Final Report

A subsequence is formed by removing zero or more characters from a given string while maintaining the original order of the remaining characters. Given two sequences X and Y, the goal is to determine the longest sequence Z such that Z is a subsequence of both X and Y. In this analysis, we compared three different approaches to solving the LCS problem:

- Algorithm 1: Recursive Approach
- Algorithm 2: Memoization Approach
- Algorithm 3: Dynamic Programming Approach

In this report, we evaluate the performance of three algorithms in terms of runtime efficiency with increasing input lengths and identify the most suitable algorithm for practical use. Each algorithm was tested with randomly generated string pairs of increasing lengths. Execution times were measured using the time module. The input length was varied as follows:

- For Algorithm 1, lengths from 1 to 15 were used due to exponential growth in runtime.
- For Algorithm 2 and Algorithm 3, lengths up to 100 were tested to explore their scalability.

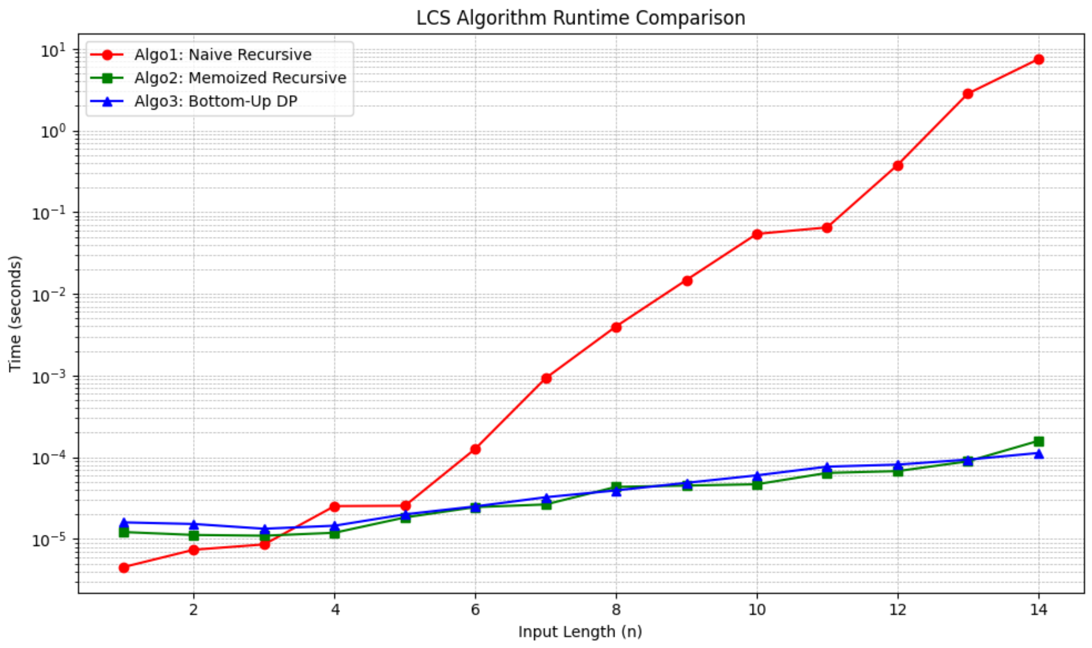

Fig.1: Runtime Performance (Up to Length 15).

Figure 1 displays all three algorithms on a logarithmic time scale: Algorithm 1 (Naive Recursive) shows exponential growth, becoming infeasible beyond length 15. Algorithm 2 (Memoized) and Algorithm 3 (Bottom-up DP) remain efficient, with near-linear growth. This plot highlights the inefficiency of naive recursion even at small input sizes and the clear improvement offered by memoization and bottom-up strategies.

