      28      27      24      2
4       5       -2      -6      -10     -11     -25     -25     -32     -38     -41     -42     -44     -70     -
304     -307
```

图 3-1 最长递减子序列执行结果图

对于序列 1,两个算法算出的最长递减子序列序列长度都为 40。对于序列 2,两个算法算出的最长递减子序列长度都为 30。

4 最大子段和问题

4.1 算法设计

4.1.1 算法设计思想

（1）判断最优子结构性质

对于一个数据序列 a[n]，它的前缀 a[j]的最大子段和表示的是从 a[i]到 a[j] 这一段序列所有值相加的总和。

因此我们可以划分出两种情况：

1. a[j]的最大子段和是单独的第 j 个元素

2. a[j]的最大子段和是从 a[i]到 a[j]的总和，其中 i≠j

对于一个最大子问题 a[j]，我们可以将其分解为两个子问题：

1.求 a[j-1]的最大子段和

2.求 a[j]的大小

一个大问题可以右多个子问题合成求解，证明了其具有最优子结构性质。

（2）问题递归结构/状态方程

对于所求的数据序列 a[n]的最大子段和，有如下关系：

$$\max_{1\le i\le j\le n}\sum_{k=i}^{j}a_k=\max_{1\le j\le n}\max_{1\le i\le j}\sum_{k=i}^{j}a_k=\max_{1\le j\le n}b[j]$$

我们设置一个 b[j]，b[j]从右端 a[j]处开始，一定包括了 a[j]

1. 当 b[j-1] > 0 时， b[j] = b[j-1] + a[j]

2. 当 b[j-1] <= 0 时， b[j] = a[j]

基于此，我们得到了一个递归方程：

$$b[j]=\max\{b[j-1]+a[j],a[j]\},\qquad 1\le j\le n$$

（3）算法实现步骤

（1）初始化变量 sum、index、数组 b、数组 c，b[0]=data[0]

（2）从序列左端第 1 个元素 a[1]开始，自左向右，根据递归方程，逐步计算 b[i]，1≤i≤n。

如果 b[i-1]大于 0，那么 b[i] = b[i-1] + data[i]，c[i]设成 c[i-1]

如果 b[i-1]小于等于 0，那么 b[i] = data[i]，c[i]设成 i

（3）判断最大的子段和有没有小于 b[i]，如果有，则更新。

（4）输出最大子段和，以及所在位置，还有里面的元素值。

4.1.2 算法实现

```
1.  void maxSumSeq(vector<int> data){
2.      int sum = 0, index = -1;
3.      int *b = new int[data.size()];
4.      int *c = new int[data.size()];
5.      b[0] = data[0];
6.      for(int i = 1; i < data.size(); i++){
7.          if(b[i-1] > 0){
8.              b[i] = b[i-1] + data[i];
9.              c[i] = c[i-1];
10.         }else{
11.             b[i] = data[i];
12.             c[i] = i;
13.         }
```

```
14.         if(b[i] > sum){
15.             sum = b[i];
16.             index = i;
17.         }
18.     }
19.     cout << "最大子段和为: " << sum <<"\t" << "所在位置是: "
20.         << c[index]+1 << "--" << index+1 << endl << "里面的元素如下: " << endl;
21.     for(int i = c[index]; i <= index; i++)
22.         cout << data[i] << "  ";
23.     cout << endl;
24. }
```

4.1.3 复杂度分析

时间复杂度:

只有一层循环，并且每层循环都只有 O(1)的算法步骤，因此时间复杂度为 O(n)。

空间复杂度:

由于需要 b 和 c 两个数组，数组大小都是 n，因此需要 O(n)的空间复杂度。

4.2 算法改进

4.2.1 算法改进思想

由于在每次迭代的过程中，b[i]的值要么由 b[i-1]得到，要么由数据序列 data 得到，所以我们可以将 b 数组简化为一个变量 b，从而降低空间复杂度。

具体实现步骤如下:

(1) 初始化变量 sum、index、b、c、end、start，b=data[0]。

(2) 从序列左端第 1 个元素 a[1]开始，自左向右，逐步计算 b，1≤i≤n。

如果 b 大于 0，那么 b = b + data[i]。

如果 b 小于等于 0，那么 b = data[i]，c 设成 i。

（3）判断最大的子段和有没有小于 b，如果有，则更新 b、start、end。

（4）输出最大子段和，以及所在位置，还有里面的元素值。

4.2.2 改进算法实现

