**《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 4](#_Toc152921956)

[1.1 作业1.1 最长公共子序列 4](#_Toc152921957)

[1.2 作业1.2 最长递减子序列 5](#_Toc152921958)

[1.3 作业2 最大子段和 5](#_Toc152921959)

[1.4 评判标准 5](#_Toc152921960)

[2 最长公共子序列问题 6](#_Toc152921961)

[2.1 算法设计 6](#_Toc152921962)

[2.1.1 算法设计思想 6](#_Toc152921963)

[2.1.2 基础算法代码 8](#_Toc152921964)

[2.1.3 复杂性分析 9](#_Toc152921965)

[2.2 算法改进 10](#_Toc152921966)

[2.2.1 算法改进思想 10](#_Toc152921967)

[2.2.2 算法改进实现代码 10](#_Toc152921968)

[2.2.3 改进后的复杂度分析 11](#_Toc152921969)

[2.3 程序实现 12](#_Toc152921970)

[2.4 输出结果 16](#_Toc152921971)

[3 最长递减子序列问题 16](#_Toc152921972)

[3.1 算法设计 16](#_Toc152921973)

[3.1.1 算法设计思想 16](#_Toc152921974)

[3.1.2 算法基础代码 17](#_Toc152921975)

[3.1.3 复杂度分析 18](#_Toc152921976)

[3.2 最长递减子序列另一种实现 19](#_Toc152921977)

[3.2.1 逆向扫描算法思路 19](#_Toc152921978)

[3.2.2 算法实现 19](#_Toc152921979)

[3.2.3 复杂度分析 20](#_Toc152921980)

[3.3 程序实现 20](#_Toc152921981)

[3.4 执行结果 23](#_Toc152921982)

[4 最大子段和问题 24](#_Toc152921983)

[4.1 算法设计 24](#_Toc152921984)

[4.1.1 算法设计思想 24](#_Toc152921985)

[4.1.2 算法实现 25](#_Toc152921986)

[4.1.3 复杂度分析 26](#_Toc152921987)

[4.2 算法改进 26](#_Toc152921988)

[4.2.1 算法改进思想 26](#_Toc152921989)

[4.2.2 改进算法实现 27](#_Toc152921990)

[4.2.3 改进算法复杂度分析 27](#_Toc152921991)

[4.3 程序实现 28](#_Toc152921992)

[4.4 执行结果 30](#_Toc152921993)

# 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,其中1≤i≤m≤n，并且,则称序列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xm=yn，则zk=xm=yn，且Z(k-1)={z1, z2,…,zk-1 }是X(m-1)= {x1,x2,…,xm-1}和Y(n-1)={y1,y2,…,yn-1}的最长公共子序列。

- 情况2：若xm≠yn，zk=yn，则Z(k)是X(m-1)和Y(n)的的最长公共子序列。

- 情况3：若xm≠yn且zk≠yn，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

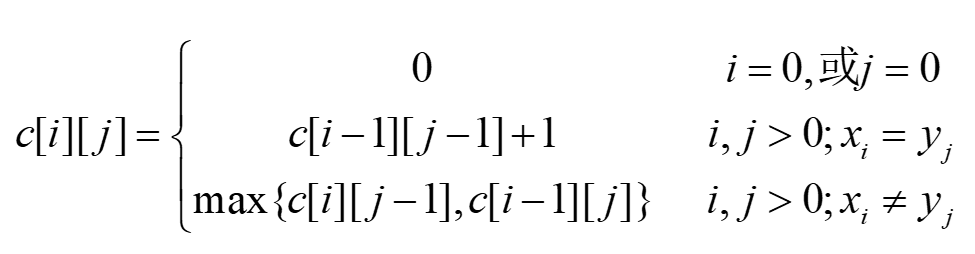
因此， 最长公共子序列问题具有最优子结构性质。因此，根据小规模子问题<m-1, n-1><m-1, n><m, n-1>的解，可以自下而上构造问题<m, n>的解。

#### （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)={x1,x2,…,xi}，Y(j)={y1,y2,…,yj}，当i=0或j=0时，即其中1个序列为空，则空序列是Xi和Yj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：