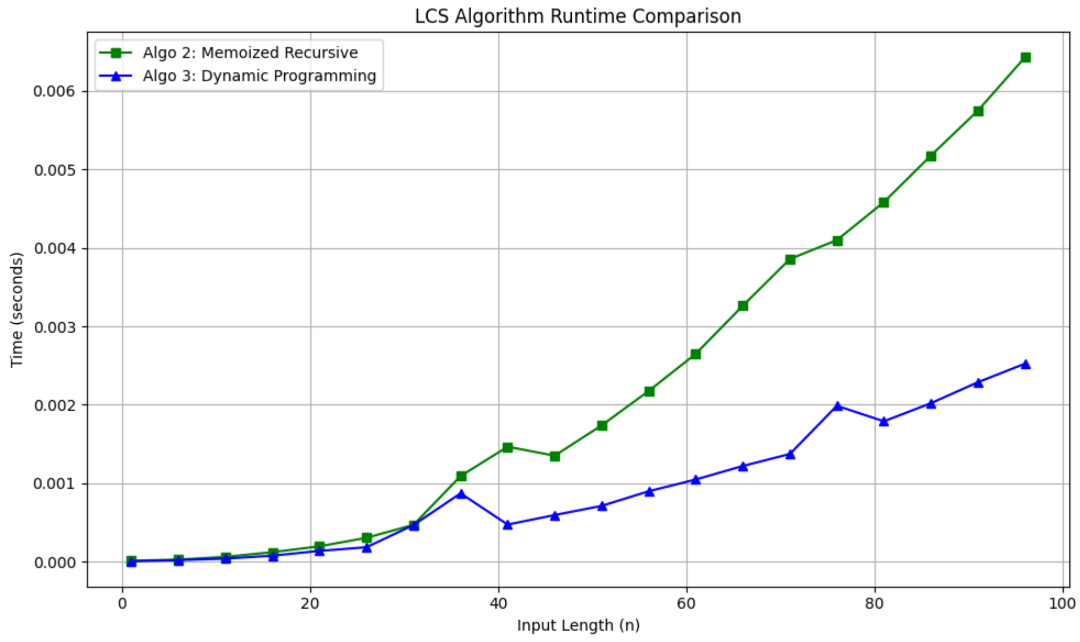

Fig.2: Runtime Performance (Up to Length 100).

Algorithm 1 is capped at 15 due to impractical runtimes. In Figure 2, both Algorithm 2 and Algorithm 3 performs consistently well up to input size 100. But Algorithm 3 is marginally faster in some cases due to extra overhead in memoization table. Despite these minor differences, both algorithms are highly efficient and scalable.

The three LCS algorithms differ significantly in both time and space complexity, which directly impacts their runtime performance and memory usage. The naive recursive algorithm has an exponential time complexity of O(2^n), as it explores all possible subsequences without storing intermediate results, and a space complexity of O(n) due to the recursion stack. This makes it highly inefficient for larger inputs, as evidenced by its rapidly increasing runtime beyond input length 15. The memoized version improves time complexity to O(n×m) by caching results of subproblems, greatly reducing redundant calculations. However, it still has a space complexity of O(n×m+n) because of the memoization table and recursion stack. The bottom-up dynamic programming approach also has a time complexity of O(n×m), but with a more controlled and typically lower space usage of O(n×m) due to its iterative structure and absence of recursive overhead. As a result, Algorithm 3 is the most efficient and scalable in both time and space, making it the preferred method for handling large input sizes.

It will be recommended to use Algorithm 1 only for educational purposes or for extremely small input sizes where simplicity is needed. Algorithm 2 is recommended for most real-world applications where recursion is more intuitive or where space optimization is possible using sparse memo tables. Algorithm 3 is preferable when performance is critical and stack overflow risks must be avoided, such as in embedded systems or memory-constrained environments.

Appendix

Algorithm 1 (Recursive Longest Common Subsequence):

```python
import time
import random
import string
import matplotlib.pyplot as plt

def lcs_recursive(X, Y, m, n):
    if m == 0 or n == 0:
        return 0
    elif X[m - 1] == Y[n - 1]:
        return 1 + lcs_recursive(X, Y, m - 1, n - 1)
    else:
        return max(lcs_recursive(X, Y, m, n - 1), lcs_recursive(X, Y, m - 1, n))

# Function to generate random test cases
def generate_random_string(length):
    return ''.join(random.choices(string.ascii_uppercase, k=length))

# Performance testing
input_sizes = list(range(1, 15))  # Small sizes due to exponential complexity
runtimes = []

for size in input_sizes:
    X = generate_random_string(size)
    Y = generate_random_string(size)

    start_time = time.time()
    lcs_recursive(X, Y, len(X), len(Y))
    end_time = time.time()
    runtimes.append(end_time - start_time)

    # Log individual result
    print("X: " + X + " Y: " + Y)
    print("Time: ", end_time - start_time)


# Plot the results
plt.plot(input_sizes, runtimes, marker='o', linestyle='-')
plt.xlabel("Input Length (n)")
plt.ylabel("Time (seconds)")
plt.title("Runtime of Recursive LCS Algorithm")
plt.show()
```

The recursive approach uses recursion to find the LCS by checking whether characters of the two sequences match or not. If the last characters of both sequences match, the LCS length is incremented by 1, and the problem is solved for the remaining substrings. Otherwise, the function is called recursively for both cases: excluding the last character from one sequence at a time.

