Longest Common Subsequence*

1nd Haidy Ahmed
Faculty of Computer Science
Misr International University
haidy2305019@miueygpt.edu.eg

2nd Youssef Ahmed Rashad
Faculty of Computer Science
Misr International University
youssef2302641@miueygpt.edu.eg

3rd Hana Ahmed
Faculty of Computer Science
Misr International University
hana2305980@miueygpt.edu.eg

4th Passant Saher
Faculty of Computer Science
Misr International University
passant2305487@miueygpt.edu.eg

6thWaleed Amr Attia
Faculty of Computer Science
Misr International University
waleed2302412@miueygpt.edu.eg

***Abstract*—Abstract—This research looks at and compares three different ways to solve the Longest Common Subsequence (LCS) problem: Brute Force, Divide and Conquer, and Dynamic Programming. The LCS problem is a common problem in computer science and is useful in areas like comparing files, compressing data, and studying DNA. The Divide and Conquer method uses a step-by- step, recursive approach to break the problem into smaller parts. This method works better than the simple Brute Force method in many cases. The Brute Force method in a complete generates and compares all subsequences, highlighting its ideal simplicity but suffering from exponential time complexity. The Dynamic Programming method uses a smart way of solving problems by breaking them into smaller parts and reuse answers to those parts. This helps find the LCS quickly and efficiently. In this paper, we show how each method works using C++ code. We also look at how fast they are and how much memory they use. Our results show that each method has its own strengths and weaknesses. This helps people choose the right method depending on the situation. Index Terms—LCS, Divide and Conquer, Brute Force, Dynamic Programming, Algorithm-Comparation, String Matching**

***Index Terms*—LCS, Divide and Conquer, Brute-Force , Dynamic-Programing, algorithms-comparison , string-matching**

I. Introduction

The Longest Common Subsequence (LCS) problem revolves finding the longest subsequence common of two strings,not necessarily connected. It serves as a foundation for various string comparison tasks. Many approaches can be used to solve this problem, including brute force, divide and conquer and dynamic programming. This paper goal is to implement and evaluate the three methods and the complexity

II. Divide and Conquer Approach

The concept known as divide-and-conquer involves breaking down a complex problem into smaller or similar subproblems. These sub-problems are further divided until a straightforward solution appears for each one. Once the simplest instances are solved, the solutions are merged to create the final result. This approache forms the foundation for several algorithms, including merge sort and fast transform. In the the Longest Common Subsequence (LCS), the divide- and-conquer algorithm recursively compares characters of two identical If none, delete this. In the context of the Longest Common Subsequence (LCS), the divide-and-conquer algorithm make input strings. If the final characters of both strings match then that character is considered a part of the LCS, and the problem is reduced by removing those characters. Or the problem is split into two cases: One by removing the last character of the first string and One by removing as well the last character of the second string The recursive calls continue until one of the strings becomes empty. The LCS is constructed by combining the special results from the sub-cases. Recursion plays a critical role in this strategy. The way of the LCS problem makes recursion suitable, as each problem is a smaller version of the original one. However, the absence of it leads to repeated calculations and result in time complexity.

Identify applicable funding agency here. If none, delete this.

III. BRUTE FORCE APPROACH

The brute force method is the most straightforward way to solve the LCS problem as it's approach explores all possible subsequences of the given strings and determines the longest sequence common path to both of them. To achieve the algorithm it generates every possible solution of the first and second strings. Then compares each of the subsequences to identify that they are common to each others. The longest among these is selected to be the final answer. While this method guarantees correctness, it is totally expensive. The number of subsequences for a string of length n is$2n$by comparing two strings of lengths m and n leads to $0(2m.2n)$time complexity. Regardless the efficiency, the brute force approach is important for understanding the problem's structure and for testing the accurate of the algorithms. Its implementation is ideal simple and does not require advanced programming constructs, makes it simillar for educational purposes

IV. DYNAMIC PROGRAMMING APPROACH

Dynamic programming is a powerful technique that solves complex problems by breaking them down into simpler subproblems and storing the results of these sub-problems to avoid redundant computations.

