1143. Longest Common Subsequence

Medium | String | Dynamic Programming | Leetcode Link

Problem Description

In this LeetCode problem, we are dealing with finding the longest common subsequence between two strings `text1` and `text2`. A **subsequence** is defined as a sequence that can be derived from another sequence by deleting some elements without changing the order of the remaining elements. A **common subsequence** means a sequence that exists in both strings.

The challenge is to determine the length of the longest sequence that exists in both `text1` and `text2`. If there is no such sequence, then the result should be `0`.

The task is not to be confused with finding common substrings (which are required to occupy consecutive positions within the original strings). Subsequences are more forgiving as they are not bound by the need for continuity.

Intuition

The solution leverages **Dynamic Programming (DP)**, a method for solving complex problems by breaking them down into simpler subproblems. The idea is to build up a solution using previously computed results for smaller problems.

To approach this, we create a 2D array, `f`, with dimensions `(m+1) x (n+1)`, where `m` and `n` are the lengths of `text1` and `text2`, respectively. The value of `f[i][j]` will hold the length of the longest common subsequence between the first `i` characters of `text1` and the first `j` characters of `text2`.

We iterate over both strings and fill this table based on the following rules:

- If the characters at the current position in both strings match, then the longest common subsequence would be that of the previous characters of both strings plus one (because we include this matching character).
- If the characters do not match, we take the maximum of two possible cases:
  - Including one less character from `text1` and the current number of characters from `text2`.
  - Including the current number of characters from `text1` and one less from `text2`.

The final answer, the length of the longest common subsequence, will be the value stored in `f[m][n]` after the entire table is filled.

Solution Approach

The implemented solution uses a two-dimensional dynamic programming table `f` to store the lengths of longest common subsequences for different parts of `text1` and `text2`. The structure of `f` makes it easy to build the solution incrementally. Each entry `f[i][j]` in this table represents the length of the longest common subsequence between the first `i` characters of `text1` and the first `j` characters of `text2`.

The dynamic programming algorithm follows these steps:

1. Initialize a 2D array `f` with `(m + 1)` rows and `(n + 1)` columns to `0`, where `m` is the length of `text1` and `n` is the length of `text2`. The extra row and column are used to handle the base case where one of the strings is of length `0`, which naturally results in a common subsequence of length `0`.

2. Iterate over the rows of the array (corresponding to each character in `text1`) and the columns (corresponding to each character in `text2`), starting from index `1` to include the first character from each string.

3. At each `(i, j)` position, check if characters `text1[i - 1]` and `text2[j - 1]` are equal. If they are, set `f[i][j] = f[i - 1][j - 1] + 1`. This step follows from the fact that if the current characters match, the longest common subsequence would extend from the longest subsequence obtained without including these characters, hence we add `1`.

4. If the characters at `text1[i - 1]` and `text2[j - 1]` aren't equal, we have to choose the longer subsequence that we could obtain by either discarding the current character from `text1` or `text2`. Thus we set `f[i][j] = max(f[i - 1][j], f[i][j - 1])`.

5. After completing the iteration process, the value `f[m][n]` gives the length of the longest common subsequence of `text1` and `text2`.

This algorithm uses dynamic programming's bottom-up approach, where we start by solving the smallest subproblems (i.e., subsequences of length `0` or `1`) and use their solutions to build up to the answer for the entire sequence. The choice between when to add `+ 1` to the subsequence length or when to take the maximum of two choices is determined by the match between characters.

The beauty of this approach lies in its efficiency and the fact that it guarantees the optimal solution by systematically exploring all possibilities and making the optimal choice at each step, based on the choices made in the previous steps.

Example Walkthrough

Let's take a small example with `text1 = "abcde"` and `text2 = "ace"`. Here, we want to find the length of the longest common subsequence between these two strings.

1. We initialize a 2D array `f` with dimensions `(5 + 1) x (3 + 1)` to `0`, resulting in a table with 6 rows and 4 columns. Initially, the table is filled with zeros. The extra row and column act as base cases for subproblems where one of the strings is empty.

2. We start iterating from `i = 1` to `5` (for `text1`) and `j = 1` to `3` (for `text2`). For understanding, let's describe each character's ASCII value from each string: `text1` 'a'=97, 'b'=98, etc.; `text2` 'a'=97, 'c'=99, 'e'=101.

3. When `i = 1` and `j = 1`, we compare `text1[i - 1]` (which is 'a') with `text2[j - 1]` (which is also 'a'). Since they match, we set `f[i][j]` to `f[i - 1][j - 1] + 1`. Thus `f[1][1] = f[0][0] + 1 = 1`.

4. We continue this process. When `i = 2` and `j = 1`, the characters `text1[i - 1]` (which is 'b') and `text2[j - 1]` (which is 'a') do not match. We compare `f[i - 1][j]` and `f[i][j - 1]` (both are `1`), and take the maximum, which is `1`. So `f[2][1]` remains `1`.

5. The process repeats. At `i = 3`, `j = 2`, 'c' matches 'c', so `f[3][2] = f[2][1] + 1 = 2`.

After iterating through all the characters, the table `f` would look like this:

```
0  0  0  0
0  1  1  1
0  1  1  1
0  1  2  2
0  1  2  2
0  1  2  3
```

6. The last cell `f[5][3]` contains the value `3`, which is the length of the longest common subsequence of `text1` and `text2`. This sequence can be read off from the table by tracing back the cells that gave rise to the maximum values. In this case, the sequence is "ace" which is indeed the longest common subsequence of `text1` and `text2`.

