

JSS MAHAVIDYAPEETHA
JSS SCIENCE AND TECHNOLOGY UNIVERSITY
SRI JAYACHAMARAJENDRA COLLEGE OF ENGINEERING
JSS Technical Institutions Campus, Mysuru – 570 006



A seminar report on

“LONGEST COMMON SUBSEQUENCE DISTANCE”

(MINIMUM NUMBER OF INSERTIONS AND DELETIONS TO CONVERT ONE STRING TO OTHER)

20IS410

**Bachelor of Engineering
in
INFORMATION SCIENCE AND ENGINEERING**

Submitted by,

**SUHAS M S 01JST20IS045
SUPRITH P 01JST20IS047
VINAY PATIL 01JST20IS052**

Submitted to

SINDHU G
Assistant professor
ISE Department

AUGUST 2021

TABLE OF CONTENTS:

1.INTRODUCTION.....	3
2.EXAMPLE.....	4
3.LCS.....	4
4.ALGORITHM.....	8
5.ANALYSIS OF LCS DISTANCE.....	9
6.RESULTS.....	10

INTRODUCTION:

Sometimes it is important to know how similar or dissimilar two strings are. There are many applications of this concept. A common problem in text editing is that of finding the occurrence of a given string in a file. This is usually done to locate some content in the file or to replace the occurrence of one string by another. Apart from text editing, spelling correction has numerous applications in text and document retrieval word processing, designing effective spelling checkers.

1)File Comparison:

Unix command `diff f1 f2` finds the difference between two files `f1` and `f2` by using LCS distance to convert `f1` to `f2`. `diff` treats whole line as a "character" and uses special LCS distance algorithm to find matches between elements of two strings(files).

2)Spelling Correction:

Algorithms related to LCS distance may be used in spelling correctors. If a text contains a word that is not in dictionary then a close word is suggested.

CONCEPT:

- Given two strings 'str1' and 'str2' of size m and n respectively.
 - The task is to convert one string 'str1' to 'str2' that is to delete and insert the minimum number of characters from/in str1 to transform it into str2.
 - It could be possible that the same character needs to be deleted from one point of str1 and inserted to some another point.
- Algorithm used is Longest Common Subsequence using Dynamic Programming approach.

Example 1:

Input: str1 = "heap", str2 = "pea"

Output : Minimum Deletion = 2 and Minimum Insertion = 1

Explanation: 'p' and 'h' are deleted from 'heap'. Then, 'p' is inserted at the beginning.

Here, even though 'p' was required yet it was deleted first from its position and then it is inserted to some other position. Thus, 'p' contributes one to the deletion count and one to the insertion count.

Example 2:

Input: str1 = "abcd", str2 = "anc"

Output: Minimum Deletion = 2 and Minimum Insertion = 1

Explanation: 'a' and 'd' are deleted from 'abcd'. Then, 'n' is inserted in between remaining 'ac' string and 'anc' is obtained. Here, 'a' and 'd' are deleted hence they contribute to two delete count and adding 'n' contributes to one insertion count.

LONGEST COMMON SUBSEQUENCE:

The longest common subsequence (LCS) is defined as the longest subsequence that is common to all the given sequences, provided that the elements of the subsequence are not required to occupy consecutive positions within the original sequences.

If S_1 and S_2 are the two given sequences then, Z is the common subsequence of S_1 and S_2 if Z is a subsequence of both S_1 and S_2 . Furthermore, Z must be a strictly increasing sequence of the indices of both S_1 and S_2 .

In a strictly increasing sequence, the indices of the elements chosen from the original sequences must be in ascending order in Z .

If $S_1 = \{B, C, D, A, A, C, D\}$

Then, $\{A, D, B\}$ cannot be a subsequence of S_1 as the order of the elements is not the same (ie. not strictly increasing sequence).

Let us consider the following example

$S1 = \{B, C, D, A, A, C, D\}$

$S2 = \{A, C, D, B, A, C\}$

Then, common subsequences are $\{B, C\}, \{C, D, A, C\}, \{D, A, C\}, \{A, A, C\}, \{A, C\}, \{C, D\}, \dots$

Among these subsequences, $\{C, D, A, C\}$ is the longest common subsequence.

