1771. Maximize Palindrome Length From Subsequences

Hard String Dynamic Programming Leetcode Link

Problem Description

In this problem, we are tasked with creating the longest palindrome possible by concatenating a subsequence from `word1` with a subsequence from `word2`. A subsequence is a sequence that can be derived from another sequence by deleting some elements without changing the order of the remaining elements. A palindrome is a word or phrase that reads the same backwards as forwards.

The goal is to figure out the length of the longest palindrome that can be formed in this way. If it's not possible to create any palindrome, the result should be 0. It is important to note that subsequence1 and subsequence2 must both be non-empty.

Intuition

To arrive at the solution, we have to consider that a palindrome reads the same forwards and backwards. Therefore, we need to look for matching characters in `word1` and `word2`. Since we can form subsequences, we are not restricted to contiguous characters and we can skip some in order to maximize the length of the palindrome.

The idea is to concatenate `word1` and `word2` into one string and then apply a dynamic programming approach to find the longest palindrome subsequence. The dynamic programming (DP) solution uses a 2D table `f` where `f[i][j]` represents the length of the longest palindromic subsequence between index `i` and `j` in the concatenated string `s`. We initialize this table with zeros and fill it in a way that each cell `f[i][j]` takes into account the characters at `s[i]` and `s[j]`. If they match and belong to different parts (one from `word1` and the other from `word2`), we can increase the length of a valid palindrome by 2. We iterate through the string in a manner such that we solve smaller subproblems first, which then can be used to construct solutions for larger subproblems.

While filling the DP table, we keep an eye out for the characters that come from the point where `word1` ends and `word2` begins. If `s[i]` and `s[j]` match and `i` belongs to `word1` while `j` belongs to `word2`, it implies that we have a palindrome that uses characters from both words. This is when we update our answer (`ans`), which keeps track of the maximum length found so far.

In summary, the solution approach consists of the following steps:

1. Concatenate `word1` and `word2`.
2. Initialize a DP table to store the length of the longest palindromic subsequence.
3. Update the table according to the matching characters in the string.
4. Keep track of the maximum length whenever characters in `word1` and `word2` contribute to the palindrome.
5. Return the maximum length recorded.

Solution Approach

The solution is accomplished by applying dynamic programming (DP), which is an algorithmic technique used to solve problems by breaking them down into subproblems, solving each subproblem only once, and storing their solutions.

Here's a step-by-step explanation of the implementation:

1. **Concatenating the Strings**: The first step is to concatenate `word1` and `word2` into a single string `s`.

2. **Initializing the DP Table**: A 2D DP table `f` is initialized with dimensions `n x n`, where `n` is the length of the concatenated string `s`. Each cell `f[i][j]` in this table will eventually contain the length of the longest palindromic subsequence between indices `i` and `j`. For single characters, which can be considered as the shortest palindromes, the diagonal `f[i][i]` is filled with `1`.

3. **Bottom-Up DP Computation**: The table is filled in a bottom-up manner. The loops are arranged to have `i` decrease from `n-1` to `0` and `j` increase from `i+1` to `n-1`. This ensures that before we try to compute `f[i][j]`, the subproblems `f[i+1][j-1]`, `f[i+1][j]`, and `f[i][j-1]` are already computed.

4. **Updating the DP Table**:

   - **Case When Characters Match**: If `s[i]` matches `s[j]`, it means that characters at these indices can potentially form or extend a palindrome. Therefore, we set `f[i][j] = f[i+1][j-1] + 2`. The term `f[i+1][j-1]` represents the length of the longest palindromic subsequence within the substring that excludes the current matching characters that are just added to the ends of the palindrome. The addition of `2` accounts for the two matching characters.

   - **Checking the Words' Boundaries**: If the matching characters are such that `s[i]` is in `word1` and `s[j]` is in `word2`, it indicates the formation of a palindrome using parts of both `word1` and `word2`. Thus, the current palindrome length `f[i][j]` is compared with the maximum length found so far, and if it's larger, `ans` is updated.

   - **Case When Characters Don't Match**: If `s[i]` does not match `s[j]`, then the longest palindrome at `f[i][j]` is the maximum of the longest palindromic subsequences found by either including `s[i]` and excluding `s[j]` (`f[i+1][j]`), or excluding `s[i]` and including `s[j]` (`f[i][j-1]`). Essentially, this step omits one of the mismatching characters and looks at the best result achieved by the remaining string.

5. **Returning the Result**: The variable `ans` keeps track of the maximum length of a palindrome that can be constructed by using the aforementioned rules. After the loops are finished, `ans` contains the length of the longest palindrome possible, and that is what is returned by the function.

