Permutation in String LeetCode

You are given two strings, **s1** and **s2**. You need to determine if **s2** contains a permutation of **s1**.

A permutation of a string is any rearrangement of its characters. For example, "abc" and "acb" are permutations of each other.

Examples:

1. Input: **s1 = "ab" s2 = "eiddfbaiiiooo"** Output: Permutation of s1 exists in s2. Explanation: The string "ba" is a permutation of "ab" and it exists in s2.
2. Input: **s1 = "abc" s2 = "eidcbbad"** Output: Permutation of s1 does not exist in s2. Explanation: There is no permutation of "abc" that exists in s2.

Approach 1: Function to check if permutation of s1 exists in s2 using sliding window approach

- The function **checkInclusion** takes two strings **s1** and **s2** as input and uses a sliding window approach to check if a permutation of **s1** exists in **s2**.
- It initializes two count arrays, **count1** and **count2**, to store the frequencies of characters in **s1** and the current window of **s2**, respectively.
- The function calculates the frequencies of characters in **s1** and initializes the count array **count1** accordingly.
- It then initializes a window of size **windowSize** (length of **s1**) in **s2** and stores the frequencies of characters in the count array **count2**.
- The function checks if **count1** and **count2** are equal using the **checkEqual** helper function. If they are equal, it means a permutation of **s1** exists in **s2**, and the function returns **true**.
- The function then slides the window to the right and updates the count array **count2** by incrementing the count of the new character and decrementing the count of the character that goes out of the window.
- This sliding window process continues until the end of **s2**.
- If at any point **count1** and **count2** are equal, the function returns **true**. Otherwise, it returns **false** if no permutation of **s1** is found in **s2**.

Time Complexity: O(N)

- The function iterates through **s2** once using the sliding window approach.
- The while loop runs for **s2Len** iterations, where **s2Len** is the length of **s2**.

- Inside the loop, the **checkEqual** function iterates over the count arrays, which has a constant size of 26.
- Thus, the overall time complexity is O(N), where N is the length of **s2**.

Space Complexity: O(1)

- The space complexity is constant because the count arrays have a fixed size of 26, which is independent of the input size.
- Additionally, the other variables used in the function (such as **i**, **windowSize**, **left**, and **right**) have constant space complexity.

Approach 2: Function to check if permutation of s1 exists in s2 using two-pointer approach

- The function **checkInclusionTwoPointers** also takes two strings **s1** and **s2** as input and uses a two-pointer approach to check if a permutation of **s1** exists in **s2**.
- It initializes a count array **count** to store the frequencies of characters in **s1**.
- The function then uses two pointers, **left** and **right**, to maintain a window in **s2**.
- The right pointer is moved to the right while decrementing the count of the current character in **count**.
- If the size of the window (**right - left + 1**) exceeds or is equal to the length of **s1**, the left pointer is moved to the right while incrementing the count of the character that goes out of the window.
- At each step, the function checks if the counts in **count** are equal to zero for all characters. If they are, it means a permutation of **s1** exists in the current window of **s2**, and the function returns **true**.
- If no permutation of **s1** is found in **s2**, the function returns **false**.

Time Complexity: O(N)

- The function also iterates through **s2** once using the two-pointer approach.
- The while loop runs for **s2Len** iterations, where **s2Len** is the length of **s2**.
- Inside the loop, the function performs constant-time operations, such as incrementing and decrementing counts and comparing them.
- Thus, the overall time complexity is O(N), where N is the length of **s2**.

Space Complexity: O(1)

- The space complexity is constant because the count array has a fixed size of 26, which is independent of the input size.

- Additionally, the other variables used in the function (such as **left**, **right**, and **s1Len**) have constant space complexity.