Let us take two sequences:

X

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

 The first sequence

Y

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

 Second Sequence

The following steps are followed for finding the longest common subsequence.

1. Create a table of dimension $n+1*m+1$ where n and m are the lengths of **X** and **Y** respectively. The first row and the first column are filled with zeros.

		C	B	D	A
	0	0	0	0	0
A	0				
C	0				
A	0				
D	0				
B	0				

2. Fill each cell of the table using the following logic.

- If the character corresponding to the current row and current column are matching, then fill the current cell by adding one to the diagonal element. Point an arrow to the diagonal cell.
- Else take the maximum value from the previous column and previous row element for filling the current cell. Point an arrow to the cell with maximum value. If they are equal, point to any of them.

		C	B	D	A
	0	0	0	0	0
A	0	0	0	0	1
C	0				
A	0				
D	0				
B	0				

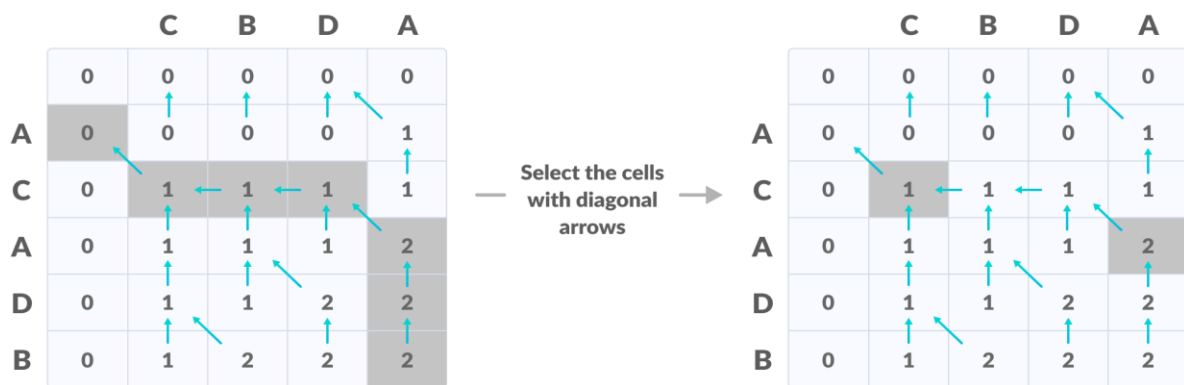
- Step 2 is repeated until the table is filled.

		C	B	D	A
	0	0	0	0	0
A	0	0	0	0	1
C	0	1	1	1	1
A	0	1	1	1	2
D	0	1	1	2	2
B	0	1	2	2	2

- The value in the last row and the last column is the length of the longest common subsequence. The bottom right corner is the length of the LCS.

		C	B	D	A
A	0	0	0	0	0
C	0	0	0	0	1
A	0	1	1	1	1
D	0	1	1	1	2
B	0	1	2	2	2

7. In order to find the longest common subsequence, start from the last element and follow the direction of the arrow. The elements corresponding to () symbol form the longest common subsequence. Create a path according to the arrows.



Thus, the longest common subsequence is CA.

C A

LCS

Efficiency of dynamic programming over recursive algorithm to solve LCS:

- The method of dynamic programming reduces the number of function calls. It stores the result of each function call so that it can be used in future calls without the need for redundant calls. In the above dynamic algorithm, the results obtained from each comparison between elements of X and the elements of Y are stored in a table so that they can be used in future computations. So, the time taken by a dynamic approach is

the time taken to fill the table (ie. $O(mn)$). Whereas, the recursion algorithm has the complexity of $2^{\max(m, n)}$.

