Permutation In String

solved by	Senan
Platform	LeetCode
difficulty	Medium
# Serial	567
tags	Permutation Sliding Window Vector
language	C++
solved on	@05/10/2024
link	https://leetcode.com/problems/permutation-in-string/description/
Completion	☑

Intuition

The problem is to check if one string (`s1`) is a permutation of another string (`s2`). Specifically, we need to check if a permutation of `s1` is a substring of `s2`.

The idea is based on the sliding window approach. The length of the window will be equal to the length of `s1`, and as we slide this window over `s2`, we can compare the frequency counts of characters within the window to the frequency count of characters in `s1`. If at any point the frequency counts match, then a permutation of `s1` is present in `s2`.

Approach

1. **Base case:** If the length of `s1` is greater than `s2`, we can immediately return false since `s1` cannot be a substring of `s2`.
2. **Frequency arrays:**
   - Create two frequency arrays (`count1` and `count2`) to store the counts of characters (from 'a' to 'z', so of size 26) in `s1` and the current window in `s2`.
   - Initialize the first window in `s2` by populating the frequency count for the first `n1` characters (where `n1 = s1.size()`).
3. **Sliding Window:**
   - Start sliding the window from the beginning of `s2`. For each position `i`, if the frequency arrays `count1` and `count2` match, return `true` since we found a valid permutation.
   - Update the frequency array `count2` by removing the effect of the character that's sliding out of the window and adding the effect of the new character coming into the window.
4. **Final check:** After completing the sliding process, perform a final comparison between `count1` and `count2` to ensure the last window is checked.
5. **Return false** if no valid permutation is found.

Complexity

Time Complexity:

- Constructing the frequency arrays initially takes **O(n1)**, where `n1` is the size of `s1`.

- We slide the window across `s2` of size `n2`. For each window, comparing the frequency arrays takes constant time (**O(1)**) because they have a fixed size of 26.
- Sliding the window through `s2` takes **O(n2)** time.

Overall time complexity: **O(n1 + n2)**.

Space Complexity:

- We use two arrays (`count1` and `count2`) of size 26 to store character frequencies.

Space complexity: **O(1)** (constant space).

Code

```
class Solution {
public:
    bool checkInclusion(string s1, string s2) {
        int n1 = s1.size();
        int n2 = s2.size();

        if(n1 > n2) return false;
                 vector<int> count1(26, 0), count2(26, 0);

        for(int i = 0; i < n1; i++) {
            count1[s1[i] - 'a']++;
            count2[s2[i] - 'a']++;
        }

        for(int i = 0; i < n2 - n1; i++) {
            if(count1 == count2) return true;
            count2[s2[i] - 'a']--;
            count2[s2[i + n1] - 'a']++;
        }

        return count1 == count2;
    }
};
```