1638. Count Substrings That Differ by One Character

Medium | Hash Table | String | Dynamic Programming | Leetcode Link

Problem Description

The goal of this problem is to count the number of ways in which we can select a non-empty substring from a string `s` and replace exactly one character in it such that the modified substring matches some substring in another string `t`. This is equivalent to finding substrings in `s` that differ by precisely one character from any substring in `t`.

For example, take `s = "computer"` and `t = "computation"`. If we look at the substring `s[0:7] = "compute"` from `s`, we can change the last character `e` to `a`, obtaining `s[0:6] + 'a' = "computa"`. This new substring, `computa`, is also a substring of `t`, thus constituting one valid way.

The problem requires us to find the total number of such valid ways for all possible substrings in `s`.

Intuition

To solve this problem efficiently, we can apply dynamic programming (DP). The intuition for using DP hinges on two key observations:

1. For every pair of indices `(i, j)` where `s[i]` is equal to `t[j]`, the substrings ending at these indices can form a part of larger matching substrings, barring the one character swap.
2. If `s[i]` is not equal to `t[j]`, then we have found the place where a single character difference occurs. All substring pairs ending at `(i, j)` and having this as their only difference should be counted.

Given these observations, we can define two DP tables:

- `f[i][j]` that stores the length of the matching substring of `s[0..i-1]` and `t[0..j-1]` ending at `s[i-1]` and `t[j-1]`. Essentially, it tells us how far back we can extend the matching part to the left, before encountering a mismatch or the start of the strings.
- `g[i][j]` serves a similar purpose but looks to the right of `i` and `j`, telling us how far we can extend the matching part of the substrings starting at `s[i]` and `t[j]`.

The variable `ans` accumulates the number of valid ways. For each mismatch found (where `s[i] != t[j]`), we calculate how many valid substrings can be formed and add this to `ans`. The number of valid substrings is computed as `(f[i][j] + 1) * (g[i + 1][j + 1] + 1)`. This accounts for the one-character swap by combining the lengths of matching substrings directly to the left and right of `(i, j)`.

We iterate through each pair of indices `(i, j)` comparing characters from `s` and `t`, and whenever we find a mismatch, we perform the above calculation. Finally, we return the value of `ans` as our answer.

Solution Approach

The solution implements a dynamic programming approach to efficiently solve the problem by considering all possible substrings of `s` and `t` and determining if they differ by exactly one character.

Here's a step-by-step breakdown of the solution code:

1. **Initialize Variables:** The solution begins by initializing an accumulator `ans` to keep track of the count of valid substrings, and the lengths of the strings `s` and `t`, denoted as `m` and `n` respectively.
2. **Create DP Tables:** Two DP tables `f` and `g` are created with dimensions `(m+1) x (n+1)`, initializing all their entries to `0`. Each cell `f[i][j]` will hold the length of the matching substring pair ending with `s[i-1]` and `t[j-1]`. Similarly, `g[i][j]` will hold the length of the matching substring pair starting with `s[i]` and `t[j]`.
3. **Fill the `f` Table:** The nested loop over `i` and `j` fills the `f` table. For each pair `(i, j)`, if `s[i-1]` is equal to `t[j-1]`, then `f[i][j]` is set to the value of `f[i-1][j-1] + 1`, which extends the length of the matching substring by one. Otherwise, `f[i][j]` is already `0`, indicating no match.
4. **Fill the `g` Table and Calculate `ans`:** Another nested loop, counting downwards from `m-1` to `0` for `i` and from `n-1` to `0` for `j`, fills the `g` table. If `s[i]` is equal to `t[j]`, then `g[i][j]` is set to `g[i+1][j+1] + 1`. But when `s[i]` does not match `t[j]`, a mismatch has been found. The product `(f[i][j] + 1) * (g[i+1][j+1] + 1)` calculates the number of valid ways considering the mismatch as the single difference point, and this number is added to the accumulator `ans`.
5. **Return the Result:** After iterating through all possible substrings, the `ans` variable will have accumulated the total number of valid substrings where a single character replacement in `s` can result in a substring in `t`. This accumulated result is returned as the final answer.

In this solution, dynamic programming tables `f` and `g` efficiently store useful computed values which are used to keep track of matches and calculate the number of possible valid substrings dynamically for each pair of indices `(i,j)`. By avoiding redundant comparisons and reusing previously computed values, the algorithm avoids the naive approach's extensive computation, thus improving efficiency.

