1682. Longest Palindromic Subsequence II

Medium String Dynamic Programming Leetcode Link

Problem Description

The problem gives us a string `s` and asks us to find the *longest good palindromic subsequence*. A *good palindromic subsequence* is one that:

- Is a subsequence of the original string.
- Reads the same forwards and backwards (is a palindrome).
- Has an even number of characters.
- Has no two consecutive characters that are the same, except for the middle two characters in the subsequence.

A subsequence is a sequence that can be derived from another sequence by deleting some or no elements without changing the order of the remaining elements.

The goal is to calculate the length of this longest subsequence for the provided string.

Intuition

The intuition behind the solution to this problem is derived from classic Dynamic Programming (DP) techniques used in solving palindromic problems, including the Longest Palindromic Subsequence problem.

The idea is to define a recursive function, `dfs(i, j, x)` to handle the following scenarios:

- `i` and `j` are indices that define the current substring `s[i...j]` we are considering. Initially, `i` is 0 (start of the string) and `j` is `len(s)-1` (end of the string).
- `x` is the character that we have just included in our subsequence. This is used to ensure that we do not pick the same character again to satisfy the condition of not having two equal consecutive characters.

If `s[i]` is equal to `s[j]` and different from `x`, `s[i...j]` can contribute to a good palindromic subsequence, and we add 2 to our subsequence count and recurse into the subproblem `dfs(i+1, j-1, s[i])`.

Otherwise, we cannot pick `s[i]` and `s[j]` together. So, we have two options - either skip `s[i]` or skip `s[j]`, and take the maximum result from these two choices. That is `max(dfs(i + 1, j, x), dfs(i, j - 1, x))`.

Using caching (`@cache` decorator from Python's functools library), we avoid recomputation of the same subproblems, thus, optimizing our solution by storing the results of previous computations.

After the recursive process, we obtain an answer from our `dfs` function, which is the length of the longest good palindromic subsequence. We then clear the cache to free up memory and return the computed answer.

Solution Approach

The implementation uses a top-down approach with memoization, a common technique in dynamic programming. Here's a step-by-step explanation of the algorithm and the various components used in the solution:

1. **Memoization Decorator `@cache`:**
   - The `@cache` decorator from Python's `functools` library is used to automatically memorize the results of the recursive function `dfs`. This means that any previously computed values for a particular set of arguments will not be recomputed; instead, the cached value will be returned. The function `dfs.cache_clear()` is used to clear the cache after the main computation is complete to avoid holding onto unnecessary memory references.
2. **Recursive Function `dfs(i, j, x)`:**
   - The `dfs` function is the core of this solution. It takes three parameters: `i` and `j` are the indices indicating the subset of the string `s` we are currently looking at; `x` is a character that represents the last character that was added to the palindromic subsequence.
   - Base Case: When `i >= j`, it means that the pointers have crossed each other, or they are at the same position, which implies there are no characters left to consider for the palindromic subsequence. In this scenario, since we are looking for an even-length palindrome, the function returns 0.
3. **Palindrome and Character Condition Check:**
   - The condition `if s[i] == s[j] and s[i] != x:` checks whether the characters at the start and end of our current string subset can be added to form a longer palindrome without violating the rule of not having two identical consecutive characters (other than the middle two in an even-length palindrome).
4. **Processing and Recursion:**
   - If the condition is met, we extend our good palindromic subsequence by two (`+2`, for `s[i]` and `s[j]`) and recursively call the function for the next subset `dfs(i + 1, j - 1, s[i])`.
   - `s[i]` is passed as the new value of `x` because it is the character that has just been chosen. We essentially look at the remaining substring inside the current palindrome boundaries, excluding the matching characters.
5. **Exploring Alternate Subsequences:**
   - If the above condition does not hold, we have two other possible subsequences to consider. One where we exclude `s[i]` and another where we exclude `s[j]`. We recursively call `dfs(i + 1, j, x)` and `dfs(i, j - 1, x)` and then select the maximum value as the current result.
6. **Returning the Result:**
   - The initial call to the `dfs` function starts with the full string and an empty string as `x` to denote that no character has been chosen yet. The final result is the longest length of the good palindromic subsequence obtained.

By this process, we're ensuring that only valid characters are added to the subsequence while maximizing the length by considering all possible subsequences, and we're doing it efficiently by caching intermediate results.

Example Walkthrough

Let's illustrate the solution approach using a simple example. Consider the string `s = "abbad"`. We want to find the length of the longest good palindromic subsequence.

Here's a step-by-step walkthrough using the recursive function `dfs(i, j, x)`:

1. **Initial Call**: We start with `dfs(0, len(s)-1, '')`, which means our current string is "abbad" and we haven't chosen any character yet (represented by `x = ''`).

2. **First Recursive Step**: Since `s[0]` ('a') is not equal to `s[4]` ('d'), we can't choose both, so we need to decide whether to include 'a' or 'd'.

