Design and Analysis of Algorithms

Project Report

Spring 2025

Professor Reham Aburas

Longest Common Subsequence Algorithms

April 23, 2025

Omar Elshall — B00096779

Aafan Kashif — B00096430

Yara Zendaki — G00095603

1. Introduction

The longest common subsequence (LCS) problem is a fundamental computational task in string processing and comparative analysis. Given two strings, the objective is to find the longest subsequence that appears in both strings. Unlike the related longest common substring problem, a subsequence need not consist of consecutive characters and may have interruptions while maintaining the original order.

This problem has significant applications in various fields:

- **Bioinformatics**: Finding common DNA or protein sequences
- **Plagiarism detection**: Identifying copied text segments
- **Data compression**: Recognizing repeated patterns
- **Information retrieval**: Text similarity measurement

Our project implements and empirically compares two distinct algorithmic approaches to solve the LCS problem:

1. A brute-force method that examines all possible subsequences
2. A dynamic programming solution that builds a table of optimal subproblems

Through rigorous performance analysis with strings of varying lengths, we demonstrate the practical efficiency advantages of the dynamic programming approach over the brute-force method. We also provide visualizations to illustrate the algorithmic behavior and step-by-step execution of the dynamic programming solution.

2. Algorithmic Approaches

2.1 Brute Force Algorithm

The brute force approach systematically generates all possible subsequences from the first string and checks if each one appears in the second string, maintaining the longest match found.

Pseudocode:

```
Function BRUTE_FORCE_LCS(string str1, string str2):
    If str1 is empty OR str2 is empty:
        Return "", 0

    longest ← ""
    max_length ← 0

    # Generate all possible subsequences
    For each possible subsequence 'seq' of str1 (using
combinations):
        # Check if it's a subsequence of str2
        If IS_SUBSEQUENCE(seq, str2) AND LEN(seq) >
max_length:
            max_length ← LEN(seq)
            longest ← seq

    Return longest, max_length
```

Complexity Analysis:

Time Complexity: O(2^n * m)

Proof:

1. The algorithm generates all possible subsequences of the first string (str1):
    - There are 2^n possible subsequences for a string of length n
    - Each subsequence generation requires O(n) operations

- Thus, generating all subsequences takes O(n * 2^n) operations where n is the length of str1

3. For each generated subsequence, we check if it exists in the second string (str2):
   - Checking if a string is a subsequence of another takes O(m) time where m is the length of str2
   - This check is performed for each subsequence
4. Therefore, the total time complexity is:
   - O(2^n) subsequences × O(m) per check = O(2^n * m)

Space Complexity: O(n)

Proof:

1. We store the current longest subsequence, which in the worst case has length n
2. No additional data structures with size dependent on input are used
3. Therefore, the space complexity is O(n)

2.2 Dynamic Programming Algorithm

The dynamic programming approach constructs a 2D table where each cell [i,j] represents the length of the longest common subsequence considering the first i characters of str1 and the first j characters of str2.

Pseudocode:

```
Function DP_LCS(string str1, string str2):
    If str1 is empty OR str2 is empty:
        Return "", 0
    m ← LEN(str1)
    n ← LEN(str2)

    // Create DP table of size (m+1) × (n+1) initialized with
zeros
    dp[0...m, 0...n] ← 0

    // Fill the DP table
    For i ← 1 to m:
        For j ← 1 to n:
            If str1[i-1] equals str2[j-1]:
                dp[i,j] ← dp[i-1, j-1] + 1
            Else:
                dp[i,j] ← MAX(dp[i-1, j], dp[i, j-1])

    // Backtrack to find the actual subsequence
    lcs ← ""
    i ← m, j ← n

    While i > 0 AND j > 0:
        If str1[i-1] equals str2[j-1]:
            lcs ← str1[i-1] + lcs
            i ← i - 1
            j ← j - 1
        Else If dp[i-1, j] > dp[i, j-1]:
            i ← i - 1
        Else:
            j ← j - 1

    Return lcs, LEN(lcs)
```

Complexity Analysis:

Time Complexity: O(nm)

Proof:

1. The algorithm fills a 2D dynamic programming table of size (m+1) × (n+1):
    - The outer loop runs m times (for each character in str1)
    - The inner loop runs n times (for each character in str2)
    - Each cell computation takes constant O(1) time
2. Therefore, the total time complexity is:
    - O(m × n × 1) = O(nm)

Space Complexity: O(nm)

Proof:

1. The algorithm requires a 2D table of size (m+1) × (n+1)
2. Each cell in the table stores a single integer
3. Therefore, the space complexity is O(nm)

3. Performance Analysis and Results

3.1 Experimental Setup

To empirically evaluate both algorithms, we conducted extensive performance testing with the following parameters:

- String lengths varying from 10 to 1000 characters for DP and 10 to 30 for brute force
- Step size of 1 for all tests
- Multiple random test cases for each size combination
- Execution times averaged over multiple runs to reduce variability
- Measurements performed on the same hardware for fair comparison

The brute-force algorithm tests were limited to strings with maximum length of 30 to avoid excessive execution times, as running these tests takes approximately 2 hours to complete.

