730. Count Different Palindromic Subsequences

Hard · String · Dynamic Programming

Problem Description

The problem requests to find the number of unique non-empty palindromic subsequences within a given string `s`. 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. A palindromic sequence is one that reads the same backward as forward. Unique subsequences mean that they differ from each other by at least one element at some position.

The answer could be very large, so it is required to return the result modulo `10^9 + 7`. This is a common practice in many problems to avoid very large numbers that can cause arithmetic overflow or that are outside the range of standard data type variables.

A key point to remember is that characters can be deleted to form subsequences; they don't have to be contiguous in the original string. Also, since subsequences are not required to be substrings; their characters can be dispersed throughout the string.

Intuition

The solution to this problem is based on dynamic programming. The main idea is to use a 3-dimensional array `dp`, where `dp[i][j][k]` represents the count of palindromic subsequences of type `k` in the substring `s[i:j+1]`. Here, `i` and `j` are the start and end indices of the substring, and `k` refers to the different characters (since this is limited to 4 different characters 'a', 'b', 'c', 'd', the third dimension has a size of 4).

The approach iterates over all substrings of `s`, and for each substring and each character type 'a' to 'd':

1. If both ends of the substring `i` and `j` have the same character `c`, then the palindrome count for this character in `dp[i][j][k]` includes all the palindromes within `s[i+1: j-1]` plus 2 (for the subsequences made by the single character `c` and the one formed by `cc`).
2. If only one end has the character `c`, then the palindrome count for this character in `dp[i][j][k]` is the same as in the substring `s[i: j]` or `s[i+1: j+1]`, depending on which end has the character `c`.
3. If neither end has the character `c`, then the count remains the same as in the substring `s[i+1: j-1]`.

After filling the `dp` array, we sum up the counts for all types of characters at the first and last position `dp[0][n-1]`, where `n` is the length of the string and take modulo `10^9 + 7` for the final answer.

This method ensures that all possible unique palindromic subsequences are counted without duplication, as the dynamic programming array stores the results of previous computations and avoids re-calculating them.

Solution Approach

The given Python solution implements a dynamic programming approach to solve the problem efficiently.

Breaking down the implementation we have:

1. **Initialization:** The solution creates a 3-dimensional list (or array) `dp` which is initialized to hold all zeros. The dimensions of `dp` are `[n][n][4]`, where `n` is the length of the string and `4` represents the four possible characters ('a', 'b', 'c', 'd').
2. **Base Case:** For each character in the input string `s`, we set `dp[i][i][ord(c) - ord('a')] = 1`. This means that for every character in the string, there is exactly one palindromic subsequence of length 1.
3. **Main Loop:** The solution then iterates over all possible lengths of substrings `l` starting from `2` up to `n` inclusive. For each length, it loops over all possible starting indices `i` of the substring.
4. **State Transitions:**
   - If the characters at both ends are the same and are the character `c`, the count `dp[i][j][k]` is calculated by adding `2` (accounting for the subsequences made by the single character and the one formed by two characters) plus the sum of counts of all palindromic subsequences within the substring (ignoring the two ends). This is represented by `sum(dp[i + 1][j - 1])`.
   - If only one of the ends matches the character `c`, the count should be the same as the count of palindromic subsequences ending before the unmatched character.
   - If neither of the ends is the character `c`, we fall back to the count for the smaller substring inside the current substring.
5. **Modular Arithmetic:** As the answer can be very large, every time we calculate the sum, we take modulo `10^9 + 7` to keep the number within the integer range.
6. **Final Answer:** After populating the `dp` array, the answer is the sum of `dp[0][n - 1]` (representing counts of different palindromic subsequences in the entire string from the first to the last character) modulo `10^9 + 7`.

The key data structure used here is the 3D list which holds counts of palindromic subsequences for all substrings and for each character type. The algorithm utilizes dynamic programming by breaking down the problem into smaller subproblems and uses previously computed values to calculate the larger ones, thus optimizing the number of computations required.

Example Walkthrough

Let's consider a small example to illustrate the solution approach. Take the string `s = "abca"`.

