The Longest Common Palindromic Subsequence Problem: Algorithms and Analysis

Ali Ansari

Abstract

In this paper, we investigate the *Longest Common Palindromic Subsequence* (LCPS) problem, which seeks to identify the longest sequence that is both a subsequence of two given strings and a palindrome. We begin by introducing a brute-force algorithm to serve as a baseline for comparison. Subsequently, we propose a one-step dynamic programming solution with a time complexity of $\mathcal{O}(n^4)$. Finally, we present an improved two-step algorithm, derived from the longest common subsequence (LCS) problem, that achieves a more efficient time complexity of $\mathcal{O}(n^2)$. For each proposed method, we provide detailed proofs of correctness, analyses of time and space complexity, and a discussion of potential real-world applications.

1 Introduction

A palindrome is a sequence that reads the same backward as forward. The Longest Common Palindromic Sequence (LCPS) problem involves finding the longest subsequence that is both common to two input strings and a palindrome. A similar well-known problem to this is The Longest Common Subsequence (LCS) which can be solved using Dynamic Programming algorithm. This problem has applications in bioinformatics, text processing, and data compression.

2 Problem Definition

Given two strings X and Y, find the longest subsequence P such that P is a palindrome and P is a subsequence of both X and Y.

3 Brute Force Approach

The brute-force approach finds all possible subsequences of X, checks if they are subsequences of Y, and verifies if they are palindromes. The longest such sequence is the LCPS.

3.1 Pseudocode

Algorithm 1 Brute-Force LCPS

```
1: procedure BruteForceLCPS(X, Y)
2:     n ← |X|, m ← |Y|
3:     maxLen ← 0
4:     result ← ""
5:     for i = 0 to 2^n − 1 do                    ▷ Generate all subsequences of X
6:         subseq ← subsequence of X corresponding to binary i
7:         if subseq is a subsequence of Y and subseq is a palindrome then
8:             if |subseq| > maxLen then
9:                 maxLen ← |subseq|
10:                result ← subseq
11:            end if
12:        end if
13:    end for
14:    return result
15: end procedure
```

3.2 Proof of Correctness

Proof by Contradiction. Suppose the brute-force algorithm returns a palindromic subsequence P of X and Y, but P *is not* the true longest common palindromic subsequence.

By our assumption, there must exist some palindromic subsequence U of X and Y with

$$|U| > |P|.$$

However, the brute-force algorithm enumerates *all* subsequences of X, including U. Moreover, during its execution, the algorithm checks whether each candidate subsequence is also a subsequence of Y and whether it is a palindrome. Since U satisfies both properties, the algorithm must have identified U as a valid candidate and should have chosen it if $|U| > |P|$.

This contradicts the fact that the algorithm returned P and not U. Therefore, our initial assumption that P is not the longest such subsequence must be false. Hence, the subsequence P returned by the algorithm is indeed the longest common palindromic subsequence of X and Y. □

3.3 Time Complexity

For a string S of length n, there are 2^n subsequences. Checking if each is a subsequence of T (length m) takes $O(m)$ time, and verifying palindromicity takes $O(n)$. Thus, the runtime is $O(2^n \cdot (m+n))$. Storage is $O(n)$ for the current subsequence. This approach is impractical for large inputs but serves as a baseline.

4 Dynamic Programming Approach

To improve upon the brute-force algorithm, we exploit the problem's optimal substructure and overlapping subproblems. We define a 4D dynamic programming table to store the lengths of LCPS for substrings of X and Y.[1]

4.1 Subproblem Definition

Let $lcps[i, j, k, \ell]$ be the length of the longest common palindromic subsequence between $X_{i,j}$ and $Y_{k,\ell}$.