   - We consider two recursive calls: `dfs(0 + 1, 4, '')` and `dfs(0, 4 - 1, '')`.
3. **Exploring Options**:

   - For `dfs(1, 4, '')`, our current string is "bbad". The first and last characters are 'b' and 'd', which are not the same, so we again have two options: `dfs(2, 4, '')` and `dfs(1, 3, '')`.

   - For `dfs(0, 3, '')`, our current string is "abba". Here, the first and last characters are the same, and since `x` is empty (meaning the last included character isn't 'a'), we can include them in our subsequence, leading to a new call `dfs(1, 2, 'a')`.

4. **Finding a Match**:

   - Now, in `dfs(1, 2, 'a')`, we have the string "bb" which is not allowed since it contains consecutive 'b's, so we can't choose both. We explore `dfs(2, 2, 'a')` (excluding the first 'b') and `dfs(1, 1, 'a')` (excluding the second 'b').
5. **Completing the Recursion**:

   - When `i` equals `j`, we've reached a single character, which cannot form a good palindrome by itself, so in both `dfs(2, 2, 'a')` and `dfs(1, 1, 'a')`, the result would be 0.
6. **Backtracking and Picking the Best Option**:

   - As we backtrack, we realize that choosing 'a' (from "abba") was the best decision. We add 2 to our count (for the two 'a's), and since we've exhausted the string, the recursion starts returning to the initial call, keeping track of the best length found.
7. **Result**: After examining all possibilities, we find that the longest good palindromic subsequence is "abba" with a length of 4.

By caching results along the way, if at any point the same subproblem occurs, the algorithm will fetch the result from the cache, improving efficiency by reducing redundant computations. Finally, we clear the cache to ensure no memory is wasted once we have our result.

Python Solution

```python
from functools import lru_cache

class Solution:
    def longest_palindrome_subseq(self, s: str) -> int:
        # Decorator for memoization to optimize the recursive function.
        @lru_cache(maxsize=None)
        def dfs(start, end, last_char):
            # Base case: if pointers cross, no palindrome can be formed.
            if start >= end:
                return 0

            # Recursive case: if characters at start and end match,
            # and they are different from the last character in the sequence
            # add 2 to the length (for the two matching characters) and
            # move both pointers inward.
            if s[start] == s[end] and s[start] != last_char:
                return dfs(start + 1, end - 1, s[start]) + 2
            else:
                # Recursive case: if the characters don't match,
                # or are the same as last_char, try removing one character from
                # either the start or the end and take the max.
                return max(dfs(start + 1, end, last_char), dfs(start, end - 1, last_char))

        # Call the dfs function with initial values.
        result = dfs(0, len(s) - 1, '')

        # Clear the cache after completing the calculation.
        # This is particularly useful if the method is used multiple times.
        dfs.cache_clear()

        # Return the result of the longest palindromic subsequence.
        return result

# Example usage:
# sol = Solution()
# print(sol.longest_palindrome_subseq("bbbab"))  # Output: 4

```

Java Solution

```java
import java.util.Arrays; // Import Arrays utility for filling the array

class Solution {
    // Declare a 3D array to memoize the results.
    private int[][][] memo;
    // Declare a variable to hold the input string.
    private String str;

    // Method to find the length of the longest palindromic subsequence.
    public int longestPalindromeSubseq(String s) {
        // Length of the string.
        int n = s.length();
        // Initialize the string.
        this.str = s;
        // Initialize the 3D array with size [n][n][27] and default values -1.
        memo = new int[n][n][27];
        for (int[][] l1Array : memo) {
            for (int[] l2Array : l1Array) {
                Arrays.fill(l2Array, -1); // Fill second level arrays with -1.
            }
        }
        // Start the depth-first search from the whole string and character 'z' + 1 as the default previous character.
        return dfs(0, n - 1, 26);
    }

    // Depth First Search (dfs) to calculate the longest palindromic subsequence.
    private int dfs(int i, int j, int prevCharIdx) {
        // Base case: if the start index is greater or equal to the end index, return 0.
        if (i >= j) {
            return 0;
        }
        // If the result is already computed, return it instead of recomputing.
        if (memo[i][j][prevCharIdx] != -1) {
            return memo[i][j][prevCharIdx];
        }
        // Initialize result (ans) variable.
        int ans = 0;
        // If both characters are the same and different from the previous considered character,
        // then we can count this pair and move both pointers.
        if (str.charAt(i) == str.charAt(j) && str.charAt(i) - 'a' != prevCharIdx) {
            ans = dfs(i + 1, j - 1, str.charAt(i) - 'a') + 2;
        } else {
            // Else, try moving either of the pointers to find the longest sequence.
            ans = Math.max(dfs(i + 1, j, prevCharIdx), dfs(i, j - 1, prevCharIdx));
        }

        // Store the computed result in memo array.
        memo[i][j][prevCharIdx] = ans;

        // Return the computed longest length.
        return ans;
    }
}