For the LCS problem, a two-dimensional table (2D) is used. Each cell dp [i] [j] in the table represents the length of the LCS for the substrings X [0...i-1] and Y [0... j-1]. The table is filled using the following logic:

- If the characters at positions $i1$ and $j1$ match, the value is$1 + dp[i1][j1]$.
- If they do not match, the value is the maximum between dp[i 1][j] and dp[i][j 1].

Once the table is made, the LCS length is set at dp[m][n], and the actual sequence can be created by tracing back through the table. This approach has a time complexity of$O(m.n)$ and a space complexity of $O(m.n)$, making it the most efficient solution for medium to large-sized inputs. Divide and conquer, dynamic programming does not make repeated calculations and brute force, it does not explore all possibilities. Dynamic programming effectively balances performance and accuracy, making it the preferred method in real-world applications where efficiency is critical.

V. The Longest Common Subsequence (LCS)

The Longest Common Subsequence (LCS) problem is a well-known problem in computer science. It is used in many areas such as studying DNA, checking file changes, and comparing text. The goal is to find the longest group of letters that appear in the same order in both strings (they don't have to be next to each other). This paper introduces three simple ways to solve the LCS problem: Brute Force, Divide and Conquer, and Dynamic Programming. We will talk about how each one works, how fast they are, and how useful they can be. LCS OVERVIEW. The LCS of two strings is defined as the longest sequence that appears in both strings in the same order. It is a key metric in evaluation sequence similarity. Unlike substrings, subsequences do not need to occupy the same positions. The LCS problem holds up many tools used in file comparison, diff utilities, DNA sequence alignment, and spell checkers.

VI. The Longest Common Subsequence :divide and conque

The divide and conquer technique is an algorithmic paradigm that recursively breaks a problem into smaller subproblems, solves each independently, and combines their results. For the LCS problem, this method is applied as follows: 1)If either input string is empty, return an empty LCS.

2)If the last characters of the strings match, they are part of the LCS and the problem reduces to the remaining substrings.

3)If they do not match, recursively compute the LCS by:

- Removing the last character from the first string Removing the last character from the second string Returning the longer of the two results

This recursive process continues until base conditions are met.

The divide and conquer approach highlights the recursive nature of LCS computation. However, due to overlapping subproblems and lack of memoization, the approach exhibits exponential time complexity. -Advantages:

.Reflects the theoretical recursive formulation of LCS .Easy to understand and implement for small inputs

-Disadvantages:

Time complexity of$O(2^n)$, where n is the string length Inefficient for large inputs due to redundant computations Increased call stack usage and potential memory overflow

VII. The Longest Common Subsequence : Dynamic Programming

Dynamic Programming (DP) solves the LCS problem efficiently by storing intermediate results in a table to avoid redundant computations: 1)Create a 2D matrix dp[i][j] representing LCS length for the first i characters of X and first j of Y 2)Use the recurrence:

- If $X[i-1] == Y[j-1]$, $dp[i][j] = dp[i-1][j-1] + 1$ Else, $dp[i][j] = max(dp[i-1][j], dp[i][j-1])$

3)Trace back from dp[m][n] to reconstruct the LCS

-Advantages:

- Time complexity: $O(m.n)$
- Efficient and scalable
- Suitable for real-world use cases

-Disadvantages:

- Space complexity:$O(m.n)$, though optimizations exist
- Requires backtracking to reconstruct the LCS string

VIII. The Longest Common Subsequence : Brute force

The brute force method systematically generates all possible subsequences of both strings and compares each pair to find the longest common subsequence. The process can be described as: 1)Generate all2^nsubsequences of the first string. 2)Generate all 2^msubsequences of the second string. 3)Compare each pair for equality. 4)Return the longest sequence found in both.

This method ensures correctness by exhaustively exploring the solution space but is highly inefficient in practice. -Advantages:

- Straightforward and guarantees correct result.
- Useful for validating other algorithm implementations.

-Disadvantages:

- Exponential time complexity of$O(2^n * 2^m)$.
- Memory usage grows rapidly due to storing subsequences.
- Impractical for sequences longer than a few characters.

IX. OBJECTIVE

- To explain the concept and relevance of the LCS problem.
- To implement and evaluate the Divide and Conquer, Brute Force, and Dynamic Programming approaches.
- To compare their computational complexity and memory usage.
- To determine the most suitable approach for practical applications.

X. METHODOLOGY

A. Algorithm Implementation All three algorithms were implemented in C++ using consistent input and output formats for fair comparison.