```
1.  void maxSumSeqPro (vector<int> data){
2.      int sum = 0, b = 0, c = 0, start = 0, end = 0;
3.      for(int i = 0; i < data.size(); i++){
4.          if(b > 0){
5.              b += data[i];
6.          }else{
7.              b = data[i];
8.              c = i;
9.          }
10.         if(b > sum){
11.             sum = b;
12.             end = i;
13.             start = c;
14.         }
15.     }
16.     cout << "升级后的最大子段和为：" << sum <<"\t" << "所在位置是：" 
17.         << start+1 << "--" << end+1 << endl << "里面的元素如下：
    " << endl;
18.     for(int i = start; i <= end; i++)
19.         cout << data[i] << "  ";
20.     cout << endl;
21. }
```

4.2.3 改进算法复杂度分析

改进的算法时间复杂度并没有改变。

空间复杂度：因为原先用了一个数组 b 来保存最大子段和，用一个数组 c 来保存每段最大子段和的开始位置。改进之后现在将 b 数组换成单一变量 b，c 数

组换成了单一变量 c,又多了两个变量 start 和 end 用了存放整段最大子段和的起止位置。这样看来，除了必要的数据数组用的空间是 O(n)，其余的变量开销都为 O(1)。因此，空间复杂度降低到了 O(1)。

4.3 程序实现

```
1.  // 编码 GBK
2.  #include <iostream>
3.  #include <fstream>
4.  #include <vector>
5.  using namespace std;
6.
7.  void maxSumSeq(vector<int> data){
8.      int sum = 0, index = -1;
9.      int *b = new int[data.size()];
10.     int *c = new int[data.size()];
11.     b[0] = data[0];
12.     for(int i = 1; i < data.size(); i++){
13.         if(b[i-1] > 0){
14.             b[i] = b[i-1] + data[i];
15.             c[i] = c[i-1];
16.         }else{
17.             b[i] = data[i];
18.             c[i] = i;
19.         }
20.         if(b[i] > sum){
21.             sum = b[i];
22.             index = i;
23.         }
24.     }
25.     cout << "最大子段和为: " << sum <<"\t" << "所在位置是: "
26.         << c[index]+1 << "--" << index+1 << endl << "里面的元素如下:
    " << endl;
27.     for(int i = c[index]; i <= index; i++)
28.         cout << data[i] << "  ";
29.     cout << endl;
30. }
31.
32. void maxSumSeqPro (vector<int> data){
33.     int sum = 0, b = 0, c = 0, start = 0, end = 0;
```

```
    for(int i = 0; i < data.size(); i++){
        if(b > 0){
            b += data[i];
        }else{
            b = data[i];
            c = i;
        }
        if(b > sum){
            sum = b;
            end = i;
            start = c;
        }
    }
    cout << "升级后的最大子段和为: " << sum <<"\t" << "所在位置是: "
        << start+1 << "--" << end+1 << endl << "里面的元素如下: " << endl;
    for(int i = start; i <= end; i++)
        cout << data[i] << "  ";
    cout << endl;
}

int main(){

    vector <int> data1, data2;// 存放数据序列
    ifstream inpuFile("./附件 2.最大子段和输入数据-序列 1-2023.txt");
    if(!inpuFile.is_open()){
        cerr << "打开失败: 附件 2, 序列 1" << endl;
        return 1;
    }
    int value;
    while(inpuFile >> value){
        data1.push_back(value);
    }
    inpuFile.close();

    ifstream inputFile("./附件 2.最大子段和输入数据-序列 2-2023.txt");
    if(!inputFile.is_open()){
        cerr << "打开失败: 附件 2, 序列 2" << endl;
        return 1;
    }
    while(inputFile >> value){
        data2.push_back(value);
    }
```