3.2 Runtime Comparison

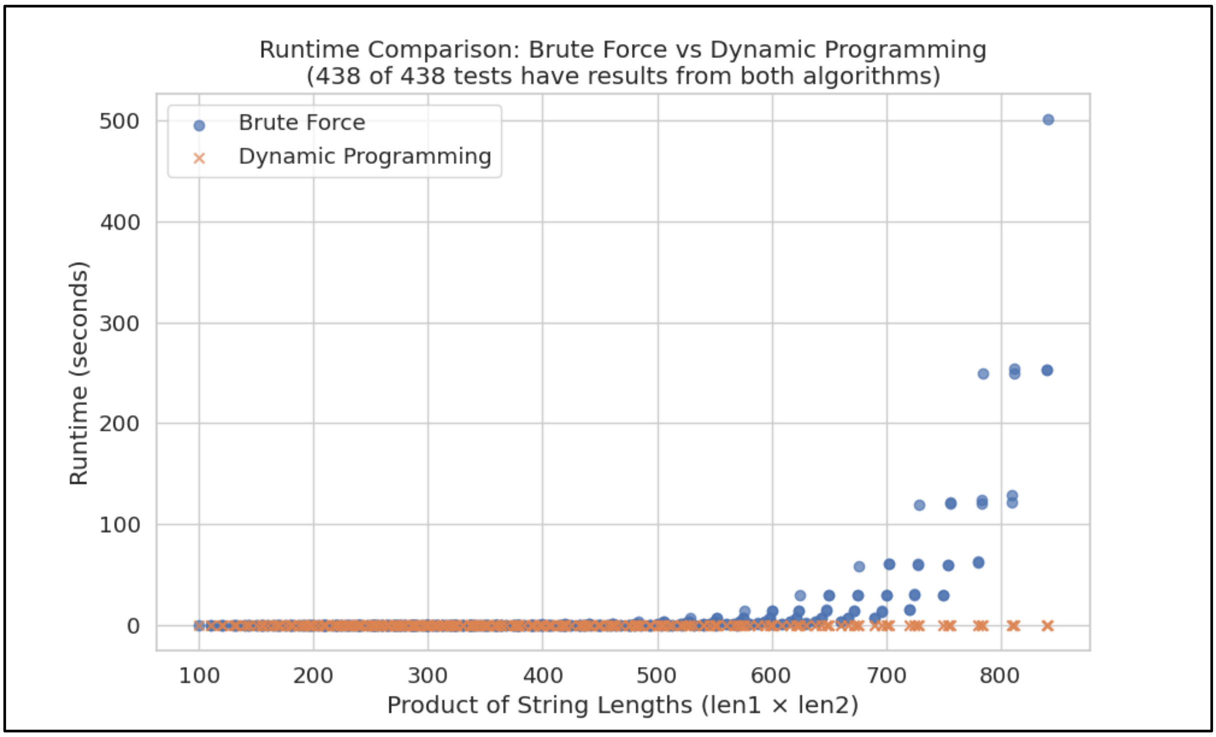

Figure 1: *Runtime comparison between brute-force and dynamic programming approaches*

The runtime comparison demonstrates how execution time increases with input size (measured as the product of string lengths) for both algorithms. Key observations:

- The brute-force algorithm (blue) shows a steeper increase in runtime as input size grows
- The dynamic programming algorithm (orange) maintains considerably lower execution times
- The trend lines illustrate the significant performance gap that widens with larger inputs
- The empirical results align with theoretical complexity predictions (O(2^n) vs. O(nm))