1. **Initialization:** We create a 3D array `dp` of size `[4][4][4]`, as the string length `n` is 4. This array will hold zeros initially.
2. **Base Case:** We iterate over each character in `s`. For `i = 0...3` (string indices):
   - `dp[0][0][0] = 1` (for 'a')
   - `dp[1][1][1] = 1` (for 'b')
   - `dp[2][2][2] = 1` (for 'c')
   - `dp[3][3][0] = 1` (for 'a')

In each case, we're adding 1 for a single-character palindrome subsequence.

3. **Main Loop:** We loop through all possible substring lengths `l = 2...4`. In this case, let's look at `l = 2`. We loop over all start indices `i` for this length.
   - For `l = 2`, substring `ab` (`s[0:2]`), we have `i = 0`:
     - The characters at `s[0]` and `s[1]` are different, so for character 'a', `dp[0][1][0] = dp[0][0][0] = 1`.
     - For character 'b', `dp[0][1][1] = dp[1][1][1] = 1`.
4. **State Transitions:** For `l = 2`, continue with the state transitions:
   - Look at substring `bc` (`s[1:3]`), where `i = 1`:
     - Characters at `s[1]` and `s[2]` are different, so `dp[1][2][1] = dp[1][1][1] = 1`, and `dp[1][2][2] = dp[2][2][2] = 1`.
5. **Continue Looping:** Continue with `l = 3` and then `l = 4`, updating the `dp` array based on the rules defined in the solution approach.

   For example, if we check for `l = 3`:

   - Substring `abc` (`s[0:3]`), since both ends are different, `dp[0][2][0] = dp[1][1][0]` and `dp[0][2][1] = dp[1][1][1]`.
6. **Final Answer:** After populating the `dp` array for all substrings of all lengths, our answer would be the sum `sum(dp[0][3])` modulo `10^9 + 7` for the full string length (from the first to the last character).

For our example string `s = "abca"`, the unique palindromic subsequences are 'a', 'b', 'c', 'aa', 'aca'. The count is 5 which is the sum we expect at `dp[0][3]` before taking the modulo. Note that in reality, the `dp` array would hold more data points and aggregating them would include performing modulo operations to keep within numeric limits.

Python Solution

```python
class Solution:
    def countPalindromicSubsequences(self, string: str) -> int:
        # Define the modulus to ensure the return value is within the expected range.
        MOD = 10**9 + 7

        # Get the length of the string to iterate through it.
        length = len(string)

        # Initialize a 3D array to store the count of palindromic subsequences.
        # dp[i][j][k] will store the count for the substring string[i:j+1] with character 'abcd'[k].
        dp = [[[0] * 4 for _ in range(length)] for _ in range(length)]

        # Base case initialization for substrings of length 1.
        for index, character in enumerate(string):
            dp[index][index][ord(character) - ord('a')] = 1

        # Starting from substrings of length 2, build up solutions for longer substrings.
        for substring_length in range(2, length + 1):
            for start in range(length - substring_length + 1):
                end = start + substring_length - 1
                for char in 'abcd':
                    char_index = ord(char) - ord('a')
                    # If the current characters at start and end indices match and match 'char',
                    # count the inner subsequences twice (for including both start and end characters,
                    # and excluding both), and add the counts of all subsequences in between.
                    if string[start] == string[end] == char:
                        dp[start][end][char_index] = (2 + sum(dp[start + 1][end - 1])) % MOD
                    # If only the start character matches 'char', carry over the count from the previous.
                    elif string[start] == char:
                        dp[start][end][char_index] = dp[start][end - 1][char_index]
                    # If only the end character matches 'char', carry over the count from the previous.
                    elif string[end] == char:
                        dp[start][end][char_index] = dp[start + 1][end][char_index]
                    # If neither character matches 'char', the count remains the same as the inner substring.
                    else:
                        dp[start][end][char_index] = dp[start + 1][end - 1][char_index]

        # Finally, sum up all the counts for the entire string and each character 'abcd',
        # and apply the modulus for the result.
        return sum(dp[0][length - 1]) % MOD