#### （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，Xi和Yj的最长公共子序列由Xi-1和Yj-1的最长公共子序列，在尾部加上xi得到。

3.如果为2，说明是情况2，Xi和Yj的最长公共子序列与Xi-1和Yj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，Xi和Yj的最长公共子序列与Xi和Yj-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**;
4. // 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
5. **if**(b[i][j] == 1){
6. LCS(i-1, j-1, x, b);
7. cout << x[i];
8. }**else** **if**(b[i][j] == 2){
9. LCS(i-1, j, x, b);
10. }**else**{
11. LCS(i, j-1, x, b);
12. }
13. }
15. **void** LCSLengthPro(**int** m, **int** n, string x, string y, **int** \*\*c){
16. **int** i, j;
17. // Y[j]空时，最小子问题<i, 0>
18. **for**(i = 1; i <= m; i++)
19. c[i][0] = 0;
21. // X[i]空时，最小子问题<0, j>
22. **for**(i = 1;i <= n; i++)
23. c[0][i] = 0;
24. **for**(i = 1; i <= m; i++){
25. **for**(j = 1; j <= n; j++){
26. **if**(x[i] == y[j]){
27. c[i][j] = c[i-1][j-1] + 1;
28. }**else** **if**(c[i-1][j] >= c[i][j-1]){
29. c[i][j] = c[i-1][j];
30. }**else**{
31. c[i][j] = c[i][j-1];
32. }
33. }
34. }
35. }

### 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;
20. // X[i]空时，最小子问题<0, j>
21. **for**(i = 1;i <= n; i++)
22. c[0][i] = 0;
23. **for**(i = 1; i <= m; i++){
24. **for**(j = 1; j <= n; j++){
25. **if**(x[i] == y[j]){
26. c[i][j] = c[i-1][j-1] + 1;
27. }**else** **if**(c[i-1][j] >= c[i][j-1]){
28. c[i][j] = c[i-1][j];
29. }**else**{
30. c[i][j] = c[i][j-1];
31. }
32. }
33. }
34. }
36. **void** LCSpro(**int** i, **int** j, string x, **int** \*\*c){
37. **if**(i ==0 || j== 0){
38. **return**;
39. }
40. // 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
41. **if**(checkBIJ(c, i, j) == 1){
42. LCSpro(i-1, j-1, x, c);
43. cout << x[i];
44. }**else** **if**(checkBIJ(c, i, j) == 2){
45. LCSpro(i-1, j, x, c);
46. }**else**{
47. LCSpro(i, j-1, x, c);
48. }
49. }

### 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>
7. **using** **namespace** std;
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. }
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. }
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;
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. }
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. }
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;
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. }
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. }
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. }
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];
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);
126. newArray(bcd, C.length()+1, B.length()+1);
127. newArray(ccd, C.length()+1, B.length()+1);
129. newArray(bad, A.length()+1, D.length()+1);
130. newArray(cad, A.length()+1, D.length()+1);
132. newArray(bcb, C.length()+1, B.length()+1);
133. newArray(ccb, C.length()+1, B.length()+1);
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);
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

