Number of substring containing all three characters

Problem Statement: Given a string s , consisting only of characters 'a' , 'b' , 'c'.Find the number of substrings that contain at least one occurrence of all these characters 'a' , 'b' , 'c'.

Examples

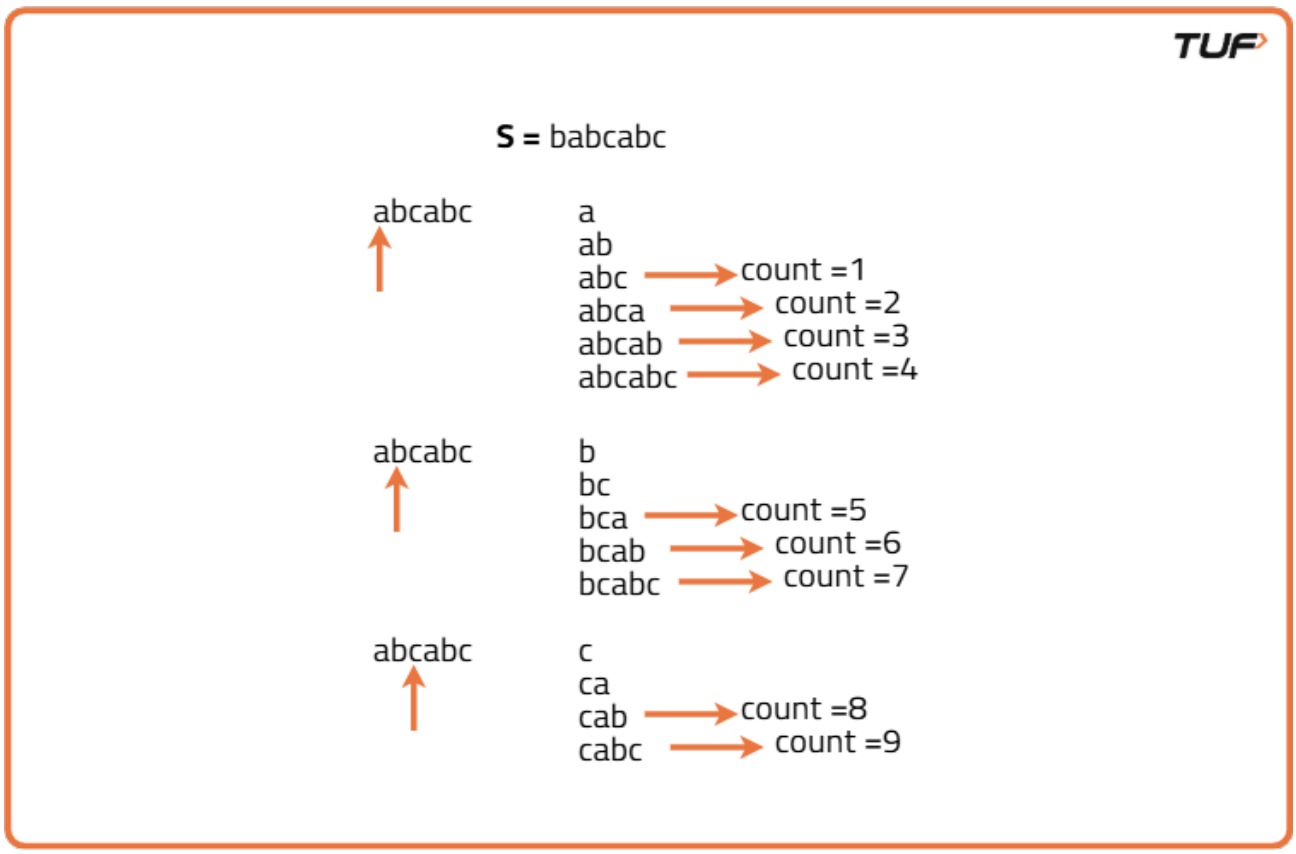
```
Input : s = "abcba"
Output :  5
Explanation : The substrings containing at least one occurrence of the
characters 'a' , 'b' , 'c' are "abc" , "abcb" , "abcba" , "bcba" , "cba".

Input : s = "ccabcc"
Output : 8
Explanation : The substrings containing at least one occurrence of the
characters 'a' , 'b' , 'c' are "ccab" , "ccabc" , "ccabcc" , "cab" , "cabc" ,
"cabcc" , "abc" , "abcc".
```

