567. Permutation in String

Medium | Hash Table | Two Pointers | String | Sliding Window

Problem Description

Given two strings `s1` and `s2`, your task is to determine whether `s2` contains a permutation of `s1`. In other words, you must check if any substring of `s2` has the same characters as `s1`, in any order. The function should return `true` if at least one permutation of `s1` is a substring of `s2`, otherwise, it should return `false`.

Intuition

The intuition behind the solution is to use a sliding window approach along with character counting. We want to slide a window of size equal to the length of `s1` over `s2` and check if the characters inside this window form a permutation of `s1`. The key idea is to avoid recomputing the frequency of characters in the window from scratch each time we slide the window; instead, we can update the count based on the character that is entering and the character that is leaving the window.

To implement this, we use a counter data structure to keep track of the difference between the number of occurrences of each character in the current window and the number of occurrences of each character in `s1`. Initially, the counter is set by decrementing for `s1` characters and incrementing for the first window in `s2`. We can then iterate through `s2`, moving the window to the right by incrementing the count for the new character and decrementing for the character that's no longer in the window.

The difference count is the sum of the non-zero values in the counter. If at any point the difference count is zero, it means the current window is a permutation of `s1`, and we return `true`. If we reach the end of `s2` without finding such a window, we return `false`.

Solution Approach

The problem is solved efficiently by using the sliding window technique, coupled with a character counter that keeps track of the frequencies of characters within the window.

Here are the key steps of the algorithm:

1. Initialize a `Counter` object that will track the frequency difference of characters between `s1` and the current window of `s2`.
2. Set up the initial count by decrementing for each character in `s1` and incrementing for each character in the first window of `s2`.
3. Calculate the initial difference count, which is the sum of non-zero counts in the `Counter`. This represents how many characters' frequencies do not match between `s1` and the current window.
4. Start traversing `s2` with a window size of `s1`. For each step, do the following:
   - If the difference count is zero, return `true`.
   - Update the `Counter` by incrementing the count for the new character entering the window, and decrementing the count for the character leaving the window.
   - Adjust the difference count if the updated character counts change from zero to non-zero, or vice versa.
5. If the loop completes without the difference count reaching zero, return `false`.

The implementation takes `O(n + m)` time where `n` is the length of `s1` and `m` is the length of `s2`. The space complexity is `O(1)` since the counter size is limited to the number of possible characters, which is constant.

The key data structures and patterns used in this solution are:

- `Counter` from the Python `collections` module to keep track of frequencies of characters.
- Sliding window technique to efficiently inspect substrings of `s2` without re-counting characters each time.
- Two-pointers pattern to represent the current window's start and end within `s2`.

This approach effectively checks every possible window in `s2` that could be a permutation of `s1`, doing so in a manner that only requires a constant amount of work for each move of the window.

Example Walkthrough

Let's consider an example to illustrate the solution approach:

Suppose we have `s1 = "abc"` and `s2 = "eidbacoo"`. We are tasked with determining if `s2` contains a permutation of `s1`.

1. First, we initialize the `Counter` object for `s1` which would look like `Counter({'a':1, 'b':1, 'c':1})` as each character in `s1` occurs once.
2. Next, we look at the first window of `s2` with the same length as `s1`, which is `eid`. We initialize another `Counter` for this window, resulting in `Counter({'e':1, 'i':1, 'd':1})`.
3. Now, we compute the initial difference count by comparing our two `Counter` objects. For characters `e`, `i`, and `d` the count increments as they appear in `s2` but not `s1`. For characters `a`, `b`, and `c`, the counts decrement for their presence in `s1` but absence in the initial window of `s2`. The sum of non-zero counts is `6`, as we have three characters in `s1` that are not in the window and three characters in the window that are not in `s1`.
4. We start sliding the window in `s2` to the right, one character at a time. The next window is `idb`. We increment the count for `b` (as it enters the window) and decrement the count for `e` (as it exits). Now the `Counter` updates, and we recalculate the difference count. Characters `i` and `d` still contribute to the difference count, but `b` does not anymore because it matches with `s1`.
5. Continue sliding the window to the right to the window `dba`, updating the `Counter` by incrementing for `a` and decrementing for `i`. The counter is now matched for `a` and `b`, but not for `d`.
6. Proceed to the window `bac`. Increment for `c` and decrement for `d`. Now the `Counter` should match `s1` completely, which means the difference count will be `0`.
7. As the difference count is `0`, it indicates that the `bac` window is a permutation of `s1`. Therefore, we return `true`.