This approach follows these steps:

1. If either string is empty, return 0 (base case).
2. If the last characters of both strings match:
   - The LCS length is `1 + LCS(X[0:m-1], Y[0:n-1])`.
3. Otherwise:
   - Compute the LCS excluding the last character from one of the strings:
     - `LCS(X[0:m-1], Y[0:n])`
     - `LCS(X[0:m], Y[0:n-1])`
   - Return the maximum of the two computed values.

Function:

```
def lcs_recursive(X, Y, m, n):
    if m == 0 or n == 0:
        return 0
    elif X[m - 1] == Y[n - 1]:
        return 1 + lcs_recursive(X, Y, m - 1, n - 1)
    else:
        return max(lcs_recursive(X, Y, m, n - 1), lcs_recursive(X, Y, m - 1, n))
```

This function is called with `lcs_recursive(X, Y, len(X), len(Y))`.

Data Structures Used:

The recursive LCS implementation primarily relies on:

- **Strings**: Immutable Python strings are used to store input sequences.
- **Call Stack**: Recursion utilizes the system call stack to store intermediate states, contributing to space complexity.

Time and Space Complexity Analysis:

Time Complexity

The worst-case complexity of this recursive approach is **$O(2^n)$** because each function call branches into two recursive calls. The recurrence relation follows:

$$T(m, n) = T(m-1, n) + T(m, n-1)$$

which expands exponentially.

Space Complexity

- The depth of the recursion tree is **O(m + n)** in the worst case.
- No extra memory is used apart from the recursive stack.

Performance Evaluation:

We tested the recursive LCS implementation on randomly generated strings of varying lengths. The results are as follows:

Runtime Results:

Input Size	X	Y	Execution Time (seconds)
1	G	H	4.53×10^{-6}
2	TR	LO	7.39×10^{-6}
3	ZOV	FKR	8.58×10^{-6}
4	UOJE	IPHB	2.53×10^{-5}
5	XIRDH	LWYGH	2.55×10^{-5}
6	ZFVDBN	RPZZND	1.27×10^{-4}
7	QYLPPNW	FYRMWNY	9.28×10^{-4}
8	FHOTPNZX	WHAPCOWE	3.99×10^{-3}
9	ISWQJNBYD	EAKGAVDWT	1.48×10^{-2}
10	HUJIWCWZAJ	DWGGBFGSCR	5.41×10^{-2}
11	GFVBBKISJWG	NQUEWPQVCUG	6.50×10^{-2}
12	XCXLRWKNIDJF	XXHIGVTXKWUO	3.77×10^{-1}
13	OWOWIMKIIGJLW	YTFSLSVZPBDJP	2.82
14	IFSBVUJRFCQSFY	XIWUDDQKHUCOVN	7.44

Observations:

- Execution time grows exponentially as the input size increases.
- At input size **n = 14**, execution time reaches **7.44 seconds**, confirming the impracticality of this approach for large inputs.

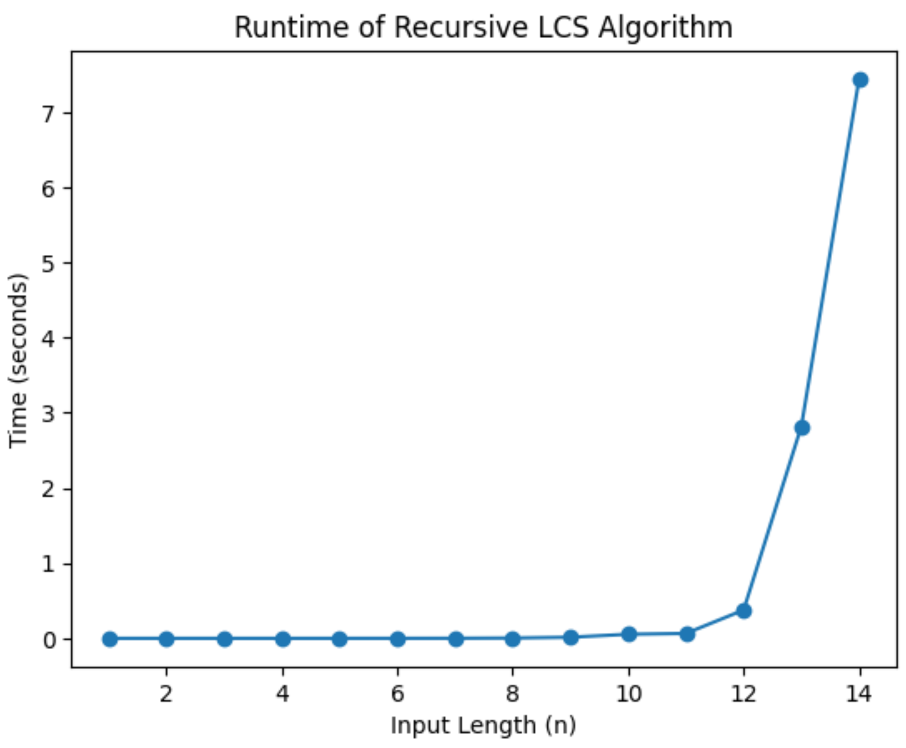

Fig.1: Runtime Performance.

