Assignment 8

Manan Patel

Question 1: Data Structure, Recursion, Similarity, and Differences

a. Data Structure

To solve the Longest Common Substring problem using dynamic programming, we utilize a 2D array `dp_table` of size $(m + 1) \times (n + 1)$, where m and n are the lengths of the input strings `str1` and `str2`. Each entry `dp_table[i][j]` represents the length of the longest common substring ending at indices `str1[i-1]` and `str2[j-1]`.

b. Recursion

The recursive relationship is defined as:

$$\texttt{dp\_table[i][j]} = \begin{cases} \texttt{dp\_table[i-1][j-1]} + 1 & \text{if } \texttt{str1[i-1]} = \texttt{str2[j-1]} \\ 0 & \text{otherwise.} \end{cases}$$

The maximum length is updated whenever `dp_table[i][j]` exceeds the current maximum.

c. Similarity

Both Longest Common Subsequence (LCS) and Longest Common Substring (LCSu) uses a 2D dynamic programming table and check for character matches between two strings.

d. Differences

1. **Longest Common Subsequence (LCS)**: A subsequence allows discontinuous characters to form the common sequence.

2. **Longest Common Substring (LCSu)**: A substring requires the matched characters to be continuous.

Question 2: Algorithm

Pseudocode

```
FUNCTION LongestCommonSubstringLength(str1, str2)
    len1 = LENGTH(str1)
    len2 = LENGTH(str2)
    INITIALIZE dp_table[len1+1][len2+1] TO 0
    max_length = 0
    end_index = 0

    FOR row FROM 1 TO len1
        FOR col FROM 1 TO len2
            IF str1[row-1] == str2[col-1] THEN
                dp_table[row][col] = dp_table[row-1][col-1] + 1
                IF dp_table[row][col] > max_length THEN
                    max_length = dp_table[row][col]
                    end_index = row
            ELSE
                dp_table[row][col] =  0
            END IF
        END FOR
    END FOR

    RETURN max_length, end_index, dp_table
END FUNCTION

FUNCTION ExtractLongestCommonSubstring(str1, end_index, max_length)
    RETURN SUBSTRING(str1, end_index - max_length, max_length)
END FUNCTION
```

Question 3: Time and Space Complexity

To analyze the time and space complexity, we will break it down by each function in the algorithm:

Function: `longest_common_substring_length`

- **Initialization of DP Table:**
  - A 2D table, `dp_table`, of size $(m+1) \times (n+1)$ is initialized with zeros.
  - This initialization takes $O(m \times n)$ time since all elements of the table must be set to 0.
- **Nested Loops to Fill the Table:**
  - The algorithm iterates over each character of `str1` (outer loop) and `str2` (inner loop), comparing them.
  - For each pair of indices (i, j), a constant amount of work is performed (comparison, assignment, and updating `max_length` and `end_index`).
  - The total number of iterations is $m \times n$, where m is the length of `str1` and n is the length of `str2`.
  - Thus, the time complexity of this step is $O(m \times n)$.
  - Combining the initialization and the nested loop, the total time complexity is $O(m \times n)$.

Function: `extract_longest_common_substring`

- **Substring Extraction:**
  - The function extracts a substring from `str1` using Python's slicing.
  - Slicing takes $O(k)$ time, where k is the length of the longest common substring. However, since $k \leq \min(m, n)$, this operation is bounded by $O(\min(m, n))$.

  - Thus, the overall time complexity for algorithm is $O(m \times n)$.

- **Space Complexity:**
  - The algorithm uses a 2D table, `dp_table`, of size $(m+1) \times (n+1)$ to store intermediate results.
  - No additional space is used beyond a few variables (`max_length`, `end_index`), which are $O(1)$.
  - Thus, the overall space complexity is $O(m \times n)$.