B. Evaluation Criteria We evaluated correctness, execution time, and memory usage across multiple input pairs of varying lengths.

C. Complexity Analysis

-Divide and Conquer:$O(2^n)$time, recursive stack space.

-Brute force: $O(2^n 2^m)$ time, high memory for subsequences.

-Dynamic programming: $O(mn)$ time and space.

TABLE I
COMPARISON OF LCS ALGORITHM COMPLEXITIES

Method	**Time Complexity**	**Space Complexity**
Divide and Conquer	$O(2^n)$	Recursive stack
Brute Force	$O(2^n \cdot 2^m)$	High (subsequences)
Dynamic Programming	$O(mn)$	$O(mn)$

XI. PROCEDURE

To evaluate the performance of the implemented LCS algorithms, we designed a series of experiments involving different string inputs. Three LCS algorithms were tested: a divide-and-conquer approach, a brute-force method, and a dynamic programming solution.

We varied the input strings in terms of length and character composition to observe how each algorithm scales with increased input size and complexity. For each pair of input strings, all three algorithms were executed independently. The resulting longest common subsequences and their lengths were recorded. The procedure was as follows:

- For the Divide and Conquer approach, two input strings were entered, and the recursive algorithm computed the LCS and its length.
  In the Brute Force method, all possible subsequences of both strings were generated and compared to find the longest common subsequence.
  The Dynamic Programming technique used a tabulated matrix to compute the LCS efficiently by building up solutions to subproblems.

Each experiment was run using the same set of input strings across all three algorithms to ensure fair comparison. The outputs—including LCS strings and their corresponding lengths—were used to assess the correctness and efficiency of each method under varying conditions.

XII. CALCULATIONS AND ANALYSIS

The results of our experiments are summarized in Tables I and II. Each table contains the length and the string of the Longest Common Subsequence (LCS) obtained by the three implemented algorithms: Divide and Conquer, Brute Force, and Dynamic Programming.

The time complexity for each algorithm, under average and worst-case scenarios, is as follows: Divide and Conquer:

- Average Case: $O(2^n)$

Brute Force:

- Average Case: $O(2^n.2^n) = O(4^n)$
- Worst Case:$O(4^n)$

Dynamic Programming:

- Average Case: $O(m.n)$
- Worst Case: $O(m.n)$

Here, m and n represent the lengths of the two input strings. The experimental results indicate that all three algorithms produce the correct LCS outputs. However, the Dynamic Programming approach significantly outperforms the other two methods in both average and worst-case scenarios. The Divide and Conquer method, although conceptually elegant, exhibits exponential growth in recursive calls, and the Brute Force method quickly becomes infeasible for longer strings due to its combinatorial explosion in subsequence generation.

XIII. DISCUSSION

The experimental analysis clearly demonstrates the efficiency advantages of the Dynamic Programming (DP) algorithm for solving the LCS problem. The DP approach reduces the problem to a manageable size by storing intermediate results and avoiding redundant computations.

In contrast, the Divide and Conquer algorithm, while elegant and straightforward, suffers from repeated subproblem computations, leading to higher time complexity—especially for longer input strings. The Brute Force method, which explores all possible subsequences, is computationally intensive and impractical for large inputs, although it is useful as a conceptual or educational reference.

Interestingly, despite these differences in efficiency, all three algorithms produced correct LCS results in the tested cases. This confirms their theoretical validity but highlights the need for algorithmic efficiency in practical applications.

XIV. CONCLUSION

In this project, we implemented and compared three algorithms for solving the Longest Common Subsequence (LCS) problem: Divide and Conquer, Brute Force, and Dynamic Programming.

While all approaches returned correct LCS outputs, the Dynamic Programming method consistently offered superior performance and scalability. The Divide and Conquer approach is conceptually valuable but computationally expensive due to overlapping subproblems. The Brute Force method, although simplest in logic, is impractical for larger inputs due to its exponential complexity.

The choice of algorithm ultimately depends on the input size and performance requirements. For real-world applications where efficiency is crucial, Dynamic Programming is the preferred solution.

Future work could explore space-optimized DP techniques or parallelized recursive methods to further enhance performance.

XV. REFRENCES

REFERENCES

[1]
[2]