The recursive LCS algorithm is theoretically correct but highly inefficient for large inputs due to its exponential time complexity. Figure 1 shows the plot of execution time vs. input size which confirms the exponential growth of the algorithm.

Algorithm 2 (Memoization-Based Longest Common Subsequence):

```
import time
import random
import string
import matplotlib.pyplot as plt

def lcs_memoized(X, Y):
    m, n = len(X), len(Y)

    # Initialize a 2D memoization table with -1
    # memo[i][j] will store the LCS length of X[0..i-1], Y[0..j-1]
    memo = [[-1 for _ in range(n + 1)] for _ in range(m + 1)]

    # Helper recursive function to compute LCS length
    # with memoization for subproblems.
    def recurse(i, j):
        if i == 0 or j == 0:
            return 0

        # Return cached result if already computed
        if memo[i][j] != -1:
            return memo[i][j]

        # If characters match, move diagonally in both strings
        if X[i - 1] == Y[j - 1]:
            memo[i][j] = 1 + recurse(i - 1, j - 1)
        else:
            # Take the maximum of excluding one character
            memo[i][j] = max(recurse(i - 1, j), recurse(i, j - 1))

        return memo[i][j]

    return recurse(m, n)

# Helper function to generate a random uppercase string of given length
def generate_random_string(length):
    return ''.join(random.choices(string.ascii_uppercase, k=length))

# Performance tests on increasing input sizes
input_sizes = list(range(1, 15))
runtimes = []

# Loop through each input size
```

```
for size in input_sizes:
    X = generate_random_string(size)
    Y = generate_random_string(size)

    print("X:", X, "Y:", Y)

    # Measure execution time of LCS computation
    start_time = time.time()
    lcs_memoized(X, Y)
    end_time = time.time()

    runtime = end_time - start_time
    print("Time:", runtime)
    runtimes.append(runtime)

# Plot the results
plt.plot(input_sizes, runtimes, marker='o', linestyle='-')
plt.xlabel("Input Length (n)")
plt.ylabel("Time (seconds)")
plt.title("Runtime of Memoization-Based LCS Algorithm")
plt.grid(True)
plt.tight_layout()
plt.show()
```

The recursive solution to the Longest Common Subsequence (LCS) problem suffers from exponential time complexity due to overlapping subproblems. The memoization approach (top-down dynamic programming) resolves this inefficiency by storing intermediate results. The memoized LCS algorithm enhances the recursive solution with a caching mechanism. It avoids recomputation by storing results of subproblems in a 2D table.

This approach follows these steps:

1. Use a 2D list *`memo[m+1][n+1]`,* initialized to *`-1`*, where *`memo[i][j]`* stores the length of LCS between *`X[0..i-1]`* and *`Y[0..j-1]`*.
2. For each pair (i, j):
    - If characters match (*`X[i-1] == Y[j-1]`*): set *`memo[i][j] = 1 + memo[i-1][j-1]`*.
    - Else: *`memo[i][j] = max(memo[i-1][j], memo[i][j-1])`*.
3. If a value is already computed, return the result from *`memo[m][n]`*.

Function:

```
def lcs_memoized(X, Y):
    m, n = len(X), len(Y)
    memo = [[-1 for _ in range(n + 1)] for _ in range(m + 1)]

    def recurse(i, j):
        if i == 0 or j == 0:
            return 0
        if memo[i][j] != -1:
            return memo[i][j]
        if X[i - 1] == Y[j - 1]:
            memo[i][j] = 1 + recurse(i - 1, j - 1)
        else:
            memo[i][j] = max(recurse(i - 1, j), recurse(i, j - 1))
        return memo[i][j]

    return recurse(m, n)
```

This function is called with `lcs_memoized (X, Y).`

Data Structures Used:

The memoized LCS implementation primarily relies on:

- **Strings**: Immutable Python strings are used to store input sequences.
- **2D List (memo):** A 2D array of size *(m+1)×(n+1)* initialized with -1, used to store intermediate LCS results for subproblems. This avoids recomputation and drastically reduces time complexity.
- **Recursive Call Stack:** Utilizes the system call stack to store intermediate states, contributing to space complexity. Limited to depth m + n.