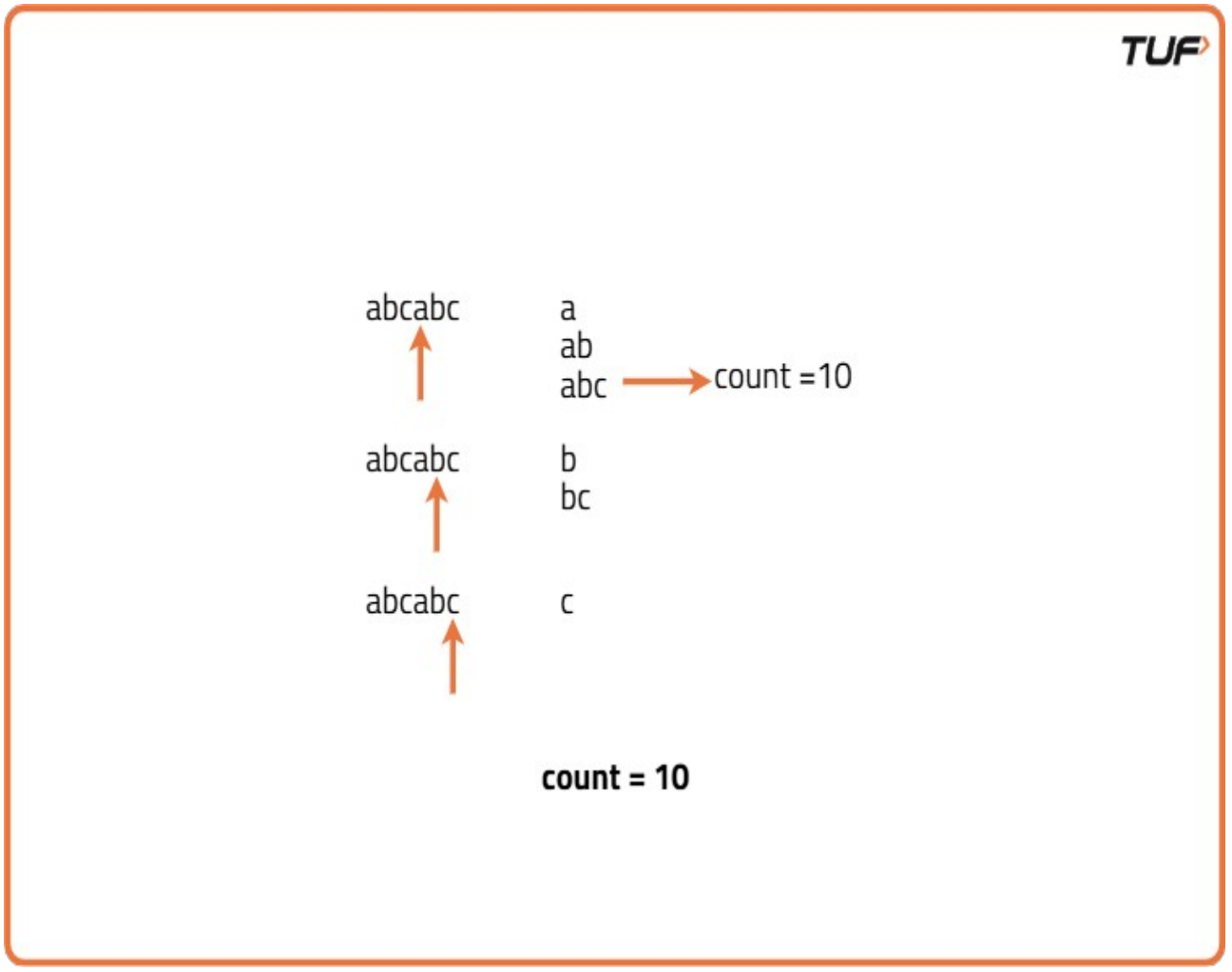
Brute force Approach

Algorithm

We iterate over all possible substrings of the given string and check whether each one contains all three characters 'a', 'b', and 'c'.We do this by using two nested loops:The outer loop picks the starting index of the substring and the inner loop expands the substring one character at a time and updates a set to keep track of distinct characters. If at any point the set contains all three characters, we increase our count.

- Initialize a counter to track the total valid substrings.
- For every starting index in the string:
    - Create an empty set to track the characters.
    - From that index, expand the substring to the right character by character.
    - At each step, insert the current character into the set.
    - If the set size becomes 3 (contains `'a'`, `'b'`, `'c'`), increment the count.
    - We can break early once all three characters are found, since adding more characters won't change the fact that it contains all three.
- Return the total count.


TUF
S = babcabc
abcabc
a
ab
abc
count =1
abca
count =2
abcab
count =3
abcabc
count =4
abcabc
b
bc
bca
count =5
bcab
count =6
bcabc
count =7
abcabc
c
ca
cab
count =8
cabc
count =9


TUF
abcabc
a
ab
abc
count =10
abcabc
b
bc
abcabc
c
count = 10

Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
    // Function to count substrings containing at least one 'a', one 'b', and
one 'c'
    int numberOfSubstrings(string s) {
        // Variable to store the final count
        int count = 0;
        // Length of the input string
        int n = s.length();

        // Outer loop to fix the start of the substring
        for (int i = 0; i < n; i++) {
            // Array to track the count of 'a', 'b', and 'c'
            vector<int> freq(3, 0);

            // Inner loop to fix the end of the substring
            for (int j = i; j < n; j++) {
                // Update frequency for current character
                freq[s[j] - 'a']++;

                // Check if all three characters are present
                if (freq[0] > 0 && freq[1] > 0 && freq[2] > 0) {
                    // Add valid substring
                    count++;
                }
            }
        }
        return count;
    }
};