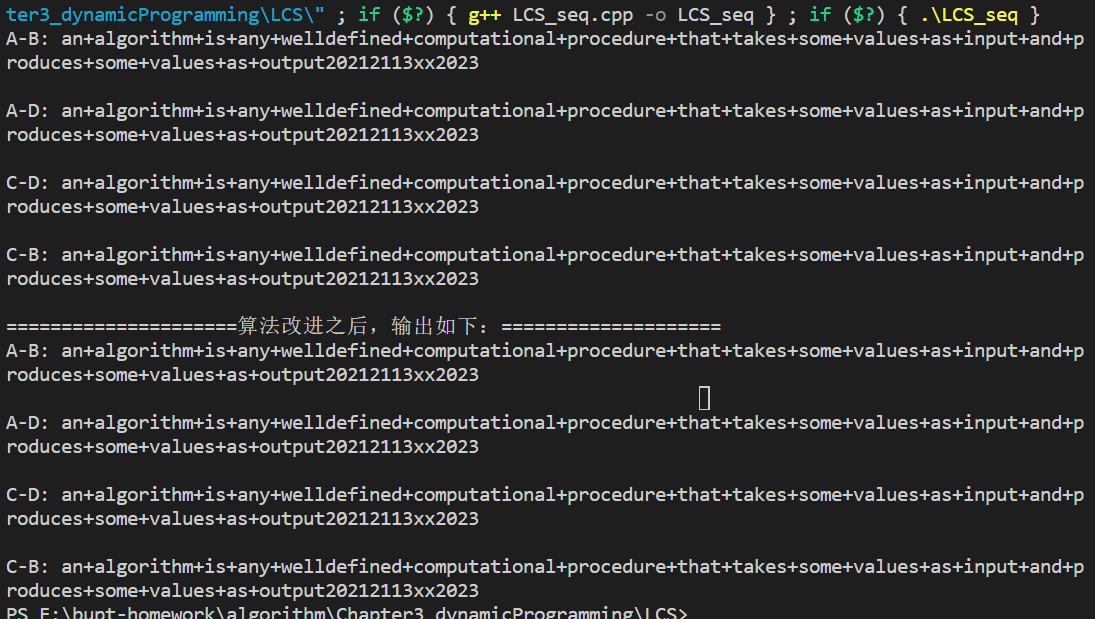


图2-1 最长公共子序列输出结果图

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**X(m)**={x1,x2,…,xm}最长递减子序列为**Z(k)**={z1,z2,…,zk} ，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){
2. **int** n = data.size();
3. vector<**int**> c(n, 1);// 初始化全为1
4. vector<**int**> s(n, -1);
5. **for**(**int** i = 1; i < n ;i++){
6. **for**(**int** j = 0; j < i; j++){
7. **if**(data[j] >= data[i] && c[j]+1 > c[i]){
8. c[i] = c[j]+1;
9. s[i] = j;
10. }
11. }
12. }
13. **int** maxLen = \*max\_element(c.begin(), c.end());
14. **int** i = distance(c.begin(), max\_element(c.begin(), c.end()));
15. vector<**int**> ldsSeq;
16. **while**(i!=-1){
17. ldsSeq.push\_back(data[i]);
18. i = s[i];
19. }
20. reverse(ldsSeq.begin(), ldsSeq.end());
21. cout << "最长递减子序列的长度是：" << maxLen << endl;
22. **for**(auto item : ldsSeq){
23. cout << item << "\t";
24. }
25. cout << endl;
26. }

### 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;
12. s[i] = j;
13. max = c[j];
14. **if**(maxLength < c[i]){
15. maxLength = c[i];
16. ret\_index = i;
17. }
18. }
19. }
20. }
21. **return** ret\_index;
22. }
24. vector<**int**> getLDSpro(vector<**int**> data, **int** \*s, **int** index){
25. vector<**int**> retSeq;
26. retSeq.push\_back(data[index]);
27. **while**(index != s[index]){
28. index = s[index];
29. retSeq.push\_back(data[index]);
30. }
31. **return** retSeq;
32. }

