1794. Count Pairs of Equal Substrings With Minimum Difference

Medium Greedy Hash Table String

Problem Description

In this problem, we are given two strings, `firstString` and `secondString`, and our task is to count the number of unique quadruples of indices `(i,j,a,b)` that meet certain conditions. Each quadruple represents a matching substring between `firstString` and `secondString` with the indices fulfilling the following:

1. `0 <= i <= j < firstString.length`, meaning that `i` and `j` are valid indices within the `firstString`, and `i` can be equal to `j` to allow for single-character substrings.
2. `0 <= a <= b < secondString.length`, indicating that `a` and `b` are valid indices within the `secondString`, with the same provision for `a` to equal `b`.
3. The substring from `i` to `j` (inclusive) in `firstString` is exactly the same as the substring from `a` to `b` in `secondString`.
4. `j - a` should be the smallest possible difference for all such valid quadruples. This implies that we're interested in matching substrings that are closest to the start of `firstString`.

The goal is to return the count of such quadruples.

Intuition

To find a solution, we need to identify all possible matching substrings between `firstString` and `secondString`, but with an added twist of minimizing the `j - a` difference. This suggests that a brute force method of checking every possible substring between the two strings would be inefficient, particularly with large strings. Instead, we should find a way to both confirm when substrings match and also ensure we're doing so in a way that minimizes `j - a`.

A key observation is that for any matching substring, the last character of the substring in `secondString` must be the closest it can possibly be to the start of `secondString` compared to its index in `firstString`. So, for every character in `firstString`, we need to find the last occurrence of that character in `secondString` and track the minimum `j - a` value.

The intuition behind the solution is to keep track of the closest last occurrence of each character from `secondString` as we iterate through `firstString`. We maintain a dictionary, `last`, that maps each character to its last occurrence in `secondString`. As we go through each character `c` in `firstString`, we calculate the difference between the current index in `firstString` and the index of the last occurrence of `c` in `secondString`. If the current difference is less than the minimum difference found so far (`mi`), we update `mi` and reset the count of valid quadruples to 1, as this is the new minimum difference. If we encounter the same minimum difference again, we increment our count.

By tracking the minimum difference as we iterate through `firstString` and updating when necessary, we are able to tally the number of quadruples efficiently and ensure that the condition `j - a` is minimized.

Solution Approach

The solution uses a dictionary to store the mapping of characters to their last occurrence in `secondString`. This is crucial because the conditions specify that we want to match substrings and minimize `j - a`, which inherently requires us to know the last (closest to the end) index of a character.

The steps of the implementation are as follows:

1. We first initialize the dictionary `last` to map each character `c` in `secondString` to the index `i` of its last occurrence, as given by `{c: i for i, c in enumerate(secondString)}`. This operation happens once and is an O(n) operation where n is the length of `secondString`.
2. We set `ans` to 0, which will keep the count of the number of valid quadruples, and `mi` to `inf` (infinity), which is a placeholder for the minimum value of `j - a` we have seen so far.
3. We then iterate through each character `c` and its index `i` in `firstString`, checking if `c` is present in `last`. If it is not, then there's no matching substring ending with that character, and we move on.
4. When we do find a matching character, we calculate the difference `t = i - last[c]`, which represents `j - a`.
5. We then compare `t` with `mi`. If `t` is smaller, it means we have found a closer match in terms of the indices, and we update `mi` to `t` and reset `ans` to 1 because we've found a new minimum. If `t` is equal to `mi`, it means we've found another quadruple with the same minimum difference, so we increment `ans` by 1.
6. Finally, we return the value of `ans`, which is the total count of such quadruples after iterating through the entire `firstString`.

This approach is efficient as it only requires O(n + m) time complexity, where n is the length of `firstString` and m is the length of `secondString`, because we go through each string only once. It avoids the much less efficient O(n*m*min(n,m)) time complexity that would be required if we were to naively look for matching substrings with a nested loop and substring comparisons.

A key algorithmic principle at play here is the use of a "greedy" strategy that aims to always choose the closest last occurrence of a character when considering matching substrings, ensuring the smallest possible `j - a` as required by the problem.

Example Walkthrough

Let's illustrate the solution approach with a small example:

Suppose we have `firstString = "abac"` and `secondString = "cabba"`. We want to find the number of unique quadruples where the substrings match and have the smallest `j - a` difference.

