Previous Next

567. Short Permutation in a Long String

April 29, 2017 | 131.9K views

Average Rating: 4.50 (50 votes)

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
```

Constraints:

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

Solution

Approach #1 Brute Force [Time Limit Exceeded]

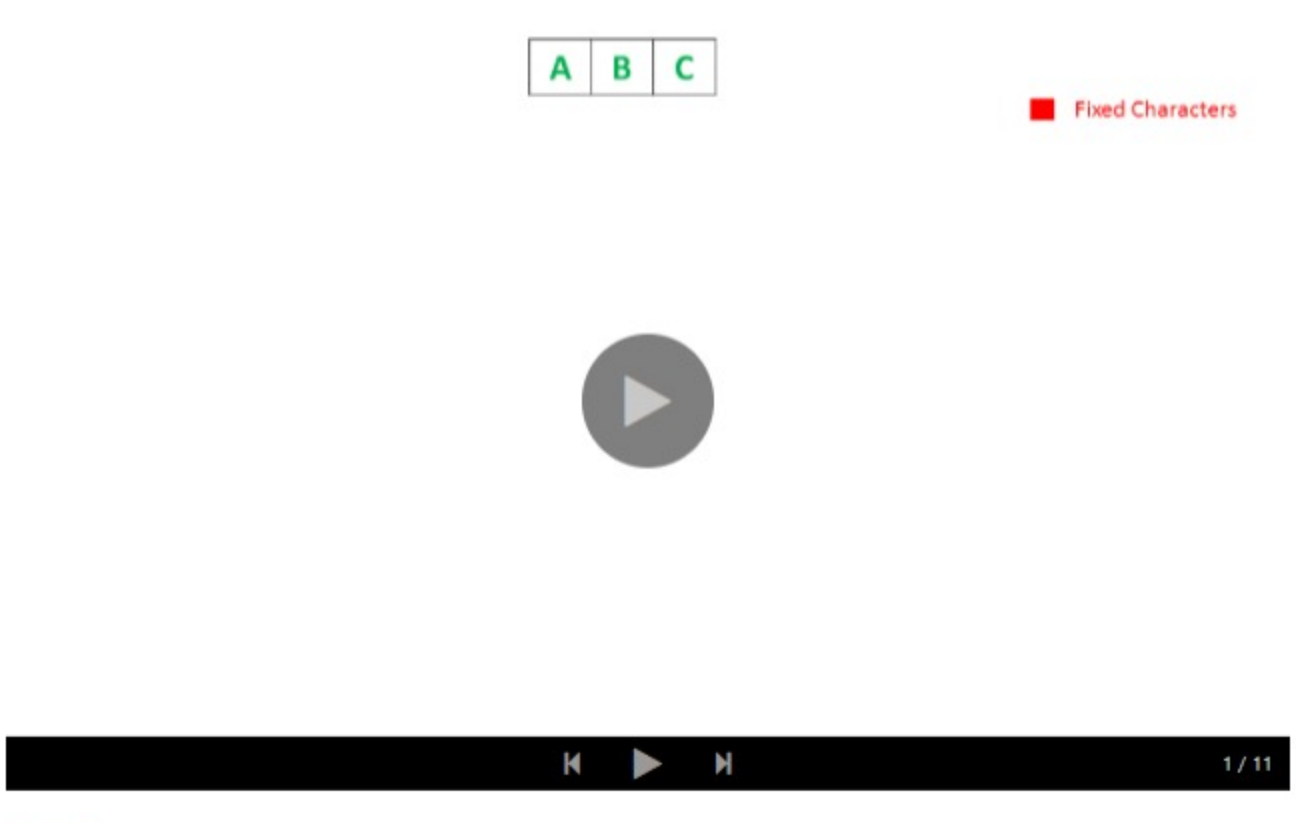
Algorithm

The simplest method is to generate all the permutations of the short string and to check if the generated permutation is a substring of the longer string.

In order to generate all the possible pairings, we make use of a function `permute(string_1, string_2, current_index)`. This function creates all the possible permutations of the short string $s1$.

To do so, permute takes the index of the current element $current_index$ as one of the arguments. Then, it swaps the current element with every other element in the array, lying towards its right, so as to generate a new ordering of the array elements. After the swapping has been done, it makes another call to permute but this time with the index of the next element in the array. While returning back, we reverse the swapping done in the current function call.

Thus, when we reach the end of the array, a new ordering of the array's elements is generated. The following animation depicts the process of generating the permutations.



Java