### 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

1. // 编码 GBK
2. #include <iostream>
3. #include <fstream>
4. #include <vector>
5. #include <algorithm>
6. **using** **namespace** std;
8. **void** LDSpro1(vector<**int**>data){
9. **int** n = data.size();
10. vector<**int**> c(n, 1);// 初始化全为1
11. vector<**int**> s(n, -1);
12. **for**(**int** i = 1; i < n ;i++){
13. **for**(**int** j = 0; j < i; j++){
14. **if**(data[j] >= data[i] && c[j]+1 > c[i]){
15. c[i] = c[j]+1;
16. s[i] = j;
17. }
18. }
19. }
20. **int** maxLen = \*max\_element(c.begin(), c.end());
21. **int** i = distance(c.begin(), max\_element(c.begin(), c.end()));
22. vector<**int**> LDSpro1Seq;
23. **while**(i!=-1){
24. LDSpro1Seq.push\_back(data[i]);
25. i = s[i];
26. }
27. reverse(LDSpro1Seq.begin(), LDSpro1Seq.end());
28. cout << "最长递减子序列的长度是：" << maxLen << endl;
29. **for**(auto item : LDSpro1Seq){
30. cout << item << "\t";
31. }
32. cout << endl;
33. }
35. **int** LDSpro2(vector<**int**>data, **int**\*c, **int**\*s, **int** n){
36. **int** ret\_index = -1;
37. **int** maxLength = 0;
38. // 从后往前找最长递增子序列
39. **for**(**int** i = n-1; i >= 0; i--){
40. c[i] = 1;
41. s[i] = i;
42. **int** max = 0;
43. **for**(**int** j=i+1; j < n; j++){
44. **if** (data[j] <= data[i] && max < c[j]){
45. c[i] = c[j] + 1;
46. s[i] = j;
47. max = c[j];
48. **if**(maxLength < c[i]){
49. maxLength = c[i];
50. ret\_index = i;
51. }
52. }
53. }
54. }
55. **return** ret\_index;
56. }
58. vector<**int**> getLDSpro2(vector<**int**> data, **int** \*s, **int** index){
59. vector<**int**> retSeq;
60. retSeq.push\_back(data[index]);
61. **while**(index != s[index]){
62. index = s[index];
63. retSeq.push\_back(data[index]);
64. }
65. **return** retSeq;
66. }
68. **int** main(){
69. vector <**int**> data1, data2;// 存放数据序列
70. ifstream inpuFile("./附件2.最大子段和输入数据-序列1-2023.txt");
71. **if**(!inpuFile.is\_open()){
72. cerr << "打开失败：附件2，序列1" << endl;
73. **return** 1;
74. }
75. **int** value;
76. **while**(inpuFile >> value){
77. data1.push\_back(value);
78. }
79. inpuFile.close();
81. ifstream inputFile("./附件2.最大子段和输入数据-序列2-2023.txt");
82. **if**(!inputFile.is\_open()){
83. cerr << "打开失败：附件2，序列2" << endl;
84. **return** 1;
85. }
86. **while**(inputFile >> value){
87. data2.push\_back(value);
88. }
89. inputFile.close();

92. **int** s1[data1.size()], s2[data2.size()], c1[data1.size()], c2[data2.size()];
93. **int** index1, index2;
94. index1 = LDSpro2(data1, c1, s1, data1.size());
95. index2 = LDSpro2(data2, c2, s2, data2.size());
96. vector<**int**> seq1 = getLDSpro2(data1, s1, index1);
97. vector<**int**> seq2 = getLDSpro2(data2, s2, index2);
98. cout << "序列1的最长递减序列长度为: " << seq1.size() << "  序列如下:" << endl;
99. **for**(**int** i = 0; i < seq1.size(); i++){
100. cout << seq1[i] << "\t";
101. }
102. cout << endl;
103. cout << "序列2的最长递减序列长度为: " << seq2.size() << "  序列如下:" << endl;
104. **for**(**int** i = 0; i < seq2.size(); i++){
105. cout << seq2[i] << "\t";
106. }
107. cout << endl;
108. LDSpro1(data1);
109. LDSpro1(data2);
110. }