4.2 Base Cases

- If either of the substrings is empty, i.e., $i > j$ or $k > \ell$, then there can be no common subsequence. Hence, we define:

$$\texttt{lcps}[i, j, k, \ell] = 0 \quad \text{if } i > j \text{ or } k > \ell$$

- If either substring has length one (i.e., $i = j$ or $k = \ell$), then the length of the LCPS is either 0 or 1 depending on whether the single character occurs in both substrings. In this case:

$$\texttt{lcps}[i, j, k, \ell] = \begin{cases} 1 & \text{if } x_i = y_k \text{ or } x_i = y_\ell \text{ or } x_j = y_k \text{ or } x_j = y_\ell \\ 0 & \text{otherwise} \end{cases} \quad \text{when } i = j \text{ or } k = \ell$$

4.3 Recursive Computation

The dynamic programming table `lcps` is a 4-dimensional array of size $n \times n \times n \times n$, where each entry `lcps`[i, j, k, ℓ] is computed by considering the lengths of all smaller substrings. We compute values in a bottom-up manner, ensuring that results for shorter substrings (length v) are available before computing those of longer substrings (length $v + 1$).

The final result, i.e., the length of the LCPS between the full strings X and Y, is stored at:

$$\texttt{lcps}[1, n, 1, n]$$

Algorithm 2 outlines the dynamic programming procedure LCPSLength, which takes as input two strings X and Y of length n and returns both the filled table `lcps` and the value at `lcps`[1, n, 1, n].

4.4 Recurrence Relation

$$lcps[i, j, k, \ell] = \begin{cases} 0 & \text{if } i > j \text{ or } k > \ell \\ 1 & \text{if } (i = j \text{ or } k = \ell) \text{ and } x_i = y_k \\ 2 + lcps[i + 1, j - 1, k + 1, \ell - 1] & \text{if } x_i = x_j = y_k = y_\ell \\ \max \left\{ \begin{array}{l} lcps[i + 1, j, k, \ell], \\ lcps[i, j - 1, k, \ell], \\ lcps[i, j, k + 1, \ell], \\ lcps[i, j, k, \ell - 1] \end{array} \right\} & \text{otherwise} \end{cases}$$

4.5 Pseudocode

Algorithm 2 LCPSLength(X, Y)

```
1: n ← length of X
2: Initialize 4D table lcps[n + 1][n + 1][n + 1][n + 1] ← 0
3: for i = n downto 1 do
4:     for j = i to n do
5:         for k = n downto 1 do
6:             for ℓ = k to n do
7:                 if x_i = x_j = y_k = y_ℓ then
8:                     lcps[i][j][k][ℓ] ← 2 + lcps[i + 1][j − 1][k + 1][ℓ − 1]
9:                 else
10:                    lcps[i][j][k][ℓ] ← max(lcps[i + 1][j][k][ℓ], lcps[i][j − 1][k][ℓ], lcps[i][j][k + 1][ℓ], lcps[i][j][k][ℓ − 1])
11:                end if
12:            end for
13:        end for
14:    end for
15: end forreturn lcps[1][n][1][n]
```

4.6 Proof of Correctness

Let X and Y be two sequences of length n, and let $X_{i,j} = x_i x_{i+1} \dots x_j$ and $Y_{k,\ell} = y_k y_{k+1} \dots y_\ell$ denote substrings of X and Y, respectively. Suppose $Z = z_1 z_2 \dots z_u$ is the Longest Common Palindromic Subsequence (LCPS) of $X_{i,j}$ and $Y_{k,\ell}$. The correctness of the dynamic programming approach relies on the following key observations:

1. **Case 1: Matching Boundaries.** If the boundary characters match across both substrings, i.e., $x_i = x_j = y_k = y_\ell = a$ for some $a \in \Sigma$, then the first and last characters of Z must also be a, i.e., $z_1 = z_u = a$. The subsequence $z_2 z_3 \dots z_{u-1}$ must then be an LCPS of the inner substrings $X_{i+1,j-1}$ and $Y_{k+1,\ell-1}$.

   Proof: Since Z is a palindrome, we know $z_1 = z_u$. Suppose, for contradiction, that $z_1 = z_u \neq a$. In that case, we could construct a longer common palindromic subsequence by surrounding Z with a on both ends, yielding a string of length $u+2$. This contradicts the assumption that Z is the longest such subsequence. Hence, $z_1 = z_u = a$, and the middle portion $z_2 \dots z_{u-1}$ must be a palindrome common to both $X_{i+1,j-1}$ and $Y_{k+1,\ell-1}$.