3.3 Speedup Analysis

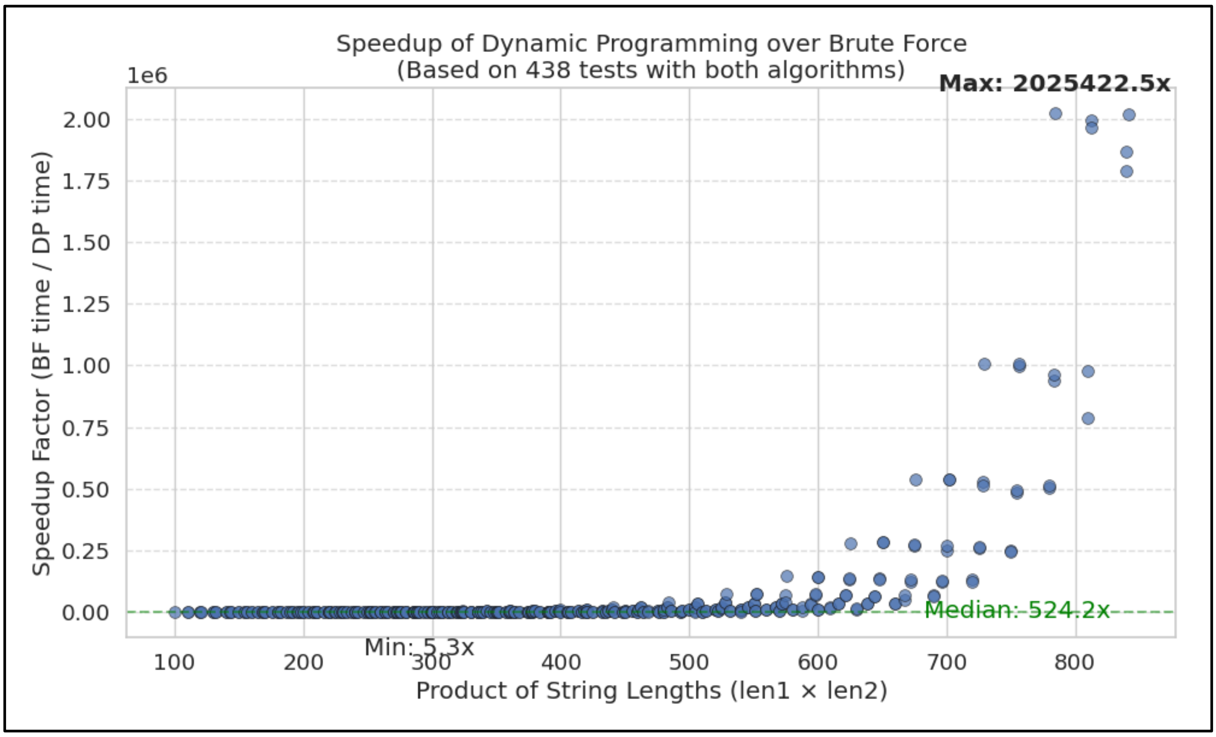

Figure 2: *Speedup factor of dynamic programming over brute force approach*

The speedup visualization is now presented as a scatter plot that quantifies the performance advantage of the dynamic programming approach over the brute-force method. Notable findings:

- Speedup factors range from 5× to over 2000000× for the tested input sizes
- The performance gap increases as the input size grows
- The maximum and minimum speedup points are clearly labeled, along with a median line
- The trend confirms that the dynamic programming approach provides substantially better performance for realistic use cases
- For very large inputs, the brute-force approach becomes prohibitively expensive