## 3.4 执行结果

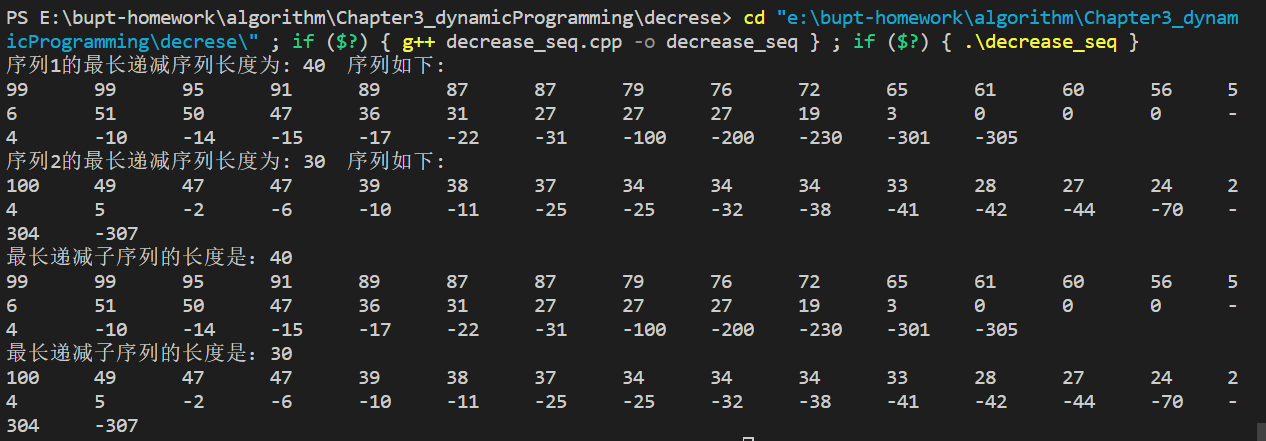


图3-1 最长递减子序列执行结果图

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：

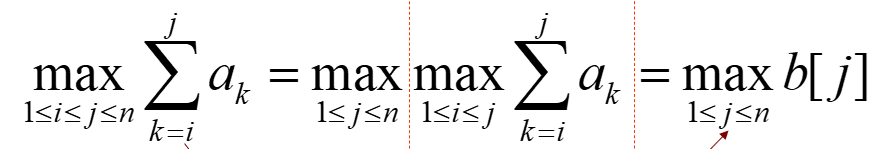
1.求a[j-1]的最大子段和

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：



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：



#### （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;
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. }
32. **void** maxSumSeqPro (vector<**int**> data){
33. **int** sum = 0, b = 0, c = 0, start = 0, end = 0;
34. **for**(**int** i = 0; i < data.size(); i++){
35. **if**(b > 0){
36. b += data[i];
37. }**else**{
38. b = data[i];
39. c = i;
40. }
41. **if**(b > sum){
42. sum = b;
43. end = i;
44. start = c;
45. }
46. }
47. cout << "升级后的最大子段和为：" << sum <<"\t" << "所在位置是: "
48. << start+1 << "--" << end+1 << endl << "里面的元素如下：" << endl;
49. **for**(**int** i = start; i <= end; i++)
50. cout << data[i] << "  ";
51. cout << endl;
52. }
54. **int** main(){
56. vector <**int**> data1, data2;// 存放数据序列
57. ifstream inpuFile("./附件2.最大子段和输入数据-序列1-2023.txt");
58. **if**(!inpuFile.is\_open()){
59. cerr << "打开失败：附件2，序列1" << endl;
60. **return** 1;
61. }
62. **int** value;
63. **while**(inpuFile >> value){
64. data1.push\_back(value);
65. }
66. inpuFile.close();
68. ifstream inputFile("./附件2.最大子段和输入数据-序列2-2023.txt");
69. **if**(!inputFile.is\_open()){
70. cerr << "打开失败：附件2，序列2" << endl;
71. **return** 1;
72. }
73. **while**(inputFile >> value){
74. data2.push\_back(value);
75. }
76. inputFile.close();
77. **int**\* c1 = **new** **int**[data1.size()];
78. **int**\* c2 = **new** **int**[data2.size()];
80. maxSumSeq(data1);
81. maxSumSeq(data2);
82. maxSumSeqPro(data1);
83. maxSumSeqPro(data2);
84. **return** 0;
85. }