Example Walkthrough

Let's walk through a small example to illustrate the solution approach using strings `s = "acdb"` and `t = "abc"`.

1. **Initialize Variables:** We set `ans` to 0. The lengths of the strings `m` and `n` are 4 and 3 respectively.
2. **Create DP Tables:** We initialize the DP tables `f` and `g` as 5×4 matrices, as `s` has length 4, and `t` has length 3.
3. **Fill the `f` Table:** We iterate through strings `s` and `t` with indices `i` and `j`. When characters match, we set `f[i][j] = f[i-1][j-1] + 1`. Here's how `f` looks like after filling:

		a	b	c
	0	0	0	0
a	0	1	0	0
c	0	0	0	1
d	0	0	0	0
b	0	0	1	0

4. **Fill the `g` Table and Calculate `ans`:** Next, we fill in the `g` table by iterating backwards. On mismatches, we calculate `(f[i][j] + 1) * (g[i+1][j+1] + 1)` and add it to `ans`.

   Steps while filling `g`:

   - Start at `s[3] = 'b'` and `t[2] = 'c'`, no match, set `g[3][2] = 0`.
   - Move to `s[3] = 'b'` and `t[1] = 'b'`, they match, so set `g[3][1] = g[4][2] + 1` which is `1`.
   - Continue for other elements. Discrepancies are found at `s[2] = 'd'` and `t[0] = 'a'`, as well as `s[0] = 'a'` and `t[1] = 'b'`. We calculate the valid ways for these mismatches, adding the results to `ans`.

   Here's the `g` matrix:

		a	b	c
	1	0	0	
a	0	1	0	
c	0	0	1	
d	0	0	0	
b	0	1	0	

   In this case, `ans` will be calculated as follows:

   - For `s[2] != t[0]`, `(f[2][0] + 1) * (g[3][1] + 1) = (0 + 1) * (0 + 1) = 1`.
   - For `s[0] != t[1]`, `(f[0][1] + 1) * (g[1][2] + 1) = (0 + 1) * (0 + 1) = 1`.

   We add these to `ans`, getting us an `ans = 2`.

5. **Return the Result:** The variable `ans` now has the value `2`, which is the number of valid substrings in `s` that can match substrings in `t` by changing exactly one character. We return this value as the answer.

Python Solution

```python
class Solution:
    def count_substrings(self, s: str, t: str) -> int:
        # Initializes the count of valid substrings
        count = 0
        # Lengths of the input strings
        len_s, len_t = len(s), len(t)
        # Initialize the forward and backward match length tables
        forward_match = [[0] * (len_t + 1) for _ in range(len_s + 1)]
        backward_match = [[0] * (len_t + 1) for _ in range(len_s + 1)]

        # Populate the forward match length table
        for i, char_s in enumerate(s, 1):
            for j, char_t in enumerate(t, 1):
                if char_s == char_t:
                    forward_match[i][j] = forward_match[i - 1][j - 1] + 1

        # Populate the backward match length table, and calculate the count
        for i in range(len_s - 1, -1, -1):
            for j in range(len_t - 1, -1, -1):
                if s[i] == t[j]:
                    backward_match[i][j] = backward_match[i + 1][j + 1] + 1
                else:
                    # When characters do not match, we multiply the forward match
                    # and backward match lengths and add these matched substrings to the count.
                    count += (forward_match[i][j] + 1) * (backward_match[i + 1][j + 1] + 1)

        # Return the total count of valid substrings
        return count

```

Java Solution

```java
class Solution {
    public int countSubstrings(String s, String t) {
        int count = 0; // Initialize count to track the number of valid substrings
        int m = s.length(), n = t.length(); // Lengths of the input strings

        // f[i][j] will store the length of the common substring of s and t ending with s[i-1] and t[j-1]
        int[][] commonSuffixLength = new int[m + 1][n + 1];

        // g[i][j] will store the length of the common substring of s and t starting with s[i] and t[j]
        int[][] commonPrefixLength = new int[m + 1][n + 1];

        // Compute the length of the common suffixes for all pairs of characters from s and t
        for (int i = 0; i < m; i++) {
            for (int j = 0; j < n; j++) {
                if (s.charAt(i) == t.charAt(j)) {
                    commonSuffixLength[i + 1][j + 1] = commonSuffixLength[i][j] + 1;
                }
            }
        }

        // Compute the length of the common prefixes and the number of valid substrings
        for (int i = m - 1; i >= 0; i--) {
            for (int j = n - 1; j >= 0; j--) {
                if (s.charAt(i) == t.charAt(j)) {
                    // If characters match, extend the common prefix by 1
                    commonPrefixLength[i][j] = commonPrefixLength[i + 1][j + 1] + 1;
                } else {
                    // When there is a mismatch, count the valid substrings using the common prefix and suffix
                    count += (commonSuffixLength[i][j] + 1) * (commonPrefixLength[i + 1][j + 1] + 1);
                }
            }
        }

        return count; // Return the total count of valid substrings
    }
}
```

