Longest Common Subsequence

Source Code:

```
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 = []
```

```python
# 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()
```

Algorithm 2 (Memoization-Based Longest Common Subsequence):

A subsequence is formed by removing zero or more characters from a given string while maintaining the original order of the remaining characters. For example, the subsequences of “ABC” include “A”, “B”, “C”, “AB”, “AC”, “BC”, and “ABC”. Given two sequences, we need to find the length of their longest subsequence that appears in both sequences in the same order but not necessarily contiguously.

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}

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.

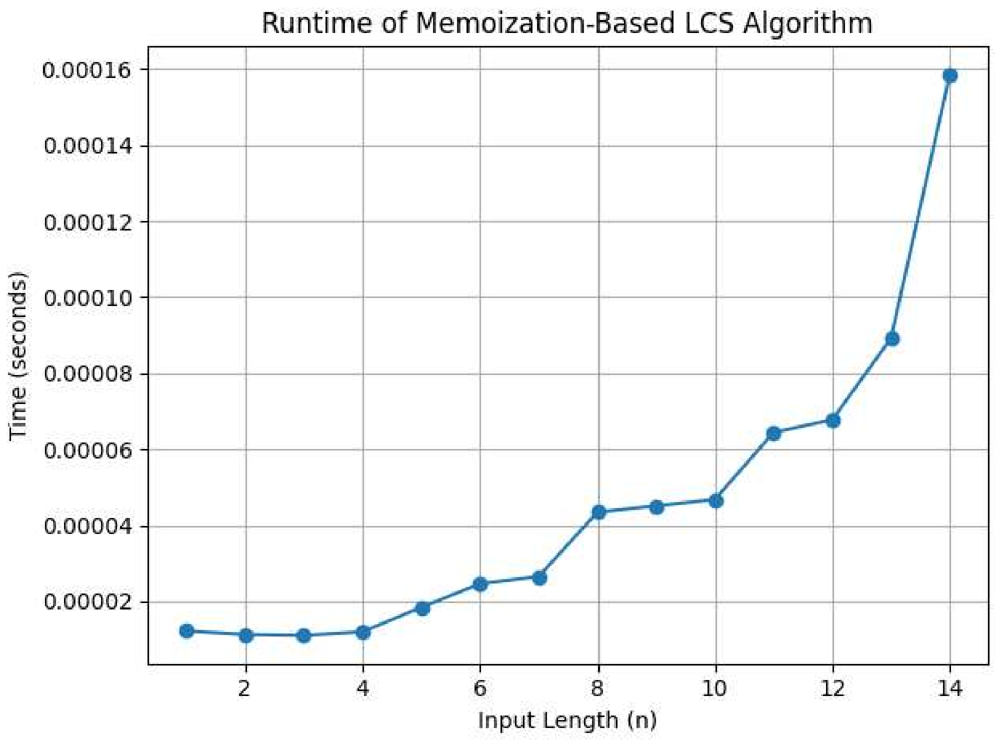

Fig.1: Runtime Performance.

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.