By using the sliding window and the `Counter`, we moved through `s2` efficiently, avoiding recalculating the frequency of characters from scratch. We found that `s2` contains a permutation of `s1`, demonstrating the solution approach effectively.

Python Solution

```python
from collections import Counter

class Solution:
    def check_inclusion(self, pattern: str, text: str) -> bool:
        # Calculate the length of both the pattern and text
        pattern_length, text_length = len(pattern), len(text)

        # If the pattern is longer than the text, the inclusion is not possible
        if pattern_length > text_length:
            return False

        # Initialize a counter for the characters in both strings
        char_counter = Counter()

        # Decrease the count for pattern characters and increase for the first window in text
        for pattern_char, text_char in zip(pattern, text[:pattern_length]):
            char_counter[pattern_char] -= 1
            char_counter[text_char] += 1

        # Calculate the number of characters that are different
        diff_count = sum(x != 0 for x in char_counter.values())

        # If no characters are different, we found an inclusion
        if diff_count == 0:
            return True

        # Slide the window over text, one character at a time
        for i in range(pattern_length, text_length):
            # Get the character that will be removed from the window and the one that will be added
            char_out = text[i - pattern_length]
            char_in = text[i]

            # Update diff_count if the incoming character impacts the balance
            if char_counter[char_in] == 0:
                diff_count += 1
            char_counter[char_in] += 1
            if char_counter[char_in] == 0:
                diff_count -= 1

            # Update diff_count if the outgoing character impacts the balance
            if char_counter[char_out] == 0:
                diff_count += 1
            char_counter[char_out] -= 1
            if char_counter[char_out] == 0:
                diff_count -= 1

            # If no characters are different, we have found an inclusion
            if diff_count == 0:
                return True

        # If inclusion has not been found by the end of the text, return False
        return False

```

Java Solution

```java
class Solution {
    public boolean checkInclusion(String s1, String s2) {
        int length1 = s1.length();
        int length2 = s2.length();

        // If the first string is longer than the second string,
        // it's not possible for s1 to be a permutation of s2.
        if (length1 > length2) {
            return false;
        }

        // Array to hold the difference in character counts between s1 and s2.
        int[] charCountDelta = new int[26];

        // Populate the array with initial counts
        for (int i = 0; i < length1; ++i) {
            charCountDelta[s1.charAt(i) - 'a']--;
            charCountDelta[s2.charAt(i) - 'a']++;
        }

        // Counts the number of characters with non-zero delta counts.
        int nonZeroCount = 0;
        for (int count : charCountDelta) {
            if (count != 0) {
                nonZeroCount++;
            }
        }

        // If all deltas are zero, s1 is a permutation of the first part of s2.
        if (nonZeroCount == 0) {
            return true;
        }

        // Slide the window of length1 through s2
        for (int i = length1; i < length2; ++i) {
            int charLeft = s2.charAt(i - length1) - 'a'; // Character going out of the window
            int charRight = s2.charAt(i) - 'a'; // Character coming into the window

            // Update counts for the exiting character
            if (charCountDelta[charRight] == 0) {
                nonZeroCount++;
            }
            charCountDelta[charRight]++;
            if (charCountDelta[charRight] == 0) {
                nonZeroCount--;
            }

            // Update counts for the entering character
            if (charCountDelta[charLeft] == 0) {
                nonZeroCount++;
            }
            charCountDelta[charLeft]--;
            if (charCountDelta[charLeft] == 0) {
                nonZeroCount--;
            }

            // If all deltas are zero, s1's permutation is found in s2.
            if (nonZeroCount == 0) {
                return true;
            }
        }

        // If we reach here, no permutation of s1 is found in s2.
        return false;
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    // This function checks if s1's permutation is a substring of s2
    bool checkInclusion(string s1, string s2) {
        int len1 = s1.size(), len2 = s2.size();

        // If length of s1 is greater than s2, permutation is not possible
        if (len1 > len2) {
            return false;
        }

        // Vector to store character counts
        vector<int> charCount(26, 0);

        // Initialize the character count vector with the first len1 characters
        for (int i = 0; i < len1; ++i) {
            --charCount[s1[i] - 'a']; // Decrement for characters in s1
            ++charCount[s2[i] - 'a']; // Increment for characters in the first window of s2
        }

        // Calculate the difference count
        int diffCount = 0;
        for (int count : charCount) {
            if (count != 0) {
                ++diffCount;
            }
        }

        // If diffCount is zero, a permutation exists in the first window
        if (diffCount == 0) {
            return true;
        }

        // Slide the window over s2 and update the counts and diffCount
        for (int i = len1; i < len2; ++i) {
            int index1 = s2[i - len1] - 'a'; // Index for the old character in the window
            int index2 = s2[i] - 'a'; // Index for the new character in the window

            // Before updating charCount for the new character
            if (charCount[index2] == 0) {
                ++diffCount;
            }
            ++charCount[index2]; // Include the new character in the window

            // After updating charCount for the new character
            if (charCount[index2] == 0) {
                --diffCount;
            }

            // Before updating charCount for the old character
            if (charCount[index1] == 0) {
                ++diffCount;
            }
            --charCount[index1]; // Remove the old character as we move the window

            // After updating charCount for the old character
            if (charCount[index1] == 0) {
                --diffCount;
            }

            // If the diffCount is zero after the updates, a permutation is found
            if (diffCount == 0) {
                return true;
            }
        }

        // No permutation was found
        return false;
    }
};