   Now, assume that this inner subsequence is not the LCPS of $X_{i+1,j-1}$ and $Y_{k+1,\ell-1}$—that is, suppose there exists a longer common palindromic subsequence W of these substrings. Then adding a to both ends of W would yield a palindromic subsequence of $X_{i,j}$ and $Y_{k,\ell}$ longer than Z, which is a contradiction. Therefore, $z_2 \dots z_{u-1}$ must indeed be the LCPS of the inner substrings.

2. **Case 2: Boundary Mismatch.** If the characters at the boundaries do not all match (i.e., at least one of x_i, x_j, y_k, or y_ℓ differs from the others), then the LCPS Z must be contained

entirely within one of the following subproblems:

$$X_{i+1,j} \text{ and } Y_{k,\ell}, \quad X_{i,j-1} \text{ and } Y_{k,\ell}, \quad X_{i,j} \text{ and } Y_{k+1,\ell}, \quad X_{i,j} \text{ and } Y_{k,\ell-1}$$

Proof: Again, since Z is a palindrome, we know $z_1 = z_u$. However, because the boundary condition does not hold, at least one of the boundary characters is not equal to z_1 and thus cannot be part of the LCPS. This means Z must be fully contained in at least one of the smaller subproblems formed by removing a character from the ends of $X_{i,j}$ or $Y_{k,\ell}$.

Suppose, for contradiction, that none of these smaller subproblems contain a longer LCPS than Z. Then Z remains the best solution. However, if any one of these subproblems had a longer common palindromic subsequence W than Z, then W would also be a valid common palindromic subsequence of $X_{i,j}$ and $Y_{k,\ell}$—which contradicts the assumption that Z is the longest. Hence, one of these recursive subproblems must contain Z.

This recursive structure forms the basis of the dynamic programming recurrence and guarantees that each optimal solution can be built from optimal solutions to smaller subproblems.

4.7 Time and Space Complexity

- **Time Complexity:** We fill a 4D table of size $O(n^4)$ and each entry takes $O(1)$ time to compute. Hence, total time is $O(n^4)$.
- **Space Complexity:** We store $O(n^4)$ values, resulting in $O(n^4)$ space usage. Space can be optimized with memoization or smart layering in practice.

5 A two-step but more efficient approach

In this section, we present a two-step algorithm for computing the *Longest Common Palindromic Subsequence (LCPS)* of two input strings X and Y[2]. Our approach builds upon the classical dynamic programming method for computing the *Longest Common Subsequence (LCS)*. The algorithm proceeds as follows:

1. Compute the Longest Common Subsequence (LCS) of X and Y, denoted as Z.
2. Compute the Longest Palindromic Subsequence (LPS) of Z by finding the LCS of Z and its reverse Z'.

The output of this procedure is the Longest Common Palindromic Subsequence of the original strings X and Y.

5.1 Step 1: Longest Common Subsequence (LCS)

Let X be of length n and Y of length m. Define a DP table $dp_1[i][j]$ representing the length of the LCS of $X[1..i]$ and $Y[1..j]$.

Recurrence Relation

$$dp_1[i][j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ dp_1[i-1][j-1] + 1 & \text{if } X[i] = Y[j] \\ \max(dp_1[i-1][j], dp_1[i][j-1]) & \text{otherwise} \end{cases}$$

5.2 Step 2: Longest Palindromic Subsequence (LPS) of Z

Let $Z[1..n]$ be the input string, and $Z'[1..n]$ be the reverse of Z. If P is a palindromic subsequence of Z, then P is also a subsequence of Z'. Therefore, the length of the longest palindromic subsequence (LPS) in Z is equal to the length of the longest common subsequence (LCS) between Z and Z'.