This example walkthrough illustrates how dynamic programming methodically fills out the 2D table to come up with the length of the longest common subsequence. Starting from small subproblems, it builds up the solution to the entire problem, ensuring that all possibilities are considered and the most optimal decision is made at every step.

Python Solution

```python
class Solution:
    def longestCommonSubsequence(self, text1: str, text2: str) -> int:
        # Get the lengths of both input strings
        len_text1, len_text2 = len(text1), len(text2)

        # Initialize a 2D array (list of lists) with zeros for dynamic programming
        # The array has (len_text1 + 1) rows and (len_text2 + 1) columns
        dp_matrix = [[0] * (len_text2 + 1) for _ in range(len_text1 + 1)]

        # Loop through each character index of text1 and text2
        for i in range(1, len_text1 + 1):
            for j in range(1, len_text2 + 1):
                # If the characters match, take the diagonal value and add 1
                if text1[i - 1] == text2[j - 1]:
                    dp_matrix[i][j] = dp_matrix[i - 1][j - 1] + 1
                else:
                    # If the characters do not match, take the maximum of the value from the left and above
                    dp_matrix[i][j] = max(dp_matrix[i - 1][j], dp_matrix[i][j - 1])

        # The bottom-right value in the matrix contains the length of the longest common subsequence
        return dp_matrix[len_text1][len_text2]

```

Java Solution

```java
class Solution {
    public int longestCommonSubsequence(String text1, String text2) {
        // Lengths of the input strings
        int length1 = text1.length();
        int length2 = text2.length();

        // Create a 2D array to store the lengths of longest common subsequences
        // for all subproblems, initialized with zero
        int[][] dp = new int[length1 + 1][length2 + 1];

        // Build the dp array from the bottom up
        for (int i = 1; i <= length1; ++i) {
            for (int j = 1; j <= length2; ++j) {
                // If characters match, take diagonal value and add 1
                if (text1.charAt(i - 1) == text2.charAt(j - 1)) {
                    dp[i][j] = dp[i - 1][j - 1] + 1;
                }
                // If characters do not match, take the maximum value from
                // the left (dp[i][j-1]) or above (dp[i-1][j])
                else {
                    dp[i][j] = Math.max(dp[i - 1][j], dp[i][j - 1]);
                }
            }
        }
        // The bottom-right cell contains the length of the longest
        // common subsequence of text1 and text2
        return dp[length1][length2];
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    // Function to find the length of the longest common subsequence in two strings.
    int longestCommonSubsequence(string text1, string text2) {
        int text1Length = text1.size(), text2Length = text2.size();
        // Create a 2D array to store lengths of common subsequence at each index.
        int dp[text1Length + 1][text2Length + 1];

        // Initialize the 2D array with zero.
        memset(dp, 0, sizeof dp);

        // Loop through both strings and fill the dp array.
        for (int i = 1; i <= text1Length; ++i) {
            for (int j = 1; j <= text2Length; ++j) {
                // If current characters match, add 1 to the length of the sequence
                // until the previous character from both strings.
                if (text1[i - 1] == text2[j - 1]) {
                    dp[i][j] = dp[i - 1][j - 1] + 1;
                } else {
                    // If current characters do not match, take the maximum length
                    // achieved by either skipping the current character of text1 or text2.
                    dp[i][j] = max(dp[i - 1][j], dp[i][j - 1]);
                }
            }
        }

        // Return the value in the bottom-right cell which contains the
        // length of the longest common subsequence for the entire strings.
        return dp[text1Length][text2Length];
    }
};

```

Typescript Solution

```typescript
// Defines a function that computes the length of the longest common subsequence
// between two strings, 'text1' and 'text2'.
function longestCommonSubsequence(text1: string, text2: string): number {
    const text1Length = text1.length; // length of the first input text
    const text2Length = text2.length; // length of the second input text
    // Create a 2D array 'dp' (for Dynamic Programming) to store lengths of
    // subsequences. +1 is for the base case where one of the strings is empty.
    const dp: number[][] = Array.from({ length: text1Length + 1 }, () => Array(text2Length + 1).fill(0));

    // Populate the 'dp' array
    for (let i = 1; i <= text1Length; i++) {
        for (let j = 1; j <= text2Length; j++) {
            // Check if the current character of 'text1' matches with the current
            // character of 'text2'
            if (text1[i - 1] === text2[j - 1]) {
                // If there's a match, increment the length of the subsequence
                // considering characters up to i and j are part of the subsequence
                dp[i][j] = dp[i - 1][j - 1] + 1;
            } else {
                // If there's no match, take the maximum length from either
                // excluding the current character of 'text1' or 'text2'
                dp[i][j] = Math.max(dp[i - 1][j], dp[i][j - 1]);
            }
        }
    }

    // The bottom-right value in 'dp' matrix contains the length of the longest
    // common subsequence of 'text1' and 'text2'
    return dp[text1Length][text2Length];
}

```

Time and Space Complexity

The given code defines a function `longestCommonSubsequence` that calculates the length of the longest common subsequence between two strings, `text1` and `text2`. Here is an analysis of its time and space complexities:

- **Time Complexity:** The function uses two nested loops that iterate over the lengths of `text1` and `text2`. Each cell `f[i][j]` is computed only once and in constant time. Therefore, the total number of operations is proportional to the product of the lengths of the two strings. This results in a time complexity of `O(m * n)` where `m` is the length of `text1` and `n` is the length of `text2`.

- **Space Complexity:** Space is allocated for a 2D list `f` with `(m + 1) * (n + 1)` elements, where each element takes up constant space. Considering `m` and `n` as the lengths of `text1` and `text2` respectively, the space complexity is `O(m * n)`, because it is proportional to the product of the two lengths.