Interactive Longest Common Subsequenc (LCS)Algorithm Visualizer in C++: Report

Project Report: LCS (Longest Common Subsequence) Visualizer

Course: Algorithm Design and Analysis

Submitted to :

Name: Nasif Istiak Remon
(Senior Lecturer Metropolitan University)

Submitted By:

Name : Anij Fatema Lisa(232-115-034)

Name : Fatema Begum(232-115-003)

Date: 13-04-2025

1.Introduction

The Longest Common Subsequence (LCS) problem is a classic dynamic programming problem that aims to find the longest subsequence present in both input strings in the same order, but not necessarily contiguously. This project implements an LCS visualizer in C++ to demonstrate the dynamic programming table construction step-by-step.

2. Objective

To compute the LCS of two input strings. To visualize the table-filling process of the dynamic programming approach. To display the intermediate states and final LCS for better understanding.

4. Methodology

The algorithm uses bottom-up dynamic programming to fill a 2D table dp[i][j], where each cell represents the LCS length of the first i characters of string X and the first j characters of string Y. Algorithm Steps:

1. Initialize a 2D table with dimensions (m+1) x (n+1) to 0.
2. Iterate through both strings:

If characters match: dp[i][j] = 1 + dp[i-1][j-1] Else: dp[i][j] = max(dp[i-1][j], dp[i][j-1])

3. Reconstruct the LCS by tracing back from dp[m][n].
4. Visualize the table after every update using formatted output and brief delays.

5. Code Explanation

The code is divided into three main parts: printTable(): Responsible for visualizing the DP table with a delay using std::this_thread::sleep_for. LCS(): Core function that computes the LCS and prints intermediate tables. main(): Takes user input and calls the LCS function.

6. Problem Description

The Longest Common Subsequence (LCS) problem is a fundamental problem in computer science, particularly in string processing and dynamic programming. Given two strings, the goal is to find the longest sequence of characters that appear in both strings in the same order, but not necessarily contiguously.

Example:
String A = "ABCDGH"
String B = "AEDFHR"

LCS = "ADH"

The LCS helps in applications like:

- Comparing DNA sequences.
- File and version comparison.
- Diff tools in programming.

The visualizer in this project shows how the LCS is computed using a 2D dynamic programming table, updating and highlighting the process step by step.

7. Solution Approach

Dynamic Programming (Bottom-Up) To solve the LCS problem efficiently, we use Dynamic Programming (DP). The idea is to build a 2D table dp where:

- dp[i][j] represents the length of LCS of the first i characters of string X and the first j characters of string Y.

Steps:

- <u>Initialization:</u> Set all dp[0][*] and dp[*][0] to 0 because LCS of an empty string with any string is 0.
- <u>Filling the Table:</u> If X[i-1] == Y[j-1], then dp[i][j] = 1 + dp[i-1][j-1] Else, dp[i][j] = max(dp[i-1][j], dp[i][j-1])
- <u>Backtracking (Optional):</u> Trace back through the table from dp[m][n] to reconstruct the actual LCS string. The table is updated step-by-step and printed after every change to help visualize the algorithm’s progress. ---

8. Code Structure and Design

The code is modular and divided into three key parts:

- **printTable() Function:** Takes the DP table and strings as input. Displays the current state of the table. Highlights the current cell being filled. Adds a delay using this_thread::sleep_for to simulate visualization.
- **LCS() Function**: Builds the DP table using nested loops. Uses the printTable() function after each cell update. Reconstructs and prints the final LCS string and its length.
- **main() Function**: Takes user input for the two strings. Calls the LCS() function to start the visualization and computation.

This modular structure ensures clarity, reusability, and easy understanding of each component of the program.

9. Time Complexity

Let:

- m = length of string X
- n = length of string Y

Time Complexity:

Filling the DP Table: We iterate through a table of size (m+1) x (n+1) => O(m * n)

Backtracking to build the LCS string: At most m + n steps => O(m + n) (but this is minor compared to table filling) .

Total Time Complexity: O(m * n)

Space Complexity: The DP table uses O(m * n) space.

10.Additional Considerations

• User Experience: Visualization includes delays and table highlights to aid learning. Speed control or pause options could further enhance interactivity. • Educational Value: Narrating decisions (e.g., match found, taking max) would turn the visualization into a guided learning experience. • Scalability: For large inputs, performance may drop. Consider adaptive display (e.g., skipping steps or summarizing progress). • Real-World Relevance: Can be adapted for fields like bioinformatics or text comparison by using real data examples. • Future Scope: Easily extendable to other dynamic programming problems or web-based interactive tools.

11.Future Enhancements

GUI-based visualization using graphics libraries. Real-time user control over visualization speed. Visualization of backtracking steps for LCS reconstruction.

12.Conclusion

This LCS visualizer effectively demonstrates how dynamic programming solves the LCS problem. It is a helpful educational tool for understanding the DP matrix construction and the decision-making process involved in the algorithm.