By employing this DP approach, we can efficiently compute the longest palindromic sequence that can be constructed from `word1` and `word2` by reusing solutions to subproblems and building up to the final solution.

Example Walkthrough

Let's say we have `word1 = "abaxy"` and `word2 = "azby"` and we want to find the length of the longest palindrome that can be formed by concatenating subsequences of these two words.

Following the steps of the solution:

1. **Concatenating the Strings**: By concatenating `word1` and `word2`, we get `s = "abaxyazby"`.

2. **Initializing the DP Table**: We create a 2D table `f` with dimensions 10 × 10, as `s` has a length of 10.

3. **Bottom-Up DP Computation**: We traverse the table by starting from the end of `s` and working backwards, ensuring that the substring problems are solved first.

4. **Updating the DP Table**:
   - As we iterate over the table, we check for every pair `s[i]` and `s[j]`.
   - For `i = 4` (the last character of `word1`) and `j = 5` (the first character of `word2`):
     - We note that `s[i] = y` and `s[j] = a` do not match, so `f[4][5]` will be the max of `f[5][5]` and `f[4][4]`, which are both 1, so `f[4][5] = 1`.
   - Continuing the process:
     - For `i = 1` and `j = 8`, we see that `s[i] = b` and `s[j] = b` do match. Therefore, we set `f[1][8] = f[2][7] + 2`. Assume `f[2][7]` was already computed and is 1, so `f[1][8] = 1 + 2 = 3`. This is a valid palindrome consisting of the substrings "b" from `word1` and "b" from `word2`.
   - We continue filling out the table in this manner, looking for matches and updating the length of potential palindromes accordingly.
5. **Returning the Result**: Assume after iterating through the entire table and considering all possible subsequence pairs, the maximum length recorded in our `ans` is 3 (from the palindrome "aba" or "bab"). This means the longest palindrome that can be formed by concatenating subsequences from `word1` and `word2` is of length 3.

In this example, the steps have given us a systematic approach to finding matches between two words and using these to find a palindrome by employing dynamic programming to avoid redundant calculations.

Python Solution

```python
class Solution:
    def longestPalindrome(self, word1: str, word2: str) -> int:
        # Concatenate both strings to find palindromes across them.
        concatenated_word = word1 + word2
        total_length = len(concatenated_word)

        # Initialize the DP table where dp_table[i][j] will represent
        # the longest palindromic subsequence's length in the substring concatenated_word[i..j].
        dp_table = [[0] * total_length for _ in range(total_length)]

        # Every single character is a palindrome of length 1.
        for i in range(total_length):
            dp_table[i][i] = 1

        # Answer to maintain the maximum length of palindromic subsequence
        # which starts in word1 and ends in word2.
        max_length = 0

        # Fill out the table
        # Start from the end of the string and compare characters.
        for i in range(total_length - 1, -1, -1):
            for j in range(i + 1, total_length):
                # If characters match, add 2 to the length of the palindrome
                # (since 'i' and 'j' are the same, they make a pair).
                if concatenated_word[i] == concatenated_word[j]:
                    dp_table[i][j] = dp_table[i + 1][j - 1] + 2

                    # If the palindrome starts in the first word and ends in the second word,
                    # update the max palindrome length.
                    if i < len(word1) and j >= len(word1):
                        max_length = max(max_length, dp_table[i][j])
                else:
                    # If the characters don't match, propagate the max length so far.
                    dp_table[i][j] = max(dp_table[i + 1][j], dp_table[i][j - 1])

        return max_length