// Driver code
int main() {
    Solution sol;
    string s = "abcabc";
    cout << sol.numberOfSubstrings(s) << endl;
    return 0;
}
```

Complexity Analysis

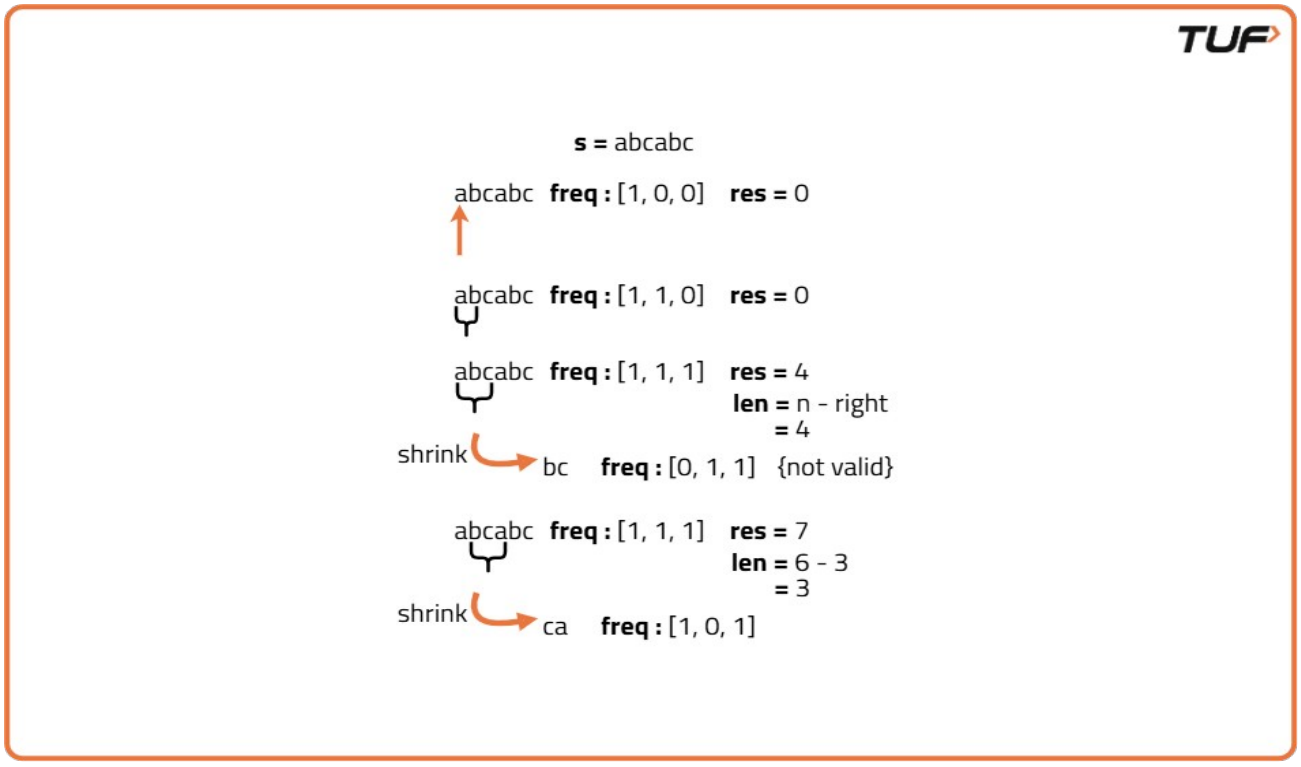
Time Complexity: O(n^2), where n is the length of the input string.We iterate through all possible starting indices from 0 to n−1, and for each starting index, we traverse the substring until we find a valid one (containing at least one 'a', 'b', and 'c'). In the worst case, the inner loop can run up to n times for each outer loop iteration, leading to a total of O(n^2) operations.

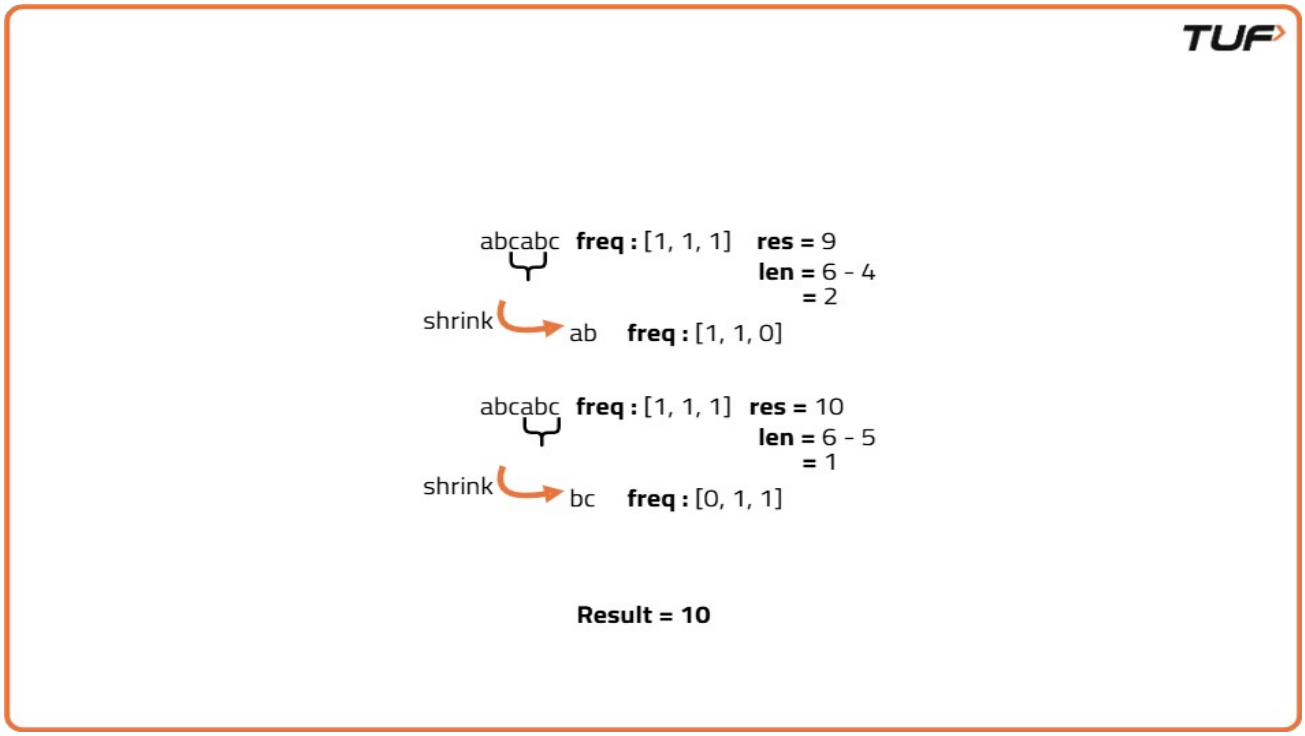
Space Complexity: O(1), constant space.We use a frequency map of fixed size (only for characters 'a', 'b', and 'c'). Regardless of input size, the space used remains constant. Hence, space complexity is O(1).

Optimal Approach
Algorithm
Instead of checking every possible substring, we use the sliding window technique. We maintain a window [left, right] such that it contains at least one of each character 'a', 'b', and 'c'. Once we get

such a window, any substring that starts at or before left and ends at right will also be valid. So we can add left + 1 to the result for each valid right. This avoids rechecking every substring and reduces unnecessary checks.

- Initialize a result counter to store the total number of valid substrings.
- Maintain a hash map or frequency array to track the count of each character 'a', 'b', and 'c' in the current window.
- Initialize a left pointer to 0. We'll use it to track the start of the sliding window.
- Iterate through the string using a right pointer from 0 to the end of the string
- For each character at the right pointer, increment its count in the frequency map.





Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
    // Function to count substrings containing at least one 'a', 'b', and 'c'
using sliding window
    int numberOfSubstrings(string s) {
        // Initialize frequency map for 'a', 'b', and 'c'
        vector<int> freq(3, 0);

        // Initialize result to store count of valid substrings
        int res = 0;

        // Initialize left pointer of the sliding window
        int left = 0;

        // Traverse the string using right pointer
        for (int right = 0; right < s.length(); right++) {
            // Increment frequency of current character
            freq[s[right] - 'a']++;

            // Shrink the window from the left while all three characters are
present
            while (freq[0] > 0 && freq[1] > 0 && freq[2] > 0) {
                // Count all substrings from current right to end
                res += (s.length() - right);

                // Decrease frequency of character at left and move left forward
                freq[s[left] - 'a']--;
                left++;
            }
        }

        return res;
    }
};

// Driver code
int main() {
    Solution sol;
    string s = "abcabc";
    cout << sol.numberOfSubstrings(s) << endl;
    return 0;
}
```

Complexity Analysis

Time Complexity:O(n) ,We traverse the string once with the right pointer and adjust the left pointer in a linear pass. Each character is processed at most twice (once by the right pointer and once by the left), resulting in linear time complexity.

Space Complexity: O(1),We only use a constant-size frequency array for three characters ('a', 'b', 'c'), hence the space usage does not grow with input size.