1. We start by creating the `last` dictionary mapping each character in `secondString` to its last occurrence: `{'c': 0, 'a': 4, 'b': 3}`.
2. We set `ans` to 0 and `mi` (minimum difference) to infinity.
3. Iterate through each character in `firstString`:
    - For `i = 0` with `c = 'a'`, `last['a']` is 4, so `t = 0 - 4 = -4`. Since `t` is less than `mi`, we set `mi` to -4 and `ans` to 1.
    - For `i = 1` with `c = 'b'`, there is no 'b' in `last`, so we move on.
    - For `i = 2` with `c = 'a'`, `last['a']` is still 4, so `t = 2 - 4 = -2`. `t` is greater than `mi`, so we ignore this.
    - For `i = 3` with `c = 'c'`, `last['c']` is 0, so `t = 3 - 0 = 3`. `t` is greater than `mi`, so again, we ignore this.
4. After iteration, we find that `ans` is 1, which indicates there is only one quadruple satisfying the conditions: `(0, 0, 4, 4)`, representing the substring 'a' from both strings.

Thus, the function would return 1 as the count of such quadruples.

By following this approach, we efficiently calculated the result without having to compare each possible substring pair, saving computation time and adhering to the problem's constraints.

Solution Implementation

Python

```python
class Solution:
    def countQuadruples(self, first_string: str, second_string: str) -> int:
        # Create a dictionary to map each character in the second string to the index of its last occurrence.
        last_occurrence_index = {char: index for index, char in enumerate(second_string)}

        # Initialize answer to 0, and minimum difference as infinity (a large number).
        answer = 0
        minimum_difference = float('inf')  # 'inf' represents infinity in Python.

        # Loop over each character and its index in the first string.
        for index, char in enumerate(first_string):
            # Check if the current character exists in the second string.
            if char in last_occurrence_index:
                # Calculate the difference between the index of the character in
                # the first string and the last occurrence index in the second string.
                difference = index - last_occurrence_index[char]

                # If the current difference is less than the minimum difference recorded:
                if minimum_difference > difference:
                    # Update minimum difference and reset answer to 1.
                    minimum_difference = difference
                    answer = 1
                # If the current difference matches the minimum difference:
                elif minimum_difference == difference:
                    # Increment the answer by 1.
                    answer += 1

        # Return the final count of quadruples.
        return answer
```

Java

```java
class Solution {
    public int countQuadruples(String firstString, String secondString) {
        // Initialize an array to store the last occurrence index of each character
        int[] lastOccurrence = new int[26];
        // Traverse through the second string to fill the last occurrence array
        for (int i = 0; i < secondString.length(); ++i) {
            lastOccurrence[secondString.charAt(i) - 'a'] = i + 1;
        }

        int count = 0; // Counter for the number of quadruples
        int minDifference = Integer.MAX_VALUE; // Variable to track the minimum difference

        // Traverse through the first string to find valid quadruples
        for (int i = 0; i < firstString.length(); ++i) {
            // Get the last occurrence of the current character from the first string in the second string
            int j = lastOccurrence[firstString.charAt(i) - 'a'];
            // Ensure that the character also occurs in the second string
            if (j > 0) {
                int difference = i - j; // Calculate the difference
                // If the difference is less than the current minimum, update the minimum and reset the count
                if (minDifference > difference) {
                    minDifference = difference;
                    count = 1;
                } else if (minDifference == difference) {
                    // If the difference is the same as the current minimum, increment the count
                    ++count;
                }
            }
        }
        // Return the count of quadruples
        return count;
    }
}
```

C++

```cpp
#include <string>
#include <vector>
#include <climits>

class Solution {
public:
    // Count the number of quadruples where the last occurrence of a character in the first string
    // comes before the last occurrence of the same character in the second string
    int countQuadruples(string firstString, string secondString) {
        // Initialize an array to store the last occurrence indices for each character in the second string
        // Note that the array is initialized with zeros (0) which signifies that the character hasn't been found yet.
        int lastOccurrence[26] = {};

        // Find the last occurrence of each character in the second string
        for (int i = 0; i < secondString.size(); ++i) {
            // Using character 'a' as base index 0 for 'a' to 'z' as 0 to 25
            lastOccurrence[secondString[i] - 'a'] = i + 1; // Store index + 1 to differentiate between found and not found.
        }

        int count = 0; // Will hold the final count of quadruples
        int minDifference = INT_MAX; // Start with the maximum value as the difference to find the minimum

        // Iterate through the first string to find the quadruples
        for (int i = 0; i < firstString.size(); ++i) {
            int currentIndex = lastOccurrence[firstString[i] - 'a']; // Get the last occurrence index from the second string

            // If the character is present in the second string
            if (currentIndex) {
                int difference = i - (currentIndex - 1); // Calculate the difference i - j

                // If a new minimum difference is found, update 'minDifference' and reset count to 1
                if (minDifference > difference) {
                    minDifference = difference;
                    count = 1;
                }
                // If the same minimum difference is found again, increment count
                else if (minDifference == difference) {
                    ++count;
                }
            }
        }

        // Return the total count of quadruples found
        return count;
    }
};
```

