Green University of Bangladesh

Department of Computer Science and Engineering (CSE)

Faculty of Sciences and Engineering

Semester: (Fall, Year: 2023), B.Sc. in CSE (Day)

Lab Report NO: 10

Course Title	**: Algorithm Lab**
Course Code	**: CSE 204**
Section	**: D - 9**

Lab Experiment Name : Implemention of Longest Common Subsequence (LCS) Algorithm

<u>Student Details</u>

Name		**ID**
1.	Jahidul Islam	221002504

Lab Date	**: 04/12/2023**
Submission Date	**: 31/12/2023**
Course Teacher's Name	**: Md. Abu Rahman Refat**

<u>Lab Report Status</u>

Marks: ………………………………… **Signature:.....................**

Comments:... **Date:..............................**

1. TITLE:

Implemention of Longest Common Subsequence (LCS) Algorithm

2. OBJECTIVES/AIM:

The primary objectives or aim of this lab experiment are:

- Understand the significance of efficient string searching algorithms in computer science.
- Explore and implement the KMP algorithm as a solution for improving string matching efficiency.
- Develop a Java program to apply the KMP algorithm in the context of finding indexes where a given pattern matches a given array.
- Analyze the time complexity and advantages of the KMP algorithm in comparison to other string-matching algorithm

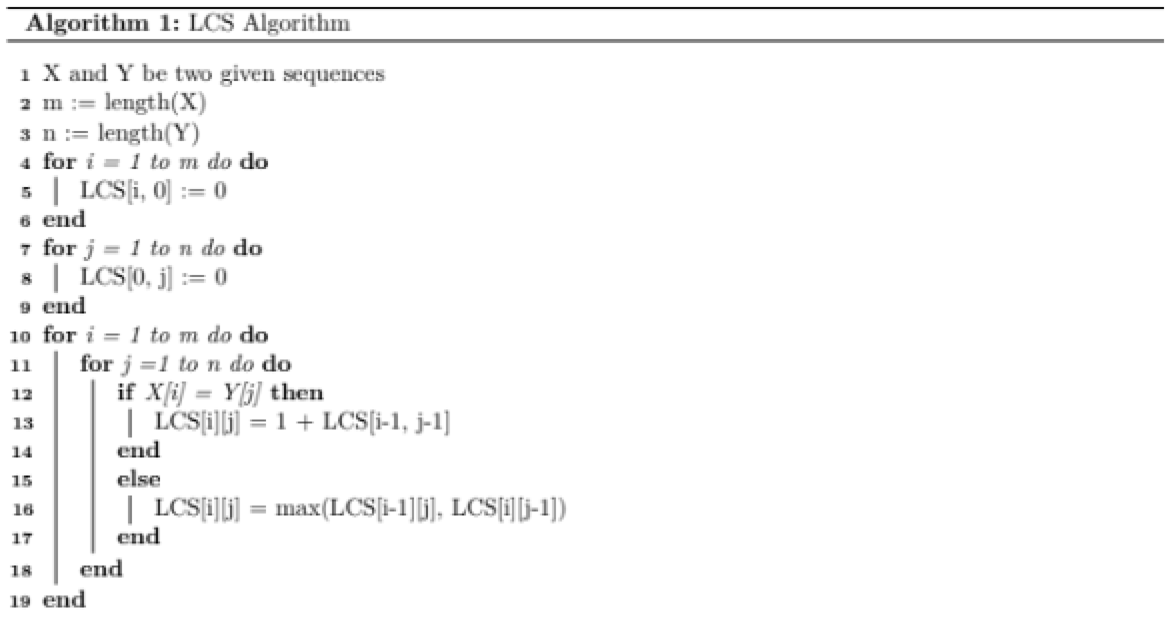
3. PROBLEM ANALYSIS & PROCEDURE:

A subsequence is a group of sequences that appear in the same relative order, whether they are adjacent or not. The longest subsequence that is shared by all the given sequences, is known as the longest common subsequence (LCS), assuming that the elements of the subsequence are not needed to occupy consecutive position within the original sequences.

Let the sequences are X=x1,x2,x3,...,xm and Y=y1,y2,y3,...,ym. We will use the following steps to determine the length of the longest common subsequence.