Algorithm for LCS distance:

- str1 and str2 be the given strings.
- m and n be their lengths respectively.
- len be the length of the longest common subsequence of str1 and str2
- minimum number of deletions $\text{minDel} = m - \text{len}$ (as we only need to delete from str1 because we are reducing it to str2)
- minimum number of Insertions $\text{minInsert} = n - \text{len}$ (as we are inserting x in str1 , x is a number of characters in str2 which are not taking part in the string which is longest common subsequence)

Complete algorithm:


```

int lcs(string X[], string Y[], int m, int n)
{
    int LCS[m + 1][n + 1]
    for(int i = 0; i <= m; i = i + 1)
        LCS[i][0] = 0
    for(int j = 0; j <= n; j = j + 1)
        LCS[0][j] = 0

    for(int i = 1; i <= m; i = i + 1)
    {
        for(int j = 1; j <= n; j = j + 1)
        {
            if(X[i - 1] == Y[j - 1])
                LCS[i][j] = 1 + LCS[i - 1][j - 1]
            else
                LCS[i][j] = max (LCS[i - 1][j], LCS[i][j - 1])
        }
    }
    return LCS[m][n]
}

```

ANALYSIS OF LCS DISTANCE:

TIME COMPLEXITY:

Since we are using two for loops for both the strings ,therefore the time complexity of finding the longest common subsequence using dynamic programming approach is **$O(n * m)$** where n and m are the lengths of the strings.

- Worst case time complexity: **$O(n*m)$**
- Average case time complexity: **$O(n*m)$**
- Best case time complexity: **$O(n*m)$**

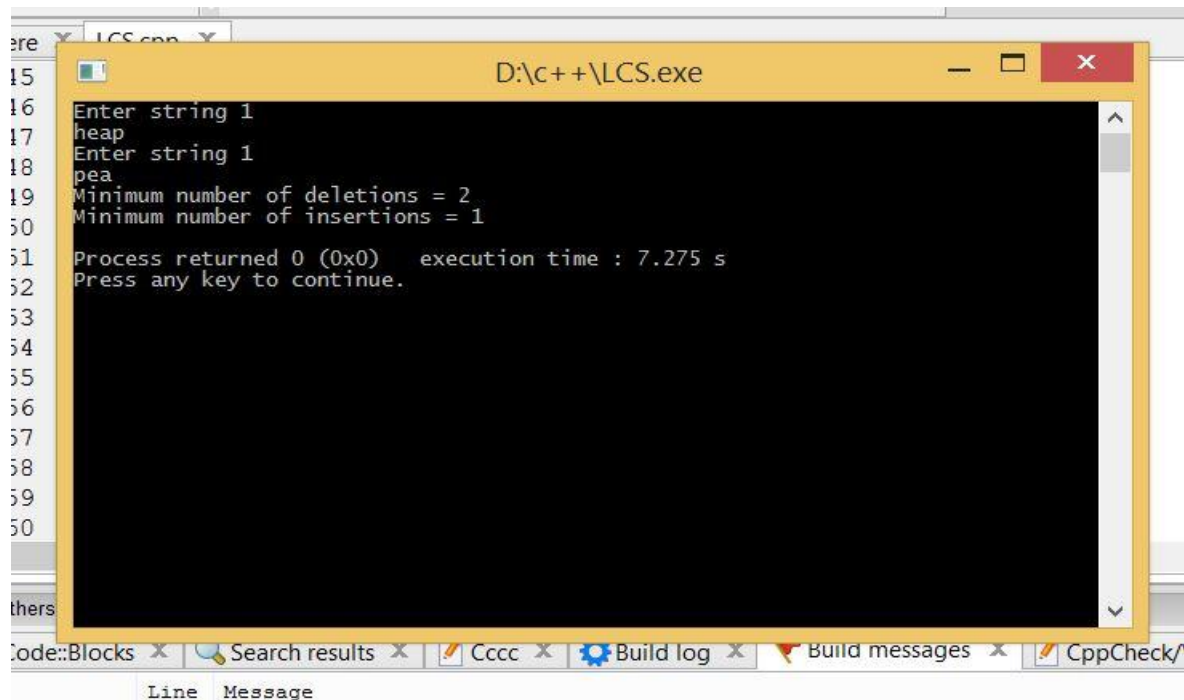
The time might sometimes be better if not all array entries get filled out. For instance, if the two strings match exactly, we'll only fill in diagonal entries and the algorithm will be fast.