```

C++ Solution

```cpp
#include <cstring>
#include <functional>
#include <algorithm>

class Solution {
public:
    int memo[251][251][27]; // Memoization table for dynamic programming

    // The main function to calculate the length of the longest palindromic subsequence
    int longestPalindromeSubseq(string s) {
        int n = s.size(); // The length of the string
        memset(memo, -1, sizeof memo); // Initializes the memoization table to -1

        // Depth-first search function to compute the length of LPS for substring [i, j] with previous character index x
        // i: start index of substring, j: end index of substring, x: previous character index
        std::function<int(int, int, int)> dfs = [&](int i, int j, int previousCharIndex) -> int {
            if (i >= j) return 0; // If the substring length is 0 or 1, no palindrome can be formed
            if (memo[i][j][previousCharIndex] != -1) return memo[i][j][previousCharIndex]; // If already computed, return the value

            int longestLength = 0; // Holds the length of the longest palindromic subsequence found
            // Check if characters at indices i and j are the same and not equal to previousCharIndex
            // (represented by the corresponding alphabet index)
            if (s[i] == s[j] && s[i] - 'a' != previousCharIndex)
                // Characters are the same and not just repetitions from before
                // Move inward and add two to count for both characters
                longestLength = dfs(i + 1, j - 1, s[i] - 'a') + 2;
            else
                // Characters are different or repeats, take the max after excluding either character
                longestLength = std::max(dfs(i + 1, j, previousCharIndex), dfs(i, j - 1, previousCharIndex));

            // Store the result in the memo table
            memo[i][j][previousCharIndex] = longestLength;

            return longestLength; // Return the length found
        };

        // Start from the full string and with no previous character (26 is used to represent this)
        return dfs(0, n - 1, 26);
    }
};

```

Typescript Solution

```typescript
// Typescript does not support triple size arrays directly, use a Map for memoization instead.
const memo: Map<string, number> = new Map();

// Utilize a helper function to create the key for our memo map.
function createMemoKey(i: number, j: number, previousCharIndex: number): string {
    return `${i}_${j}_${previousCharIndex}`;
}

// Main function to calculate the length of the longest palindromic subsequence.
function longestPalindromeSubseq(s: string): number {
    const n = s.length;  // The length of the string.

    // Function to compute the length of LPS for substring [i, j] with previous character index 'x'.
    function dfs(i: number, j: number, previousCharIndex: number): number {
        // Base case: if the substring is of length 0 or 1, no palindrome can be formed.
        if (i >= j) return 0;

        // Use the helper function to get the key for our memo map.
        const key = createMemoKey(i, j, previousCharIndex);

        // If already computed, return the value from the memo map.
        if (memo.has(key)) return memo.get(key)!;

        let longestLength = 0;  // Holds the length of the longest palindromic subsequence found.

        // Check if characters at indices i and j are the same and not equal to previousCharIndex.
        if (s[i] === s[j] && (s.charCodeAt(i) - 97) !== previousCharIndex) {
            // Characters are the same and not just repetitions from before.
            // Move inward and add two to count for both characters.
            longestLength = dfs(i + 1, j - 1, s.charCodeAt(i) - 97) + 2;
        } else {
            // Characters are different or repeats, take the max after excluding either character.
            longestLength = Math.max(
                dfs(i + 1, j, previousCharIndex),
                dfs(i, j - 1, previousCharIndex)
            );
        }

        // Store the result in the memo map.
        memo.set(key, longestLength);

        // Return the length found.
        return longestLength;
    }

    // Start from the full string and with no previous character ('26' is used to represent this).
    return dfs(0, n - 1, 26);
}

```

Time and Space Complexity

The code is a recursive function with memoization to find the length of the longest palindromic subsequence in a string. The function `dfs` uses memoization through the `cache` decorator, which means it stores results of subproblems to avoid recomputation.

Time Complexity

The time complexity of the `dfs` function is `O(n^2)` where `n` is the length of the string `s`. This is because in the worst case, we need to compute the result for each pair of starting and ending indices `(i, j)` which are `n*(n-1)/2` pairs, approximately `n^2/2`. However, since results are cached, each pair is computed only once. Therefore, we ignore the constant factor and the complexity is `O(n^2)`.

Space Complexity

The space complexity is also `O(n^2)` due to memoization. The cache needs to store an entry for each unique call to `dfs`, which, as discussed above, has at most `n^2` different argument pairs `(i, j, x)`. The third argument, `x`, does not significantly increase the number of possible states because it represents the previous character and there are only `n` possibilities for it. In practice, `x` is just a character from the input string `s`, so its impact on the complexity is bounded by the length of `s`.