- Create an empty adjacency table with the dimensions n m, where n is the length of the X sequence and m is the length of the Y sequence. The elements in sequence X are represented by the table’s rows, and the elements in sequence Y are represented by the table’s columns.
- Rows and columns starting at zero must contain only zeros. And the remaining values are filled in based on different cases, by maintaining a counter value.
- The number is increased by one if it comes across a common element in both the X and Y sequences.
- To fill in T[i, j] if the counter does not pass into any common elements in the X and Y sequences, choose the biggest value between T[i-1, j] and T[i, j-1].
- Once the table is filled, backtrack from the last value in the table. Backtracking here is done by tracing the path where the counter incremented first.
- The longest common subsequence which can be determined by observing the path’s elements.

4. ALGORITHM:



Algorithm 1: LCS Algorithm

```
X and Y be two given sequences
m := length(X)
n := length(Y)
for i = 1 to m do do
    LCS[i, 0] := 0
end
for j = 1 to n do do
    LCS[0, j] := 0
end
for i = 1 to m do do
    for j =1 to n do do
        if X[i] = Y[j] then
            LCS[i][j] = 1 + LCS[i-1, j-1]
        end
        else
            LCS[i][j] = max(LCS[i-1][j], LCS[i][j-1])
        end
    end
end
```

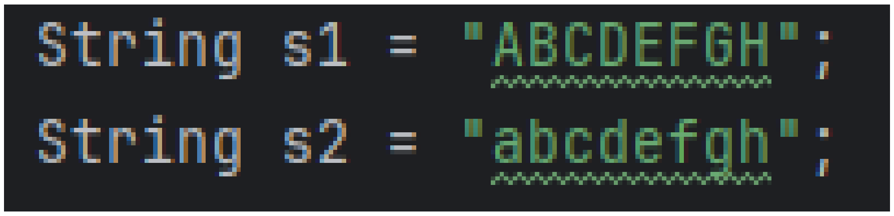
5. IMPLEMENTATION:

Code in Java:

```
import java.util.*;

class TUF {
    // Recursive function to find the length of the Longest Common
Subsequence (LCS)
    static int lcsUtil(String s1, String s2, int ind1, int ind2, int[][] dp)
{
        // Base case: If either of the strings reaches the end, return 0
        if (ind1 < 0 || ind2 < 0)
            return 0;

        // If the result for this subproblem has already been calculated,
return it
        if (dp[ind1][ind2] != -1)
            return dp[ind1][ind2];

        // If the characters at the current indices are the same, increment
the LCS length
        if (s1.charAt(ind1) == s2.charAt(ind2))
            return dp[ind1][ind2] = 1 + lcsUtil(s1, s2, ind1 - 1, ind2 - 1,
dp);

            // If the characters are different, choose the maximum LCS length
by either
```