Time and Space Complexity Analysis:

Time Complexity

- Each subproblem `(i, j)` is computed at most once.
- For sequences of lengths m and n, worst case time complexity will be **O(m × n)**.

Space Complexity

- The memoization table requires **O(m × n)** space.
- The recursion stack depth is **O(m + n)** in the worst case.

Performance Evaluation:

We tested the memoized LCS implementation on randomly generated strings of varying lengths. The results are as follows:

Runtime Results:

Input Size	X	Y	Execution Time (seconds)
1	W	A	1.21×10^{-5}
2	VL	LU	1.12×10^{-5}
3	ZOV	FKR	1.09×10^{-5}
4	UOJE	IPHB	1.19×10^{-5}
5	XIRDH	LWYGH	1.83×10^{-5}
6	ZFVDBN	RPZZND	2.46×10^{-5}
7	QYLPPNW	FYRMWNY	2.65×10^{-5}
8	FHOTPNZX	WHAPCOWE	4.34×10^{-5}
9	ISWQJNBYD	EAKGAVDWT	4.51×10^{-5}
10	HUJIWCWZAJ	DWGGBFGSCR	4.67×10^{-5}
11	GFVBBKISJWG	NQUEWPQVCUG	6.44×10^{-5}
12	ORILKRNAEULJ	TAGHZMYJAESJ	6.77×10^{-5}
13	GGCPHWUWBRKKA	BKRGFZWXZKIEC	8.92×10^{-5}
14	QYDXGKHXQRCASS	OBOERCNLNAYXAT	1.59×10^{-4}

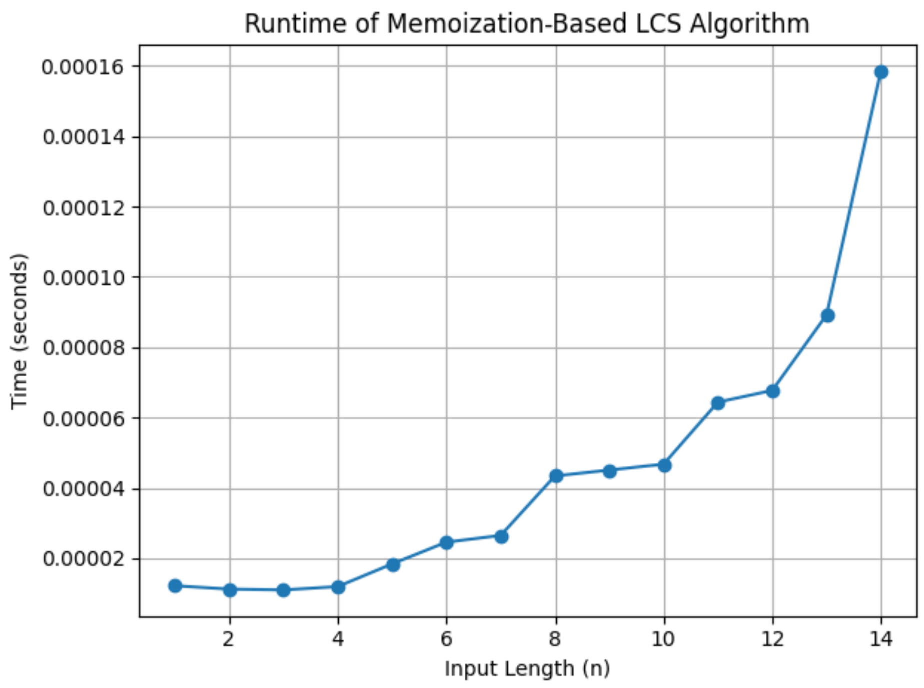

Fig.1: Runtime Performance.

Observations:

- Execution time grows in polynomial time rather than exponential as input size increases.
- At input size **n = 14**, execution time reaches **0.00016 seconds**, showing the improvement of this approach for large inputs.
- Memory usage can be high due to the 2D memoization table.

The memoization approach significantly optimizes the recursive LCS algorithm by caching subproblem results. It transitions the solution from exponential to polynomial time, making it scalable for moderately large input sizes. Figure 1 shows the plot that displays the runtime performance with input length on the x-axis and time on the y-axis. The graph confirms a polynomial growth trend, consistent with the expected O(m x n) complexity. The growth is much flatter than the exponential recursion, highlighting the advantage of memorization.