SPACE COMPLEXITY:

Since this implementation involves only n rows and m columns for building $dp[][]$, therefore, the space complexity would be $O(n * m)$.

RESULTS:

RUN 1:

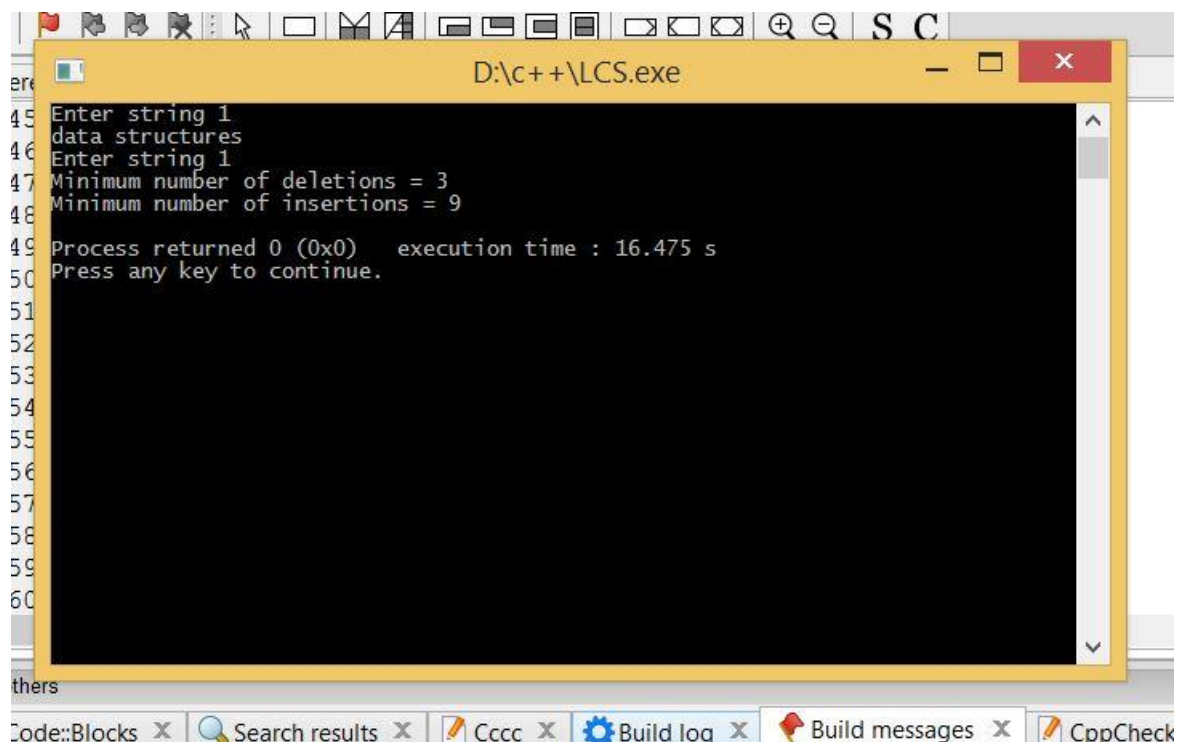


```
15
16 Enter string 1
17 heap
18 Enter string 1
19 pea
19 Minimum number of deletions = 2
20 Minimum number of insertions = 1
51 Process returned 0 (0x0)   execution time : 7.275 s
52 Press any key to continue.
53
54
55
56
57
58
59
60
```

Code::Blocks | Search results | Cccc | Build log | Build messages | CppCheck/

Line	Message
------	---------

RUN 2:



```
45 Enter string 1
46 data structures
47 Enter string 1
48 data structures
47 Minimum number of deletions = 3
48 Minimum number of insertions = 9
49 Process returned 0 (0x0)   execution time : 16.475 s
50 Press any key to continue.
51
52
53
54
55
56
57
58
59
60
```

Code::Blocks | Search results | Cccc | Build log | Build messages | CppCheck/