Longest Common Subsequence

Source Code:

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

def lcs_dynamic(X, Y):
    """
    Parameters:
    - X (str): First input string
    - Y (str): Second input string

    Returns:
    - int: Length of the LCS between X and Y
    """
    m, n = len(X), len(Y)

    # Create a 2D list (DP table) initialized with 0
    # dp[i][j] represents the LCS length between X[0..i-1] and Y[0..j-1]
    dp = [[0] * (n + 1) for _ in range(m + 1)]

    # Fill the DP table row by row
    for i in range(m):
        for j in range(n):
            if X[i] == Y[j]:
                # If characters match, add 1 to the value from the
previous diagonal cell
                dp[i + 1][j + 1] = dp[i][j] + 1
            else:
                # Otherwise, take the maximum from left or above
                dp[i + 1][j + 1] = max(dp[i][j + 1], dp[i + 1][j])

    # The bottom-right cell contains the final LCS length
    return dp[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))
```

```python
# Performance tests on increasing input sizes
input_sizes = list(range(1, 16))
runtimes = []

# Loop over different input sizes
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_dynamic(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 DP LCS Algorithm")
plt.grid(True)
plt.tight_layout()
plt.show()
```

Algorithm 3 (Dynamic Programming 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 recursive and memoized approaches are functionally correct but may face stack limitations or inefficiencies in practice. The bottom-up dynamic programming (DP) method resolves these problems by iteratively solving subproblems and filling a table of solutions. Instead of using recursion, we build a 2D table dp[i][j], where each cell represents the length of the LCS between the first i characters of string X and the first j characters of string Y.

This approach follows these steps:

1. Use a 2D list `dp[m+1][n+1],` initialized to `0`, where `dp[i][j]` stores the length of LCS between `X[0..i-1]` and `Y[0..j-1]`.
2. For each character `X[i-1]` and `Y[i-1]:`
   - If they match: dp`[i][j] =` dp`[i - 1][j - 1] + 1`.
   - Else: `dp[i][j] = max(dp[i - 1][j], dp[i][j - 1])`.
3. The final result `dp[m][n]` gives the length of the LCS.

Function:

```
def lcs_dynamic(X, Y):
    m, n = len(X), len(Y)
    dp = [[0] * (n + 1) for _ in range(m + 1)]

    for i in range(m):
        for j in range(n):
            if X[i] == Y[j]:
                dp[i + 1][j + 1] = dp[i][j] + 1
            else:
                dp[i + 1][j + 1] = max(dp[i][j + 1], dp[i + 1][j])

    return dp[m][n]
```

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

Data Structures Used:

The memoized LCS implementation primarily relies on:

- **Strings**: Immutable Python strings are used to store input sequences.
- **2D List (dp):** A 2D array of size `(m+1)×(n+1)` initialized with 0, used to store intermediate LCS results for subproblems. This avoids recomputation and drastically reduces time complexity.
- No recursion is used, thus avoiding stack overhead.

Time and Space Complexity Analysis:

Time Complexity

- Each subproblem `(i, j)` is computed at most once.
- For strings of lengths m and n, we compute every combination of prefixes.
- For sequences of m and n, worst case time complexity will be = $m \times n \rightarrow$ **O(m × n)**.

Space Complexity

- Requires a 2D table of size $(m + 1) \times (n + 1) \rightarrow$ **O(m × n)**

Performance Evaluation:

We tested the bottom-up Dynamic Programming 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	1.59×10^{-5}
2	VL	LU	1.52×10^{-5}
3	ZOV	FKR	1.33×10^{-5}
4	UOJE	IPHB	1.45×10^{-5}
5	XIRDH	LWYGH	2.00×10^{-5}
6	ZFVDBN	RPZZND	2.50×10^{-5}
7	QYLPPNW	FYRMWNY	3.24×10^{-5}
8	FHOTPNZX	WHAPCOWE	3.93×10^{-5}
9	ISWQJNBYD	EAKGAVDWT	4.86×10^{-5}
10	HUJIWCWZAJ	DWGGBFGSCR	6.00×10^{-5}
11	GFVBBKISJWG	NQUEWPQVCUG	7.67×10^{-5}
12	ORILKRNAEULJ	TAGHZMYJAESJ	8.13×10^{-5}
13	GGCPHWUWBRKKA	BKRGFZWXZKIEC	9.36×10^{-5}
14	QYDXGKHXQRCASS	OBOERCNLNAYXAT	1.12×10^{-4}

Observations:

- Execution time grows in polynomial time rather than exponential as input size increases.
- At input size **n = 14**, execution time reaches **0.00012 seconds**, showing the improvement of this approach for large inputs.

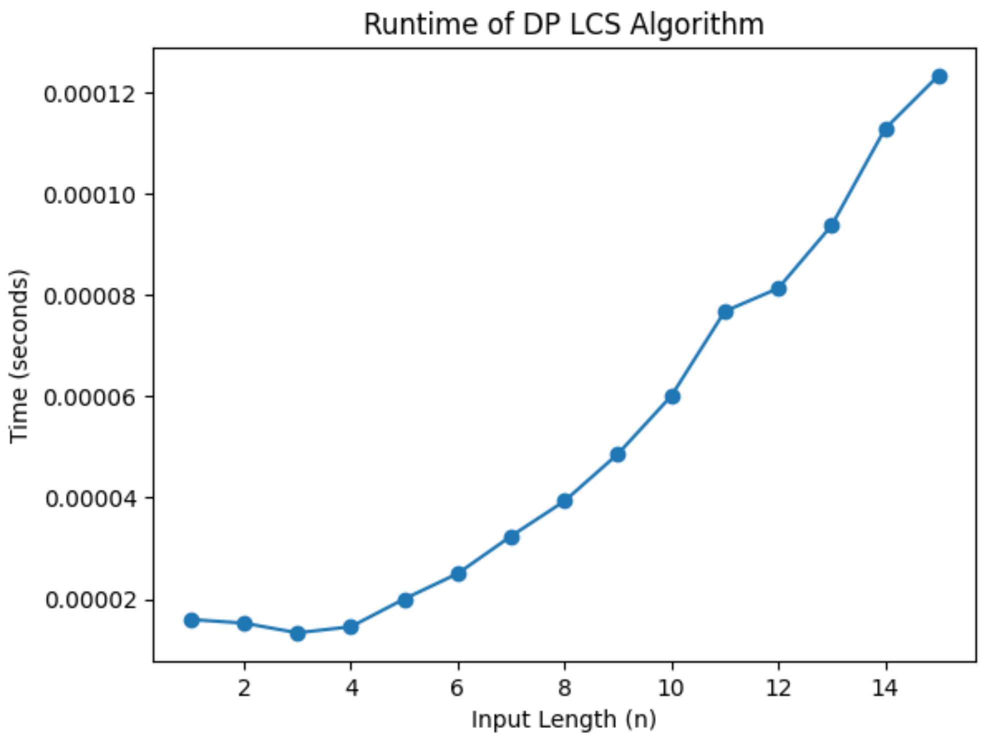

Fig.1: Runtime Performance.

The dynamic programming approach significantly optimizes the recursive LCS algorithm by caching subproblem results and the memorization algorithm by removing the memorization table. 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.