```

Java Solution

```java
class Solution {
    // Define the modulus constant for the problem
    private static final int MOD = 1_000_000_007;

    public int countPalindromicSubsequences(String s) {
        int n = s.length(); // Length of the string

        // 3D dynamic programming array to store results
        long[][][] dp = new long[n][n][4];

        // Base case: single character strings
        for (int i = 0; i < n; ++i) {
            dp[i][i][s.charAt(i) - 'a'] = 1;
        }

        // Loop over all possible substring lengths
        for (int len = 2; len <= n; ++len) {
            // Iterate over all possible starting points for substring
            for (int start = 0; start + len <= n; ++start) {
                int end = start + len - 1; // Calculate end index of substring
                // Try for each character a, b, c, d
                for (char c = 'a'; c <= 'd'; ++c) {
                    int charIndex = c - 'a'; // Convert char to index (0 to 3)
                    // Case 1: Characters at both ends match the current character
                    if (s.charAt(start) == c && s.charAt(end) == c) {
                        // Count is sum of inner substring counts + 2 (for the two characters added)
                        dp[start][end][charIndex] = 2 + dp[start + 1][end - 1][0]
                            + dp[start + 1][end - 1][1] + dp[start + 1][end - 1][2]
                            + dp[start + 1][end - 1][3];
                        dp[start][end][charIndex] %= MOD; // Ensure mod operation
                    }
                    // Case 2: Only the start character matches
                    else if (s.charAt(start) == c) {
                        dp[start][end][charIndex] = dp[start][end - 1][charIndex];
                    }
                    // Case 3: Only the end character matches
                    else if (s.charAt(end) == c) {
                        dp[start][end][charIndex] = dp[start + 1][end][charIndex];
                    }
                    // Case 4: Neither end matches the character
                    else {
                        dp[start][end][charIndex] = dp[start + 1][end - 1][charIndex];
                    }
                }
            }
        }

        // Summation of counts for 'a', 'b', 'c', 'd' for the whole string
        long result = 0;
        for (int k = 0; k < 4; ++k) {
            result += dp[0][n - 1][k];
        }

        return (int) (result % MOD); // Final result with mod operation
    }
}

```

C++ Solution

```cpp
using ll = long long; // Define a type alias for long long integers.

class Solution {
public:
    // Method to count all unique palindromic subsequences in the string.
    int countPalindromicSubsequences(string s) {
        const int mod = 1e9 + 7; // Define the modulo value for large numbers.
        int n = s.size(); // Get the length of the input string.

        // Initialize 3D dynamic programming array where dp[i][j][k] will contain the count
        // of palindromic subsequences starting at i, ending at j,
        // and beginning with the character 'a' + k.
        vector<vector<vector<ll>>> dp(n, vector<vector<ll>>(n, vector<ll>(4, 0)));

        // Base cases: single-character palindromes.
        for (int i = 0; i < n; ++i) {
            dp[i][i][s[i] - 'a'] = 1;
        }

        // Start populating the DP array for subsequences of length l.
        for (int l = 2; l <= n; ++l) {
            for (int i = 0; i + l <= n; ++i) {
                int j = i + l - 1; // Calculate the end index of the current subsequence.
                for (char c = 'a'; c <= 'd'; ++c) { // Iterate over each character from 'a' to 'd'.
                    int k = c - 'a'; // Convert char to corresponding index.
                    if (s[i] == c && s[j] == c) {
                        // If both ends match, all internal subsequences count twice (once including each end).
                        // Also, add 2 for the subsequences formed by the matching ends themselves.
                        dp[i][j][k] = 2 + accumulate(dp[i + 1][j - 1].begin(), dp[i + 1][j - 1].end(), 0ll) % mod;
                    } else if (s[i] == c) {
                        // If only the starting character matches c, inherit count from subsequences ending before j.
                        dp[i][j][k] = dp[i][j - 1][k];
                    } else if (s[j] == c) {
                        // If only the ending character matches c, inherit count from subsequences starting after i.
                        dp[i][j][k] = dp[i + 1][j][k];
                    } else {
                        // If neither end matches c, inherit count from internal subsequence.
                        dp[i][j][k] = dp[i + 1][j - 1][k];
                    }
                }
            }
        }

        // Collect the final answer from dp[0][n - 1], which contains all valid subsequences for the entire string.
        ll ans = 0;
        for (int k = 0; k < 4; k++) {
            ans += dp[0][n - 1][k];
            ans %= mod; // Ensure we take modulo after each addition.
        }

        return static_cast<int>(ans); // Cast the long long result back to int before returning.
    }
};