C++ Solution

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

class Solution {
public:
    // Function to count the number of good substrings.
    int countSubstrings(string s, string t) {
        int answer = 0; // Initialize answer to zero.
        int m = s.length(), n = t.length(); // Get the lengths of strings s and t.

        // Dynamic programming arrays to keep track of matching substrings.
        int matchingSuffix[m + 1][n + 1];
        int matchingPrefix[m + 1][n + 1];

        // Initialize the DP tables with zeros.
        memset(matchingSuffix, 0, sizeof(matchingSuffix));
        memset(matchingPrefix, 0, sizeof(matchingPrefix));

        // Build the DP table for suffix matching substrings.
        for (int i = 0; i < m; ++i) {
            for (int j = 0; j < n; ++j) {
                if (s[i] == t[j]) {
                    // If characters match, extend the suffix by 1.
                    matchingSuffix[i + 1][j + 1] = matchingSuffix[i][j] + 1;
                }
            }
        }

        // Build the DP table for prefix matching substrings.
        for (int i = m - 1; i >= 0; --i) {
            for (int j = n - 1; j >= 0; --j) {
                if (s[i] == t[j]) {
                    // If characters match, extend the prefix by 1.
                    matchingPrefix[i][j] = matchingPrefix[i + 1][j + 1] + 1;
                } else {
                    // If characters don't match, calculate the count of
                    // good substrings ending here.
                    answer += (matchingSuffix[i][j] + 1) * (matchingPrefix[i + 1][j + 1] + 1);
                }
            }
        }

        // Return the total number of good substrings.
        return answer;
    }
};
```

Typescript Solution

```typescript
// Function to count the number of good substrings.
function countSubstrings(s: string, t: string): number {
    let answer = 0; // Initialize answer to zero.
    const m = s.length; // Get the length of string s.
    const n = t.length; // Get the length of string t.

    // Initialize arrays to keep track of matching substrings.
    const matchingSuffix: number[][] = Array.from({ length: m + 1 }, () => Array(n + 1).fill(0));
    const matchingPrefix: number[][] = Array.from({ length: m + 1 }, () => Array(n + 1).fill(0));

    // Build the table for suffix matching substrings.
    for (let i = 0; i < m; i++) {
        for (let j = 0; j < n; j++) {
            if (s[i] === t[j]) {
                // If characters match, extend the suffix by 1.
                matchingSuffix[i + 1][j + 1] = matchingSuffix[i][j] + 1;
            }
        }
    }

    // Build the table for prefix matching substrings.
    for (let i = m - 1; i >= 0; i--) {
        for (let j = n - 1; j >= 0; j--) {
            if (s[i] === t[j]) {
                // If characters match, extend the prefix by 1.
                matchingPrefix[i][j] = matchingPrefix[i + 1][j + 1] + 1;
            } else {
                // If characters don't match, calculate the count of
                // good substrings ending at this position.
                answer += (matchingSuffix[i][j] + 1) * (matchingPrefix[i + 1][j + 1] + 1);
            }
        }
    }

    // Return the total number of good substrings.
    return answer;
}

```

Time and Space Complexity

Time Complexity

The given code consists of a nested loop structure, where two independent loops iterate over the length of `s` and `t`. The outer loop runs for `m * n` times and the inner nested loops run `m * n` times separately for the loops that build the `f` and `g` 2D arrays. The computation within the inner loops operates in `O(1)` time. Therefore, the total time complexity combines the `O(m * n)` for the outer loops and `O(m * n)` for the inner nested loops, resulting in `O(m * n)` overall.

Space Complexity

The space complexity is determined by the size of the two-dimensional arrays `f` and `g`, each of which has a size of `(m + 1) * (n + 1)`. Hence, the space used by these data structures is `O(m * n)`. No other data structures are used that grow with the input size, so the total space complexity is `O(m * n)`.