```

Typescript Solution

```typescript
function checkInclusion(s1: string, s2: string): boolean {
    // If s1 is longer than s2, it's impossible for s1 to be a permutation of s2.
    if (s1.length > s2.length) {
        return false;
    }

    // Helper function to convert characters into zero-based indices
    function charToIndex(char: string): number {
        return char.charCodeAt(0) - 'a'.charCodeAt(0);
    }

    // Helper function to check if both character frequency arrays match
    function doArraysMatch(freqArray1: number[], freqArray2: number[]): boolean {
        for (let i = 0; i < 26; i++) {
            if (freqArray1[i] !== freqArray2[i]) {
                return false;
            }
        }
        return true;
    }

    const s1Length = s1.length;
    const s2Length = s2.length;
    // Arrays to store the frequency of each letter in s1 and the current window in s2
    const freqArray1 = new Array(26).fill(0);
    const freqArray2 = new Array(26).fill(0);

    // Filling freqArray1 with frequencies of letters in s1
    for (const char of s1) {
        const index = charToIndex(char);
        freqArray1[index]++;
    }

    // Filling freqArray2 with frequencies of the first window of s2 with size equal to s1 length
    for (let i = 0; i < s1Length; i++) {
        const index = charToIndex(s2[i]);
        freqArray2[index]++;
    }

    // Sliding window to check each substring in s2
    for (let left = 0, right = s1Length; right < s2Length; left++, right++) {
        // Check if the current window is a permutation of s1
        if (doArraysMatch(freqArray1, freqArray2)) {
            return true;
        }

        // Slide the window forward: remove the left character and add the right character
        const leftIndex = charToIndex(s2[left]);
        const rightIndex = charToIndex(s2[right]);
        freqArray2[leftIndex]--;
        freqArray2[rightIndex]++;
    }

    // Check the last window after the loop
    return doArraysMatch(freqArray1, freqArray2);
}

```

Time and Space Complexity

Time Complexity

The time complexity of the provided code is `O(n + m)`, where `n` is the length of `s1` and `m` is the length of `s2`. Here's why:

1. `zip(s1, s2)` takes `O(n)` time to iterate through the elements of the shorter string, which is `s1` in this case as we return `False` immediately if `s1` is longer than `s2`.
2. The `sum(x != 0 for x in cnt.values())` takes `O(1)` time since there can be at most 26 characters (assuming lowercase English letters), so the number of different characters is constant.
3. The main loop runs from `n` to `m`, which executes `m - n + 1` times (inclusive of `n`). Each iteration of the loop has a constant number of operations that do not depend on the size of `n` or `m`. Therefore, this part also takes `O(m)` time.
4. Combining these parts, we get a total time complexity of `O(n + m)`.

Space Complexity

The space complexity of the code is `O(1)` because the `cnt` counter will contain at most 26 key-value pairs (if we are considering the English alphabet). The number of keys in `cnt` does not grow with the size of the input strings `s1` and `s2`, thus it is a constant space overhead.