3.4 Theoretical vs. Experimental Complexity

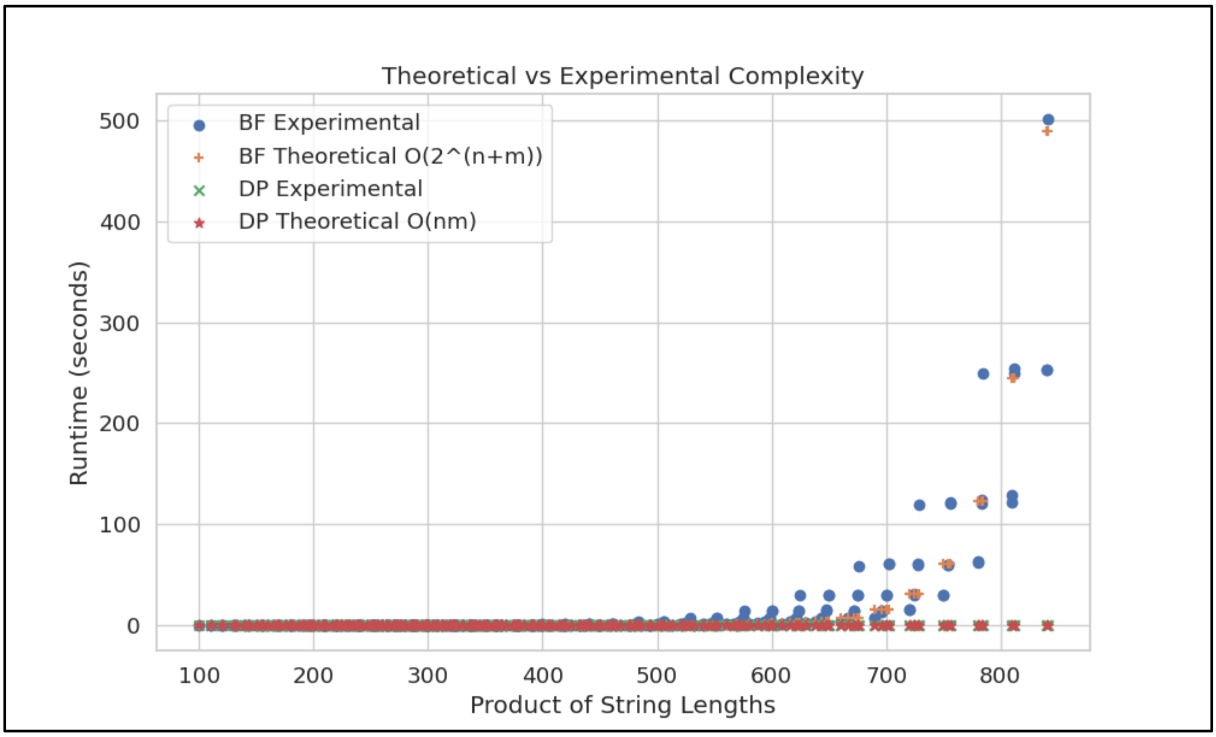

Figure 3: *Comparison of theoretical complexity models with experimental measurements*

This visualization compares theoretical complexity models with actual measured performance, using only test cases where both algorithms successfully completed:

- Brute-force experimental times (circles) closely follow the theoretical O(2^n) curve (plus signs)
- Dynamic programming experimental times (x marks) align with the theoretical O(nm) model (asterisks)
- The close correspondence validates our complexity analysis
- The gap between the two algorithms' performance is consistent with theoretical predictions and clearly demonstrates why the dynamic programming approach is superior

