Longest Common Subsequence

Source Code:

```
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()
```

Algorithm 1 (Recursive 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 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.

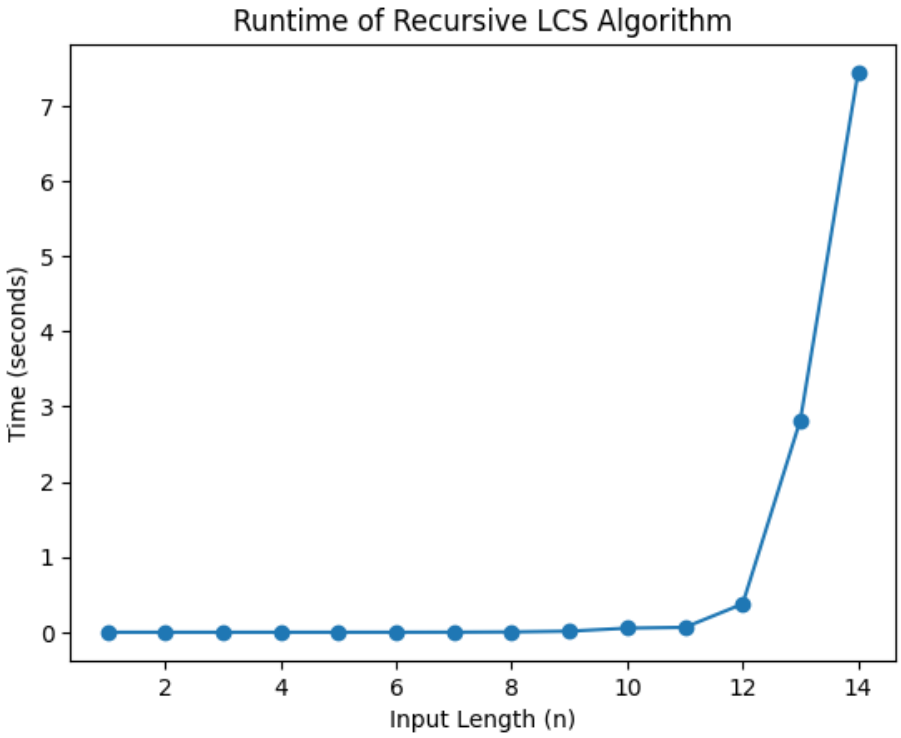

Fig.1: Runtime Performance.

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.