```
76.     inputFile.close();
77.     int* c1 = new int[data1.size()];
78.     int* c2 = new int[data2.size()];
79.
80.     maxSumSeq(data1);
81.     maxSumSeq(data2);
82.     maxSumSeqPro(data1);
83.     maxSumSeqPro(data2);
84.     return 0;
85. }
```

4.4 执行结果

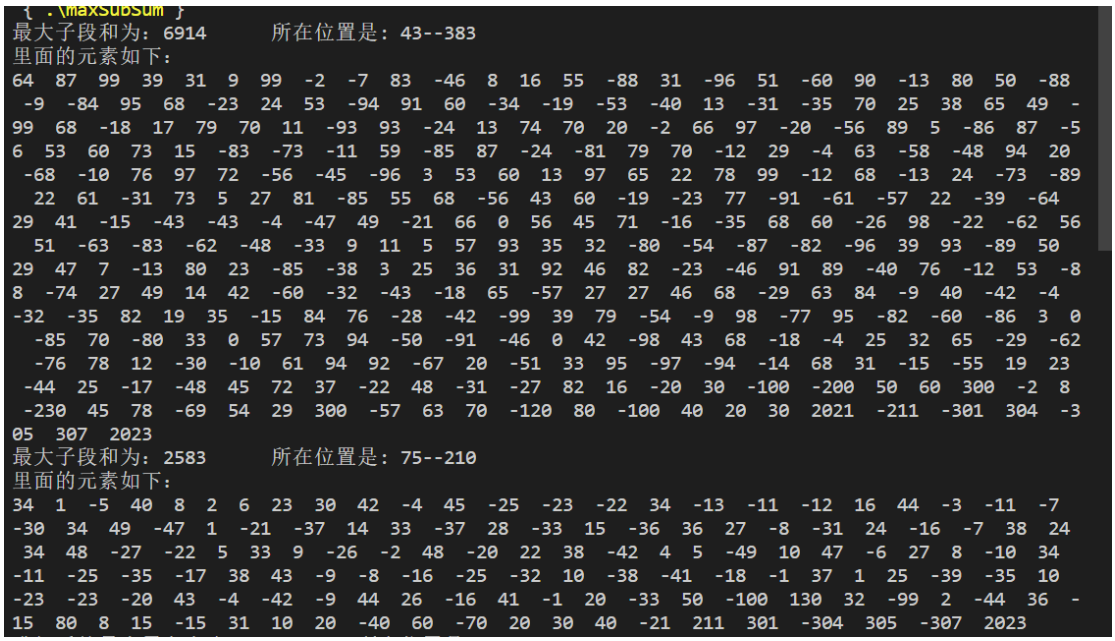

图 4-1 初始算法执行结果图

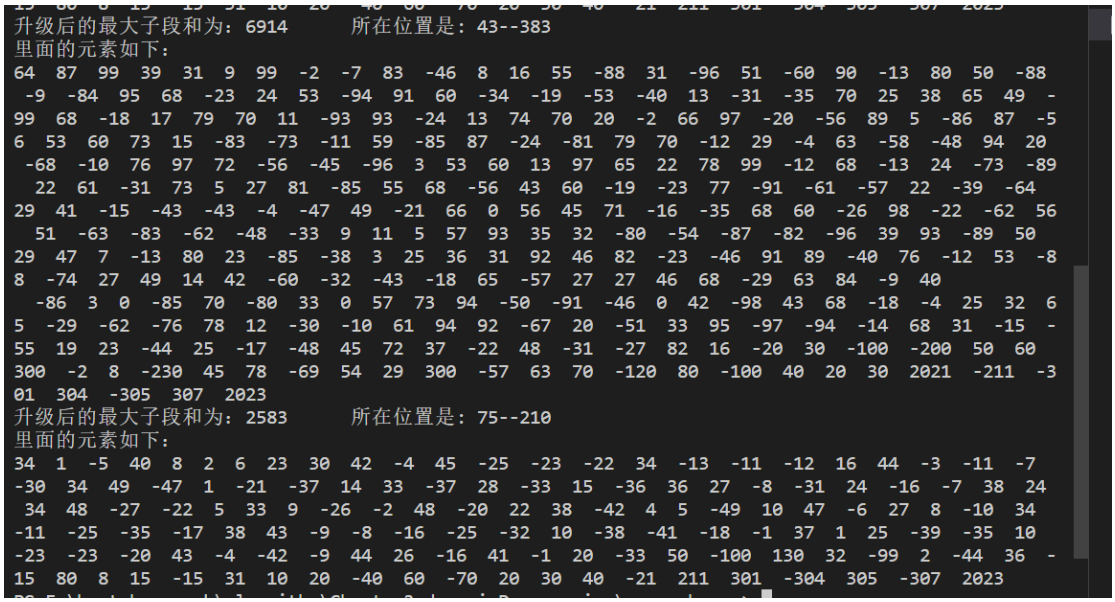

图 4-2 改进算法执行结果图

两个算法计算出的序列 1 的最大子段和都是 6914，所在位置是第 43 个元素到最后一个元素（以第一个元素为起点）。对于序列 2，算出的最大子段和都为 2583，所在位置是序列的第 75 个元素到最后一个元素。