```

Typescript Solution

```typescript
type ll = number; // Define a type alias for long long integers (use 'number' in TypeScript).

// Define the modulo value for large numbers.
const MOD = 1e9 + 7;

// Method to count all unique palindromic subsequences in the string.
function countPalindromicSubsequences(s: string): number {
    const n: number = s.length; // Get the length of the input string.

    // Initialize 3D dynamic programming array where dp[i][j][k] will contain the count
    // of palindromic subsequences starting at i, ending at j,
    // and beginning with the character 'a' + k.
    const dp: ll[][][] = Array.from({ length: n }, () =>
        Array.from({ length: n }, () => Array(4).fill(0))
    );

    // Base cases: single-character palindromes.
    for (let i = 0; i < n; ++i) {
        dp[i][i][s.charCodeAt(i) - 'a'.charCodeAt(0)] = 1;
    }

    // Start populating the DP array for subsequences of length l.
    for (let l = 2; l <= n; ++l) {
        for (let i = 0; i + l <= n; ++i) {
            let j = i + l - 1; // Calculate the end index of the current subsequence.
            for (let c = 'a'.charCodeAt(0); c <= 'd'.charCodeAt(0); ++c) { // Iterate over each character from 'a' to 'd'.
                let k = c - 'a'.charCodeAt(0); // Convert char code to corresponding index.
                if (s[i] === String.fromCharCode(c) && s[j] === String.fromCharCode(c)) {
                    // If both ends match, all internal subsequences count twice (once including each end).
                    // Also, add 2 for the subsequences formed by the matching ends themselves.
                    dp[i][j][k] = (2 + dp[i + 1][j - 1].reduce((sum, val) => (sum + val) % MOD, 0)) % MOD;
                } else if (s[i] === String.fromCharCode(c)) {
                    // If only the starting character matches c, inherit count from subsequences ending before j.
                    dp[i][j][k] = dp[i][j - 1][k];
                } else if (s[j] === String.fromCharCode(c)) {
                    // If only the ending character matches c, inherit count from subsequences starting after i.
                    dp[i][j][k] = dp[i + 1][j][k];
                } else {
                    // If neither end matches c, inherit count from internal subsequence.
                    dp[i][j][k] = dp[i + 1][j - 1][k];
                }
            }
        }
    }

    // Collect the final answer from dp[0][n - 1], which contains all valid subsequences for the entire string.
    let ans: ll = 0;
    for (let k = 0; k < 4; k++) {
        ans = (ans + dp[0][n - 1][k]) % MOD; // Ensure we take modulo after each addition.
    }

    return ans; // Return the result.
}

// Example usage:
// console.log(countPalindromicSubsequences("abcd"));

```

Time and Space Complexity

Time Complexity

The time complexity of the provided code is primarily determined by the triple nested loops:

- The outermost loop runs for all subsequence lengths, from `2` to `n` (the string length), resulting in `O(n)` iterations.
- The middle loop runs for each starting index of the subsequence, which is also `O(n)`.
- The innermost loop runs for each character `c` in `'abcd'`, leading to `4` iterations as there are `4` characters.

The core operation inside the innermost loop takes constant time except for the `sum(dp[i + 1][j - 1])` operation, which is done in `O(4)` or constant time since it is a summation over an array with `4` elements.

Hence, the overall time complexity is `O(n^2 * 4)`, which simplifies to `O(n^2)`.

Space Complexity

The space complexity is determined by the size of the `dp` array, which is a three-dimensional array.

- The first dimension has `n` elements where `n` is the length of the string.
- The second dimension also has `n` elements representing the ending index of the subsequence.
- The third dimension has `4` elements corresponding to each character `c` in `'abcd'`.

Thus, the space complexity is `O(n^2 * 4)`. Since `4` is a constant and doesn't depend on `n`, it can be dropped in Big O notation, making the space complexity `O(n^2)`.