```java

public class Solution {
    boolean flag = false;
    public boolean checkInclusion(String s1, String s2) {
        permute(s1, s2, 0);
        return flag;
    }
    public String swap(String s, int i0, int i1) {
        if (i0 == i1)
            return s;
        String s1 = s.substring(0, i0);
        String s2 = s.substring(i0 + 1, i1);
        String s3 = s.substring(i1 + 1);
        return s1 + s.charAt(i1) + s2 + s.charAt(i0) + s3;
    }
    void permute(String s1, String s2, int l) {
        if (l == s1.length()) {
            if (s2.indexOf(s1) >= 0)
                flag = true;
        } else {
            for (int i = l; i < s1.length(); i++) {
                s1 = swap(s1, l, i);
                permute(s1, s2, l + 1);
                s1 = swap(s1, l, i);
            }
        }
    }
}
```

Complexity Analysis

- Time complexity : $O(n!)$. We match all the permutations of the short string $s1$, of length $s1$, with $s2$. Here, n refers to the length of $s1$.
- Space complexity : $O(n^2)$. The depth of the recursion tree is n(n refers to the length of the short string $s1$). Every node of the recursion tree contains a string of max. length n.

Approach #2 Using sorting [Time Limit Exceeded]:

Algorithm

The idea behind this approach is that one string will be a permutation of another string only if both of them contain the same characters the same number of times. One string x is a permutation of other string y only if $sorted(x) = sorted(y)$.

In order to check this, we can sort the two strings and compare them. We sort the short string $s1$ and all the substrings of $s2$, sort them and compare them with the sorted $s1$ string. If the two match completely, $s1$'s permutation is a substring of $s2$, otherwise not.

Java

```java
public class Solution {

    public boolean checkInclusion(String s1, String s2) {
        s1 = sort(s1);
        for (int i = 0; i <= s2.length() - s1.length(); i++) {
            if (s1.equals(sort(s2.substring(i, i + s1.length()))))
                return true;
        }
        return false;
    }
    public String sort(String s) {
        char[] t = s.toCharArray();
        Arrays.sort(t);
        return new String(t);
    }
}
```

Complexity Analysis

- Time complexity : $O\left(l_1 log(l_1) + (l_2 - l_1) l_1 log(l_1)\right)$. where l_1 is the length of string l_1 and l_2 is the length of string l_2.
- Space complexity : $O(l_1)$. t array is used .

Approach #3 Using Hashmap [Time Limit Exceeded]

Algorithm

As discussed above, one string will be a permutation of another string only if both of them contain the same charaters with the same frequency. We can consider every possible substring in the long string $s2$ of the same length as that of $s1$ and check the frequency of occurence of the characters appearing in the two. If the frequencies of every letter match exactly, then only $s1$'s permutation can be a substring of $s2$.

In order to implement this approach, instead of sorting and then comparing the elements for equality, we make use of a hashmap $s1map$ which stores the frequency of occurence of all the characters in the short string $s1$. We consider every possible substring of $s2$ of the same length as that of $s1$, find its corresponding hashmap as well, namely $s2map$. Thus, the substrings considered can be viewed as a window of length as that of $s1$ iterating over $s2$. If the two hashmaps obtained are identical for any such window, we can conclude that $s1$'s permutation is a substring of $s2$, otherwise not.

Java