TypeScript

```typescript
// The function takes two strings and counts the number of quadruples where the last occurrence of a
// character in the first string comes before the last occurrence of the same character in the second string.
function countQuadruples(firstString: string, secondString: string): number {
    // An array to keep track of the last occurrence indices for each letter in the second string.
    // A value of 0 indicates the character has not been found yet.
    let lastOccurrence: number[] = new Array(26).fill(0);

    // Find the last occurrence index of each character in the second string.
    for (let i = 0; i < secondString.length; i++) {
        // Convert the character to an index from 0 to 25 (for 'a' to 'z').
        let charIndex = secondString.charCodeAt(i) - 'a'.charCodeAt(0);
        // Store the last occurrence index of the current character + 1 (to differentiate between found and not found).
        lastOccurrence[charIndex] = i + 1;
    }

    let count = 0; // Used to keep track of the number of quadruples found.
    let minDifference = Number.MAX_SAFE_INTEGER; // Initialize with the largest safe integer value as a starting point.

    // Iterate through each character in the first string to find quadruples.
    for (let i = 0; i < firstString.length; i++) {
        // Get the index of the last occurrence of the current character from the second string.
        let charIndex = firstString.charCodeAt(i) - 'a'.charCodeAt(0);
        let currentIndex = lastOccurrence[charIndex];

        // Check if the current character is present in the second string.
        if (currentIndex) {
            // Calculate the difference between indices (i - j).
            let difference = i - (currentIndex - 1);

            // Update 'minDifference' and reset 'count' if a new minimum difference is found.
            if (minDifference > difference) {
                minDifference = difference;
                count = 1;
            }
            // If the same minimum difference is found, increment the 'count'.
            else if (minDifference === difference) {
                count++;
            }
        }
    }

    // Return the final count of quadruples found.
    return count;
}
```

```python
class Solution:
    def countQuadruples(self, first_string: str, second_string: str) -> int:
        # Create a dictionary to map each character in the second string to the index of its last occurrence.
        last_occurrence_index = {char: index for index, char in enumerate(second_string)}

        # Initialize answer to 0, and minimum difference as infinity (a large number).
        answer = 0
        minimum_difference = float('inf')  # 'inf' represents infinity in Python.

        # Loop over each character and its index in the first string.
        for index, char in enumerate(first_string):
            # Check if the current character exists in the second string.
            if char in last_occurrence_index:
                # Calculate the difference between the index of the character in
                # the first string and the last occurrence index in the second string.
                difference = index - last_occurrence_index[char]

                # If the current difference is less than the minimum difference recorded:
                if minimum_difference > difference:
                    # Update minimum difference and reset answer to 1.
                    minimum_difference = difference
                    answer = 1
                # If the current difference matches the minimum difference:
                elif minimum_difference == difference:
                    # Increment the answer by 1.
                    answer += 1

        # Return the final count of quadruples.
        return answer
```

Time and Space Complexity

The provided Python code defined within the `Solution` class is designed to find the number of times a minimal difference between indices of matching characters from `firstString` and `secondString` occurs. The time complexity and space complexity of this code are as follows:

Time Complexity

The time complexity of this code can be split into two major components:

1. Building the `last` dictionary, where the last occurrence of each character from `secondString` is stored with the character as the key and its index as the value. In the worst case, `enumerate(secondString)` will iterate through all characters of `secondString`, which takes `O(n)`, where `n` refers to the length of `secondString`.
2. Iterating over `firstString` to calculate the differences and counting the minimal differences. This is another loop that goes through all the characters in `firstString`, which, in the worst case, results in `O(m)`, where `m` refers to the length of `firstString`.

Since these operations are sequential, the overall time complexity is the sum of the two individual complexities: `O(n) + O(m)`. Since these two strings are independent, simplifying the expression does not combine the terms, and the final time complexity is `O(m + n)`.

Space Complexity

The space complexity is determined by the additional space required by the algorithm which is not part of the input or the output. In this case, it's:

1. The space used by the `last` dictionary which, in the worst case, contains an entry for every unique character in `secondString`, and since there is a fixed limit to the character set (in the case of ASCII, a maximum of 128 characters), the space taken by `last` could be considered `O(1)`.
2. The two variables `ans` and `mi` occupy constant space `O(1)`.

Therefore, the space complexity is the larger of the space used by the `last` dictionary and the space for the variables `ans` and `mi`, which results in `O(1)` space complexity.

Overall, the provided code has a **time complexity** of `O(m + n)` and a **space complexity** of `O(1)`.