3.5 Dynamic Programming Table Visualization

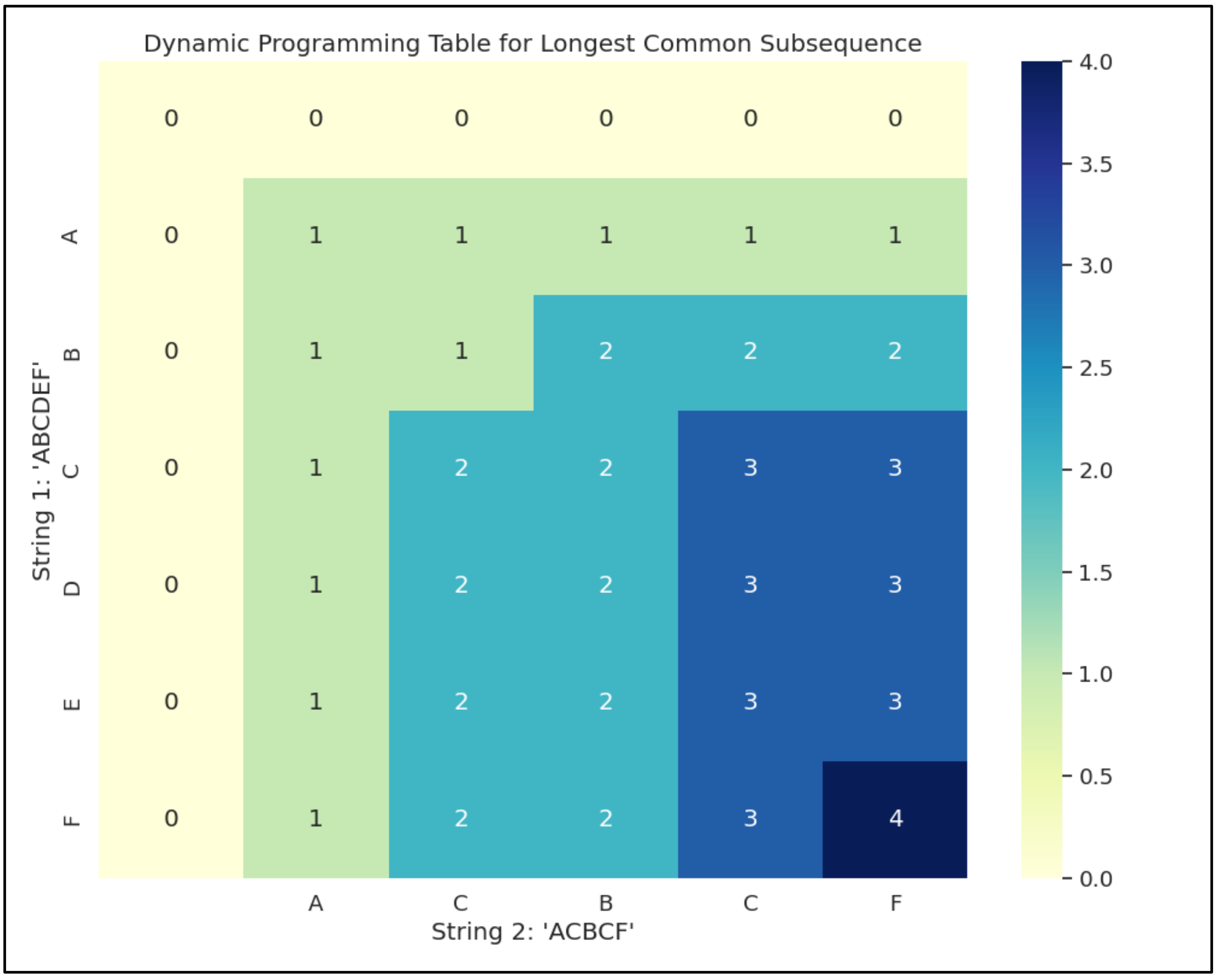

Figure 4: *Visual representation of the dynamic programming table*

We created a heatmap visualization of the DP table for a more intuitive understanding of the dynamic programming approach. This visualization:

- Shows how the table is populated with subsequence lengths
- Illustrates patterns that emerge during the computation
- Helps identify how the longest common subsequence is constructed
- Provides insight into the algorithm's internal operations

4. Interactive Visualization Component

As a bonus component, we developed an interactive web-based visualization and standalone executable demonstrating how the dynamic programming table is built for the longest common subsequence problem.

4.1 Visualization Features

The interactive visualization:

- Displays the DP table and updates it in real-time as the algorithm progresses
- Highlights the current cell being processed (in red)
- Highlights the cells that form the longest common subsequence path (in yellow)
- Provides detailed descriptions of each algorithm step
- Allows navigation through the algorithm execution (forward and backward)
- Enables users to input custom strings to visualize different scenarios
- Shows the final result with the identified longest common subsequence

4.2 Implementation Details

We implemented this visualization using:

- A Python-based HTTP server to create a local web interface
- HTML and CSS for the user interface and styling
- JavaScript for interactive controls
- A step-by-step algorithm state generator to track table updates
- A standalone executable version using PyInstaller for easy distribution across platforms (Windows, macOS, Linux)

This interactive tool serves as both an educational resource for understanding the algorithm and a demonstration of the dynamic programming approach's efficiency.

5. Conclusions

Our comprehensive implementation and analysis of the Longest Common Subsequence problem yielded several significant findings:

1. **Performance Advantage**: The dynamic programming approach dramatically outperforms the brute-force method, with speedup factors increasing with input size.
2. **Scalability**: While the brute-force algorithm becomes impractical for even moderately sized inputs, the dynamic programming approach remains efficient for substantially larger strings.
3. **Theoretical Validation**: The empirical results closely match the theoretical complexity analysis, confirming the O(2^n) complexity of the brute-force approach and the O(nm) complexity of the dynamic programming solution.
4. **Space-Time Tradeoff**: The dynamic programming approach trades increased memory usage (O(nm) vs. O(n)) for significantly improved time efficiency.
5. **Visualization Insights**: The interactive visualization demonstrates how dynamic programming breaks down the problem into overlapping subproblems and reuses previous computations.

The dynamic programming approach is unequivocally the superior choice for real-world applications of the Longest Common Subsequence problem due to its practical efficiency and scalability. The performance advantage becomes increasingly significant as input sizes grow, making it the only viable option for applications involving long strings.

6. Team Members and Contributions

Our team collaborated equally on all aspects of the project, with shared responsibility for report writing, documentation, and code implementation. We worked together on all major components with the following contributions:

Omar Elshall

- Algorithm implementation (LCS algorithms)
- Report writing and documentation
- Interactive visualization component
- Performance testing

Aafan Kashif

- Report writing and documentation
- String generation and testing utilities
- Data visualization
- Algorithm implementation

Yara Zendaki

- Report writing and documentation
- Executable packaging and build system
- Algorithm implementation
- Performance analysis