```java

public class Solution {
    public boolean checkInclusion(String s1, String s2) {
        if (s1.length() > s2.length())
            return false;
        HashMap < Character, Integer > s1map = new HashMap < > ();
        for (int i = 0; i < s1.length(); i++)
            s1map.put(s1.charAt(i), s1map.getOrDefault(s1.charAt(i), 0) + 1);
        for (int i = 0; i <= s2.length() - s1.length(); i++) {
            HashMap < Character, Integer > s2map = new HashMap < > ();
            for (int j = 0; j < s1.length(); j++) {
                s2map.put(s2.charAt(i + j), s2map.getOrDefault(s2.charAt(i + j), 0) + 1);
            }
            if (matches(s1map, s2map))
                return true;
        }
        return false;
    }
    public boolean matches(HashMap < Character, Integer > s1map, HashMap < Character, Integer > s2map) {
        for (char key: s1map.keySet()) {
            if (s1map.get(key) - s2map.getOrDefault(key, -1) != 0)
                return false;
        }
        return true;
    }
}

```

Complexity Analysis

- Time complexity : $O(l_1 + 26*l_1*(l_2 - l_1))$. hashmap contains atmost 26 keys. where l_1 is the length of string l_1 and l_2 is the length of string l_2.
- Space complexity : $O(1)$. hashmap contains atmost 26 key-value pairs.

Approach #4 Using Array [Accepted]

Algorithm

Instead of making use of a special HashMap datastructure just to store the frequency of occurence of characters, we can use a simpler array data structure to store the frequencies. Given strings contains only lowercase alphabets ('a' to 'z'). So we need to take an array of size 26.The rest of the process remains the same as the last approach.

Java

```java
public class Solution {
    public boolean checkInclusion(String s1, String s2) {
        if (s1.length() > s2.length())
            return false;
        int[] s1map = new int[26];
        for (int i = 0; i < s1.length(); i++)
            s1map[s1.charAt(i) - 'a']++;
        for (int i = 0; i <= s2.length() - s1.length(); i++) {
            int[] s2map = new int[26];
            for (int j = 0; j < s1.length(); j++) {
                s2map[s2.charAt(i + j) - 'a']++;
            }
            if (matches(s1map, s2map))
                return true;
        }
        return false;
    }
    public boolean matches(int[] s1map, int[] s2map) {
        for (int i = 0; i < 26; i++) {
            if (s1map[i] != s2map[i])
                return false;
        }
        return true;
    }
}

```

Complexity Analysis

- Time complexity : $O(l_1 + 26*l_1*(l_2 - l_1))$, where l_1 is the length of string l_1 and l_2 is the length of string l_2.
- Space complexity : $O(1)$. $s1map$ and $s2map$ of size 26 is used.

Approach #5 Sliding Window [Accepted]:

Algorithm

Instead of generating the hashmap afresh for every window considered in $s2$, we can create the hashmap just once for the first window in $s2$. Then, later on when we slide the window, we know that we remove one preceding character and add a new succeeding character to the new window considered. Thus, we can update the hashmap by just updating the indices associated with those two characters only. Again, for every updated hashmap, we compare all the elements of the hashmap for equality to get the required result.

Java

```java
public class Solution {
    public boolean checkInclusion(String s1, String s2) {
        if (s1.length() > s2.length())
            return false;
        int[] s1map = new int[26];
        int[] s2map = new int[26];
        for (int i = 0; i < s1.length(); i++) {
            s1map[s1.charAt(i) - 'a']++;
            s2map[s2.charAt(i) - 'a']++;
        }
        for (int i = 0; i < s2.length() - s1.length(); i++) {
            if (matches(s1map, s2map))
                return true;
            s2map[s2.charAt(i + s1.length()) - 'a']++;
            s2map[s2.charAt(i) - 'a']--;
        }
        return matches(s1map, s2map);
    }
    public boolean matches(int[] s1map, int[] s2map) {
        for (int i = 0; i < 26; i++) {
            if (s1map[i] != s2map[i])
                return false;
        }
        return true;
    }
}
```

Complexity Analysis

- Time complexity : $O(l_1 + 26*(l_2 - l_1))$, where l_1 is the length of string l_1 and l_2 is the length of string l_2.
- Space complexity : $O(1)$. Constant space is used.

Approach #6 Optimized Sliding Window [Accepted]:

Algorithm

The last approach can be optimized, if instead of comparing all the elements of the hashmaps for every updated $s2map$ corresponding to every window of $s2$ considered, we keep a track of the number of elements which were already matching in the earlier hashmap and update just the count of matching elements when we shift the window towards the right.

To do so, we maintain a $count$ variable, which stores the number of characters(out of the 26 alphabets), which have the same frequency of occurence in $s1$ and the current window in $s2$. When we slide the window, if the deduction of the last element and the addition of the new element leads to a new frequency match of any of the characters, we increment the $count$ by 1. If not, we keep the $count$ intact. But, if a character whose frequency was the same earlier(prior to addition and removal) is added, it now leads to a frequency mismatch which is taken into account by decrementing the same $count$ variable. If, after the shifting of the window, the $count$ evaluates to 26, it means all the characters match in frequency totally. So, we return a True in that case immediately.

Java

