Permutation in String

Given two strings **s1** and **s2**, write a function to return true if **s2** contains the permutation of **s1**. In other words, one of the first string's permutations is the **substring** of the second string.

Example 1:

```
Input:s1 = "ab" s2 = "eidbaooo"
Output:True
Explanation: s2 contains one permutation of s1 ("ba").
```

Example 2:

```
Input:s1= "ab" s2 = "eidboaoo"
Output: False
```

Note:

1. The input strings only contain lower case letters.
2. The length of both given strings is in range [1, 10,000].

Solution 1

1. How do we know string `p` is a permutation of string `s`? Easy, each character in `p` is in `s` too. So we can abstract all permutation strings of `s` to a map (Character -> Count). i.e. `abba` -> `{a:2, b:2}`. Since there are only 26 lower case letters in this problem, we can just use an array to represent the map.
2. How do we know string `s2` contains a permutation of `s1`? We just need to create a sliding window with length of `s1`, move from beginning to the end of `s2`. When a character moves in from right of the window, we subtract `1` to that character count from the map. When a character moves out from left of the window, we add `1` to that character count. So once we see all zeros in the map, meaning equal numbers of every characters between `s1` and the substring in the sliding window, we know the answer is true.

```
public class Solution {
    public boolean checkInclusion(String s1, String s2) {
        int len1 = s1.length(), len2 = s2.length();
        if (len1 > len2) return false;

        int[] count = new int[26];
        for (int i = 0; i < len1; i++) {
            count[s1.charAt(i) - 'a']++;
            count[s2.charAt(i) - 'a']--;
        }
        if (allZero(count)) return true;

        for (int i = len1; i < len2; i++) {
            count[s2.charAt(i) - 'a']--;
            count[s2.charAt(i - len1) - 'a']++;
            if (allZero(count)) return true;
        }

        return false;
    }

    private boolean allZero(int[] count) {
        for (int i = 0; i < 26; i++) {
            if (count[i] != 0) return false;
        }
        return true;
    }
}
```

written by shawngao original link here

Solution 2

```java
    public boolean checkInclusion(String s1, String s2) {
        int[] count = new int[128];
        for(int i = 0; i < s1.length(); i++) count[s1.charAt(i)]--;
        for(int l = 0, r = 0; r < s2.length(); r++) {
            if (++count[s2.charAt(r)] > 0)
                while(--count[s2.charAt(l++)] != 0) { /* do nothing */}
            else if ((r - l + 1) == s1.length()) return true;
        }
        return s1.length() == 0;
    }
```

Update:

I gonna use pictures to describe what the above code does. The first "*for*" loop counts all chars we need to find in a way like digging holes on the ground:

Blank bars are the holes that we need to fill.

We scan each one char of the string *s2* (by moving index **r** in above code) and put it in the right hole:

The blue blocks are chars from s2.

But if the char in s2 is not in s1, or, the count of the char is more than the count of the same char in s1, we got some thing like this:

Note the last blue block sticks out of ground. Any time we encounter a sticking out block - meaning a block with value 1 - we stop scanning (that is moving "***r***"). At this point, there is only one sticking out block.

Now, we have an invalid substring with either invalid char or invalid number of chars. How to remove the invalid char and continue our scan? We use a left index ("***l***" in above code) to remove chars in the holes in the same order we filled them into the holes. We stop removing chars until the only sticking out block is fixed - it has a value of 0 after fixing. Then, we continue our scanning by moving right index "***r***".

Our target is to get:

To check if all holes are filled perfectly - no more, no less, all have value of 0 - we just need to make sure (r - l + 1) == s1.length().

Update 2:
Thanks to mylemoncake comment. I have updated the last line to : return s1.length() == 0; This takes care of the case s1 is an empty string.

written by tyuan73 original link here

Solution 3

For each `window` representing a substring of `s2` of length `len(s1)`, we want to check if the count of the window is equal to the count of `s1`. Here, the count of a string is the list of: [the number of `a`'s it has, the number of `b`'s,... , the number of `z`'s.]

We can maintain the window by deleting the value of s2[i - len(s1)] when it gets larger than `len(s1)`. After, we only need to check if it is equal to the target. Working with list values of [0, 1,..., 25] instead of 'a'-'z' makes it easier to count later.

```
def checkInclusion(self, s1, s2):
    A = [ord(x) - ord('a') for x in s1]
    B = [ord(x) - ord('a') for x in s2]

    target = [0] * 26
    for x in A:
        target[x] += 1

    window = [0] * 26
    for i, x in enumerate(B):
        window[x] += 1
        if i >= len(A):
            window[B[i - len(A)]] -= 1
        if window == target:
            return True
    return False
```

written by awice original link here