5.3 Pseudocode

Algorithm 3 LCPS(X, Y)

Let dp_1 be an array of size $(n+1) \times (m+1)$
for $i = 1$ to n **do**
    for $j = 1$ to m **do**
        if $X[i-1] = Y[j-1]$ **then**
            $dp_1[i][j] \leftarrow dp_1[i-1][j-1] + 1$
        else
            $dp_1[i][j] \leftarrow \max(dp_1[i-1][j], dp_1[i][j-1])$
        end if
    end for
end for
Construct string Z from dp_1 table by backtracking
Let $Z_{\text{rev}} \leftarrow$ reverse of Z
Let dp_2 be an array of size $(k+1) \times (k+1)$ where $k = |C|$
for $i = 1$ to n **do**
    for $j = 1$ to n **do**
        if $Z[i-1] = Z_{\text{rev}}[j-1]$ **then**
            $dp_2[i][j] \leftarrow dp_2[i-1][j-1] + 1$
        else
            $dp_2[i][j] \leftarrow \max(dp_2[i-1][j], dp_2[i][j-1])$
        end if
    end for
end for
return $dp_2[n][n]$

5.4 Proof of Correctness

We prove the correctness of the algorithm by reasoning about each step independently.

Step 1: Correctness of LCS(X, Y)

We prove that dp$[i][j]$ correctly computes the length of the LCS of $X[1..i]$ and $Y[1..j]$.

Base Case

If $i = 0$ or $j = 0$, one of the strings is empty. Then $\text{dp}[i][j] = 0$, which is correct.

Inductive Step

Assume that $\text{dp}[i'][j']$ is correct for all $i' < i$ and $j' < j$.

- **Case 1:** If $X[i] = Y[j]$, then the character contributes to the LCS. Thus, $\text{dp}[i][j] = \text{dp}[i-1][j-1] + 1$ is correct.
- **Case 2:** If $X[i] \neq Y[j]$, we exclude one character and take the maximum of the two options: $\text{dp}[i-1][j]$ or $\text{dp}[i][j-1]$. Therefore, $\text{dp}[i][j] = \max(\text{dp}[i-1][j], \text{dp}[i][j-1])$ is correct.

By induction, the recurrence builds correct values for $\text{dp}[i][j]$ from smaller subproblems, ensuring that $\text{dp}[n][n]$ is the correct length of the longest common subsequence of X and Y.

Step 2: Correctness of LPC(Z)

Let L be any LCS of Z and Z'. Because L is a subsequence of both Z and Z' (which is the reverse of Z), the indices in Z and Z' that form L are symmetric.

That is, if $L = Z[i_1], Z[i_2], \ldots, Z[i_k]$ and $L = Z'[j_1], Z'[j_2], \ldots, Z'[j_k]$, then due to the reversal, we must have:

$$Z[i_1] = Z[n - j_1 + 1], \quad Z[i_2] = Z[n - j_2 + 1], \quad \ldots$$

This implies that $L = L^R$, so L is palindromic.

Composition Correctness

Since the first step guarantees the longest sequence common to both X and Y, and the second step guarantees the longest palindromic subsequence of that result, the final output is by definition the longest subsequence that:

- appears in both X and Y, and
- is a palindrome.

Hence, the composition of these two correct algorithms yields the correct result for the Longest Common Palindromic Subsequence (LCPS) between X and Y.

5.5 Time Complexity Analysis

Let $n = |X|$ and $m = |Y|$. Let k be the length of the LCS of X and Y.

- **Step 1: LCS of X and Y**

  We use a dynamic programming table of size $(n+1) \times (m+1)$. Each entry takes constant time to compute.

  Time Complexity: $\mathcal{O}(nm)$

- **Step 2: LPS of Z**

  To compute the LPS, we calculate the LCS of Z and its reverse. Since Z has length k, this takes $\mathcal{O}(k^2)$ time.

  $$\text{Time Complexity: } \mathcal{O}(k^2)$$

- **Overall Time Complexity**

  The total time complexity of the algorithm is:

  $$\mathcal{O}(nm + k^2)$$

  Since $k \leq \min(n, m)$, the second term is always bounded by $\mathcal{O}(n^2)$ in the worst case.