```java
public class Solution {
    public boolean checkInclusion(String s1, String s2) {
        if (s1.length() > s2.length())
            return false;
        int[] s1map = new int[26];
        int[] s2map = new int[26];
        for (int i = 0; i < s1.length(); i++) {
            s1map[s1.charAt(i) - 'a']++;
            s2map[s2.charAt(i) - 'a']++;
        }
        int count = 0;
        for (int i = 0; i < 26; i++)
            if (s1map[i] == s2map[i])
                count++;
        for (int i = 0; i < s2.length() - s1.length(); i++) {
            int r = s2.charAt(i + s1.length()) - 'a', l = s2.charAt(i) - 'a';
            if (count == 26)
                return true;
            s2map[r]++;
            if (s2map[r] == s1map[r])
                count++;
            else if (s2map[r] == s1map[r] + 1)
                count--;
            s2map[l]--;
            if (s2map[l] == s1map[l])
                count++;
            else if (s2map[l] == s1map[l] - 1)
                count--;
```

Complexity Analysis

- Time complexity : $O(l_1 + (l_2 - l_1))$. where l_1 is the length of string l_1 and l_2 is the length of string l_2.
- Space complexity : $O(1)$. Constant space is used.

Rate this article: ★★★★★

Previous Next

Comments: 27

Sort By

Type comment here... (Markdown is supported)

Preview Post

zdxiq125 ★ 162 May 18, 2020 12:50 PM

Exactly the same as 438. Find All Anagrams in a String, of which the article is more concise.

31 Share Reply

SHOW 1 REPLY

huyouhyw ★ 12 May 8, 2018 5:01 AM

shouldn't complexity of Approach #3 be: O(L1 + (26+L1)*(L2-L1))? because we are only comparing 26 keys of HashMap1 & HashMap2 for (L2-L1) times right? @vinod23

12 Share Reply

SHOW 2 REPLIES

LJerRRy ★ 42 May 2, 2017 7:08 AM

i impove the 4th solution

```
    public class Solution {
            public boolean checkInclusion(String s1, String s2) {
                int n = s1.length(), m = s2.length();
```

Read More

11 Share Reply

SHOW 4 REPLIES

navinojha1996 ★ 69 May 8, 2020 12:12 PM

can anyone tell me even if the time complexity of approach 3 and approach 4 is same why it is written For Aprroach 3 Time Limit Exceeded and Approach 4 Accepted

3 Share Reply

SHOW 2 REPLIES

lishichengyan ★ 233 January 14, 2020 6:02 AM

so sad I could only reach solution4...

3 Share Reply

nehaagrawal2210 ★ 3 May 2, 2017 11:02 PM

Accepted Solution, different logic, no need to compare arrays or maps, just keeping track of length and move window to next possible character for substring, when new character is scanned

public class Solution {
public boolean checkInclusion(String s1, String s2) {

Read More

3 Share Reply

lian9 ★ 11 June 19, 2019 9:15 AM

There is an error in Approach 4. The 2nd for loop should be "i < s2.length() - s1.length() + 1" instead of "i < s2.length() - s1.length() ". Or the answer will be wrong.

3 Share Reply

SHOW 1 REPLY

himanshukandwal ★ 131 June 30, 2017 8:48 AM

More simple Solution:

public boolean checkInclusion(String s1, String s2) {
int [] map = new int [256];
for (char ch : s1.toCharArray()) map [ch] ++;

Read More

3 Share Reply

SHOW 1 REPLY

YangCao ★ 34 August 27, 2019 12:12 PM

Concise solution:

```
    public boolean checkInclusion(String s1, String s2) {
        if (s1.length() > s2.length()) return false;
```

Read More

2 Share Reply

SHOW 1 REPLY

GeoLin ★ 0 January 6, 2019 8:11 PM

I want to ask a question, would the situation be a true answer? aab, caba, what if I input these strings, what is the answer?

0 Share Reply

SHOW 1 REPLY

‹ 1 2 3 ›