```

Java Solution

```java
class Solution {
    public int longestPalindrome(String word1, String word2) {
        // Combine both words to create one string
        String combined = word1 + word2;
        int length = combined.length();

        // Create a memoization table to store lengths of palindromic subsequences
        int[][] dpTable = new int[length][length];

        // Every single character is a palindrome of length 1
        for (int i = 0; i < length; ++i) {
            dpTable[i][i] = 1;
        }

        int maxPalindromeLength = 0; // This will store the length of the longest palindrome

        // Fill the memoization table with lengths of longest palindromic subsequences
        for (int i = length - 2; i >= 0; --i) {
            for (int j = i + 1; j < length; ++j) {
                // If characters at current positions in the string are the same
                if (combined.charAt(i) == combined.charAt(j)) {
                    // Increment the length by 2 (plus the length of the palindrome in between the current characters)
                    dpTable[i][j] = dpTable[i + 1][j - 1] + 2;

                    // Check if the current palindrome includes characters from both word1 and word2
                    if (i < word1.length() && j >= word1.length()) {
                        // Update maxPalindromeLength if the current one is the longest so far
                        maxPalindromeLength = Math.max(maxPalindromeLength, dpTable[i][j]);
                    }
                } else {
                    // Choose the longer palindrome obtained either by excluding the current char from word1 or word2
                    dpTable[i][j] = Math.max(dpTable[i + 1][j], dpTable[i][j - 1]);
                }
            }
        }

        // Return the length of the longest palindrome which includes characters from both word1 and word2
        return maxPalindromeLength;
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <string>
#include <cstring>
#include <algorithm>

using namespace std;

class Solution {
public:
    // Function to find the longest palindrome that can be created using the concatenation of word1 and word2
    int longestPalindrome(string word1, string word2) {
        // Combine the two strings
        string combined = word1 + word2;

        // Get the length of the combined string
        int length = combined.size();

        // Create a 2D array to store the longest palindromic subsequence length
        vector<vector<int>> dp(length, vector<int>(length, 0));

        // A single letter is a palindrome of length 1
        for (int i = 0; i < length; ++i)
            dp[i][i] = 1;

        // Initialize the variable to store the result
        int longest = 0;

        // Fill the dp array bottom-up
        for (int i = length - 2; i >= 0; --i) {
            for (int j = i + 1; j < length; ++j) {
                // If the characters match, add 2 to the value from the diagonal
                if (combined[i] == combined[j]) {
                    dp[i][j] = dp[i + 1][j - 1] + 2;
                    // If string is such that 'i' is from word1 and 'j' is from word2
                    // Update the longest palindrome
                    if (i < word1.size() && j >= word1.size()) {
                        longest = max(longest, dp[i][j]);
                    }
                } else {
                    // If the characters don't match, take the max of skipping either character
                    dp[i][j] = max(dp[i + 1][j], dp[i][j - 1]);
                }
            }
        }

        // Return the length of the longest palindrome
        return longest;
    }
};

```

Typescript Solution

```typescript
// Importing necessary functionality for using arrays.
// TypeScript has no standard library like C++, so here we are just using basic arrays.
function longestPalindrome(word1: string, word2: string): number {
    // Combine the two strings
    const combined: string = word1 + word2;

    // Get the length of the combined string
    const length: number = combined.length;

    // Create a 2D array to store the longest palindromic subsequence length
    const dp: number[][] = Array.from({ length }, () => Array(length).fill(0));

    // A single letter is a palindrome of length 1
    for (let i = 0; i < length; ++i)
        dp[i][i] = 1;

    // Initialize the variable to store the maximum length of a palindrome
    let maxPalindromeLength: number = 0;

    // Fill the dp array from bottom to top
    for (let i = length - 2; i >= 0; --i) {
        for (let j = i + 1; j < length; ++j) {
            // If the characters match, add 2 to the value from the diagonal
            if (combined[i] === combined[j]) {
                dp[i][j] = dp[i + 1][j - 1] + 2;

                // If 'i' is from word1 and 'j' is from word2, update maxPalindromeLength
                if (i < word1.length && j >= word1.length) {
                    maxPalindromeLength = Math.max(maxPalindromeLength, dp[i][j]);
                }
            } else {
                // If the characters don't match, take the maximum of skipping either character
                dp[i][j] = Math.max(dp[i + 1][j], dp[i][j - 1]);
            }
        }
    }

    // Return the length of the longest palindrome
    return maxPalindromeLength;
}

```

Time and Space Complexity

Time Complexity

The function `longestPalindrome` uses dynamic programming to solve the problem of finding the longest palindromic subsequence that can be formed using the concatenation of `word1` and `word2`. The time complexity can be analyzed as follows:

- There is a nested loop structure where the outer loop runs for `n` times, where `n` is the length of the concatenated string `s` (that is, `n = len(word1) + len(word2)`). The inner loop also runs for `n` times, although not all iterations are full due to the starting point of the inner loop depending on the outer loop.
- In each iteration, operations are performed with constant time complexity, specifically: assignment, comparison, and arithmetic operations.

Therefore, the total time complexity is `O(n^2)`, where `n` is the length of the concatenated string `s`.

Space Complexity

The space complexity of the function can be determined as follows:

- A 2D list `f` of size `n * n` is created to store the lengths of palindromic subsequences, where `n` is the length of the string `s`. Each cell of the list stores a single integer.
- Constant space is used for variables like `i`, `j`, `ans`, and for some temporary variables used during comparisons and assignments.

As the dominant space requirement is the 2D list `f`, the space complexity is `O(n^2)`, where `n` is the length of the concatenated string `s`.