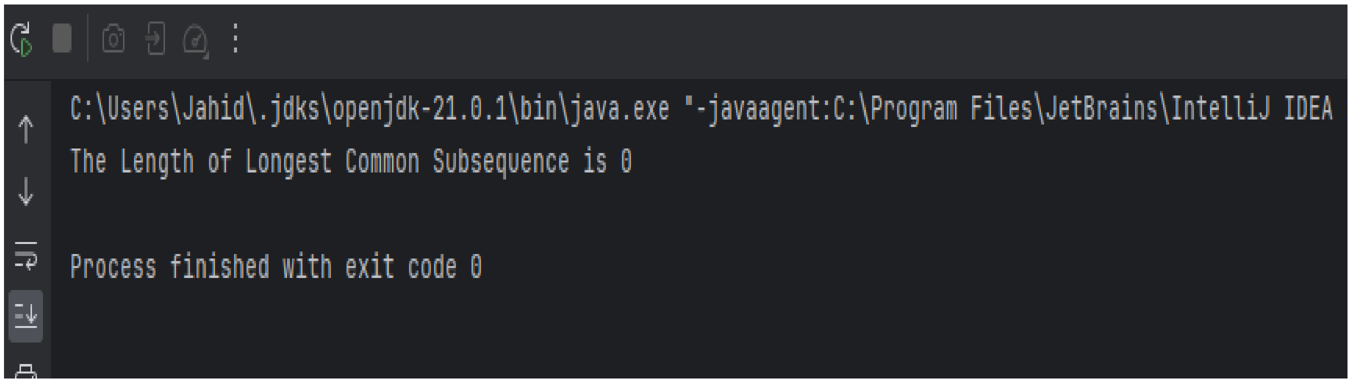
```java
            // skipping a character in s1 or skipping a character in s2
        else
            return dp[ind1][ind2] = Math.max(lcsUtil(s1, s2, ind1, ind2 - 1, 
dp),
                    lcsUtil(s1, s2, ind1 - 1, ind2, dp));
    }

    // Function to find the length of the Longest Common Subsequence (LCS)
    static int lcs(String s1, String s2) {
        int n = s1.length();
        int m = s2.length();

        // Create a 2D array to store results of subproblems
        int dp[][] = new int[n][m];

        // Initialize the dp array with -1 to indicate that subproblems are 
not solved yet
        for (int rows[] : dp)
            Arrays.fill(rows, -1);

        // Call the recursive function to find the LCS length
        return lcsUtil(s1, s2, n - 1, m - 1, dp);
    }

    public static void main(String args[]) {
        String s1 = "ABCDEFGH";
        String s2 = "abcdefgh";

        // Call the lcs function and print the result
        System.out.println("The Length of Longest Common Subsequence is " + 
lcs(s1, s2));
    }
}
```

6. INPUT STRING:



```java
String s1 = "ABCDEFGH";
String s2 = "abcdefgh";
```

7. TEST RESULT / OUTPUT:



```
C:\Users\Jahid\.jdks\openjdk-21.0.1\bin\java.exe "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA
The Length of Longest Common Subsequence is 0

Process finished with exit code 0
```

8. DISCUSSION AND ANALYSIS:

The discussion and analysis phase centers around evaluating the effectiveness and efficiency of the implemented Longest Common Subsequence (LCS) algorithm:

We implemented the code for **multiple pairs of given strings and patterns.**

- **Time Complexity Analysis:**
  i. The time complexity for computing the length of the LCS between two sequences of lengths N and M is determined by the dynamic programming approach. The algorithm builds a table to store intermediate results, and each cell is computed in constant time. Hence, the **overall time complexity is O(N*M).**

- **Advantages of the KMP Algorithm:**

  i. **Efficient Dynamic Programming Approach**:
     - The LCS algorithm employs dynamic programming, providing an efficient solution to finding the longest common subsequence between two sequences.

  ii. **Applicability to Various Domains:**
     - The algorithm is versatile and finds applications in diverse domains, including bioinformatics, data comparison, and version control systems.

  iii. **Facilitates Sequence Comparison:**
     - It is particularly effective for comparing sequences, offering insights into shared elements and their arrangement.

- **Use Cases and Applications:**

    i. **Bioinformatics:**
        - In bioinformatics, the LCS algorithm is instrumental for comparing genetic sequences, identifying common elements, and understanding evolutionary relationships.

    ii. **Data Comparison:**
        - The algorithm is valuable in scenarios where comparing data sequences is crucial, such as identifying common patterns in time-series data.

    iii. **Version Control Systems:**
        - Version control systems leverage the LCS algorithm to track changes between different versions of files, aiding in conflict resolution and merging.

- **Limitations:**

    a. **Quadratic Time Complexity in Worst Case:**
        - While the dynamic programming approach provides an efficient solution, the worst-case time complexity of O(N*M) could be a limitation for extremely long sequences.

    b. **Memory Consumption:**
        - The algorithm's space complexity is proportional to the product of the lengths of the input sequences, potentially leading to high memory consumption for large sequences.

9. CONCLUSION:

In summary, the Longest Common Subsequence (LCS) algorithm efficiently identifies shared subsequences in sequences of varying lengths. Its dynamic programming methodology, while contributing to effective sequence analysis, presents a quadratic time complexity of O(N*M). Despite this limitation, the algorithm finds versatile applications in fields like bioinformatics and version control systems. Its balance between adaptability and efficiency makes it a valuable tool for uncovering common patterns, with potential for further optimization in future developments.