6 Experiments

To contrast the time complexity of our solutions, we gave both a brute-force solution and the two-step dynamic programming (DP) solution. Implementation notes, including the experimental results, are available on our GitHub repository (GitHub). In the brute-force algorithm, we saw that for string sizes between 18 characters and below, the time was roughly 200 seconds, therefore exhibiting an empirical time complexity that appeared to grow exponentially, roughly $O(2^n)$. This exponential growth is clearly illustrated in Figure 1.

In comparison to that, the DP approach was implemented using random strings of lengths ranging from 1 to 4,000 characters, which were incremented by 50 characters for each trial. In comparison to brute-force method, it only took 6 seconds to find LCPS of the strings for length 4000 word-string. The time required for every string size was recorded and the data plotted as a function of input size. From Figure 1, the experimental data follows a quadratic growth pattern consistent with the theoretical time complexity of $O(n^2)$ of the dynamic programming solution.

All tests were run on a T4 GPU courtesy of Google Colab, which guaranteed equal computational power for all experiments. Experimental outputs are consistent with our theory projections, thereby affirming the algorithm's scalability and performance trend as a function of input size.

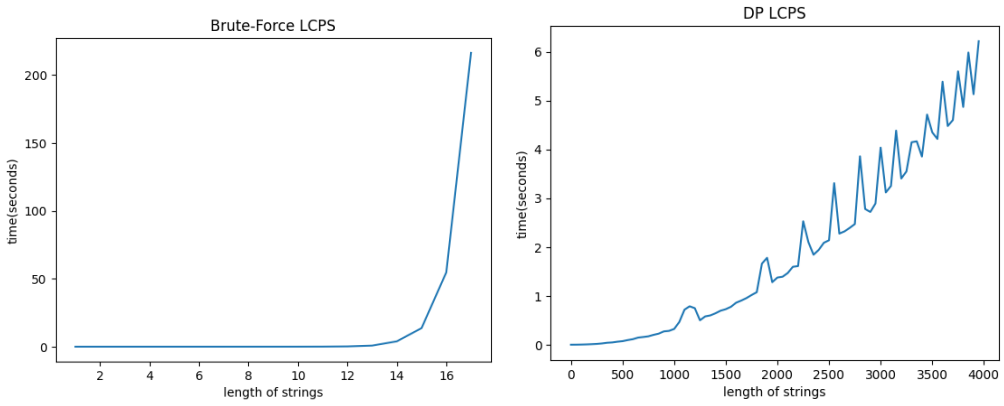

Figure 1: Execution time of the algorithm as a function of input string length, demonstrating $O(n^2)$ complexity.

7 Discussion

The brute-force approach, while correct, suffers from exponential time complexity, rendering it impractical for even moderately sized inputs. The dynamic programming approach improves the feasibility of the problem but at the cost of high-dimensional memory and computation, specifically $\mathcal{O}(n^4)$ space and time. However, it is exact and systematic and may be appropriate when resources permit or when inputs are small.

The two-step approach demonstrates a significant improvement in efficiency by leveraging the structural properties of the problem: first isolating the shared subsequence and then checking for palindromicity. This modular strategy leads to an effective solution with $\mathcal{O}(n^2)$ time complexity, which is far more scalable and practical.

8 Conclusion

In this paper, we introduced and analyzed three algorithms for solving the Longest Common Palindromic Subsequence (LCPS) problem. We began with a brute-force method to establish a baseline, then introduced a dynamic programming algorithm with polynomial complexity in four dimensions. Lastly, we proposed an efficient two-step method by composing LCS and LPS subroutines, significantly reducing computational overhead.

We provided formal correctness proofs for each method and analyzed their time and space complexities. Our results demonstrate that the two-step approach offers an effective and scalable alternative for solving LCPS in practical scenarios, balancing accuracy and computational cost.

References

[1] Sumaiya Iqbal, Mohammad Rahman, Md. Hasan, and Shihabur Rahman Chowdhury. Computing a longest common palindromic subsequence. In *Proceedings of the 13th International Symposium on Algorithms and Data Structures (WADS 2013)*, 2012.

[2] Constantine Liopoulos and M. Sohel Rahman. A new efficient algorithm for computing the longest common subsequence. *Theory of Computing Systems*, 45:355–371, 2009.