## 4.4 执行结果

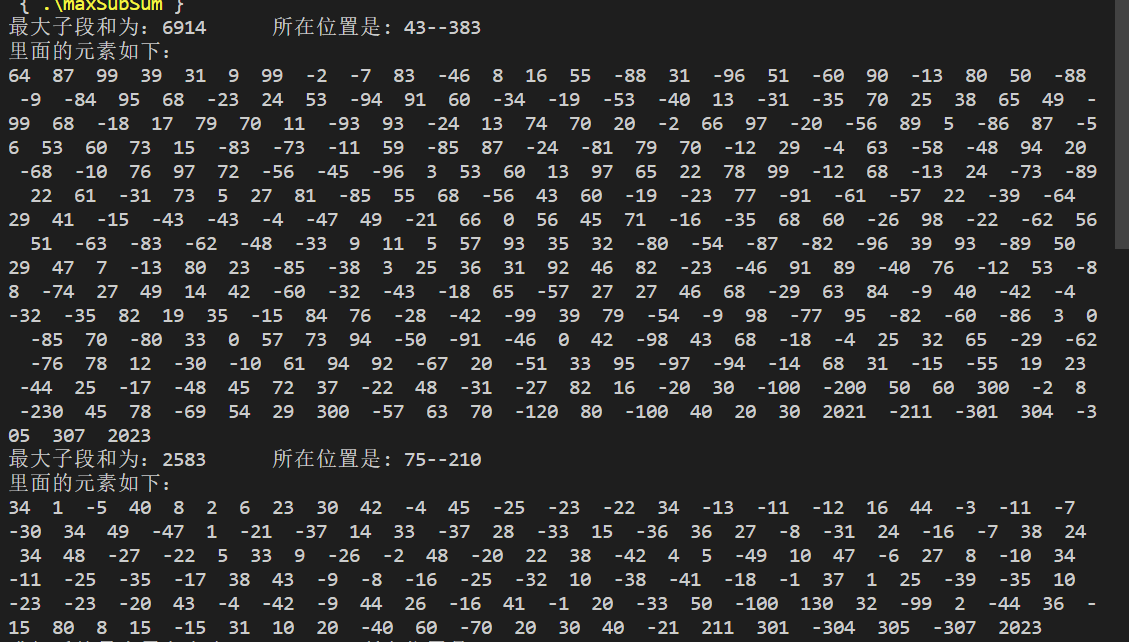


图4-1 初始算法执行结果图

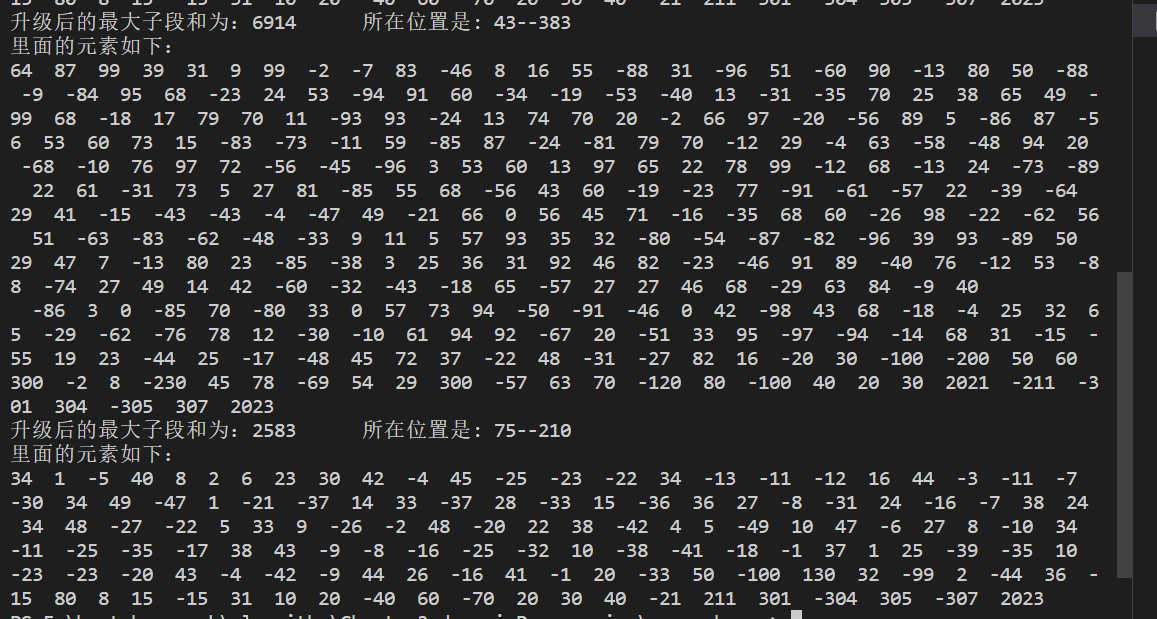


图4-2改进算法执行结果图

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。